ABSTRACT

Robust Geolocation Techniques for Multiple Receiver Systems

Gregory W. Fisher, M.S

Mentor: Michael W. Thompson, Ph.D.

The purpose of this thesis is to investigate signal processing algorithms that allow multiple moving receivers to locate a stationary emitter. This problem has received considerable attention over the past 50 years, yet advances in computational power, sensor technologies and increasingly complex battle space scenarios continue to drive interest in this area. This work focuses on implementing well-known least squares and Kalman based algorithms within a realistic three dimensional simulation model. Techniques for evaluating the performance of various algorithms include generating ellipse-shaped confidence regions that bound the target under consideration, along with generating polygon shaped confidence regions based on intersecting regions from multiple receivers. The presence of outlier angle of arrival measurements is shown to significantly degrade the performance of geolocation algorithms. Methods for imparting robustness to outlier angle of arrival measurements are developed and shown to mitigate the corresponding loss in performance that would otherwise occur.

Robust Geolocation Techniques for Multiple Receiver Systems

by

Gregory W. Fisher, B.S.

A Thesis

Approved by the Department of Electrical and Computer Engineering

_____
Kwang Y. Lee, Ph.D. , Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

Approved by the Thesis Committee

_____
Michael W. Thompson, Ph.D., Chairperson

_____
Russell W. Duren, Ph.D.

_____
William Poucher, Ph.D.

Accepted by the Graduate School
May 2011

_____
J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Dr. Thompson and Dr. Duren for giving me the opportunity to come to Baylor University and work on this project as a graduate student. The support they have given me has served to enhance my knowledge and experience and to expand my personal and professional contacts and opportunities. I would also like to thank the rest of the Electrical Engineering faculty at Baylor University who have worked to provide me, as well as every other student, an outstanding education.

CHAPTER ONE

Introduction

Geolocation

Geolocation, for the purposes of this project, as well as what is widely defined by those who do research in that field, refers to a method of locating an object or set of objects relative to one's own position and mapping that result to an absolute position in a known coordinate system. The most widely used coordinate system is latitude, longitude, and elevation used in global positioning systems (GPS). In military applications, geolocation usually refers to methods of determining the positions of one or many signal emitters, such as radar stations, by one or many aircraft or unmanned aerial vehicles (UAV) [**1**] [**2**]. In order to do this, the aircraft or UAVs have antenna arrays distributed about their frames from which measurements of either or both signal distance or direction can be taken [**3**]. Utilizing GPS measurements, we can accurately know the position of the receiver, which therefore allows us to closely estimate the position of the emitter. GPS systems also have non-military applications, though the commercial GPS units are less accurate. This is mostly due to the quality of the components utilized. In the past, government restrictions placed on civilian GPS usage also affected its accuracy, though in recent years this is no longer true. Commercial GPS is sufficient for many activities such as mapmaking, cellular telephone integration, and recreational applications.

There are three primary techniques that can be used for military geolocation applications: Direction of Arrival (DOA), Time Difference of Arrival (TDOA), and

Frequency Difference of Arrival (FDOA). These methods can be used independently or in combination with any other method for geolocation. These methods have been the basis for many studies on geolocation methods, such as the ones presented in [**4**] [**5**] [**6**] [**7**] [**2**].

Time Difference of Arrival

The Time Difference of Arrival method attempts to estimate a position based on the difference in time between measurements at two different locations. Specifically, this method refers to the estimation of arrival time of a signal transmitted from an emitter to multiple sensors. These sensors can be on the same platform or on two different platforms. Complete geolocation using only this method must be accomplished using either three or more receivers, or by multiple measurements from two receivers [**1**]. However, by utilizing this method in combination with an additional method, accurate results can be generated at the expense of additional computational power requirements [**7**].

Because of the nature of the measurements, the relative positions of the observers taking the measurements can be a large factor in the accuracy of these measurements. Optimal geometry for this method would assume that each combination of receivers would be at right angles with each other, with the vertex of the angle being the emitter. Deviation from this pattern can produce a sub-optimal geometric pattern that could result in a greatly distorted ellipse of probability, thus affecting the quality of the estimate. The least optimal geometry would be the case where the two receivers were in a straight line with each other and the emitter. This would produce a very long and narrow ellipse of probability [**8**].

This method also assumes that the data that is being shared can be synchronized in time, so that an accurate time difference can be obtained. Very accurate clock synchronization between multiple observers is absolutely essential for this method to work. At the speed at which the aircraft would be flying, any loss of synchronization between clock sources would introduce error in the time difference measurements. Such loss of synchronization would be a result of imperfections or degradations in the local clock oscillators, as well as by any noise in the transmission of synchronization data in the case where a clock can be synchronized remotely [**8**].

Direction of Arrival

Direction of Arrival methods, also called Angle of Arrival (AOA), attempt to estimate a position based on comparing multiple readings of the direction that the signal came from. The most common method of obtaining a DOA reading utilizes the TDOA measurements of individual antennas in a single antenna array to determine the most likely direction that a signal arrived from. However, this method usually involves a larger antenna array than is practical for a single small vehicle. In this case, a simpler and older method is used, which involves rotating an antenna to locate the direction from which a signal provides the largest amplitude response [**9**].

From these readings, several algorithms can be applied to find the region of maximum likelihood of the location of the emitter. Just as in TDOA measurements, DOA methods can be applied to a multiple receiver system, as well as a single receiver system that takes multiple measurements over a period of time. If a direction vector can accurately be estimated a number of times over a certain flight path, the intersection of those vectors would give an accurate estimate of the position of the emitter. DOA

measurements are inherently noisy and require processing in order to produce a sufficiently accurate location estimate

Because DOA readings are derived from TDOA readings, the same limitations and reliance on geometry are present. However, the restrictions on clock synchronization between multiple remote receivers are not present because the TDOA reading is performed locally on a single array with either a single antenna or multiple antennas. This greatly increases the functionality and, in many cases, the robustness of the method.

Frequency Difference of Arrival

Frequency Difference of Arrival methods estimate a position based on the difference in frequency between the measurements of the received signal, taken at two different locations. This frequency difference is a result of the Doppler shift that occurs when a moving object observes a carrier wave. This method generally assumes a stationary emitter. As with TDOA methods, FDOA methods utilize multiple observers which share data, or a single observer which takes multiple readings. Complete geolocation can be calculated using measurements from two observers, or by being combined with the results of other methods. Typically, TDOA and FDOA methods are utilized in combination with each other to increase the accuracy of the results [**8**] [**10**]. If the two methods are combined, the TDOA reading represents the bearing data, while the FDOA reading represents the distance measurement [**8**] [**7**].

Similar to TDOA methods, the relative positions of the observers are a factor in the accuracy of the measurements, though this does play a much larger factor in the resulting readings than it does in TDOA methods. In addition to this, the velocities of the observers, both magnitude and direction, also factor in to the measurements. This is due

to the fact that instead of a probability ellipse, the FDOA method results in a line of constant frequency that the emitter could rest on that is calculated from the Doppler shifts of the received frequency measurements. Any change in relative velocity alters that Doppler shift and thus alters the line of constant frequency that is computed [**8**] [**8**].

Unlike the TDOA method, complete clock synchronization is not a requirement, as time is not a factor in the measurements. That being said, if FDOA is used in combination with TDOA, clock synchronization is still a strict requirement.

Our Project

For the purposes of our experiments, we utilize Angle of Arrival measurements as the input to her algorithms. Several variations of this method are described in the next section. One method is to estimate the emitter's location in three dimensions, assuming that the observer knows nothing about the surrounding environment. While this is a good general-purpose method, for most cases it can safely be assumed that the observer does in fact have data about the surrounding location. Therefore, the second method simplifies the three-dimensional case by reducing the estimation calculation to two dimensions and then projecting that result onto a known elevation or other GPS mapping data.

Though most of the methods we utilize are based on currently existing techniques, the focus of our study is to come up with more robust methods of reducing the error inherent in the system. For all of the above techniques, the main result is not a specific point of estimation. Rather, it is an ellipse that encompasses a certain degree of error, or to be more specific, an ellipse in which the emitter might be located with a reasonable degree of certainty. There are many factors that affect the shape and volume of that ellipse, such as relative position in the case of DOA and TDOA, and position and

velocity, in the case of FDOA [7]. There are things that can be done to reduce the size of that error ellipse while still maximizing the inclusion of the true emitter location within that ellipse.

It should be noted that while combining the results of two different measurement and calculation types might achieve optimal results, the hardware specifications of the aircraft must be taken into consideration. For our methods, we are attempting to maximize our results while not overtaxing the hardware present in comparatively older, yet not obsolete, aircraft. Therefore, one of the main focuses of our experiments was reducing calculation time and complexity. Additionally, because such aircraft may not be able to synchronize the clocks on their hardware from a distance, TDOA methods would not be optimal for our situation.

One of the methods that is used in our simulation is the concept of overlapping error estimates. Our conjecture is that if the error ellipses of two sets of quality measurements are overlapped, the result will be more accurate than either of the two ellipses separately. This should also reduce some of the error bias that results from the positioning of the aircraft that are taking measurements.

Two methods that we will be using to combine the recorded data into estimates are a block processing method and an iterative method. The block processing method that we use calculates a result based on a block of data that has already been obtained, such as the most recent 50 or 100 readings. The idea is that as more data is collected, the estimate will be increasingly accurate. Additionally, as will be explained below, any errors in the data will eventually fall out of the data block as new data is brought in. The

block processing method that is being utilized in this process is the windowed least-squares method.

The iterative method utilizes recursion to conserve storage space. Using this method, previous data will not need to be retained, as the estimate is fed back through the algorithm and combined with the next reading. The drawback to this method is that an iterative algorithm must be initialized well in order to work properly, and a poor initialization or a large block of bad data will render the process unrecoverable. The iterative process utilized as an alternative to the least-squares method in this study is the Kalman filtering method.

Real-World Error Sources

Physical Sources of Error

In a real-world situation, most errors that are encountered can be explained through the observance of physical considerations. A receiver generally observes the signal from the emitter at multiple points either simultaneously or in sequence, and from those observed measurements a line of bearing is determined. These lines of bearing have varying degrees of accuracy depending on the position of the receiver relative to the emitter as well as the accuracy of the receiver equipment.

One of the main causes of geometric position-based errors occurs when the emitter is located at a great distance from the receiver array. In any situation where the line-of-bearing measurements are nearly parallel, the effect of any other errors on the system is magnified. This is the primary reason that our simulated noise model scales with distance, as discussed later in this document.

System Errors

There are several sources of error internal to any radar receiver system that one must account for. The most common type of error that affects any hardware system is the issue of calibration. The effects of temperature, weather conditions, and physical wear and tear over time can all cause significant alterations in the results that a hardware system obtains. In general, these errors can be accounted for through maintenance and proper calibration of the system, though sudden variations in the ambient surroundings can cause calibration errors in a short amount of time.

Another issue that arises is the stability of the receiver array, specifically in turbulent environments. Small variations in the stability can cause a certain degree of noise in any line of bearing measurement that depends on precision with regards to position. Additionally, the angle at which the array is positioned relative to horizontal will affect the incoming measurements. This would be an issue that arises primarily in the case of the aircraft changing direction or performing similar maneuvers.

Interference from other sources can also contribute to uncertainty in the signals measured by the receiver. It can be assumed that any mission environment would have multiple emitters with varying signal strength. The tracking of one specific signal can be momentarily interrupted by interference from a higher-strength signal, or even a lower-strength one that resonates with the surrounding signals. Any receiver and geolocation system must be prepared to account for errors and inconsistencies such as these [**3**].

Random Errors

In addition to the sources of physical error described above, there may be other error sources that are not known about or specifically accounted for. In simulation, these errors can be modeled using random number generators, and the algorithm can be developed such that even a new or previously unknown error can be accounted for if its behavior is similar to other sources of error that are known. In general, the most prominent sources of error present in line of bearing measurements have already been well-documented and therefore a receiver system with robust error correction methods is desirable in order to deal with these types of random errors that appear in measurements [**11**] [**5**] [**3**].

CHAPTER TWO

Algorithms

Simulation Model

Preparation of the Path

To show the operation of this algorithm, a MATLAB simulation has been created which implements the method multiple times along the same path taken by the moving receiver. The first step in the simulation is to determine where the emitter lies and what path the receiver takes. For the purposes of this simulation, the emitter is shown to lie at the Cartesian coordinates [1, 1, 1]. This is a simple example to test the correct operation of the algorithm, though any point in space can be used. The path can either be given directly by editing the m-file, or by calling the function [xx yy zz] = addPlot(). The latter method allows the user to input the path graphically using the built in MATLAB function ginput(). After the path is specified, it is then scaled by a parameter that can be changed to obtain the desired distance range and then interpolated to obtain the desired number of flight path data points. The resolution of this interpolation can also be changed by the user at the start of the m-file. After obtaining the necessary data slices along the path, a time vector is created and the velocity differentials are calculated based on the equation:

$$dv = \frac{x_i - x_{i-1}}{t_i - t_{i-1}} \qquad (1)$$

Note that the initial velocity of the receiver must be set by the user at the beginning of the m-file. The default value is 0. Also note that it would be relatively easy

to perform these calculations in reverse, starting with a user-generated velocity curve and from there obtaining the path of the receiver.

Preparation of the input signal

In the real world, the geolocation algorithms would operate on the data acquired by the receiver. However, for the purposes of this MATLAB simulation, the user must generate a signal that approximates a real-world situation. In order to generate this signal, the simulation uses as reference the mathematical 'perfect case' as a reference. Thus the vector B that points from the receiver to the emitter is nothing more than the difference between the receiver's position and the emitter's position, or more simply:

$$B = S - R \qquad (2)$$

Noise is then added to the vector B to simulate noisy mesurements. This is accomplished through the use of the MATLAB randn() function, which generates a pseudo-random value using a standard Gaussian distribution. This random value is then used in the equation:

$$N_x = A|B_x|^2 \qquad (3)$$

This equation calculates a random noise value, N_x, based on the square of the distance away from the emitter that the receiver is, as well as by a noise scale parameter A that can be chosen by the user to approximate a real-world noise distribution. This calculation is repeated to obtain noise values N_y and N_z, for the y and z components, respectively.

The presence of error in the angle of the received signal is also accounted for in the preparation of the simulated input signal. The directional angular components are modeled through the directional cosine method as follows:

$$\theta = \arccos\left(\frac{B_x}{|B|}\right) \qquad (4)$$

In this case, θ is the angle between the R vector and the Cartesian x-axis. This calculation is repeated twice more to obtain ρ, the angle between the R vector and the y-axis, and ϕ, the angle between the R vector and the z-axis.

Once these angles are calculated, angular variance can then be inserted through the use of another random process based on an error parameter, ε, also chosen by the user to approximate a real world situation. For this process, a random value is drawn from a uniform distribution on the range [0,1] using the rand() function in MATLAB. If the random value is less than ε noise is added to the angular components. The value for the noise is drawn from a uniform random distribution on the range [-π/2, π/2]. After this noise is added, the component vectors are recalculated using the inverse of the directional cosine equation:

$$B_{xnoise} = |B| \cos\theta \qquad (5)$$

After summing the noise, the vectors are then normalized to remove any distance component from the incoming signal, as in a real world situation the receiver would know nothing of the distance away that the emitter is, only the direction of the incoming signal. The normalization method is as follows:

$$B_n = \frac{Bynoise}{\sqrt{B_{xnoise}^2 + B_{ynoise}^2}} - \frac{Bxnoise}{\sqrt{B_{xnoise}^2 + B_{ynoise}^2}} \qquad (6)$$

Least-Squares Estimator

Estimating the Emitter's Location in Two Dimensions

In order to estimate the location of the emitter in two dimensions, a matrix, H, must be defined as the transpose of the normalized B vector:

$$H = B_n^T \quad (7)$$

The H matrix is related to the true emitter position by the following relation:

$$Z = HX + noise \quad (8)$$

In this equation, X is the true emitter position. The presence of noise makes an exact estimation of X impossible, as the noise cannot be known. However, the least-squares estimator operates by obtaining a solution for X that minimizes the distance between HX and Z [**9**]. This relationship can be shown by representing it as a cost function, J.

$$J = |HX - Z|^2 \quad (9)$$

The minimum of this cost function is obtained by setting the gradient to zero, as shown.

$$\nabla\{|HX - Z|^2\} = 0 \quad (10)$$

This expression is then expanded and the gradient computed.

$$\nabla[(HX - Z)^T(HX - Z)] = 0 \quad (11)$$

$$2X^T H^T H - 2Z^T H = 0 \quad (12)$$

The least-squares solution X can therefore be defined [**9**].

$$X = (H^T H)^{-1} H^T Z \quad (13)$$

From here, the matrix $H^{\#}$ which is the pseudoinverse of H, can be defined as follows [**9**]:

$$H^{\#} = (H^T H)^{-1} H^T \quad (14)$$

Additionally, the vector Z must be defined as the following summation, with N equal to the number of total data points in the simulation:

$$Z = \sum_{k=0}^{N} B_{nk} R_k \quad (15)$$

This once again makes sure that any residual distance component is removed from the incoming signal. The resulting Z vector is then transposed so that the subsequent calculations are possible. From here, a point X is calculated using the equation:

$$X = H^{\#}Z \qquad (16)$$

This point X is the estimate that the algorithm has made as to the position of the emitter. It should be noted that MATLAB gives an error if this equation is used, as the matrix H'H is oftentimes very close to singular. In practice, a pseudoinverse function pinv(H) can be used as a more numerically stable method of calculating $H^{\#}$ [**9**].

Estimating the Emitter's Location in Three Dimensions

The above procedure can be expanded to three dimensions by altering the method of normalization. In the two-dimensional case, the distance component is removed by finding an orthonormal vector to the B vector. This is expanded into three dimensions by computing two orthonormal vectors and using those to remove the distance component of the B vector. Since in three dimensions the orthonormal vectors can point in any direction as long as they are normal to the B vector, the vector B_{init} below can be chosen arbitrarily:

$$B_{P1} = [\, -B_{initX} \quad -B_{initY} \quad \frac{B_{initX}B_x + B_{initY}B_y}{B_z} \,] \qquad (17)$$

Once the first orthonormal vector is calculated, the second vector, which must be orthonormal to both previous vectors, can be computed by taking the cross product of the vectors.

$$B_{Pa} = B \times B_{P1} \qquad (18)$$

The vectors are then normalized to their unit vector equivalents:

$$\hat{B}_{P1} = \frac{B_{P1}}{|B_{P1}|} \qquad (19)$$

$$\hat{B}_{Pa} = \frac{B_{Pa}}{|B_{Pa}|}$$

These vectors are used to construct the H matrix, as shown:

$$H = \begin{bmatrix} \hat{B}_{P1}{}^{T} \\ \hat{B}_{Pa}{}^{T} \\ \dots \\ \hat{B}_{PN}{}^{T} \\ \hat{B}_{PM}{}^{T} \end{bmatrix} \qquad (20)$$

Additionally, the Z matrix is constructed using the dot products of the orthonormal vectors and the initial bearing vectors:

$$Z = \begin{bmatrix} \hat{B}_{P1} \cdot R_1 \\ \hat{B}_{Pa} \cdot R_1 \\ \dots \\ \hat{B}_{PN} \cdot R_N \\ \hat{B}_{PM} \cdot R_N \end{bmatrix} \qquad (21)$$

The final calculation is performed the same way as it is in the two-dimensional case:

$$X = H^{\#}Z \qquad (22)$$

Numerical Example

Suppose that in a noise-free example we assume an emitter position S.

$$S = [1 \quad 1 \quad 1] \qquad (23)$$

And create the path vector R such that:

$$R = \begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} = \begin{bmatrix} 2 & 3 & 4 \\ -1 & -2 & -3 \\ 0 & 1 & 2 \end{bmatrix} \qquad (24)$$

From here, we calculate the B vector as follows, with the S matrix transposed to fit our MATLAB example:

$$B = S^T - R = \begin{bmatrix} -1 & -2 & -3 \\ 2 & 3 & 4 \\ 1 & 0 & -1 \end{bmatrix} \tag{25}$$

At this point, as this is a noise-free example, we can compute the two orthonormal vectors for each point of data:

$$\hat{B}_{P1} = \begin{bmatrix} -0.1592 \\ -0.7961 \\ 0.5838 \end{bmatrix} \tag{26}$$

$$\hat{B}_{Pa} = \begin{bmatrix} -0.9504 \\ 0.2837 \\ 0.1277 \end{bmatrix} \tag{27}$$

$$\hat{B}_{P2} = \begin{bmatrix} -0.1506 \\ -0.7532 \\ 0.6403 \end{bmatrix} \tag{28}$$

$$\hat{B}_{Pb} = \begin{bmatrix} 0.9162 \\ -0.3497 \\ -0.1958 \end{bmatrix} \tag{29}$$

$$\hat{B}_{P3} = \begin{bmatrix} -0.1925 \\ -0.9623 \\ -0.1925 \end{bmatrix} \tag{30}$$

$$\hat{B}_{Pc} = \begin{bmatrix} -0.6804 \\ 0.2722 \\ -0.6804 \end{bmatrix} \tag{31}$$

Therefore, the H matrix is:

$$H = \begin{bmatrix} -0.1592 & -0.7961 & 0.5838 \\ -0.9504 & 0.2837 & 0.1277 \\ -0.1506 & -0.7532 & 0.6403 \\ 0.9162 & -0.3497 & -0.1958 \\ -0.1925 & -0.9623 & -0.1925 \\ -0.6804 & 0.2722 & -0.6804 \end{bmatrix} \tag{32}$$

And the Z matrix is calculated as shown above:

$$Z = \begin{bmatrix} -0.3715 \\ -0.5390 \\ -0.2636 \\ 0.3707 \\ -1.3472 \\ -1.0887 \end{bmatrix} \qquad (33)$$

And the final result is:

$$X = H^{\#}Z = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad (34)$$

As noted before, this estimate is found in the absence of noise, and so the answer we get is the exact location of the emitter. In a real-world situation, the estimate that this method obtains would be close to, but not exactly, the true emitter location.

Surface Bearing Projections

While the above three-dimensional example can be adapted to any three-dimensional space, in a real-world situation there will be certain assumptions that will be made about the emitter. One such assumption could be that it is located on the surface of the earth, and therefore the elevation of that surface will be known to the system in advance. Therefore, it makes sense to simplify the calculations by taking into account the known surface and projecting the emitter location estimate onto that surface.

The approach to this method is similar, though there are a couple of intermediate steps that are added into the process. Additionally, the process assumes that for each estimate, there is some constant value, Z_0, that represents the known elevation of the surface of the earth. Starting from the conditions above, with a bearing vector, R, and a (possibly noisy) emitter direction vector B_{noisy}, an intermediate vector, Q, is formed for use in the calculations:

$$Q = R + B_{noisy} \qquad (35)$$

From here, the individual emitter estimates for each data point can be made using the following equation:

$$M = Rt + Q(1 - t) \qquad (36)$$

For the purposes of subsequent calculations, the vectors are expanded into their components:

$$[M_x \quad M_y \quad M_z] = [R_x \quad R_y \quad R_z]t + [Q_x \quad Q_y \quad Q_z](1 - t) \qquad (37)$$

Now, because we already know the elevation of the earth's surface, we already know the M_z component of the estimate vector:

$$M_z = Z_0 \qquad (38)$$

Therefore, the value of t can be solved for using this information:

$$Z_0 = R_z t + Q_z(1 - t) \qquad (39)$$

After algebraic simplification, this becomes:

$$t = \frac{Z_0 - Q_z}{R_z - Q_z} \qquad (40)$$

This value of t can then be used to solve for M_x and M_y in the following equations:

$$M_x = R_x t + Q_x(1 - t) \qquad (41)$$

$$M_y = R_y t + Q_y(1 - t) \qquad (42)$$

These are the estimated x and y coordinates of the emitter. In order to obtain the minimum mean-squared estimate, the following approach is utilized:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ & \dots \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} \hat{X}_0 \\ \hat{Y}_0 \end{pmatrix} = \begin{bmatrix} M_{1x} & M_{1y} \\ M_{2x} & M_{2y} \\ \dots & \dots \\ M_{Nx} & M_{Ny} \end{bmatrix} \tag{43}$$

This equation fits into the Cartesian Pseudo-Linear Estimation form:

$$Hx = z \tag{44}$$

Since the equation is still in this form, it can still be solved using the MATLAB function x = pinv(H)*z. However, with the H matrix in the above form, the calculation can be simplified to averaging the minimum mean-squared estimates, as shown below:

$$\hat{X}_0 = \frac{1}{N} \sum_{k=1}^{N} M_{kx} \tag{45}$$

$$\hat{Y}_0 = \frac{1}{N} \sum_{k=1}^{N} M_{ky} \tag{46}$$

These values will give a continually-updated estimate of the location of the emitter, projected onto the surface of the earth.

Numerical Example

For this case, the initial conditions shall be as follows. Suppose we set the emitter position:

$$S = \begin{bmatrix} 3 & 5 & 1 \end{bmatrix} \tag{47}$$

And create the path vector R such that:

$$R = \begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} = \begin{bmatrix} 2 & 3 & 4 \\ -1 & -2 & -3 \\ 2 & 3 & 4 \end{bmatrix} \tag{48}$$

From here, we calculate the B vector as follows, with the S matrix transposed to fit our MATLAB example:

$$B = S^T - R = \begin{bmatrix} 1 & 0 & -1 \\ 6 & 7 & 8 \\ -1 & -2 & -3 \end{bmatrix} \quad (49)$$

From here, for each set of coordinates, the Q matrix is calculated:

$$Q_1 = [3 \quad 5 \quad 1] \quad (50)$$

$$Q_2 = [3 \quad 5 \quad 1] \quad (51)$$

$$Q_3 = [3 \quad 5 \quad 1] \quad (52)$$

And then the value of t is calculated for each set of coordinates as well:

$$t_1 = 0 \quad t_2 = 0 \quad t_3 = 0 \quad (53)$$

And the M estimate matrices are then computed:

$$M_x = [3 \quad 3 \quad 3] \qquad M_y = [5 \quad 5 \quad 5] \quad (54)$$

The minimum mean-square estimates are then computed through averaging. Since there is no noise in this example, this part is straightforward.

$$\hat{X}_0 = \frac{1}{3}\sum_{k=1}^{3} M_{kx} = 3 \quad (55)$$

$$\hat{Y}_0 = \frac{1}{3}\sum_{k=1}^{3} M_{ky} = 5 \quad (56)$$

This calculation seems very straightforward and some parts of it might seem unnecessary, but when random noise is added to the calculations, M_x and M_y do not always contain the same repeated values.

Kalman Filter

The LMS algorithm described previously performs adequately for the case of the stationary emitter and any number of observer aircraft. However, there are two main issues that must be addressed with this method. The first deals with computation resources and memory usage. In order to get the best estimate, more measurement data

must be kept and used in the covariance matrix calculations. The second issue deals with the case of the moving emitter. If the emitter is not stationary, the LMS algorithm is far less accurate due to its complete reliance on previous data [**12**].

A potential solution to both of these issues is the Kalman filter. Instead of completely relying on past data to update the covariance matrix, the Kalman filter uses a combination of the previous covariance matrix estimation and the current measurement in order to update the covariance estimate. This is accomplished using a state-space model so that once the current state is updated the past data can be discarded.

Another advantage of the Kalman filter is that it can update the state of multiple sets of measurements when not all of those measurements are available. For instance, if a system is keeping track of the state of position, velocity, and acceleration, the Kalman filter can update the state of all of these if only position or only velocity are measured. This is accomplished using a measurement matrix, referred to below as H. This matrix relates the data used in updating the state and covariance to the data that has actually been measured. In the example below, the H matrix relates measured position to the actual position; however, it can also be adapted to many other situations [**13**].

Preparation of the Data

When creating the Kalman filter using most methods, it is necessary to construct a measurement matrix, H, that relates the measured values to the actual values. In most cases, this transform matrix is usually simply an input selection matrix consisting of ones or zeros that determines which quantity is actually being measured. However, in our case, we are actually measuring a vector that is being projected onto a plane. Therefore, our H matrix will be used in the calculation that actually accounts for that projection. In

other words, we are taking our previous estimate of the actual position of the data source and applying the H matrix to it in order to produce our current measurement of the data from the source. The process that we use is described below and has, in fact, been utilized naturally as a result of the vector projections we were already performing.

For the purposes of the Kalman filter, the H matrix is utilized as follows:

$$\hat{x} = H\hat{y} + v \qquad (57)$$

Where $\hat{y}$ is the previous guess as to the location of the data source, $\hat{x}$ is the data that will actually be predicted at the receiver, and v is some white Gaussian noise. Using the notation that we have been using in our application so far, this can be written:

$$\begin{bmatrix} m_x \\ m_y \end{bmatrix} = H \begin{bmatrix} x(n) \\ y(n) \end{bmatrix} + v \qquad (58)$$

Where m_x and m_y are the calculated measurements of the x and y values of the emitter location (based on a known elevation, as we will show below) and $x(n)$ and $y(n)$ are the previously-estimated values of the true location of the emitter.

In our example, we first construct a vector, b, which represents the true location of the emitter relative to the receiver. We then add noise to it such that:

$$b_n = b + v \qquad (59)$$

In this equation, b represents the actual known location of the emitter relative to the path r. From here, we construct an intermediate vector q such that:

$$q = r + b_n \qquad (60)$$

Note here that q, r, and b_n are all three-dimensional vectors with x, y, and z components to them. Given that fact, we can calculate a projection constant, p, such that:

$$p = \frac{z_0 - q_z}{r_z - q_z} \qquad (61)$$

In this equation, r_z and q_z are the z components of r and q, respectively, and z_0 is the known elevation of the emitter, based on previously-known terrain data. From there, our method calculates m_x as shown below:

$$m_x = r_x p + q_x * (1 - p) \qquad (62)$$

The value for m_y is calculated in a similar manner, so only the m_x calculation needs to be shown here. By expanding q_x in this equation, the following is produced:

$$m_x = r_x p + r_x(1 - p) + b_n(1 - p) \qquad (63)$$

And this, through reduction, produces:

$$m_x = r_x + b_n(1 - p) \qquad (64)$$

Since b_n can be said to represent the known location of the emitter with noise added, in this situation the H matrix would simply be:

$$H = (1 - p) \qquad (65)$$

Or, in its expanded form,

$$H = (1 - \frac{z_0 - q_z}{r_z - q_z}) \qquad (66)$$

Implementation

In order to implement the Kalman filter into our algorithm, first a set of preliminary measurements must be made using another method, such as the least squares method described in previous sections. From this preliminary data, an initial covariance matrix can be calculated. These measurements and the resulting covariance matrix represents the a priori data that initializes the filter. The measurement data covariance matrix is represented below by R_y, the error covariance is represented by R_w, and R_v is a constant representing the error in the measurement equipment which is known beforehand [**12**].

The optimal estimate evolves as follows. First, measurements of the estimated position of the emitter, m_x and m_y, are read and stored:

$$\hat{x} = \begin{bmatrix} m_x \\ m_y \end{bmatrix} \qquad (67)$$

From here, the Kalman gain for the present iteration is calculated using the a priori error covariance values:

$$K = R_{\hat{y}} H^H R_w^{-1} \qquad (68)$$

Additionally, the error covariance is also updated:

$$R_w = H R_{\hat{y}} H^H + R_v \qquad (69)$$

From here, the optimal estimate is determined using a combination of the previous estimate and the new measurement weighted by the Kalman gain:

$$\hat{y} = \hat{y} + K[\hat{x} - \hat{y}] \qquad (70)$$

Now that the optimal estimate has been acquired, the a posteriori covariance is calculated:

$$R_{\hat{y}} = [I - KH] R_{\hat{y}} \qquad (71)$$

This covariance will be used as the a priori covariance on the next iteration of the filter [**12**].

CHAPTER THREE

Simulation

Introduction to Simulation

For our project, the algorithms that have been developed will be simulated using the MATLAB mathematical development environment in order to verify their general operation and functional robustness. MATLAB provides a great deal of computational ability as well as several toolboxes that can be used for data visualization and confirmation. Our simulation will include tests of various sample flight paths and formations, as well as the ability to test and switch between our multiple algorithms on the fly. Additionally, several sources of error, both physical and computational, will be modeled and the methods of accounting for and removing those errors will be developed and tested extensively.

Why Simulate

Simulation is an important part of any development process. Through simulation, results can be obtained and small errors in development can be corrected more easily through trial and error. Simulation eliminates the need to construct hardware in order to test that a system will behave mathematically the way it is expected to.

Simulation on a common platform such as MATLAB also increases the portability of the design. In the case of a demonstration of concept design, development using a single set of hardware makes such a demonstration intrinsically specialized to the hardware it is developed on. However, if a simulation is created using a common tool

with as broad of a feature set as possible, the algorithms can then be transported to more specialized hardware as required after the base functionality is established.

In the case of the system covered in this paper, simulation of the algorithms removes the need for the actual hardware to be present at all. As this is only a preliminary verification and 'proof of concept', any reliance on actual hardware would have been counterproductive to our efforts.

Sources of Measurement Error

As mentioned in the introductory chapter, there are many sources of physical error in receiver systems that are influenced by the geometry of the system as well as by environmental factors. In addition to this, there are additional considerations within the instruments themselves that contribute to minor, but not negligible, errors in the recorded measurements. As mentioned above, calibration errors are important to consider, as are the specified error tolerances of a global positioning system in general. In order to avoid the need to model each specific source of error for each specific case, the various sources of error have been generalized into two categories: Random Noise errors and Angle Outlier errors. Because these generalizations have been made and the possibility exists that they can be tested for accuracy, the modeling of the actual error sources can be simplified for the purposes of simulation, and the algorithms that are being simulated can be adapted to real-world situations with few changes.

Random Noise

In general, error sources such as calibration errors and the minor intrinsic inaccuracies of a global positioning system, as well as other minor environmental errors

such as wind and weather effect and air pressure, can be considered to have a collective contribution to the system inaccuracy as a whole [**3**]. These sources can therefore be modeled as a random process such as white Gaussian noise. The reason this process was chosen is due to the fact that it is unknown exactly how many sources of inaccuracy are contributing at any given time, and in general when a large number of finite variance random variables are summed together, the result, based on the Central Limit Theorem is a Gaussian random variable.

Another observation that is addressed by this system is the idea that in any radio system, the signal to noise ratio decreases as the receiver gets further away from the emitter. Though the magnitude of this decrease depends heavily on the initial strength of the signal coming from the emitter as well as on the quality of the components and design of both the emitter and receiver, it can be assumed that the signal to noise ratio increases proportional to the square of the distance between the receiver and emitter. This is due partially to signal attenuation at greater distances and partially to the fact that in a system such as this, greater distances increase the effect of any error in the system, thus causing even smaller errors to become significant when applied over a larger distance. Therefore, incorporating a squared distance component into the noise model accounts for this effect.

Angle Outliers

There are other sources of error in emitter-receiver systems that contribute more significantly than a white Gaussian process can effectively model. There is a great deal of electromagnetic interference in the atmosphere that can contribute to erroneous measurements of the signal. Additionally, there could be more than one emitter in the area, thus causing some ambiguity as to which signal originated from which emitter [**3**].

Another source of error is the presence of the ground and the ionosphere. When an emitter broadcasts a signal over a wide angle, components of that signal that are broadcast downward reflect off the ground and can arrive at the receiver at a different angle than the primary line of bearing. Similarly, components of the signal that are broadcast upward will reflect off of the ionosphere and can cause similar effects [**3**].

In general, it cannot be determined exactly when these interferences will occur in such a way that the design of the receiver system will not account for. Therefore, this error can be modeled as two random processes, one dependent on the other. The first random process is a uniform process with a defined threshold value. Whenever the output of the process exceeds this threshold, an angle outlier is said to occur. This triggers a second process that determines how distorted the angle of arrival is from what it should have been. Unlike the Gaussian process above, this process is not dependent on distance at all, and can occur any time and with any magnitude. However, it does not occur very often [**3**] [**11**].

Noise Modeling

There are two ways that the Gaussian white noise can be modeled in the data simulation. In the first method, each B vector is split into its Cartesian components, as shown below.

$$B = \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} \qquad (72)$$

For each of these components, a noise component is calculated using the Gaussian normal distribution, the parameter ‘a’, and the length of the B vector components, as shown below.

$$n_x = N(0,1) \cdot a \cdot |B_x|^2 \qquad (73)$$

$$n_y = N(0,1) \cdot a \cdot |B_y|^2 \qquad (74)$$

$$n_z = N(0,1) \cdot a \cdot |B_z|^2 \qquad (75)$$

These noise components are then added to the B vector to produce a noisy simulation vector as shown.

$$B_{noisy} = \begin{bmatrix} B_x + n_x \\ B_y + n_y \\ B_z + n_z \end{bmatrix} \qquad (76)$$

The second method of modeling the Gaussian noise involves converting the B vectors from Cartesian to spherical coordinates, adding the noise to the resulting angular measurements, and then converting back.

In order to represent the angle outlier component of the noise, the B vector is first represented in terms of its spherical components, as shown.

$$B_{noisy} = \begin{bmatrix} \theta \\ \varphi \\ r \end{bmatrix} \qquad (77)$$

From here, a random variable is created that will determine whether an angle outlier occurs. This variable is uniform over the interval from zero to one. It is then compared against a chosen constant to determine the occurrence of the angle outlier.

$$U[0,1] < \varepsilon \qquad (78)$$

If this occurs, the magnitude of the angle outlier is determined by a second set of random processes applied to the angular components of the B vector.

$$n_\theta = U[-\tfrac{\pi}{2}, \tfrac{\pi}{2}] \qquad (79)$$

$$n_\varphi = U[-\tfrac{\pi}{2}, \tfrac{\pi}{2}] \qquad (80)$$

From here, the noise is added into the angular components as follows.

$$B_{phase} = \begin{bmatrix} \theta + n_\theta \\ \varphi + n_\varphi \\ r \end{bmatrix} \qquad (81)$$

Finally, the noisy B vector is converted back to Cartesian coordinates for use in any subsequent calculations.

Flight Path Examples

In order to simulate the data, first a set of flight paths must be created. In a real-world situation, there exist geometries for both single and multiple aircraft flight paths that are best suited for each type of geolocation algorithm. Conversely, there are also geometries that are not ideal and produce suboptimal performance of the algorithms. The performance of various flight patterns paired with specific types of algorithms has been evaluated and documented to varying extents in past experiments and papers [**10**] [**14**] [**15**] [**1**] [**2**]. As mentioned in the introduction sections, the performance of Time Difference of Arrival and Angle of Arrival algorithms depend only on the position of the aircraft in the flight pattern and the performance of Frequency Difference of Arrival algorithms depends on both position and velocity of the aircraft in the flight pattern. Therefore, our test flight patterns are concerned mostly with the difference in position of the aircraft involved in the pattern.

The commonly accepted theory is that in Angle of Arrival algorithms, parallel paths produce similar regions of error [**11**] [**8**]. Therefore, little new information is conferred through the addition of more aircraft to the flight path and the performance of the algorithm does not improve. If aircraft in the flight pattern take differing paths, the regions of error are different and therefore confer differing information. The intersection

of the regions of error is smaller and will give a more accurate representation of the location of the emitter.

The flight paths below were chosen to be representative of both good and poor flight pattern geometries. The first patterns that will be examined in Figure 1 through Figure 4 are the flight paths of single aircraft. The remaining flight paths in Figure 5 through Figure 12 include two and three aircraft flight patterns.

Single Aircraft

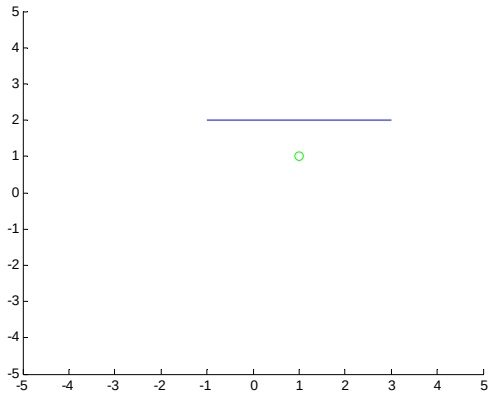

Figure 1: Horizontal Flight Path

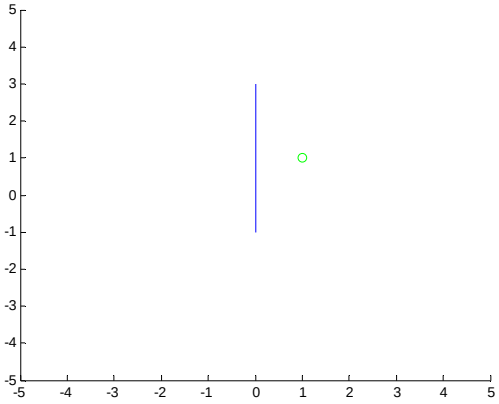

Figure 2: Vertical Flight Path

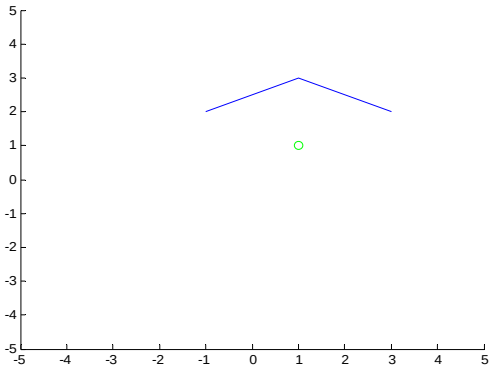

Figure 3: Curved Concave Flight Path

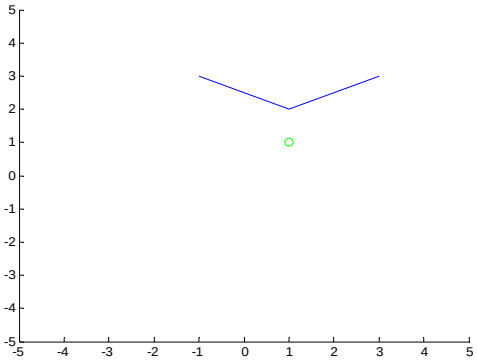

Figure 4: Curved Convex Flight Path

Multiple Aircraft

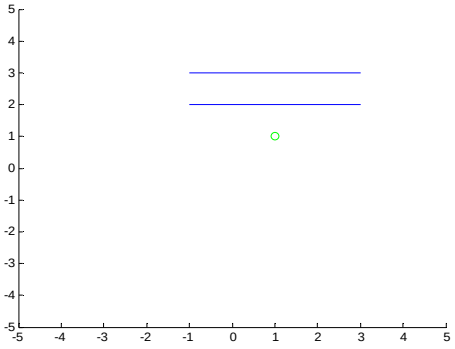

Figure 5: Parallel Flight Paths

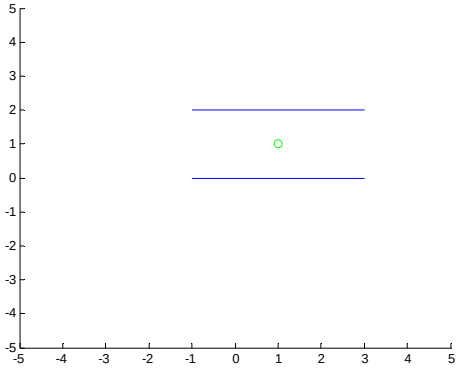

Figure 6: Parallel Flight Paths, Opposite Sides

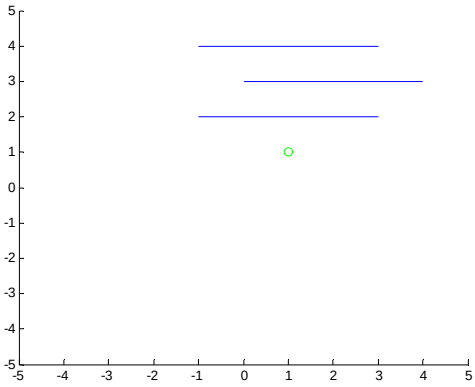

Figure 7: Parallel Flight Paths, Three Aircraft

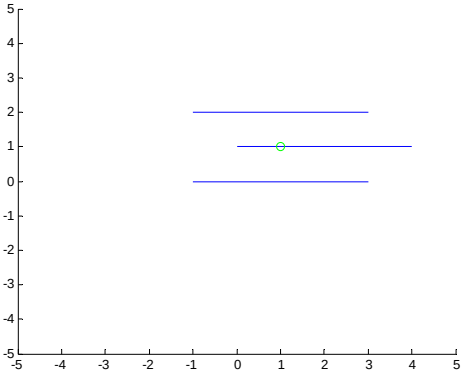

Figure 8: Parallel Flight Paths, Three Aircraft, Opposite Sides

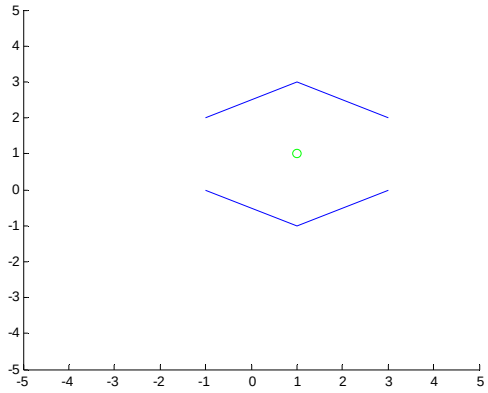

Figure 9: Diamond Formation

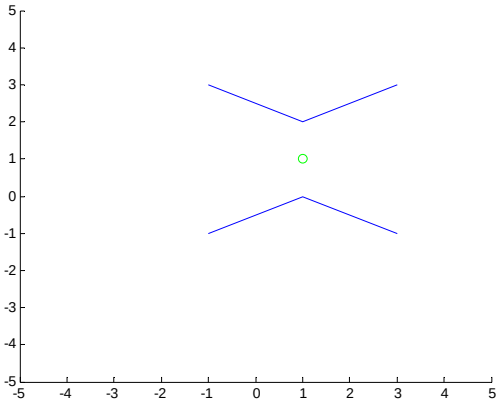

Figure 10: Hourglass Formation

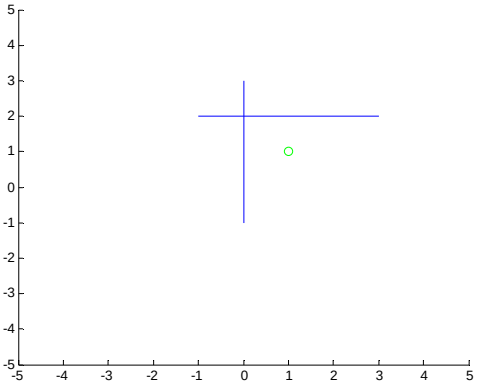

Figure 11: Perpendicular Flight Paths

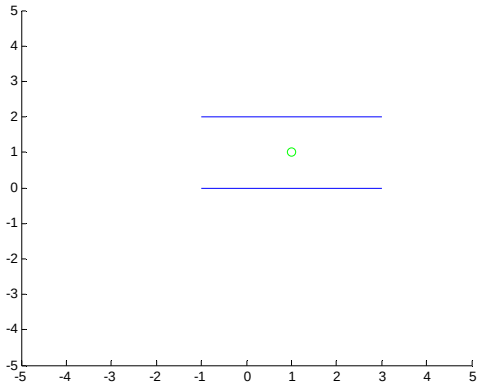

Figure 12: Parallel Flight Paths, Opposite Direction

Performance of Optimal Routing with Gaussian Noise

In order to determine how well a certain algorithm or flight path performs, one must first determine what constitutes 'good performance'. While the algorithms are quite different from each other, the estimate data is recorded in a manner that is consistent no matter which algorithm is used. Therefore, relative performance can be measured. The metric that we used to determine 'good performance' is the percentage of time that the region of probability includes the actual location of the emitter. While it is assumed that in the real world this will not happen every single time, an algorithm can be considered to have good performance if the error ellipse includes the emitter location 95% of the time. This idea is analogous to confidence intervals from statistics literature. For a given flight path, the amount of noise is not something that can be changed. However, what can be changed is the way that the region of probability is constructed. Specifically, what the tests will look at is the constant by which the size of the region of probability is scaled by in order to achieve the confidence specification. Table 1, shown below, shows the inclusion percentage and the scaling constant that must be used to achieve that

percentage. It should be noted that achieving 95% inclusion with a lower scaling constant is preferable.

The data in Table 1 shows that each method behaves the way one would expect. The windowed least-squares method behaves with greatly increased reliability when compared with the method that does not utilize a data window. The Kalman filtering method shows an even greater increase in reliability to the point that the effects of the Gaussian noise are nearly negated.

Performance of Optimal Routing with Angle Outliers

As described in earlier sections, there are other sources of error that cannot be estimated simply with Gaussian noise. The angle outlier error estimation is one of those cases. Therefore, once the optimal estimation has been made using simple Gaussian noise, another test was performed that kept all metrics the same save for the inclusion of these angle outliers. The data from this test was recorded in Table 2 below.

As the data in Table 2 shows, the presence of angle outliers does not greatly affect the data in the windowed method. This is due to the fact that the bad data eventually falls out of the window and does not affect the results for any great length of time. If the data window is not utilized, the presence of angle outliers affects the data to a much greater extent. When angle outliers are introduced to the Kalman filter, reliable performance was difficult to achieve because the data shows excessive deviation from the modeled bounds. This deviation may cause the output of the filter to become unstable, especially if it deviation occurs near the beginning of the filter's operation. As a result, when angle outliers are introduced to the Kalman filter, it is absolutely essential to utilize robustness techniques to ensure continued operation of the filter.

Table 1: Performance with Gaussian Noise

Flight Path	Method	Ellipse Inclusion	Polygon Inclusion	Scaling Constant
Path 1	window	97.49%	98.70%	40
	no window	93.16%	95.87%	200
	kalman	96.49%	98.11%	5
Path 2	window	94.30%	96.90%	30
	no window	87.38%	91.15%	200
	kalman	91.28%	94.21%	5
Path 3	window	96.43%	98.18%	30
	no window	89.59%	94.09%	200
	kalman	95.24%	96.37%	5
Path 4	window	93.58%	96.98%	30
	no window	89.65%	91.52%	200
	kalman	87.62%	93.99%	5
Path 5	window	91.56%	95.03%	30
	no window	85.72%	92.11%	200
	kalman	93.12%	96.09%	5
Path 6	window	91.04%	95.43%	30
	no window	80.75%	88.88%	200
	kalman	93.75%	95.46%	5
Path 7	window	89.29%	94.49%	35
	no window	61.19%	66.72%	200
	kalman	94.76%	97.84%	8
Path 8	window	91.28%	95.63%	30
	no window	84.39%	89.63%	200
	kalman	91.97%	96.15%	5
Path 9	window	93.67%	96.22%	30
	no window	86.98%	91.46%	200
	kalman	92.70%	95.27%	5
Path 10	window	91.02%	94.95%	30
	no window	82.08%	87.47%	200
	kalman	92.86%	96.17%	7
Path 11	window	92.29%	95.62%	30
	no window	80.28%	86.75%	200
	kalman	95.73%	96.71%	5

Table 2: Performance with Angle Outliers and Gaussian Noise

Flight Path	Method	Ellipse Inclusion	Polygon Inclusion	Scaling Constant
Path 1	window	97.53%	98.68%	30
	no window	60.26%	61.34%	200
	kalman	n/a	n/a	5
Path 2	window	94.32%	96.60%	30
	no window	63.25%	64.64%	200
	kalman	n/a	n/a	5
Path 3	window	96.00%	97.12%	30
	no window	57.95%	61.98%	200
	kalman	n/a	n/a	5
Path 4	window	95.07%	96.79%	30
	no window	69.12%	70.51%	200
	kalman	n/a	n/a	5
Path 5	window	90.62%	94.09%	30
	no window	51.93%	53.59%	200
	kalman	n/a	n/a	5
Path 6	window	90.91%	94.12%	30
	no window	44.72%	46.36%	200
	kalman	n/a	n/a	5
Path 7	window	89.99%	93.64%	35
	no window	40.17%	44.79%	200
	kalman	n/a	n/a	8
Path 8	window	90.95%	93.86%	30
	no window	46.63%	49.56%	200
	kalman	n/a	n/a	5
Path 9	window	89.39%	93.06%	30
	no window	45.45%	47.57%	200
	kalman	n/a	n/a	5
Path 10	window	91.11%	94.68%	30
	no window	51.52%	53.56%	200
	kalman	n/a	n/a	7
Path 11	window	88.75%	93.49%	30
	no window	49.20%	52.91%	200
	kalman	n/a	n/a	5

CHAPTER FOUR

Error Simulation

Error Estimation

Methods of Estimating Error

Error Ellipses. In a real-world example, the actual location of the emitter is never known. Therefore, obtaining an estimate for that position consisting of a single point has limited value due to the fact that it cannot be known how close that estimate is to the actual emitter location. What can be known, however, is the relation of one estimate to the previous estimates, providing they are stored in memory or data from them is otherwise made available. From the current and past data, one can easily use a statistical model to place bounds on the precise emitter location. From Chebyshev's inequality, it is known that the emitter will be located within a spatial region defined in terms of a multiple of the standard deviation of the estimate data to within a prescribed confidence level. Our approach for defining the bounding spatial location of the emitter is described below:

Given a set of estimates produced by an algorithm, the following covariance matrix can be obtained:

$$A = \begin{bmatrix} \sigma_{xx}^2 & \sigma_{xy}^2 \\ \sigma_{yx}^2 & \sigma_{yy}^2 \end{bmatrix} \qquad (82)$$

Where:

$$\sigma_{xx}^2 = E[(X - \mu_x)^2] \qquad (83)$$

$$\sigma_{yy}^2 = E[(X - \mu_y)^2] \qquad (84)$$

$$\sigma_{xy}^2 = \sigma_{yx}^2 = E[(X - \mu_x)(Y - \mu_y)] \qquad (85)$$

From here, singular value decomposition is performed on the inverse of A to obtain the following matrices:

$$A^I = UDV \qquad (86)$$

Where U and V contain the singular values of A^I, and D is the scaling vector that determines the orientation of the matrix.

From these matrices, a parametric equation for an ellipse can be calculated, such that:

$$S_1 = \frac{\cos\theta}{\sqrt{D_{11}}} \qquad (87)$$

$$S_2 = \frac{\sin\theta}{\sqrt{D_{22}}} \qquad (88)$$

Where θ is continuous on the interval:

$$\theta = [0 \quad 2\pi] \qquad (89)$$

Additionally, the ellipse is centered at the mean of the data set:

$$C = \begin{bmatrix}\mu_x \\ \mu_y\end{bmatrix} \qquad (90)$$

From here, the points on the ellipse can be completely defined as follows:

$$P = VS + C \qquad (91)$$

Where

$$S = \begin{bmatrix}S_1 \\ S_2\end{bmatrix} \qquad (92)$$

By plotting all the points present in P, an ellipse can be drawn that represents the region that has the greatest probability of including the emitter location.

Simulation based studies are effective because we are able to know the exact location of the emitter in order to evaluate the algorithm's performance. We then take the approach of determining a proposed method for spatially bounding the emitter and using the simulation, we can verify whether or not the emitter actually lies within this region. To accomplish this, we must consider the mathematical equation of an ellipse:

$$(X - C)^T A(X - C) \leq 1 \qquad (93)$$

Where X is a point in space, and C is the center of the ellipse. The matrix A in this case can be viewed as a scaling matrix. In order to test if a point X is inside the ellipse represented by this equation, C and A must be known values. From the calculations above, it is already known where the center of the ellipse, C, lies. The scaling matrix A is simply the inverse of the covariance matrix above. Thus, to evaluate whether the bounding region actually contains the emitter, we test to see if equation (94) is satisfied. The "test condition" equation of the ellipse can be rewritten as:

$$(E - C)^T A^I (E - C) \leq 1 \qquad (94)$$

Where E is the location of the emitter:

$$E = \begin{bmatrix} E_x \\ E_y \end{bmatrix} \qquad (95)$$

We reiterate that this test is only for simulation verification purposes, as in a real world situation the exact location of the emitter is likely to be unknown.

Ellipses estimated with Parallelograms. In most computer programs as well as most hardware applications, ellipses and other curved lines are not represented simply as a single line that has a curve. Instead, the representation often is achieved by considering a number of points on the ellipse that are connected with a series of straight lines. To the

human eye, this representation takes on the appearance of a curved line. However, this method of representing an ellipse is quite computationally intensive due to the number of points and lines that must be plotted, especially if the ellipse is constantly shifting.

An alternative to this method is to pick a small number of points consistent with the mathematical representation of the ellipse and connect those points with straight lines. This will create a parallelogram that approximates the shape of the ellipse. If the points are chosen correctly, the ellipse will be completely circumscribed by the parallelogram.

In the above calculations for the complete ellipse, if θ is changed so that it is now

$$\theta = \left[0, \quad \frac{\pi}{2}, \quad \pi, \quad \frac{3\pi}{2}\right] \qquad (96)$$

When these points are plotted and connected in the final step of the calculation, the points will describe a parallelogram inscribed inside the ellipse instead of the ellipse itself. If this parallelogram is scaled by a value of 2, it will completely circumscribe the ellipse.

If the region of probability of the location of the emitter is represented in this manner, it will lead to a less computationally intensive design, and as such fewer resources should be needed to display the error regions to a display.

Intersecting Regions of Error. One advantage to representing the possible location of the emitter as a region of probability is that if the data from multiple flight paths is being compared, one can easily see how the estimations of one aircraft compare with the estimations of another. An additional advantage is that if the assumption can be made that the aircraft share data in some manner, or if the data from multiple aircraft is collected and processed in a central location, the data can be compared in order to further narrow the region of likelihood of the location of the emitter.

Error Mitigation

Error Mitigation typically refers to the practice by which a digital system can be encoded to account for the effect of analog errors and loss of data. It can also refer to the practice by which an algorithm in a digital system can be specifically coded to reduce or eliminate the effect of errors in measurement data on the resulting calculations. As it has been shown in previous sections, the nature of the hardware that these algorithms are being developed for lends itself to certain degrees of measurement error, both known and unknown. Therefore in the simulations performed on those algorithms, error mitigation techniques can be used to represent actual error control systems that can be implemented in a real-world design.

Methods of Error Mitigation

Data windowing. There are several methods with which corruption in the input data can be accounted for. The first of these methods is the window method, in which only a certain window of data is utilized in the geolocation calculation. When this method is used, bad data eventually 'falls out' of the calculation after a corresponding amount of new data comes in. This method also contributes to the practicality of the system as a whole, because in a real-world situation only a certain number of data points will be retained for calculation anyway.

In order for this method to be utilized, the buffer window must first be filled. This will produce a small delay before the first actual results can be produced. Once the buffer window is filled, the calculations can be performed at a rate of one calculation per new data point.

Soft censoring of data. There are some cases in which simply windowing the data is not sufficient to limit the effects of measurement errors. Specifically, in the case of the angle outliers described in a previous section, a data point that falls well outside the range of the majority of the other data points will greatly skew the emitter estimate. If only the 'windowing' method is used, the erroneous data point will continue to affect the estimation until it falls out of the data window.

There are several proposed answers to this problem. The first of which is to censor the data using a 'soft limiter'. In this method, a certain limit is set in the software. This limit is based on a multiple of the standard deviation of a subset of the data. If any data falls outside of this limit, the data is considered clearly in error and is accounted for. Through the use of this method, many potential angle outliers can be identified, flagged, and processed. In the simulation, the data point with the angle outlier is simply replaced by the previous data point.

This method works quite well for both the data-windowed and non-data-windowed versions of the CPLE algorithm. However, this method produces suboptimal results when paired with the Kalman filter algorithm. The reason for this is that the Kalman filter must be initialized with data obtained from other, less accurate methods. This data will factor into the standard deviation calculation used in the 'soft limiter'. Therefore, an angle outlier that occurs when the Kalman filter is first activated may not necessarily fall outside the standard deviation of the data used to initialize it and as a result will not be censored. This, coupled with the fact that erroneous data near the beginning of a Kalman filtered data stream will greatly skew the results of the filter, results in poor performance.

Data mismatch censoring. In order to produce a method of data censoring that would produce more optimal results when paired with the Kalman filtering algorithm, a number of alternatives were considered. The alternative that made the most sense was an averaging filter, or 'data mismatch censoring'. In this method, the mean value of a set of data which includes the most recent N data points is calculated:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} m_i \qquad (97)$$

From here, the magnitude distance from each data point to the average is calculated:

$$D_i = \sqrt{m_i - \mu} \qquad (98)$$

The point with the maximum distance from the average is then replaced by the mean value of the data set, thus reducing its effect on the emitter location calculations. Because no knowledge of when an angle outlier occurs is assumed, this calculation is present in each iteration of the algorithm, thereby improving the algorithm's general effectiveness even though 'good' data is sometimes discarded.

How the data is censored and how many previous data points are used in the calculation of the average is situation-dependent. In the windowed CPLE algorithm, all of the data in the window is used to calculate the mean. The point with the greatest distance from the mean is discarded from the window and not used in the current calculation, but is retained in memory for use in subsequent averaging and estimation calculations. The reasoning is that if an angle outlier occurs, this point will always be the greatest distance from the average and will always be the point that is discarded until it falls out of the data window.

In the Kalman filter algorithm, the value of N is arbitrary and can be adjusted to improve performance. Additionally, only the most recent data point is used to compute the estimate. Therefore removing the data point that falls furthest from the average makes no sense for the following reasons: if the point in question is the most recent point, the Kalman filter will not have any new data to use in its calculation, and if the point in question is not the most recent point removing it will not matter. It is also not enough to simply replace the affected data point with the point used previously. Doing so will eventually cause the Kalman filter to converge to a single point and therefore limit its effectiveness as an evolving adaptive algorithm. The proposed solution in this case is to either replace the affected data point with the mean value that is used in determining the outlier, or to replace it with some other limit that, while still keeping the point as an outlier, will not catastrophically hinder the performance of the Kalman filter.

CHAPTER FIVE

MATLAB Coding

MATLAB Implementation

In order to test our algorithm in the absence of an opportunity for real-world trials, a suitable test model must be created that can accurately represent real-world behavior and sources of error. When constructing this model, a test environment must be chosen that is robust enough to perform the necessary calculations, as well as adaptable enough to allow for ease in altering and correcting the model. MATLAB was chosen as the environment for constructing and simulating the mathematical model.

The first thing that happens is the construction of the receiver test path. The path is constructed using waypoints and time steps provided in a comma separated value (CSV) file that is imported into the MATLAB script using the csvread MATLAB function. This stores the x, y, z, and time components of the waypoints into their respective vectors for processing. More than one path can be added to the simulation as long as the waypoints are defined in the order: 'x; y; z; t'. The positions of the waypoints are interpolated along the time axis so that they are separated into evenly-spaced time slices. From these time slices, velocity at any point can be easily calculated. Figures 13 and 14 show the positions of the waypoints relative to the position of the emitter and how the intermediate data points are interpolated and connected.

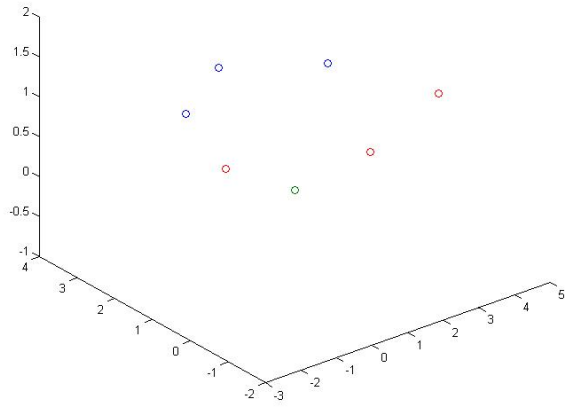

Figure 13: Waypoints of Two Paths in Relation to Emitter

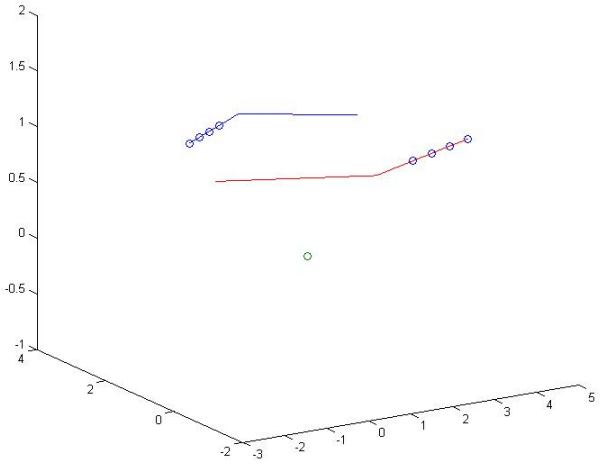

Figure 14: Flight Paths with First Four Time Slices Highlighted

From the path vector, the B vectors are calculated as described in the previous section. Because the location of the emitter is assumed to be unknown, and because in the real world a certain degree of error is assumed in the readings obtained from the emitter, noise must be added to the B vectors to ensure an accurate model. The noise is

constructed based on the square of the distance, and is a Gaussian random variable of the form randn().*a.*abs(b).^2. In this equation, a is a specified noise parameter chosen to model real-world noise power as closely as possible. This equation relates noise power to the distance away from the emitter that a receiver is, by a factor of the square of the magnitude of the distance.

After the noise is added into the signal, the algorithm is run just as described above. There are three modes the algorithm can be run in. The first mode simply aggregates all of the data and produces a result. This mode is advantageous for small datasets as it does not throw away any data and as such reaches the most accurate conclusion possible from the available data. The second mode 'windows' the data, or throws away old data when new data enters the system. In order for this method to work, the window buffer is 'filled' before any calculations are performed, thus ensuring there is always a certain number of back data to perform calculations on. This method has the advantage of eliminating the need for a large data storage space for a long burst of data from the emitter, as well as providing a level of assurance that bad data does not persist and continually disrupt calculations. The nature of a data window ensures that bad data will eventually 'fall out' of the window. The third method is the Kalman filtering method that only stores the most recent data point and uses a combination of that point and the covariance calculated from the previous data points in order to determine the most likely location of the emitter.

Various plots are utilized to aid in the visualization of the data simulations. One plot displays the flight paths in three dimensions, as well as a visual representation of a selection of the noisy B vectors pointing at the general area of the emitter. A second plot

shows a top-down view of the flightpaths, so that the relationship between the flightpaths and the emitter can be more easily shown. A third plot shows the locations of the estimates, as well as an ellipse depicting the region of probability of the location of the emitter, which will be discussed below. A fourth plot shows the region of intersection of a polygon inscribed in the previous ellipse, which is also discussed below.

The ellipse of probability is computed using the covariance matrix A:

$$A = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{yx} & \sigma_{yy} \end{bmatrix} \qquad (99)$$

Where:

$$\sigma_{xx} = var(x) \qquad (100)$$

$$\sigma_{yy} = var(y) \qquad (101)$$

$$\sigma_{xy} = \sigma_{yx} = \sum_{n=1}^{N} \frac{(x_n - \mu_x)(y_n - \mu_y)}{N} \qquad (102)$$

Therefore, the emitter probability ellipse can be shown in the form:

$$(X - C)^T A^I (X - C) = 1 \qquad (103)$$

Where:

$$X = \begin{bmatrix} x \\ y \end{bmatrix} \qquad (104)$$

$$C = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix} \qquad (105)$$

From this data, an ellipse can be derived using singular value decomposition and then plotted over the scatterplot of the guesses. Figures 15 and 16 show the overlapping ellipses as well as the regions of intersection formed by the ellipses when they are combined.

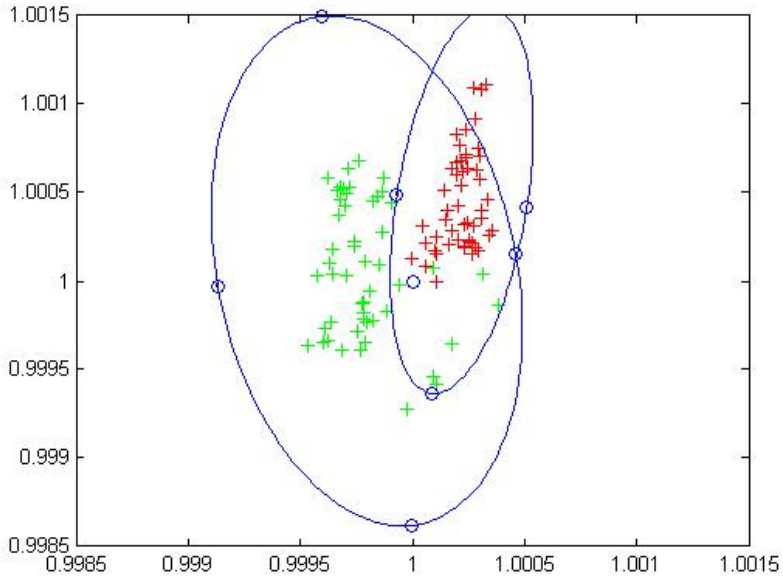

Figure 15: Overlapping Ellipses with Vertices of Polygons Marked

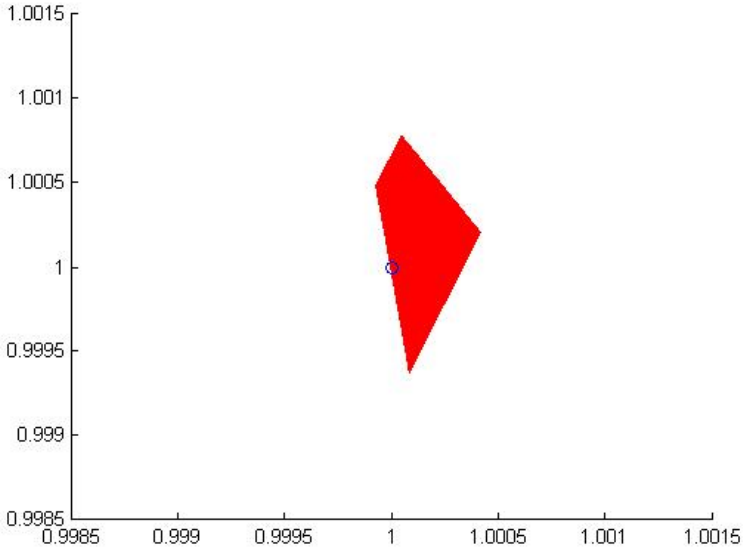

Figure 16: Region of Intersection of the Polygons

If there is more than one path, and therefore more than one error ellipse, the intersecting probability can be formed by inscribing a polygon into each ellipse and then

computing the intersection of those polygons. This is accomplished using the polybool function in the MATLAB mapping toolkit, converting the result to face-vertex form using poly2fv, and then creating a patch plot of the resulting polygon. This graphical form is more easily displayed on a wider range of instruments than an elliptical plot is, due to the fact that in most instruments, and in MATLAB itself, an ellipse is created by mapping points on the curve and then connecting those points as if it were a polygon. Plotting a more basic polygon will reduce the computational power required for display. The polygons are suitable approximations for the ellipses due to the fact that if the polygon is drawn in such a way that the ellipse is completely contained in the polygon, if a point is in the ellipse it is therefore also in the polygon. Thus, the computational requirements to display the region of intersection can be reduced without a loss of accuracy in the estimation.

In order to test the validity of the simulation, it is necessary to determine how often the actual location of the emitter is contained in the emitter probability ellipse. This can easily be determined using the ellipse equation above, and plugging the actual location of the emitter, which is assumed to be known only for simulation purposes, into X. If the resulting value is less than 1, the emitter is inside the ellipse. By counting the number of timeslices in which this is true, and then normalizing by the total number of time slices, a percentage can be acquired. The size of the ellipse can be altered by scaling the A matrix by a constant. If this is done, then the inclusion of the emitter location is tested against that scaling value instead of by 1.

MATLAB GUI

In order to facilitate a more user-friendly testing environment, a MATLAB graphical user interface (GUI) was constructed using the MATLAB graphical user interface development environment (GUIDE) tool. The purpose of the GUI is to provide an interface to directly adjust the most commonly accessed values as well as to collect the resulting measurements in one easily-readable place. From the GUI, it will be possible to adjust values such as the emitter location, the noise constant, and the scaling constant for the error ellipse, as well as to adjust how many times the simulation is run and whether or not plots are displayed.

The GUI also makes it possible to select which version of the algorithm to run if there are multiple versions. For example, there is a version that includes the data 'window', and a version that does not. Using the GUI, it is easy to select which version executes. It also makes it easy to add different algorithms and select which one is used.

The GUI interacts with the algorithm code through the use of the 'handles' structure. When creating a GUI, the handles store the global variables and flags that affect what can be accessed by the GUI. By passing the handles structure into the algorithm code and updating the handles from there, the algorithm code can directly control the GUI display. In general, the handles that determine which algorithm or portions of the algorithm are to be executed are set by the GUI, and then when the 'Execute' button is pressed those handles get passed into the algorithm code where they affect the execution of the program.

Figure 17 shows an image of the GUI, and the various fields will be explained in the following sections.

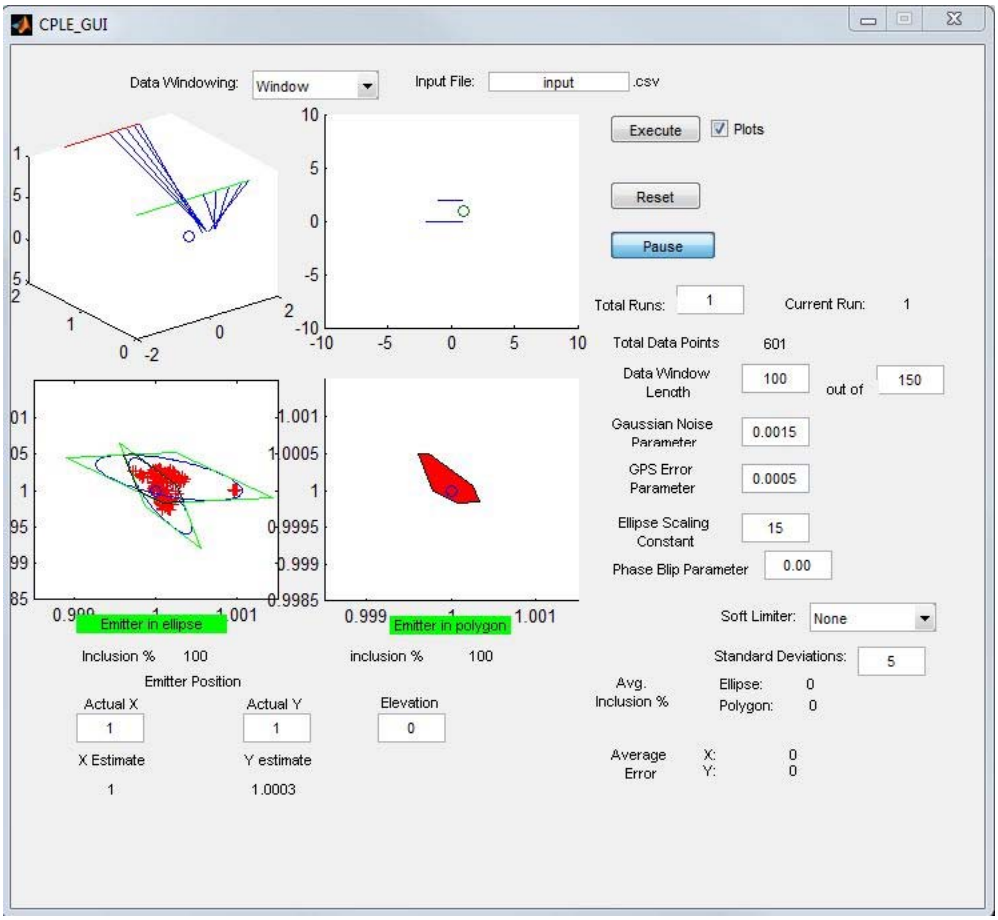

Figure 17: The Graphical User Interface

Data Windowing. The 'data windowing' menu is a drop-down menu that determines which version of the algorithm will execute.

Execute. The 'execute' button causes the algorithm selected in the 'data windowing' menu to be run.

Plots. The 'Plots' checkbox next to the 'execute' button will control whether or not data plots are produced. For more than one or two runs of the data, not producing plots will greatly decrease the execution time of the code.

Reset. The 'reset' button clears the handles structure and resets all plots.

Total Runs. The 'total runs' field adjusts how many times the data for the specified flight path is run. Running the path several times will produce a better average, as there will be more data to compare. If the number of runs is greater than one or two, it is not recommended to produce plots as this will considerably slow the execution of the code.

Current Run. The 'current run' field will inform the user which iteration out of the total number is currently being computed.

Total Data Points. This field informs the user how many time slices the interpolated path is broken up into.

Data Window Length. This field lets the user adjust the size of the data window for a windowed signal. For a non-windowed signal, it will still affect the spacing of the visualization of the B vectors, though it will not affect calculations in any way.

Gaussian Noise Parameter. This field lets the user adjust the Gaussian noise constant 'a'. A greater number in this field will result in more noise being added to the signal in the simulation.

Ellipse Scaling Constant. This field lets the user adjust the constant by which the A matrix is scaled in order to produce the emitter probability ellipse. A greater number in this field results in a larger bound for the error ellipse.

'Emitter in Ellipse' Box. This indicator informs the user in real time whether the emitter is located within the intersection of all error ellipses.

Inclusion Percentage. This indicator displays the percentage of time for a single run that the emitter is located in the ellipses. This value is updated in real time.

Average Inclusion Percentage. This indicator displays the average percentage of time, over all runs of the flight path, that the emitter was located in the ellipse. This value is displayed after every other computation has been completed.

Average Error. This indicator displays the average error among guesses, calculated from the average of all guesses. This is displayed after every other computation has been completed.

Emitter Position. This allows the user to adjust the actual emitter position and elevation before the calculations are performed. Below the edit boxes, the current estimates of position are displayed and updated in real time.

Upper Left Plot. This figure shows, in three dimensions, the flight paths, the emitter, and a selection of the B vectors pointing to the guesses. This figure is updated in real time.

Upper Right Plot. This figure shows a top-down view of the flight paths and the emitter. This figure is updated in real time.

Lower Left Plot. This figure shows a scatterplot of the guesses of the emitter location, and overlays the emitter probability ellipses for each flight path. This figure is updated in real time.

Lower Right Plot. This figure shows the intersection of polygons inscribed in the emitter probability ellipses, superimposed over the emitter location. This figure is updated in real time.

CHAPTER SIX

Conclusions

The first purpose of the simulations presented in this thesis was to determine whether an algorithm could be developed that could provide reliable performance while utilizing reduced computational resources such as those found in legacy hardware. This was accomplished through the use of two algorithms. The first algorithm, the least-squares estimator, provided a solution that operates with a very low processing overhead. Due to the use of surface bearing projections, the matrix inversion, which is the most computationally intensive portion of the least-squares estimator, can be avoided entirely. However, the least-squares estimator does need increased storage resources depending on the amount of data being considered in a data block. The second algorithm, the Kalman filter, is an iterative algorithm that reduces the need for storage space, though at the cost of increased processing requirements.

The more important focus of this thesis was determining how the algorithms would perform under the presence of multiple types of noise or other data corruption and to add error mitigation methods in order to improve stability and robustness. It has been shown through simulation that in low-noise systems the least-squares estimator provides adequate performance. In the same situations, the performance of the Kalman filter shows a great deal of improvement over the least-squares estimator and can reduce the effects of Gaussian noise to a great degree.

When angle outliers were added to the simulation, the advantage of the least-squares estimator became obvious. The use of block processing gives the algorithm a resistance to the lingering effects of an angle outlier due to the fact that the outlier does eventually fall out of the window. In contrast to this, the Kalman filter showed a high susceptibility to angle outliers that occurred near the beginning of the filter's operation. Such an occurrence causes the filter results to not converge. As a result, the Kalman filter must have additional error mitigation techniques to reduce the effects of the angle outliers.

Given that both algorithms utilize robust error mitigation techniques, the Kalman filter shows a much better overall inclusion rate and stability in regards to the orientation of the region of confidence. The drawback to the Kalman filter is its reliance on initialization data. The filter must have some preexisting data in order to determine its initial parameters. A good initialization will result in better precision for the Kalman filter, while a poor initialization will reduce overall precision and as a result expand the region of confidence. An unacceptably poor initialization could also cause the filter to not converge. However, even with an initialization that results in a larger region of confidence, the inclusion rate when utilizing the Kalman filter is still much higher than the least-squares algorithm, and the reduced effects of noise still contribute to its overall exceptional stability.

APPENDICES

APPENDIX A

CPLE_GUI.m

```
function varargout = CPLE_GUI(varargin)
% CPLE_GUI M-file for CPLE_GUI.fig
%      CPLE_GUI, by itself, creates a new CPLE_GUI or raises the
existing
%      singleton*.
%
%      H = CPLE_GUI returns the handle to a new CPLE_GUI or the handle
to
%      the existing singleton*.
%
%      CPLE_GUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in CPLE_GUI.M with the given input
arguments.
%
%      CPLE_GUI('Property','Value',...) creates a new CPLE_GUI or
raises the
%      existing singleton*.  Starting from the left, property value
pairs are
%      applied to the GUI before CPLE_GUI_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to CPLE_GUI_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only
one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help CPLE_GUI

% Last Modified by GUIDE v2.5 11-Jan-2011 13:39:58

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @CPLE_GUI_OpeningFcn, ...
                   'gui_OutputFcn',  @CPLE_GUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end
```

```
if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before CPLE_GUI is made visible.
function CPLE_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to CPLE_GUI (see VARARGIN)

% Choose default command line output for CPLE_GUI
handles.output = hObject;
% handles.plotFlag = 0;
% Update handles structure
guidata(hObject, handles);

% UIWAIT makes CPLE_GUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = CPLE_GUI_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in run_pushbutton.
function run_pushbutton_Callback(hObject, eventdata, handles)
% hObject    handle to run_pushbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%lets the user select the data censoring method to use
switch get(handles.popupmenu2,'Value')
    case 1
        [handles] = surface_bearing_cple_3d_gui_v2(handles);
    case 2
        [handles] = surface_bearing_cple_3d_gui_nowindow(handles);
    case 3
        [handles] = surface_bearing_kalman_3d_gui(handles);
    otherwise
end
guidata(hObject, handles);
function reset_pushbutton_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to run_pushbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%resets all data and clears all axes
cla(handles.axes1,'reset')
cla(handles.axes2,'reset')
cla(handles.axes3,'reset')
cla(handles.axes4,'reset')

set(handles.avgErrX_staticText,'String','0');
set(handles.avgErrY_staticText,'String','0');
set(handles.averagePercent_staticText,'String','0');
set(handles.averagePercent2_staticText,'String','0');
set(handles.emitterXGuess_staticText,'String','0');
set(handles.emitterYGuess_staticText,'String','0');
set(handles.dataPoints_staticText,'String','0');
set(handles.percent_staticText,'String','0');
set(handles.percent2_staticText,'String','0');
set(handles.currentRun_staticText,'String','1');

guidata(hObject, handles);



function numRuns_editText_Callback(hObject, eventdata, handles)
% hObject    handle to numRuns_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of numRuns_editText as
text
%        str2double(get(hObject,'String')) returns contents of
numRuns_editText as a double

%store the contents of numRuns_editText as a string. if the string
%is not a number then input will be empty
input = str2num(get(hObject,'String'));

%checks to see if input is empty. if so, default numRuns_editText to
one
if (isempty(input))
     set(hObject,'String','1')
end
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function numRuns_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to numRuns_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called
% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in plots_checkbox.
function plots_checkbox_Callback(hObject, eventdata, handles)
% hObject    handle to plots_checkbox (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%checkboxStatus = 0, if the box is unchecked,
%checkboxStatus = 1, if the box is checked
% checkboxStatus = get(handles.plots_checkbox,'Value');
% if(checkboxStatus)
%     %if box is checked, plots are turned on
%     handles.plotFlag=1;
% else
%     %if box is unchecked, text is set to normal
%     handles.plotFlag=0;
% end

% Hint: get(hObject,'Value') returns toggle state of plots_checkbox



function emitterX_editText_Callback(hObject, eventdata, handles)
% hObject    handle to emitterX_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of emitterX_editText as
text
%        str2double(get(hObject,'String')) returns contents of
emitterX_editText as a double

%store the contents of emitterX_editText as a string. if the string
%is not a number then input will be empty
input = str2num(get(hObject,'String'));

%checks to see if input is empty. if so, default emitterX_editText to
zero
if (isempty(input))
     set(hObject,'String','0')
end

% --- Executes during object creation, after setting all properties.
function emitterX_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to emitterX_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called
```

```
% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function emitterY_editText_Callback(hObject, eventdata, handles)
% hObject    handle to emitterY_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of emitterY_editText as
text
%        str2double(get(hObject,'String')) returns contents of
emitterY_editText as a double

%store the contents of emittery_edittext as a string. if the string
%is not a number then input will be empty
input = str2num(get(hObject,'String'));

%checks to see if input is empty. if so, default emittery_edittext to
zero
if (isempty(input))
     set(hObject,'String','0')
end

% --- Executes during object creation, after setting all properties.
function emitterY_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to emitterY_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function emitterZ_editText_Callback(hObject, eventdata, handles)
% hObject    handle to emitterZ_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of emitterZ_editText as
text
%        str2double(get(hObject,'String')) returns contents of
emitterZ_editText as a double
```

```
%store the contents of emitterZ_editText as a string. if the string
%is not a number then input will be empty
input = str2num(get(hObject,'String'));

%checks to see if input is empty. if so, default emitterZ_editText to
zero
if (isempty(input))
     set(hObject,'String','0')
end

% --- Executes during object creation, after setting all properties.
function emitterZ_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to emitterZ_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function windowSize_editText_Callback(hObject, eventdata, handles)
% hObject    handle to windowSize_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of windowSize_editText
as text
%        str2double(get(hObject,'String')) returns contents of
windowSize_editText as a double

%store the contents of windowSize_editText as a string. if the string
%is not a number then input will be empty

%lets the user set the size of the block processing window
input = str2num(get(hObject,'String'));

%checks to see if input is empty. if so, default windowSize_editText to
%twenty
if (isempty(input))
     set(hObject,'String','20')
end

% --- Executes during object creation, after setting all properties.
function windowSize_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to windowSize_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called
% Hint: edit controls usually have a white background on Windows.
```

```
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function noise_editText_Callback(hObject, eventdata, handles)
% hObject    handle to noise_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of noise_editText as
text
%        str2double(get(hObject,'String')) returns contents of
noise_editText as a double

%store the contents of noise_editText as a string. if the string
%is not a number then input will be empty

%lets the user set the gaussian noise scale parameter
input = str2num(get(hObject,'String'));

%checks to see if input is empty. if so, default noise_editText to
0.0015
if (isempty(input))
     set(hObject,'String','0.0015')
end

% --- Executes during object creation, after setting all properties.
function noise_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to noise_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function sigmaScale_editText_Callback(hObject, eventdata, handles)
% hObject    handle to sigmaScale_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of sigmaScale_editText
as text
```

```
%        str2double(get(hObject,'String')) returns contents of
sigmaScale_editText as a double

%store the contents of sigmaScale_editText as a string. if the string
%is not a number then input will be empty

%lets the user select the region of confidence ellipse scale
input = str2num(get(hObject,'String'));

%checks to see if input is empty. if so, default sigmaScale_editText to
%five
if (isempty(input))
     set(hObject,'String','5')
end

% --- Executes during object creation, after setting all properties.
function sigmaScale_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to sigmaScale_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on selection change in popupmenu2.
function popupmenu2_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns popupmenu2 contents
as cell array
%        contents{get(hObject,'Value')} returns selected item from
popupmenu2



% --- Executes during object creation, after setting all properties.
function popupmenu2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called
% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on button press in pause_toggleButton.
function pause_toggleButton_Callback(hObject, eventdata, handles)
% hObject    handle to pause_toggleButton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of pause_toggleButton



function fpNoise_editText_Callback(hObject, eventdata, handles)
% hObject    handle to fpNoise_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of fpNoise_editText as
text
%        str2double(get(hObject,'String')) returns contents of
fpNoise_editText as a double
input = str2num(get(hObject,'String'));

%checks to see if input is empty. if so, default noise_editText to 0.00
if (isempty(input))
     set(hObject,'String','0.00')
end

% --- Executes during object creation, after setting all properties.
function fpNoise_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to fpNoise_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on button press in softLimiter_checkbox.
function softLimiter_checkbox_Callback(hObject, eventdata, handles)
% hObject    handle to softLimiter_checkbox (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of
softLimiter_checkbox



function softLimit_editText_Callback(hObject, eventdata, handles)
% hObject    handle to softLimit_editText (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of softLimit_editText
as text
%        str2double(get(hObject,'String')) returns contents of
softLimit_editText as a double


% --- Executes during object creation, after setting all properties.
function softLimit_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to softLimit_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function epsilon_editText_Callback(hObject, eventdata, handles)
% hObject    handle to epsilon_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of epsilon_editText as
text
%        str2double(get(hObject,'String')) returns contents of
epsilon_editText as a double


% --- Executes during object creation, after setting all properties.
function epsilon_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to epsilon_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function inputFile_editText_Callback(hObject, eventdata, handles)
% hObject    handle to inputFile_editText (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of inputFile_editText
as text
%        str2double(get(hObject,'String')) returns contents of
inputFile_editText as a double


% --- Executes during object creation, after setting all properties.
function inputFile_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to inputFile_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in popupmenu3.
function popupmenu3_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu3
contents as cell array
%        contents{get(hObject,'Value')} returns selected item from
popupmenu3


% --- Executes during object creation, after setting all properties.
function popupmenu3_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function bufferLength_editText_Callback(hObject, eventdata, handles)
% hObject    handle to bufferLength_editText (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of
bufferLength_editText as text
%        str2double(get(hObject,'String')) returns contents of
bufferLength_editText as a double


% --- Executes during object creation, after setting all properties.
function bufferLength_editText_CreateFcn(hObject, eventdata, handles)
% hObject    handle to bufferLength_editText (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

APPENDIX B

Surface_bearing_cple_3d_gui_v2.m

```
function [handles] = surface_bearing_cple_3d_gui_v2(handles)
%%%%%%%% This code will obtain the location trace of the airplane given
the velocities%%%%%%%%%%

%%%%%%%%%%% DO THIS TO GET VELOCITY FROM LOCATION
TRACE%%%%%%%%%%%%%%%%%%
%Knobs to turn
plots = get(handles.plots_checkbox,'Value');          %turns plots on
(1) or off(0)
bscale = 1;
scale=10^0;              %scale of the map
resolution=scale/200;  %x-resolution of interpolation
a=str2num(get(handles.noise_editText,'String'));   %Gaussian noise
parameter
f=str2num(get(handles.fpNoise_editText,'String'));   %Noise parameter
for flight path error
epsilon=str2num(get(handles.epsilon_editText,'String'));
%Epsilon for phase blips
N=str2num(get(handles.numRuns_editText,'String'));  %Number of times to
run simulation
s=[str2num(get(handles.emitterX_editText,'String'))
str2num(get(handles.emitterY_editText,'String'))
str2num(get(handles.emitterZ_editText,'String'))]';
%Position of emitter [x y z]
z0=s(3);  %known elevation value from GPS readings
window_size=str2num(get(handles.windowSize_editText,'String'));
%Size of block processing window
buffer_length=str2num(get(handles.bufferLength_editText,'String'));
%Size of input data buffer
axislimit = 0.0015*scale^2;   %sets limits of axis on ellipse plot. For
a scale of 1, .0015 works well.
sigma_scale = str2num(get(handles.sigmaScale_editText,'String'));
%parameter to scale the error ellipse by.
soft_limiter = get(handles.popupmenu3,'Value');            %selects
which soft limiter to use
soft_limit = str2num(get(handles.softLimit_editText,'String')); %user
input to determine how much to limit
inputname=get(handles.inputFile_editText,'String');   %name of the csv
input file to use
inputfile=[inputname '.csv'];


inputvectors = csvread(inputfile);  %reads in input points from CSV
file
inputsize = size(inputvectors);
```

```
numpaths = inputsize(1)/4;  %stores the number of flight paths for
later use

xx=[];
yy=[];
zz=[];
t1=[];

for h=1:4:inputsize(1)           %parses CSV input and stores to
component vectors
   xx = [xx; inputvectors(h,:)];
   yy = [yy; inputvectors(h+1,:)];
   zz = [zz; inputvectors(h+2,:)];
   t1 = [t1; inputvectors(h+3,:)];
end

xx = xx.*scale;  %multiplies by 10 because all points will be
yy = yy.*scale;  %between 0 and 1
zz = zz.*scale;

t=0:resolution:max(t1(:,length(t1))); %creates time vector

vx = zeros(numpaths,length(t));  %creates initial velocity matrix
vy = zeros(numpaths,length(t));
vz = zeros(numpaths,length(t));

for n=1:numpaths
    xi2=interp1(t1(n,:),xx(n,:),'linear','pp');
    xi(n,:)=ppval(xi2,t);

    yi2=interp1(t1(n,:),yy(n,:),'linear','pp');
    yi(n,:)=ppval(yi2,t);

    zi2=interp1(t1(n,:),zz(n,:),'linear','pp');  %this interpolates the
zi curve as well.
    zi(n,:)=ppval(zi2,t);

    for i=2:length(t)
         vx(n,i)=(xi(n,i)-xi(n,(i-1)))/(t(i)-t(i-1));  %calculates
velocity differential
         vy(n,i)=(yi(n,i)-yi(n,(i-1)))/(t(i)-t(i-1));
         vz(n,i)=(zi(n,i)-zi(n,(i-1)))/(t(i)-t(i-1));  %based on
position and time differences
    end
end

vs=sqrt(vy.^2+vx.^2);

%%%%%%%%%%%%%%%%%%%%%This "finds" the emitter with noise
added%%%%%%%%%%%%%%%%%%%%


r=zeros(3,length(t),numpaths);
```

```
for n=1:numpaths  %constructs flight heading from component vectors
 r(1,:,n) = xi(n,:);
 r(2,:,n) = yi(n,:);
 r(3,:,n) = zi(n,:);

 %adds GPS uncertainty factor to flight path
 rnoisy(:,:,n) = r(:,:,n) + f.*randn(size(r(:,:,n)));
    for i=1:length(r(1,:,n))
        b(1,i,n)=s(1)-r(1,i,n);
        b(2,i,n)=s(2)-r(2,i,n);   %constructs initial B vectors
        b(3,i,n)=s(3)-r(3,i,n);
    end
end

set(handles.dataPoints_staticText,'String',num2str(length(t)));
%displays number of data points to GUI

xguess = zeros(2,N,numpaths);   %preallocation of arrays, for speed
countmem = zeros(1,N);
verr=zeros(2,1,numpaths);
percent = zeros(1,N);
percent2 = zeros(1,N);
error=zeros(2,N,numpaths);
vol=zeros(N,length(b),numpaths);

% figure
for j=1:N
    pause(0.1)
    set(handles.currentRun_staticText,'String',num2str(j));

    xvec = zeros(2,length(b),numpaths); %More preallocation, for speed.

    bxnoisy = zeros(numpaths,buffer_length);
    bynoisy = zeros(numpaths,buffer_length);
    bznoisy = zeros(numpaths,buffer_length);

    mx = zeros(numpaths,buffer_length);
    my = zeros(numpaths,buffer_length);


     bn=zeros(3,2*buffer_length,numpaths);  %creates empty vectors.

    window_count=0;
    success=0;
    success2=0;
    verr=zeros(2,1,numpaths);
    errind=[];
    limitind=[];
    errthetavec=[];
    errphivec=[];
    count=zeros(1,n);
    countlimit=zeros(1,n);

    for k=1:(buffer_length)   %fills the buffer window
```

```
        for n=1:numpaths
        %adds noise to the component vectors. Noise in each direction
is
        %independent of noise in any other direction.
            bxnoisy(n,1:(buffer_length-1))=bxnoisy(n,2:buffer_length);
            nx=randn(1).*a.*abs(b(1,k,n)).^2;
            bxnoisy(n,buffer_length) = b(1,k,n)+nx;

            bynoisy(n,1:(buffer_length-1))=bynoisy(n,2:buffer_length);
            ny=randn(1).*a.*abs(b(2,k,n)).^2;
            bynoisy(n,buffer_length) = b(2,k,n)+ny;

            bznoisy(n,1:(buffer_length-1))=bznoisy(n,2:buffer_length);
            nz=randn(1).*a.*abs(b(3,k,n)).^2;
            bznoisy(n,buffer_length) = b(3,k,n)+nz;


        %Derives Q parameter by adding R and B
          qx=rnoisy(1,k,n)+bxnoisy(n,buffer_length);
          qy=rnoisy(2,k,n)+bynoisy(n,buffer_length);
          qz=rnoisy(3,k,n)+bznoisy(n,buffer_length);

          %Derives parameter 't' for use in surface projection
          t2=(z0-qz)/(r(3,k,n)-qz);

          %updates the data window
          mx(n,1:(buffer_length-1)) = mx(n,2:buffer_length);
          my(n,1:(buffer_length-1)) = my(n,2:buffer_length);
          %stores surface projection measurements into the window
          mx(n,buffer_length)=rnoisy(1,k,n)*t2+qx*(1-t2);
          my(n,buffer_length)=rnoisy(2,k,n)*t2+qy*(1-t2);

          %averages data in the data window in order to obtain emitter
          %estimate
          x0 = [mean(mx(n,(buffer_length-(k-1)):buffer_length))
mean(my(n,(buffer_length-(k-1)):buffer_length))]';

          %stores emitter estimate for use in covariance calculations
          xvec(:,k,n) = x0;


        end

    end


    for k=(buffer_length+1):length(b)


         for n=1:numpaths
        % adds noise to the component input vectors. Noise in each
        % direction is independent of the other directions.
            bxnoisy(n,1:(buffer_length-1))=bxnoisy(n,2:buffer_length);
            nx=randn(1).*a.*abs(b(1,k,n)).^2;
```

```
            bxnoisy(n,buffer_length) = b(1,k,n)+nx;

            bynoisy(n,1:(buffer_length-1))=bynoisy(n,2:buffer_length);
            ny=randn(1).*a.*abs(b(2,k,n)).^2;
            bynoisy(n,buffer_length) = b(2,k,n)+ny;

            bznoisy(n,1:(buffer_length-1))=bznoisy(n,2:buffer_length);
            nz=randn(1).*a.*abs(b(3,k,n)).^2;
            bznoisy(n,buffer_length) = b(3,k,n)+nz;

            %adds angle outliers based on pseudorandom number
generation
            [theta2 phi2 rho2]=cart2sph(bxnoisy(n,buffer_length),
bynoisy(n,buffer_length), bznoisy(n, buffer_length));

            decider=rand(1);
            if decider<=epsilon
                count(n) = count(n)+1;
                errind=[errind k];
                errtheta=pi*rand(1)-pi/2;
                theta2=theta2+errtheta;
                errphi=pi*rand(1)-pi/2;
                phi2=phi2+errphi;
                errthetavec=[errthetavec errtheta];
                errphivec=[errphivec errphi];
            end

            [bxnoisy(n,buffer_length) bynoisy(n,buffer_length)
bznoisy(n,buffer_length)]=sph2cart(theta2, phi2, rho2);


         end
         %plots flight paths and bearing vectors
        if plots==1
            if mod(k,round(window_size/4))==1
                %constructs axes and plots flight path in 3D
                axes(handles.axes1),
plot3(xi(1,1:k),yi(1,1:k),zi(1,1:k), 'r-' ,s(1),s(2),s(3),'o');
                hold on
                %plots past k/4 B vectors as illustration
                plot3([r(1,k,1)
r(1,k,1)+bscale*bxnoisy(1,window_size)], [r(2,k,1)
r(2,k,1)+bscale*bynoisy(1,window_size)], [r(3,k,1)
r(3,k,1)+bscale*bznoisy(1,window_size)]);
                plot3([r(1,k-round(window_size/4),1) r(1,k-
round(window_size/4),1)+bscale*bxnoisy(1,window_size-
round(window_size/4))], [r(2,k-round(window_size/4),1) r(2,k-
round(window_size/4),1)+bscale*bynoisy(1,window_size-
round(window_size/4))], [r(3,k-round(window_size/4),1) r(3,k-
round(window_size/4),1)+bscale*bznoisy(1,window_size-
round(window_size/4))]);
                plot3([r(1,k-round(window_size/2),1) r(1,k-
round(window_size/2),1)+bscale*bxnoisy(1,window_size-
round(window_size/2))], [r(2,k-round(window_size/2),1) r(2,k-
round(window_size/2),1)+bscale*bynoisy(1,window_size-
```

```
round(window_size/2))], [r(3,k-round(window_size/2),1) r(3,k-
round(window_size/2),1)+bscale*bznoisy(1,window_size-
round(window_size/2))]);
                plot3([r(1,k-round(3*window_size/4),1) r(1,k-
round(3*window_size/4),1)+bscale*bxnoisy(1,window_size-
round(3*window_size/4))], [r(2,k-round(3*window_size/4),1) r(2,k-
round(3*window_size/4),1)+bscale*bynoisy(1,window_size-
round(3*window_size/4))], [r(3,k-round(3*window_size/4),1) r(3,k-
round(3*window_size/4),1)+bscale*bznoisy(1,window_size-
round(3*window_size/4))]);
                plot3([r(1,k-window_size+1,1) r(1,k-
window_size+1,1)+bscale*bxnoisy(1,window_size-window_size+1)], [r(2,k-
window_size+1,1) r(2,k-window_size+1,1)+bscale*bynoisy(1,window_size-
window_size+1)], [r(3,k-window_size+1,1) r(3,k-
window_size+1,1)+bscale*bznoisy(1,window_size-window_size+1)]);

                %does the above for all subsequent flight paths
                if numpaths>1
                    for n=2:numpaths
                    plot3(xi(n,1:k),yi(n,1:k),zi(n,1:k), 'g-');

                    plot3([r(1,k,n)
r(1,k,n)+bscale*bxnoisy(n,window_size)], [r(2,k,n)
r(2,k,n)+bscale*bynoisy(n,window_size)], [r(3,k,n)
r(3,k,n)+bscale*bznoisy(n,window_size)]);
                    plot3([r(1,k-round(window_size/4),n) r(1,k-
round(window_size/4),n)+bscale*bxnoisy(n,window_size-
round(window_size/4))], [r(2,k-round(window_size/4),n) r(2,k-
round(window_size/4),n)+bscale*bynoisy(n,window_size-
round(window_size/4))], [r(3,k-round(window_size/4),n) r(3,k-
round(window_size/4),n)+bscale*bznoisy(n,window_size-
round(window_size/4))]);
                    plot3([r(1,k-round(window_size/2),n) r(1,k-
round(window_size/2),n)+bscale*bxnoisy(n,window_size-
round(window_size/2))], [r(2,k-round(window_size/2),n) r(2,k-
round(window_size/2),n)+bscale*bynoisy(n,window_size-
round(window_size/2))], [r(3,k-round(window_size/2),n) r(3,k-
round(window_size/2),n)+bscale*bznoisy(n,window_size-
round(window_size/2))]);
                    plot3([r(1,k-round(3*window_size/4),n) r(1,k-
round(3*window_size/4),n)+bscale*bxnoisy(n,window_size-
round(3*window_size/4))], [r(2,k-round(3*window_size/4),n) r(2,k-
round(3*window_size/4),n)+bscale*bynoisy(n,window_size-
round(3*window_size/4))], [r(3,k-round(3*window_size/4),n) r(3,k-
round(3*window_size/4),n)+bscale*bznoisy(n,window_size-
round(3*window_size/4))]);
                    plot3([r(1,k-window_size+1,n) r(1,k-
window_size+1,n)+bscale*bxnoisy(n,window_size-window_size+1)], [r(2,k-
window_size+1,n) r(2,k-window_size+1,n)+bscale*bynoisy(n,window_size-
window_size+1)], [r(3,k-window_size+1,n) r(3,k-
window_size+1,n)+bscale*bznoisy(n,window_size-window_size+1)]);

                    end
                end
                hold off
```

```
                %plots flight path in top-down 2D
                if mod(k,4)==1
                    axes(handles.axes3), axis([-scale*10 scale*10 -
scale*10 scale*10]);
                    hold on
                    axis manual
                    for n=1:numpaths
                      plot(xi(n,1:k),yi(n,1:k),s(1), s(2),'o');
                    end
                    hold off
                end
            end

        end

        for n=1:numpaths
            %Derives Q parameter by adding R and B.
          qx=rnoisy(1,k,n)+bxnoisy(n,buffer_length);
          qy=rnoisy(2,k,n)+bynoisy(n,buffer_length);
          qz=rnoisy(3,k,n)+bznoisy(n,buffer_length);

          %Derives parameter 't' for use in surface projection
          t2=(z0-qz)/(rnoisy(3,k,n)-qz);

          %sets limits for soft limiter
          xlimith=mean(mx(n,:))+soft_limit*std(mx(n,:));
          xlimitl=mean(mx(n,:))-soft_limit*std(mx(n,:));
          ylimith=mean(my(n,:))+soft_limit*std(my(n,:));
          ylimitl=mean(my(n,:))-soft_limit*std(my(n,:));

          %updates data window
          mx(n,1:(buffer_length-1)) = mx(n,2:buffer_length);
          my(n,1:(buffer_length-1)) = my(n,2:buffer_length);

          %reads in a new data measurement
          mx(n,buffer_length)=rnoisy(1,k,n)*t2+qx*(1-t2);
          my(n,buffer_length)=rnoisy(2,k,n)*t2+qy*(1-t2);

          %uses soft limiters to censor data

          %replacing limiter - replaces outlier with previous data
point to
          %preserve consistency.
          if soft_limiter==2
              if (mx(n,buffer_length) > xlimith) ||
(mx(n,buffer_length) < xlimitl) || (my(n,buffer_length) > ylimith) ||
(my(n,buffer_length) < ylimitl)
                  mx(n,buffer_length)=mx(n,buffer_length-1);
                  my(n,buffer_length)=my(n,buffer_length-1);
                  countlimit(n) = countlimit(n)+1;
                  limitind=[limitind k];
              end
              window_x=mx(n,(buffer_length-
window_size+1):buffer_length);
```

```
                window_y=my(n,(buffer_length-
window_size+1):buffer_length);

                %averaging limiter - removes from consideration one point
in
                %the window that is furthest from the average
            elseif soft_limiter==3
                window_x=mx(n,(buffer_length-window_size):buffer_length);
                window_y=my(n,(buffer_length-window_size):buffer_length);

                average_x=mean(window_x);
                average_y=mean(window_y);

                for kk=1:length(window_x)
                    dist(kk)=sqrt((window_x(kk)-
average_x)^2+(window_y(kk)-average_y)^2);
                end
                ind(k)=min(find(dist==max(dist)));
                window_x(ind(k))=[];
                window_y(ind(k))=[];
                if (isempty(errind)==0)&&((k-(window_size+1-
ind(k)))==errind(end))
                  limitind=[limitind k-(window_size+1-ind(k))];
                end
            else
                window_x=mx(n,(buffer_length-
window_size+1):buffer_length);
                window_y=my(n,(buffer_length-
window_size+1):buffer_length);

            end

            %averages data in window to find emitter estimate
            x0 = [mean(window_x) mean(window_y)]';
            %stores emitter estimate for covariance calculations
            xvec(:,k,n) = x0;
            %displays current estimate to GUI

set(handles.emitterXGuess_staticText,'String',num2str(x0(1)));

set(handles.emitterYGuess_staticText,'String',num2str(x0(2)));

            %calculates average error
            x1=rnoisy(1:2,k,n);
            x2=rnoisy(1:2,k,n)+b(1:2,k,n);


            t2=-dot(x1-x0,x2-x1)/(sqrt(sum((x2-x1).^2))).^2;

             v=x1+(x2-x1)*t2;
             eterm1=(v-x0);
             verr(:,:,n)=eterm1.^2+verr(:,:,n);

        end
```

```
%%%%%calculates error elipse and updates after every window shift%%%%


        plot_points=[];
        for n=1:numpaths
            x=xvec(1,(k-window_size):k,n);
            y=xvec(2,(k-window_size):k,n);

            sigxx=var(x);
            sigyy=var(y);

            sigxy=sum((x-mean(x)).*(y-mean(y)))/length(x);
            %covariance matrix
            A=[sigxx sigxy
                sigxy sigyy];
            %inverts covariance matrix
            Ai(:,:,n)=inv(A);
            %sets the center of the ellipse
            c(:,:,n)=[mean(x); mean(y)];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This code taken from Ellipse_plot.m
%  Copyright Nima Moshtagh
%  nima@seas.upenn.edu
%  University of Pennsylvania
%  Feb 1, 2007

            [U D V] = svd(Ai(:,:,n)/sigma_scale);

            majaxis = 1/sqrt(D(1,1));
            minaxis = 1/sqrt(D(2,2));

            if plots==1 && mod(k,4)==1
                theta = [0:1/20:2*pi+1/20];

                % Parametric equation of the ellipse
                %----------------------------------------
                state(1,:) = majaxis*cos(theta);
                state(2,:) = minaxis*sin(theta);

                % Coordinate transform
                %----------------------------------------
                X = V * state;
                X(1,:) = X(1,:) + c(1,:,n);
                X(2,:) = X(2,:) + c(2,:,n);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                %plots all data points in window for all flight paths
                axes(handles.axes2), plot(xvec(1,(k-window_size):k,n),
xvec(2,(k-window_size):k,n),'r+', s(1),s(2), 'o');
                hold on
                axis(handles.axes2, [(s(1)-axislimit) (s(1)+axislimit)
(s(2)-axislimit) (s(2)+axislimit)]);
                axis manual
                %plots confidence region ellipse
                plot(X(1,:),X(2,:));
```

```
            end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This code taken from Ellipse_plot.m
%  Copyright Nima Moshtagh
%  nima@seas.upenn.edu
%  University of Pennsylvania
%  Feb 1, 2007

            [U D V] = svd(Ai(:,:,n)/(sigma_scale*2));

            majaxis = 1/sqrt(D(1,1));
            minaxis = 1/sqrt(D(2,2));

            theta = [0 pi/2 pi 3*pi/2];

            % Parametric equation of the ellipse
            %----------------------------------------
            state2(1,:) = majaxis*cos(theta);
            state2(2,:) = minaxis*sin(theta);

            % Coordinate transform
            %----------------------------------------
            ppts = V * state2;
            ppts(1,:) = ppts(1,:) + c(1,:,n);
            ppts(2,:) = ppts(2,:) + c(2,:,n);
            plot_points = [plot_points; ppts];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        end
            %plots vertices of region of confidence parallelogram
            plot_points = [plot_points plot_points(:,1)];

            %determines region of confidence inclusion
            if numpaths > 1
                [Dx, Dy] = polybool('intersection', plot_points(1,:),
plot_points(2,:), plot_points(3,:), plot_points(4,:));
            else
                Dx=plot_points(1,:);
                Dy=plot_points(2,:);
            end

            if numpaths>2
               for n=3:numpaths
                  if(isempty(Dx)==0 && isempty(Dy)==0)
                     [Dx, Dy] = polybool('intersection', Dx, Dy,
plot_points(2*n-1,:), plot_points(2*n,:));
                  end
               end
            end

            [f, vp] = poly2fv(Dx, Dy);

            %plots error ellipses and parallelograms
        if plots==1 && mod(k,4)==1
```

```
            for n=1:numpaths
                plot(plot_points(2*n-1,:), plot_points(2*n,:),'g-');
            end
            plot(Dx, Dy, 'k-');
            axes(handles.axes2), hold off



            axes(handles.axes4), cla
            axis(handles.axes4, [(s(1)-axislimit) (s(1)+axislimit)
(s(2)-axislimit) (s(2)+axislimit)]);
            hold on
            axis manual
            plot(s(1), s(2), 'o');
            patch('Faces', f, 'Vertices', vp, 'FaceColor',
'r','EdgeColor', 'none')
            plot(s(1), s(2), 'o');
            plot(Dx, Dy, 'k-');
            hold off


            pause(0.1);

        end
        locemit = [s(1); s(2)];

        pflag = zeros(1,numpaths);
        for n=1:numpaths
            if ((locemit-c(:,:,n))'*Ai(:,:,n)*(locemit-
c(:,:,n))<sigma_scale)
                pflag(n) = 1;
            end
        end

        %displays to the GUI whether emitter is in ellipse
        if sum(pflag) == numpaths
            success = success+1;
            set(handles.inclusion_staticText,'String','Emitter in
ellipse');
            set(handles.inclusion_staticText,'BackgroundColor','g');
        else
            set(handles.inclusion_staticText,'String','Emitter NOT in
ellipse');
            set(handles.inclusion_staticText,'BackgroundColor','r');
        end

set(handles.percent_staticText,'String',num2str((success/(k-
buffer_length))*100));

            %displays to the GUI whether emitter is in polygon
        if isempty(Dx) == 0 && isempty(Dy) == 0
            if inpolygon(s(1), s(2), Dx, Dy)
                success2 = success2+1;
                set(handles.inclusion2_staticText,'String','Emitter in
polygon');
```

```
set(handles.inclusion2_staticText,'BackgroundColor','g');
            else
                set(handles.inclusion2_staticText,'String','Emitter NOT
in polygon');

set(handles.inclusion2_staticText,'BackgroundColor','r');
            end

set(handles.percent2_staticText,'String',num2str((success2/(k-
buffer_length))*100));
        end
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      while get(handles.pause_toggleButton,'Value')
          pause(1);
      end


    end

    %calculates and displays inclusion percentage for ellipse and
polygon
    for n=1:numpaths
    xguess(:,j,n) = xvec(:,k,n);
    percent(j) = (success*100)/(length(b)-buffer_length);
    percent2(j) = (success2*100)/(length(b)-buffer_length);
    error(:,j,n)=verr(:,:,n);
    end
    countmem(j,n)=count(n);
    errind
    limitind


end

  %calculates and displays average inclusion and average error
  xguess
  percent
  percent_avg=sum(percent)/N
  percent_avg2=sum(percent2)/N
  set(handles.averagePercent_staticText,'String',num2str(percent_avg));

set(handles.averagePercent2_staticText,'String',num2str(percent_avg2));
  error
  error_avg=[(sum(error(1,:,1))+sum(error(1,:,2)))/(2*N) ;
(sum(error(2,:,1))+ sum(error(2,:,2)))/(2*N)]
  set(handles.avgErrX_staticText,'String',num2str(error_avg(1)));
  set(handles.avgErrY_staticText,'String',num2str(error_avg(2)));

  %writes a CSV file containing the volumes of the ellipses
  if plots==1
        csvwrite('Volume.csv', vol);
  end
```

APPENDIX C

Surface_bearing_cple_3d gui_nowindow.m

```
function [handles] = surface_bearing_cple_3d_gui_nowindow(handles)
%%%%%%%% This code will obtain the location trace of the airplane given
the velocities%%%%%%%%%%

%%%%%%%%%%% DO THIS TO GET VELOCITY FROM LOCATION
TRACE%%%%%%%%%%%%%%%%%%
%Knobs to turn
plots = get(handles.plots_checkbox,'Value');          %turns plots on
(1) or off(0)
bscale = 1;
scale=10^0;              %scale of the map
resolution=scale/200;  %x-resolution of interpolation
a=str2num(get(handles.noise_editText,'String'));   %Gaussian noise
parameter
f=str2num(get(handles.fpNoise_editText,'String')); %Noise parameter for
flight path error
epsilon=str2num(get(handles.epsilon_editText,'String'));
%Epsilon for phase blips
N=str2num(get(handles.numRuns_editText,'String'));  %Number of times to
run simulation
s=[str2num(get(handles.emitterX_editText,'String'))
str2num(get(handles.emitterY_editText,'String'))
str2num(get(handles.emitterZ_editText,'String'))]';
%Position of emitter [x y z]
z0=s(3);
window_size=str2num(get(handles.windowSize_editText,'String'));
%Size of block processing window
buffer_length=str2num(get(handles.bufferLength_editText,'String'));
%Size of input data buffer
prime_length=10;
axislimit = 0.0015*scale^2;   %sets limits of axis on ellipse plot. For
a scale of 1, .0015 works well.
sigma_scale = str2num(get(handles.sigmaScale_editText,'String'));
%parameter to scale the error ellipse by.
soft_limiter = get(handles.popupmenu3,'Value');   %selects which soft
limiter to use
soft_limit = str2num(get(handles.softLimit_editText,'String'));   %user
input to determine how much to limit
inputfile=[get(handles.inputFile_editText,'String') '.csv'];  %name of
the csv input file to use

inputvectors = csvread(inputfile);  %reads in input points from CSV
file
inputsize = size(inputvectors);
numpaths = inputsize(1)/4;  %stores the number of flight paths for
later use
xx=[];
```

```
yy=[];
zz=[];
t1=[];

for h=1:4:inputsize(1)    %parses CSV input and stores to component
vectors
   xx = [xx; inputvectors(h,:)];
   yy = [yy; inputvectors(h+1,:)];
   zz = [zz; inputvectors(h+2,:)];
   t1 = [t1; inputvectors(h+3,:)];
end

xx = xx.*scale;  %multiplies by 10 because all points will be
yy = yy.*scale;  %between 0 and 1
zz = zz.*scale;

t=0:resolution:max(t1(:,length(t1))); %creates time vector

vx = zeros(numpaths,length(t));  %creates initial velocity matrix
vy = zeros(numpaths,length(t));
vz = zeros(numpaths,length(t));

for n=1:numpaths
    xi2=interp1(t1(n,:),xx(n,:),'linear','pp');
    xi(n,:)=ppval(xi2,t);

    yi2=interp1(t1(n,:),yy(n,:),'linear','pp');
    yi(n,:)=ppval(yi2,t);

    zi2=interp1(t1(n,:),zz(n,:),'linear','pp');  %this interpolates the
zi curve as well.
    zi(n,:)=ppval(zi2,t);

    for i=2:length(t)
         vx(n,i)=(xi(n,i)-xi(n,(i-1)))/(t(i)-t(i-1));  %calculates
velocity differential
         vy(n,i)=(yi(n,i)-yi(n,(i-1)))/(t(i)-t(i-1));
         vz(n,i)=(zi(n,i)-zi(n,(i-1)))/(t(i)-t(i-1));  %based on
position and time differences
    end
end

vs=sqrt(vy.^2+vx.^2);


%%%%%%%%%%%%%%%%%%%%This "finds" the emitter with noise
added%%%%%%%%%%%%%%%%%%%%

r=zeros(3,length(t),numpaths);
%airplane's path r(path,axis(1=x, 2=y, 3=z),element)
for n=1:numpaths      %constructs flight heading from component vectors
 r(1,:,n) = xi(n,:);
 r(2,:,n) = yi(n,:);
 r(3,:,n) = zi(n,:);
```

```
 %adds GPS uncertainty factor to flight path
 rnoisy(:,:,n) = r(:,:,n) + f.*randn(size(r(:,:,n)));
    for i=1:length(r(1,:,n))
        b(1,i,n)=s(1)-r(1,i,n);
        b(2,i,n)=s(2)-r(2,i,n);       %constructs initial B vectors
        b(3,i,n)=s(3)-r(3,i,n);
    end
end

set(handles.dataPoints_staticText,'String',num2str(length(t)));
%displays number of data points to GUI
%b is our array of vectors pointing towards the emitter

xguess = zeros(2,N,numpaths);       %preallocation of arrays, for speed
countmem = zeros(1,N);
verr=zeros(2,1,numpaths);
percent = zeros(1,N);
error=zeros(2,N,numpaths);
vol=zeros(N,length(b),numpaths);
count=0;

% figure
for j=1:N
    pause(0.1)
    set(handles.currentRun_staticText,'String',num2str(j));

    xvec = zeros(2,length(b),numpaths); %More preallocation, for speed.

    bxnoisy = zeros(numpaths,length(b));
    bynoisy = zeros(numpaths,length(b));
    bznoisy = zeros(numpaths,length(b));


    bn=zeros(3,2*length(b),numpaths);  %creates empty vectors.

    window_count=0;
    success=0;
    success2=0;
    verr=zeros(2,1,numpaths);
    errind=[];
    limitind=[];
    count=zeros(1,n);
    count2=0;
    countlimit=zeros(1,n);
    errthetavec=[];
    errphivec=[];

    for k=1:length(b)


         for n=1:numpaths
        % adds noise to the component input vectors. Noise in each
        % direction is independent of the other directions.
```

```
            nx=randn(1).*a.*abs(b(1,k,n)).^2;
            bxnoisy(n,k) = b(1,k,n)+nx;


            ny=randn(1).*a.*abs(b(2,k,n)).^2;
            bynoisy(n,k) = b(2,k,n)+ny;


            nz=randn(1).*a.*abs(b(3,k,n)).^2;
            bznoisy(n,k) = b(3,k,n)+nz;

            if k>prime_length
                 %adds angle outliers based on pseudorandom number
                 %generation
                [theta2 phi2 rho2]=cart2sph(bxnoisy(n,k), bynoisy(n,k),
bznoisy(n, k));

                decider=rand(1);
                if decider<=epsilon
                    count(n) = count(n)+1;
                    errind=[errind k];
                    errtheta=pi*rand(1)-pi/2;
                    theta2=theta2+errtheta;
                    errphi=pi*rand(1)-pi/2;
                    phi2=phi2+errphi;
                    errthetavec=[errthetavec errtheta];
                    errphivec=[errphivec errphi];
                end

                [bxnoisy(n,k) bynoisy(n,k)
bznoisy(n,k)]=sph2cart(theta2, phi2, rho2);
            end

         end
         %plots flight paths and bearing vectors
        if plots==1
            if (mod(k,round(window_size/4))==1 && k > window_size)
                 %constructs axes and plots flight path in 3D
                axes(handles.axes1),
plot3(xi(1,1:k),yi(1,1:k),zi(1,1:k), 'r-' ,s(1),s(2),s(3),'o');
                hold on
                %plots past k/4 B vectors as illustration
                plot3([r(1,k,1) r(1,k,1)+bscale*bxnoisy(1,k)],
[r(2,k,1) r(2,k,1)+bscale*bynoisy(1,k)], [r(3,k,1)
r(3,k,1)+bscale*bznoisy(1,k)]);
                plot3([r(1,k-round(window_size/4),1) r(1,k-
round(window_size/4),1)+bscale*bxnoisy(1,k-round(window_size/4))],
[r(2,k-round(window_size/4),1) r(2,k-
round(window_size/4),1)+bscale*bynoisy(1,k-round(window_size/4))],
[r(3,k-round(window_size/4),1) r(3,k-
round(window_size/4),1)+bscale*bznoisy(1,k-round(window_size/4))]);
                plot3([r(1,k-round(window_size/2),1) r(1,k-
round(window_size/2),1)+bscale*bxnoisy(1,k-round(window_size/2))],
[r(2,k-round(window_size/2),1) r(2,k-
round(window_size/2),1)+bscale*bynoisy(1,k-round(window_size/2))],
```

```
[r(3,k-round(window_size/2),1) r(3,k-
round(window_size/2),1)+bscale*bznoisy(1,k-round(window_size/2))]);
                plot3([r(1,k-round(3*window_size/4),1) r(1,k-
round(3*window_size/4),1)+bscale*bxnoisy(1,k-round(3*window_size/4))],
[r(2,k-round(3*window_size/4),1) r(2,k-
round(3*window_size/4),1)+bscale*bynoisy(1,k-round(3*window_size/4))],
[r(3,k-round(3*window_size/4),1) r(3,k-
round(3*window_size/4),1)+bscale*bznoisy(1,k-round(3*window_size/4))]);
                plot3([r(1,k-window_size+1,1) r(1,k-
window_size+1,1)+bscale*bxnoisy(1,k-window_size+1)], [r(2,k-
window_size+1,1) r(2,k-window_size+1,1)+bscale*bynoisy(1,k-
window_size+1)], [r(3,k-window_size+1,1) r(3,k-
window_size+1,1)+bscale*bznoisy(1,k-window_size+1)]);
                 %does the above for all subsequent flight paths
                if numpaths>1
                    for n=2:numpaths
                    plot3(xi(n,1:k),yi(n,1:k),zi(n,1:k), 'g-');

                    plot3([r(1,k,n) r(1,k,n)+bscale*bxnoisy(n,k)],
[r(2,k,n) r(2,k,n)+bscale*bynoisy(n,k)], [r(3,k,n)
r(3,k,n)+bscale*bznoisy(n,k)]);
                    plot3([r(1,k-round(window_size/4),n) r(1,k-
round(window_size/4),n)+bscale*bxnoisy(n,k-round(window_size/4))],
[r(2,k-round(window_size/4),n) r(2,k-
round(window_size/4),n)+bscale*bynoisy(n,k-round(window_size/4))],
[r(3,k-round(window_size/4),n) r(3,k-
round(window_size/4),n)+bscale*bznoisy(n,k-round(window_size/4))]);
                    plot3([r(1,k-round(window_size/2),n) r(1,k-
round(window_size/2),n)+bscale*bxnoisy(n,k-round(window_size/2))],
[r(2,k-round(window_size/2),n) r(2,k-
round(window_size/2),n)+bscale*bynoisy(n,k-round(window_size/2))],
[r(3,k-round(window_size/2),n) r(3,k-
round(window_size/2),n)+bscale*bznoisy(n,k-round(window_size/2))]);
                    plot3([r(1,k-round(3*window_size/4),n) r(1,k-
round(3*window_size/4),n)+bscale*bxnoisy(n,k-round(3*window_size/4))],
[r(2,k-round(3*window_size/4),n) r(2,k-
round(3*window_size/4),n)+bscale*bynoisy(n,k-round(3*window_size/4))],
[r(3,k-round(3*window_size/4),n) r(3,k-
round(3*window_size/4),n)+bscale*bznoisy(n,k-round(3*window_size/4))]);
                    plot3([r(1,k-window_size+1,n) r(1,k-
window_size+1,n)+bscale*bxnoisy(n,k-window_size+1)], [r(2,k-
window_size+1,n) r(2,k-window_size+1,n)+bscale*bynoisy(n,k-
window_size+1)], [r(3,k-window_size+1,n) r(3,k-
window_size+1,n)+bscale*bznoisy(n,k-window_size+1)]);

                    end
                end
                hold off

                %plots flight path in top-down 2D
                if mod(k,4)==1
                    axes(handles.axes3), axis([-scale*10 scale*10 -
scale*10 scale*10]);
                    hold on
                    axis manual
                    for n=1:numpaths
```

```
                    plot(xi(n,1:k),yi(n,1:k),s(1), s(2),'o');
                end
                hold off
            end
        end
    end

    for n=1:numpaths
        %Derives Q parameter by adding R and B.
      qx=rnoisy(1,k,n)+bxnoisy(n,k);
      qy=rnoisy(2,k,n)+bynoisy(n,k);
      qz=rnoisy(3,k,n)+bznoisy(n,k);

       %Derives parameter 't' for use in surface projection
      t2=(z0-qz)/(rnoisy(3,k,n)-qz);

      %sets limits for soft limiter
      if k>prime_length
          xlimith=mean(mx(n,:))+soft_limit*std(mx(n,:));
          xlimitl=mean(mx(n,:))-soft_limit*std(mx(n,:));
          ylimith=mean(my(n,:))+soft_limit*std(my(n,:));
          ylimitl=mean(my(n,:))-soft_limit*std(my(n,:));
      end

      mx(n,k)=rnoisy(1,k,n)*t2+qx*(1-t2);
      my(n,k)=rnoisy(2,k,n)*t2+qy*(1-t2);

      %uses soft limiters to censor data

      %replacing limiter - replaces outlier with previous data
point to
      %preserve consistency.
      if soft_limiter==2 && k>prime_length
          if (mx(n,k) > xlimith) || (mx(n,k) < xlimitl) || (my(n,k)
> ylimith) || (my(n,k) < ylimitl)
              mx(n,k)=mx(n,k-1);
              my(n,k)=my(n,k-1);
              countlimit(n) = countlimit(n)+1;
              limitind=[limitind k];
          end

          %averaging limiter - removes from consideration one point
in
          %the window that is furthest from the average
      elseif soft_limiter==3 && k>prime_length
          average_x=mean(mx(n,k-9:k));
          average_y=mean(my(n,k-9:k));

          for kk=1:10
              dist(kk)=sqrt((mx(n,k-(10-kk))-average_x)^2+(my(n,k-
(10-kk))-average_y)^2);
          end
          ind=min(find(dist==max(dist)));
          mx(n,k-(10-ind))=average_x;
          my(n,k-(10-ind))=average_y;
```

```
            if (isempty(errind)==0)&&(k-(10-ind)==errind(end))
              limitind=[limitind k-(10-ind)];
            end

        end

        %averages data to find emitter estimate
        x0 = [mean(mx(n,:)) mean(my(n,:))]';
        %stores emitter estimate for covariance calculations
        xvec(:,k,n) = x0;
        %displays current estimate to GUI

set(handles.emitterXGuess_staticText,'String',num2str(x0(1)));

set(handles.emitterYGuess_staticText,'String',num2str(x0(2)));

        %calculates average error
        x1=rnoisy(1:2,k,n);
        x2=rnoisy(1:2,k,n)+b(1:2,k,n);


        t2=-dot(x1-x0,x2-x1)/(sqrt(sum((x2-x1).^2))).^2;

         v=x1+(x2-x1)*t2;
         eterm1=(v-x0);
         verr(:,:,n)=eterm1.^2+verr(:,:,n);

      end


%%%%%calculates error elipse and updates after every window shift%%%%
      if count2<=prime_length
          count2 = count2+1;
      else
       plot_points=[];
       for n=1:numpaths
           x=xvec(1,prime_length:k,n);
           y=xvec(2,prime_length:k,n);

           sigxx=var(x);
           sigyy=var(y);

           sigxy=sum((x-mean(x)).*(y-mean(y)))/length(x);
           %covariance matrix
           A=[sigxx sigxy
               sigxy sigyy];
           %inverts covariance matrix
           Ai(:,:,n)=inv(A);
           %sets the center of the ellipse
           c(:,:,n)=[mean(x); mean(y)];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This code taken from Ellipse_plot.m
%  Copyright Nima Moshtagh
%  nima@seas.upenn.edu
%  University of Pennsylvania
```

```
%  Feb 1, 2007

            [U D V] = svd(Ai(:,:,n)/sigma_scale);

            majaxis = 1/sqrt(D(1,1));
            minaxis = 1/sqrt(D(2,2));

            if plots==1 && mod(k,4)==1
                theta = [0:1/20:2*pi+1/20];

                % Parametric equation of the ellipse
                %----------------------------------------
                state(1,:) = majaxis*cos(theta);
                state(2,:) = minaxis*sin(theta);

                % Coordinate transform
                %----------------------------------------
                X = V * state;
                X(1,:) = X(1,:) + c(1,:,n);
                X(2,:) = X(2,:) + c(2,:,n);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                %plots all data points in window for all flight paths
                axes(handles.axes2), plot(xvec(1,prime_length:k,n),
xvec(2,prime_length:k,n),'r+', s(1),s(2), 'o');
                hold on
                axis(handles.axes2, [(s(1)-axislimit) (s(1)+axislimit)
(s(2)-axislimit) (s(2)+axislimit)]);
                axis manual
                %plots confidence region ellipse
                plot(X(1,:),X(2,:));
            end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This code taken from Ellipse_plot.m
%  Copyright Nima Moshtagh
%  nima@seas.upenn.edu
%  University of Pennsylvania
%  Feb 1, 2007

            [U D V] = svd(Ai(:,:,n)/(sigma_scale*2));

            majaxis = 1/sqrt(D(1,1));
            minaxis = 1/sqrt(D(2,2));

            theta = [0 pi/2 pi 3*pi/2];

            % Parametric equation of the ellipse
            %----------------------------------------
            state2(1,:) = majaxis*cos(theta);
            state2(2,:) = minaxis*sin(theta);

            % Coordinate transform
            %----------------------------------------
            ppts = V * state2;
            ppts(1,:) = ppts(1,:) + c(1,:,n);
```

```
            ppts(2,:) = ppts(2,:) + c(2,:,n);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            plot_points = [plot_points; ppts];

        end
            %plots vertices of region of confidence parallelogram
            plot_points = [plot_points plot_points(:,1)];

            %determines region of confidence inclusion
            if numpaths > 1
                [Dx, Dy] = polybool('intersection', plot_points(1,:),
plot_points(2,:), plot_points(3,:), plot_points(4,:));
            else
                Dx=plot_points(1,:);
                Dy=plot_points(2,:);
            end

            if numpaths>2
               for n=3:numpaths
                  if(isempty(Dx)==0 && isempty(Dy)==0)
                     [Dx, Dy] = polybool('intersection', Dx, Dy,
plot_points(2*n-1,:), plot_points(2*n,:));
                  end
               end
            end

            [f, vp] = poly2fv(Dx, Dy);

            %plots error ellipses and parallelograms
        if plots==1 && mod(k,4)==1

            for n=1:numpaths
                plot(plot_points(2*n-1,:), plot_points(2*n,:),'g-');
            end
            plot(Dx, Dy, 'k-');
            axes(handles.axes2), hold off
            axes(handles.axes4), cla
            axis(handles.axes4, [(s(1)-axislimit) (s(1)+axislimit)
(s(2)-axislimit) (s(2)+axislimit)]);
            hold on
            axis manual
            plot(s(1), s(2), 'o');
            patch('Faces', f, 'Vertices', vp, 'FaceColor',
'r','EdgeColor', 'none')
            plot(s(1), s(2), 'o');
            plot(Dx, Dy, 'k-');
            hold off
            pause(0.1);

        end
        locemit = [s(1); s(2)];

        pflag = zeros(1,numpaths);
        for n=1:numpaths
```

```
            if ((locemit-c(:,:,n))'*Ai(:,:,n)*(locemit-
c(:,:,n))<sigma_scale)
                pflag(n) = 1;
            end
        end

        %displays to the GUI whether emitter is in ellipse
        if sum(pflag) == numpaths
            success = success+1;
            set(handles.inclusion_staticText,'String','Emitter in
ellipse');
            set(handles.inclusion_staticText,'BackgroundColor','g');
        else
            set(handles.inclusion_staticText,'String','Emitter NOT in
ellipse');
            set(handles.inclusion_staticText,'BackgroundColor','r');
        end

set(handles.percent_staticText,'String',num2str((success/(k-
prime_length))*100));

            %displays to the GUI whether emitter is in polygon
        if isempty(Dx) == 0 && isempty(Dy) == 0
            if inpolygon(s(1), s(2), Dx, Dy)
                success2 = success2+1;
                set(handles.inclusion2_staticText,'String','Emitter in
polygon');

set(handles.inclusion2_staticText,'BackgroundColor','g');
            else
                set(handles.inclusion2_staticText,'String','Emitter NOT
in polygon');

set(handles.inclusion2_staticText,'BackgroundColor','r');
            end

set(handles.percent2_staticText,'String',num2str((success2/(k-
prime_length))*100));
        end
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       end
      while get(handles.pause_toggleButton,'Value')
          pause(1);
      end
    end
    count2=0;

    %calculates and displays inclusion percentage for ellipse and
polygon
    for n=1:numpaths
    xguess(:,j,n) = xvec(:,k,n);
    percent(j) = (success*100)/(length(b)-prime_length);
    percent2(j) = (success2*100)/(length(b)-prime_length);
    error(:,j,n)=verr(:,:,n);
    end
    countmem(j,n)=count(n);
```

```
    errind
    limitind

end

  %calculates and displays average inclusion and average error
  xguess
  percent
  percent_avg=(sum(percent)/N)
  percent_avg2=sum(percent2)/N
  set(handles.averagePercent_staticText,'String',num2str(percent_avg));

set(handles.averagePercent2_staticText,'String',num2str(percent_avg2));
  error
  error_avg=[(sum(error(1,:,1))+sum(error(1,:,2)))/(2*N) ;
(sum(error(2,:,1))+ sum(error(2,:,2)))/(2*N)]
  set(handles.avgErrX_staticText,'String',num2str(error_avg(1)));
  set(handles.avgErrY_staticText,'String',num2str(error_avg(2)));


  if plots==1
        csvwrite('Volume.csv', vol);
  end
```

APPENDIX D

Surface_bearing_kalman_3d gui.m

```
function [handles] = surface_bearing_kalman_3d_gui(handles)
%%%%%%%% This code will obtain the location trace of the airplane given
the velocities%%%%%%%%%%

%%%%%%%%%%% DO THIS TO GET VELOCITY FROM LOCATION
TRACE%%%%%%%%%%%%%%%%%%
%Knobs to turn
plots = get(handles.plots_checkbox,'Value');          %turns plots on
(1) or off(0)
bscale = 1;
scale=10^0;              %scale of the map
resolution=scale/200;  %x-resolution of interpolation
a=str2num(get(handles.noise_editText,'String'));   %Gaussian noise
parameter
f=str2num(get(handles.fpNoise_editText,'String'));   %Noise parameter
for flight path error
epsilon=str2num(get(handles.epsilon_editText,'String'));
%Epsilon for phase blips
N=str2num(get(handles.numRuns_editText,'String'));  %Number of times to
run simulation
s=[str2num(get(handles.emitterX_editText,'String'))
str2num(get(handles.emitterY_editText,'String'))
str2num(get(handles.emitterZ_editText,'String'))]';
%Position of emitter [x y z]
z0=s(3);
window_size=str2num(get(handles.windowSize_editText,'String'));
%Size of block processing window
buffer_length=str2num(get(handles.bufferLength_editText,'String'));
%Size of input data buffer
axislimit = 0.0015*scale^2;   %sets limits of axis on ellipse plot. For
a scale of 1, .0015 works well.
sigma_scale = str2num(get(handles.sigmaScale_editText,'String'));
%parameter to scale the error ellipse by.
soft_limiter = get(handles.popupmenu3,'Value');            %selects
which soft limiter to use
soft_limit = str2num(get(handles.softLimit_editText,'String')); %user
input to determine how much to limit
inputname=get(handles.inputFile_editText,'String');   %name of the csv
input file to use


inputvectors = csvread(inputfile);  %reads in input points from CSV
file
inputsize = size(inputvectors);
numpaths = inputsize(1)/4;  %stores the number of flight paths for
later use
```

```
xx=[];
yy=[];
zz=[];
t1=[];

for h=1:4:inputsize(1)    %parses CSV input and stores to component
vectors
   xx = [xx; inputvectors(h,:)];
   yy = [yy; inputvectors(h+1,:)];
   zz = [zz; inputvectors(h+2,:)];
   t1 = [t1; inputvectors(h+3,:)];
end

xx = xx.*scale;  %multiplies by 10 because all points will be
yy = yy.*scale;  %between 0 and 1
zz = zz.*scale;

t=0:resolution:max(t1(:,length(t1))); %creates time vector

vx = zeros(numpaths,length(t));  %creates initial velocity matrix
vy = zeros(numpaths,length(t));
vz = zeros(numpaths,length(t));

for n=1:numpaths
    xi2=interp1(t1(n,:),xx(n,:),'linear','pp');
    xi(n,:)=ppval(xi2,t);

    yi2=interp1(t1(n,:),yy(n,:),'linear','pp');
    yi(n,:)=ppval(yi2,t);

    zi2=interp1(t1(n,:),zz(n,:),'linear','pp');  %this interpolates the
zi curve as well.
    zi(n,:)=ppval(zi2,t);

    for i=2:length(t)
         vx(n,i)=(xi(n,i)-xi(n,(i-1)))/(t(i)-t(i-1));  %calculates
velocity differential
         vy(n,i)=(yi(n,i)-yi(n,(i-1)))/(t(i)-t(i-1));
         vz(n,i)=(zi(n,i)-zi(n,(i-1)))/(t(i)-t(i-1));  %based on
position and time differences
    end
end

vs=sqrt(vy.^2+vx.^2);


%%%%%%%%%%%%%%%%%%%%%This "finds" the emitter with noise
added%%%%%%%%%%%%%%%%%%%%

r=zeros(3,length(t),numpaths);
%airplane's path r(path,axis(1=x, 2=y, 3=z),element)
for n=1:numpaths
 r(1,:,n) = xi(n,:);       %constructs flight heading from component
vectors
```

```
 r(2,:,n) = yi(n,:);
 r(3,:,n) = zi(n,:);

 %adds GPS uncertainty factor to flight path
 rnoisy(:,:,n) = r(:,:,n) + f.*randn(size(r(:,:,n)));
    for i=1:length(r(1,:,n))
        b(1,i,n)=s(1)-r(1,i,n);
        b(2,i,n)=s(2)-r(2,i,n);     %constructs initial B vectors
        b(3,i,n)=s(3)-r(3,i,n);
    end
end

set(handles.dataPoints_staticText,'String',num2str(length(t)));
%displays number of data points to GUI
%b is our array of vectors pointing towards the emitter

xguess = zeros(2,N,numpaths);       %preallocation of arrays, for speed
countmem = zeros(1,N);
verr=zeros(2,1,numpaths);
percent = zeros(1,N);
percent2 = zeros(1,N);
error=zeros(2,N,numpaths);
vol=zeros(N,length(b),numpaths);
Rw=zeros(numpaths,length(b));
% figure
for j=1:N
    pause(0.1)
    set(handles.currentRun_staticText,'String',num2str(j));

    xvec = zeros(2,length(b),numpaths); %More preallocation, for speed.

    bxnoisy = zeros(numpaths,window_size);
    bynoisy = zeros(numpaths,window_size);
    bznoisy = zeros(numpaths,window_size);

    mx = zeros(numpaths,length(r));
    my = zeros(numpaths,length(r));


    bn=zeros(3,2*window_size,numpaths);  %creates empty vectors.

    window_count=0;
    success=0;
    success2=0;
    verr=zeros(2,1,numpaths);
    errind=[];
    limitind=[];
    count=zeros(1,n);
    countlimit=zeros(1,n);
    errthetavec=[];
    errphivec=[];


    for k=1:(window_size)   %computes some least-squares data for
initialization
```

```
        for n=1:numpaths
        %adds noise to the component vectors. Noise in each direction
is
        %independent of noise in any other direction.
            bxnoisy(n,1:(window_size-1))=bxnoisy(n,2:window_size);
            nx=randn(1).*a.*abs(b(1,k,n)).^2;
            bxnoisy(n,window_size) = b(1,k,n)+nx;

            bynoisy(n,1:(window_size-1))=bynoisy(n,2:window_size);
            ny=randn(1).*a.*abs(b(2,k,n)).^2;
            bynoisy(n,window_size) = b(2,k,n)+ny;

            bznoisy(n,1:(window_size-1))=bznoisy(n,2:window_size);
            nz=randn(1).*a.*abs(b(3,k,n)).^2;
            bznoisy(n,window_size) = b(3,k,n)+nz;


        %Derives Q parameter by adding R and B
          qx=rnoisy(1,k,n)+bxnoisy(n,window_size);
          qy=rnoisy(2,k,n)+bynoisy(n,window_size);
          qz=rnoisy(3,k,n)+bznoisy(n,window_size);

          %Derives parameter 't' for use in surface projection
          t2=(z0-qz)/(r(3,k,n)-qz);

           %stores surface projection measurements into initialization
           %vector
          mx(n,k)=rnoisy(1,k,n)*t2+qx*(1-t2);
          my(n,k)=rnoisy(2,k,n)*t2+qy*(1-t2);

           %stores emitter estimate for use in covariance calculations
          xvec(:,k,n) = [mx(n,k) my(n,k)]';
        end

    end

  yhat=xvec;   %yhat initialization for kalman filter

    for n=1:numpaths
        vxh(n)=var(mx(n,:)); %variance of X (sigma-x)
        vyh(n)=var(my(n,:)); %variance of Y (sigma-x)
        Ry(:,:,n)=[vxh(n) 0  %a priori covariance matrix
                   0 vyh(n)];
    end
    Rw(:,1:window_size)=ones(numpaths,window_size);  %initializes a
posteriori covariance to 1

    Rv=.000015;  %sigma-e, the variance of error

    for k=(window_size+1):length(b)


         for n=1:numpaths
        % adds noise to the component input vectors. Noise in each
```

```
        % direction is independent of the other directions.
            bxnoisy(n,1:(window_size-1))=bxnoisy(n,2:window_size);
            nx=randn(1).*a.*abs(b(1,k,n)).^2;
            bxnoisy(n,window_size) = b(1,k,n)+nx;

            bynoisy(n,1:(window_size-1))=bynoisy(n,2:window_size);
            ny=randn(1).*a.*abs(b(2,k,n)).^2;
            bynoisy(n,window_size) = b(2,k,n)+ny;

            bznoisy(n,1:(window_size-1))=bznoisy(n,2:window_size);
            nz=randn(1).*a.*abs(b(3,k,n)).^2;
            bznoisy(n,window_size) = b(3,k,n)+nz;

            %adds angle outliers based on pseudorandom number
generation
            [theta2 phi2 rho2]=cart2sph(bxnoisy(n,window_size),
bynoisy(n,window_size), bznoisy(n, window_size));

            decider=rand(1);
            if decider<=epsilon
                count(n) = count(n)+1;
                errind=[errind k];
                errtheta=pi*rand(1)-pi/2;
                theta2=theta2+errtheta;
                errphi=pi*rand(1)-pi/2;
                phi2=phi2+errphi;
                errthetavec=[errthetavec errtheta];
                errphivec=[errphivec errphi];
            end

            [bxnoisy(n,window_size) bynoisy(n,window_size)
bznoisy(n,window_size)]=sph2cart(theta2, phi2, rho2);


         end
         %plots flight paths and bearing vectors
        if plots==1
            if mod(k,round(window_size/4))==1
                %constructs axes and plots flight path in 3D
                axes(handles.axes1),
plot3(xi(1,1:k),yi(1,1:k),zi(1,1:k), 'r-' ,s(1),s(2),s(3),'o');
                hold on
                %plots past k/4 B vectors as illustration
                plot3([r(1,k,1)
r(1,k,1)+bscale*bxnoisy(1,window_size)], [r(2,k,1)
r(2,k,1)+bscale*bynoisy(1,window_size)], [r(3,k,1)
r(3,k,1)+bscale*bznoisy(1,window_size)]);
                plot3([r(1,k-round(window_size/4),1) r(1,k-
round(window_size/4),1)+bscale*bxnoisy(1,window_size-
round(window_size/4))], [r(2,k-round(window_size/4),1) r(2,k-
round(window_size/4),1)+bscale*bynoisy(1,window_size-
round(window_size/4))], [r(3,k-round(window_size/4),1) r(3,k-
round(window_size/4),1)+bscale*bznoisy(1,window_size-
round(window_size/4))]);
```

```
                plot3([r(1,k-round(window_size/2),1) r(1,k-
round(window_size/2),1)+bscale*bxnoisy(1,window_size-
round(window_size/2))], [r(2,k-round(window_size/2),1) r(2,k-
round(window_size/2),1)+bscale*bynoisy(1,window_size-
round(window_size/2))], [r(3,k-round(window_size/2),1) r(3,k-
round(window_size/2),1)+bscale*bznoisy(1,window_size-
round(window_size/2))]);
                plot3([r(1,k-round(3*window_size/4),1) r(1,k-
round(3*window_size/4),1)+bscale*bxnoisy(1,window_size-
round(3*window_size/4))], [r(2,k-round(3*window_size/4),1) r(2,k-
round(3*window_size/4),1)+bscale*bynoisy(1,window_size-
round(3*window_size/4))], [r(3,k-round(3*window_size/4),1) r(3,k-
round(3*window_size/4),1)+bscale*bznoisy(1,window_size-
round(3*window_size/4))]);
                plot3([r(1,k-window_size+1,1) r(1,k-
window_size+1,1)+bscale*bxnoisy(1,window_size-window_size+1)], [r(2,k-
window_size+1,1) r(2,k-window_size+1,1)+bscale*bynoisy(1,window_size-
window_size+1)], [r(3,k-window_size+1,1) r(3,k-
window_size+1,1)+bscale*bznoisy(1,window_size-window_size+1)]);

                %does the above for all subsequent flight paths
                if numpaths>1
                    for n=2:numpaths
                    plot3(xi(n,1:k),yi(n,1:k),zi(n,1:k), 'g-');

                    plot3([r(1,k,n)
r(1,k,n)+bscale*bxnoisy(n,window_size)], [r(2,k,n)
r(2,k,n)+bscale*bynoisy(n,window_size)], [r(3,k,n)
r(3,k,n)+bscale*bznoisy(n,window_size)]);
                    plot3([r(1,k-round(window_size/4),n) r(1,k-
round(window_size/4),n)+bscale*bxnoisy(n,window_size-
round(window_size/4))], [r(2,k-round(window_size/4),n) r(2,k-
round(window_size/4),n)+bscale*bynoisy(n,window_size-
round(window_size/4))], [r(3,k-round(window_size/4),n) r(3,k-
round(window_size/4),n)+bscale*bznoisy(n,window_size-
round(window_size/4))]);
                    plot3([r(1,k-round(window_size/2),n) r(1,k-
round(window_size/2),n)+bscale*bxnoisy(n,window_size-
round(window_size/2))], [r(2,k-round(window_size/2),n) r(2,k-
round(window_size/2),n)+bscale*bynoisy(n,window_size-
round(window_size/2))], [r(3,k-round(window_size/2),n) r(3,k-
round(window_size/2),n)+bscale*bznoisy(n,window_size-
round(window_size/2))]);
                    plot3([r(1,k-round(3*window_size/4),n) r(1,k-
round(3*window_size/4),n)+bscale*bxnoisy(n,window_size-
round(3*window_size/4))], [r(2,k-round(3*window_size/4),n) r(2,k-
round(3*window_size/4),n)+bscale*bynoisy(n,window_size-
round(3*window_size/4))], [r(3,k-round(3*window_size/4),n) r(3,k-
round(3*window_size/4),n)+bscale*bznoisy(n,window_size-
round(3*window_size/4))]);
                    plot3([r(1,k-window_size+1,n) r(1,k-
window_size+1,n)+bscale*bxnoisy(n,window_size-window_size+1)], [r(2,k-
window_size+1,n) r(2,k-window_size+1,n)+bscale*bynoisy(n,window_size-
window_size+1)], [r(3,k-window_size+1,n) r(3,k-
window_size+1,n)+bscale*bznoisy(n,window_size-window_size+1)]);
```

```
                    end
                end
                hold off
            %plots flight path in top-down 2D
                if mod(k,4)==0
                    axes(handles.axes3), axis([-scale*10 scale*10 -
scale*10 scale*10]);
                    hold on
                    axis manual
                    for n=1:numpaths
                      plot(xi(n,1:k),yi(n,1:k),s(1), s(2),'o');
                    end
                    hold off
                end
            end
        end

        for n=1:numpaths

            %Derives Q parameter by adding R and B.
          qx=rnoisy(1,k,n)+bxnoisy(n,window_size);
          qy=rnoisy(2,k,n)+bynoisy(n,window_size);
          qz=rnoisy(3,k,n)+bznoisy(n,window_size);

          %Derives parameter 't' for use in surface projection
          t2=(z0-qz)/(rnoisy(3,k,n)-qz);

          %creates H matrix for use in Kalman filter
          H=[1-t2 1-t2];

          %sets limits for soft limiter

xlimith=mean(mx(n,window_size:k))+soft_limit*std(mx(n,window_size:k));
          xlimitl=mean(mx(n,window_size:k))-
soft_limit*std(mx(n,window_size:k));

ylimith=mean(my(n,window_size:k))+soft_limit*std(my(n,window_size:k));
          ylimitl=mean(my(n,window_size:k))-
soft_limit*std(my(n,window_size:k));

          %reads in a new data measurement
          mx(n,k)=rnoisy(1,k,n)*t2+qx*(1-t2);
          my(n,k)=rnoisy(2,k,n)*t2+qy*(1-t2);

          %uses soft limiter to censor data

          %replacing limiter - replaces outlier with previous data
point to
          %preserve consistency.
          if soft_limiter==2
              if (mx(n,k) > xlimith) || (mx(n,k) < xlimitl) || (my(n,k)
> ylimith) || (my(n,k) < ylimitl)
                  mx(n,k)=mx(n,k-1);
                  my(n,k)=my(n,k-1);
                  countlimit(n) = countlimit(n)+1;
```

```
                limitind=[limitind k];
            end

            %averaging limiter - removes from consideration one point
in
            %the window that is furthest from the average
        elseif soft_limiter==3
            average_x=mean(mx(n,k-9:k));
            average_y=mean(my(n,k-9:k));

            for kk=1:10
                dist(kk)=sqrt((mx(n,k-(10-kk))-average_x)^2+(my(n,k-
(10-kk))-average_y)^2);
            end
            ind=min(find(dist==max(dist)));
            if mx(n,k-(10-ind)) > mx(n,k-(10-ind)-1)
                mx(n,k-(10-ind)) = mx(n,k-(10-ind)-1) + Ry(1,1,n);
            else
                mx(n,k-(10-ind)) = mx(n,k-(10-ind)-1) - Ry(1,1,n);
            end

            if my(n,k-(10-ind)) > my(n,k-(10-ind)-1)
                my(n,k-(10-ind)) = my(n,k-(10-ind)-1) + Ry(2,2,n);
            else
                my(n,k-(10-ind)) = my(n,k-(10-ind)-1) - Ry(2,2,n);
            end

            if (isempty(errind)==0)&&(k-(10-ind)==errind(end))
              limitind=[limitind k-(10-ind)];
            end

        end

        %xhat is the new data reading
        xhat=[mx(n,k) my(n,k)]';

        %computes the kalman gain using the a priori and previous a
        %posteriori covariance measurements
        K=Ry(:,:,n)*H'*inv(Rw(n,k-1));

        %updates the a posteriori covariance using H, Ry and Rv
        Rw(n,k)=H*Ry(:,:,n)*H'+Rv;

        %computes new estimate using previous estimate and the new
data
        %measurement, weighted by the Kalman gain
        yhat(:,k,n) = yhat(:,k-1,n) + K.*(xhat-yhat(:,k-1,n));

        %updates the a priori covariance for use in the next
iteration
        Ry(:,:,n)=[eye(size(Ry(:,:,n)))-K*H]*Ry(:,:,n);

        %most recent estimate
        x0 = [yhat(1,k,n) yhat(2,k,n)]';
```

```
          %stores most recent estimate for use in covariance
calculations
          xvec(:,k,n) = x0;

          %displays most recent estimate

set(handles.emitterXGuess_staticText,'String',num2str(x0(1)));

set(handles.emitterYGuess_staticText,'String',num2str(x0(2)));

          %calculates average error
          x1=rnoisy(1:2,k,n);
          x2=rnoisy(1:2,k,n)+b(1:2,k,n);

          t2=-dot(x1-x0,x2-x1)/(sqrt(sum((x2-x1).^2))).^2;

           v=x1+(x2-x1)*t2;
           eterm1=(v-x0);
           verr(:,:,n)=eterm1.^2+verr(:,:,n);

        end


%%%%%calculates error elipse and updates after every window shift%%%%


        plot_points=[];
        for n=1:numpaths
            x=xvec(1,(window_size):k,n);
            y=xvec(2,(window_size):k,n);

            sigxx=var(x);
            sigyy=var(y);
            sigxy=sum((x-mean(x)).*(y-mean(y)))/length(x);
            %covariance matrix
            A=[sigxx sigxy
                sigxy sigyy];
            %inverts covariance matrix
            Ai(:,:,n)=inv(A);
            %sets the center of the ellipse
            c(:,:,n)=[mean(x); mean(y)];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This code taken from Ellipse_plot.m
%  Copyright Nima Moshtagh
%  nima@seas.upenn.edu
%  University of Pennsylvania
%  Feb 1, 2007
            [U D V] = svd(Ai(:,:,n)/sigma_scale);

            majaxis = 1/sqrt(D(1,1));
            minaxis = 1/sqrt(D(2,2));

            if plots==1 && mod(k,4)==0
```

```
                theta = [0:1/20:2*pi+1/20];

                % Parametric equation of the ellipse
                %---------------------------------------
                state(1,:) = majaxis*cos(theta);
                state(2,:) = minaxis*sin(theta);

                % Coordinate transform
                %---------------------------------------
                X = V * state;
                X(1,:) = X(1,:) + c(1,:,n);
                X(2,:) = X(2,:) + c(2,:,n);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                %plots all data points in window for all flight paths
                axes(handles.axes2), plot(xvec(1,(k-window_size):k,n),
xvec(2,(k-window_size):k,n),'r+', s(1),s(2), 'o');
                hold on
                axis(handles.axes2, [(s(1)-axislimit) (s(1)+axislimit)
(s(2)-axislimit) (s(2)+axislimit)]);
                axis manual
                %plots confidence region ellipse
                plot(X(1,:),X(2,:));
            end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This code taken from Ellipse_plot.m
%  Copyright Nima Moshtagh
%  nima@seas.upenn.edu
%  University of Pennsylvania
%  Feb 1, 2007

            [U D V] = svd(Ai(:,:,n)/(sigma_scale*2));

            majaxis = 1/sqrt(D(1,1));
            minaxis = 1/sqrt(D(2,2));

            theta = [0 pi/2 pi 3*pi/2];

            % Parametric equation of the ellipse
            %---------------------------------------
            state2(1,:) = majaxis*cos(theta);
            state2(2,:) = minaxis*sin(theta);

            % Coordinate transform
            %---------------------------------------
            ppts = V * state2;
            ppts(1,:) = ppts(1,:) + c(1,:,n);
            ppts(2,:) = ppts(2,:) + c(2,:,n);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            plot_points = [plot_points; ppts];

        end
        %plots vertices of region of confidence parallelogram
            plot_points = [plot_points plot_points(:,1)];
```

```
            %determines region of confidence inclusion
            if numpaths > 1
                [Dx, Dy] = polybool('intersection', plot_points(1,:),
plot_points(2,:), plot_points(3,:), plot_points(4,:));
            else
                Dx=plot_points(1,:);
                Dy=plot_points(2,:);
            end

            if numpaths>2
               for n=3:numpaths
                  if(isempty(Dx)==0 && isempty(Dy)==0)
                     [Dx, Dy] = polybool('intersection', Dx, Dy,
plot_points(2*n-1,:), plot_points(2*n,:));
                  end
               end
            end

            [f, vp] = poly2fv(Dx, Dy);

            %plots error ellipses and parallelograms
        if plots==1 && mod(k,4)==1

            for n=1:numpaths
                plot(plot_points(2*n-1,:), plot_points(2*n,:),'g-');
            end
            plot(Dx, Dy, 'k-');
            axes(handles.axes2), hold off
            axes(handles.axes4), cla
            axis(handles.axes4, [(s(1)-axislimit) (s(1)+axislimit)
(s(2)-axislimit) (s(2)+axislimit)]);
            hold on
            axis manual
            plot(s(1), s(2), 'o');
            patch('Faces', f, 'Vertices', vp, 'FaceColor',
'r','EdgeColor', 'none')
            plot(s(1), s(2), 'o');
            plot(Dx, Dy, 'k-');
            hold off
            pause(0.1);

        end
        locemit = [s(1); s(2)];

        pflag = zeros(1,numpaths);
        for n=1:numpaths
            if ((locemit-c(:,:,n))'*Ai(:,:,n)*(locemit-
c(:,:,n))<sigma_scale)
                pflag(n) = 1;
            end
        end

        %displays to the GUI whether emitter is in ellipse
        if sum(pflag) == numpaths
            success = success+1;
```

```
            set(handles.inclusion_staticText,'String','Emitter in
ellipse');
            set(handles.inclusion_staticText,'BackgroundColor','g');
        else
            set(handles.inclusion_staticText,'String','Emitter NOT in
ellipse');
            set(handles.inclusion_staticText,'BackgroundColor','r');
        end

set(handles.percent_staticText,'String',num2str((success/(k-
window_size))*100));

            %displays to the GUI whether emitter is in polygon
        if isempty(Dx) == 0 && isempty(Dy) == 0
            if inpolygon(s(1), s(2), Dx, Dy)
                success2 = success2+1;
                set(handles.inclusion2_staticText,'String','Emitter in
polygon');

set(handles.inclusion2_staticText,'BackgroundColor','g');
            else
                set(handles.inclusion2_staticText,'String','Emitter NOT
in polygon');

set(handles.inclusion2_staticText,'BackgroundColor','r');
            end

set(handles.percent2_staticText,'String',num2str((success2/(k-
window_size))*100));
        end
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      while get(handles.pause_toggleButton,'Value')
          pause(1);
      end

    end

     %calculates and displays inclusion percentage for ellipse and
polygon
    for n=1:numpaths
    xguess(:,j,n) = xvec(:,k,n);
    percent(j) = (success*100)/(length(b)-window_size);
    percent2(j) = (success2*100)/(length(b)-window_size);
    error(:,j,n)=verr(:,:,n);
    end
    countmem(j,n)=count(n);
    errind
    limitind

end
%calculates and displays average inclusion and average error
  xguess
  percent
  percent_avg=sum(percent)/N
  percent_avg2=sum(percent2)/N
  set(handles.averagePercent_staticText,'String',num2str(percent_avg));
```

```
set(handles.averagePercent2_staticText,'String',num2str(percent_avg2));
  error
  error_avg=[(sum(error(1,:,1))+sum(error(1,:,2)))/(2*N) ;
(sum(error(2,:,1))+ sum(error(2,:,2)))/(2*N)]
  set(handles.avgErrX_staticText,'String',num2str(error_avg(1)));
  set(handles.avgErrY_staticText,'String',num2str(error_avg(2)));

   %writes a CSV file containing the volumes of the ellipses
  if plots==1
        csvwrite('Volume.csv', vol);
  end
```

BIBLIOGRAPHY

[1] Nickens Okello, "Emitter Geolocation with Multiple UAVs," in *9th International Conference on Information Fusion*, Florence, 2006, p. 1.

[2] Paul Scerri, Robert Glinton, Sean Owens, David Scerri, and Katia Sycara, "Geolocation of RF Emitters by Many UAVs," Carnegie Mellon University, Pittsburch, Paper 2007.

[3] Herndon H Jenkins, *Small-Aperture Radio Direction-Finding*, 1st ed. Boston, United States of America: Artech House, Inc., 1991.

[4] Hassan Elkamchouchi and Mohamed Abd Elsalam Mofeed, "Direction-Of-Arrival Methods (DOA) and Time Difference of Arrival (TDOA) Position Location Technique," in *Twenty Second National Radio Science Conference*, Cairo, 2005, pp. B12 1-10.

[5] Mark L Fowler and Xi Hu, "Signal Models for TDOA/FDOA Estimation," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 44, no. 4, pp. 1543-1550, October 2008.

[6] Daniel E Gisselquist, "A Comparison of Stationary and Cyclostationary TDOA Estimators," National Security Agency, Fort Meade, Defense Report.

[7] Darko Musicki and Wolfgang Koch, "Geolocation using TDOA and FDOA Measurements," in *11th International Conference on Information Fusion*, Cologne, 2008.

[8] Mike Grabbe, "IEEE AESS Sept. 28 Topic: Geolocation," in *Unpublished*, Dallas.

[9] Don Koks, "Numerical Calculations for Passive Geolocation Scenarios," Defence Science and Technology Organisation, Edinburgh, Defense Report 2007.

[10] Michael T Grabbe, "Globally-Convergent Geo-Location Algorithm," Software 12/485.409, December 16, 2010.

[11] Derek Elsaesser, "The Discrete Probability Density Method for Emitter Geolocation," Department of National Defence Canada, Ottawa, Defense Report 2006.

[12] Dimitris G Manolakis, Vinay K Ingle, and Stephen M Kogon, *Statistical and Adaptive Signal Processing: Spectral Estimation, Signal Modeling, Adaptive Filtering, and Array Processing*, 1st ed., Stephen W Director, Ed. Boston, United States of America: McGraw-Hill Higher Education, 2000.

[13] Roger du Plessis, *Poor Man's Explanation of Kalman Filtering or How I Stopped Worrying and Learned to Love Matrix Inversion*, 1st ed. Anaheim, United States of America: North American Rockwell Electronics Group, 1967.

[14] Michael T Grabbe, "Method and Apparatus for Signal Tracking Utilizing Universal Algorithm," Software 11/316,298, July 12, 2007.

[15] Derek Elsaesser, "Emitter Geolocation using Low-Accuracy Direction-Finding Sensors," in *Proceedings of the 2009 IEEE Symposium on Computational Intelligence in Security and Defense Applications*, 2009.

[16] Mariette Conning and Ferdie Potgieter, "Analysis of Measured Radar Data for Specific Emitter Identification," Defence, Peace, Safety, and Security Council for Scientific and Industrial Research, Pretoria, Defense Report 2010.

[17] Lal C Godara, "Limitations and Capabilities of Directions-of-Arrival Estimation Techniques wing an Array of Antennas: A Mobile Communications Perspective," The University of New South Wales, Canberra, Defense Report 1996.

[18] Alon Amar and Anthony J Weiss, "Fundamental Limitations on the Number of Resolvable Emitters Using a Geolocation System," *IEEE Transactions on Signal Processing*, vol. 55, no. 5, pp. 2193-2202, May 2007.