University of Alabama in Huntsville
LOUIS

Theses

UAH Electronic Theses and Dissertations

2007

Trajectory reconstruction with a multi-order least squares sliding window (LSSW) filter

Melissa S. Farbman

Follow this and additional works at: https://louis.uah.edu/uah-theses

Recommended Citation

Farbman, Melissa S., "Trajectory reconstruction with a multi-order least squares sliding window (LSSW) filter" (2007). *Theses*. 443.
https://louis.uah.edu/uah-theses/443

This Thesis is brought to you for free and open access by the UAH Electronic Theses and Dissertations at LOUIS. It has been accepted for inclusion in Theses by an authorized administrator of LOUIS.

TRAJECTORY RECONSTRUCTION WITH A MULTI-ORDER LEAST SQUARES SLIDING WINDOW (LSSW) FILTER

by

MELISSA S. FARBMAN

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in
The Department of Mechanical and Aerospace Engineering
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2007

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Melissa D. Farlman 11-15-07

(student signature) (date)

THESIS APPROVAL FORM

Submitted by Melissa S. Farbman in partial fulfillment of the requirements for the degree of Master of Science in Engineering with an option in Aerospace Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Engineering with an option in Aerospace Engineering.

7/13/07 Committee Chair
(Date)

7/13/07

7-13-07

7/31/07 Department Chair

7/31/07 College Dean

11/15/07 Graduate Dean

ABSTRACT
The School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science in Engineering College/Dept Engineering/Mechanical and Aerospace Engineering

Name of Candidate Melissa S. Farbman

Title Trajectory Reconstruction with a Multi-Order Least Squares Sliding Window (LSSW) Filter

Trajectory reconstruction is essential for verifying the accuracy of missile system simulations. This thesis presents an innovative method for performing trajectory reconstruction from flight test data, using a unique application of the least squares approach. The filter design allows variable order in addition to adjustable bandwidth. The filter was tested using a typical Tactical Ballistic Missile trajectory. Results show that the filter has excellent noise rejection properties. In addition, the higher-order states allow for improved extrapolation capabilities. The filter is very useful in processing noisy data with irregular data rates and drop-outs to produce a smooth output trajectory (including higher order states) at a uniform data rate.

Abstract Approval: Committee Chair 7/13/07

Department Chair 7/31/07

Graduate Dean 11/15/07

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Dr. Brian Landrum, for his tremendous help and guidance in my research work, and for his understanding and encouragement during my time at the University of Alabama at Huntsville. I also want to thank my committee members, Dr. Robert A. Frederick and Dr. Nathan J. Slegers, for their suggestions, contributions, and encouragement. In particular Dr. Slegers provided invaluable guidance, encouragement, and detailed suggestions that have significantly improved this thesis.

Thanks are in order to the US Army Research Development and Engineering Command.

Special thanks are due to Kevin Jackson, Joe Hill, Kyle Mixon, John Jordan, John Wells, and Wade Shirley for the creation of the original DGPS filter which this work was based upon. Without their help, this work would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

LIST OF SYMBOLS

BW	= bandwidth
σ	= sigma, the noise added to the trajectory
T	= time interval
τ	= time constant
ΔT	= input time interval
x_n	= position state of target
$\dot{x}_n$	= velocity state of target, the first derivative of the position
$\ddot{x}_n$	= acceleration state of target, the second derivative of the position
$\dddot{x}_n$	= jerk state of target, the third derivative of the position
$\ddddot{x}_n$	= snap state of target, the fourth derivative of the position

DEDICATION

This thesis is dedicated to my parents, Stephen Blevins and Marilyn Gartman, who have supported me all the way since the beginning of my studies. It is also for my husband, Joel, who has been a great source of motivation and inspiration, and Susan Farbman.

CHAPTER 1

INTRODUCTION

The air defense missile community has an immediate need for the precise reconstruction of test-flight target trajectories in order to evaluate current missile guidance algorithms. In the development of missile and air defense systems, high fidelity 6-DoF simulations are heavily relied upon to predict missile system performance. Because of the importance of the simulation tools, much effort is directed in making them as accurate and reliable as possible. In order to validate the accuracy of a high fidelity missile system simulation, test flights are orchestrated for the measurement and collection of vital missile system information, which is later compared to simulation data during post-flight reconstruction. An integral part of the post-flight reconstruction is the generation of a target trajectory that is relatively noise-free and matches the flight-test target trajectory very closely. The target trajectory is measured, normally via radar, and recorded for post-flight analysis. As a result, these radar measurements are noisy due to effects such as thermal noise, glint, and scintillation. Also, measurement drop-outs are common. The data used for the sensor truth models and lethality routines, however, must be relatively smooth and at a fixed time step. For the post-flight analysis, the target trajectory must be reconstructed and filtered from the recorded radar measurements before being played back through the missile simulation.

In addition to dealing with noise and data drop-outs, trajectory reconstruction often includes extrapolation beyond the measured data. In the course of a simulation run, the ground system tracks the target (the recording from the flight test) and initiates an engagement, while a simulated missile attempts to intercept the target. Because of variations in interceptor launch time, missile trajectory, etc., the simulated intercept may not occur at exactly the same altitude as the flight test. In successful flight tests, the target is destroyed at intercept. Consequently, the recorded target trajectory ends abruptly at that point. If the simulated intercept occurs at a slightly higher altitude, no data manipulation is required. However, if the simulated intercept occurs at a slightly lower altitude, then the recorded target trajectory will come to an end before the simulated engagement is completed. In this case, it is necessary to extrapolate the target trajectory which is experiencing significant aerodynamic effects.

This thesis focuses on an innovative method for performing trajectory reconstruction from flight test data that is noisy with irregular data rates and drop-outs. The Least Squares Sliding Window Filter (LSSW Filter) is designed to solve these problems and reconstruct the flight-test target trajectory in order to permit a meaningful post-flight analysis. By employing this method, the user will benefit from improved post-flight analyses and simulation validation activities which lead, ultimately, to a better missile system simulation.

CHAPTER 2

BACKGROUND

2.1 The Problem of Trajectory Reconstruction

This thesis addresses the problem of processing real target trajectories to support simulation post-flight analysis. The problem can be divided into two primary tasks:

- generation of smooth target trajectory data at regular intervals of time through filtering; and
- extrapolation of target trajectory states a few seconds beyond end of data.

Data recording during a flight test is frequently performed at a low data rate with the possibility of irregular update intervals and drop-outs. Also, the recorded data generally include radar measurement errors or noise from other sources. Any data filtering algorithm must deal with these issues to produce a trajectory with a uniform data rate and a suitably low noise level for use in missile simulations.

Since TBM targets typically experience significant deceleration in the altitude regime of interest (called the battle space for aerodynamically controlled air defense interceptors), a simple first-order extrapolation (constant velocity) is not sufficient. Also, since deceleration forces increase as the TBM continues to descend toward the ground, even a second-order extrapolation (constant acceleration) may not be sufficient.

However, a third-order filter could estimate the acceleration and higher order states, such as jerk and snap, and extrapolate to make reasonable predictions of target states several seconds beyond where the recorded target trajectory ends. The higher order states such as acceleration, jerk, and snap are defined as the second, third and fourth derivatives of position.

2.2 Filter Considerations and Requirements

The trajectory reconstruction solution is driven by the environment and known requirements.

The environment for the trajectory reconstruction filter includes

- asynchronous/low input data rate
- possibility of input drop-outs
- noisy input with possible target maneuver acceleration
- advance input data

Requirements for the trajectory reconstruction filter include

- filter outputs which include position, velocity, and acceleration states
- good noise rejection
- responsiveness to target maneuver
- optimal use of advance input data
- coast capability
- re-init capability
- high output rate (independent of input data rate)
- adjustable time constant
- auto-initialize capability

In addition to the above factors, the type of trajectory under evaluation must be considered. Air Breathing Threats (ABTs) have very different dynamics from Tactical Ballistic Missiles (TBMs). ABTs, in general, have either no acceleration or a constant lateral component of acceleration during turns. The following characteristics are representative of ABT trajectories from flight tests:

- constant velocity, constant altitude
- velocity ≈ 250 m/s
- measurement errors : $1\sigma = 10$ *meters*
- possible course changes, up to 3G's
- no intentional course changes near the end of trajectory

TBMs are usually characterized by high velocity, substantial longitudinal accelerations, and occasionally lateral maneuvers. The early phase of a TBM's flight trajectory tends to be relatively quiet and predictable, but as the flight progresses, it tends to exhibit large aerodynamic loads and rapid decelerations. This type of behavior is clearly nonlinear in nature and, thus, much harder to estimate. Some basic characteristics of TBM trajectories include

- rapid decelerations and descents
- velocity ≈ 1000 m/s to 2500 m/s
- measurement errors: $1\sigma = 10$ *meters*
- target accelerations likely near the end of the trajectory

The modeling methodology required for this application must to be flexible enough to allow the filter to be capable of handling a wide range of trajectories. In order to accomplish this objective, the filter should be kept as simple as possible.

2.3 Candidate Trajectory Reconstruction Techniques

There are several methods that are considered for trajectory reconstruction. Each method has its own advantages and disadvantages depending on the application for which it will be used. For many cases only a practical comparison can be used to choose between them. The most common methods such as such as Type 2 Low-Pass, Least Squares, Kalman, and Artificial Neural Networks (among others), have been reviewed in several papers and books [1-5] The Least Squares technique is primarily used for post processing applications where as the Kalman typically finds its main use in real-time applications [1]. Trajectory reconstruction for TBM re-entry vehicles is often performed via Kalman filters [2]. This approach works well in cases in which the input data rate is high enough to reject the noise while still maintaining responsiveness to target accelerations. However, due to the low input rates of the target trajectory, other approaches must be considered [3]. Artificial Neural Networks, Type 1 and 2 Low-Pass, Kalman, and Least Squares filters were evaluated as potential candidates in this thesis.

Artificial Neural Networks [4] were considered in the filter design process. The concept behind neural networks is based on a training process by which a set of coefficients are determined. A problem arises from the fact that different motion scenarios require different training procedures. For example, the motion of an ABT is quite different from that of a TBM, as discussed above. So, if there is any abrupt change in this motion, it could cause the system to fail. Neural networks tend to be situation specific, meaning that these networks have to be individually trained for each target scenario. This was the primary reason for not considering this concept further. Neural networks could be considered in future work if the availability of specific information was made available for a large number of targets to allow training [4 and 5].

A third-order Type 1 filter and a fifth-order Type 2 filter were developed for trajectory reconstruction. These are classical low-pass filters. A Type 1 filter produces zero steady state error, given a step input. A Type 2 filter produces zero steady state error, given a step or ramp input. Both of these filters work well for filtering noise from the input trajectory when set to a low bandwidth. Both are also effective at following accelerating targets when set to a high bandwidth. However, given the low sample rates, there is no existing bandwidth at which these filters can reject noise and follow the accelerations of a TBM.

A Kalman filter was tested. Like the low-pass filters, the Kalman filter could not satisfy the noise rejection and responsiveness requirements given the low sample rate [6]. The recursive Kalman filter tested is not able to optimize use of advance input data.

A Least Squares approach was also considered as a candidate. One of the main advantages of this method is that it requires less modeling and computational effort as compared to some of the other methods. This method was inspired by an earlier application called the DGPS (Differential Global Positioning System) target state filter [7]. The DGPS target state filter was originally designed for use in the U.S. Army's Virtual Engagement Simulation Testbed (VEST). VEST used the filter to smooth DGPS target position data in order to generate target truth states. VEST performed real-time simulations but had advance data.

The real breakthrough was to use data both before and after the current filter output time in the solution. This led to the creation of the Least Squares Sliding Window Filter (LSSW Filter). This is possible only because the entire measured trajectory is available during the reconstruction. However, it is problematic to integrate the full

benefit of the advance data into the low-pass or Kalman designs. In contrast, the Least Squares approach easily lends itself to incorporation of the advance data. Initially, to use a Least Squares filter to follow an accelerating target, the sliding window feature was added. This approach allows the Least Squares filter to fit the trajectory, beginning at the time just before the current filter output time and ending just afterwards. This interval of time or "window" slides forward in time as the current filter output time advances. Also, the duration of time or "window" utilized can be adjusted to effectively change the bandwidth of the filter. In general, larger windows give a greater filtering effect and are preferable for systems that require higher order functional estimations.

2.4 Summary of Candidate Methods for Filter Generation

Table 2.1 summarizes the performance of each candidate filtering method with respect to the given requirements. Most of the requirements are reasonably met by the Least Squares approach. These involve the ability to handle asynchronous input, coast, re-initialize, high output rates, and extrapolation. Also, as a result of the sliding window feature, the Least Squares approach is unlikely to experience divergence problems resulting from target accelerations. For these reasons, it was determined that the Least Squares approach was best suited for the filter.

Table 2.1 Candidate Solution Requirements Evaluation

	Asynchronous Input	**Coast Capability**	**Re-Init Capability**	**Noise Rejection**	**Responsiveness to Maneuver**	**Fully Utilizes Advance Input Data**	**High Output Rate**
Type 1 Low Pass	No	Yes	Yes	Limited given low data rate & target acc.	Limited given low data rate & noisy input	No	Yes
Type 2 Low Pass	No	Yes	Yes	Limited given low data rate & target acc.	Limited given low data rate & noisy input	No	Yes
Kalman	Yes	Yes	Yes	Limited given low data rate & target acc.	Limited given low data rate & noisy input	No	Yes
Least Squares	Yes	Yes	Yes	Enhanced due to advance input data	Enhanced due to advance input data	Yes	Yes

CHAPTER 3

LSSW FILTER DEVELOPMENT

The LSSW filter is divided into two stages: the input stage and the output stage (Figure 3.1). This structure allows for the independent operation of the two stages (i.e., low input rate/high output rate or irregular input rate/uniform output rate), as needed for each application. The following sections provide detailed descriptions for each of the LSSW filter components.

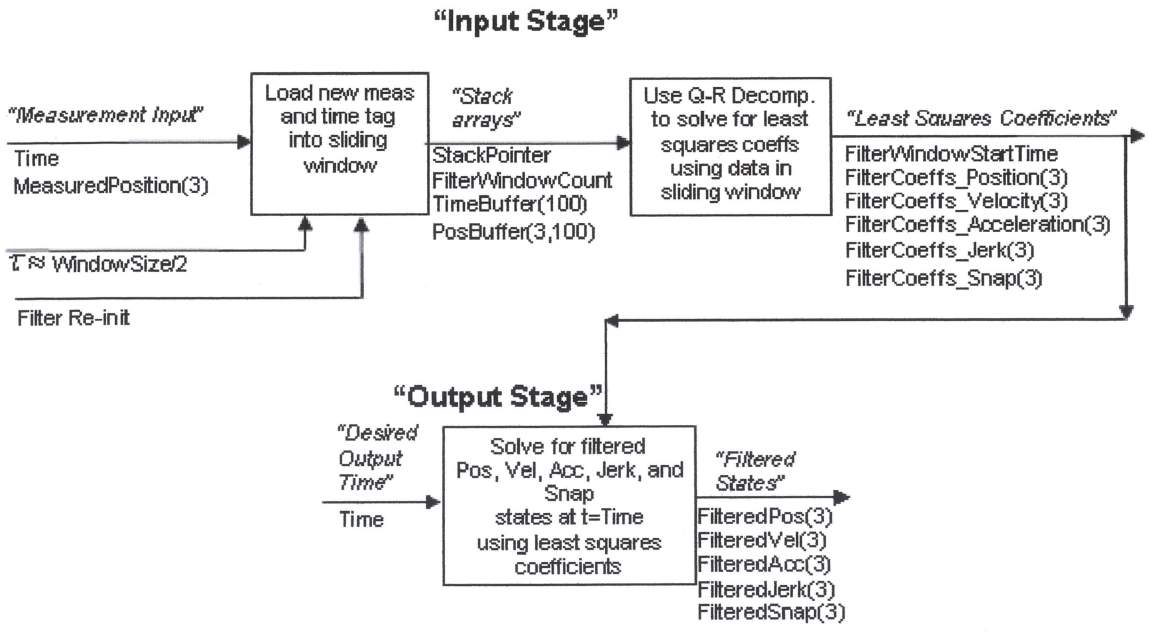

Figure 3.1 LSSW Filter Block Diagram

3.1 Input Stage

The input stage executes upon arrival of new trajectory data (time and measured position: X, Y, and Z). These new trajectory data are loaded into the LSSW filter circular stack. Next, a "window" of data is selected from the circular stack to be used in the solution of the least squares problem. This "window" of data is selected so that it is centered over the output time at which the output states are to be provided. Therefore, as output time is incremented, the "window" of data must slide forward in time in order to remain centered over output time. The filter time constant, τ, can be adjusted by changing the filter window size. The time constant is the measure of the time it takes the filter output to reach 63 percent of the filter input, given a step input. The final step in the input stage is the solution of the least squares problem for the selected "window" of data. Solution of the least squares problem results in coefficients for a polynomial math model that best fits the given set of data points. The least squares problem is solved using the QR decomposition method, which is further explained in Appendix A. This solution produces the following updated coefficients to be used in the output stage.

Table 3.1 LSSW Filter Coefficients

ORDER	**LSSW COEFFICIENTS**
second	position, velocity, acceleration
third	position, velocity, acceleration, jerk
fourth	position, velocity, acceleration, jerk, snap

Velocity, acceleration, jerk, and snap are defined as the first, second, third and fourth derivatives of position.

3.1.1 Sliding Window Implementation

The LSSW filter approach uses a window of near data that is centered at the time for which the estimate is desired. This window consists of m data points, where m is referred to as the window size. A polynomial least squares fit is generated using only the data contained in the window. This method helps eliminate noise while providing excellent estimates of position, velocity, acceleration, and higher order derivatives, as desired. The window size effectively determines the bandwidth of the LSSW filter. The window size must be sufficiently large in order to achieve good noise rejection, but not so large that faithful tracking of trajectory dynamics is sacrificed. Therefore, the window size is a design parameter that is application-specific.

As previously discussed, the LSSW filter must address two different problems when performing trajectory reconstruction. For the first problem of filtering noisy measurement data, one window size may be optimal for getting smooth estimates of position, velocity, and acceleration. However, for the second problem of extrapolating filter states beyond the end of the measurement data, a larger window size performs better because it is necessary also to smooth the higher order states used in the extrapolation. Because of the competing window size requirements of the two problems, provisions have been made to use one window size (*FilterWindowSize*) during periods when measurement data are available and to transition to a different window size (*FlaredFilterWindowSize*) when the measurement data end. *FilterWindowSize* and *FlaredFilterWindowSize* are parameters defined in the namelist.

A typical window size profile consisting of 50 data points is shown in Figure 3.2. Figure 3.3 shows the initialization phase as the filter loads data points until reaching the steady state window size. Figure 3.4 shows the transition to a larger window size of

70 data points as the data file ends at a time of 410 seconds. The beginning of the transition is triggered when the leading edge of the window encounters the end-of-file. The rate of window expansion is set so that the transition to the larger window is complete when filter output time reaches the end-of-file. Figures 3.5 and 3.6 illustrate how the window straddles the output time for the LSSW filter. Figure 3.5 shows the leading edge of the window is (*FilterWindowSize*/2)* ΔT seconds ahead of the filter output time and the trailing edge lags the filter output time by the same amount. Figure 3.6 shows this for the transition at the end-of-file. Beginning at t = 410 sec, the end-of-file has been reached and the LSSW filter is in extrapolation mode.

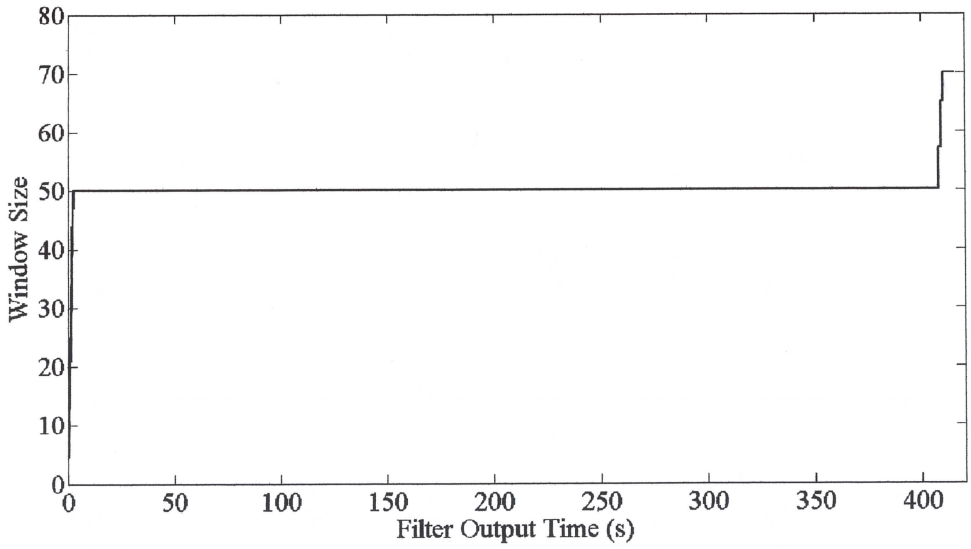

Figure 3.2 Filter Window Size Versus Time

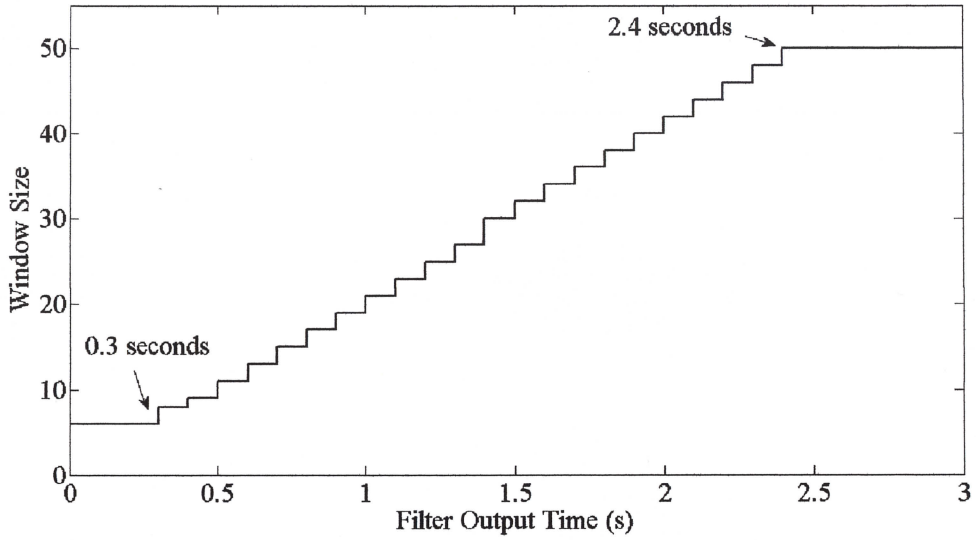

Figure 3.3 Increase of Filter Window Size Versus Time at the Beginning of the Data File

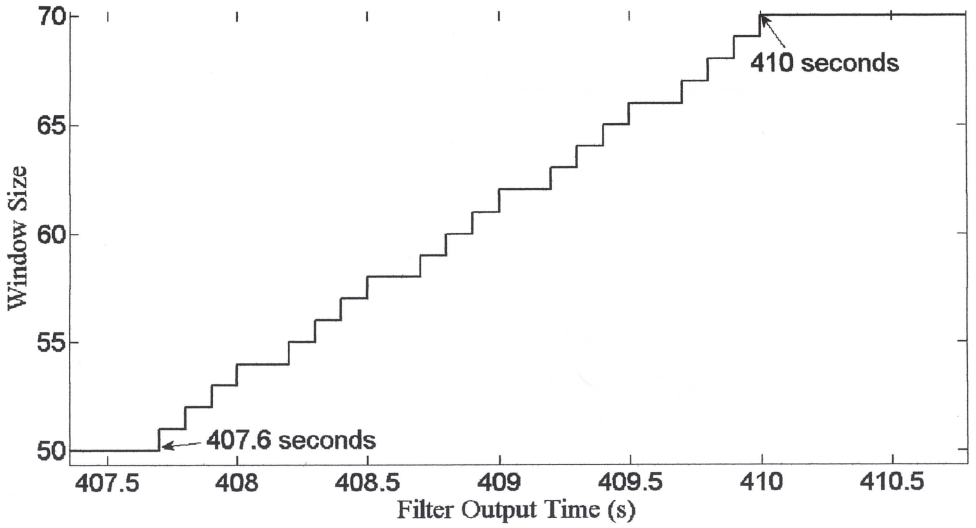

Figure 3.4 Expansion of Filter Window Size Versus Time at the End of the Data File

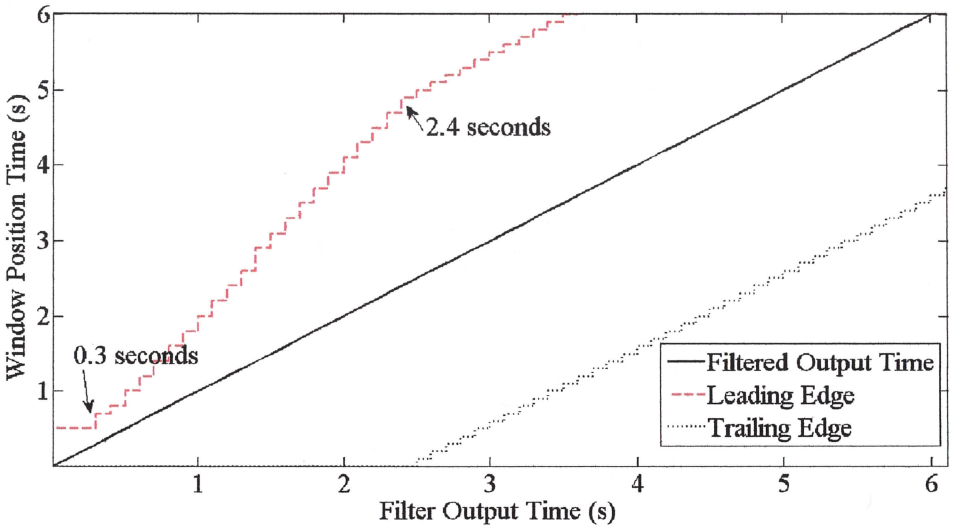

Figure 3.5 Filter Window Leading and Trailing Edges at the Beginning of the Data File

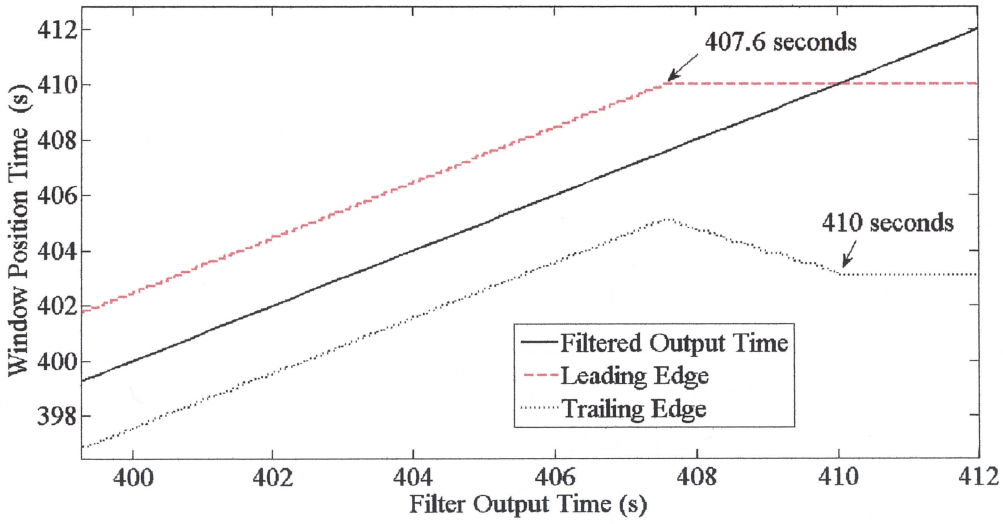

Figure 3.6 Filter Window Leading and Trailing Edges at the End of the Data File

3.1.2 Least Squares Solution

The basic operation of any least-squares solution routine is to minimize the mean square error between the values predicted by a polynomial function and the measured or actual values. The function's absolute squared error is then minimized by optimizing the coefficients of the function [8].

In order to achieve the objectives of target trajectory reconstruction, the LSSW filter must compute several derivatives of position. The first, second, and third derivatives of position are commonly defined as *velocity, acceleration, and jerk*. For this thesis the term *snap* is defined as the fourth derivative of position. The symbols x_n $\dot{x}_n$ $\ddot{x}_n$ $\dddot{x}_n$ $\ddddot{x}_n$ denote the position, velocity, acceleration, jerk, and snap of the target, respectively.

The LSSW filter is capable of utilizing a first, second, third, or fourth-order least-squares solution (2, 3, 4, or 5 states, respectively). The assumed measurement models for each of the

LSSW filter configurations are as follows:

First-Order LSSW Filter

The first-order filter assumes constant velocity over interval T, and is represented by

$$(x_{n+1})_{meas} = (x_n)_{true} + (\dot{x}_n)_{true} * T_n + noise \,. \tag{3.1}$$

Second-Order LSSW Filter

The second-order filter assumes constant acceleration over interval T, and is represented by

$$(x_{n+1})_{meas} = (x_n)_{true} + (\dot{x}_n)_{true} * T_n + (\ddot{x}_n)_{true} * (.5 * T_n^{\,2}) + noise \,. \tag{3.2}$$

Third-Order LSSW Filter

The third-order filter assumes constant jerk (the third derivative of position) over interval T, and is represented by

$$(x_{n+1})_{meas} = (x_n)_{true} + (\dot{x}_n)_{true} * T_n + (\ddot{x}_n)_{true} * (.5 * T_n^{\,2}) + (\dddot{x}_n)_{true} * \left(\frac{1}{6} * T_n^{\,3}\right) + noise \,. \tag{3.3}$$

Fourth-Order LSSW Filter

The fourth-order filter assumes constant snap (the fourth derivative of position) over interval T, and is represented by

$$(x_{n+1})_{meas} = (x_n)_{true} + (\dot{x}_n)_{true} * T_n + (\ddot{x}_n)_{true} * (.5 * T_n^{\,2}) + (\dddot{x}_n)_{true} * \left(\frac{1}{6} * T_n^{\,3}\right) + (\ddddot{x}\,n)_{true} * \left(\frac{1}{24} * T_n^{\,4}\right) + noise \ . \qquad (3.4)$$

There are several methods of solving for least squares; these include solving the normal equations and QR decomposition by both reflectors and rotators [9-11]. Each solution method has its own advantages and disadvantages. Matrix inverse and Gaussian elimination are the two most commonly used methods in solving the normal equations. However, these methods make the assumption that the inverse always exists, which in trajectory reconstruction may not always be the case. Stability problems may occur when an inverse cannot be calculated. As an alternative, QR decomposition has been proven to be more stable and less computationally costly than the normal equation solution.

When the least squares problem is solved using the normal equations, the result is a solution to the equation $Ax = b$. As an example, consider the second-order mathematical model of a trajectory given by

$$f(t_i) = x_1 + x_2 * t_i + x_3 * t_i^{\,2} \ , \qquad (3.5)$$

where x_1, x_2, and x_3 are unknown coefficients, t_i is the measurement time, and $f(t_i)$ is the least squares solution at time, t_i. The *square error* is given by

$$E = \sum_{i=1}^{n} [\, y_i - f(t_i) \,]^2 = \sum_{i=1}^{n} \{\, y_i - [x_1 + x_2 * t_i + x_3 * t_i^2 \,]\}^2 \quad , \tag{3.6}$$

where E is the total square error and y_i is the measurement at time, t_i.

The square error is minimized by setting the partial derivatives of error with respect to each of the unknown coefficients equal to zero.

$$\frac{\partial E}{\partial x_1} = -2 * \sum_{i=1}^{n} \{\, y_i - [x_1 + x_2 * t_i + x_3 * t_i^2 \,]\} = 0 \tag{3.7}$$

$$\frac{\partial E}{\partial x_2} = -2 * t_i * \sum_{i=1}^{n} \{\, y_i - [x_1 + x_2 * t_i + x_3 * t_i^2 \,]\} = 0 \tag{3.8}$$

$$\frac{\partial E}{\partial x_3} = -2 * t_i^2 * \sum_{i=1}^{n} \{\, y_i - [x_1 + x_2 * t_i + x_3 * t_i^2 \,]\} = 0 \quad . \tag{3.9}$$

The *normal equations* for this system are given by

$$\Sigma[\, y_i \,] = x_1 * \sum_{i=1}^{n} [1] + x_2 * \Sigma [\, t_i \,] + x_3 * \Sigma [\, t_i^2 \,] \tag{3.10}$$

$$\sum_{i=1}^{n} [\, t_i * y_i \,] = x_1 * \Sigma [\, t_i \,] + x_2 * \Sigma [\, t_i^2 \,] + x_3 * \Sigma [\, t_i^3 \,] \tag{3.11}$$

$$\Sigma[t_i^2 * y_i \,] = x_1 * \sum_{i=1}^{n} [\, t_i^2 \,] + x_2 * \Sigma [\, t_i^3 \,] + x_3 * \Sigma [\, t_i^4 \,] \quad . \tag{3.12}$$

The normal equations for the second order system can be written in standard matrix, $Ax = b$, form as shown below [12].

$$\begin{bmatrix} n & \Sigma T_i & \Sigma T_i^2 \\ \Sigma T_i & \Sigma T_i^2 & \Sigma T_i^3 \\ \Sigma T_i^2 & \Sigma T_i^3 & \Sigma T_i^4 \end{bmatrix} \begin{pmatrix} \hat{x} \\ \dot{\hat{x}} \\ \frac{1}{2}\ddot{\hat{x}} \end{pmatrix} = \begin{pmatrix} \Sigma y_i \\ \Sigma T_i y_i \\ \Sigma T_i^2 y_i \end{pmatrix} \quad , \tag{3.14}$$

where $T_i = t_i - t_{i-1}$. The solution to the normal equations is $x = A^{-1}b$. It is obtained through the use of inverse matrices or Gaussian elimination. This approach was used for initial development of the filter.

The next two methods use QR decomposition in order to find a solution. Of the several types of QR decomposition, the two most common methods use reflectors and rotators. It was eventually decided that QR decomposition by reflectors would be used in the final filter model. This was because the reflectors method has inherent stability and speed superior to the other methods.

A detailed description of the algorithm from James F. Epperson is included in Appendix A [13]. The algorithm is based on the decomposition by reflectors.

3.2 Output Stage

The output stage uses the coefficients generated in the input stage to compute output states at time, t. The number of states written to the output file depends on the order of the filter. The addition of another order will produce three additional outputs.

Table 3.2 LSSW Output States

ORDER	LSSW OUTPUT STATES
second	position, velocity, acceleration
third	position, velocity, acceleration, jerk
Fourth	position, velocity, acceleration, jerk, snap

3.3 LSSW Filter Frequency Response

When a filter is discussed, the topic of frequency response typically arises. In order to find the frequency response of the LSSW filter, two approaches were taken:

analytical and experimental. The analytical approach takes the basic filter equations and solves for the closed form solution at a constant time step, ΔT. The frequency response can then be found for each order at varying window sizes. A Matlab code was written to analytically determine the LSSW filter frequency response. Table 3.3 shows the normalized 3dB bandwidth as a function of window size and filter order. Normalized bandwidth is defined as bandwidth divided by the sample frequency.

Figures 3.7, 3.8, and 3.9 were generated with the Matlab *fvtool*. These plots show the filter response for each order in graphical form. The plots show the filter responded as expected. The results show that as the order of the filter is increased the bandwidth slightly increases and as the window size is increased, the bandwidth decreases. The plots show that bandwidth is more sensitive to window size than filter order.

Table 3.3 Closed Form LSSW Frequency Response

Order	Windowsize	3 dB Normalized Bandwidth
2nd	20	0.0443
	40	0.0221
	60	0.0147
	80	0.0111
	100	0.0088
3rd	20	0.1069
	40	0.0532
	60	0.0355
	80	0.0266
	100	0.0213
4th	20	0.1069
	40	0.0532
	60	0.0355
	80	0.0266
	100	0.0213

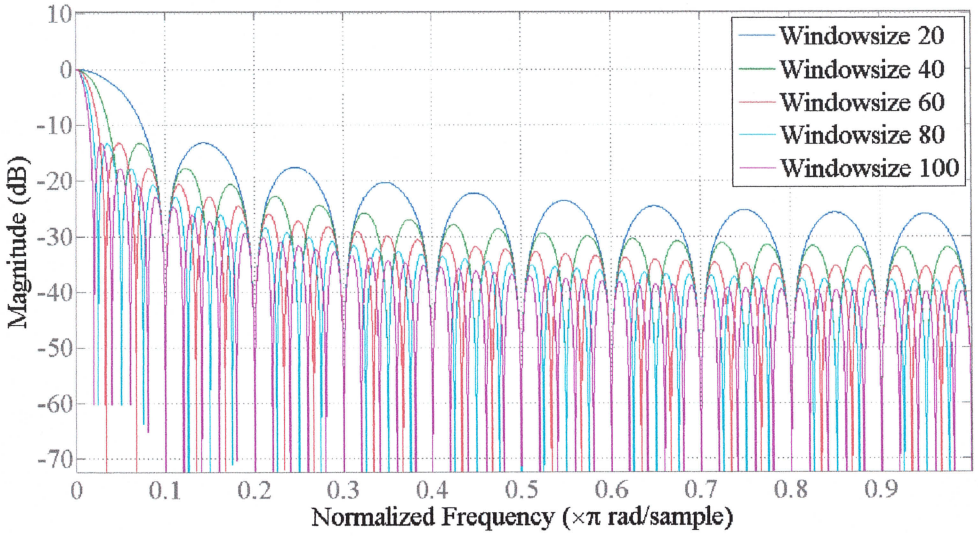

Figure 3.7 Second-Order Frequency Response

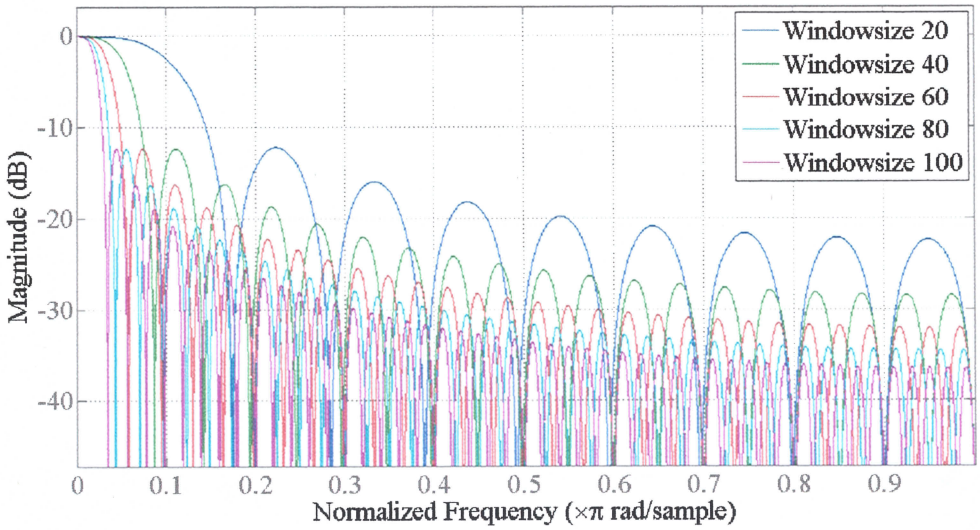

Figure 3.8 Third-Order Frequency Response

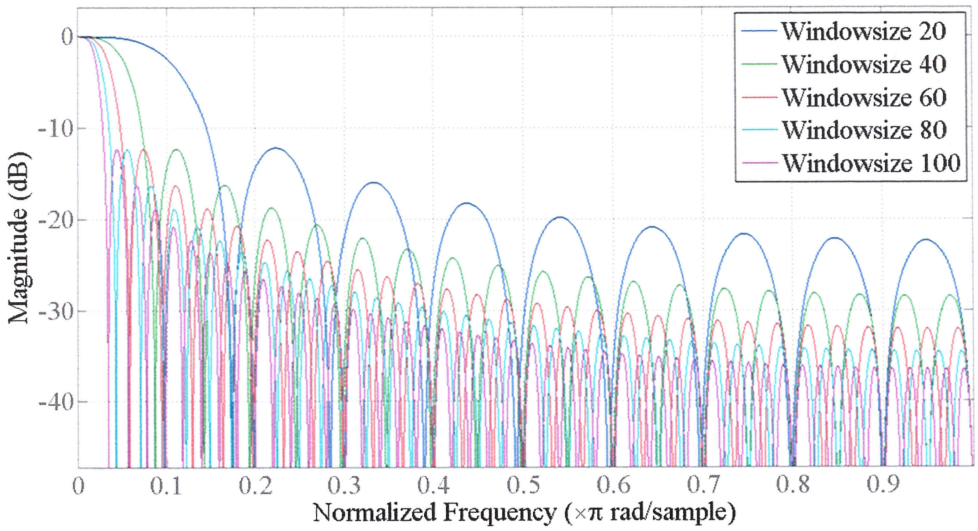

Figure 3.9 Fourth-Order Frequency Response

In order to establish a connection between the analytical frequency response described and the actual LSSW filer implementation, spot checks were performed using an experimental approach. For this approach, a simple sine wave was generated and passed through the filter. The sine wave was created using the makesin.f routine, which is included in Appendix D. The input frequency range was 10, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 Hertz with an amplitude of 1. These twenty-one sine waves were then run with the filter for second, third, and fourth-orders with window sizes of 20, 40, 60, 80, and 100 points. There was no extrapolation, and the sample rate was fixed at $\Delta T = 0.0001$ sec. An abbreviated set of results for 10, 100, 500, and 1000 Hz is summarized in Table 3.4. The experimental results in Table 3.4 show that a fourth-order filter increases the bandwidth slightly over second and third-order filters. These experimental results match the trend obtained in the analytical solution, again showing that the bandwidth is a stronger function of window size than the order. This can be seen in Figure 3.10, which shows the normalized 3dB

bandwidth versus window size. Normalized bandwidth is bandwidth divided by the sample frequency. From this data the bandwidth can be estimated. In order to establish a simple relationship between bandwidth and window size, empirical equations were developed to fit the data, which are also shown in Figure 3.10.

Table 3.4 LSSW Filter Frequency Response to a Sine Wave Input

Hz	Window Size	Second-Order	Third-order	Fourth-order
10	20	1	1	1
10	40	1	1	1
10	60	1	1	1
10	80	1	1	1
10	100	1	1	1
100	20	0.999	0.999	1
100	40	0.992	0.992	1
100	60	0.961	0.961	0.999
100	80	0.888	0.888	0.994
100	100	0.76	0.76	0.978
500	20	0.757	0.77	0.978
500	40	0.189	0.189	0.437
500	60	0.084	0.085	0.249
500	80	0.047	0.048	0.151
500	100	0.03	0.031	0.1
1000	20	0.187	0.187	0.448
1000	40	0.047	0.047	0.151
1000	60	0.021	0.021	0.07
1000	80	0.012	0.012	0.04
1000	100	0.007	0.007	0.026

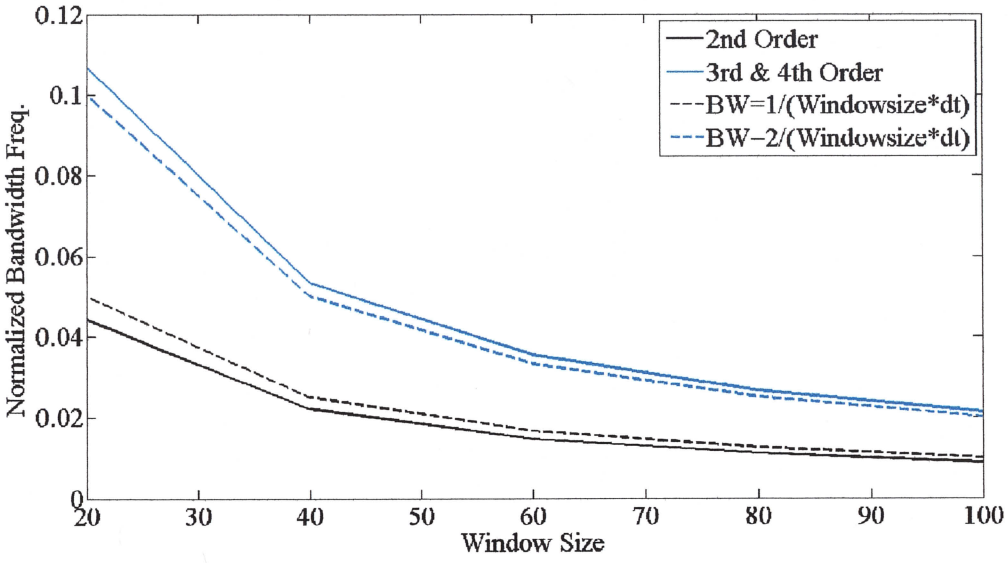

Figure 3.10 3dB Bandwidth Versus WindowSize

From the sine wave input data, the bandwidth (Hz) for the LSSW filter, can be approximated by

$$BW = \frac{2}{\text{WindowSize} * dt} \quad . \tag{3.15}$$

The time constant, τ, can be computed from bandwidth as

$$\tau = \frac{1}{BW * 2\pi} \quad . \tag{3.16}$$

CHAPTER 4

MATLAB APPLICATIONS FOR LSSW FILTER

The LSSW filter developed in Chapter 3 can be run two ways: with a namelist exclusively, or with Matlab. If the filter is being run directly through a namelist, a data file, the executable, and the specific namelist are essential. In each case, the input data file must be in the 10 column format with no headers. The required column inputs, also shown in Table 4.1, are

Time Position1 Position2 Position3 Velocity1 Velocity2 Velocity3 Acceleration1 Acceleration2 Acceleration3.

Table 4.1 Format for 10 Column Data File

	Position (X, Y, Z)			Velocity			Acceleration		
Time	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
0	1.79	-0.22	-0.96	13.04	-1.86	-15.94	92.73	-11.18	-37.99
0.1	3.55	-0.46	-2.76	22.35	-3.03	-19.73	92.03	-10.90	-41.12
0.2	6.25	-0.82	-4.93	31.55	-4.17	-23.83	91.49	-10.58	-44.31
more	.	...	...	...	...	...	...	...	...

The namelist should look like

File:LSSWFilter.NAM

```
&INPUT
InputFileName = "yourdatafile.dat"
OutputFileName = "present_output.xls"
AnalysisFileName = "present_analysis.xls"
FilterWindowSize = 50
FlaredFilterWindowSize=80
OutputDT = 0.1
NORDER=3
TIMEPAST=5
/
```

This takes the input file *yourdatafile.dat* and filters it using the third-order LSSW solution, with a window size of 50, a flared filter window size of 80, an output data rate of 0.1seconds, and outputs 2 files (*present_output.xls* and *present_analysis.xls*). The flared window size is the window size used when extrapolation mode is implemented. At the end of data, the filter transitions from the original value for *FilterWindowSize* of 50 to the value of *FlaredFilterWindowSize* of 80. The increase in *FlaredFilterWindowSize* is necessary to accurately compute the higher order derivative used in the extrapolation. These output files are in the following 13 column format like the input file with the jerk states added to the end:

Time Position1 Position2 Position3 Velocity1 Velocity2 Velocity3 Acceleration1 Acceleration2 Acceleration3 Jerk1 Jerk2 Jerk3

This format is also shown in Table 4.2. The window size, the order, and the extrapolation time can be changed. The filter is implemented with two FORTRAN files, which are included in the Appendices B and C.

Table 4.2 Format for 13 Column Data File

	Position			Velocity			Acceleration			Jerk		
Time	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
0	1.79	-0.22	-0.96	13.04	-1.86	-15.94	92.73	-11.18	-37.99			
0.1	3.55	-0.46	-2.76	22.35	-3.03	-19.73	92.03	-10.90	-41.12			
0.2	6.25	-0.82	-4.93	31.55	-4.17	-23.83	91.49	-10.58	44.31			
more	.	...	...	...	...	...	...	...	...			

The second method of running the filter requires the commercial package Matlab. This method takes advantage of Matlab's visualization tools in order to determine the best window size for the filter. In most cases, the user can make an accurate estimation of the window size before running the filter, but the window size may need to be further refined. This method requires a copy of Matlab, the folder *LSSWVizMatlab*, and a data file in the 10 column format of Table 4.1.

In order to optimize the process of selecting the best window size, a Matlab script was written that allows the user to choose visually the optimal window size from 1 to 100. This script prompts the user to read in a trajectory file to be filtered. It then runs the filter for window sizes of 10-100 in increments of 10 and opens Matlab to visualize the position states. The user then further refines the window range and selects the best window size, for which the output trajectory file is then generated.

To implement the Matlab LSSW filter method, the user performs the following steps:

1.) Copy the *LSSWVizMatlab* folder to the computer. This folder contains 12 Matlab scripts, 1 DOS batch script, and 2 executable files, which are shown in Table 4.3.

Table 4.3 Matlab Computer Files

DESCRIPTION	**FILE NAME**
drives the main program	maindriver.m
plots the data for the three position states for each of the window ranges	testplot, testplot0 thru 9
DOS batch script that runs the filer in increments of 10 from 1 to 100	Runset
executable file for the filter	LSSW filterCartesian.exe
executable file for generating the 100 namelists	gennam

2.) The user next places the trajectory file in the same folder used in step (1). The program is next executed by typing *maindriver* in the command window.

3.) The user will is next prompted to enter:

- the filename of the trajectory file (The user must place the trajectory file in the directory from which the bat file is run.)
- the output ΔT
- second, third, or fourth-order
- time past last data point (i.e., 5 seconds of extrapolation)
- flared filter window size

At this point Matlab generates three plots: X,Y, and Z, each versus time. Each of these three plots shows 10 curves (i.e., window sizes of 10-100 in increments of 10). Based on the results of the first cut of the window size parameterization, the user can then focus on a narrower range of window size (i.e., 21-30 in increments of 1). The user then picks the optimal window size for which the output file, *outputfile.xls*, is generated.

CHAPTER 5

LSSW FILTER DEMONSTRATION

A generic TBM trajectory was used to demonstrate the LSSW filter. This example exhibits characteristics typically associated with TBM trajectories that might affect filter performance. The true trajectory includes only dynamics effects. The measured trajectory (measurement data) is equivalent to the true trajectory plus measurement noise. This measured trajectory is input to the LSSW filter, which, in turn, produces the filtered trajectory.

The first part of this test demonstrates the LSSW filter's noise rejection capability during time periods when measurement data are available. The figure of merit is residual standard deviation. This allows for comparison of noise rejection properties. The second part of this test demonstrates the LSSW filter's capability to extrapolate filtered states in the absence of measurement data. The figure of merit for the extrapolation mode is position error, which is the difference between the true position and the filter estimation. For this example this is done at five seconds after the end of measurement data.

5.1 Example Problem Description

The TBM trajectory used for this example was generated using a generic flight dynamics simulation and input parameters that are not keyed to any specific real-world

TBM entity. Any resemblance to actual trajectories of actual TBM vehicles is strictly coincidental. The factors that prevent these data from being arbitrary include the preservation of the governing laws of physics and the use of sound engineering judgment to select input parameters considered to fall within real-world realizable boundaries.

The generic TBM trajectory is defined by the following parameters:

Time of flight = 425 seconds

Maximum downrange travel of 500 km

Apogee of 160 km

No crossrange component

Missile burnout = 70 seconds

Noise: 1 sigma σ = normally distributed with zero mean and 5 meter standard deviation in each dimension

Data rate: 10 Hz

From these parameters, a complete three-dimensional trajectory was generated and corresponding real or "true" velocity and acceleration components were calculated. All position information is provided in units of meters; velocity is in meters per second; and time is in seconds. The data is in target launcher reference frame and considered in the North East Down (NED) Cartesian coordinate system.

To make the analysis more realistic, measurement noise was introduced. This noise was added with the FORTRAN program *addnoise.f*, which is included in Appendix E. The noise is normally distributed with zero mean and 5 meter standard deviation in each dimension.

5.2 Trajectory Reconstruction

The following plots illustrate a typical non-maneuvering TBM. This TBM is traveling in the X-Z plane, where X is downrange and Z is down. The missile's general behavior is indicated by plots showing the true, measured, and filtered data along each axis. Error plots are provided to demonstrate the difference in the true, measured, and filtered data.

Figures 5.1 and 5.2 show the missile's true position, measured position, and filtered position components of the trajectory with respect to time for a fourth-order filter with a window size of 50. This filter configuration was chosen to be a reasonably good fit for the trajectory behavior, but not necessarily optimal. Figure 5.1 shows the maximum down range position to be 500 km. Figure 5.2 shows the altitude profile of the TBM. Since the coordinates are in North East Down (NED), the values are negative. In this example there is no crossrange component. The intent of these plots is to show the general nature of the TBM trajectory. The differences between the three curves are better shown by the position error plots which are included later.

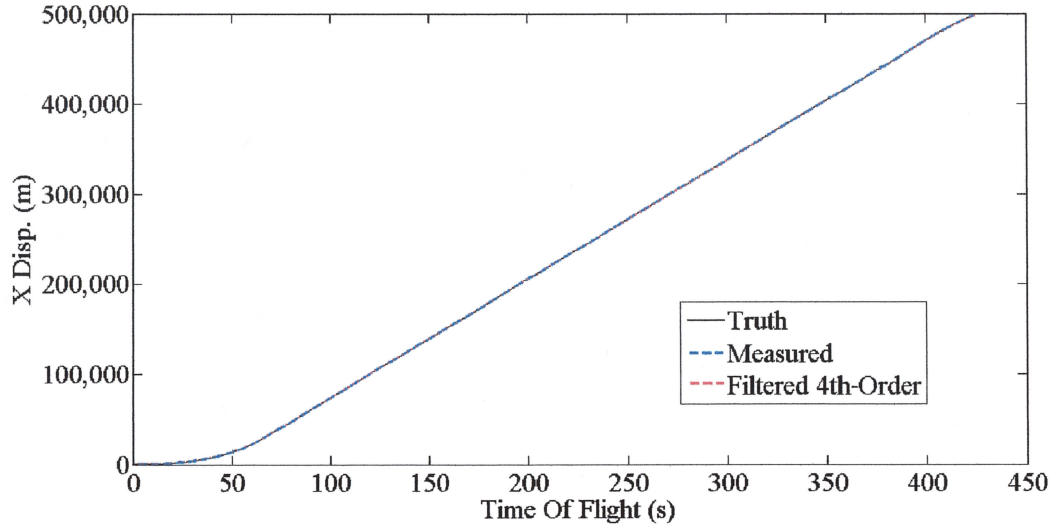

Figure 5.1 True, Measured, and Filtered Target Trajectories: X Position Versus Time

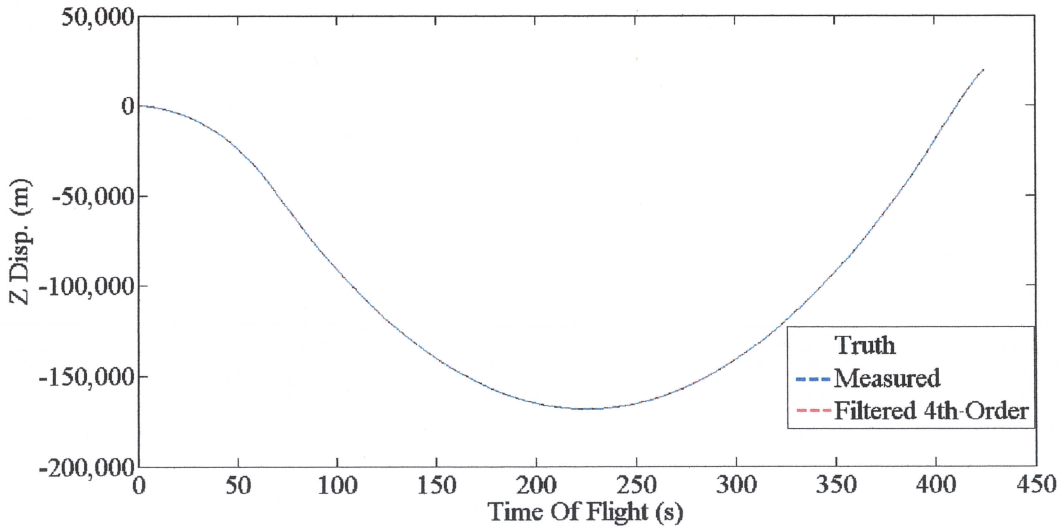

Figure 5.2 True, Measured, and Filtered Target Trajectories: Z Position Versus Time

Figures 5.3 and 5.4 illustrate the missile's true velocity, measured velocity, and filtered velocity trajectory components with respect to time for a fourth-order filter with a window size of 50. Figures 5.3 and 5.4 show that the missile burnout occurs at 70 seconds. The X velocity increases after launch until burnout where a maximum is

reached around 1300 m/s. The maximum Z velocity is approximately 1600 m/s. The velocity then remains relatively constant until the end of flight where the missile decelerates.

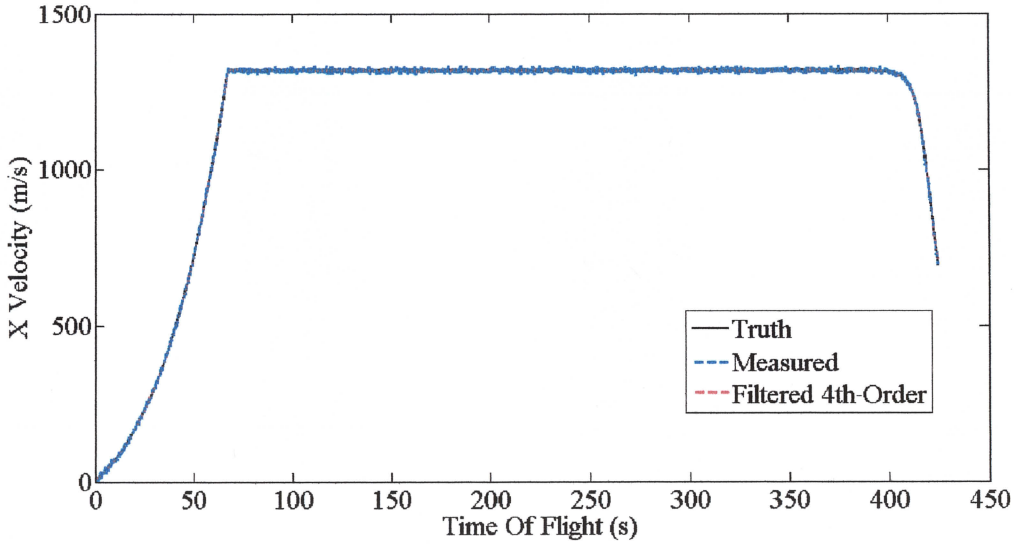

Figure 5.3 True, Measured, and Filtered Target Trajectories: X Velocity Versus Time

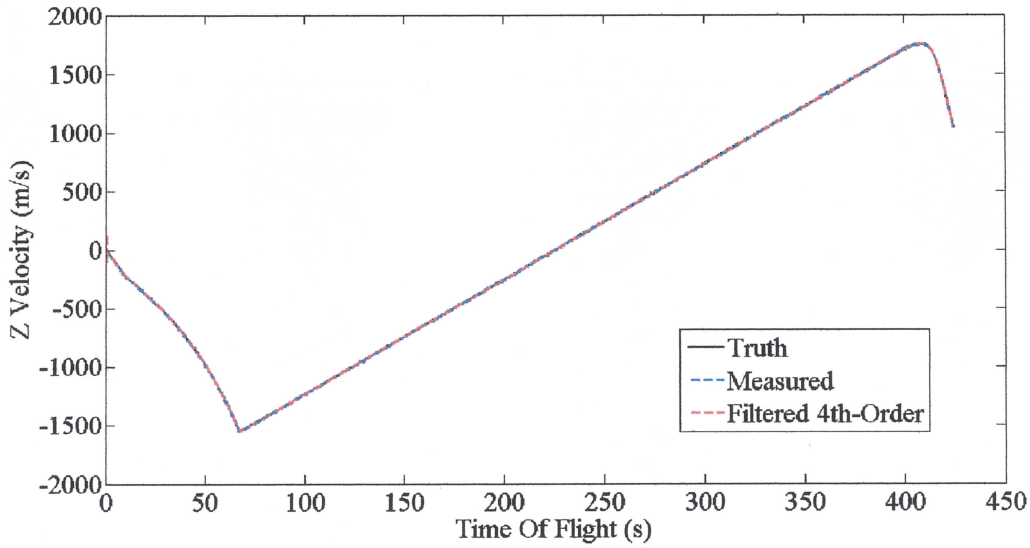

Figure 5.4 True, Measured, and Filtered Target Trajectories: Z Velocity Versus Time

Figures 5.5 and 5.6 illustrate the missile's true acceleration, measured acceleration, and filtered acceleration components of the trajectory with respect to time for a fourth-order filter with a window size of 50. Again the burnout at 70 seconds can be seen. The noise level of the filtered fourth-order data is +/- 3 m/s^2.

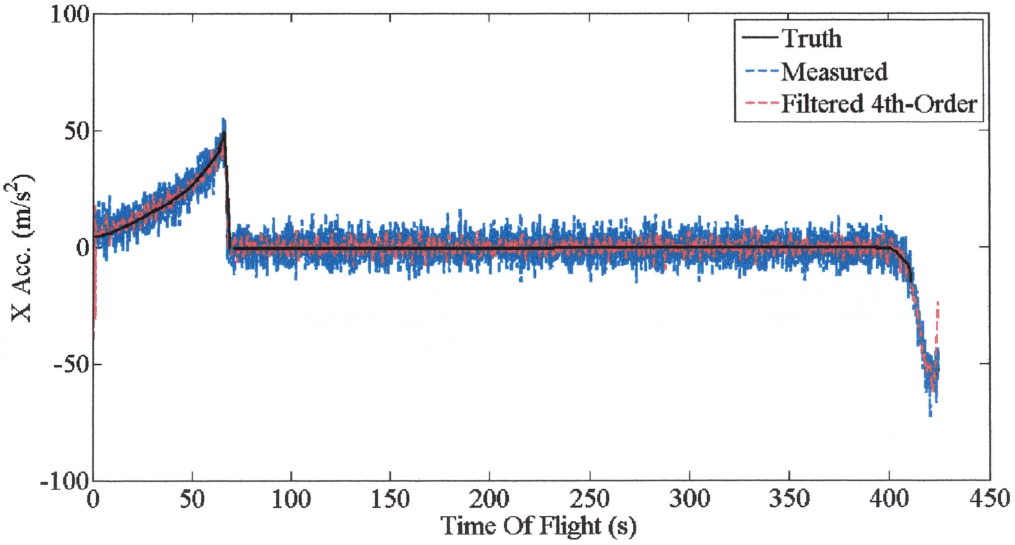

Figure 5.5 True, Measured, and Filtered Target Trajectories: X Acceleration Versus Time

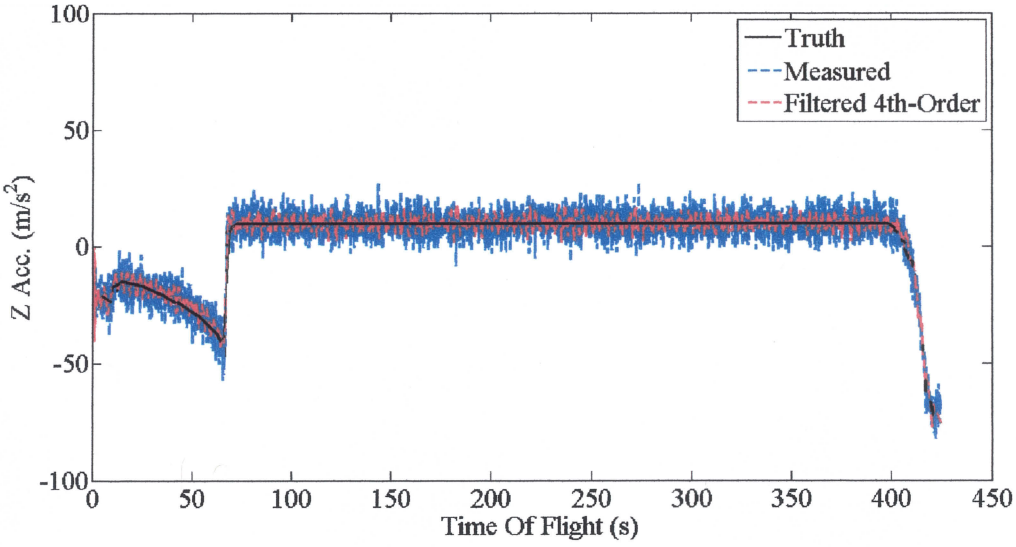

Figure 5.6 True, Measured, and Filtered Target Trajectories: Z Acceleration Versus Time

Figures 5.7 and 5.8 show the missile's true jerk, measured jerk, and filtered jerk components of the trajectory with respect to time for a fourth-order filter with a window size of 50. These figures show jerk to be constant except during missile motor burnout and at the end of the flight.

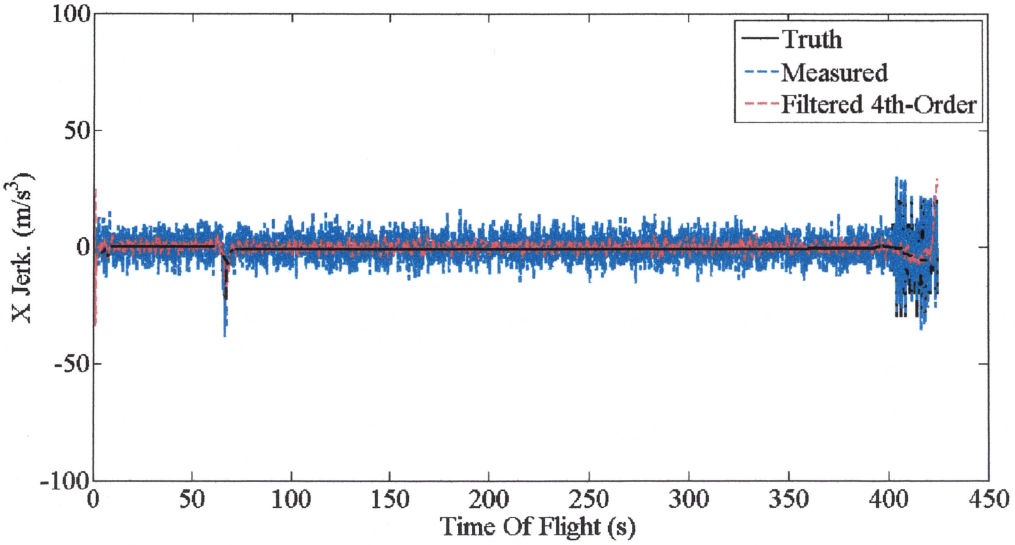

Figure 5.7 True, Measured, and Filtered Target Trajectories: X Jerk Versus Time

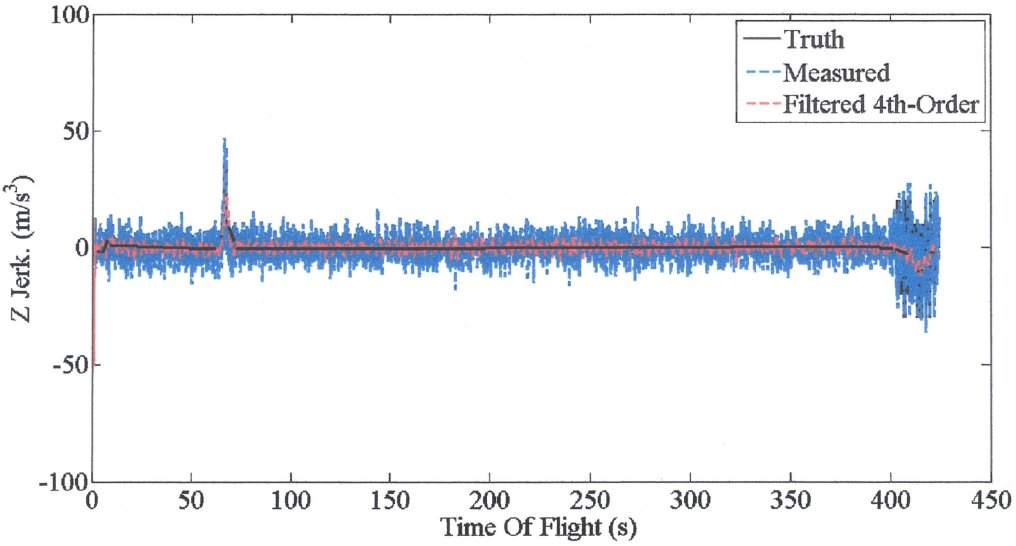

Figure 5.8 True, Measured, and Filtered Target Trajectories: Z Jerk Versus Time

Figures 5.9 and 5.10 show measured position error and filtered position error of the trajectory with respect to time for a fourth-order filter with a window size of 50. The measured position errors are consistent with the 1 sigma noise variation that was added to the original trajectory. The filtered position error plots show that the LSSW filter reduces noise without introducing any bias or divergence problems.

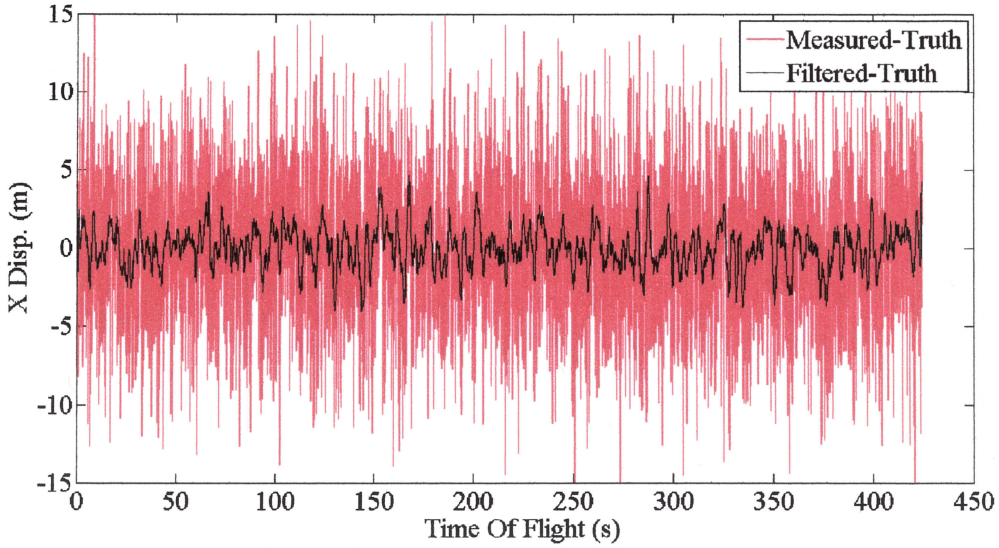

Figure 5.9 Measured and Filtered X Position Error Versus Time

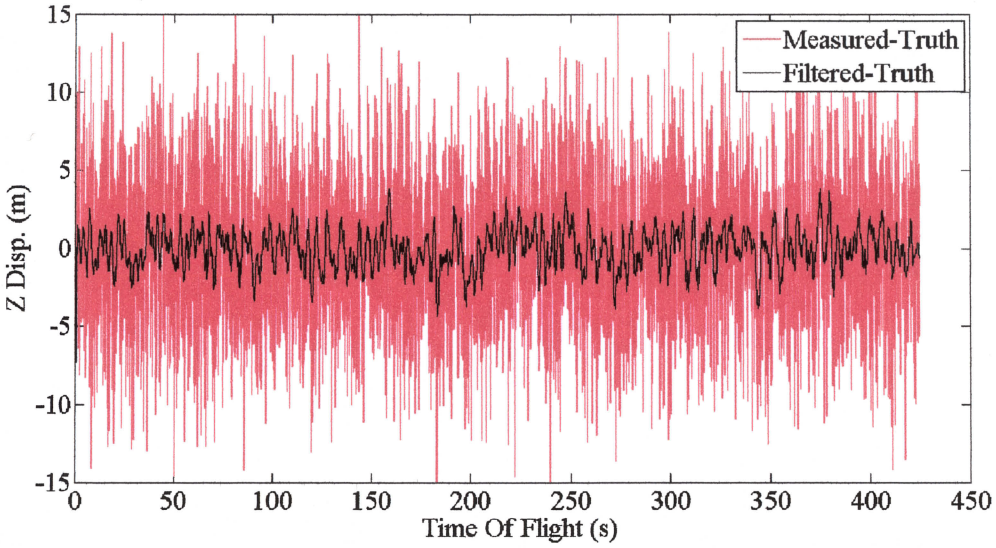

Figure 5.10 Measured and Filtered Z Position Error Versus Time

Figures 5.11 and 5.12 show the measured velocity errors and filtered velocity errors of the trajectory with respect to time for a fourth-order filter with a window size of 50. As seen in the position plots, the noise is reduced without introducing unwanted errors.

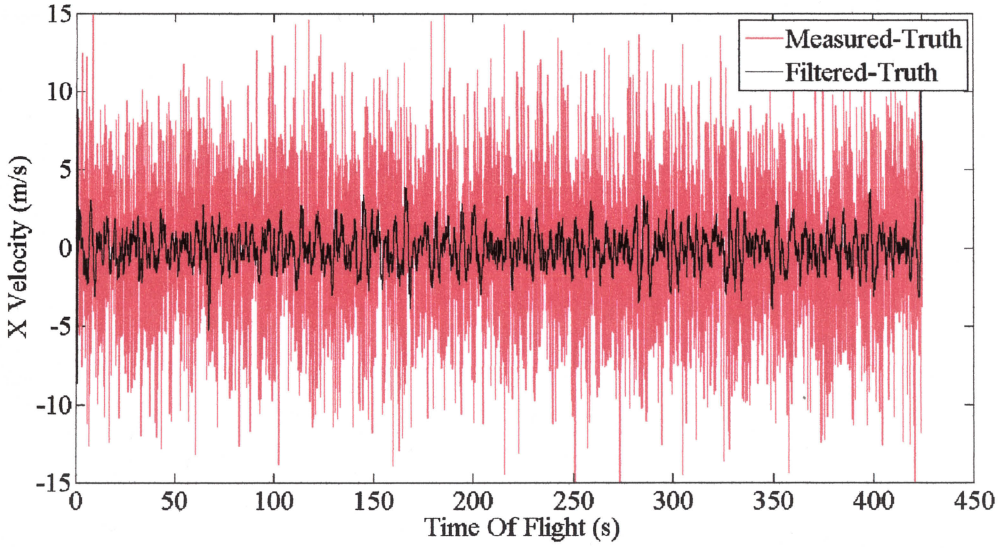

Figure 5.11 Measured and Filtered X Velocity Error Versus Time

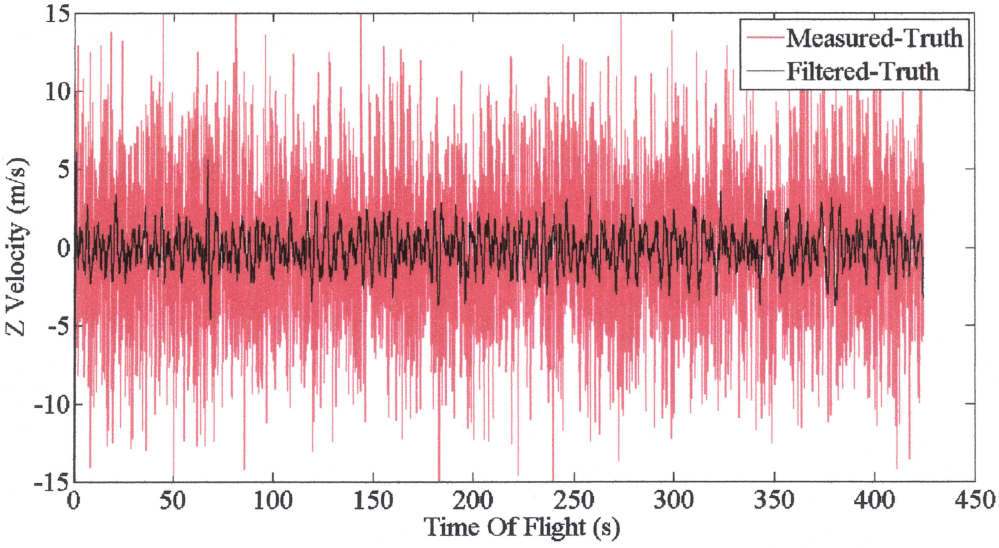

Figure 5.12 Measured and Filtered Z Velocity Error Versus Time

Figures 5.13 and 5.14 show the measured acceleration errors and filtered acceleration errors of the trajectory with respect to time for a fourth-order filter with a window size of 50. These plots show that the LSSW filter introduces no bias or divergence problems.

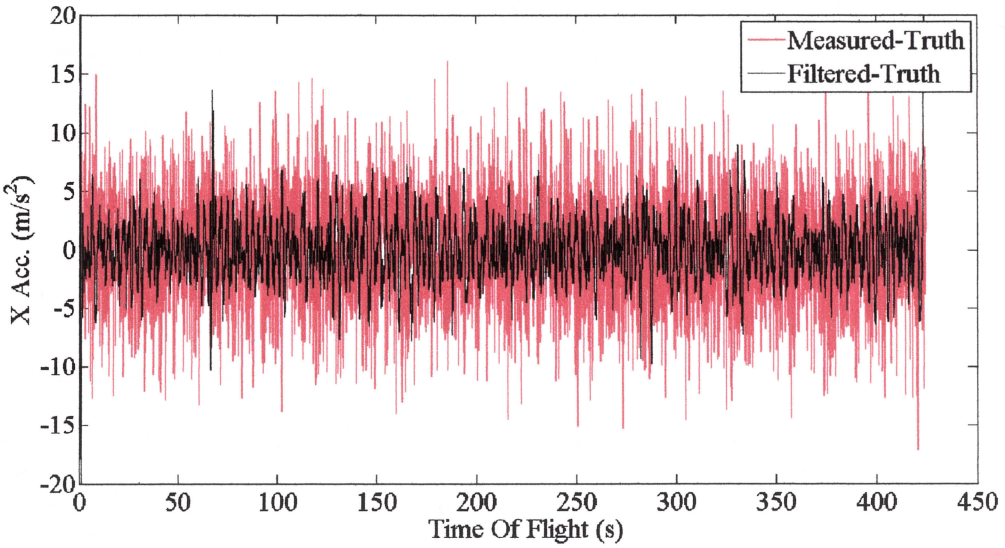

Figure 5.13 Measured and Filtered X Acceleration Error Versus Time

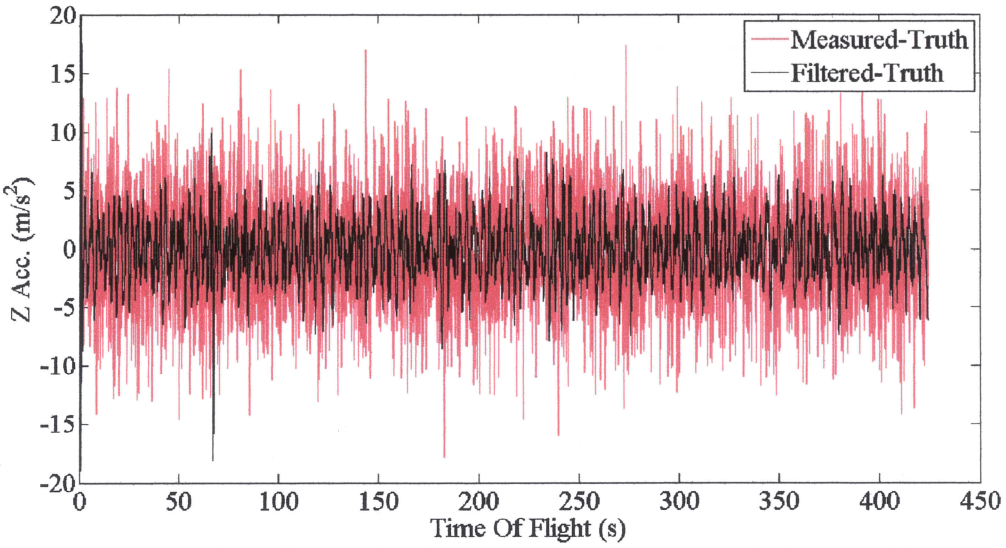

Figure 5.14 Measured and Filtered Z Acceleration Error Versus Time

Residual standard deviations were used as the figure of merit in order to compare noise rejection properties of the various configurations of the LSSW filter (order and window size). Table 5.1 shows residual standard deviation of X position errors for the generic TBM. This example case was configured as fourth-order with a fixed window size of 50, which corresponds to a residual standard deviation of 1.317. Table 5.1 can be used to determine the optimal window size and filter order, based on the smallest residual computed. The optimal filter configuration for this example, as show in Table 5.1, would be fourth-order with a window size of 100. This corresponds to the smallest residual standard deviation.

Table 5.1 Residual Standard Deviation for Example 1

	Window								
Order	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
2	1.6322	1.3505	1.2115	1.1293	1.0937	1.1026	1.1750	1.3288	1.6022
3	1.6442	1.3540	1.2114	1.1236	1.0866	1.0911	1.1500	1.2718	1.4620
4	2.0376	1.6653	1.4439	1.3171	1.2254	1.1601	1.1137	1.0831	1.0703

Now, the effect of filter input drop-outs on filter performance must be addressed. Figures 5.15-5.18 show the actual filter response to drop-outs for the generic TBM example. The same data set is used except that a three second measurement drop-out that begins at a time of 165 seconds into flight and ends at 168 seconds has been simulated. The filter remains configured as fourth-order with a window size of 50, and at a data rate of 10 Hz. These plots show that during the three second measurement drop-out, the filter outputs remain well-behaved. The filter output transitions smoothly through the drop-out period without discontinuities or transients when the input stream returns.

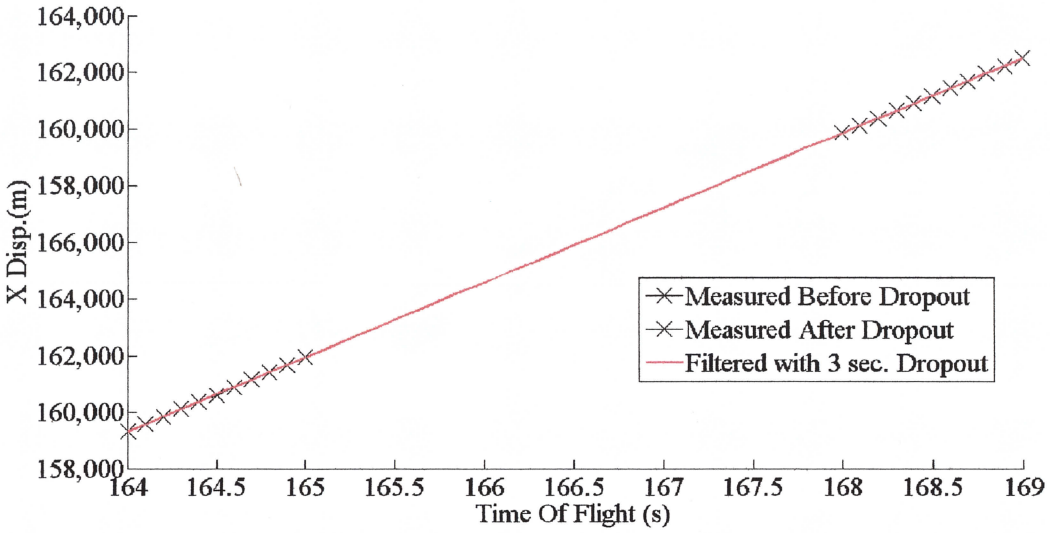

Figure 5.15 X Displacement with a Drop-out Versus Time

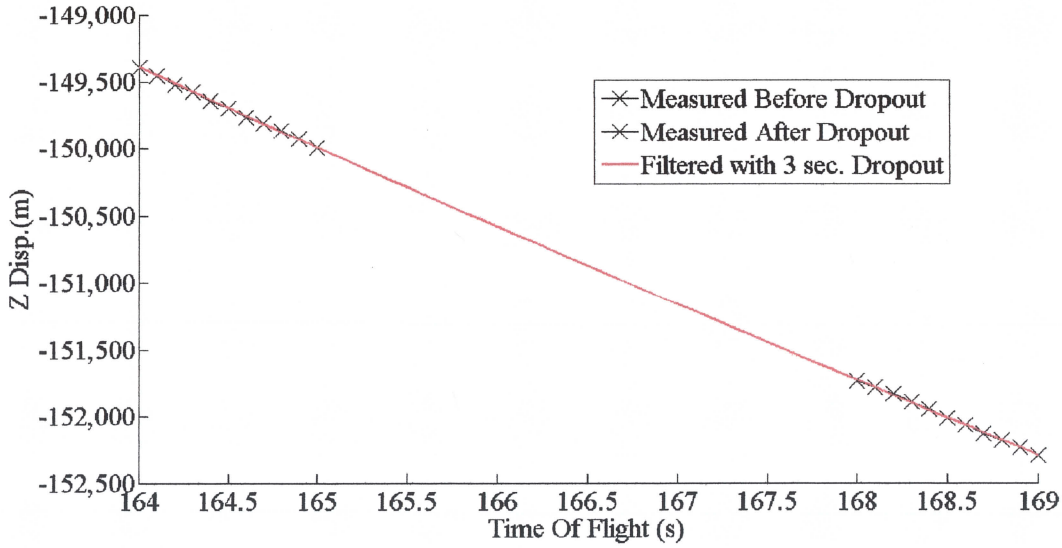

Figure 5.16 Z Displacement with a Drop-out Versus Time

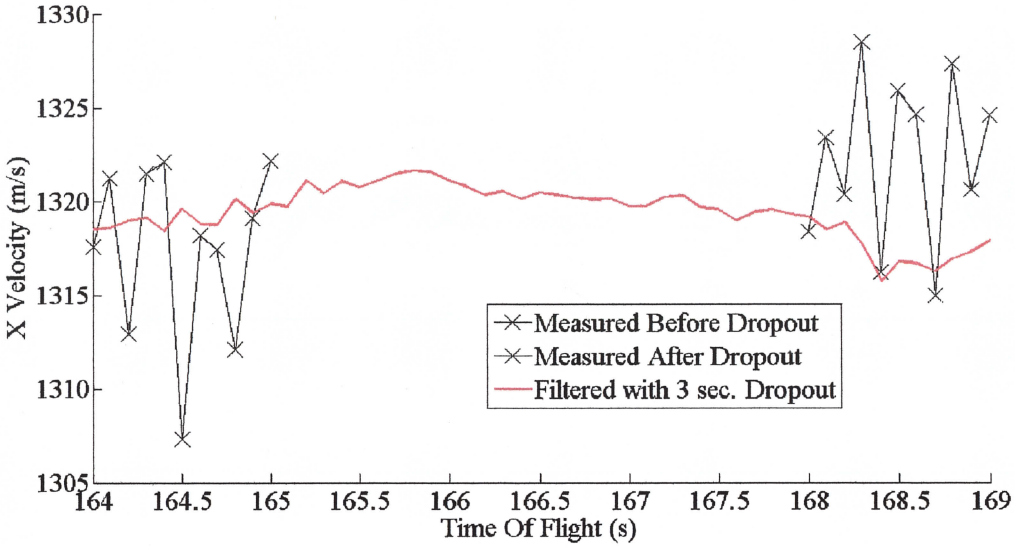

Figure 5.17 X Velocity with a Drop-out Versus Time

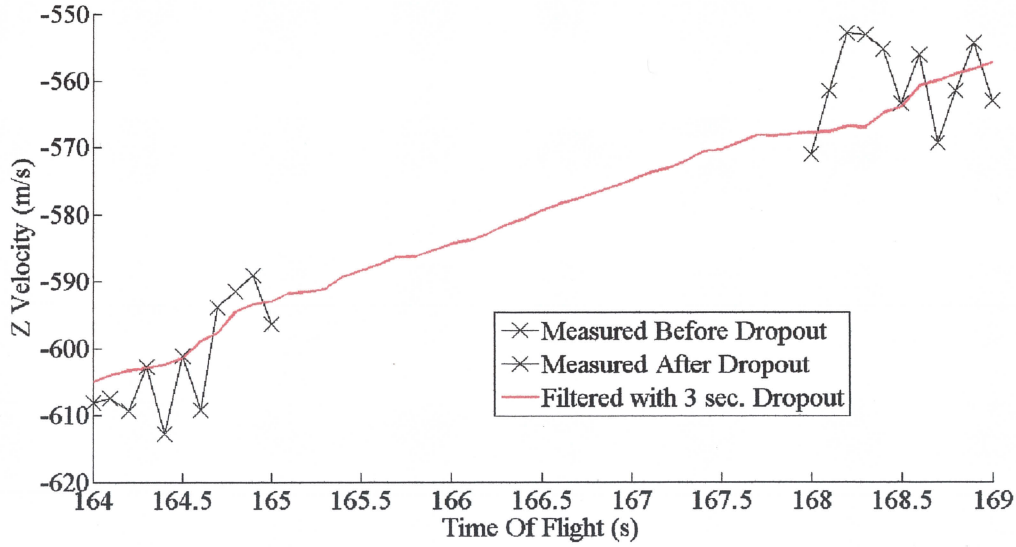

Figure 5.18 Z Velocity with a Drop-out Versus Time

To evaluate the effect of drop-outs on the LSSW filter frequency response, Matlab was used. Since the window moves over the entire drop-out period, it is necessary to select several cases representative of this behavior for analysis in Matlab.

The first case involves data drop-out near the leading edge of the window. The second case simulates data drop-out occurring in the center of the window. The third case simulates drop-out near the trailing edge of the window.

Table 5.2 shows normalized bandwidth frequency for data drop-out periods ranging from 0 to 30 samples in duration and for each of the three cases to assess sensitivity to the positioning of the window relative to the drop-out period. Table 5.2 shows that as the duration of the drop-out period increases, there is some affect on the frequency response, but this is not considered significant. Also Table 5.2 shows some variation in frequency response due to relative positioning of the window with respect to the drop-out period (cases 1-3). Again this variation is considered to be relatively small in terms of its impact on filter performance. Figures 5.15-5.19 show that even with a three second drop-out, the filter still performs as desired.

Table 5.2 Drop-out Frequency Response

Number of Drop-outs	Case 1	Case 2	Case 3
0	0.0426	0.0426	0.0426
1	0.0422	0.0435	0.0422
2	0.0418	0.0446	0.0419
3	0.0414	0.0456	0.0415
4	0.0411	0.0468	0.041
5	0.0408	0.0479	0.0404
6	0.0405	0.049	0.0398
7	0.0404	0.051	0.0393
8	0.0402	0.0512	0.039
9	0.0401	0.0523	0.039
10	0.0401	0.0533	0.0393
11	0.0401	0.0544	0.0402
12	0.0401	0.0553	0.0416
13	0.0401	0.0563	0.0435
14	0.0402	0.0572	0.0459
15	0.0403	0.0581	0.0485
16	0.0403	0.0589	0.0511
17	0.0404	0.0598	0.0534
18	0.0405	0.0606	0.0553
19	0.0405	0.0613	0.057
20	0.0406	0.0621	0.0585
25	0.041	0.0653	0.0631
30	0.0413	0.0681	0.0656

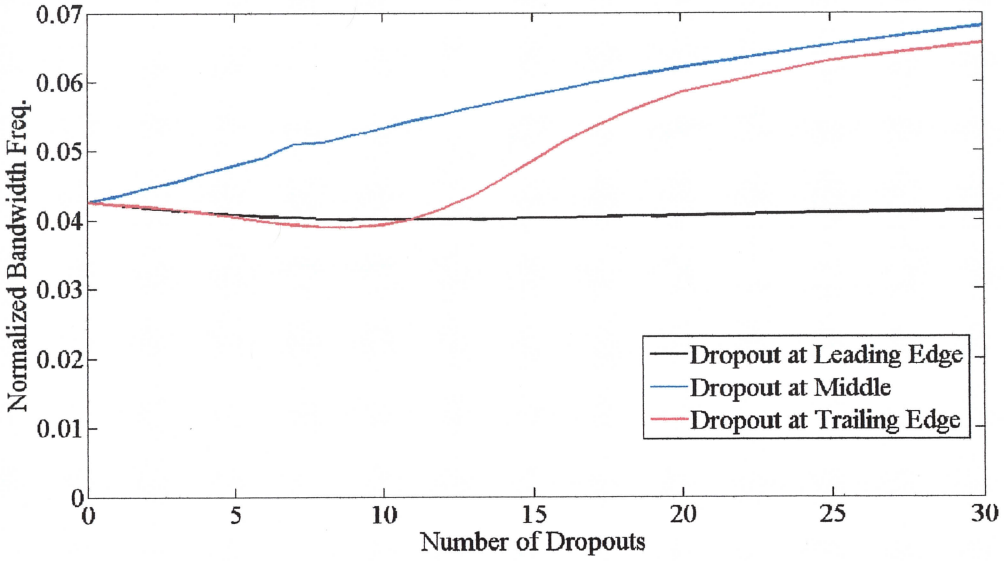

Figure 5.19 Number of Drop-outs for each Case

5.3 Trajectory Extrapolation Beyond End-of-Data

Sometimes during post-flight analysis the target is intercepted at a lower altitude. In cases such as these, the measured data stream ends early. As a result, the data must be extrapolated for use in the simulation. The following example demonstrates the extrapolation capabilities of the LSSW filter.

For this example, the generic TBM trajectory that was generated previously was used. As mentioned earlier, the original trajectory is 425 seconds long. The original data file was truncated at 410 seconds to provide sufficient data for comparison. The truncated data set was run through first, second, third, and fourth-order filters, and extrapolated past the end of data for five seconds. The following namelist was used for the fourth-order filter:

```
&INPUT
 InputFileName   = "meas10c_410.dat"
 OutputFileName  = "filt16.dat"
 AnalysisFileName = "present_analysis.dat"
 OutputFileName2= "generictbm.txt"
 FilterWindowSize = 50
 FlaredFilterWindowSize=70
 OutputDT = 0.1
 NORDER=4
 TIMEPAST=5
/
```

To simplify matters, only the X position, velocity, and acceleration were analyzed. The filter window size and the flared filter window size were set at 50 and 70, respectively, for each run. Figures 5.20-5.25 show the filter's output for five seconds past the end of data for X position, velocity, and acceleration.

The usefulness of the higher order configurations is best shown in Figures 5.21, 5.22, 5.23, 5.24, and 5.25. These figures demonstrate that the third and fourth-orders show significant improvement in the ability to estimate the target behavior beyond the end of data. This is clearly shown in the extrapolation mode figure of merit. For this case, the filter extrapolates 5 seconds past the end of data, representing a worst case scenario. Ordinarily, only one or two seconds of extrapolation would be required.

Figure 5.20 is a plot of the X position for the truth, measured and filtered first through fourth-orders for the entire trajectory. The filter's extrapolation performance is clearly illustrated in Figure 5.21 which is a zoom-in of the last 5 seconds when the extrapolation occurs. When extrapolating, a first-order approximation is traditionally used. However, this obviously does not render the most accurate results. One way to

improve reconstruction accuracy is to use higher orders to estimate the trajectory. Figure 5.21 shows that as the order increases, the filter more closely approximates the truth. For this case, the fourth-order filter gives the closest approximation of the actual trajectory. In fact, in most situations involving extrapolating TBM states, the fourth-order filter would be appropriate without inducing excessive computational burden.

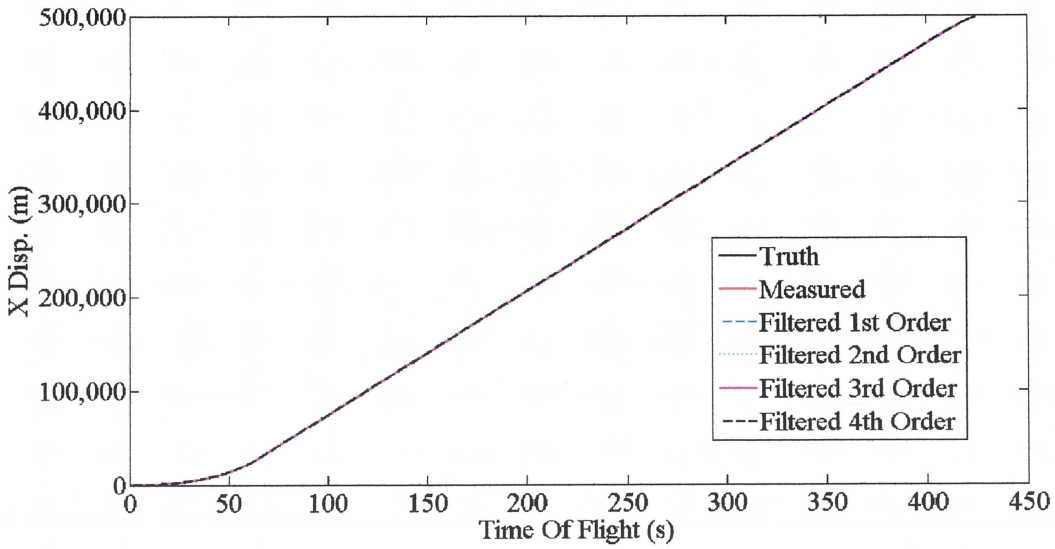

Figure 5.20 True, Measured, and Filtered Extrapolated X Position Versus Time

Figure 5.21 is a zoom-in of the end of the trajectory where the extrapolation takes place.

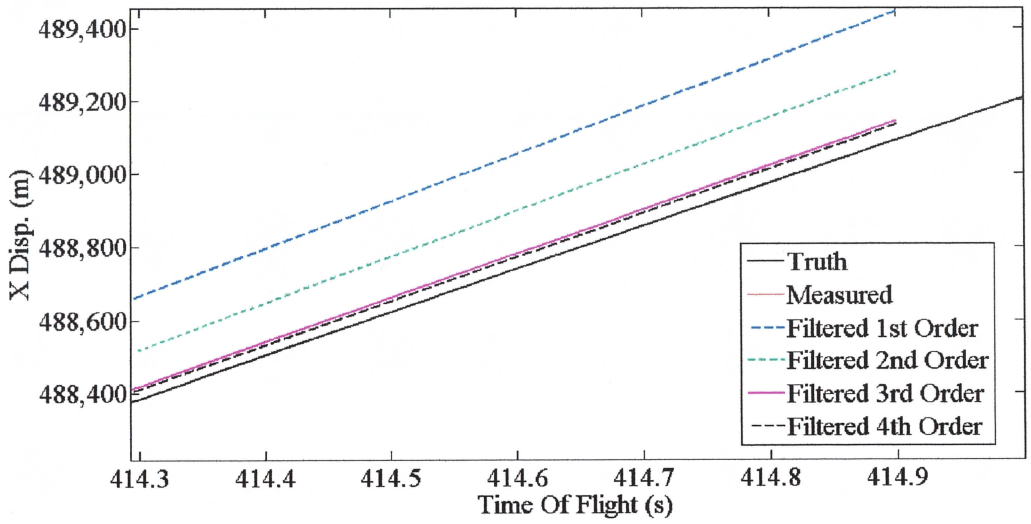

Figure 5.21 Zoom of True, Measured, and Filtered Extrapolated X Position Versus Time

Figures 5.22 and 5.23 show the X velocity components for the truth and first through fourth-order filtered data. Figure 5.23 is a zoom-in of the extrapolation. Similarly to the X position plots, Figures 5.22 and 5.23 also show that an increase in order yields a better approximation of the truth. Again the fourth-order is the closest to the original trajectory. The poor performance of the first and second orders illustrates the necessity of using the higher order filter for extrapolation.

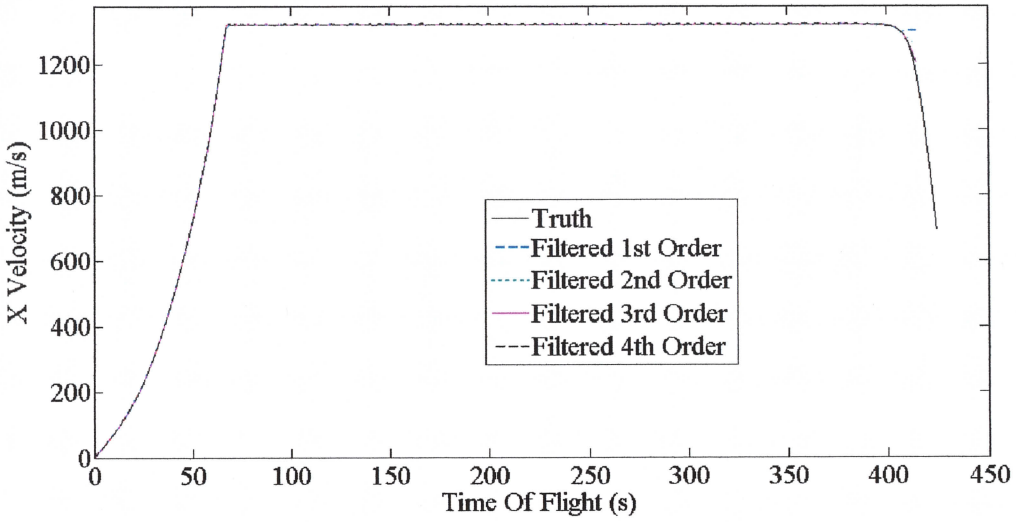

Figure 5.22 True and Filtered Extrapolated Velocity Versus Time

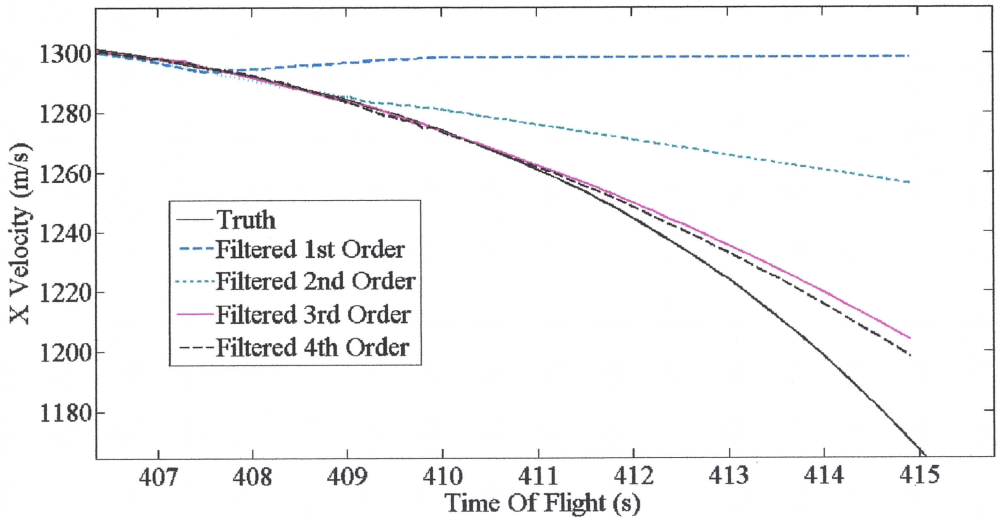

Figure 5.23 Zoom of True and Filtered Extrapolated Velocity Versus Time

Figures 5.24 and 5.25 show the acceleration in the X direction for the truth and filtered first through fourth-order data. Figure 5.25 is a zoom-in of Figure 5.24 and shows the last 14 seconds of the trajectory. As with the position and velocity, the

acceleration data show that a fourth-order filter produces the best representation of the true trajectory.

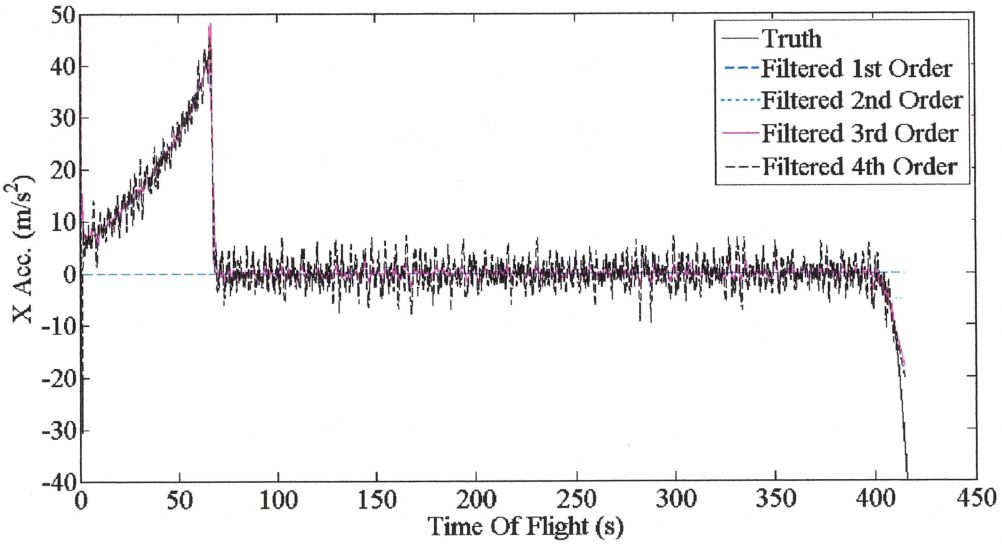

Figure 5.24 True and Filtered Extrapolated Acceleration Versus Time

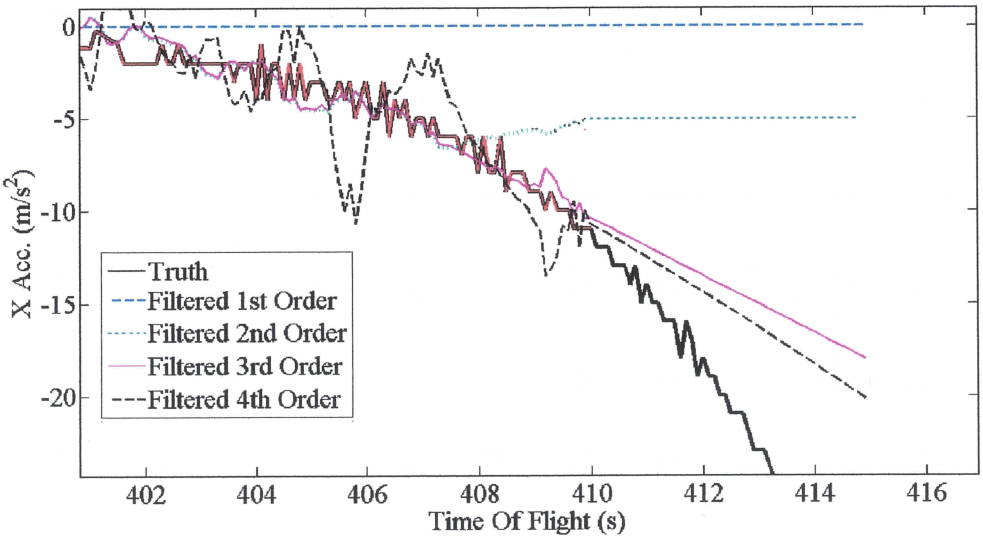

Figure 5.25 Zoom of True and Filtered Extrapolated Acceleration Versus Time

These figures demonstrate that the third and fourth-order filters show significant improvement in the ability to estimate the target behavior beyond the end of data. This is

clearly shown in the extrapolation mode figure of merit. For this case, the filter extrapolates 5 seconds past the end of data, representing a worst case scenario. Ordinarily, only one or two seconds of extrapolation would be required.

CHAPTER 6

CONCLUSION

In this thesis, a new tool for trajectory reconstruction is proposed. This tool is necessary to fulfill the air defense missile community's need for accurate target trajectories in order to evaluate the performance of current missile guidance algorithms. The Least Squares Sliding Window filter (LSSW Filter) is designed to perform trajectory reconstruction from flight test data that is noisy with irregular data rates and drop-outs. This is done with an innovative method that utilizes a sliding window. The user can choose a first-order to fourth-order filter depending on the simulation requirements. The lower order configurations allow for better noise rejection and lend themselves to targets that are less dynamic, such as Air Breathing Threats (ABTs). The higher order configurations still exhibit good noise rejection properties but also provide high-order extrapolation capabilities. In general, higher order derivative models are considered better for tracking agile maneuvers, such as those exhibited by Tactical Ballistic Missiles (TBMs).

Simulation results presented were based on two example cases of a generic TBM missile. Through the example cases, it was demonstrated that the LSSW filter is capable of rejecting the noise at low-input data rate without sacrificing responsiveness to target

dynamics. It was also shown that the filter provides the ability to extrapolate states in the absence of measurement data. The sliding window technique provides the key for this method's success. It allows incorporation of advance measurements and also limits the undesirable effects of old measurements. Overall, for the TBM scenario tested, the LSSW filter satisfies all of the original requirements.

One area with room for improvement is during extrapolation of TBM states beyond the end of data. Future development could also include the addition an atmospheric model. This model could be initialized with filter states in order to more accurately predict TBM accelerations. Another possible improvement would be to adjust the window size dynamically depending on target behavior.

The LSSW filter can be utilized for multiple purposes. It can be used to perform merely as a simple data filter, to manipulate trajectory sample rate, to reconstruct the flight-test target trajectory in order to permit a meaningful post-flight analysis, or even to extend the data set with the extrapolation mode. Use of this method will allow for improved post-flight analysis to be conducted in support of the ultimate goal of missile system simulation validation.

APPENDICES

APPENDIX A
QR DECOMPOSITION

QR Decomposition By Reflectors

1. For i= 1 to n do (where n is the order of the system)

a. {Compute the i^{th} reflector and apply it to the i^{th} column}

i. $\mu = \max_{j \geq i} |a_{ji}|$

ii. for j = i to m do $a_{ji} = a_{ji} / \mu$

iii. $\hat{\sigma} = \text{sgn}(a_{ii}) \sqrt{\sum_{j=i}^{m} \alpha_{ji}^2}$

iv. $u_i = \alpha_{ii} + \hat{\sigma}$

for j = i + 1 to m do $u_j = \alpha_{ji}$

$\gamma = (\hat{\sigma} u_i)^{-1}, \sigma = \hat{\sigma}\mu$

$a_{ii} = -\sigma$

b. for j = i + 1 do {Apply the reflector to the other columns of A.}

i. $\beta = \gamma \sum_{k=i}^{m} u_k a_{kj}$

ii. for k = i to m do

$a_{kj} = a_{kj} - \beta u_k$

c.{Apply the reflector to the right side vector, b.}

i. $\beta = \gamma \sum_{k=i}^{m} u_k b_k$

ii. for k = i to m do

$b_k = b_k - \beta u_k$

APPENDIX B
LSSW FILTER CODE

The following Fortran 90 code was created for the LSSW filter. It is the main driver for the filter and uses the QRDecomposition.f90 subroutine which is presented in Appendix C.

```
PROGRAM LSSWFilter_Cartesian
 USE QRDecomposition
 INTEGER(KIND=4), PARAMETER :: MaxFilterWindowSize = 100
 INTEGER(KIND=4), SAVE :: FilterWindowSize
 REAL(KIND=8), DIMENSION(MaxFilterWindowSize), SAVE :: TimeBuffer
 REAL(KIND=8), DIMENSION(MaxFilterWindowSize), SAVE :: PosBufferX,    &
.                          PosBufferY,PosBufferZ
 REAL(KIND=8), DIMENSION(MaxFilterWindowSize,5), SAVE :: DTMatrix
 REAL(KIND=8), DIMENSION(5), SAVE :: FilterCoeffsX, FilterCoeffsY,    &
.                          FilterCoeffsZ
 REAL(KIND=8), DIMENSION(3,MaxFilterWindowSize), SAVE ::              &
.    PosBufferXYZ,VelBufferXYZ,AccBufferXYZ,JrkBufferXYZ,SnapBufferXYZ
 INTEGER(KIND=4), SAVE :: FilterWindowCount
 INTEGER(KIND=4), SAVE :: StackPointer
 INTEGER(KIND=4), SAVE :: NumberValid
 INTEGER(KIND=4), SAVE :: IORDER
 INTEGER(KIND=4), SAVE :: NORDER
 INTEGER(KIND=4), SAVE :: TIMEPAST
```

```
INTEGER(KIND=4), SAVE :: PNTCNT
INTEGER(KIND=4), SAVE :: IExtrap
INTEGER(KIND=4), SAVE :: FlaredFilterWindowSize
REAL(KIND=8), SAVE :: FilterWindowStartTime,FilterWindowEndTime
INTEGER(KIND=4), PARAMETER :: POS = 1
INTEGER(KIND=4), PARAMETER :: VEL = 2
INTEGER(KIND=4), PARAMETER :: ACC = 3
INTEGER(KIND=4), PARAMETER :: JRK = 4
INTEGER(KIND=4), PARAMETER :: SNAP= 5
INTEGER(KIND=4) :: I, jj
LOGICAL, SAVE :: EndOfFile
REAL(KIND=8) :: InputTime,DeltaInputTime,StartInputTime,EndInputTime
REAL(KIND=8) :: OutputTime,OutputDT,LastTime,DeltaTimeToTableEnd
REAL(KIND=8) :: MeasuredX,MeasuredY,MeasuredZ
REAL(KIND=8) :: FilteredX(5),FilteredY(5),FilteredZ(5)
INTEGER(KIND=4) :: IARGC
CHARACTER(LEN=255) ::                                          &
.      NamelistFileName,InputFileName,OutputFileName,          &
.      AnalysisFileName,OutputFileName2
REAL(KIND=8), DIMENSION(1:3) :: MeasuredXYZPosition,           &
.           MeasuredXYZVelocity, MeasuredXYZAcceleration,      &
.           MeasuredXYZJerk, MeasuredXYZSnap
REAL(KIND=8), DIMENSION(1:3) ::                                &
```

```
.       LastMeasuredXYZPosition,LastMeasuredXYZVelocity,                    &
.       LastMeasuredXYZAcceleration,LastMeasuredXYZJerk
  REAL(KIND=8), DIMENSION(1:3) ::                                           &
.         OutputXYZPosition,OutputXYZVelocity,OutputXYZAcceleration,        &
.         OutputXYZJerk,OutputXYZSnap
  NAMELIST /INPUT/ InputFileName,OutputFileName,AnalysisFileName,           &
FilterWindowSize,FlaredFilterWindowSize,OutputDT,NORDER,TIMEPAST,           &
OutputFileName2
!*****************************************************************
!PURPOSE:
!       This is the main driver for the Sliding Window Filter .
!       It reads in position data in a 10 column format and outputs files
!       with the filtered data for position, velocity, acceleration, and
!       jerk. The user can define several parameters for the filter.
!       The user enters these parameter via namelist input. The user has
!       control of:
!             the order of the filter (second or third)
!             the output DT
!             the number of seconds past the end of data for extrapolation
!             flared window size
!
!INPUTS:
!    Namelist File-
```

```
!       Example:
!        &INPUT
!         InputFileName  = measzz2.dat
!         OutputFileName = "present_output10.xls"
!         AnalysisFileName = "present_analysis10.xls"
!         OutputFileName2= "outputtraj10.trj"
!         OutputDT =  0.1
!         FilterWindowSize= 10
!         FlaredFilterWindowSize= 80
!         NORDER=3
!         TIMEPAST=   0
!        /
!
!    Input File-
!            format:
!             1st line = total number of lines (delete for plotting, etc.)
!             Columns are:
!              1   - Time (s)
!              2-4 - position (m)
!              5-7 - velocity (m/s)
!              8-10- Acceleration(m/s^2)
!
! OUTPUTS;
```

! Analysis File

! format:

! Columns are:

! 1 - Time (s)

! 2-4 - Position (m)

! 5-7 - Velocity (m/s)

! 8-10- Acceleration(m/s^2)

! 11-13 Jerk(m/s^3)

! 14-15 Snap (m/s^4)

! Output File 1

! format:

! Columns are:

! 1 - Time (s)

! 2-4 - Position (m)

! 5-7 - Velocity (m/s)

! 8-10- Acceleration(m/s^2)

! 11-13 Jerk(m/s^3)

! 14-16 Snap (m/s^4)

! Output File 2

! format:

! 1st line = total number of lines

! Columns are:

! 1 - Time (s)

```
!           2-4 - position (m)
!           5-7 - velocity (m/s)
*****************************************************************
  PNTCNT=0
! Check for command line input of namelist file name
! The Function IARGC() returns the argument count.  If it finds nothing there it
! prompts the user to enter the file name
  IF (IARGC() == 0) THEN
    WRITE(*,*) 'Enter Namelist Input File Name:'
    READ(*,'(A)') InputFileName
  ELSE
    CALL GETARG(1,InputFileName)
  END IF
!             Opens the input file for read
  OPEN(1,FILE=InputFileName,STATUS='OLD',POSITION='REWIND')
!             Read in driver namelist
  READ(1,NML=INPUT)
!             Closes the namelist file and keeps it
  CLOSE(1,STATUS='KEEP')
!             Initialize filter
  CALL InitializeFilter
!             Open Files
  OPEN(1,FILE=InputFileName,STATUS='OLD',POSITION='REWIND')
```

```
  OPEN(2,FILE=OutputFileName,STATUS='REPLACE',POSITION='REWIND')
  OPEN(3,FILE=AnalysisFileName,STATUS='REPLACE',POSITION='REWIND')
 OPEN(4,FILE=OutputFileName2,STATUS='REPLACE',POSITION='REWIND')
!               Start filter
  EndOfFile = .FALSE.
  CALL StartFilter
!               Start time loop
  OutputTime = StartInputTime
  EndInputTime = 1.D30
  TimeLoop: DO
!               Check to see if any new input data is needed
    InputLoop: DO
      IF (OutputTime <= (FilterWindowEndTime - DeltaTimeToTableEnd)) EXIT
InputLoop
      CALL InputMeasurement
      IF (EndOfFile) EXIT InputLoop
    END DO InputLoop
!               Raw (measured) states at output time
    CALL OutputRawStates(OutputTime)
!               Filter states at output time
    CALL OutputFilterStates(OutputTime)
!               Output data
! uncomment to write to screen
```

```
!    WRITE(*,'(F12.3,3X,3f12.3,3X,3F12.3,3X,3F12.4,3X,3F14.4)')
&
OutputTime,MeasuredXYZPosition,OutputXYZPosition,                    &
OutputXYZVelocity,OutputXYZAcceleration,OutputXYZJerk                &
!     WRITE(*,*) 'OutputTime  :MeasuredXYZPosition', OutputTime,      &
MeasuredXYZPosition
     WRITE(2,'(F12.3,3X,3F12.3,3X,3F12.4,3X,3F12.4,3X,3F14.4,3X,3F12.3)')  &
OutputTime, OutputXYZPosition, OutputXYZVelocity, OutputXYZAcceleration,  &
OutputXYZJerk, OutputXYZSnap
     WRITE(3,'(F12.3,3X,3F12.3,3X,3F12.4,3X,3F12.4,3X,3F14.4,3X,3F12.3)')  &
OutputTime, MeasuredXYZPosition, MeasuredXYZVelocity,                &
MeasuredXYZAcceleration, MeasuredXYZJerk, MeasuredXYZSnap            &
!                    Update time, exit when last input time point reached  &
     OutputTime = OutputTime + OutputDT
     PNTCNT=PNTCNT+1
          IF (OutputTime > EndInputTime+TIMEPAST) EXIT TimeLoop    !msf
   END DO TimeLoop
!                    Close files & exit
    CLOSE(2,STATUS='KEEP')
    OPEN(2,FILE=OutputFileName,STATUS='OLD',POSITION='REWIND')
! This section of code was added to create a trajectory file in the 7 column format
! Trajectory file format is:
! 1st line = total number of lines (delete for plotting, etc.)
! Columns are:
```

```
! 1 - time (s)
! 2-4 - ENU position (m)
! 5-7 - ENU velocity (m/s)
    WRITE(4,*) PNTCNT
    IF (NORDER.EQ.3) THEN
     DO jj=1,PNTCNT
      READ(2,'(F12.3,3X,3F12.3,3X,3F12.4,3X,3F12.4,3X,3F14.4)')
OutputTime,OutputXYZPosition,OutputXYZVelocity,OutputXYZAcceleration,     &
OutputXYZJerk
    WRITE(4,'(F12.3,3X,3F12.3,3X,3F12.4,3X,3F12.4)')                      &
OutputTime,OutputXYZPosition,OutputXYZVelocity
     ENDDO
    ELSEIF (NORDER.EQ.2) THEN
     DO jj=1,PNTCNT
      READ(2,'(F12.3,3X,3F12.3,3X,3F12.4,3X,3F12.4,3X,3F14.4)')
OutputTime,OutputXYZPosition,OutputXYZVelocity,OutputXYZAcceleration
      WRITE(4,'(F12.3,3X,3F12.3,3X,3F12.4,3X,3F12.4)')
OutputTime,OutputXYZPosition,OutputXYZVelocity
     ENDDO
    ENDIF
  CLOSE(1,STATUS='KEEP')
  CLOSE(3,STATUS='KEEP')
```

```
CONTAINS

  SUBROUTINE InitializeFilter
    NumberValid = 0
    FilterWindowCount = 0
    StackPointer = 0
    IExtrap = 0
    FilterWindowStartTime = 0.0D0
    FilterWindowEndTime   = 0.0D0
    TimeBuffer = 0.0D0
    PosBufferX = 0.0D0
    PosBufferY = 0.0D0
    PosBufferZ = 0.0D0
    DTMatrix = 0.0D0
    FilterCoeffsX = (/ 0.0D0,0.0D0,0.0D0,0.0D0,0.0D0 /)
    FilterCoeffsY = (/ 0.0D0,0.0D0,0.0D0,0.0D0,0.0D0 /)
    FilterCoeffsZ = (/ 0.0D0,0.0D0,0.0D0,0.0D0,0.0D0 /)
  END SUBROUTINE InitializeFilter
  SUBROUTINE StartFilter
    INTEGER(KIND=4) :: NumberPoints
!               Clear past values
    LastTime = 0.0D0
    LastMeasuredXYZPosition = 0.0D0
    LastMeasuredXYZVelocity = 0.0D0
```

```
      LastMeasuredXYZAcceleration= 0.0D0
       LastMeasuredXYZJerk= 0.0D0
!                Puts  points into filter
      DO NumberPoints = 1,5
        CALL InputMeasurement
      END DO
!                Load first buffer points for vel, acc

      VelBufferXYZ(1:3,1) = VelBufferXYZ(1:3,2)
      AccBufferXYZ(1:3,1) = AccBufferXYZ(1:3,3)
     AccBufferXYZ(1:3,2) = AccBufferXYZ(1:3,3)
      JrkBufferXYZ(1:3,1) = JrkBufferXYZ(1:3,4)
      JrkBufferXYZ(1:3,2) = JrkBufferXYZ(1:3,4)
      JrkBufferXYZ(1:3,3) = JrkBufferXYZ(1:3,4)
     SnapBufferXYZ(1:3,1) = SnapBufferXYZ(1:3,5)
      SnapBufferXYZ(1:3,2) = SnapBufferXYZ(1:3,5)
      SnapBufferXYZ(1:3,3) = SnapBufferXYZ(1:3,5)
      SnapBufferXYZ(1:3,4) = SnapBufferXYZ(1:3,5)
!    Compute DT to table end (output from middle of table)
      DeltaTimeToTableEnd = 0.5D0 * DeltaInputTime * REAL(FilterWindowSize,8)
   END SUBROUTINE StartFilter
   SUBROUTINE InputMeasurement
     real zero3(3)
```

```
!               Read input data
    IF(.NOT.EndOfFile)THEN
     READ(1,*,END=99) InputTime,MeasuredXYZPosition(1:3),zero3,zero3
     GO TO 100
!               End of file condition
99    CONTINUE
     EndOfFile = .TRUE.
     EndInputTime = InputTime
100   CONTINUE
    ENDIF
!               Put measurement into filter

    CALL UpdateFilter (InputTime, MeasuredXYZPosition(1),                &
MeasuredXYZPosition(2), MeasuredXYZPosition(3))
!               Exit if end of input file reached
    IF (EndOfFile) RETURN
!               Record first time
    IF (FilterWindowCount == 1) StartInputTime = InputTime
!               Compute measured velocity,acceleration,jerk,and snap from input data

    IF (FilterWindowCount > 1) THEN
     DeltaInputTime = InputTime - LastTime
     IF (DeltaInputTime > 0.0D0) THEN
         MeasuredXYZVelocity  =  (MeasuredXYZPosition -                  &
```

```
LastMeasuredXYZPosition) / DeltaInputTime
        IF (FilterWindowCount > 2) &
          MeasuredXYZAcceleration = (MeasuredXYZVelocity -                &
LastMeasuredXYZVelocity) / DeltaInputTime
          MeasuredXYZJerk = (MeasuredXYZAcceleration-                     &
LastMeasuredXYZAcceleration) / DeltaInputTime
          MeasuredXYZSnap = (MeasuredXYZJerk-LastMeasuredXYZJerk) /       &
DeltaInputTime
        END IF
      END IF
!               Load states into buffer (stack)
   PosBufferXYZ(1:3,StackPointer) = MeasuredXYZPosition(1:3)
   VelBufferXYZ(1:3,StackPointer) = MeasuredXYZVelocity(1:3)
   AccBufferXYZ(1:3,StackPointer) = MeasuredXYZAcceleration(1:3)
   JrkBufferXYZ(1:3,StackPointer) = MeasuredXYZJerk(1:3)
   SnapBufferXYZ(1:3,StackPointer)= MeasuredXYZSnap(1:3)
!               Save last data
  LastTime = InputTime
   LastMeasuredXYZPosition     = MeasuredXYZPosition
   LastMeasuredXYZVelocity     = MeasuredXYZVelocity
   LastMeasuredXYZAcceleration = MeasuredXYZAcceleration
   LastMeasuredXYZJerk         = MeasuredXYZJerk
   DeltaTimeToTableEND =0.5D0 * (FilterWindowEndTime- FilterWindowStartTime)
```

```
!               Normal exit
    RETURN

  END SUBROUTINE InputMeasurement

  SUBROUTINE UpdateFilter(Time,MeasuredX,MeasuredY,MeasuredZ)
    REAL(KIND=8), INTENT(IN) :: Time,MeasuredX,MeasuredY,MeasuredZ
    INTEGER(KIND=4) :: WinIndex,WinPointer,StartPointer
    REAL(KIND=8) :: DeltaTime
    REAL(KIND=8), DIMENSION(MaxFilterWindowSize) :: LSX,LSY,LSZ

!               Update counter & window size
    IF(.NOT.EndOfFile)THEN
     NumberValid = NumberValid + 1
     FilterWindowCount = MIN(NumberValid,FilterWindowSize)
!               Load new measurement onto stack
     StackPointer = MOD(StackPointer,MaxFilterWindowSize) + 1
     TimeBuffer(StackPointer) = Time
     PosBufferX(StackPointer) = MeasuredX
     PosBufferY(StackPointer) = MeasuredY
     PosBufferZ(StackPointer) = MeasuredZ
    ENDIF
!               For Extrapolation cases ONLY (TIMEPAST>0),
!               when EOD detected at leading edge of window, expand window
(FilterWindowCount)
```

```
!                to

    IF(TIMEPAST.GT.0 .AND. EndOfFile)THEN

     IExtrap = IExtrap + 1

FilterWindowCount= MIN ( NumberValid, FilterWindowSize +                    &

MIN(FilterWindowSize/2, IExtrap)

       *(FlaredFilterWindowSize-FilterWindowSize)/(FilterWindowSize/2))

    ELSEIF(TIMEPAST.LE.0 .AND. EndOfFile)THEN

     RETURN

    ENDIF

!              Load proper data into sliding window

    StartPointer = StackPointer - FilterWindowCount + MaxFilterWindowSize

    DO WinIndex = 1,FilterWindowCount

      WinPointer = MOD(StartPointer + (WinIndex-1),MaxFilterWindowSize) + 1

      IF (WinIndex == 1)            FilterWindowStartTime = TimeBuffer(WinPointer)

      IF (WinIndex == FilterWindowCount) FilterWindowEndTime   =

TimeBuffer(WinPointer)

      DeltaTime = TimeBuffer(WinPointer) - FilterWindowStartTime

      LSX(WinIndex) = PosBufferX(WinPointer)

      LSY(WinIndex) = PosBufferY(WinPointer)

      LSZ(WinIndex) = PosBufferZ(WinPointer)

      DTMatrix(WinIndex,1) = 1.0D0

      DTMatrix(WinIndex,2) = DeltaTime
```

```
        DTMatrix(WinIndex,3) = DeltaTime**2
                DTMatrix(WinIndex,4) = DeltaTime**3
                DTMatrix(WinIndex,5) = DeltaTime**4
    END DO
!           Solve least squares problem
 IORDER=min(NORDER,FilterWindowCount-1)
    CALL QRDECOMP                                                    &
(DTMatrix,LSX,FilterCoeffsX,FilterWindowCount,MaxFilterWindowSize,IORDER)
    CALL QRDECOMP                                                    &
(DTMatrix,LSY,FilterCoeffsY,FilterWindowCount,MaxFilterWindowSize,IORDER)
    CALL QRDECOMP                                                    &
(DTMatrix,LSZ,FilterCoeffsZ,FilterWindowCount,MaxFilterWindowSize,IORDER)
  END SUBROUTINE UpdateFilter
  SUBROUTINE OutputFilterStates(Time)
    REAL(KIND=8), INTENT(IN) :: Time
    REAL(KIND=8) :: DeltaTime
!           Compute delta time from start of filter window
    DeltaTime = Time - FilterWindowStartTime
!           Current filter output states
! 2nd order
!  Position    = X + VdT +  AdT^2
!  Velocity    =    V   + 2AdT
!  Acceleration =          2A
```

! **3rd order**

! Position = $X + VdT + AdT^2 + JdT^3$

! Velocity = $V + 2AdT + 3JdT^2$

! Acceleration = $2A + 6JdT$

! Jerk = $6J$

! **4th order**

! Position = $X + VdT + AdT^2 + JdT^3 + SdT^4$

! Velocity = $V + 2AdT + 3JdT^2 + 4SdT^3$

! Acceleration = $2A + 6JdT + 12SdT^2$

! Jerk = $6J + 24SdT$

! Snap = $24S$

```
   FilteredX(POS) = FilterCoeffsX(POS) + DeltaTime*FilterCoeffsX(VEL) +       &
(DeltaTime**2)*FilterCoeffsX(ACC) + (DeltaTime**3)*FilterCoeffsX(JRK) +       &

(DeltaTime**4)*FilterCoeffsX(SNAP)

    FilteredY(POS) = FilterCoeffsY(POS) + DeltaTime* FilterCoeffsY(VEL) +     &
(DeltaTime**2)* FilterCoeffsY(ACC) + (DeltaTime**3)* FilterCoeffsY(JRK) +     &
(DeltaTime**4)* FilterCoeffsY(SNAP)

    FilteredZ(POS) = FilterCoeffsZ(POS) + DeltaTime* FilterCoeffsZ(VEL) +     &
(DeltaTime**2)* FilterCoeffsZ(ACC) + (DeltaTime**3)* FilterCoeffsZ(JRK) +     &
(DeltaTime**4)* FilterCoeffsZ(SNAP)

   FilteredX(VEL) = FilterCoeffsX(VEL) + 2.0D0*DeltaTime* FilterCoeffsX(ACC)+ &
3.0D0*(DeltaTime**2)* FilterCoeffsX(JRK) + 4.0D0*(DeltaTime**3)*              &
FilterCoeffsX(SNAP)
```

```
      FilteredY(VEL) = FilterCoeffsY(VEL)+2.0D0*DeltaTime* FilterCoeffsY(ACC) + &
3.0D0*(DeltaTime**2)* FilterCoeffsY(JRK) + 4.0D0*(DeltaTime**3)*          &
FilterCoeffsY(SNAP)
      FilteredZ(VEL) = FilterCoeffsZ(VEL)+ 2.0D0*DeltaTime* FilterCoeffsZ(ACC) + &
3.0D0*(DeltaTime**2)* FilterCoeffsZ(JRK) + 4.0D0*(DeltaTime**3)*          &
FilterCoeffsZ(SNAP)
      FilteredX(ACC) = 2.0D0*FilterCoeffsX(ACC)  + 6.0D0*DeltaTime*             &
FilterCoeffsX(JRK)+ 12.0D0*(DeltaTime**2)*FilterCoeffsX(SNAP)
      FilteredY(ACC) = 2.0D0*FilterCoeffsY(ACC)  + 6.0D0*DeltaTime*             &
FilterCoeffsY(JRK)+ 12.0D0*(DeltaTime**2)*FilterCoeffsY(SNAP)
      FilteredZ(ACC) = 2.0D0*FilterCoeffsZ(ACC)  + 6.0D0*DeltaTime*             &
FilterCoeffsZ(JRK) + 12.0D0*(DeltaTime**2)*FilterCoeffsZ(SNAP)
      FilteredX(JRK) = 6.0D0*FilterCoeffsX(JRK)+                                &
24.0D0*DeltaTime*FilterCoeffsX(SNAP)
      FilteredY(JRK) = 6.0D0*FilterCoeffsY(JRK) +                               &
24.0D0*DeltaTime*FilterCoeffsY(SNAP)
      FilteredZ(JRK) = 6.0D0*FilterCoeffsZ(JRK) +                               &
24.0D0*DeltaTime*FilterCoeffsZ(SNAP)
      FilteredX(SNAP) = 24.0D0*FilterCoeffsX(SNAP)
      FilteredY(SNAP) = 24.0D0*FilterCoeffsY(SNAP)
      FilteredZ(SNAP) = 24.0D0*FilterCoeffsZ(SNAP)
!                 Rotate to Radar XYZ
   OutputXYZPosition(1)  = FilteredX(POS)
```

```
OutputXYZPosition(2)  = FilteredY(POS)
OutputXYZPosition(3)  = FilteredZ(POS)
OutputXYZVelocity(1)  = FilteredX(VEL)
OutputXYZVelocity(2)  = FilteredY(VEL)
OutputXYZVelocity(3)  = FilteredZ(VEL)
OutputXYZAcceleration(1) = FilteredX(ACC)
OutputXYZAcceleration(2) = FilteredY(ACC)
OutputXYZAcceleration(3) = FilteredZ(ACC)
OutputXYZJerk(1) = FilteredX(JRK)
OutputXYZJerk(2) = FilteredY(JRK)
OutputXYZJerk(3) = FilteredZ(JRK)
OutputXYZSnap(1) = FilteredX(SNAP)
OutputXYZSnap(2) = FilteredY(SNAP)
OutputXYZSnap(3) = FilteredZ(SNAP)
END SUBROUTINE OutputFilterStates

SUBROUTINE OutputRawStates(Time)
  REAL(KIND=8), INTENT(IN) :: Time
  INTEGER(KIND=4) :: WinIndex,WinPointer,StartPointer
!               Find table index corresponding to input Time
  StartPointer = StackPointer - FilterWindowCount + MaxFilterWindowSize
  TimeLoop: DO WinIndex = 1,FilterWindowCount
    WinPointer = MOD(StartPointer + (WinIndex-1),MaxFilterWindowSize) + 1
```

```
      IF (TimeBuffer(WinPointer) > Time) THEN
        IF (WinIndex > 1) WinPointer = MOD(StartPointer + (WinIndex-          &
2),MaxFilterWindowSize) + 1
        EXIT TimeLoop
      END IF
    END DO TimeLoop
!             Extrapolate states to output time
    DeltaTime = Time - TimeBuffer(WinPointer)
        IF (FilterWindowCount == 1) StartInputTime = InputTime
! TO STILL HAVE THE CAPABILITY OF STILL HAVING A 2ND ORDER LEAST
! SQUARES FIT THIS WAS ADDED SO DATA WOULD MATCH THE PREVIOUS
!VERSION
MeasuredXYZPosition(1:3) = PosBufferXYZ(1:3,WinPointer) + DeltaTime *      &
VelBufferXYZ(1:3,WinPointer) +                                             &
(0.5D0*DeltaTime**2)*AccBufferXYZ(1:3,WinPointer)
MeasuredXYZVelocity(1:3) = VelBufferXYZ(1:3,WinPointer)+DeltaTime *        &
AccBufferXYZ(1:3,WinPointer)
MeasuredXYZAcceleration(1:3) = AccBufferXYZ(1:3,WinPointer)
IF(NORDER == 3)THEN
 MeasuredXYZPosition(1:3) = PosBufferXYZ(1:3,WinPointer) + DeltaTime *     &
VelBufferXYZ(1:3,WinPointer) +  (0.5D0*DeltaTime**2)*                      &
AccBufferXYZ(1:3,WinPointer)+                                              &
JrkBufferXYZ(1:3,WinPointer)*(DeltaTime**3)/6.0
MeasuredXYZVelocity(1:3) = VelBufferXYZ(1:3,WinPointer)+DeltaTime*         &
```

```
AccBufferXYZ(1:3,WinPointer) +                                    &
JrkBufferXYZ(1:3,WinPointer)*(DeltaTime*DeltaTime)/2.0
 MeasuredXYZAcceleration(1:3) = AccBufferXYZ(1:3,WinPointer)+     &
JrkBufferXYZ(1:3,WinPointer)*DeltaTime
MeasuredXYZJerk(1:3) = JrkBufferXYZ(1:3,WinPointer)               &
ENDIF
IF(NORDER == 4)THEN
MeasuredXYZPosition(1:3) = PosBufferXYZ(1:3,WinPointer) + DeltaTime *  &
VelBufferXYZ(1:3,WinPointer) +                                    &
(0.5D0*DeltaTime**2)*AccBufferXYZ(1:3,WinPointer) +               &
JrkBufferXYZ(1:3,WinPointer)*(DeltaTime**3)/6.0 +                 &
SnapBufferXYZ(1:3,WinPointer)*(DeltaTime**4)/24.0
MeasuredXYZVelocity(1:3)= VelBufferXYZ(1:3,WinPointer)+DeltaTime  &
*AccBufferXYZ(1:3,WinPointer) +                                   &
JrkBufferXYZ(1:3,WinPointer)*(DeltaTime*DeltaTime)/2.0 +          &
SnapBufferXYZ(1:3,WinPointer)*(DeltaTime**3)/6.0
MeasuredXYZAcceleration(1:3) = AccBufferXYZ(1:3,WinPointer) +     &
JrkBufferXYZ(1:3,WinPointer)*DeltaTime +                          &
SnapBufferXYZ(1:3,WinPointer)*(DeltaTime*DeltaTime)/2.0
MeasuredXYZJerk(1:3)= JrkBufferXYZ(1:3,WinPointer)+               &
SnapBufferXYZ(1:3,WinPointer) *DeltaTime
MeasuredXYZSnap(1:3) = SnapBufferXYZ(1:3,WinPointer)
ENDIF
```

```
    END SUBROUTINE OutputRawStates

END PROGRAM LSSWFilter_Cartesian
```

APPENDIX C
QR DECOMPOSITION SUBROUTINE

The following is a subroutine that solves the Least Square problem (AX = B) using QR Decomposition.

```
**********************************************************************
* Function -- Matrix utilities required for least squares filter     *
* Author   -- John Wells  - Nichols Research  (256) 955-7287         *
*                                                                    *
*          Modified by Melissa Farbman 4-7-2005                      *
*                                                                    *
*          to be of gereric order                                    *
*                                                                    *
**********************************************************************

MODULE QRDecomposition

  IMPLICIT NONE
  PRIVATE
  PUBLIC :: QRDECOMP

CONTAINS

  SUBROUTINE QRDECOMP(A,B,X,M,MMAX,NORDER)
!
! Solve 2nd Order Least Squares Problem (AX = B) using QR Decomposition
! via Reflectors
```

```
!
    INTEGER(KIND=4), INTENT(IN) :: MMAX
    INTEGER(KIND=4), INTENT(IN) :: M
    INTEGER(KIND=4), INTENT(IN) :: NORDER
    REAL(KIND=8), DIMENSION(MMAX,NORDER+1), INTENT(IN) :: A
    REAL(KIND=8), DIMENSION(MMAX), INTENT(IN) :: B
    REAL(KIND=8), DIMENSION(NORDER+1), INTENT(OUT) :: X
    INTEGER(KIND=4) :: I,J,K
        REAL(KIND=8) ::
.
R(MMAX,NORDER+1),BPRIME(MMAX),U(MMAX),MU,SIGMA,GAMMA,BETA,
ROWSUM
!
! Load A matrix into R (will be reduced to proper R in algorithm)
! Load B vector into B' (reduced to proper B' in algorithm)
! Least Squares sol'n is sol'n of Rx = B'
!
    DO I = 1,M
      BPRIME(I) = B(I)
      DO J = 1,NORDER+1
        R(I,J) = A(I,J)
      ENDDO
    ENDDO
!
! Main Algorithm
```

```
!
      DO I = 1,NORDER+1
!
! Compute the Ith Reflector & Apply it to the Ith Column of the Matrix
!
        MU = ABS(R(I,I))
        DO J = I+1,M
          IF (ABS(R(J,I)) .GT. MU) MU = ABS(R(J,I))
        ENDDO
        DO J = I,M
          R(J,I) = R(J,I)/MU
        ENDDO
        SIGMA = 0.0D0
        DO J = I,M
            SIGMA = SIGMA + R(J,I)*R(J,I)
        ENDDO
        SIGMA = SIGN(1.0D0,R(I,I))*SQRT(SIGMA)
        U(I) = R(I,I) + SIGMA
        DO J = I+1,M
          U(J) = R(J,I)
        ENDDO
        GAMMA = 1.0D0/(SIGMA*U(I))
        SIGMA = SIGMA*MU
```

```
      R(I,I) = -SIGMA
!
! Apply the Ith Reflector to the Other Columns of the Matrix
!
      DO J = I+1,NORDER+1
        BETA = 0.0D0
        DO K = I,M
          BETA = BETA + U(K)*R(K,J)
        ENDDO
        BETA = GAMMA*BETA
        DO K = I,M
          R(K,J) = R(K,J) - BETA*U(K)
        ENDDO
      ENDDO
!
! Apply the Ith Reflector to the Measurement Vector
!
     BETA = 0.0D0
    DO J = I,M
      BETA = BETA + U(J)*BPRIME(J)
    ENDDO
     BETA = GAMMA*BETA
     DO J = I,M
```

```
        BPRIME(J) = BPRIME(J) - BETA*U(J)
      ENDDO
    ENDDO
!
! Now BackSolve RX = B'
!
    X(NORDER+1) = BPRIME(NORDER+1)/R(NORDER+1,NORDER+1)
    DO I = NORDER,1,-1
          ROWSUM = 0.0d0
      DO J = I+1,NORDER+1
        ROWSUM = ROWSUM + R(I,J)*X(J)
      ENDDO
      X(I) = (BPRIME(I) - ROWSUM)/R(I,I)
    ENDDO
!
  END SUBROUTINE QRDECOMP
END MODULE QRDecomposition
```

APPENDIX D
SINE GENERATION ROUTINE

The following program was developed to generate a sine wave for the analysis.

```
      implicit none

      real sigma1,xdata(10),rann
      real freq,amp,dt,time,x,y,pi
      integer itime
      integer i
      character filename*80

      data pi/3.14159265/


      open(unit=20,file='output.dat',status='unknown')

      write(*,*)'enter frequency (hz) (i.e. 2)'
      read(*,*)freq
      write(*,*)'enter amplitude (i.e. 100)'
      read(*,*)amp
      write(*,*)'enter dt (sec) (i.e. .1 )'
      read(*,*)dt

      itime=0

1000  continue
        itime=itime+1
        if(itime.gt.1025)go to 999
        time=itime*dt
        x=amp*cos(2.*pi*freq*time)
        y=0.
        y=amp*sin(2.*pi*freq*time)
        write(20,3)time,y,0,0,0,0,0,0,0,0
3     format(10e14.5)
      go to 1000
999   continue

      close(unit=20,status='keep')

      end
```

```
      real FUNCTION RANF()
      COMMON/SEEDS/OLDSEED
      INTEGER OLDSEED
      INTEGER A,P,IX,B15,B16,XHI,XALO,LEFTLO,FHI,K
      DATA A/16807/,B15/32768/,B16/65536/,P/2147483647/
      IF (OLDSEED .EQ. 0) OLDSEED = 123451
      IX=OLDSEED
      XHI=IX/B16
      XALO=(IX-XHI*B16)*A
      LEFTLO=XALO/B16
      FHI=XHI*A+LEFTLO
      K=FHI/B15
      IX=(((XALO-LEFTLO*B16)-P)+(FHI-K*B15)*B16)+K
      IF(IX.LT.0)IX=IX+P
      RANF=FLOAT(IX)*4.656612875E-10
      OLDSEED=IX
      RETURN
      END


      REAL FUNCTION RANN(MEAN,SIGMA)
      REAL MEAN,SIGMA
      INTEGER I
      REAL SUM,RANF
      SUM=0.0
      DO 10 I=1,12
10    SUM=SUM+RANF()
      RANN=(SUM-6.)*SIGMA+MEAN
      RETURN
      END
```

APPENDIX E
NOISE ADDITION ROUTINE

The following program was developed to provide a more realistic analysis by adding measurement noise.

```
 PROGRAM addnoise
     IMPLICIT NONE
     REAL sigma1,xdata(10),rann
     INTEGER  i,itime
     CHARACTER filename*80
    open(unit=20,file='output.dat',status='unknown')
     write(*,*)'enter 1 sigma'
     read(*,*)sigma1
1000  continue
     itime=itime+1
         if (itime.gt.512) then
         goto 999
         endif
          xdata(1)=itime
          do i=1,3
           xdata(1+i)=rann(0.,sigma1) !msf
           enddo
       write(20,3)xdata
3     format(f12.3,3x,3f13.2,3x,3f13.3,3x,3f13.3)
     go to 1000
```

```
999   continue
      close(unit=10,status='keep')
      close(unit=20,status='keep')
      end


      REAL FUNCTION RANF()
      COMMON/SEEDS/OLDSEED
      INTEGER OLDSEED
      INTEGER A,P,IX,B15,B16,XHI,XALO,LEFTLO,FHI,K
      DATA A/16807/,B15/32768/,B16/65536/,P/2147483647/
      IF (OLDSEED .EQ. 0) OLDSEED = 123451
      IX=OLDSEED
      XHI=IX/B16
      XALO=(IX-XHI*B16)*A
      LEFTLO=XALO/B16
      FHI=XHI*A+LEFTLO
      K=FHI/B15
      IX=(((XALO-LEFTLO*B16)-P)+(FHI-K*B15)*B16)+K
      IF(IX.LT.0)IX=IX+P
      RANF=FLOAT(IX)*4.656612875E-10
      OLDSEED=IX
      RETURN
      END
```

```
      REAL FUNCTION RANN(MEAN,SIGMA)
      REAL MEAN,SIGMA
      INTEGER I
      REAL SUM,RANF
      SUM=0.0
      DO 10 I=1,12
10    SUM=SUM+RANF()
      RANN=(SUM-6.)*SIGMA+MEAN
      RETURN
      END
```

REFERENCES

[1] Hobbs, David, and Bohn, Preben, "Precise Orbit Determination for Low Earth Orbit Satellites," *MCFA Annals,* Vol. 4, 2005.

[2] Liu, C. Y., and Lee, S. C., "Trajectory Estimation of Reentry Vehicles by Use of On-Line Input Estimator," *Journal Of Guidance, Control, and Dynamics,* Vol. 22, No. 6, Nov.–Dec. 1999, pp 808-115.

[3] Karlgaard, Christopher D., Tartabini, Paul V., Blanchard, Robert C., Kirsch, Michael, and Toniolo, Matthew D., "Hyper-X Post-Flight-Trajectory Reconstruction," *Journal of Spacecraft and Rockets,* 2006, 0022-4650, Vol. 43, No.1, pp 105-115.

[4] Napolitano, M. R., Windon, D. A., Casanova, J. L., Innocenti, M., and Silvestri, G., "Kalman Filters and Neural-Network Schemes for Sensor Validation in Flight Control Systems," *IEEE Transactions on Control Systems Technology*, 1998.

[5] Skaloud, J., "Reliability in Direct Georeferencing: An Overview of the Current Approaches and Possibilities," ISPRS-EuroSDR Workshop EuroCOW, Castelldelfels, 2006.

[6] Gelb, Arthur, *Applied Optimal Estimation*, The M.I.T. Press, 1999.

[7] Wells, John, Jackson, Kevin, Jordan, John, and Hill, Joe, "DGPS Target State Filter-- An Innovative Design," RDECOM AMRDEC, Oct. 2004.

[8] Berkey, D. D., *Calculus,* 2nd ed., Saunders College Publishing, New York, 1988.

[9] Zwillinger, Daniel, *CRC Standard Mathematical Tables*, 23rd ed., Cleveland, OH, CRC Press, 1975.

[10] Bajpai, A. C., *Numerical Methods for Engineers and Scientists*, Taylor and Francis Ltd., London, 1975.

[11] Hamming, R. W., *Numerical Methods for Scientists and Engineers*, 2nd ed., Dover Publications, Inc., New York, 1973.

[12] Watkins, David S., *Fundamentals of Matrix Computations*, John Wiley & Sons, Cambridge, MA, 1991.

[13] Epperson, James F., "Numerical Methods for Linear Algebra," Department of Mathematical Sciences, UAH, Huntsville, AL, 1995.