1-1-2002

Self-Smoothing Functional Estimation

Bronson Thomas Yake

SELF-SMOOTHING FUNCTIONAL ESTIMATION

By

Bronson Thomas Yake

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Aerospace Engineering
in the Department of Aerospace Engineering

Mississippi State, Mississippi

December 2002

SELF-SMOOTHING FUNCTIONAL ESTIMATION

By

Bronson Thomas Yake

Approved:

Name: Bronson Thomas Yake

Date of Degree: December 13, 2002

Institution: Mississippi State University

Major Field: Aerospace Engineering

Major Professor: Dr. Robert L. King

Title of Study: SELF-SMOOTHING FUNCTIONAL ESTIMATION

Pages in Study: 106

Analysis of measured data is often required when there is no deep understanding of the mathematics that accurately describes the process being measured. Additionally, realistic estimation of the derivative of measured data is often useful. Current techniques of accomplishing this type of data analysis are labor intensive, prone to significant error, and highly dependent on the expertise of the engineer performing the analysis. The “Self-Smoothing Functional Estimation” (SSFE) algorithm was developed to automate the analysis of measured data and to provide a reliable basis for the extraction of derivative information. In addition to the mathematical development of the SSFE algorithm, an example is included in Chapter III that illustrates several of the innovative features of the SSFE and associated algorithms. Conclusions are drawn about the usefulness of the algorithm from an engineering perspective and additional possible uses are mentioned.

Acknowledgements

I would like to acknowledge the contributions of my friends, family and colleagues in the development of the SSFE algorithm. I owe a special thanks to Warren Boord for trusting me with the task that motivated the development of the SSFE and the freedom to develop a new solution from my own perspective, and to Dr. Caroll Nunn for his assistance in the development of the closed-form integration routine used in conjunction with the SSFE algorithm.

I would like to thank my committee at Mississippi State University, Dr. King, Dr. McWhorter, and Dr. Bridges, your encouragement and assistance has made it possible for me to, at long last, finish this road to a Master's degree. Most of all, however, I want to thank my wife, Debbie, your not-always-so-subtle prompting has kept me thinking of the goal (getting finished), and your writing expertise has helped immeasurably in editing this thesis.

To you all, and everyone else who has been there to bat ideas around with me, or taught me something new, or just helped me focus, thanks.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The Motivating Problem

A common type of problem in engineering disciplines is the analysis of sampled data. Often the mathematical (or functional) equations that coherently describe the data would be far more useful to the engineer performing the analysis than the specific data recorded. In these instances, parameters are measured only in part for the explicit information they provide. The implied information is often just as useful as the measured data. For example, an automotive engineer may record an automobile's velocity and throttle position, RPM, etc. All of these parameters are valuable to understanding the nature of the vehicle's performance, but when properly correlated, they may imply information about the vehicle's internal performance that is not initially apparent. A proper correlation may show that the induction and or exhaust systems are sub-optimized at a particularly critical engine speed for total vehicle performance. This information would be difficult to measure directly and probably has such a complex relationship with several factors making a sufficiently accurate a-priori prediction of the true performance nearly or completely impossible to obtain.

The Specific Problem

It was precisely this class of problem that necessitated the development of the SSFE algorithm. The author was presented with images of recorded data corresponding to plots of Altitude-with-Time, Horizontal Velocity-with-Time, and Cross Range Position-with-Down Range Position data. The images showed significant measurement noise, and only one point in position and time were positively correlated between the images. The primary analysis task was to reconstruct the three-dimensional trajectory of the flight vehicle. The source of the data makes it impossible to elaborate on the specifics of the information.

Currently Existing Solution Techniques

The simplest and most readily available solution was to make piecewise linear (or at most second order) approximations of the data. Selecting the data ranges for each piecewise section was purely engineering art. The accuracy of the analysis depended entirely on the engineer's selection of data ranges. Implied values such as derivations and integrations of the data were crude approximations at best. To attempt a three-dimensional reconstruction of the data using this method would have been painfully slow and so full of error as to be useless for anything more than trend and possibly maximum and minimum information. What complicated the matter beyond hope was that not all three dimensions of trajectory information were provided with respect to time; therefore it was not even possible to choose the same independent variable values for correlation.

Another solution technique that may be and has been used in this class of problem is the cubic spline[1, 2]. The cubic spline technique is an interpolation routine that forces exact fits at each data point. Cubic splines provide cubic polynomials for the purposes of interpolation between each pair of adjacent data points. To guarantee smoothness, the first and second derivatives of adjacent functions are forced to be equal at shared data point (called a knot). The use of cubic splines solves the problem of automation and smoothness. However, it introduces other problems. Because the spline forces exact fits at each data point it assumes that the data exactly represents the measured quantity. Applying a cubic spline to a set of measured data that is dense ("dense" is loosely defined to mean "that the number of points, in any sub-region, is more than an order of magnitude larger than the number of inflection points expected in the fitted curve for that sub-region, and that there are no 'abrupt' changes in the expected second derivative"[2]) may exaggerate the influence of any measurement noise that is present.

Necessity: The Mother of SSFE

As with virtually every real engineering problem, the trajectory analysis was not performed in a vacuum. Other engineers performing other types of analyses, for example radar cross-section prediction, required the updated trajectory information as an input each time the cross-section was calculated. This fact meant that a quick best guess analysis based only on the points that were manually selected from the original images would not be adequate.

As noted previously, the required result of the trajectory analysis was a three-dimensional reconstruction of the trajectory from the images provided. There were however additional constraints. The analysis itself needed to be automated both to reduce the time required to perform the analysis and to eliminate, as much as possible, the influence of engineering art in the final results of the analysis. Another additional constraint was that the analysis needed to produce functional estimations of not only the specific data from the images but also realistic estimations of the implied data (derivation and integration values) at each point. This constraint existed because not all of the desired data was in a readily usable form; therefore it was necessary to extract the implied data from the images to be able to recreate a reasonably accurate three-dimensional trajectory. This also led to an interesting and distinctive secondary requirement: the analysis routine needed to allow the dependent variable (and the functional estimation from which it is calculated) to be re-calculated for any independent variable value within the valid range of data points.

Low-order (first and second order) polynomial estimations (curve-fits) possessed many desirable characteristics, especially that their behavior is predictable between known points. The simple technique of creating a series of first or second order piecewise estimations allowed for the re-calculation of the independent variable based on the functional estimation inside each piecewise segment. However, it did not necessarily provide realistic derivative values. This technique invariably resulted in discontinuities of the derivatives at the transition from one piecewise segment to the next. Of course, the problem of dependence on engineering art was significant.

Early in the development of the algorithm, the desire to automate the process and remove the question of the engineer's experience from the process led to the idea of using a small set of adjacent points (a window), performing a least-spares curve fit on the points within that window, and then moving to the next set of points. This automated piecewise process would be repeated throughout the data set. There were three problems with this modified piecewise process that made it impractical. First, the data set had to have the proper number of points (a multiple window size) for this to be easily automated. Second, this process provided an arbitrarily large number of transition points, and therefore points of discontinuity. Third, automating the selection of segments provided more repeatable results, but these results were not necessarily more accurate than those from manual segment selection, and often less desirable.

The next step in the conceptual development of the SSFE algorithm was to create a polynomial curve-fit for each point with a window of some set size at each point. The set of points in the last complete window would have been used for each point in it. This retained the level of automation and eliminated the need for a specific number of data points (i.e. an integer multiple of the window size). However, the increased number of polynomial estimations compounds the problem of the large number of discontinuities.

The cubic spline technique was not attempted because hitting the points exactly was not the goal. A reliable approximation of the original data was desired more than a precise match at the selected points. Sufficiently careful selection of points would have allowed the cubic spline method to work well for the given data, but since elimination of the need for dependence on engineering art was desired, a least-squares (optimal) approximation[2, 3, 4] for the data was selected.

The distinguishing feature of the SSFE algorithm was developed at this point. A relationship between the polynomial functional estimations at neighboring points was required. If a relationship existed between the functional estimations of neighboring points then the derivatives at those points would show smoother, more realistic transitions. The relationship chosen was that of averaged coefficients. Chapter II.2 details the mathematical background of how the coefficient averaging is accomplished and Appendix A.3 is the MATLAB implementation of the algorithm.

The SSFE algorithm uses a relatively small moving window of points to keep the fidelity of the curve-fits high within the window. Although the computation of averaged coefficients made the algorithm more computationally intensive than simple least-squares polynomial curve-fit, it was still relatively simple to automate. The SSFE algorithm itself or in conjunction with auxiliary routines was able to meet all analysis requirements.

One of the most significant secondary requirements of the analysis was to be able to reproduce the data, or any sub-set of it with respect to an arbitrary independent variable vector, that is, to correlate explicit and implicit data (see Chapter II.3.1). This was important for display purposes and vital for the fundamental task of trajectory reconstruction because much of the required trajectory information was not provided explicitly. Re-creation of the three-dimensional trajectory required correlation of the positional data with respect to a common time vector so that each point in time would be coordinated with all three positional coordinates (down-range, cross-range, and altitude). The linear integration (see Chapter II.3.4) of the Down-Range with Cross-Range data was correlated with the time integration of the Velocity data. From that, the Down-Range and Cross-Range values could be correlated with the time vector from the time integration of horizontal velocity.

Auxiliary Routines

The SSFE algorithm produces a functional estimation of measured data. One of the primary objectives of the SSFE algorithm is to model measured data sufficiently so that derivative and integral information based on the model closely approximate the derivative and integral information of the true performance of the measured quantity. The algorithms and equations used to extract the implied (derivative and integral) data are discussed in Chapter II.3.

CHAPTER II

MATHEMATICAL DEVELOPMENT

MATLAB building blocks

The SSFE algorithm was implemented in MATLAB script. MATLAB offers a number of advantages in solving scientific and engineering problems. One of the important features of MATLAB is that many mathematical tools are provided as efficient pre-validated functions. The least-squares polynomial curve-fit provided in MATLAB as the function “polyfit” is the most complex and important building block used in the realization of the SSFE algorithm.

The fundamental concept of least-squares optimization routines is to square the difference between the value predicted by a polynomial function at a point and the measured or actual value at that point. This difference-squared is a gauge of the accuracy of a function as compared with the measured data. The fact that the difference is squared removes the question of whether the function’s predicted result is greater or less than the measured data. The coefficients of the function are then optimized to minimize the function’s absolute squared error[1, 2, 3, 5, 6].

Algorithm development

Based on the least-squares polynomial estimation building block, the SSFE algorithm uses a small-window-at-a-time approach to attempt to achieve high fidelity at each point. The selection of the appropriately sized small window is left to engineering judgment. The window size has to be sufficiently large so that the measurement error is relatively small compared to the range of the independent variable in the window. The goal of the SSFE algorithm is to estimate the trajectory of the data at each point. If the relative size of the noise as compared with the range of the window is sufficiently large, the SSFE will perform poorly at estimating the overall data trend.

The small data window is moved incrementally so that each point has multiple polynomial estimations. (Note that points at the beginning and terminating ends of the data do not have multiple curve-fits that estimate them.) To guarantee that the functional estimation of the data, which generally changes at each point, does not change drastically or in an un-realistic fashion from point to point, the coefficients of the final functional estimation at each point are created by averaging the coefficients of each incremental polynomial estimation created using that point.

The following equations show that the estimated value resulting from a composite polynomial with averaged coefficients at a point is mathematically identical to the average of the values predicted by each polynomial estimation that touches the point of interest. For the purpose of illustration, a four data-point window is utilized in conjunction with a second order polynomial estimation for each window.

$$y_{ax} = a_2 x^2 + a_1 x + a_0 \approx y_x$$

Equation 1

$$y_{bx} = b_2 x^2 + b_1 x + b_0 \approx y_x$$

Equation 2

$$y_{cx} = c_2 x^2 + c_1 x + c_0 \approx y_x$$

Equation 3

$$y_{dx} = d_2 x^2 + d_1 x + d_0 \approx y_x$$

Equation 4

$$y_{ax} = y_{a2x} + y_{a1x} + y_{a0}$$

Equation 5

$$y_{bx} = y_{b2x} + y_{b1x} + y_{b0}$$

Equation 6

$$y_{cx} = y_{c2x} + y_{c1x} + y_{c0}$$

Equation 7

$$y_{dx} = y_{d2x} + y_{d1x} x + y_{d0}$$

Equation 8

$$y_x \approx \frac{y_{ax} + y_{bx} + y_{cx} + y_{dx}}{4} = \frac{y_{a2x} + y_{b2x} + y_{c2x} + y_{d2x}}{4} + \frac{y_{a1x} + y_{b1x} + y_{c1x} + y_{d1x}}{4} + \ldots$$

$$\frac{y_{a0} + y_{b0} + y_{c0} + y_{d0}}{4}$$

Equation 9

$$y_x \approx \left(\frac{a_2 + b_2 + c_2 + d_2}{4}\right)x^2 + \left(\frac{a_1 + b_1 + c_1 + d_1}{4}\right)x + \left(\frac{a_0 + b_0 + c_0 + d_0}{4}\right) = ax^2 + bx + g$$

Equation 10

This method not only provides a smoothed estimate of the measured data, but each point "x" has a corresponding polynomial function that is related to both the values and estimated functions of other points in its vicinity. This process is repeated for each point of the measured data set. The result is a vector of functions with the same length as the original data set. For very large sets of data this method is initially memory intensive; however, with the availability of memory and computing ability, this is no longer a significant objection. The general form of the SSFE algorithm can be expressed as:

$$\begin{aligned}
&f_1(x_i \to x_{i+n}) = a_1(m)x^m + a_1(m-1)x^{m+1} + \cdots + a_1(1)x + a_1(0)\\
&f_2(x_{i-1} \to x_{i+n-1}) = a_2(m)x^m + a_2(m-1)x^{m+1} + \cdots + a_2(1)x + a_2(0)\\
&\vdots\\
&f_n(x_{i-n} \to x_i) = a_n(m)x^m + a_n(m-1)x^{m+1} + \cdots + a_n(1)x + a_n(0)
\end{aligned}$$

Equation 11

$$\begin{aligned}
y_i \approx &\left[\frac{a_1(m) + a_2(m) + \cdots + a_n(m)}{n}\right]x^m + \cdots + \left[\frac{a_1(1) + a_2(1) + \cdots + a_n(1)}{n}\right]x\\
&+ \left[\frac{a_1(0) + a_2(0) + \cdots + a_n(0)}{n}\right]
\end{aligned}$$

Equation 12

where “n” is the number of points in the window and “m” is the order curve fit. The algorithm yields a polynomial representation of the data at the point that has a defined mathematical relationship to the representations at its neighboring points. (The relationship extends to points n-1 places before and after each point, but is strongest between immediately neighboring points.)

$$y_i \approx a_i(m)x_i^{\,m} + \cdots + a_i(1)x_i + a_i(0)$$

Equation 13

Larger windows, in general, provide a greater filtering effect and would be preferable for systems where higher order functional estimations are desired, such as when it is known that a higher order function theoretically describes the measured data. At a minimum one more point is required per order of the functional estimation.

Additional Features

The SSFE algorithm makes accomplishing many other important analytical tasks simpler. Four of the more noteworthy tasks - correlation and re-plotting, derivation, closed-form integration, and linear integration - are discussed in this section. Almost any analytical operation is simplified when a functional estimation of the measured data is available, so the four tasks mentioned in this section are by no means a complete list of additional analytical tasks that are aided by the SSFE algorithm.

Correlation and Re-plotting

The example of Chapter III will illustrate how correlation and re-plotting can be used in solving real-world engineering problems. Here the mechanics of accomplishing correlation and re-plotting are discussed.

$$Y_{(x_i)} \approx f(x_i) = a_i(m)x_i^{\,m} + \cdots + a_i(1)x_i + a_i(0)$$

Equation 14

Equation 14 yields an estimation of each Y value at each X value based on the corresponding coefficients found via the polynomial curve-fit and averaging routines discussed above. When a correlation is desired, a new vector of X values is chosen or provided for which a corresponding vector of Y values is required. The coefficients $\alpha_i()$ are interpolated based on the new X vector compared to the originally measured X vector. These new coefficients are then used in conjunction with the new X vector to produce a new vector of estimated Y values that are now correlated with the new X vector.

Re-plotting is simply an extension of the correlation process. The example of Chapter III utilizes the correlation process to allow the re-plotting of data versus a different independent variable. If two dependent variables have the same independent variable but different independent variable vectors, a new common independent variable vector can be chosen thus allowing the re-plotting of the data with the dependent variables versus each other with one acting as a new independent variable. This is especially useful analysis of time and space (trajectory) data.

Derivatives

In general, it is not advisable to take derivatives of measured data. However, real-world analysis often requires derivative information. For example, “altitude” may be measured, but for a given analysis altitude rate information may be required. Traditional analysis techniques require a significant level of engineering art to select a representative slope at a given point.

Since the SSFE algorithm produces a unique polynomial functional estimation of the data at each point, it is a relatively simple task to take a derivative at each point. This is especially useful since each polynomial is related to the polynomials at the points that are near it. Therefore its derivative is also related to the derivatives at the points near it. The general form is shown below:

$$\dot{Y}_{(x_i)} \approx \dot{f}(x_i) = m\boldsymbol{a}_{i(m)} x_i^{\,m-1} + \cdots + 2\boldsymbol{a}_{i(2)} x_i + \boldsymbol{a}_{i(1)}$$

Equation 15

In addition to the first derivative form shown above, higher order derivatives can be obtained in the same way. Another method to obtain higher-order derivative information is to take the first derivative as shown above and then re-run the SSFE algorithm on the derivative values with respect to the independent variable. This higher order SSFE derivative technique can be repeated as many times as desired, but engineering judgment still has a place in determining the order of derivative information that is trustworthy based on the measured data.

Closed-form Integration

As a rule, integration of noisy data is more reliable than derivative information of that same data. For measured data, numerical integration is typically utilized. The advantage is that numerical integration is conceptually simple and relatively quick. However, numerical integration does have the drawback of being an approximation. The measurement and numerical errors reduce the reliability of the results.

Because the SSFE algorithm produces vectors of linearly related coefficients for the functional estimation of the measured data, an exact closed form integration is possible. For the purposes of illustration, a second order polynomial is used below.

$$\int_a^b Y_x dx \approx \int_a^b f(x)dx = \int_a^b \left(a_x x^2 + b_x x + g_x\right)dx$$

Equation 16

As a result of the averaging relationship that the coefficients share, the values of the coefficients anywhere between two known points can be expressed in the following manner:

$$y_{x=a} \approx a_a x^2 + b_a x + g_a$$

Equation 17

$$y_{x=b} \approx a_b x^2 + b_b x + g_b$$

Equation 18

$$a_{a\le x\le b} = \frac{a_b - a_a}{b-a}(x-a) + a_a = \frac{\Delta a}{\Delta x}(x-a) + a_a$$

Equation 19

$$b_{a\le x\le b} = \frac{b_b - b_a}{b-a}(x-a) + b_a = \frac{\Delta b}{\Delta x}(x-a) + b_a$$

Equation 20

$$g_{a\le x\le b} = \frac{g_b - g_a}{b-a}(x-a) + g_a = \frac{\Delta g}{\Delta x}(x-a) + g_a$$

Equation 21

The fact that the coefficients have a linear relationship to the independent variable simply increases the order of the polynomial solution. The final form of the integration shown in Equation 16 is:

$$\int_a^b Y_x dx \approx \left[\frac{\Delta a x^4}{4\Delta x} + \left(a_a + \frac{\Delta b - \Delta a(a)}{\Delta x} \right)\frac{x^3}{3} + \left(b_a + \frac{\Delta g - \Delta b(a)}{\Delta x} \right)\frac{x^2}{2} + \left(g_a - \frac{\Delta g(a)}{\Delta x} \right)x \right]_a^b$$

Equation 22

The obvious advantage of this method is that the integration has no new error introduced beyond what exists in the functional estimation. This is to say, the above equation is an exact solution to the integration of the functional estimation. Another advantage is that for large data sets, this method is non-iterative so there do not need to be any additional external function calls when this is implemented.

Linear Integration

The line-integral is a specialized integral that many engineers can go for years without ever contemplating. The reason it is important here is that it is required to check the results of the closed-form integral and perform the correlation task highlighted in Chapter II.3.3 and Chapter II.3.1.

In a plane, the orthogonal coordinates X and Y define a point's location with respect to all other points. Equation 23 below shows the general form of the linear distance formula applied to points A and B at (X_A, Y_A) and (X_B, Y_B) respectively.

$$R_{AB} = \sqrt{\left(X_B - X_A\right)^2 + \left(Y_B - Y_A\right)^2}$$

Equation 23

On an arbitrary path in a plane, the distance traveled between points A and B is not necessarily found with the above equation except when the path happens to be a straight line. The distance traveled along some arbitrary path can be approximated as a summation of short intermediate straight line segments such that:

$$S_{AB} = \sum_{i=1}^{n} \Delta R_i$$

Equation 24

where

$$\Delta R_i = \sqrt{\left(X_b - X_a\right)^2 + \left(Y_b - Y_a\right)^2} = \sqrt{\left(\Delta X_i\right)^2 + \left(\Delta Y_i\right)^2} \ .$$

Equation 25

As the intermediate steps become shorter and written as a function of X,

$$S_{AB} = \int_A^B dR = \int_A^B \sqrt{dX^2 + dY^2} = \int_{X_A}^{X_B} \sqrt{1 + \left(\frac{dY}{dX}\right)^2} dX = \int_{X_A}^{X_B} \sqrt{1 + f'^{\,2}_{(X)}}\, dX \ .$$

Equation 26

This equation can become exceptionally difficult to evaluate analytically[2, 6]. Evaluating the polynomial functional estimates returned by the SSFE, is especially difficult when the fact that the polynomial coefficients themselves are functions of X.

Two methods of approximating the line integral of Equation 26 were evaluated for use in the example problem of Chapter III. Both methods yielded similar results. The first, and simplest, method was to directly sum the intermediate ranges from point to point. The second was to use a Runga-Kutta 4 (RK4)-type approximation of the path to attempt to yield even greater fidelity to the estimated trajectory[6, 8]. Equation 27 through Equation 32 were used to implement this method.

$$S \approx \sum_{i=2}^{n} \sqrt{\left(\Delta X_i\right)^2 + \left(\Delta Y_i\right)^2}$$

Equation 27

Where

$$\Delta X_i = X_i - X_{i-1} \, ,$$

Equation 28

and

$$\Delta Y_i = \frac{\Delta Y_1 + 4\Delta Y_2 + \Delta Y_3}{6}.$$

Equation 29

The intermediate values ΔY_1, ΔY_2, and ΔY_3 are calculated as:

$$\Delta Y_1 = \Delta X_i \left\lfloor \dot{Y}_{(X_{i-1})} \right\rfloor,$$

Equation 30

$$\Delta Y_2 = \Delta X_i \left[\dot{Y}_{\left(\frac{X_i + X_{i-1}}{2}\right)} \right],$$

Equation 31

and

$$\Delta Y_3 = \Delta X_i \left\lfloor \dot{Y}_{(X_i)} \right\rfloor.$$

Equation 32

CHAPTER III

EXAMPLE

Example Problem

The measure of the usefulness of an algorithm or technique is the extent to which it allows the engineer to obtain useful information from available data. This example problem shows how the SSFE algorithm aids in extracting implied data from a limited set of trajectory information.

Because the original trajectory that prompted the creation of the SSFE is classified, the author created a new sample trajectory. The complete three-dimensional trajectory and corresponding real or “true” velocity components were calculated. From this a reduced data set was used for analysis to recreate the entire trajectory. As with the original real-world engineering task, only three sets of data were used for analysis: Altitude with Time, Horizontal Velocity with Time, and X and Y (Cross Range Position with Down Range Position). All position information is provided in units of meters, velocity is in meters per second, and time is in seconds. To make the analysis more realistic, measurement noise is introduced with the use of the RANDN function in MATLAB. The RANDN function produces an array of a specified size with a zero mean and a standard deviation of one. Multiplying the noise vectors produced using RANDN with an appropriate gain value scales the noise vector so that it can be used to simulate actually measured values. It is assumed that the X and Y components can be known to within one standard deviation (one-sigma) of ±50 meters, but altitude can be known to within ±1 meter. Horizontal velocity noise is assumed to be within a one-sigma of ±2 meters per second.

The trajectory, a variant of a dog-leg maneuver, was chosen because it adds enough complexity that the solution is not trivial and does look somewhat similar to a what a real missile trajectory might be. However, none of the trajectory information is intended to represent any specific real missile.

For simplicity sake, a flat Earth is assumed, although for many real-world missile trajectories, that assumption would add a significant amount of error.

True Trajectory

Before the trajectory could be analyzed, it needed to be created[7, 8]. Appendix A contains all of the MATLAB scripts used to both create the trajectory and analyze it with the SSFE and its associated functions.

Figure 1 below shows the true position components (X, Y, and Z where Z is altitude) with respect to time. **Error! Reference source not found.** shows the velocity components as well as the total horizontal velocity, and Figure 3 shows the complete three-dimensional true trajectory.

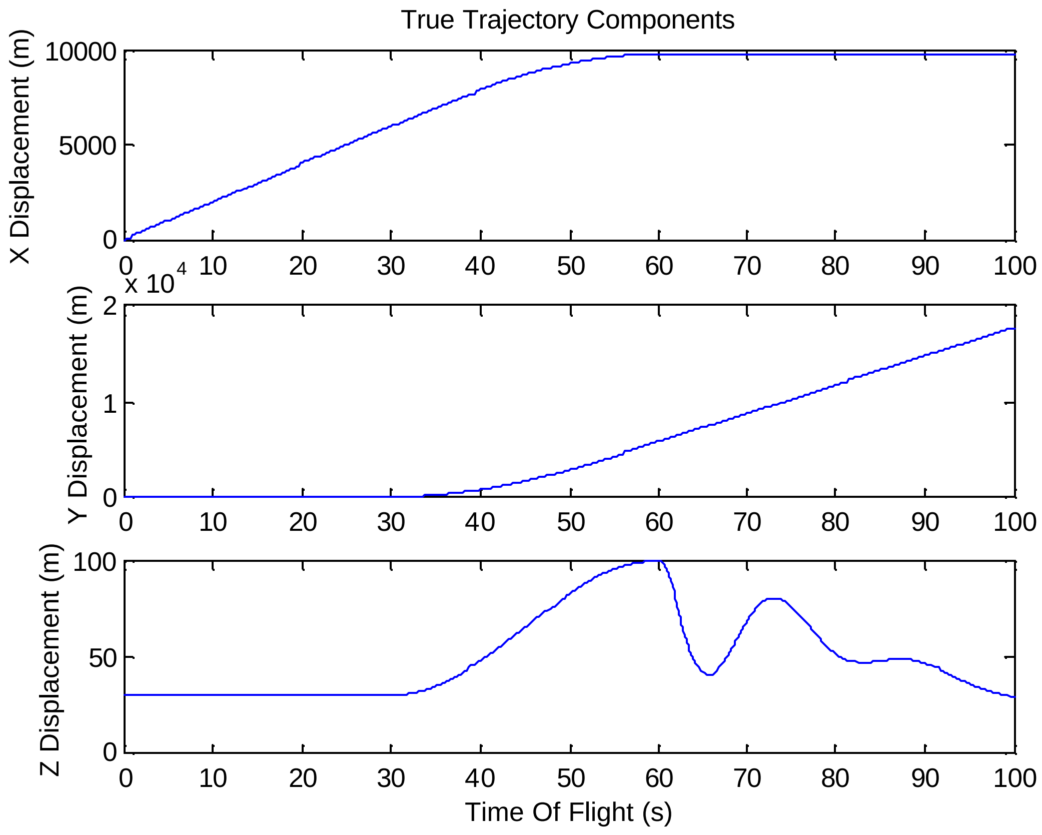

Figure 1 True trajectory displacement components with respect to Time Of Flight

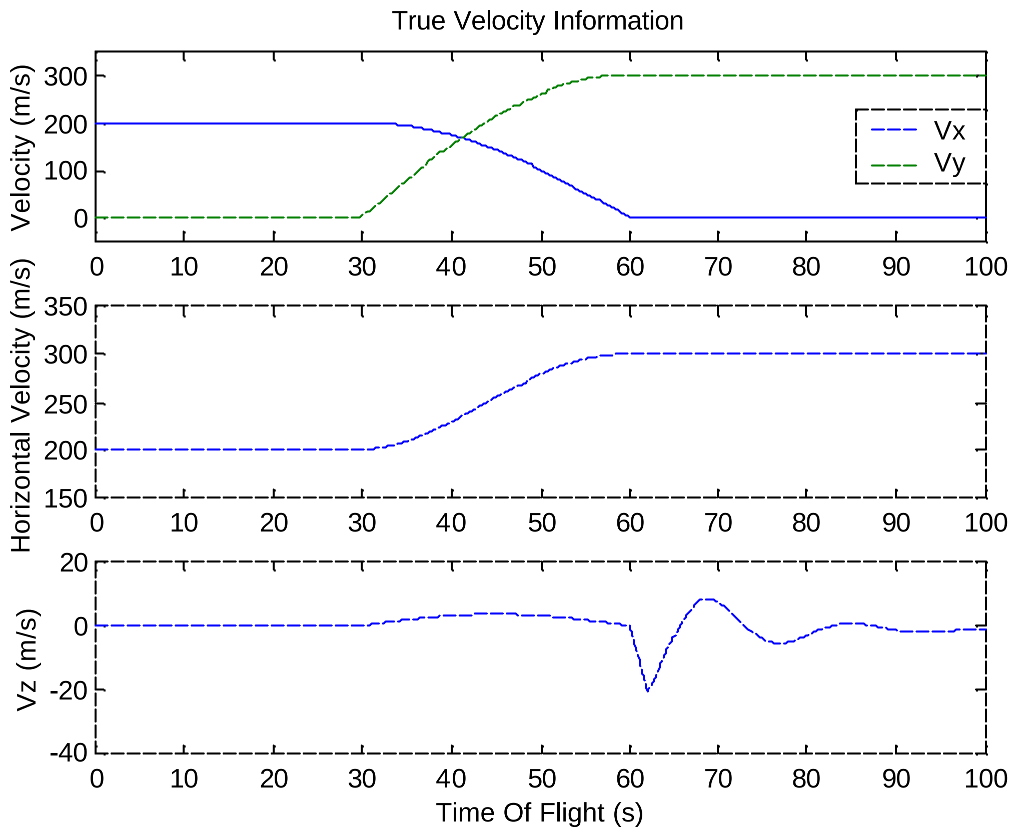

Figure 2 True trajectory velocity data with respect to Time Of Flight

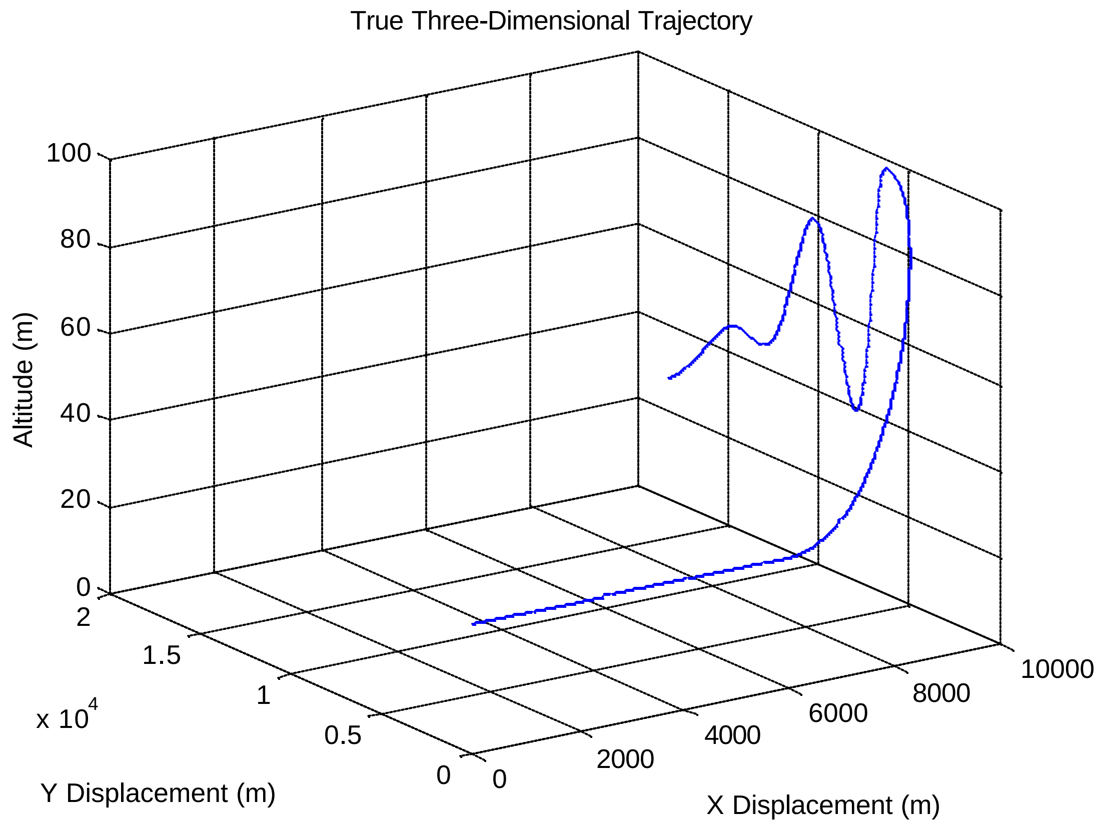

Figure 3 Three-dimensional view of true trajectory

Measured and Sampled Trajectory Data

The true data was produced at 10Hz spanning a time from 0.0 to 100.0 seconds. A random noise vector was added to the true data to simulate the uncertainty added with measurement devices. The amplitude of the noise was tailored to reasonable levels. For example with GPS and inertial navigation equipment it is quite reasonable to know the position of an object to within ±20 meters. This test of reasonableness was applied to the velocity and altitude information in a similar fashion.

The analysis script allowed two options for analysis. The first was a totally automated type that would sub-sample the noisy (10Hz) data based only on the total number of points. The alternate option was engineer-assisted sub-sampling of the noisy data.

A crude filter intended to lessen the impact of the variation caused by the noise was utilized in the sampling portion of the analysis. This filter, based on the RK4 method[1, 2, 6] was implemented as shown below in Equation 33.

$$Z_k = \frac{X_{i-1} + 4X_i + X_{i+1}}{6}$$

Equation 33

In the above equation, Z_k represents the current sample and X_i represents the index of the data selected for measurement. The X_{i-1} and X_{i+1} values are present to help reduce the effect of the random noise vector that had been added to the true data.

Figure 4 and Figure 5 show the noisy data made available for analysis. Figure 6 and Figure 7 show the data as sampled and pre-filtered for analysis.

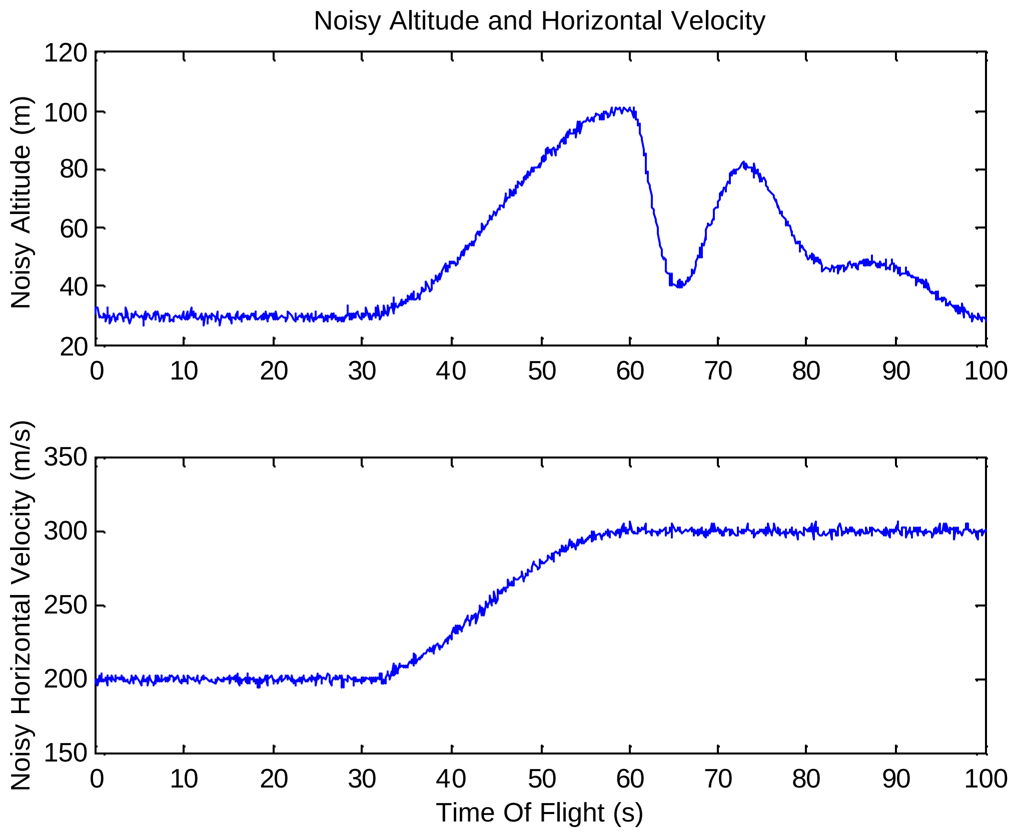

Figure 4 Noisy alt. and horizontal velocity with Time Of Flight available for analysis

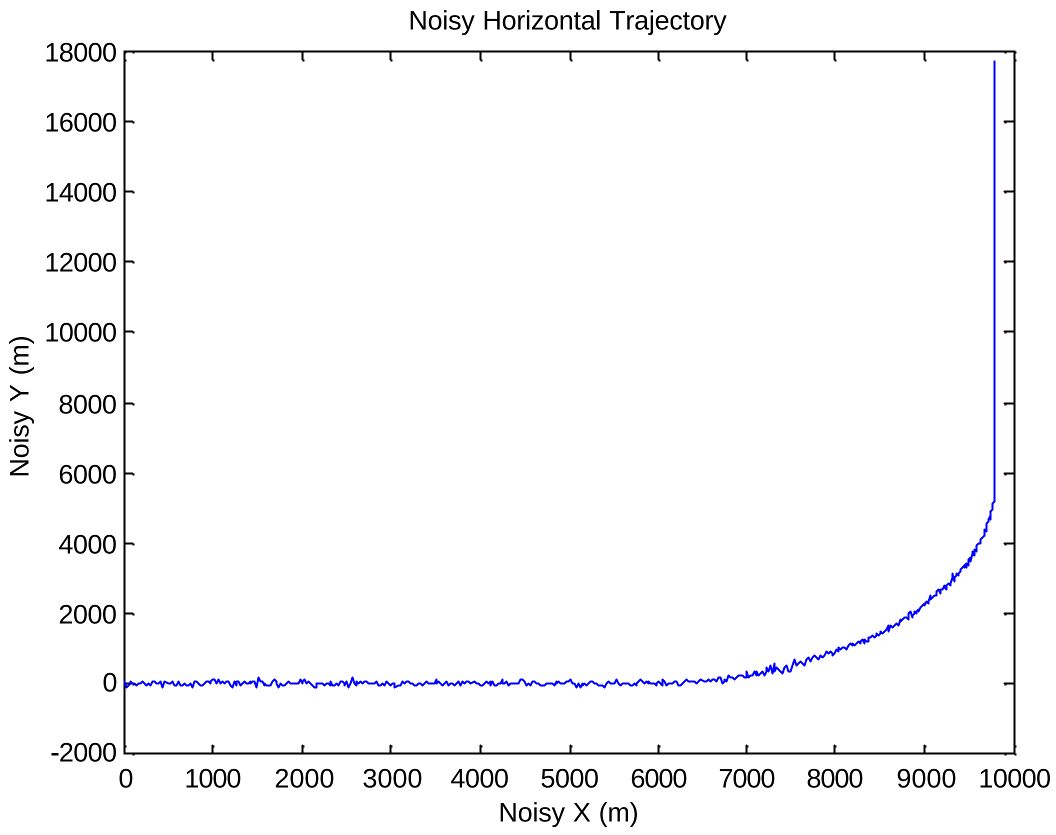

Figure 5 Noisy horizontal trajectory available for analysis

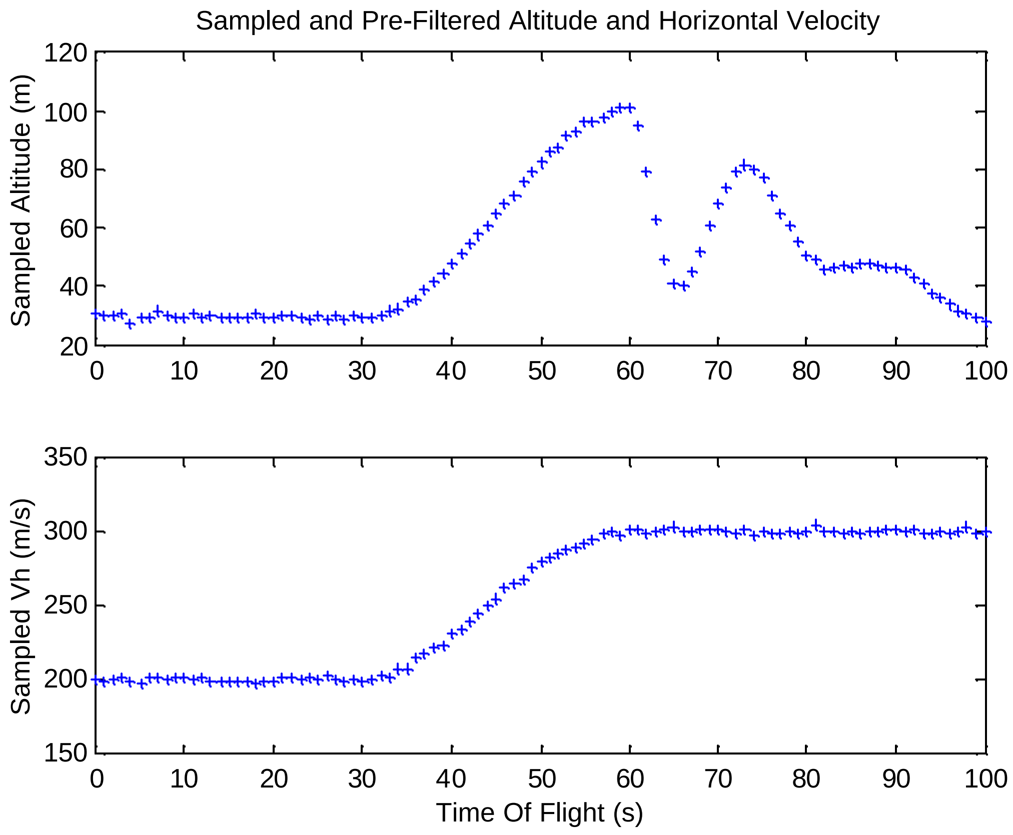

Figure 6 Sub-sampled and pre-filtered altitude and horizontal velocity with time

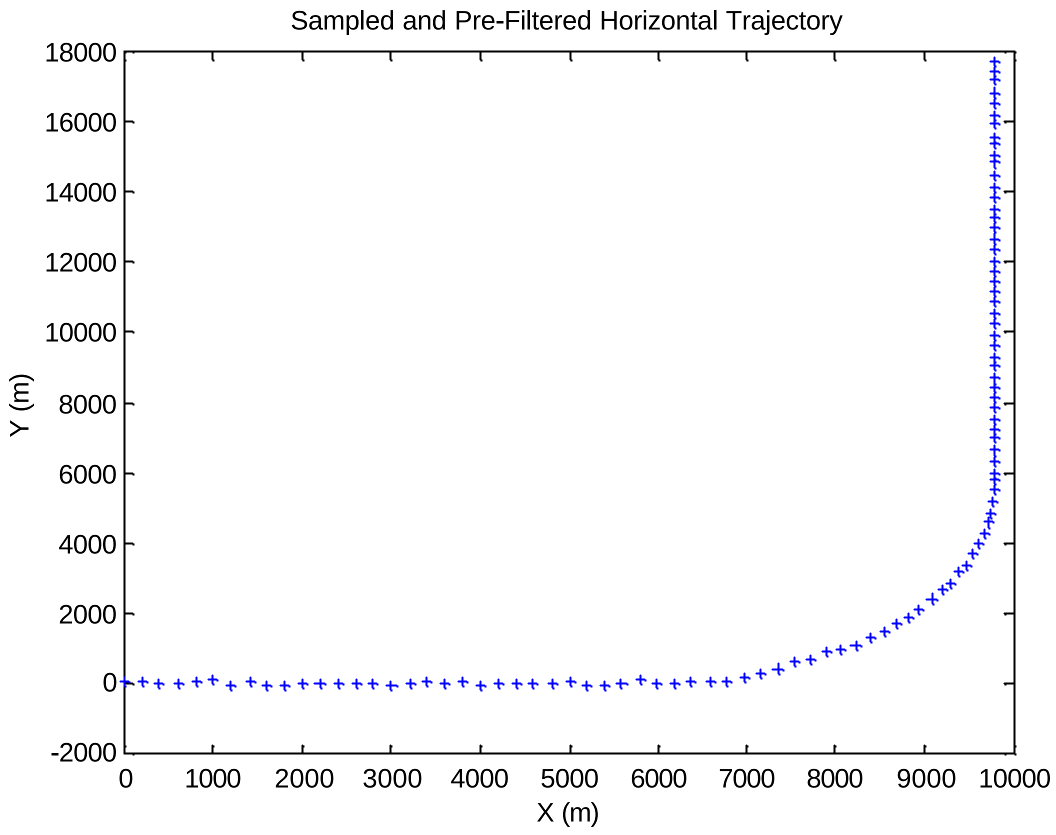

Figure 7 Sub-sampled and pre-filtered horizontal trajectory

Trajectory Reconstruction and Analysis

The first step in the trajectory reconstruction after sampling and pre-filtering the noisy data was to perform the line integration. Appendix A.6 is the routine used to perform the line integration. Because the trajectory starts completely in the X direction but ends traveling in the Y direction, the horizontal flight path is not functional with X. The line integration routine checks for this condition and will, if necessary, perform an axis transformation to rotate the perspective to allow the trajectory to be considered functional with a new set of axes. Figure 8 shows the sampled horizontal trajectory after it is rotated

15°. Once the trajectory can be considered a function of some X-axis, the horizontal distance traveled is estimated using Equation 27 through Equation 32.

After the line integration is performed on the XY flight path, the horizontal velocity versus time, shown in Figure 6, is integrated using the closed-form integration explained in Chapter III.3.3. Appendix A.5 implements this integration.

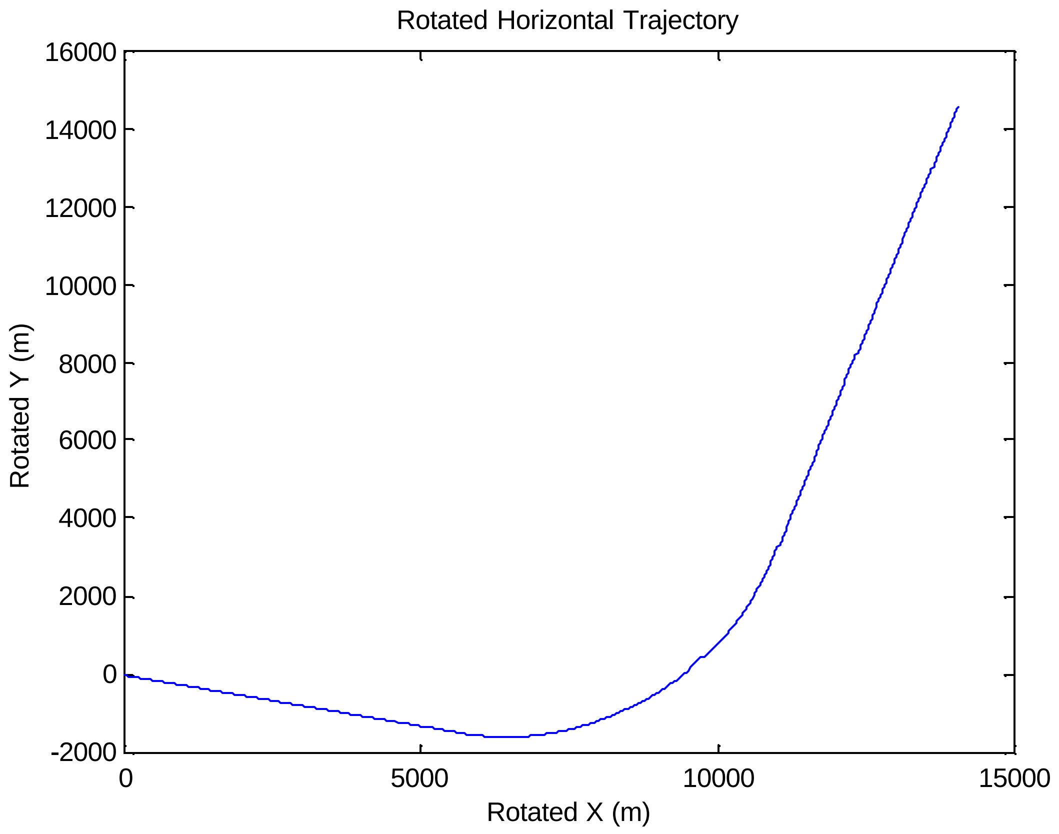

Figure 8 Horizontal trajectory rotated 15 degrees so that path can be considered a function of the rotated X axis

When the integrations are completed they are correlated. It is assumed that the final point (100.0 seconds) is known in both space and time. The difference between the time-integration of horizontal velocity distance at this time and the line integration of the horizontal path at that point was zeroed by the addition of a constant to the integrated velocity vector. This is equivalent to the constant of integration, which was not known before both integrations were completed. Because total distance traveled increases monotonically with time (the absolute value of velocity is always greater than zero), each point in time is associated with a unique value for distance traveled. Since the distance traveled (as calculated by the integration of velocity plus the constant of integration from the known point) is already associated with a specific time vector, Time Of Flight can be expressed as a function of integrated velocity as shown in Figure 9.

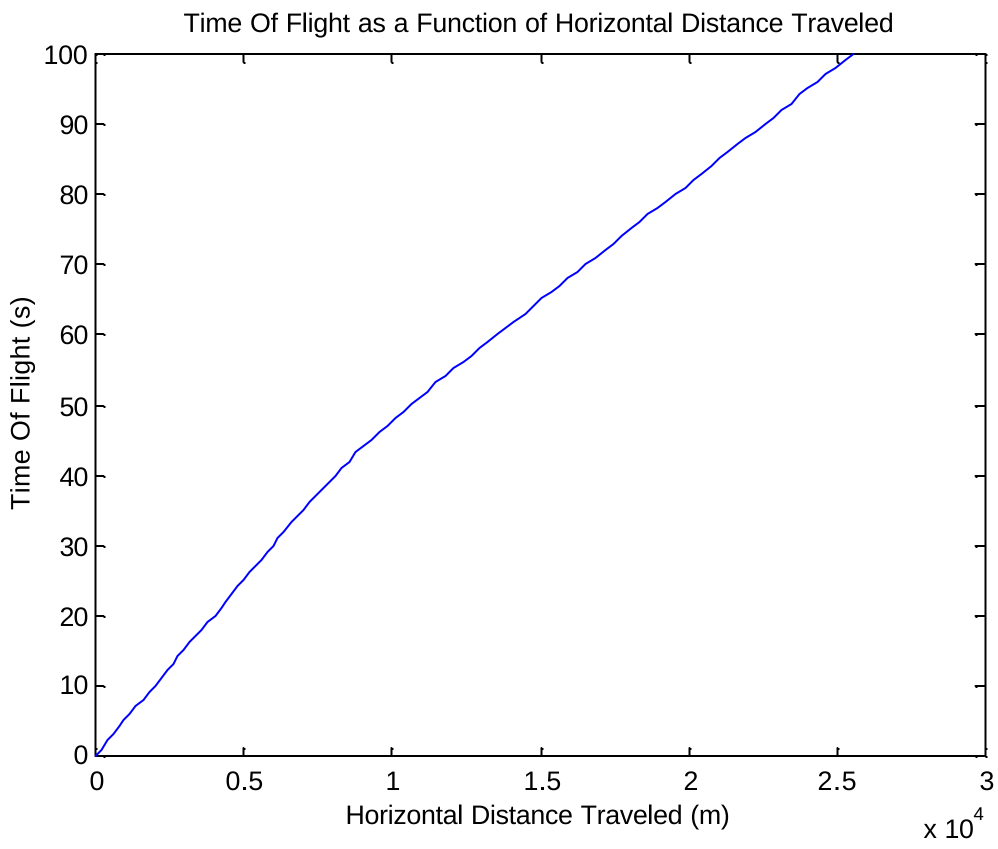

Figure 9 Time of flight as a function of integrated horizontal distance traveled

In the same manner, the rotated X position can be expressed as a function of distance traveled. The function interpextrap.m (Appendix A.7) utilizes the fact that if two vectors are describing the same quantity and one vector has a known relationship to some other quantity then by interpolation or extrapolation the second vector can be related to that other quantity as well. This correlation technique is discussed in Chapter II.3.1. Figure 10 shows the correlated calculations of distance traveled as functions of time, and Figure 11 shows the difference between the two calculations with time.

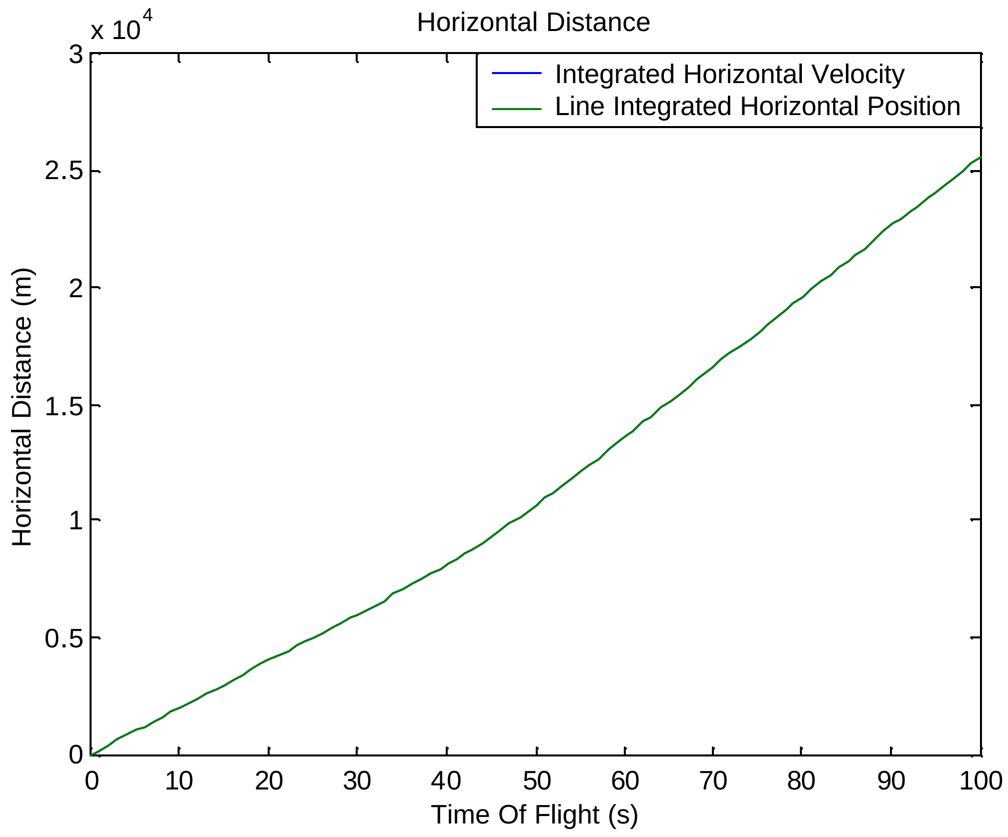

Figure 10 Correlated distance traveled as functions of Time Of Flight

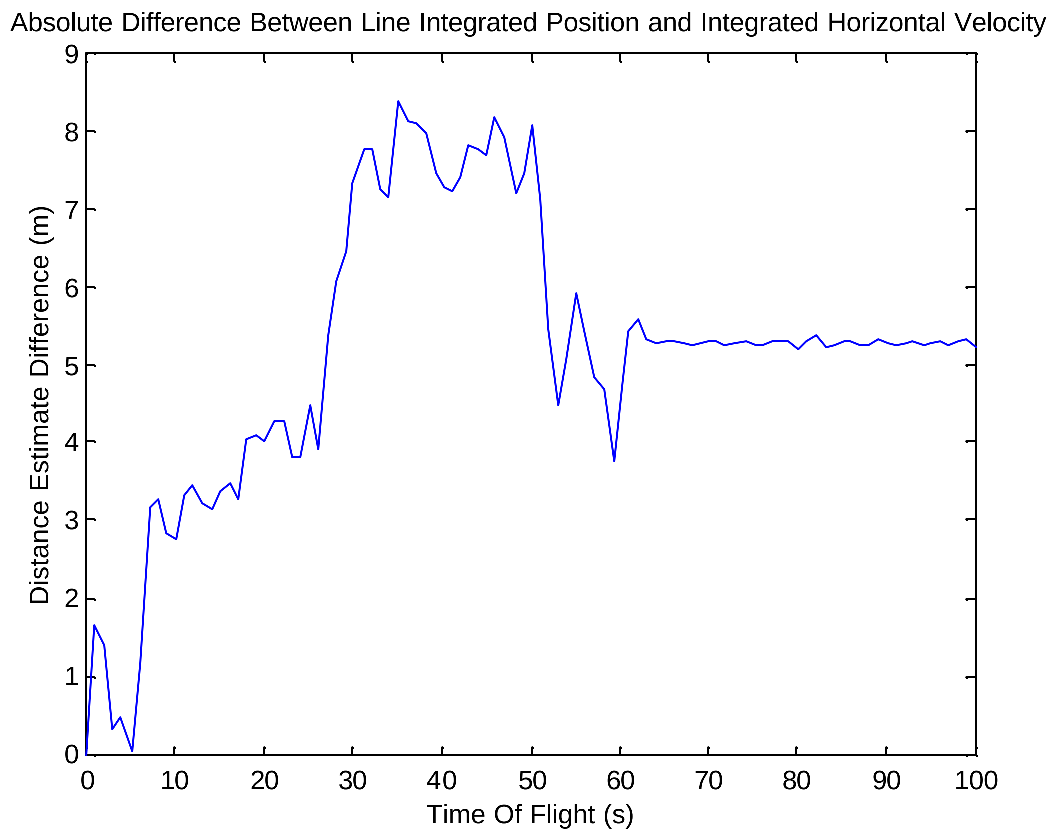

Figure 11 Absolute difference between time associated distance traveled estimations

Once the line integrated calculation of distance traveled had been associated with a time vector, rotated X and Y values were associated with the time vector in a like manner. Since the rotation angle of the X and Y information is known, the time associated X and Y vectors can be rotated back to yield time associated X and Y values in the original coordinate frame.

The final step in the trajectory reconstruction was to associate the altitude (Z) information with X and Y. Because altitude information was presented to the analysis routine with respect to time, the relationship between altitude and time can be found very quickly using the SSFE routine. Then, using the interpolation and extrapolation routine, the same time vector associated with X and Y can be associated with the altitude to complete the trajectory reconstruction.

Evaluation of the Trajectory Reconstruction Results

In evaluating the results of the trajectory reconstruction, it is important to note the two methods for obtaining data points, automated and engineer aided, provide slightly different results. Because one of the original goals of the algorithm was to reduce, as much as possible, the influence of the expertise of the engineer performing the analysis, the totally automated analysis will be reviewed in depth, and the important differences in performance will be discussed later.

Figure 12 shows the results of the analysis and the true data plotted with respect to time.

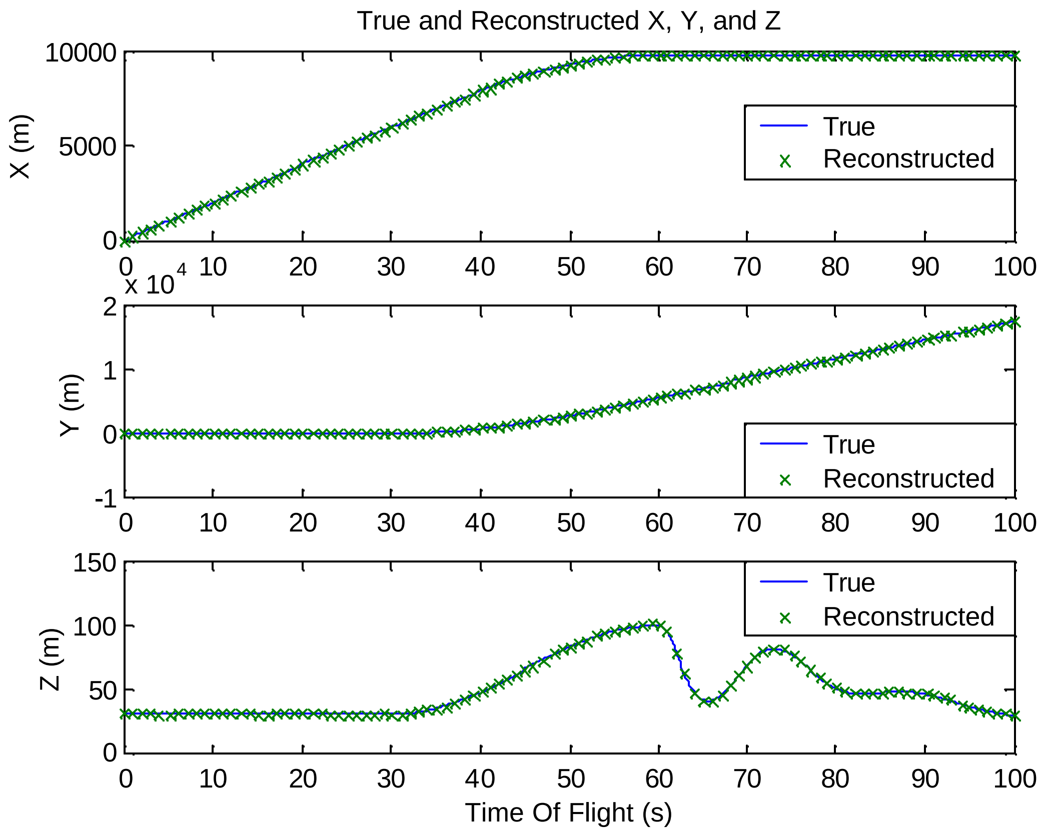

Figure 12 True and reconstructed trajectory with time

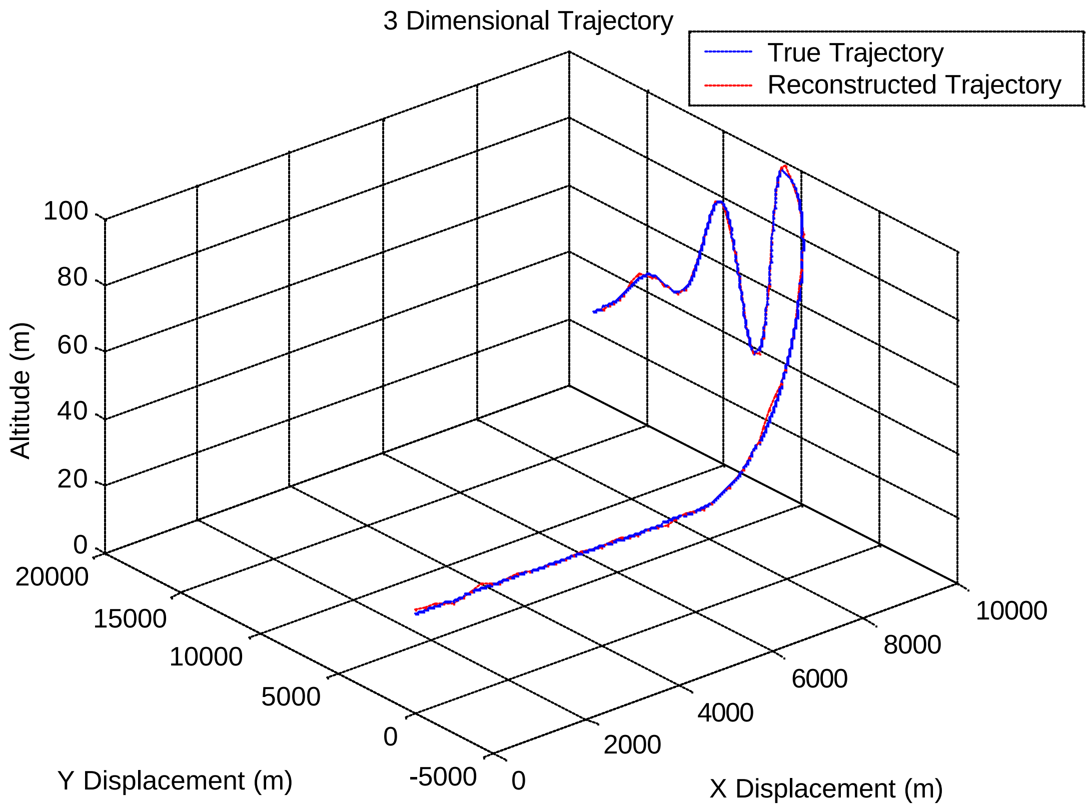

Figure 13 True and reconstructed three dimensional flight path

Figure 13 is a three dimensional view of the flight path with both the true and reconstructed flight paths shown.

Figure 14 shows the estimated velocity components and the true velocity information plotted with respect to time. The derivatives (velocities) of the reconstructed trajectory illustrate the greatest difference between the automated and the engineer aided methods of point selection. Figure 15 shows the velocity data when the engineer-aided method is used. The picked_pts.m function (Appendix A.9) includes the indices of the points selected. It is worth noting that by manually selecting the points an experienced engineer can use significantly fewer points than the automated point selection method would use to capture the shape of the data curve and perhaps more accurately capture some transitions than an automated sub-sampling routine would.

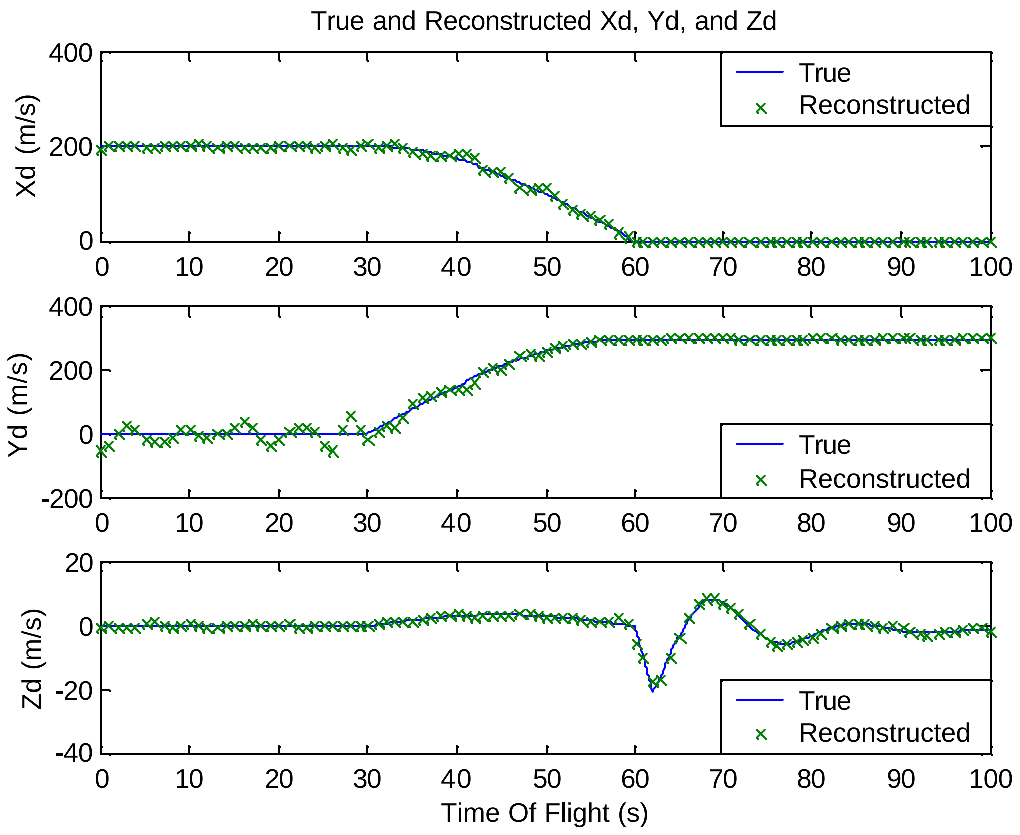

Figure 14 True and reconstructed velocity components (auto-sampled)

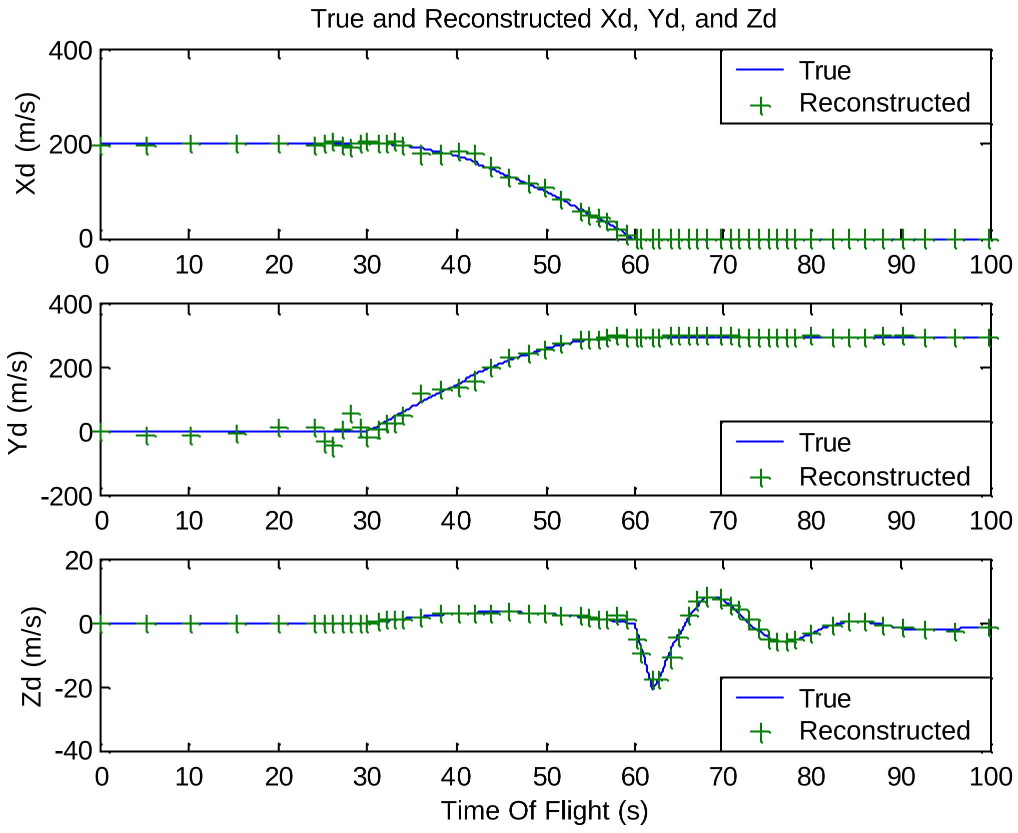

Figure 15 True and reconstructed velocity components (user-sampled)

Figure 16 through Figure 21 have the originally provided data (true and reconstructed) presented with the absolute error throughout the Time Of Flight. As would be expected, the SSFE algorithm provides some limited filtering effectiveness. The error is presented with the one-sigma variance value of the noise (the horizontal lines). The noise itself was not shown here because it thoroughly obscured the time history of the error in the reconstruction data.

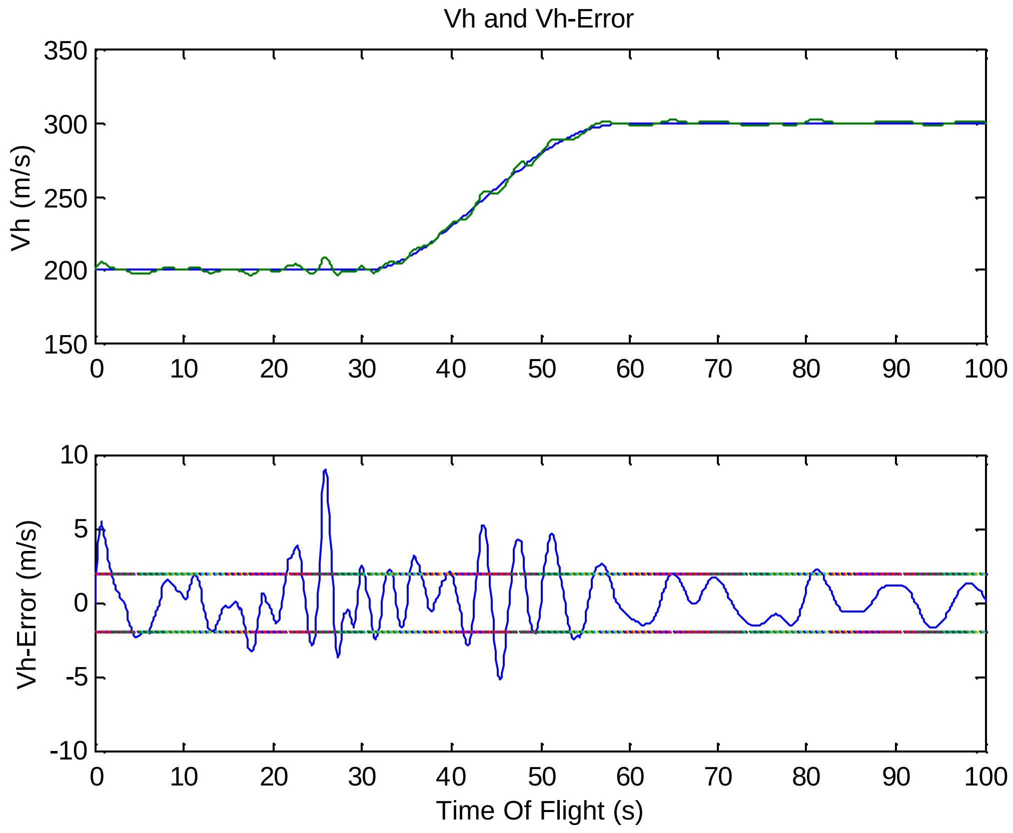

Figure 16 Vh, reconstructed Vh, and its error (auto sampled)

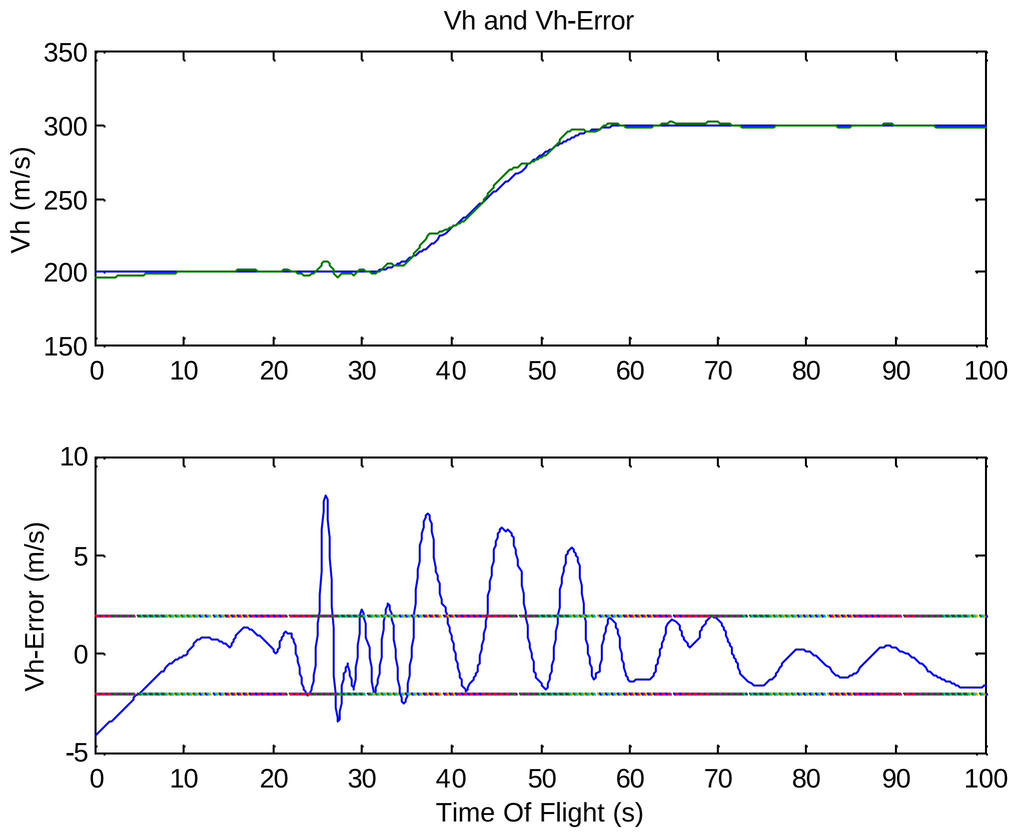

Figure 17 Vh, reconstructed Vh, and its error (user-sampled)

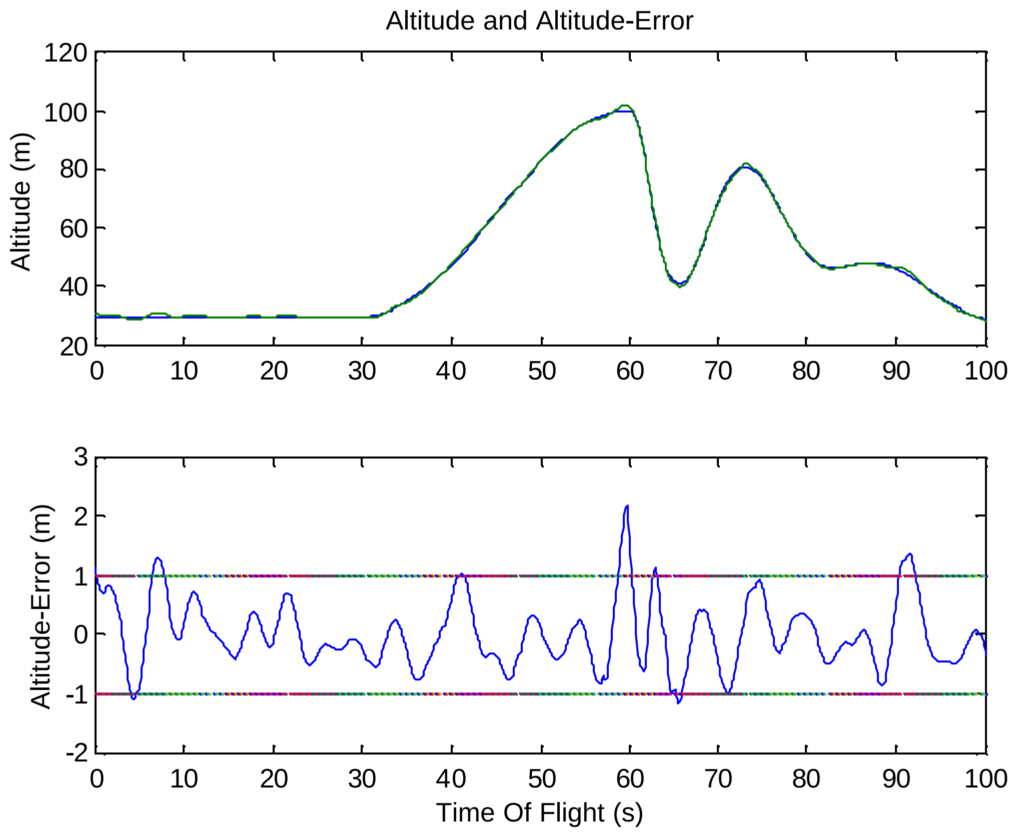

Figure 18 Alt., reconstructed alt., and its error (auto-sampled)

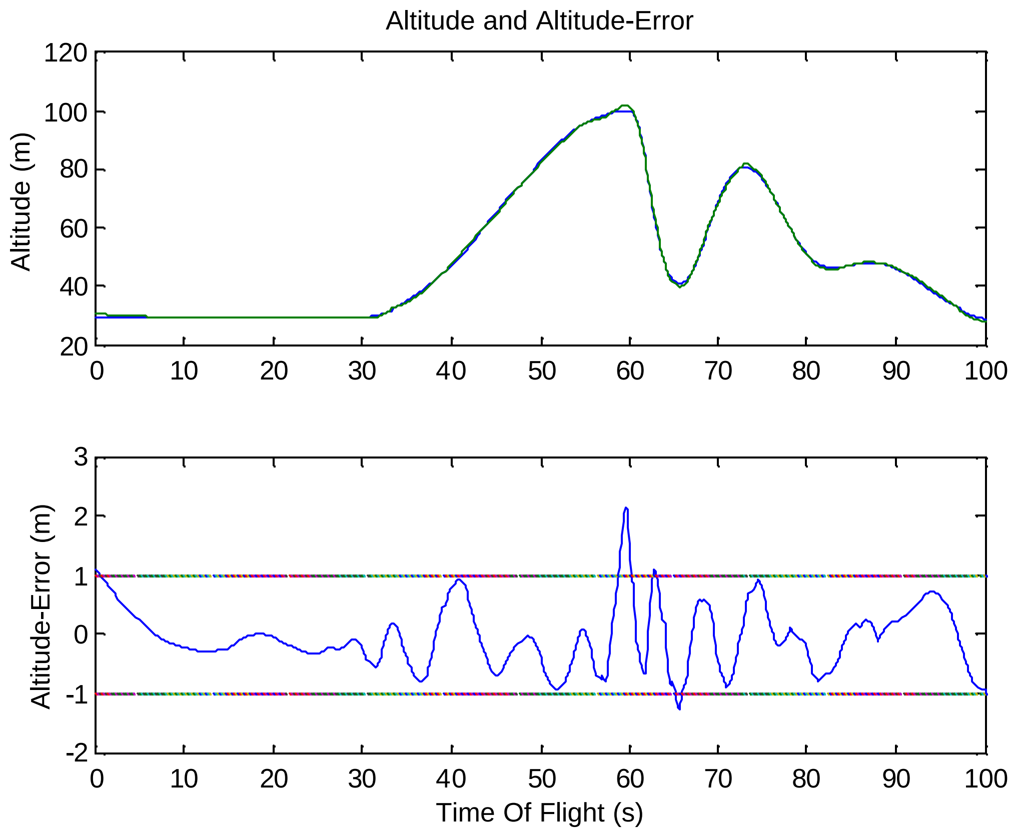

Figure 19 Alt., reconstructed alt., and its error (user-sampled)

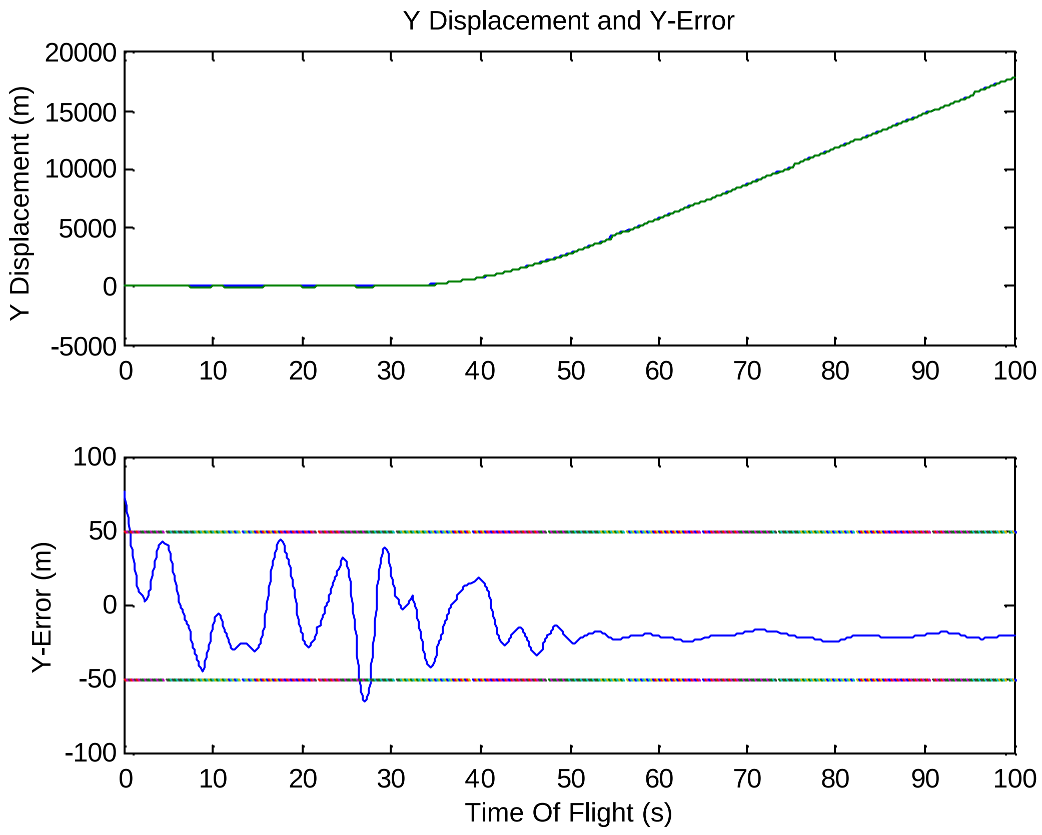

Figure 20 Y, reconstructed Y, and its error (auto-sampled)

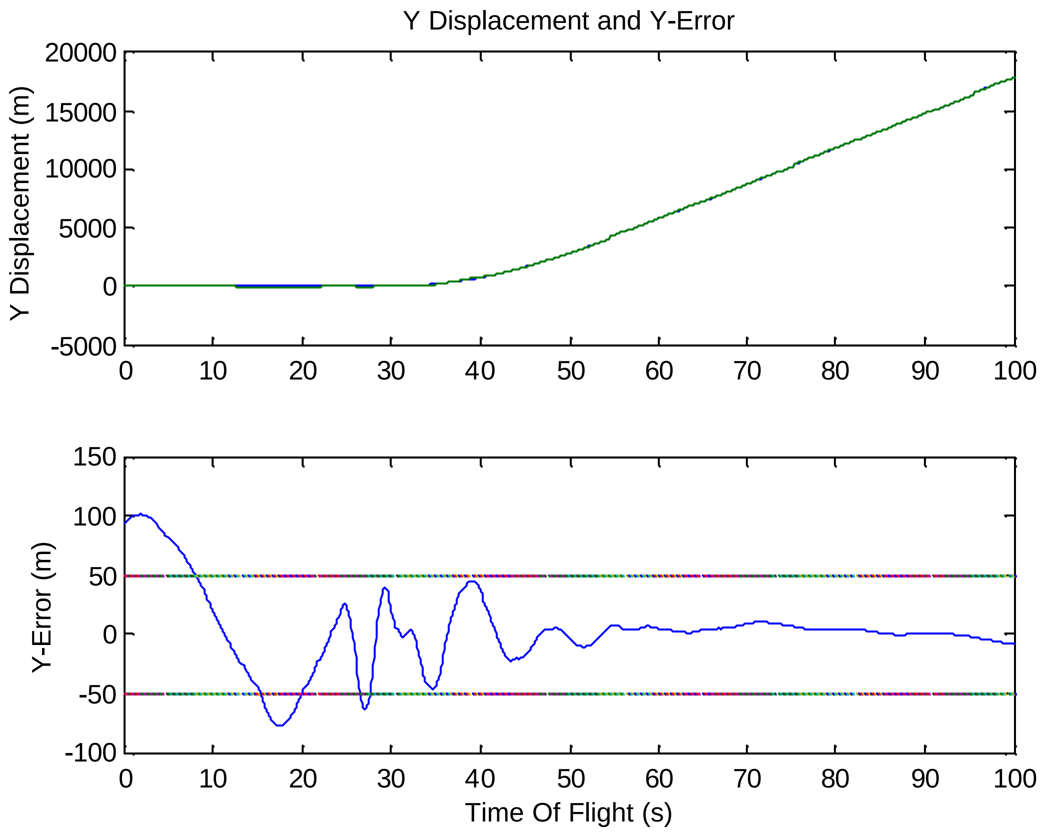

Figure 21 Y, reconstructed Y, and its error (user-sampled)

Figure 22 and Figure 23 are illustrations of the noise sampled in the analysis of the trajectory. The filtered series (the blue lines) utilize the simple filter of Equation 33. The trend of the noise was retained, but its extremes were reduced. The RMS value of the noise decreased between 25% and 30%.

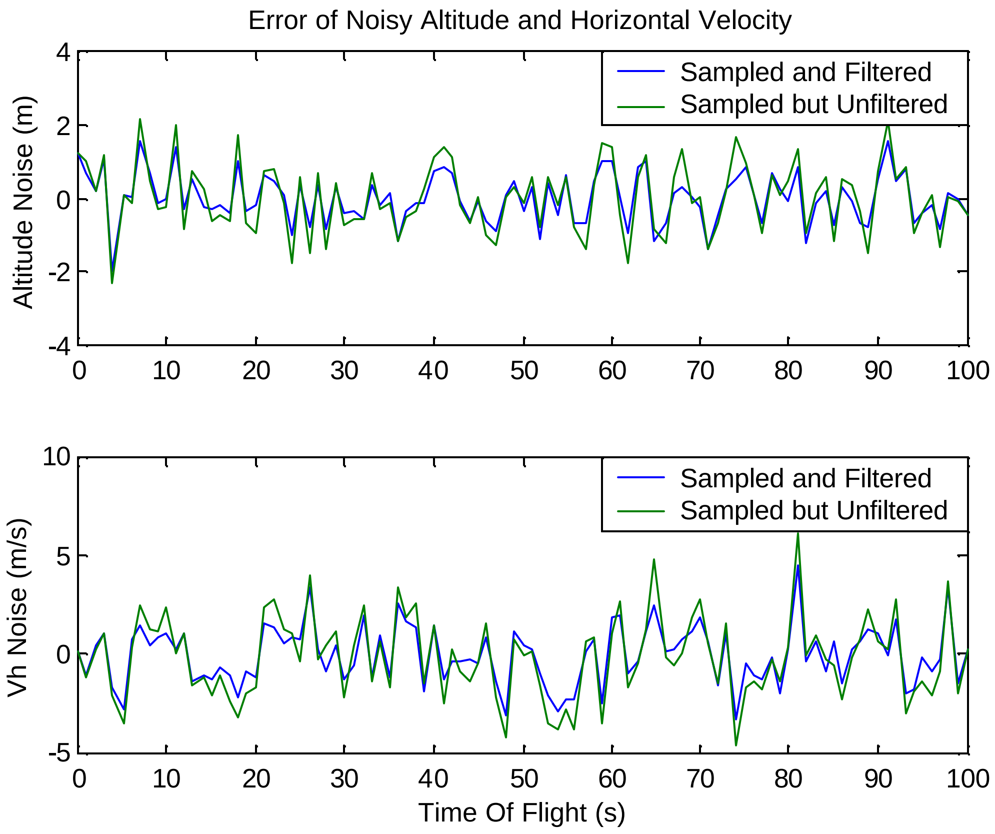

Figure 22 Noise of the sampled altitude and Vh with and without simple pre-filtering

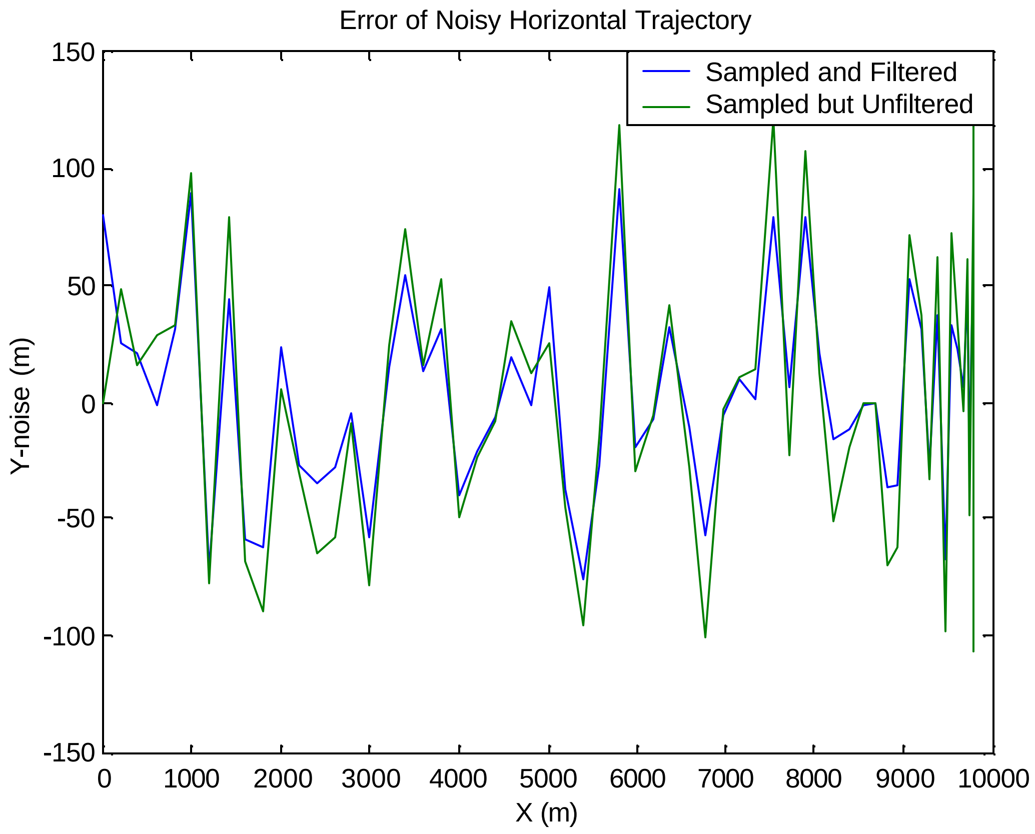

Figure 23 Noise of the sampled horizontal trajectory with and without simple pre-filtering

CHAPTER IV

CONCLUSIONS

Summary

A substantial part of almost every engineering task is extracting the implied information from the available data. In Chapter III a trajectory reconstruction problem which was patterned after the real world engineering problem that prompted the development of the SSFE algorithm, was examined. The difficulty in many situations is that to extract the required information from given data. The governing equation must either be known or its form, at the very least, must be assumed. Assumptions about the form of the governing equations invariably break down away from the point or points used to develop them. The goal of the SSFE was to produce reasonably accurate representations of the data at each point. Not only was it important for the assumed equation to approximate the data but also to estimate the derivative information in a realistic fashion.

As the results of the trajectory reconstruction showed, the SSFE algorithm enables the automated extraction of implicit data. Moreover, the first-order implied data, (velocity from position) is accurate enough to be useful.

The differences in performance between the analysis with the automatically selected samples and the analysis using the engineer selected sample points deserve mention. With automatically selected samples, it is easier to select a greater number of points. Because of the nature of the underlying least-squares polynomial curve fit, this resulted in a more accurate prediction of the three dimensional location of the missile at a greater number of specific points in time. Unfortunately, with the greater number of samples came a greater susceptibility to measurement noise, thus making the higher order derivative information less reliable. The horizontal velocity reconstruction of Figure 14 illustrates this point. The velocity information is relatively accurate, but as can readily be seen the derivatives of the velocity information are not representative of the true acceleration. Figure 15 illustrates the derivative information from an analysis using engineer-selected points. The advantage of this method is that by using points that are more widely spaced over much of the region, the variations caused by measurement noise can be spread out over greater distances. This allows the estimation of higher order information to be fairly representative of real acceleration. On the other hand, with fewer samples, the estimated functions match the true data precisely at fewer points.

The lower sub-plots of Figure 16 through Figure 21 illustrate an important characteristic of the SSFE algorithm regarding its filtering effectiveness that should be understood by an engineer before it is used on any problem. Figure 22 and Figure 23 show that, as expected, the pre-filtering performed using Equation 33 provided some of smoothing, reducing the RMS value of the noise by 25% to 30%. However, the smoothing due to pre-filtering is not so profound that the smoothing effect of the SSFE algorithm can be seen as trivial. For this reason, the SSFE algorithm can be viewed and used as a filter but its performance as a filter is sensitive to poorly pre-conditioned data. As the name of the algorithm implies, SSFE effectively smoothes the data on which it operates allowing the user to quickly obtain useful and reasonable implied information from measured data. The smoothing referenced in the name "Self-Smoothing Functional Estimation" is a reference to the algorithm's tendency to smooth derivative changes, not specifically to its effectiveness as a data filter.

Future Work

Future applications of the SSFE algorithm and its associated functions would be aided by more sophisticated pre-filtering. A well-constructed Kalman filter may prove to be a useful tool here[4, 8]. For a real-time application, such as parameter identification or vibration frequency monitoring, a simple low-pass filter may be sufficient.

Future development work will focus on quantifying the characteristics of the algorithm. Its performance as a filter, and its envelope of usability are significant areas to be investigated. Generalized guidelines for optimal performance based on the measurement density, noise to signal ratio, and higher-order derivative magnitude should also be developed. One intriguing possibility is that a method for automated adaptability (of window size and polynomial curve-fit order) may be developed to increase fidelity in regions of rapid changes of higher order derivatives. Finally, a more exhaustive search to indicate that this algorithm is a new invention is in order.

APPENDIX

CODE LISTING

This appendix includes the MATLAB scripts and functions created for the analysis the example in Chapter III.

MAKETRAJ.M

This script produced the trajectory that was analyzed. After the analysis routine had run, it called the analysis routine and plotted the results for evaluation.

ANALYZE_DATA.M

This function orchestrated the trajectory reconstruction.

SSFE.M

This function is the primary analysis tool used in the trajectory reconstruction. Given a matrix consisting of a column of independent variable values and a column of dependent variable values, it produces a second order polynomial functional estimation of the dependent variable at each point and returns the new estimated dependent variable values as well as the coefficients of the polynomials used to produce those values.

DERIVS.M

This function returns the derivative values for each independent variable entry as well as the functional representation of each derivative (first and second) at each point. This function uses the polynomial representation of the data produced by SSFE.M

POLYINTEGR8.M

This function performs a point-to-point closed form integration on the curve estimated by the polynomials produced by SSFE. This function returns a vector containing the integrated value up to each point. Because the values are solved closed-form, there is no numerical estimation error introduced with this function.

LI2.M

This function performs a numerical integration for the line integral as discussed in Chapter II.3.4.

INTERPEXTRAP.M

This function performs interpolation and extrapolation so that a new set of independent variable values can be matched to the dependent variable values and polynomials.

PICKSTP.M

This function picks the sub-sampling step size based on the length of the data vectors.

PICKED_PTS.M

This function allows the user to select the indices of the data for the analysis process.

```
%--------------------------------------------------------------
%---
%  maketraj.m
%
%  By:   B.T. Yake
%  Date: July 10, 2002
%  Rev.: v1.1
%
%  USAGE:  maketraj
%
%  PURPOSE:
%  This script is intended to create the trajectory information
%  that will be used as a thesis example for B. T. Yake.  After
%  the trajectory information is created, the analysis routine(s)
%  will be called.  Finally, the results of the analysis will be
%  plotted for examination.
%
%---
%--------------------------------------------------------------


%clear
close all
%--- CONSTANTS
```

```
grav=-9.81;      % approximation of acceleration of gravity - m/s^2

noise_on=1;      % turns measurement noise on-off (1=ON, 0=OFF)

analysis_type=1; % 1 = totally automated... 2 = engineer aided analysis

amp1=100;        % maximum Y measurement error (+ & -)

amp2=2;          % maximum Z measurement error(+ & -)

amp3=4;          % maximum Vh measurement error(+ & -)

gausnoise=1;     % noise distribution type (1=Gausian, 0=uniform)


%--- Horizontal plane

v1=200;  % starting velocity

v2=300;  % ending velocity

for ii=1:1001

  tim(ii)=(ii-1)/10;

  if ii<=300

    xd(ii)=v1;

    yd(ii)=0;

  else

    if xd(ii-1)>0.01

      xd(ii)=v1*cos(pi/60*(tim(ii)-30));

      yd(ii)=v2*sin(pi/60*(tim(ii)-30));

    else

      xd(ii)=xd(ii-1);

      yd(ii)=yd(ii-1);
      end
    end
    vh(ii)=sqrt(xd(ii)^2+yd(ii)^2);
end


% Initialize Position
x(1)=0;
y(1:300)=0;
for ii=2:300
    x(ii)=v1*tim(2)+x(ii-1);
end
for ii=301:1001
    if xd(ii-1)>0.01
      x(ii)=x(300)+60/pi*v1*sin(pi/60*tim(ii)-pi/2);
      y(ii)=y(300)-60/pi*v2*(cos(pi/60*tim(ii)-pi/2)-1);
      iimax=ii;
    else
      x(ii)=x(ii-1)+xd(ii)*tim(2);
      y(ii)=y(ii-1)+yd(ii)*tim(2);
    end
end


%--- Vertical plane
```

```
for ii=1:300
   z(ii)=30;
   zd(ii)=0;
   zdd(ii)=0;
end


for ii=301:601
   z(ii)=35*cos(pi*tim(ii)/30) + 65;
   zd(ii)=-(pi/30)*35*sin(pi*tim(ii)/30);
   zdd(ii)=-(pi/30)^2*35*cos(pi*tim(ii)/30);
end
lastii=ii;
for ii=lastii+1:621
   dt=tim(ii)-tim(lastii);
   z(ii)=z(lastii)+grav*(dt^2)/2;
   zd(ii)=grav*dt;
   zdd(ii)=grav;
end


dt=tim(2)-tim(1);
previi=lastii;
lastii=ii;
```

```
zeta=0.25;

wn=.4;

zcd=(30-z(lastii))/(100-tim(lastii));  % Commanded descent rate

bias=0;

z0=0;%z(lastii)-bias;

zd0=zd(lastii);

aa=wn*sqrt(1-zeta^2);

bb=zeta*wn;

cc=(zd0+bb*z0)/aa;


for ii=lastii+1:1001

   t=tim(ii)-tim(lastii);

   f=cc*exp(-bb*t);

   fd=-bb*cc*exp(-bb*t);

   fdd=-bb*bb*cc*exp(-bb*t);

   g=sin(aa*t);

   gd=aa*cos(aa*t);

   gdd=-aa*aa*sin(aa*t);

   h=z0*exp(-bb*t);

   hd=-bb*z0*exp(-bb*t);

   hdd=bb*bb*z0*exp(-bb*t);

   m=cos(aa*t);

   md=-aa*sin(aa*t);
```

```
  mdd=-aa*aa*cos(aa*t);
  z(ii)=real(f*g+m*h)+zcd*t+z(lastii);
  zd(ii)=real(fd*g+f*gd + hd*m+h*md)+zcd;  % <--- chain rule
  zdd(ii)=real(2*fd*gd + fd*gdd + 2*fd*md+ hd*mdd); % <--- chain rule... again
end
figure
subplot(3,1,1),plot(tim,x)
title('True Trajectory Components')
ylabel('X Displacement (m)')
subplot(3,1,2),plot(tim,y)
ylabel('Y Displacement (m)')
subplot(3,1,3),plot(tim,z)
xlabel('Time Of Flight (s)')
ylabel('Z Displacement (m)')


figure
subplot(3,1,1),plot(tim,xd,tim,yd)
title('True Velocity Information')
ylabel('Velocity (m/s)')
subplot(3,1,2),plot(tim,vh)
ylabel('Horizontal Velocity (m/s)')
subplot(3,1,3),plot(tim,zd)
```

```
xlabel('Time Of Flight (s)')

ylabel('Vz (m/s)')


figure

plot(tim,z)

xlabel('Time Of Flight (s)')

ylabel('True Altitude (m)')


figure

plot(tim,zd)

xlabel('Time Of Flight (s)')

ylabel('True Altitude Rate (m/s)')


figure

plot(tim,zdd)

xlabel('Time Of Flight (s)')

ylabel('True Altitude Acceleration (m/s^2)')


traj3d=figure;

plot3(x,y,z)

xlabel('X Displacement (m)')

ylabel('Y Displacement (m)')

zlabel('Altitude (m)')
```

```
if gausnoise==1
   %--- Noise using Gausian distribution (mean=0, var=1)
   if exist('raw_noise1')==0
      raw_noise1=randn([1,1001]);
      raw_noise2=randn([1,1001]);
      raw_noise3=randn([1,1001]);
      raw_noise4=randn([1,1001]);
      raw_noise5=randn([1,1001]);
      raw_noise6=randn([1,1001]);
   end
   %--- make noisy data
   noisyx=x;
   noisyy=y+(amp1/2)*raw_noise2*noise_on;
   noisyz=z+(amp2/2)*raw_noise3*noise_on;
   noisyvh=vh+(amp3/2)*raw_noise4*noise_on;
else
   %--- Noise using uniform distribution
   if exist('raw_noise1')==0
      raw_noise1=rand([1,1001]);
      raw_noise2=rand([1,1001]);
      raw_noise3=rand([1,1001]);
```

```
    raw_noise4=rand([1,1001]);
    raw_noise5=rand([1,1001]);
    raw_noise6=rand([1,1001]);
  end
  %--- make noisy data
  noisyx=x;
  noisyy=y+(amp1*raw_noise2-.5*amp1)*noise_on;
  noisyz=z+(amp2*raw_noise3-.5*amp2)*noise_on;
  noisyvh=vh+(amp3*raw_noise4-0.5*amp3)*noise_on;
end




%--- make noisy data
noisyx=x;
noisyy=y+(amp1/2)*raw_noise2*noise_on;
noisyz=z+(amp2/2)*raw_noise3*noise_on;
noisyvh=vh+(amp3/2)*raw_noise4*noise_on;

figure
subplot(2,1,1),plot(tim,noisyz)
title('Noisy Altitude and Horizontal Velocity')
```

```
ylabel('Noisy Altitude (m)')

subplot(2,1,2),plot(tim,noisyvh)

xlabel('Time Of Flight (s)')

ylabel('Noisy Horizontal Velocity (m/s)')


figure

plot(noisyx,noisyy)

title('Noisy Horizontal Trajectory')

xlabel('Noisy X (m)')

ylabel('Noisy Y (m)')


%truedata=[x' y' z' vh'];

xyout=[noisyx' noisyy'];

tvhout=[tim' noisyvh'];

tzout=[tim'noisyz'];

knownpt=[tim(1001) noisyx(1001) noisyy(1001) noisyz(1001)];

[recontraj,px,py,pz,ltraj]=analyze_data(xyout,tvhout,tzout,knownpt,analysis_type,…

   truedata);

%[recontraj,px,py,pz,ltraj]=analyze_data(xyout,tvhout,tzout,knownpt,analysis_type);


newtim=recontraj(:,1);

rcx=recontraj(:,2);

rcy=recontraj(:,3);
```

```
rcz=recontraj(:,4);


[mnt,nnt]=size(newtim);


[rcxdots, polxdots]=derivs(newtim,px);

rcxd=rcxdots(:,1);

[rcydots, polydots]=derivs(newtim,py);

rcyd=rcydots(:,1);

[rczdots, polzdots]=derivs(newtim,pz);

rczd=rczdots(:,1);


[bigrcx,bigpx]=interpextrap(newtim,rcx,tim');

[bigrcy,bigpy]=interpextrap(newtim,rcy,tim');

[bigrcz,bigpz]=interpextrap(newtim,rcz,tim');

[bigrcxd,bigpxd]=interpextrap(newtim,rcxd,tim');

[bigrcyd,bigpyd]=interpextrap(newtim,rcyd,tim');

[bigrczd,bigpzd]=interpextrap(newtim,rczd,tim');


figure(traj3d) % 4

hold on

title('3 Dimensional Trajectory')

plot3(rcx,rcy,rcz,'r')
```

```
figure % 9

subplot(3,1,1),plot(tim,x'-bigrcx)

title('Difference X, Y, and Z plots vs. Time')

subplot(3,1,2),plot(tim,y'-bigrcy)

subplot(3,1,3),plot(tim,z'-bigrcz)

xlabel('Time Of Flight (s)')



figure %10

subplot(3,1,1),plot(tim,xd'-bigrcxd)

title('Difference Xd, Yd, and Zd plots vs. Time')

subplot(3,1,2),plot(tim,yd'-bigrcyd)

subplot(3,1,3),plot(tim,zd'-bigrczd)

xlabel('Time Of Flight (s)')



figure %11

subplot(3,1,1),plot(tim,xd',newtim,rcxd)%tim,bigrcxd)

ylabel('Xd (m/s)')

title('True and Reconstructed Xd, Yd, and Zd');

subplot(3,1,2),plot(tim,yd',newtim,rcyd)%tim,bigrcyd)

ylabel('Yd (m/s)')
```

```
subplot(3,1,3),plot(tim,zd',newtim,rczd)%tim,bigrczd)

xlabel('Time Of Flight (s)')

ylabel('Zd (m/s)')




figure %12

subplot(3,1,1),plot(tim',x',newtim,rcx)%tim',bigrcx)%,tim',noisyx')

title('True and Reconstructed X, Y, and Z');

ylabel('X (m)')

subplot(3,1,2),plot(tim',y',newtim,rcy)%tim',bigrcy)%,tim',noisyy')

ylabel('Y (m)')

subplot(3,1,3),plot(tim',z',newtim,rcz)%tim',bigrcz)%,tim',noisyz')

xlabel('Time Of Flight (s)')

ylabel('Z (m)')




junk=0;

[trulinint,newxy,angl2]=li2([x' y'],junk);

[bigltraj,bigpltraj]=interpextrap(newtim,ltraj,tim');


figure %13

plot(trulinint,x,bigltraj,x)

ylabel('X displacement (m)')
```

```
xlabel('Total Horizantal displacement (m)')


figure %14
plot(tim,trulinint-bigltraj)
title('Linear Integration Error')
xlabel('Time Of Flight (s)')


for ii=1:1001
   bigrcvh(ii)=sqrt(bigrcxd(ii)^2+bigrcyd(ii)^2);
end
%abs_deltavh=abs(bigrcvh-vh);
deltavh=bigrcvh-vh;


figure %15
subplot(2,1,1),plot(tim,vh,tim,bigrcvh)
title('Vh and Vh-Error')
ylabel('Vh (m/s)')
%subplot(2,1,2),plot(tim,abs_deltavh,tim,amp3*0.5)
subplot(2,1,2),plot(tim,deltavh,tim,amp3*0.5,tim,-amp3*0.5)
xlabel('Time Of Flight (s)')
ylabel('Vh-Error (m/s)')


figure %16
```

```
subplot(2,1,1),plot(tim,z,tim,bigrcz)
title('Altitude and Altitude-Error')
ylabel('Altitude (m)')
%subplot(2,1,2),plot(tim,abs(bigrcz-z'),tim,amp2*0.5)
subplot(2,1,2),plot(tim,bigrcz-z',tim,amp2*0.5,tim,-amp2*0.5)
xlabel('Time Of Flight (s)')
ylabel('Altitude-Error (m)')

figure %17
subplot(2,1,1),plot(tim,y,tim,bigrcy)
title('Y Displacement and Y-Error')
ylabel('Y Displacement (m)')
%subplot(2,1,2),plot(tim,abs(bigrcy-y'),tim,amp1*0.5)
subplot(2,1,2),plot(tim,bigrcy-y',tim,amp1*0.5,tim,-amp1*0.5)
xlabel('Time Of Flight (s)')
ylabel('Y-Error (m)')

% END script: MAKETRAJ
```

```
%-----------------------------------------------------------------------
%---
%   FUNCTION: analyze_data.m
%
%   AUTHOR:   B. T. Yake
%   DATE:     7/10/2002
%   REV:      v1.1
%
%   USAGE:
%   [dataout]=analyze_data(xy,tim_vh,tim_z)
%    inputs:  xy      -> Two columns of data containing the sampled (noisy)
%                        X and Y position data for the entire flight
%             tim_vh  -> Two columns of data containg time and the sampled
%                        (noisy) horizontal velocity data
%             tim_z   -> Two columns of data containg time and the sampled
%                        (noisy) altitude data
%             known   -> A vector containing the coordinates of [t, x, y, z]
%                        for a point in the trajectory.
%             truedata-> True trajectory information.
%    outputs: dataout -> Seven column array containing
%                        [Time X Y Z Xd Yd Zd]
%   PURPOSE:
%   This routine is to a limited set of trajectory data from "maketraj.m"
%   and reconstruct the full three-dimenstional filght-path.
%
%   NOTE:
%   v1.0  - 05/01/2002 - Baseline
%   v1.1  - 07/10/2002 - Added "truedata" input so that the filtering
%                        effect of the pre-filtering could be evaluated
%---
%---------------------------------------------------------------------------

function [dataout,px,py,pz,ll_out]=analyze_data(xy,tim_vh,tim_z,known,a_type)
%function [dataout,px,py,pz,ll_out]=analyze_data(xy,tim_vh,tim_z,known,a_type,
truedata)

t_known=known(1);
x_known=known(2);
y_known=known(3);
z_known=known(4);
rng_known=sqrt(x_known^2+y_known^2);


[xyrows,n]=size(xy);
[vhrows,n]=size(tim_vh);
[zrows,n]=size(tim_z);
```

```
noisyx(:,1)=xy(:,1);

noisyy(:,1)=xy(:,2);

tvh(:,1)=tim_vh(:,1);

noisyvh(:,1)=tim_vh(:,2);

tz(:,1)=tim_z(:,1);

noisyz(:,1)=tim_z(:,2);


if (a_type==1 | a_type==2)

   analysis_type = a_type;  % 1 = totally automated... 2 = engineer aided analysis

else

   analysis_type = 1;  % 1 = totally automated... 2 = engineer aided analysis

end




%------------------------------------------------------------------------

%--- FIRST: prepare data

if analysis_type==1

   % Totally automated

   xystp=pickstp(xyrows);

   vhstp=pickstp(xyrows);

   zstp=pickstp(xyrows);
```

```
% Take a reduced sample set of data

jj=1;

sx(1,1)=noisyx(1);

sy(1,1)=noisyy(1);

stvh(1,1)=tvh(1);

svh(1,1)=noisyvh(1);

stz(1,1)=tz(1);

sz(1,1)=noisyz(1);

ufx(1,1)=sx(1);

ufy(1,1)=sy(1);

ufz(1,1)=sz(1);

ufvh(1,1)=svh(1);

uft(1,1)=stz(1);

truex(1,1)=truedata(1,1);

truey(1,1)=truedata(1,2);

truez(1,1)=truedata(1,3);

truevh(1,1)=truedata(1,4);




for kk=xystp+1:xystp:xyrows-1

  jj=jj+1;
  sx(jj,1)=(noisyx(kk+1)+4*noisyx(kk)+noisyx(kk-1))/6;

  sy(jj,1)=(noisyy(kk+1)+4*noisyy(kk)+noisyy(kk-1))/6;

  ufx(jj,1)=noisyx(kk);

  ufy(jj,1)=noisyy(kk);

  truex(jj,1)=truedata(kk,1);

  truey(jj,1)=truedata(kk,2);

end

xymax=jj+1;

sx(xymax,1)=noisyx(xyrows);

sy(xymax,1)=noisyy(xyrows);

ufx(xymax,1)=noisyx(xyrows);

ufy(xymax,1)=noisyy(xyrows);

truex(xymax,1)=truedata(xyrows,1);

truey(xymax,1)=truedata(xyrows,2);


jj=1;

for kk=vhstp+1:vhstp:vhrows-1

  jj=jj+1;

  stvh(jj,1)=(tvh(kk+1)+4*tvh(kk)+tvh(kk-1))/6;

  svh(jj,1)=(noisyvh(kk+1)+4*noisyvh(kk)+noisyvh(kk-1))/6;

  uft(jj,1)=tvh(kk);

  ufvh(jj,1)=noisyvh(kk);

  truevh(jj,1)=truedata(kk,4);
  end
  vhmax=jj+1;
  stvh(vhmax,1)=tvh(vhrows);
  svh(vhmax,1)=noisyvh(vhrows);
  ufvh(vhmax,1)=noisyvh(vhrows);
  uft(vhmax,1)=tvh(vhrows);
  truevh(vhmax,1)=truedata(vhrows,4);
  jj=1;
  for kk=zstp+1:zstp:zrows-1
    jj=jj+1;
    stz(jj,1)=(tz(kk+1)+4*tz(kk)+tz(kk-1))/6;
    sz(jj,1)=(noisyz(kk+1)+4*noisyz(kk)+noisyz(kk-1))/6;
    ufz(jj,1)=noisyz(kk);
    truez(jj,1)=truedata(kk,3);
  end
  zmax=jj+1;
  stz(zmax,1)=tz(zrows);
  sz(zmax,1)=noisyz(zrows);
  ufz(zmax,1)=noisyz(zrows);
  truez(zmax,1)=truedata(zrows,3);

else   % Engineer picks independent variable vectors
  vectorset=1;
```

```
 [xy_ndx,tvh_ndx,tz_ndx]=picked_pts(vectorset);
 [m,n]=size(xy_ndx);
 sx(1,1)=noisyx(xy_ndx(1));
 sy(1,1)=noisyy(xy_ndx(1));
 for ii=2:m-1
   sx(ii,1)=(noisyx(xy_ndx(ii)+1)+4*noisyx(xy_ndx(ii))+noisyx(xy_ndx(ii)-1))/6;
   sy(ii,1)=(noisyy(xy_ndx(ii)+1)+4*noisyy(xy_ndx(ii))+noisyy(xy_ndx(ii)-1))/6;
 end
 sx(m,1)=noisyx(xy_ndx(m));
 sy(m,1)=noisyy(xy_ndx(m));

 [m,n]=size(tvh_ndx);
 stvh(1,1)=tvh(tvh_ndx(1));
 svh(1,1)=noisyvh(tvh_ndx(1));
 for ii=2:m-1
   stvh(ii,1)=(tvh(tvh_ndx(ii)+1)+4*tvh(tvh_ndx(ii))+tvh(tvh_ndx(ii)-1))/6;
   svh(ii,1)=(noisyvh(tvh_ndx(ii)+1)+4*noisyvh(tvh_ndx(ii))+noisyvh(tvh_ndx(ii)-
1))/6;
 end
 stvh(m,1)=tvh(tvh_ndx(m));
 svh(m,1)=noisyvh(tvh_ndx(m));

 [m,n]=size(tz_ndx);
```

```
  stz(1,1)=tz(tz_ndx(1));
  sz(1,1)=noisyz(tz_ndx(1));
  for ii=2:m-1
    stz(ii,1)=(tz(tz_ndx(ii)+1)+4*tz(tz_ndx(ii))+tz(tz_ndx(ii)-1))/6;
    sz(ii,1)=(noisyz(tz_ndx(ii)+1)+4*noisyz(tz_ndx(ii))+noisyz(tz_ndx(ii)-1))/6;
  end
  stz(m,1)=tz(tz_ndx(m));
  sz(m,1)=noisyz(tz_ndx(m));

end
figure
subplot(2,1,1),plot(uft,sz-truez,uft,ufz-truez)
title('Error of Noisy Altitude and Horizontal Velocity')
ylabel('Altitude Noise (m)')
subplot(2,1,2),plot(uft,svh-truevh,uft,ufvh-truevh)
ylabel('Vh Noise (m/s)')
xlabel('Time Of Flight (s)')
figure
plot(ufx,sy-truey,ufx,ufy-truey)
title('Error of Noisy Horizontal Trajectory')
xlabel('X (m)')
ylabel('Y-noise (m)')
```

```
rng=sqrt(sx.^2+sy.^2);
[junk,known_xy_ndx]=min(abs(rng-rng_known));
[junk,known_tvh_ndx]=min(abs(stvh-t_known));
[junk,known_tz_ndx]=min(abs(stz-t_known));

sxy=[sx sy];
tsz=[stz sz];
tsvh=[stvh svh];



%--------------------------------------------------------------------------
%--- SECOND: Perform a line integration

%[rr,newyx,polyxrng]=ssfe(rngx);
%% first order numerical line integrator
%ltraj2(1,1)=0;
%for ii=2:101
%   ltraj2(ii,1)=ltraj2(ii-1,1)+sqrt((sx(ii)-sx(ii-1))^2+(sy(ii)-sy(ii-1))^2);
%end

% RK4-type line integrator
offset=0;
[ltraj,pq,theta]=li2(sxy,offset);
```

```
%------------------------------------------------------------------------
%--- THIRD: integrate horizontal velocity (closed form)

[st,newvh,polyvh]=ssfe(tsvh);
dh(1,1)=0;
[m,n]=size(polyvh);
for ii=2:m
   tempt(1:ii,1)=st(1:ii);
   tempp(1:ii,1:n)=polyvh(1:ii,1:n);
   dh(ii,1)=polyintegr8(tempt,tempp);
end


%------------------------------------------------------------------------
%--- FOURTH: correlate integrated velocity with line integration of position

% add the constant of integration from known point
deltas=ltraj(known_xy_ndx)-dh(known_tvh_ndx);
deltas=0;
newdh=dh+deltas;
```

```
% find time as a function of integrated velocity
[ndh,tofdhs,polytofdhs]=ssfe([newdh st]);

%figure
%plot(ndh,tofdhs)

% use the line integral vector to find a new time vector
[tofs,polytofs]=interpextrap(ndh,tofdhs,ltraj);



%-------------------------------------------------------------------------
%--- FIFTH: associate x and y with time

% find the functional relationship between rotated-x and rotated-y
% coordinates
[pp,qq,polyqofp]=ssfe(pq);

% calculate the rotated-x vector as a function of the line integral
[news,ppofs,polypofs]=ssfe([ltraj pp]);

% using the time vector associated with the line integral and the
% rotated-x vector associated with the same line integral; find
% the rotated-x as a function of time
```

```
[newtim,poft,polypoft]=ssfe([tofs ppofs]);


% use the time associated rotated-x vector to associate the rotated-y

% vector with time also

[qoft,polyqoft]=interpextrap(pp,qq,poft);




% find the original x and y coordinates by rotating the time associated

% rotated-x and rotated-y coordinates through -theta

oldx=poft;

oldy=qoft;

angl=-theta;

[m,n]=size(poft);

for ii=1:m

   xoft(ii,1)=cos(-angl)*oldx(ii)+sin(angl)*oldy(ii);

   yoft(ii,1)=sin(-angl)*oldx(ii)+cos(angl)*oldy(ii);

end




%-------------------------------------------------------------------------

%--- SIXTH: associate z with x and y and though use of their time vector


% find the functional relationship between z and time
```

```
[stz,newz,polyz]=ssfe(tsz);


% associate z with the same time vector used for other coordinates

[zoft,pz]=interpextrap(stz,newz,newtim);




%--------------------------------------------------------------------------

%--- SEVENTH: plot output and return data


% plot output data

[st,ndhofst,polyndhofst]=ssfe([st ndh]);

[ndhoft,polydhoft]=interpextrap(st,ndhofst,newtim);


%figure

%plot(newtim,ndhoft,newtim,ltraj)

%figure

%plot(ndhoft,xoft,ltraj,xoft)

%figure

%plot(newtim,abs(ndhoft-ltraj),newtim,ndhoft-ltraj)


[ltoft,pq2,theta2]=li2([poft qoft],offset);

%figure

%plot(newtim,abs(ndhoft-ltoft))
```

```
%title('Absolute Difference Between Line Integrated Position and Integrated Horizontal
Velocity')
%xlabel('Time Of Flight (s)')
%ylabel('Distance Estimate Difference (m)')
% get last variables ready to be returned
[ntx,xoft,px]=ssfe([newtim xoft]);
[nty,yoft,py]=ssfe([newtim yoft]);
dataout=[newtim,xoft,yoft,zoft];
ll_out=ndhoft;
%ll_out=ltraj;
```

```
%-----------------------------------------------------------------
%---
% ssfe2.m
%
% Author:   B. T. Yake
% Date:     25 June, 2002
% Rev:      2.0
%
% DESCRIPTION:
% Self Smoothing Functional Estimator:
% This algorithm creates a "rolling" average polynomial
% coeficient set over a moving 4 datapoint window. This
% routine assumes continuity between all points.
%
% USAGE:
% [xout,yout,polymat]=ssfe(data,mm,nn)
%    INPUT:  data    -> 2 column aray of sampled data
%            mm      -> polynomial order
%            nn      -> window size
%    OUTPUT: yout    -> Column vector of dependent variable
%                       based on smooth (polynomial) functional
%                       estimation
%            a       -> Array consisting of mm+1 columns.  The
%                    columns correspond to coefficients of a
%                    polynomials of the form:
%
%               Y = a(1)X^mm + a(2)X^(mm-1) +...+ a(mm)X + a(mm+1)
%
% NOTES:
%   1.0     02/19/2002 - Baseline
%   2.0     06/26/2002 - Generalized to handle analyses of data with
%                     a moving window of nn points and polynomial
%                     curve-fits of mm order. Inputs "mm" and
%                     "mm" are not currently available, but they
%                     can be set in the function explicitly.
%---
%-------------------------------------------------------------------


%function [yout,polymat]=ssfe2(data,mm,nn)
function [x,yout,a]=ssfe2(data) %,mm,nn)


mm=2;
nn=4;


% make sure window size is not too small
x=data(:,1);
```

```
y=data(:,2);


if nn<mm+1
   nn=mm+1;
end


[oo,pp]=size(data);


a=zeros(oo,mm+1);


for ii=1:oo
   if ii<nn
      den=ii;
      f=zeros(den,mm+1);
      for jj=den:-1:1
         ndx=jj;
         [f(jj,:),junk]=polyfit(x(ndx:ndx+nn-1),y(ndx:ndx+nn-1),mm);
      end
   elseif ii>oo-nn+1
      den=oo-ii+1;
      f=zeros(den,mm+1);
      for jj=den:-1:1
         ndx=ii-nn+jj;
      [f(jj,:),junk]=polyfit(x(ndx:ndx+nn-1),y(ndx:ndx+nn-1),mm);
    end
  else
    den=nn;
    f=zeros(den,mm+1);
    for jj=den:-1:1
      ndx=ii-nn+jj;
      [f(jj,:),junk]=polyfit(x(ndx:ndx+nn-1),y(ndx:ndx+nn-1),mm);
    end
  end
  for kk=1:mm+1
    for jj=1:den
      a(ii,kk)=a(ii,kk)+f(jj,kk)/den;
    end
  end
  clear f
  yout(ii,1)=polyval(a(ii,:),x(ii));
end
```

```
%--------------------------------------------------------------
%---
%   FUNCTION: derivs.m
%
%   AUTHOR:   B. T. Yake
%   DATE:     11/30/2001
%   REV:      v1.0
%
%   USAGE:
%   [ydots, polydots] = derivs(x,funcx)
%    inputs:  x        -> Independent variable vector (column)
%             funcx    -> Array containing the polynomial
%                         coefficients for the curve.
%    outputs: ydots    -> array of funcx derivatives of x.
%                         (first deriv. in first column, second in
%                         second, and so on... the number of
%                         derivatives depends on the order of
%                         funcx... i.e. a second order function
%                         will have two derivitives.)
%             polydots -> array of polynomials coefficients that
%                         correspond to ydots... the first layer to
%                         the first derivative, second to second
%                         and so on
%   PURPOSE:
%   This function calculates the derivatives of polynomial arrays
%   that are passed to it with it's independent variable.  The
%   number of derivatives calculated at each point is equal to the
%   degree of the polynomial passed to it (i.e. a second order
%   polynomial will have three coefficients but only the first and
%   second derivatives will be calculated for it.)  Each order
%   derivative is returned in the corresponding column of the output
%   variable "ydots" (i.e. 1st derivative in column 1, 2nd in column
%   2, etc...)
%
%   NOTES:
%   v1.0 - Baseline
%---
%-----------------------------------------------------------------
function [ydots, polydots]=derivs(x,funcx)


[frows,fcols]=size(funcx);
[xrows,xcols]=size(x);


if xrows==frows
    tempf=funcx;
    polydots=zeros(frows,fcols-1,fcols-1);
    for kk=1:fcols-1
        clear tf;
        tf=tempf(:,1:fcols-kk);
        [trows,tcols]=size(tf);
        for ii=1:xrows
            tempydot=0.0;
            for jj=1:tcols
                tf(ii,jj)=tf(ii,jj)*(tcols+1-jj);
                tempydot=tempydot+tf(ii,jj)*x(ii)^(tcols-jj);
            end
            ydots(ii,kk)=tempydot;
            for jj=1:tcols
                mm=fcols-1+jj-tcols;
                polydots(ii,mm,kk)=tf(ii,jj);
            end
        end
        clear tempf;
        tempf=tf;
    end
end
```

```
%---------------------------------------------------------------
%---
% polyintegr8.m
%
% Author:   B. T. Yake
% Date:     05 May, 2002
% Rev:      1.0
%
% DESCRIPTION:
% This function performs a closed-form point-to-point integration
% of a polynomial function.
% The inputs are:
%   xx         ->  X-axis vector.
%   polyin     ->  A matrix where each row consists of the coefficients
%                  that correspond to the appropriate polynomial at that
%                  point.  The first column contains the highest order
%                  coefficients of the polynomials, and the last column
%                  contains the zero-th order coefficient.
% The output is:
%   integsum   ->  The result of the sumation of the point-to-point
%                  integrations for each interval in xx (from xx(1:2)
%                  to xx(nn-1:nn))
%
```

```
% NOTE:
%   1.0     11/26/2001 – Baseline
%---
%--------------------------------------------------------------



function [integsum]=polyintegr8(xx,polyin)


[prows, pcols]=size(polyin);
[xrows, xcols]=size(xx);
integsum = 0.0;
if prows>1 & prows==xrows
   for ii=1:prows-1
      for jj = 1:pcols
         ndx=pcols+1-jj;
         mm(jj) = (polyin(ii+1,jj)-polyin(ii,jj))/(xx(ii+1)-xx(ii));
         bb(jj) = polyin(ii,jj) - mm(jj)*xx(ii);
      end
      for kk=1:pcols
         integ1 = mm(kk)*(xx(ii+1)^(pcols-kk+2)-xx(ii)^(pcols-kk+2))/(pcols-kk+2);
         integ2 = bb(kk)*(xx(ii+1)^(pcols-kk+1)-xx(ii)^(pcols-kk+1))/(pcols-kk+1);
         integsum = integsum+integ1+integ2;
     end
```

```
 end
end
```

```
%-----------------------------------------------------------------------
%---
%   FUNCTION: line_integ.m
%
%   AUTHOR:   B. T. Yake
%   DATE:     5/5/2002
%   REV:      v1.0
%
%   USAGE:
%   [integ_vals,pq,angl]=li2(xy,tim_vh,tim_z)
%    inputs:  xy          -> Two column vector containing the independent
%                          and dependent variable values respectively.
%                          X and Y position data for the entire flight
%           offset      -> Constant of integraion.
%    outputs: integ_vals  -> Column vector containing the line integrated
%                          distance along the curve with constant offset.
%           pq          -> equivelent to the xy input variable (matrix)
%                          with the exception that if the data needed to
%                          be rotated to allow y to become a function of
%                          x, then this variabe contains the rotated
%                          xy information.
%           angl        -> the angle xy is rotated through (0 or a
%                          multiple of 15 degrees)
%   PURPOSE:
%   This routine performs a line integration of %   and reconstruct the
%   full three-dimenstional filght-path.
%---
%-------------------------------------------------------------------------



function [integ_vals,pq,angl]=li2(xy,offset);




[m,n]=size(xy);
xx=xy(:,1);
yy=xy(:,2);

%-------------------------------------------------------------------------
%--- Attempt to guarentee that y is a function of x

kk=0;
angl=0;
flag=1;
while flag==1
   kk=kk+1;
   flag=0;
```

```
for ii=2:m-1
  if flag==0
    if xx(ii)>xx(ii-1)
       flag=0;  % y is still a function of x
    else
      if xx(ii+1)>xx(ii)
         flag=0;  % continue search for last functional-x index
      else
        flag=1;  % NOT FUNCTIONAL
      end
    end
  end
end
if flag==1
  oldx=xx;
  oldy=yy;
  deg=kk*15;
  angl=deg*pi/180;
  for ii=1:m
    xx(ii,1)=cos(-angl)*oldx(ii)+sin(angl)*oldy(ii);
    yy(ii,1)=sin(-angl)*oldx(ii)+cos(angl)*oldy(ii);
  end
end
```

```
end
pq=[xx yy];

%figure
%plot(xx,yy)
%title('Rotated Horizontal Trajectory')
%xlabel('Rotated X (m)')
%ylabel('Rotated Y (m)')
%stop %%%

%----------------------------------------------------------------------
%--- Estimate line integral for entire region

% find y as a function of x and representative polynomials
[x2,newyofx,polyyofx]=ssfe(pq);
[blah,coefs]=size(polyyofx);
% Calculate coefficients of dy/dx
for ii=1:m
  pwr=1;
  if coefs>1
    for jj=coefs-1:-1:1
      yd(ii,jj)=polyyofx(ii,jj)*pwr;
      pwr=pwr+1;
    end
  end
end

% set integration IC = 0
ll(1,1)=0;

% Variant of RK4 integration
for ii=2:m
  deltax=x2(ii)-x2(ii-1);
  dy1=polyval(yd(ii-1,:),x2(ii-1))*deltax;
  dy2=polyval((yd(ii-1,:)+yd(ii,:))/2,(x2(ii-1)+x2(ii))/2)*deltax;
  dy3=polyval(yd(ii,:),x2(ii))*deltax;
  dybar=(dy1+4*dy2+dy3)/6;
  deltar=sqrt(dybar^2+deltax^2);
  ll(ii,1)=ll(ii-1,1)+deltar;
end

% Add integration IC
integ_vals=ll+offset;
```

```
%--------------------------------------------------------------
%---
%   FUNCTION: interpextrap.m
%
%   AUTHOR:   B. T. Yake
%   DATE:     5/1/2002
%   REV:      v1.0
%
%   USAGE:
%   [y2,p2]=interpextrap(x1,y1,x2)
%    inputs:  x1 -> Initial independent variable vector
%             y1 -> Initial dependent variable vector
%             x2 -> Newly chosen dependent variable vector
%
%    outputs: y2 -> Dependent variable vector that corresponds
%                   to the new independent variable vector, x2
%             p2 -> Array of polynomial (second order) coefficients
%                   that would produce y2 at x2.
%
%   PURPOSE:
%   This function performs interpolation and extrapolation based on
%   the SSFE approximation of data vector y1 to independent variable
%   vector x1 to produce a new data vector y2 that matches the newly
%   chosen independent variable vector x2.  The array of polynomial
%   coefficients corresponding to y2 is also provided as output p2.
%
%   NOTES:
%   v1.0 - Baseline
%---
%-----------------------------------------------------------------


function [y2,p2]=interpextrap(x1,y1,x2)


[m,blah]=size(x1);
[newm,blah]=size(x2);
[xx,yofx,p1]=ssfe([x1 y1]);
[blah,n]=size(p1);
for ii=1:newm
   [junk,ndx]=min(abs(xx-x2(ii)));
   if junk>0  % new x is not equal to an existing x
      r=x2(ii)-xx(ndx);
      if r>0  % new x is greater than closest old x
         if ndx==m  % new x is greater than all old x... extrapolate
            p2(ii,1:n)=p1(m,1:n);
         else      % interpolate
            delt=xx(ndx+1)-xx(ndx);
```

```
        for jj=1:n
          p2(ii,jj)=(p1(ndx+1,jj)-p1(ndx,jj))*r/delt+p1(ndx,jj);
        end
      end
    else    % new x is less than closest x
      if ndx==1  % new x is less than all old x... extrapolate
        p2(ii,1:n)=p1(1,1:n);
      else      % interpolate
        delt=xx(ndx)-xx(ndx-1);
        fac=(delt+r)/delt;
        for jj=1:n
          p2(ii,jj)=(p1(ndx,jj)-p1(ndx-1,jj))*fac+p1(ndx-1,jj);
        end
      end
    end
  else       % new x is equal to an existing x
    p2(ii,1:n)=p1(ndx,1:n);
  end
  y2(ii,1)=polyval(p2(ii,:),x2(ii));
end
```

```
%--------------------------------------------------------------
%---
%   FUNCTION: pickstep.m
%
%   AUTHOR:   B. T. Yake
%   DATE:     5/1/2002
%   REV:      v1.0
%
%   USAGE:
%   stp=pickstp(xx)
%    inputs:  xx  -> scalar value indicating the length of the data
%                  vector to be analyzed
%
%    outputs: stp -> scalar value indicating how often to sample
%                  the data in question.  (i.e. stp = 5 means
%                  that every fifth data point will be sampled.
%
%   PURPOSE:
%   This small function automates the selection of points for analysis.
%
%   NOTES:
%   v1.0 - Baseline
```

```
%---

%-----------------------------------------------------------------

function stp=pickstp(xx)


stp=1;

if xx>=10000

   stp=100;

elseif xx<10000 & xx>=5000

   stp=50;

elseif xx<5000 & xx>=2000

   stp=20;

elseif xx<2000 & xx>=1000

   stp=10;

elseif xx<1000 & xx>=500

   stp=5;

elseif xx<500 & xx>=200

   stp=4;

elseif xx<200 & xx>=100

   stp=2;

end
```

```
%---------------------------------------------------------------
%---
% FUNCTION: picked_pts.m
%
% AUTHOR:   B. T. Yake
% DATE:     5/1/2002
% REV:      v1.0
%
% USAGE:
% [x_ndx,tvh_ndx,tz_ndx]=picked_pts(set)
%  inputs:  set     -> scalar value to select between sets of
%                      points chosen by the user (if available).
%
%  outputs: x_ndk   -> vector of index values corresponding to
%                      the index values of points of interest for
%                      the noisey XY matrix.
%           tvh_ndx -> vector of index values corresponding to
%                      the index values of points of interest for
%                      the noisey TVH matrix.
%           tz_ndx  -> vector of index values corresponding to
%                      the index values of points of interest for
%                      the noisey TZ matrix.
% PURPOSE:
```

```
%   This small function allows the user to select the points on the
%   X -vs- Y, T -vs-Vh, and T -vs- Z plots.  If multiple sets of
%   points are to be considered, they can be chosen using the
%   “set” variable.
%
%   NOTES:
%   v1.0 - At this time, only one set of indices have been selected
%          so the set variable is, at this time, a dummy.  The
%          switching logic that uses “set” has been commented out.
%---
%----------------------------------------------------------------function
[x_ndx,tvh_ndx,tz_ndx]=picked_pts(set)


%switch sets
%case 1
x_ndx=[1;51;101;151;201;241;251;261;271;281;291;301;311;321;331;341;361;
    381;401;421;441;461;481;501;521;541;551;561;571;581;591;601;611;
    621;631;641;651;661;671;681;701;711;721;731;741;751;761;771;781;
    801;821;841;861;881;901;931;961;1001];


tvh_ndx=[1;51;101;151;201;241;251;261;271;281;291;301;311;321;331;341;
    351;361;381;401;421;441;461;481;501;521;541;551;561;571;581;591;
    601;611;621;631;641;651;661;671;681;701;711;721;731;741;751;761;
    771;781;801;821;841;861;881;901;931;961;1001];


tz_ndx=[1;51;101;151;201;221;241;251;261;271;281;291;301;311;321;331;
    341;351;361;381;401;421;441;461;481;501;521;541;551;561;571;581;
    591;601;611;621;631;641;651;661;671;681;701;711;721;731;741;751;
    761;771;781;801;821;841;861;881;901;931;961;1001];
%case 2
%otherwise
%end
```

References

1. Borse, G. J., *Numerical Methods with MATLAB, A resource for Scientists and Engineers*, PWS Publishing Company, Boston, MA, 1997

2. Hamming, R. W., *Numerical Methods for Scientists and Engineers, 2nd edition,* Dover Publications, Inc. New York, NY., 1973

3. Bryson, A. E. Jr., Ho, Y., *Applied Optimal Control, Optimization, Estimation, and Control, Revised Printing*, Hemisphere Publishing Corp., 1975

4. Gelb, A., *Applied Optimal Estimation*, The M.I.T. Press, Cambridge, MA, 1989

5. Berkey, D. D., *CALCULUS, 2nd edition*, Saunders College Publishing, New York, NY, 1988

6. Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P., *Numerical Recipes in FORTRAN 77: The Art of Scientific Computing, 2nd edition,* Caimbridge University Press, New York, NY, 1992

7. Beyer, W. H, editor, *CRC Standard Mathematical Tables and Formulae, 29th edition*, CRC Press, Inc., Boca Raton, FL, 1991

8. Thompson, W. T., *Theory of Vibration with Applications, 4th edition*, Prentice Hall, Englewood Cliffs, NJ, 1993

Filename:	BTY_thesis10
Directory:	I:
Template:	C:\Program Files\Microsoft Office\Office\Normal.dot
Title:	Self-Smoothing Functional Estimation
Subject:	
Author:	BYake
Keywords:	
Comments:	
Creation Date:	10/24/02 8:45 AM
Change Number:	2
Last Saved On:	10/24/02 8:45 AM
Last Saved By:	BYake
Total Editing Time:	0 Minutes
Last Printed On:	10/24/02 8:45 AM
As of Last Complete Printing	
Number of Pages:	114
Number of Words:	11,922 (approx.)
Number of Characters:	67,960 (approx.)