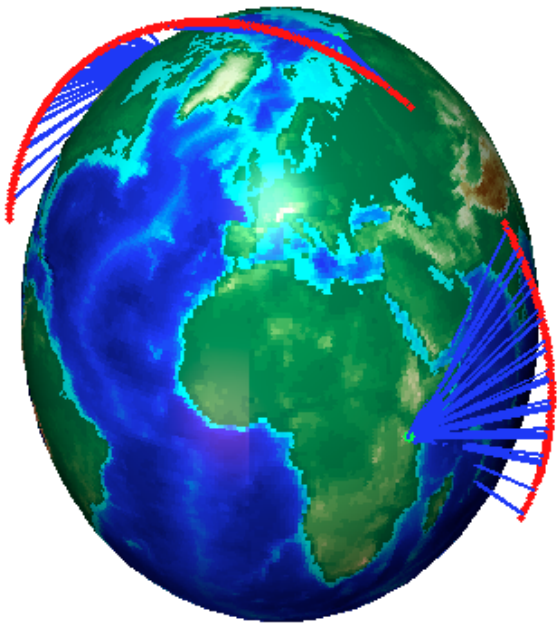
ASEN 5070-Stastistical Orbit Determination-Final Project Report

Zach Dischner
University of Colorado at Boulder
Department of Aerospace Engineering

12-20-2012



1 Introduction

This report summarizes an investigation of various methods of statistical orbit determination, as outlined in ASEN 5070. All programming was performed in Matlab, using a combination of built in functions, self-defined functions, and ones created in collaboration with others. I will examine the results and implications of various filter methodologies including:

- Batch Processor
- Conventional Kalman (Sequential) Filter
- Extended Kalman Filter
- State Noise Compensation
- Alternative Methods for Determining P, the Covariance Matrix

Henceforth, any mention of *textbook, 5070 textbook* or simply, *the book* will refer to the text written by Byron D. Tapley, Bob E. Schutz, and George H Born, titled "Statistical Orbit Determination" [4]. All equations, methodologies, and definitions were obtained from this text[1].

[1] Unless otherwise mentioned, all equations are from this text and will not be sited individually

2 Nomenclature

Term	Definition
CKF	Conventional Kalman Filter
EKF	Extended Kalman Filter
RMS	RMS of Residual Calculations
ρ	Range
$\dot{\rho}$	Range Rate
OD	Orbit Determination
X	True State Vector
X^*	Reference State Vector
C_D	Coefficient of atmospheric drag
J_2	Gravity constant for Earth Oblateness
R_E	Earth Radius
$X_s Y_s Z_s$	ECEF station coordinates
ECEF	Earth Fixed Coordinate Frame
ECI	Earth Inertial Coordinate Frame
$\dot{\theta}$	Earth Rotational Rate
μ	Earth Gravitational Parameter
SNC	State Noise Compensation
DMC	Dynamical Model Compensation
θ	Angle between ECI and ECEF
ρ_A	Atmospheric Drag
A	Cross sectional Area of Satellite

Contents

List of Figures

List of Tables

3 Background

3.1 Orbit Determination Process

3.1.1 Overview of Orbit Determination Process

The orbit determination process is, at the fundamental level, one which determines a celestial body's motion relative to another. Typically, this process is used to determine the motion of Earth-launched satellites relative to Earth. Though the problem can and is often applied to a variety of systems, but this paper will concern itself with Earth-centered satellites and their dynamical states. Even so, the same process described here can easily be applied to any dynamical system. From tracking trains and automobiles, to basic control systems for pointing mechanical linkages, the theory and procedures are generally the same.

The state of a satellite is "a set of parameters required to predict future motion of the system"[4]. These parameters include the position and velocity vectors of the satellite, and often includes other information relating to the dynamical model. Other information can include atmospheric drag, solar wind, gravity terms, tracking station information, or other system dynamics. Fundamentally, anything can be included in the state that the operator would wish to track and model.

The process of determining a satellite state at a given Epoch involves convolving information about its present and past state, in a mathematically optimized manner. Present state information comes from both physical observations of the system, as well as from a dynamical system model. Observations often comes from range, range-rate, azimuth, elevation, angel, and other physically observable quantities, provided by tracking ground station or other celestial bodies. The dynamical model is a purely mathematical approximation of the satellite's state in time. Information about the satellite's past state (*a-priori*) information) comes from the navigator's historical data.

Important to note is the fact that all information about the state is imperfect. Observations have biases and accuracy in measurements, any model will have errors and unknown factors at play, and *a-priori* information is a result of similar imperfections obtained from past information. Basically, the true state of the satellite is never known. The OD process is one which uses all of the imperfect information to generate a statistical "best" estimate of the satellite state at a given epoch.

Another key note is that the relation between observation of the state and the state itself are highly nonlinear in most applications.

3.1.2 Linearization

One of the most crucial elements of the OD process is the ability to model it in a linear manner, even when both the dynamics and observation relationships are highly nonlinear. Typical OD problems can be formulated in the following manner [4].

$$\dot{X} = F(X,t), \quad X(t_k) \equiv X_k \tag{3.1}$$

$$Y_i = G(X_i, t_i) + \epsilon_i; \quad i = 1, 2, ...\ell \tag{3.2}$$

X_k is the n-dimensional state vector we wish to track at time t_k, and Y_i the observation set used to obtain the best estimate of the state at a given time. In this problem, the

system is comprised of indirect observations of a satellite state, with inherent observation errors and biases, and a nonlinear mapping between the state and observation vector. The problem of using said nonlinear observation maps to determine the state with nonlinear dynamics at a given time is very difficult one.

The key in this OD process comes at this point. If the state X, and the reference state X^* (obtained from numerical integration) can be assumed to be reasonably close during the time under examination, the deviation between the two states can be assumed linear. This involves setting a Taylor Series expansion of the true state about the reference state, and neglecting higher order terms. Upon doing so, the full nonlinear OD problem is simplified into solving for the *state deviation vector*, x, which involves solving a simpler set of time-dependent ODEs. In this, the problem of estimating an orbit's trajectory and properties at a point in time becomes one in which we find the deviation of the observed trajectory from a reference one. For a fuller explanation of this procedure, reference the textbook [4, section 4.2].

As a result of the linearization, the OD problem described in (3.1) and (3.2) becomes a simpler one .

$$\dot{x}(t) = A(t)x(t) \tag{3.3}$$

$$y_i = \tilde{H}_i x_i + \epsilon_i; \quad i = 1, 2, ...\ell \tag{3.4}$$

With:

$$A(t) = \left[\frac{\partial F(t}{\partial X(t)}\right] \tag{3.5}$$

$$\tilde{H}_i = \left[\frac{\partial G}{\partial X}\right] \tag{3.6}$$

The A and $\tilde{H}$ matrices are critical in the solving of the OD problem. They are a result of the linearization process. This paper will discuss how to find these specifically in section 4.3.

In general, it can be seen that A is found by taking the partial derivative $\frac{\partial F}{\partial X}$. Which is the partial of the system's dynamical model with respect to the system state. In essence, this describes how the system's dynamics change as the state changes itself. The fact that this is a linear relation for most well formulated problems is what allows the OD problem to be solved in the way we do. In the same way, $\tilde{H}$ is found by taking the partial derivative $\frac{\partial G}{\partial X}$. G is the relationship between the satellite state and its observations. This partial represents how the state-observation relation changes with changes in the state. Again, this partial assumes the deviations are linear over the time in question. This holds true for well formulated problems.

3.1.3 State Transition Matrix

Equation (3.3) is a linear system of equations with time-dependent terms. A general solution to this system is presented as [4]:

$$x(t) = \Phi(t, t_k)x_k \tag{3.7}$$

This equation presents Φ, the *state transition matrix*. This matrix can be used to map the state or state deviation vector forwards and backwards in time. In this general case,

it maps x at t_k to x at some arbitrary time t. This matrix has several useful and unique properties. All of which are outlined thoroughly in the *5070 textbook*. In summary, these properties lead to the formulation of:

$$\dot{x}(t) = \dot{\Phi}(t, t_k)x_k \tag{3.8}$$

Equation (3.8) is of the same form as 3.3. That is, it describes a linear set of differentiable equations. In practice, this leads the operator to include a re-formed Φ with the state vector during numerical integration, giving a numerical solution for the state transition matrix at each state reference time. In practice, a numerical solution for Φ is all that will be obtained, as an analytical one only arises from conditions such as linearity rarely seen in real operations.

While seeming unnecessary immediately, the state transition matrix is powerful in that it can project the state or state deviation forwards or backwards in time. Often times, an OD filter must be iterated; a process in which observations and state information must be related to some epoch time. The state transition matrix is one of the simplest ways to perform this mapping.

For example, to map all observations back to a reference epoch at t_k, equation (3.4) may be formulated as:

$$\begin{aligned} y_1 &= \tilde{H}_1\Phi(t_1, t_k)x_k + \epsilon_1 \\ y_2 &= \tilde{H}_2\Phi(t_2, t_k)x_k + \epsilon_2 \\ &\vdots \\ y_\ell &= \tilde{H}_\ell\Phi(t_\ell, t_k)x_\ell + \epsilon_\ell \end{aligned} \tag{3.9}$$

In this, ℓ separate observation deviation sets are all mapped to the epoch state deviation vectors at x_k. This is desirable for iterative filtering, as well as reducing the number of equations to be solved in the system.

3.2 OD Solutions

Now, with the formulation of a workable OD problem in section 3.1, the statistical 'best' solution must be determined. Conceptually, there are an infinite number of measurable metrics, or *performance indices*, which can be design against. This is one of the most important steps in solving the OD problem. What is the best way to combine an erroneous reference trajectory and noisy/biased observations to obtain an estimate of our state that is closest to the truth trajectory? This simple question is what drives the design of OD filters and determination algorithms. Several of the basic realizations of the answer to this question will be provided here, though there are countless more.

3.2.1 Least Squares Solution

One of the most common approaches is to attempt to minimize the sum of the squares of the observation residuals. The performance index for this approach is [4]:

$$J(x) = \frac{1}{2}\epsilon^T\epsilon \tag{3.10}$$

Choosing x to minimize eq(3.10) is a natural choice. By design, this procedure is robust to the sign of the residual. If large residuals exist, but have opposite sign, the minimization attempt could yield calculated observation errors of zero. This method is also sensitive to outliers.

A solution for x must be found that minimizes the least squares index. From eq(3.4), ϵ may be solved for, and substituted into eq(3.10).

$$J(x) = \frac{1}{2}(y - Hx)^T(y - Hx) \tag{3.11}$$

Minimizing J is done by finding a zero in its partial derivative, where the second derivative is positive.

$$\frac{\partial J}{\partial x} = 0 \;\; Where \;\; \delta x^T \frac{\partial J^2}{\partial x^2} \delta x > 0$$

After some algebra, a final formulation for x that minimizes the least squares criterion is

$$\hat{x} = (H^T H)^{-1} H^T y \tag{3.12}$$

By solving for the state deviation vector in this manner, the sum of the square of the observation residuals is minimized. A full derivation of this solution can be found in the *5070 textbook* [4]. While this is a sufficient measure for some applications, more powerful realizations exist which can help obtain a more accurate prediction of the satellite.

3.2.2 Weighted Least Squares

The standard least squares formulation can be modified by adding weighting to each observation vector. In reality, not all observations have equal precision or accuracy, and the circumstances surrounding the observation may cause certainty to change. For example, a laser range finder will obtain more precise results as the satellite flies directly overhead, as opposed to near the horizon, where atmospheric turbulence will distort measurements. With the inclusion of a weighting matrix, eq(3.10) becomes

$$J(x) = \frac{1}{2}\epsilon^T W \epsilon \tag{3.13}$$

W is a diagonal weighting matrix, with size $\ell x \ell$. Each diagonal element corresponds to a weighting value to be applied to observation vector. After more algebra, the solution found in eq(3.12) with weighting included becomes

$$\hat{x} = (H^T W H)^{-1} H^T W y \tag{3.14}$$

The implications of this are the same. By determining $\hat{x}$ in this manner, we wish to minimize a performance criteria which now includes weighting factors.

3.2.3 Weighted Least Squares with *a-priori*

In the same vein as adding a weighting to the least squares solution, an even better state deviation estimate can be found by including information about the state's history, known as *a-priori* information. This information comes in the form of $\bar{x}$, the last known state deviation vector, and $\bar{W}$, the associated previous weighting matrix associated with

that state. When these terms are added into the weighted least squares solution, eq(3.14) becomes

$$\hat{x} = (H^T W H + \bar{W}_k)^{-1}(H^T W y + \bar{W}_k \bar{x}_k) \tag{3.15}$$

This solution now takes advantage of the most commonly available information about a given state/state deviation vector. With it, comes the most accurate estimate for $\hat{x}$ that has been discussed yet.

3.2.4 Minimum Variance Estimate

All above formulations of the least squares solutions makes intuitive sense when considering the physical system, however, they lack in the fact that they do not account for statistical characteristics in observation errors and *a-priori* information. One other common estimator is the *Minimum Variance* estimator. This method tries to utilize any statistical information about observations in time to generate the optimal estimate of $\hat{x}$. The derivation of this method won't be explored, but it is in the course textbook.

$$\hat{x} = P_k H^T R^{-1} y \tag{3.16}$$

Where

$$P = E[(\hat{x}_k - x_k)(\hat{x}_k - x_k)^T] \qquad R = E[\epsilon\epsilon^T] = W^{-1}$$

P is the system's covariance matrix. It holds the variances (certainty) of each state element on its diagonal, and the of diagonal elements contain the linear correlation factors between state elements. It is a symmetric, positive definite matrix by definition (when properly conditioned), and is a crucial statistical measure of a filter's certainty in its solution at a given time. R is a matrix which holds information about the observation errors, and their relation to each-other.

Again, adding *a-priori* information to eq(3.16) can help to yield a more robust estimation.

$$\hat{x} = (\tilde{H}_k R^{-1} \tilde{H}_k + \bar{P}_k^{-1})^{-1}(\tilde{H}_k R^{-1} y_k + \bar{P}_k^{-1} \bar{x}_k) \tag{3.17}$$

Theoretically, there are an infinite number of *best* estimate solutions. From a 'least cubed' or 'least 4/3' estimates, to an often used *Maximum Likelihood and Bayesian Estimation* solution, there are dozens of implemented and derived OD solutions. The general process is to first pick a performance index, $J(x)$. This should be a function which has some basis in statistical reasoning. Next, solve for a value of x which minimizes the performance index.

3.3 Orbit Determination Filters

With assumptions about the linearity of the deviation between satellite truth and reference states, and statistically derived criteria for solving for this state deviation, the process of orbit determination can begin. There are several standard approaches to solving the OD problem. This paper will only concern itself with the basic ones.

3.3.1 The Batch Processor

The Batch Processor is one formulation of the OD process. As the name implies, its aim is to process a set of observations and a reference trajectory at one time, in a single batch computation. Typically, the entire batch of computation is done in order to get a new best estimate for the state deviation vector at an epoch t_0. In operation, the satellite's current trajectory is not the only state of interest. For science missions, a very accurate estimation of the state at a given historical instance is needed for high quality analysis of data. The batch processor takes information about a time before the epoch of interest, and information afterwards to generate the best estimate of the satellite state at epoch. Using the weighted least squares with *a-priori* information, the eq(3.15) equation for the state deviation vector for a full batch of observations becomes

$$\hat{x}_0 = (H^T R^{-1} H + \bar{P}_0^{-1})^{-1}(H^T R^{-1} y + \bar{P}_0^{-1}\bar{x}_0) \tag{3.18}$$

Here, t_0 is the epoch time of interest. Since eq(3.18) is supposed to solve for $\hat{x}_0$, all quantities must be mapped to that epoch using the state transition matrices, described in section 3.1.3.

The course text book features an in-depth procedure for using the batch processor in an application, as well as insights into key terms and quantities in Chapter 4.7. It will not be discussed here for the sake of brevity. The general process is to [2]:

- Collect Observations and Residuals Over an Arc
- Process All Using Weighted Least Squares with *a-priori*
- Generate Best Estimate
- Iterate

The batch processor will yield an updated result for the state deviation vector at epoch, however it must be iterated in order to obtain the absolute best solution. The iteration process is straightforward, and is done by re-running the filter after updating the initial conditions with the new best estimates. Again, this process is outlined in the textbook.

One problem with the batch processor is the fact that a large matrix inversion is required. This is evident in eq(3.18), and this portion is called the *information matrix*, eq(3.19). In practice, Cholesky decompositions or other methods would be utilized to increase the accuracy of this inversion, but matrix inversions are still computationally heavy.

$$\Lambda_0 = (H^T R^{-1} H + \bar{P}_0^{-1}) \tag{3.19}$$

The other potential detriment to the batch processor is the fact that since it uses future information to get the best estimate of some past epoch, it is not ideally suited for live tracking situations. This is where the sequential processor comes in.

3.3.2 Sequential Filter

The conventional sequential filter, commonly referred to as a Kalman filter, KF, or CKF, is a reformulation of the batch processor described in 5.1. With perfect numerical precision, the two yield identical results.

The advantage of the CKF is that it processes observations live. As soon as a new observation is available, it will be processed. The other advantage is that not only can it process observation sets one at a time, but it can process just single observations sequentially. Even when a set is observed at the same time. This is advantageous because it cuts down on computational requirements by just involving scalar divisions [4]. The reformulation of the batch solution is as follows

$$\hat{x}_k = \bar{x}_k + K_k[y_k - \tilde{H}_k \bar{x}k] \tag{3.20}$$

Where the covariance matrix, P at new time t_k is

$$P_k = [I - K_k \tilde{H}_k]\bar{P}_k \tag{3.21}$$

And K is found at each observation time as

$$K_k \equiv \bar{P}_k \tilde{H}_k^T [\tilde{H}_k \bar{P}_k \tilde{H}_k^T + R_k]^{-1} \tag{3.22}$$

K is referred to as the Kalman Gain. It is a matrix which weighs the certainty of the observations against the certainty of the reference trajectory. Equation (3.20) shows the computation of the best estimate state deviation vector at time t_k. It is a balance between the *a-priori* state deviation vector and the newly observed difference in observed and computed observation deviation vectors. The *balancing* is performed by the Kalman gain. The computation uses weighting, as well as *a-priori* information.

As with the batch processor, the CKF should be iterated to obtain the best solution. Instead of computing the state deviation at the epoch time, here it is computed at the latest update time. To iterate, this solution must be mapped back to the epoch time using the state transition matrix (section 3.1.3).

The full procedural outline is found in Chapter 4.7 of the textbook, but is summarized here [2].

- Initialize
- Start at Reference Epoch
- Time Update

  Integrate Reference Trajectory to Current Time of Interest

  Map State Estimate and Covariance to New time
- Measurement Update

  Process New Measurement, if Available

  Update State Estimate and Covariance With New Information
- Repeat Time Update and Measurement Update
- (Optionally) Iterate Entire Process

Although the Kalman filter is discussed here in the context of orbit determination, it is no leap in reasoning to see how this has implications in other fields. Control systems, in particular, can find the sequential processor apt for their application. They must weigh some predicted behavior against a measured behavior in a precise manner, and

act accordingly. Many other applications for the Kalman filter exist, which shows how powerful this approach is.

Though powerful, the sequential processor is not without its limitations. One of which is a breakdown due to numerical precision errors. This is particularly present in the calculation of the covariance matrix in the measurement update phase. Remedies do exist, and they will be discussed in section 3.4.

Another issue with the batch and sequential processors is that, in time, the covariance matrix trends towards zero. If this happens, the Kalman gain causes the filter to become insensitive to observations. This is called *Filter Saturation*. When this happens, the filter ignores new observations, even if they point strongly towards a different trajectory than the reference one. If this happens, and the true trajectory and the reference trajectory diverge beyond the regime in which their differences can be approximated as linear, the filter will break down. This can be remedied by adding state noise compensation (section 3.4.2), among other methods.

A final issue with the sequential and batch filters is that, over a long enough observation streak, assumptions of linearity tend to break down. If, at any point, the differences between reference and truth trajectories are not within the linear regime, the process will diverge. The answer to this problem often lies in implementing an Extended Kalman Filter.

3.3.3 Extended Kalman Filter

The Extended Kalman Filter (EKF) is sometimes used to overcome errors in the assumptions of linearization. It is based on the Conventional Kalman Filter. The difference is that the reference trajectory is updated after each observation has been processed using the new best estimate for the state deviation vector. In practice, this means that, before the time update phase of the CKF, the initial conditions of the dynamical integration are updated with the latest information from previous integrations and observations.

$$(X_k^*)_{new} = X_k^* + \hat{x}_k \tag{3.23}$$

This means that after each observation, $\hat{x}_k$ will be zero, and needs to be recalculated independently.

$$\hat{x}_{K+1} = K_{k+1} y_{k+1} \tag{3.24}$$

These two changes in methodology yields a completely differently behaving filter. By updating the reference trajectory at every observation time, the EKF is less likely to diverge after long duration runs. Over time, errors in the operator's model will cause the reference trajectory to diverge significantly enough from the truth to invalidate linearity assumptions if errors are not corrected for. A full procedure for the EKF is discussed in the textbook, Chapter 4.7.3.

The EKF does have other issues and potential pitfalls which must be addressed in implementation, which will be overviewed section 5.3.

3.4 Numerical Considerations

The difference in theory and practice is sometimes enough to cripple the best formulated OD filters. There are several additive procedures that can aid filters.

3.4.1 Covariance Matrix

An often occurring problem is the breakdown of the covariance matrix. Equation (3.21) experiences these breakdowns in practice. Numerical precision errors can lead the covariance matrix to lose its positive-definiteness and symmetry. Several alternate formulations for the covariance calculation have been created.

One of the simplest examples is the Joseph formulation.

$$P_k = (I - K_k\tilde{H}_k)\bar{P}_k(I - K_k\tilde{H}_k)^T + K_k R_k K_k^T \tag{3.25}$$

This method will always yield a symmetric result due to its formulation, however, it can still lose positive definiteness. It has the advantage of being simple to implement.

There are several more complex algorithms for calculating the covariance matrix. These include the Potter formulation, and square root free Givens and Householder transformations. The text has derivations and procedures for these other methods, and they wont be discussed here for the sake of brevity.

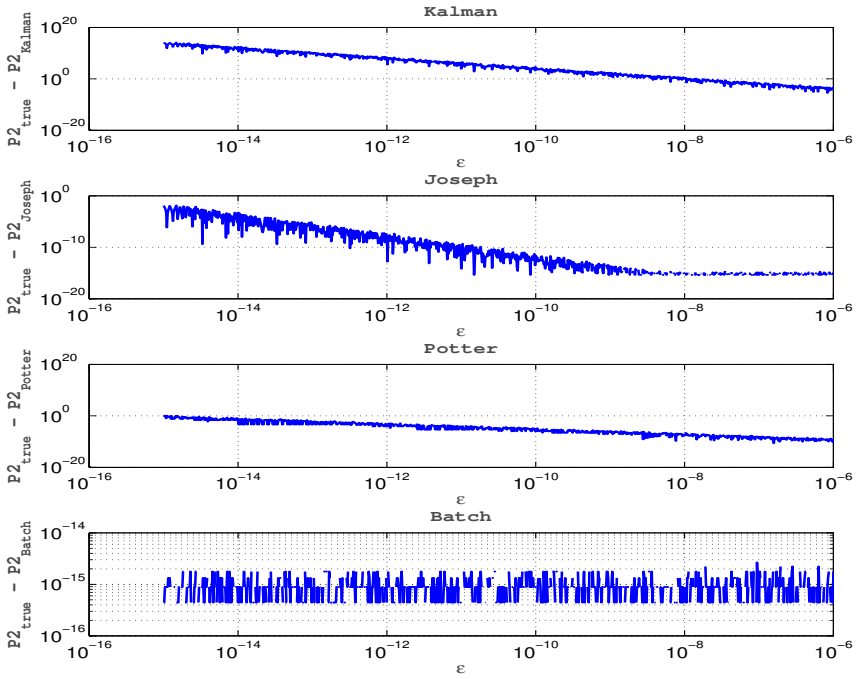


Figure 1: Comparison of Covariance Matrix Formulations

Figure 1 shows a quick examination of different methods for calculating a covariance. The textbook offers a poorly conditioned system, one which standard covariance calculations break down when small error terms are introduced. The system was originally proposed by Bierman, in 1977, and the exercise is outlined in the textbook in Chapter 4.7. It shows that the difference between true and calculated values for different methods is significant for the typical Kalman formulation. Joseph and Potter formulations improve greatly for small error values.

3.4.2 State Noise Compensation

Another common problem typically faced in OD filter implementation, mentioned in section 3.3.2 is due to filter saturation. After time, the covariance matrix along ann observation streak will approach zero. As $P \to 0$, eq(3.21) yields a Kalman gain that causes the filter to be insensitive to observations. The filter will cause the best estimated trajectory to stay close to the reference trajectory. If this happens, and the reference trajectory is too far from the true trajectory, linearization assumptions are no longer valid and the filter will break down. Filter divergence and filter saturation are big enough considerations as to sometimes cause operators to avoid the sequential algorithm for real-time tracking scenarios.

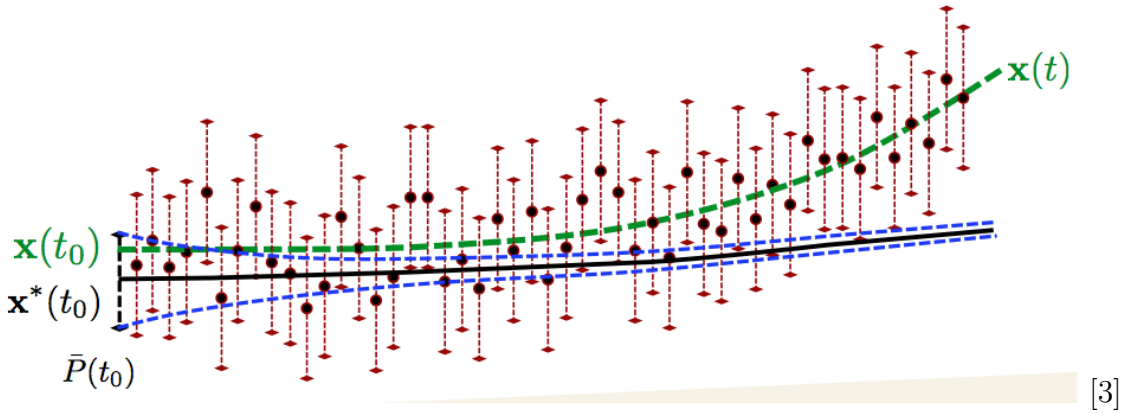


Figure 2: Filter Saturation

Figure 2 depicts a simple 2-dimensional scenario in which the true deviation trajectory (green) begins to diverge from the reference one (black). By the time this happens, the covariance matrix (blue) has become too small, and the filter no longer adjusts its reference trajectory to the truth. Observation covariances are much larger than the state deviation covariance, and are more or less ignored.

One common method of avoiding filter saturation is to artificially inflate the covariance matrix throughout the orbit. By assuming the errors in the linearized dynamics are due to process noise, the state dynamics of the system including this process noise is

$$\dot{x}(t) = A(t)x(t) + B(t)u(t) \tag{3.26}$$

Where A and B are known functions of time, and $u(t)$ can take functional form to mimc any number of processes. The simplest form of $u(t)$ is to assume that it is just white noise process.

The textbook, Chapter 4.9, outlines how this addition into the dynamical model results in a new calculation for the covariance matrix at time t_k.

$$\bar{P}_{k+1} = \Phi(t_{k+1}, t_k)P_k\Phi^T(t_{k+1}, t_k) + \Gamma(t_{k+1}, t_k)Q_k\Gamma^T(t_{k+1}, t_k) \tag{3.27}$$

Γ and Q (the process noise covariance matrix) are matrices calculated specifically for a given system at a given time, and contain tuning parameters. These parameters are often found through Monte Carlo simulations, and are not constant for a given system. This is a relatively simple system, in which the covariance matrix just gets an extra additive term at each step to keep it from approaching zero.

Besides white noise processes, $u(t)$ may take the form of constant noise, piecewise noise, and correlated noise functions. It can also be assumed to be a function, and not simply noise processes. Assuming 1st order stochastic, the SNC becomes a dynamical model compensator (DMC). This formulation not only helps the filter avoid saturation, but it also backs out the unmodeled dynamics that the system observations imply. This can be an enormously helpful feature. It can help the operator adjust their model, discover new dynamical modes that they didn't know exist, or simply be used additively to the calculation of the reference trajectory for better accuracy. The derivations for these modes are beyond the scope of this report, but they are proven methods. Chapter 4.9 in the text outlines some methods in detail, and more literature exists on the others.

4 Project Problem

With all the tools developed in 3, now its time to apply them towards an actual OD problem.

4.1 Problem Description and Setup

The goal of this project is to develop the theory and algorithms to track the trajectory of a satellite in orbit around Earth. Computational code was developed in Matlab. The simulation assumptions are as follows:

- The satellite was assumed to be approximately spherical, with even mass distribution.
- There are three tracking stations, each providing instantaneous range and range-rate data when the satellite is in view.
- The dynamical model is to include atmospheric drag and J2 gravity perturbations.

Data from each tracking station is available, and must be used in tandem with a set of system dynamics to generate best estimates of the actual satellite trajectory.

4.2 Equations of Motion

The equations of motion for this satellite were calculated, assuming only a point mass gravity model with J2 perturbations and atmospheric drag effects. The calculations were performed symbolically with Matlab, using definitions found on the course website [1].

The acceleration of a body under point mass with J2 perturbation follows the general form of an acceleration: $\ddot{\vec{r}} = \bigtriangledown U$, where U is formulated by:

$$U_{PtJ2} = U_{pt} + U_{J_2} = \frac{\mu}{r}\left[1 - J_2\left(\frac{R_E}{r}\right)^2\left(\frac{3}{2}\sin^2\phi - \frac{1}{2}\right)\right] \tag{4.1}$$

$$\ddot{\vec{r}}_{PtJ2} = \bigtriangledown U_{PtJ2} \tag{4.2}$$

With

$$J_2 = 1.082626925638815 \times 10^{-3}$$

$$\mu = 3.986004415 \times 10^{14} km^3/s^2$$

$$R_E = 6378136.3m$$
$$\phi = Latitude \rightarrow z/r$$
$$r = \sqrt{x^2 + y^2 + z^2}$$

$\bigtriangledown U$ was computed symbolically in Matlab.The result will not be shown here as it has been shown in previous assignments, and is too cumbersome to provide extra insight here. The theory above will suffice

Atmospheric drag was provided in the form of an acceleration, so implementing it was simple. It just needed to be added in to the acceleration found in eq(4.1). The acceleration the satellite experiences due to atmospheric drag is given by

$$\ddot{\vec{r}}_{drag} = -\frac{1}{2} C_D \left(\frac{A}{m}\right) \rho_A V_A \vec{V}_A \tag{4.3}$$

With constants/definitions given by

$$\begin{aligned}
C_d &= 2.0 \\
A &= 3.0m^2 \\
m &= 970kg \\
\rho_0 &= 3.614 \times 10^{-13} kg/m^3 \\
r_0 &= 700000 + R_E \\
H &= 88667m \\
\dot{\theta} &= 7.29211585530066 \times 10^{-5} rad/s \\
\rho_A &= \rho_0 e^{\frac{-(r-r_0)}{H}} \\
\vec{V}_A &= \begin{bmatrix} \dot{x} + \dot{\theta} y \\ \dot{y} - \dot{\theta} x \\ \dot{z} \end{bmatrix} \\
V_A &= \sqrt{(\dot{x} + \dot{\theta} y)^2 + (doty - \dot{\theta} x)^2 + \dot{z}^2}
\end{aligned}$$

So the equations of motion of the satellite are described for this system as a simple additive combination of eq(4.2) and eq(4.3). Matlab code for the symbolic calculations discussed here are included in the Appendix.

$$\ddot{\vec{r}} = \ddot{\vec{r}}_{PtJ2} + \ddot{\vec{r}}_{drag} \tag{4.4}$$

4.3 A and $\tilde{H}$ Matrices

With the equations of motion of the satellite derived, the next step is to solve for the A and $\tilde{H}$ matrices. Recall these are the two linearized matrices which allow for the nonlinear orbit dynamics and observation-state relation to be modeled in as a linear state-deviation system. See section 3.1.2 and equation (3.3) and (3.4).

The state for this OD problem is defined as an 18x1 vector. $\bar{X}_{site_i}$ represents the $[x, y, z]_i$ position vectors of each station, in body coordinates. With three stations, that yields a 9x1 vector of station coordinates.

$$X = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \mu \\ J2 \\ CD \\ X_{s1} \\ Y_{s1} \\ Z_{s1} \\ X_{s2} \\ Y_{s2} \\ Z_{s2} \\ X_{s3} \\ Y_{s3} \\ Z_{s3} \end{bmatrix} \tag{4.5}$$

4.3.1 *A* Matrix Derivation

Deriving the A matrix is not a trivial matter, and hence was performed symbolically in Matlab. From eq(3.5) recall that finding A involves the partial derivative:

$$A = \left[\frac{\partial F(t)}{\partial X(t)}\right]$$

Where X is the state vector seen in eq(4.5), and F is the function that represents the derivative of the state elements.

$$\dot{X} = \begin{bmatrix} \dot{\bar{r}} \\ \ddot{\bar{r}} \\ \dot{\mu_E} \\ \dot{J}_2 \\ \dot{C}_d \\ \dot{\bar{X}}_{site_i} \end{bmatrix} = F(X,t) = \begin{bmatrix} \dot{\bar{r}} \\ \ddot{\bar{r}} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \frac{-\mu_E x}{r^3}\left[1 - \frac{3}{2}J_2\left(5\left(\frac{z}{r}\right)^2 - 1\right)\right] - \frac{1}{2}C_D\frac{A}{m}\rho_A V_A\left(\dot{x} + \dot{\theta}y\right) \\ \frac{-\mu_E y}{r^3}\left[1 - \frac{3}{2}J_2\left(5\left(\frac{z}{r}\right)^2 - 1\right)\right] - \frac{1}{2}C_D\frac{A}{m}\rho_A V_A\left(\dot{y} + \dot{\theta}x\right) \\ \frac{-\mu_E z}{r^3}\left[1 - \frac{3}{2}J_2\left(5\left(\frac{z}{r}\right)^2 - 3\right)\right] - \frac{1}{2}C_D\frac{A}{m}\rho_A V_A\dot{z} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.6}$$

$\ddot{\bar{r}}$ Is the acceleration of the satellite from a 2-body J_2 acceleration and atmospheric drag, found in eq(4.4). I used Matlab's symbolic toolbox to construct these vectors. The full code set is found in the Appendix, but summarized below.

```
syms x y z xdot ydot zdot uE J2 Cd Xsite1  Ysite1  Zsite1 Xsite2 Ysite2 Zsite2
...
        Xsite3 Ysite3 Zsite3 theta theta_dot
syms R_E r Area m rho_a Va va

X       = [x ; y ; z ; xdot ; ydot ; zdot ; uE ; J2 ; Cd ; Xsite1 ; Ysite1 ;...
        Zsite1; Xsite2; Ysite2; Zsite2 ; Xsite3; Ysite3; Zsite3];

F_a = F_J2 - F_Drag; % Calculated previously, symbolically

% X' = F*X
F =[xdot ; ydot ; zdot ; F_a ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0];
```

Then, to solve for A in eq(3.5), I used Matlab's built in *jacobian* function to take the partial derivative of each element of F with respect to each element of X.

```
A = jacobian(F,X);
```

The resulting symbolic expression for A is too large to provide in any meaningful insight here. A few terms are listed, to show proof of validity. We know $\partial x \partial x$ is always supposed to be 1. This is exactly what is found by examining the A matrix.

$$A_{1,4} = \frac{\partial \dot{x}}{\partial \dot{x}} = 1 \qquad A_{2,5} = \frac{\partial \dot{y}}{\partial \dot{y}} = 1 \qquad A_{3,6} = \frac{\partial \dot{z}}{\partial \dot{z}} = 1$$

The above three elements match what we expect. Both the form matches that specified in the textbook, and the values match intuition. A second check was performed too. I obtained a numerical solution to A after the first step in integration, and compared those values to the solution provided in the course website. Below is the difference in my computed state matrix and the reference state matrix, for only the upper 9x9 portion of the matrix, as it is the only region of interest here.

The maximum difference is provided below. That difference represents the maximum deviation in calculations between my computed A matrix and the reference A_{true} matrix provided by the solutions. The result is sufficiently small to consider correct. In addition, comparing several of the symbolic expressions I obtained and those provided by website solutions yielded the same thing. Pulling out Va from the solutions provided, I get a match.

$$A_{4,6} = \left[\frac{\partial \ddot{x}}{\partial \dot{z}}\right] = a * Cd * rho_0 \dot{z} * e^{\frac{r-r_0}{H}} \frac{(\dot{x} + \dot{\theta} * y))}{2m((\dot{x} + \dot{\theta} y)^2 + (\dot{y} - \dot{\theta} x)^2 + \dot{z}^2)} \tag{4.7}$$

$$\max(A_{computed} - A_{true}) = 6.10406418193431 \times 10^{-8} \tag{4.8}$$

4.3.2 $\tilde{H}$ Matrix Derivation

The procedure for finding $\tilde{H}$ is nearly identical to that in 4.3.1. Recall from eq(3.6)

$$\tilde{H}_i = \left[\frac{\partial G}{\partial X}\right]$$

Where

$$Y = G(X,t) = \begin{bmatrix} \rho \\ \dot{\rho} \end{bmatrix}$$

First, the station ECEF coordinates must be rotated into the ECI frame. This was done with a simple DCM about the z axis.

$$\begin{bmatrix} X_s i \\ Y_s i \\ Z_s i \end{bmatrix}_{ECI} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} X_s i \\ Y_s i \\ Z_s i \end{bmatrix}_{ECEF} = \begin{bmatrix} X_s\cos\theta - Y_s\sin\theta \\ X_s\sin\theta + Y_s\cos\theta \\ Z_s \end{bmatrix} \tag{4.9}$$

Where θ is a function of time and is calculated based on the rotation rate of the earth.

$$\theta = \dot{\theta}t = 7.2921158553 \times 10^{-5}\frac{rad}{s}t$$

Now, equations for ρ and $\dot{\rho}$ can be found:

$$\rho = \sqrt{(x - X_s\cos\theta + Y_s\sin\theta)^2 + (y - X_s\sin\theta - Y_s\cos\theta)^2 + (z - Z_s)^2} \tag{4.10}$$

$$\dot{\rho} = \frac{(x - X_s\cos\theta + Y_s\sin\theta)\left(\dot{x} - X_s\sin\theta\dot{\theta} + Y_s\cos\theta\dot{\theta}\right)}{\rho} + \frac{(y - X_s\sin\theta - Y_s\cos\theta)\left(\dot{y} + X_s\cos\theta\dot{\theta} + Y_S\sin\theta\dot{\theta}\right) + (z - Z_s)\,\dot{z}}{\rho} \tag{4.11}$$

These two combine to give the G matrix. Now, eq(3.6) can be used to find $\tilde{H}$.

Again, I used the Jacobian function in Matlab to take the partial derivatives of G with respect to each element in the state vector.

```
obs = [rho;rhodot]; % rho, rhodot are symbolic expressions already determined.

% For just single station
Single_Station_X       = [x ; y ; z ; xdot ; ydot ; zdot ; uE ; J2 ; Cd ;...
                          Xsite ; Ysite ; Zsite];
Htilde = jacobian(obs,Single_Station_X);
```

Here too, the $\tilde{H}$ matrix symbolic result is too large to be displayed here. Note that this is the formulation for $\tilde{H}$ using a generic station identifier, which yields a $2x12$ matrix. This is because the equations for $\rho, \dot{\rho}$ involve only a generic ground station. When used in practice the $\tilde{H}$ matrix will be calculated, and then a $2x6$ array of zeros will be filled in in accordance with the station from which current observations originate from. Since no observations occur at the same time, this is perfectly acceptable in this situation.

4.4 A $\tilde{H}$ Implementation Considerations

At this point, symbolic expressions for everything needed by a conventional filter exist in Matlab. It would be functionally okay to just evaluate these symbolic expressions throughout the orbit pass, but horribly inefficient. For the speediest implementation possible, I took a few steps, all in Matab:

- Simplified single elements in A, $\tilde{H}$ Matrices

- Simplified entire A, $\tilde{H}$ Matrices
- Convert to function with Matlab's *matlabFunction*.m

These steps are illustrated in the following section of my script. The multiple simplifications are quite possibly redundant, but redundancy at this step really can't hurt anything.

```
%% Find A Matrix
A = jacobian(F,X);

[rows,cols] = size(A);
for ii = 1:length(rows)
    for jj = 1:length(cols)
        A(ii,jj) = simplify(A(ii,jj));
    end
end

A=simplify(A);

matlabFunction(A,'file','A_18x18.m')
```

One of the most crucial steps is using the *matlabFunction*.m function. In order to minimize computation time and effort, it is wise in a large matrix such as this to find common factors across multiple elements that can be calculated once, and subbed in multiple times. In an $18x18$ matrix like here, with complex roots and exponents, this step could save thousands of calculations per call. Typically, finding these common factors would be done by hand, and is a tedious, error prone, and not foolproof exercise. Thankfully, *matlabFunction* was built intelligently, and does this for us. It automatically finds common factors in the symbolic function, and uses those to write a callable m-file which efficiently calculates A from inputs defined by $A's$ separate symbolic variables. This method is iterable, efficient, and fast. And it cuts down on computational effort needed in the OD process, which makes for a faster tracker.

5 Project Results

The OD problem presented in section 4.1 was solved using various filters and numerical considerations. Here I will discuss the results of each, as well as lessons and insights earned from each investigation.

5.1 Batch Processor

The Batch Processor (section 5.1) was the simplest of the OD filters to implement. Previous homework assignments required nearly the fully functional batch processor for the project.

Figure 3 shows range and range rate residuals from the batch processor over three processing iterations. The first iteration is plotted on the top row, second iteration on the middle, ect. The first iteration's residuals have definite shape for range, and still a shape for range rate. By iteration 3, both residual plots look like noise. This is a good indication that the processor has performed its job correctly.

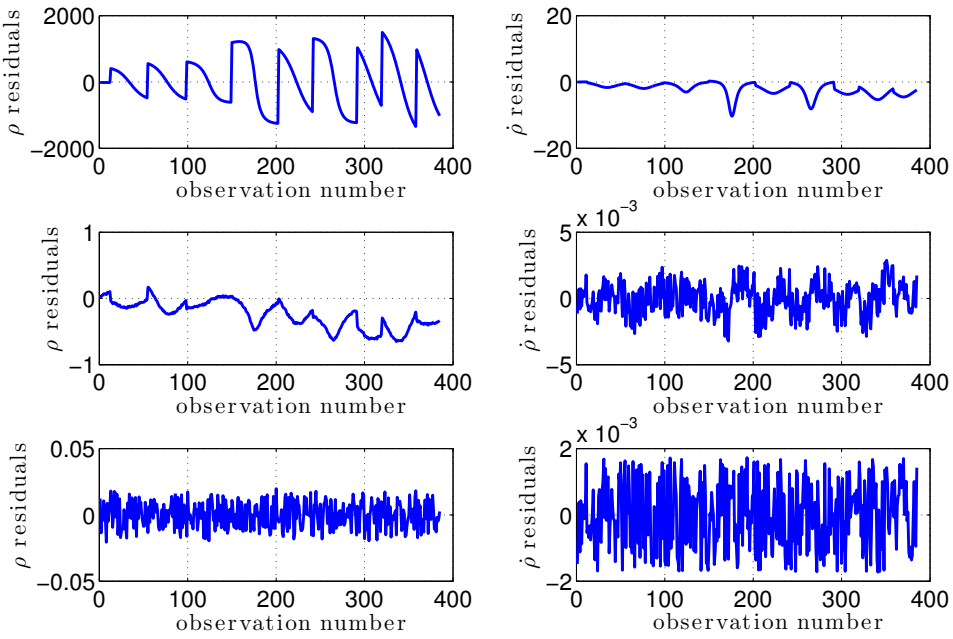


Figure 3: Batch Processor Residuals

iter	1	2	3	
ρ_{res}	732.7483	0.31957	0.0097249	[m]
$\dot{\rho}_{res}$	2.9002	0.0011997	0.00099792	[m/s]

Table 1: Batch Residuals

In the first iteration, residuals were quite large. By the last iteration, however, the range and range rate residuals have shrunk to $0.00972m$, and $0.00099792m/s$, respectively. This is magnitudes better than the first iteration, and shows how well the batch works.

State Parameter		Value at t_0	Δ in State	Variance
x [m]		757700.2904	0.29042	5.6626e-05
y [m]		5222606.5778	-0.42217	0.00013911
z [m]		4851499.7381	-0.26187	0.00022029
$\dot{x}$ [m/s]		2213.2506	0.040618	7.4632e-11
$\dot{y}$ [m/s]		4678.3727	0.032709	2.0861e-10
$\dot{z}$ [m/s]		-5371.3144	-0.014415	1.0476e-10
μ $[m^3/s^2]$		398600398730391.9	-42769608.0625	172813484323.2254
$J2$		0.001082	-6.2748e-07	5.9809e-20
CD	=	2.1887	0.1887	1.4492e-05
X_{s1} [m]		-5127510	-6.5193e-09	1e-10
Y_{s1} [m]		-3794160	-4.6566e-09	1e-10
Z_{s1} [m]		5.8322e-10	5.8322e-10	1e-10
X_{s2} [m]		3860899.9916	-10.0084	2.7786e-05
Y_{s2} [m]		3238500.0035	10.0035	7.1368e-05
Z_{s2} [m]		3898099.9764	5.9764	7.6755e-05
X_{s3} [m]		549499.9913	-5.0087	5.3859e-05
Y_{s3} [m]		-1380869.9787	2.0213	0.00016251
Z_{s3} [m]		6182199.9761	2.9761	0.00027417

Table 2: Batch State at t_0

By the end of the third run, variances on velocity components are on the order of $\times 10^{-10}$, and position components are on the order of $\times 10^{-4}$. Both offer good confidence in the final solution. There is a large variance associated with the μ parameter, but relative to its size, it is also a decent approximation.

Note that the variance on station 1 (X_{s1}, Y_{s1}, Z_{s1}) are all small. This station was 'fixed' in the ECEF at the batch initialization by a small covariance in the *a-priori* values. We have a high confidence in this station's coordinates and must fix it in space. If nothing was fixed to the ECEF, we could run into a problem. The filter would find the trajectory based on equations of motion formulated for ECEF, and compare against observations made by the station. This part would be perfectly acceptable, and the same results would be obtained. But with nothing actually tied to the reference frame, we don't really know how the satellite state relates to the earth surface. By fixing one station, everything is now computed to be relative to ECEF. If no stations were fixed, the filter could allow them all to move by a large amount, in order to minimize $\hat{x}_0$. The stations really aren't the things that are moving, so we want their motion to be relatively constant. It makes no difference which station is fixed, only that one is to make sure the computed trajectory is really relative to ECEF.

5.1.1 Range vs Range Rate

The Batch processor is especially well formulated for experimentation with different observation types used. By changing the diagonal elements of the weighting matrix, W, I was able to see how the batch processor performed when it used just range data, range rate data, and both observation types. This is shown in Table 3 after 3 batch iterations.

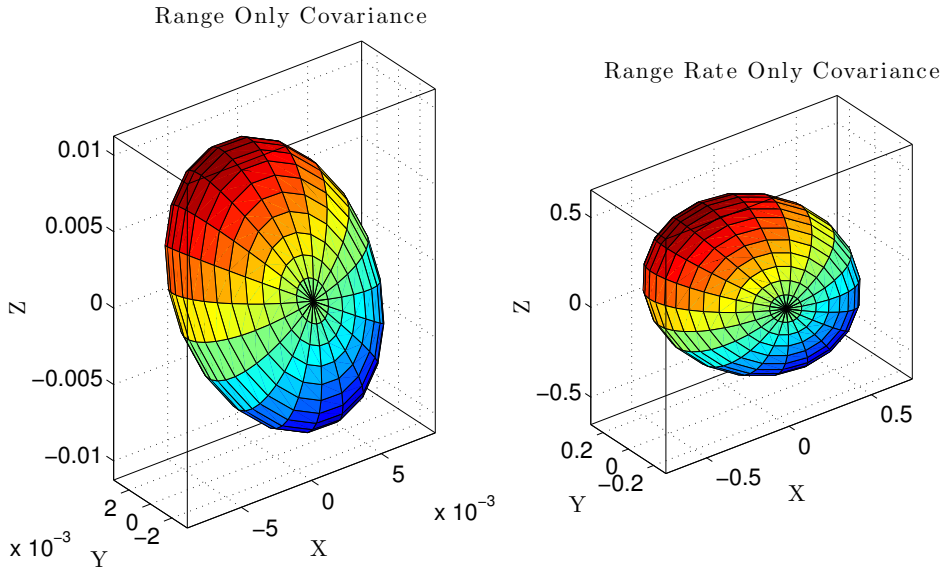


Figure 4: Comparison of Observation Type

Obs Type	**Range**	**Range Rate**	**Range and Range Rate**	
ρ_{res}	0.00972	0.24242	0.0097249	[m]
$\dot{\rho}_r es$	0.00100	0.00098	0.00099792	[m/s]

Table 3: Batch Residuals

Using just range data seemed to have no real effect on the final residuals. However, using just range rate data caused the RMS for range to increase by a few orders of magnitude, while the RMS for range rate decreased very slightly.

A final comparison between the use of each dataset is in a measure of the final covariance matrix. Using the principles from 3.1.3, the final covariance matrix can be found by

$$P_{tf} = \Phi(t_f, t_0) P_{t0} \Phi^T(t_f, t_0) \tag{5.1}$$

Where P_{t0} is given by

$$P_{t0} = \Lambda^{-1}$$

Ellipses in Figure 4 are of the position portion of the final covariance matrix. Both are most concentric in the Y axis, and have the most spread in X and Z. The range rate calculated covariance is approximately 2 orders of magnitude than that calculated with just range data. It is interesting to note that the range only data yields practically the same solution as using both data types. If this were to be true at all times, I would consider reformulating my batch processor to completely ignore any range rate calculations, potentially speeding up the processor. It makes sense to use the strongest data observation when available.

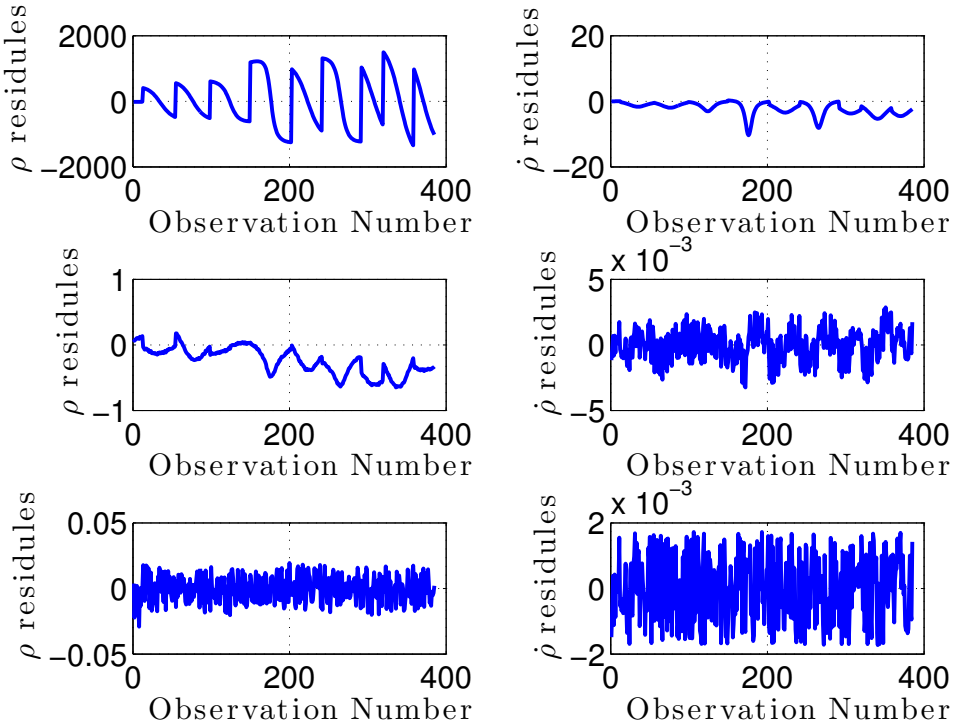


Figure 5: Kalman Filter RMS Residuals

5.2 Sequantial Kalman Filter

Moving from the batch to the sequential filter was not a particularly difficult task. The code for the Kalman filer is in Appendix B. For a comparison of RMS values as seen in 3, the Kalman-calculated RMS residuals are provided.

The RMS plots look nearly identical to those in Figure 3. This is to be expected, since the batch and Kalman filters are identical mathematically. Here again, the Kalman filter has converged on the solution 3 iterations in.

iter	1	2	3	
ρ_{res}	732.7443	0.32001	0.0097249	[m]
$\dot{\rho}_{res}$	2.9002	0.0011997	0.0009792	[m/s]

Table 4: Kalman Residuals

Comparing the RMS values between batch and Kalman methods (Table 1 and Table 4), the RMS values are almost completely identical.

A tabular comparison between final batch and final Kalman results is in Table 5. The differences are mostly minute. Variance values are on the same order of magnitude between batch and Kalman, and the state changes are similar. Timingwise, both methods took about the same amount of time, around 11 seconds.

State Parameter
x [m]
y [m]
z [m]
$\dot{x}$ [m/s]
$\dot{y}$ [m/s]
$\dot{z}$ [m/s]
μ $[m^3/s^2]$
$J2$
CD
X_{s1} [m]
Y_{s1} [m]
Z_{s1} [m]
X_{s2} [m]
Y_{s2} [m]
Z_{s2} [m]
X_{s3} [m]
Y_{s3} [m]
Z_{s3} [m]

:

ΔX **Batch** t_0	**Variance Batch**	ΔX **Kalman** t_0	**Variance Kalman**
0.29042	5.6626e-05	0.23485	0.00025253
-0.42217	0.00013911	-0.32768	1.9021e-05
-0.26187	0.00022029	-0.35247	0.00020926
0.040618	7.4632e-11	0.040674	9.9058e-10
0.032709	2.0861e-10	0.032604	5.2395e-11
-0.014415	1.0476e-10	-0.014472	2.8271e-10
-42769608.0625	172813484323.2254	-43919368.0625	251489194160.6534
-6.2748e-07	5.9809e-20	-6.2932e-07	2.5776e-19
0.1887	1.4492e-05	0.18951	1.487e-05
-6.5193e-09	1e-10	1.8608e-06	1e-10
-4.6566e-09	1e-10	1.3765e-06	1e-10
5.8322e-10	1e-10	-2.4488e-07	1e-10
-10.0084	2.7786e-05	-10.0224	3.9189e-05
10.0035	7.1368e-05	10.0657	0.00030324
5.9764	7.6755e-05	5.9587	9.718e-05
-5.0087	5.3859e-05	-5.0551	0.00018229
2.0213	0.00016251	2.1162	0.00070633
2.9761	0.00027417	2.9989	0.00030507

Table 5: Kalman and Batch Changes in X_t0

Though the batch and Kalman filter are mathematically equivalent, they both have situations in which they are particularly well suited. The Kalman filter is formulated well for live tracking solutions. The batch is good for mapping observations to a certain epoch for science missions and analysis. The Kalman filter has issues with poorly conditioned covariance matrices, and can require some computationally intensive work arounds. On the other hand, every time a new observation comes in, the Kalman filter is ready to

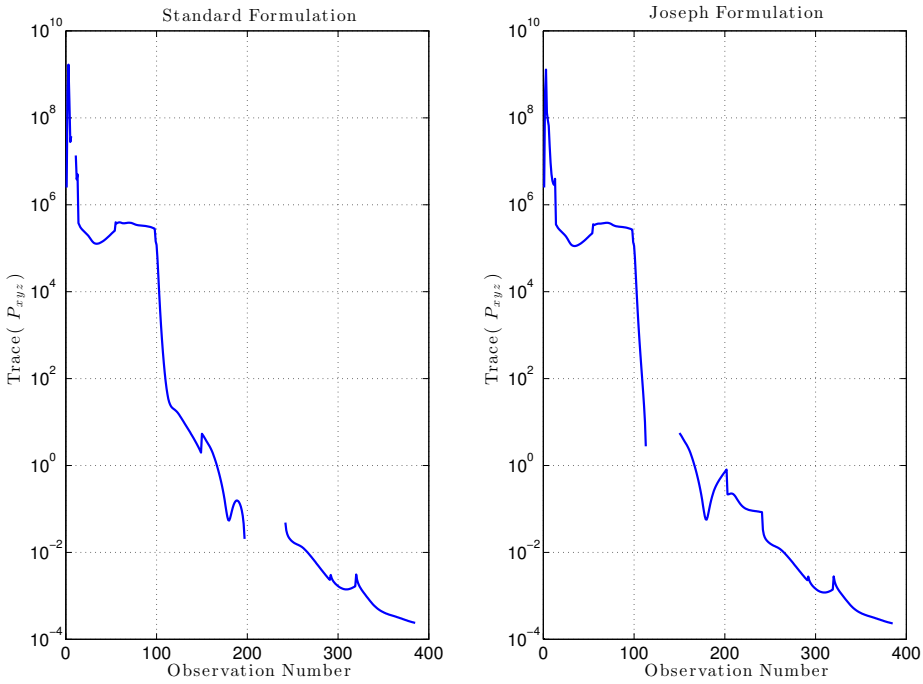


Figure 6: Kalman Covariance Trace

process it with minimal wait time. The batch processor would have to reinitialize, and reprocess everything between the epoch of interest and the new observation. It also requires the inversion of a large matrix, which is computationally intensive and can become problematic if the matrix is poorly conditioned. Both have their strengths, and both have their place in the operator's tool belt.

5.2.1 Kalman Covariance Analysis

Since the condition of the covariance matrix P is often the source of error in OD problems. For the Kalman Filter, a plot of its covariance matrix's trace versus time is provided in Figure 6. It features both the traditionally calculated covariance matrix, as well as the covariance found with the Joseph formulation.

The conventional and Joseph formulations are both very similar in how they performed. The only numerical issue that presented itself was in the plotting of the traces. Some trace values were negative, and you can't take the log of a negative number. This is shown as gaps in the plot. When analyzed, there are fewer negative trace values under the Joseph formulation, so it did do a better job of keeping the covariance matrix well conditioned.

As another examination of the covariance matrix, a 3-dimensional representation of the covariance position elements was plotted. In Figure 7, all axes represent physical space which the position elements of the covariance matrix exist. The ellipses are also positioned along the X axis as a function of time, just to show their time evolution more clearly.

Black shaded ellipsoids represent the covariance ellipses at a given time, and the blue ellipses are scaled semimajor/minor axes for viewing clarity.

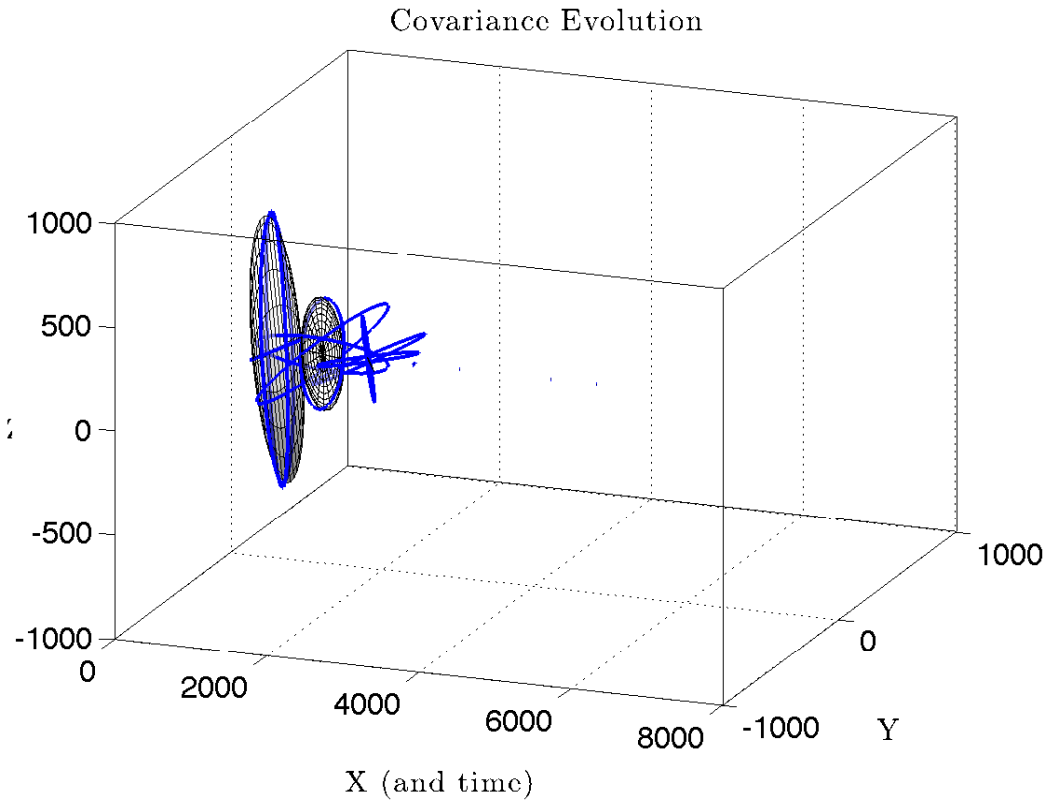


Figure 7: Time Evolution of Covariance Matrix

At the start of the tracking maneuver, near $X = 0$, the covariance ellipsoids are large, and their orientation changes quite a bit. Near the end of the tracking exercise, the ellipsoids are too small to see. Figure 8 shows a zoomed in portion of the plot, where you can see how fast the ellipsoids shrink. That trend matches the trace falloff from Figure 6. I found it particularly interesting how not just the size, but the orientation of the ellipsoids changed so much in the early times.

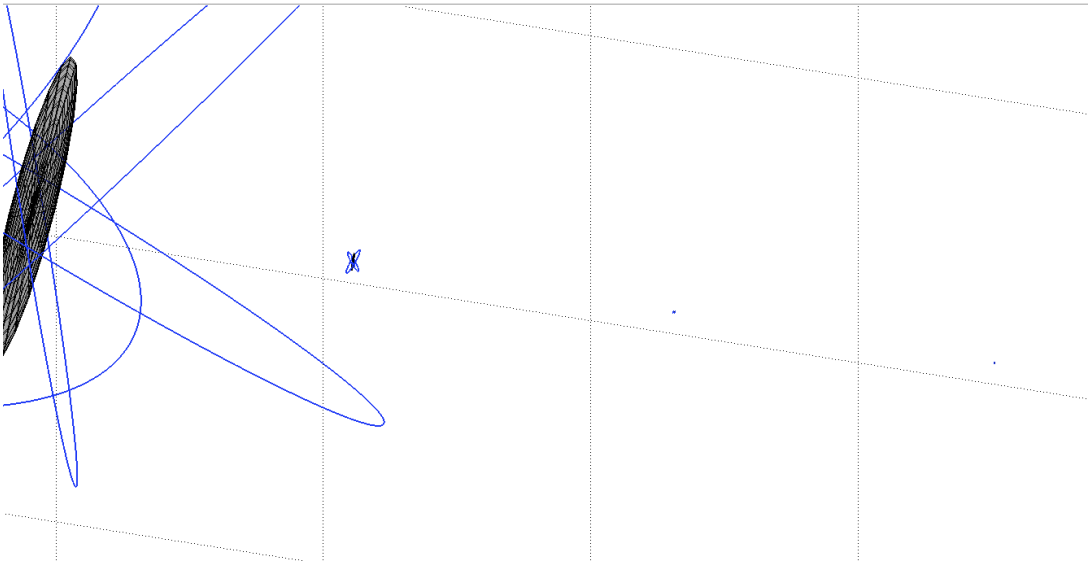


Figure 8: Close-up Time Evolution of Covariance Matrix

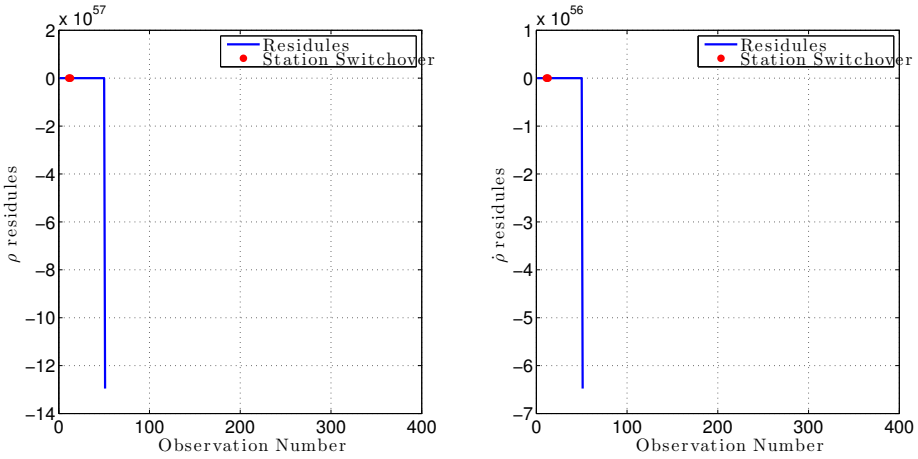


Figure 9: Raw EKF

5.3 Extended Kalman Filter

The Extended Kalman Filter (EKF, section 5.3) behaves in the same manner as the CKF in that it process observations one at a time. It differs in that with every observation, the EKF updates its reference trajectory with the latest information. The dynamical integration then uses the updated trajectory to compute a new reference trajectory which is supposed to be nearer the true one. By doing this, the aim is to not have to iterate orbit solutions, but rather use all information available to compute $\hat{x}_k$ at a given time.

It should be noted that the EKF is designed to adjust itself on the fly, so that it does not need to be iterated like the CKF and batch processors. However, because the state is reset between integrations, the state transition matrices collected cannot be used to map a trajectory back to an epoch. Hence, the EKF is not an iterable filter. It is meant for long observation streaks.

In practice, the EKF is much more sensitive to trajectory divergence, and requires more thought in implementation. Figure 9 shows the RMS results of the EKF when just the procedural changes from section 5.3 were integrated. For all calculations, the Joesph formulation for calculating P was used.

ρ_{res}	NAN	[m]
$\dot{\rho}_{res}$	NAN	[m/s]

Table 6: Raw EKF Residuals

Clearly, the filter has not converged. Immediately, the reference and true trajectory are too far apart, and updating the reference trajectory causes the trajectories to completely diverge.

5.3.1 EKF With CKF Switchover

To remedy the immediate divergence, I then tried waiting until 68 minutes into the orbit streak to start updating the reference trajectory. This essentially means that the CKF was running for a while, at which point the EKF took over and began updating its reference trajectory. I chose this because at 68 minutes, there begins a long streak of range observations (which was found to be the strong data set in section 5.1.1). See Figure 10. The hope here is that by switching to the EKF at a time when plenty of

observations are available, the true and reference trajectory won't have time to diverge. Between the first and second observation set, you can see a large gap in time. In this time, the trajectories have time to diverge and make the EKF break.

In addition, the big problem is that in the beginning of the orbit streak, the observation covariance matrix is much tighter than the main covariance matrix. The EKF immediately adjusted the orbit to match the observations, and in doing so, the trajectory diverges and voids assumptions of linearity. By processing some of the data with the CKF first, I allow the covariance matrix to shrink down as confidence in the reference trajectory is increased.

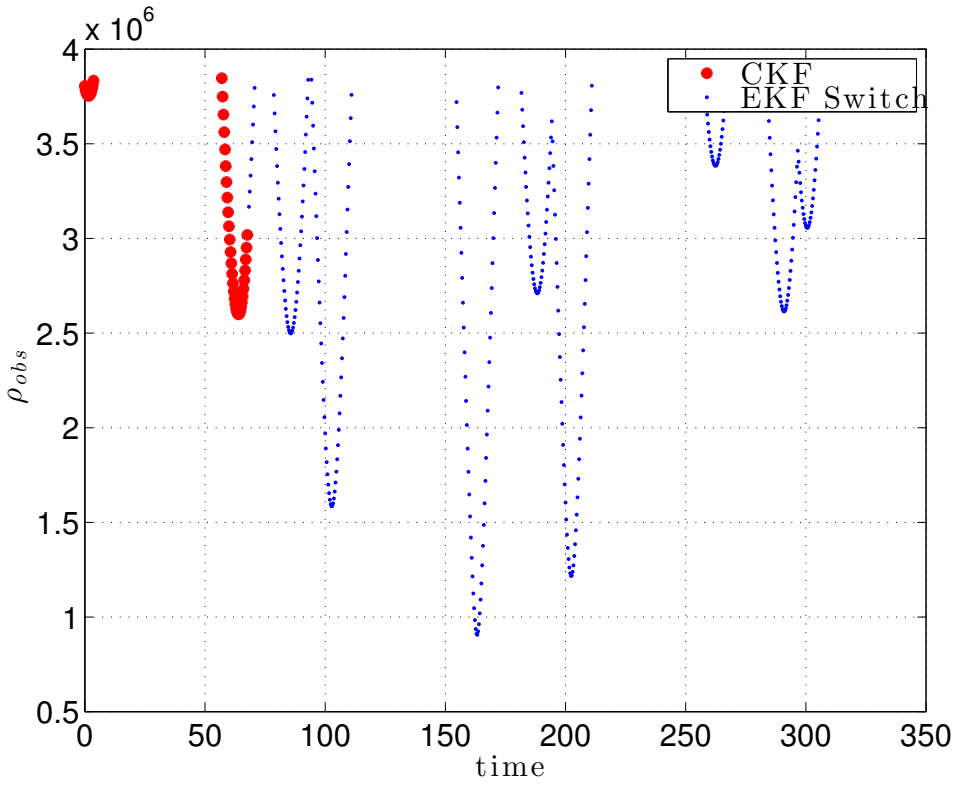


Figure 10: Range Observations

Delaying the switch over helped, but the trajectory still diverged. I decided that to switch over much longer than this would defeat the purpose of the EKF, as I wanted to see how it handled a decent chunk of the orbit. Implementing this yields Figure 11

The RMS here are still far too large to be considered a success, but they are numerical values now.

ρ_{res}	53551901211795920.00000	[m]
$\dot{\rho}_{res}$	966769064334839.75000	[m/s]

Table 7: EKF with CKF Switch Residuals

5.3.2 EKF With CKF Pre-Processing

The next addition to the EKF was a pre-processing run with the conventional Kalman filter. I believed the initial trajectory (X_0^*) was already too far diverged for the EKF to

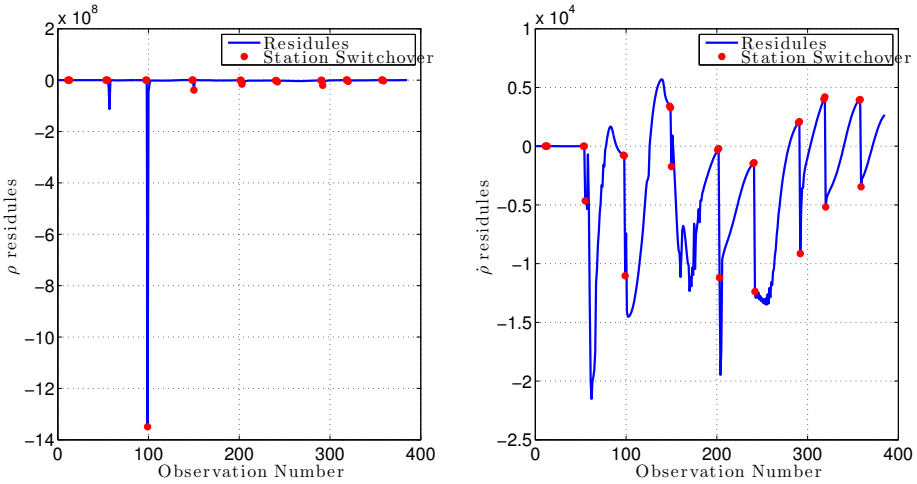


Figure 11: EKF With Kalman Switchover

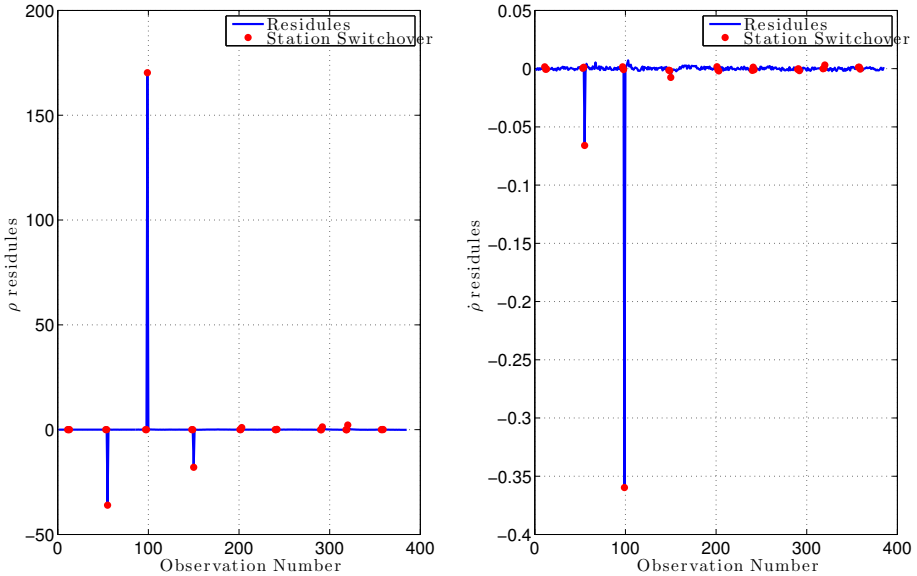


Figure 12: EKF With CKF PreProcessing and Switchover

run correctly. By processing 180 minutes worth of observations for 3 iterations with the CKF, and providing that newly found X_0^* to the EKF, a better initial guess is available and should keep the EKF from diverging as quickly. Note this is done in addition to running the CKF from 68 minutes of the orbit. One is meant to account for errors in the initial state, and one is to allow the covariance matrix to shrink.

ρ_{res}	8.91889	[m]
$\dot{\rho}_{res}$	0.01868	[m/s]

Table 8: EKF with CKF Switch and PreProcessing Residuals

Figure 12 shows the result now, with CKF preprocessing and switchover, and tabular values in Table 8

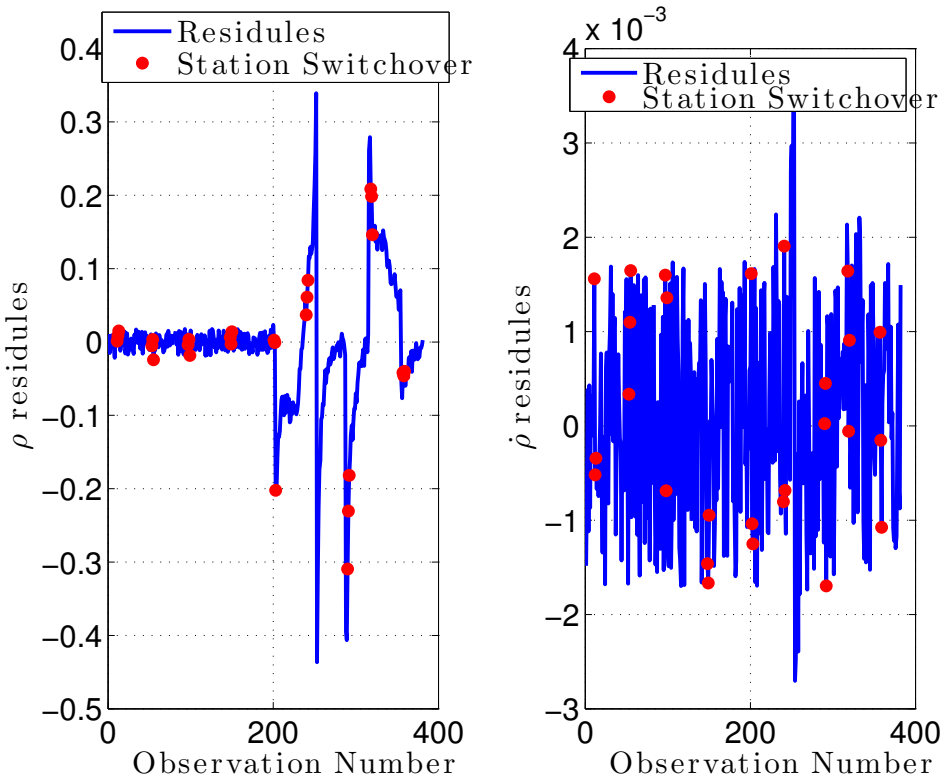


Figure 13: EKF With CKF PreProcessing and Switchover and Covariance Shrinking

5.3.3 EKF With CKF Pre-Processing and Covariance Regulation

The last adjustment applied was to try to fix the large jumps seen in 12. I plotted when each of the tracking stations switched over in red dots. Doing this, I noticed that all of the large jumps in residuals occur when stations switch. That makes since, since each station has its own section in time. When there is a large gap in the observation times, the trajectories get a chance to diverge. Then when observations come in again, the EKF overreacts and jumps towards the observation set since the observation covariance is so small.

To combat this, I manually put in a 'covariance shrinker'. When the station switched over, I shrunk the covariance matrix by a factor of 100. After that observation, I reset it its actual value. Essentially, this told the filter to give extra weight to the reference trajectory for observations located right at a station switch over.

ρ_{res}	0.08163	[m]
$\dot{\rho}_{res}$	0.00107	[m/s]

Table 9: EKF with CKF Switch and PreProcessing and Covariance Shrinking Residuals

This method appears to work the best of any observed. The residuals are nearly on the same order as the CKF. In addition, it is conceivable that with a Monte Carlo simulation or more time to tune the filter, its performance would improve.

While it seems initially to be a waste of time to try and fiddle with the EKF when the standard CKF or batch processor works better, the EKF does have its advantages. Like the CKF, the EKF is a live processor. It is ready immediately to process new observations. Its advantage over the CKF is that it can be used for longer observation runs, and can

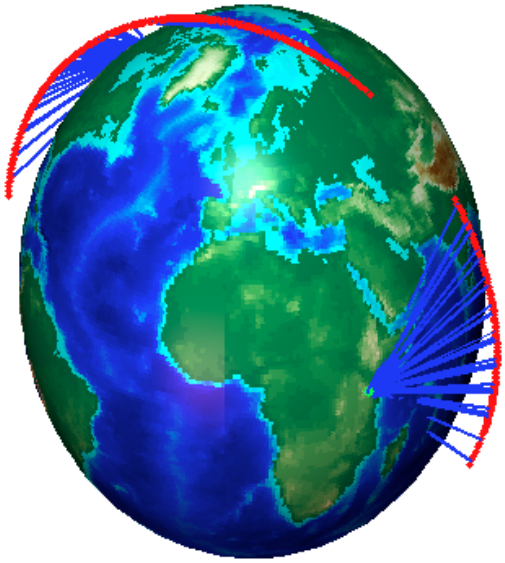


Figure 14: Trajectory Visulization

work despite slightly mismodeled dynamics. Over time, the CKF will almost always diverge, as errors in modeling or linearization assumptions cause the reference trajectory to wander from the truth. The EKF updates the reference with every observation, so it could theoretically run forever without needing to be reset in a really well formulated system.

Figure 14 shows a simulation of the orbit path, observations, and ground stations on the earth throughout this project duration. The plotter for this was developed between Pierce Martin and I, with the bulk of it found under the BSD license on the Mathworks site. It looks pretty cool, but is also a good tool to visualize the orbit problem we are seeing. Figure 10 shows large gaps between some observation streaks. By seeing the full visualization of the OD problem you can see that there is a long period as the satellite goes over the south pole where it isn't in range of a ground station. It is in these times that the reference and true trajectory is likely to have the most time to diverge, and cause the EKF to break down. As insightful as equations and matrices are, sometimes it is just really helpful to see a nice picture.

6 Conclusion

There are various ways to estimate the trajectory of a satellite. However, they are almost all rooted in the same fundamental principles. An estimator is built to determine a deviation between a true and reference trajectory by weighing observations of the satellite and a model's estimate of the trajectory, based on assumptions that the deviation is linear. The weighing is done in a way to minimize a cost function in a statistical manner.

The simplest formulation of an orbit filter is the batch processor. It is well formulated, and is typically more robust to poor conditioning. It's main feature is that it processes all observations and state estimates at a single time.

Mathematically equivalent is the sequential, or Kalman filter. As opposed to the batch processor, the CKF processes observations one at a time. It is susceptible to poorly conditioned covariance matrices, which occurs often during processing. There are several numerical fixes that have been derived to account for this phenomenon.

What both the batch and CKF lack is the ability to use their findings mid-run to adjust the reference trajectory to match what is observed. This is where the EKF comes in. By updating the reference trajectory, the EKF aims to allow for longer processing runs without reseting or iterating. It will keep the reference trajectory within a linear deviation from the true trajectory long after the batch or CKF have diverged. However, it often takes more of an effort to get to work efficiently in practice.

Overall, I really enjoyed the course and this project. It seems enormously practical, and I really think I learned a ton. In the future, I think it would be good to include a much much longer observation dataset. One where batch and CKF solutions might diverge, but the EKF could shine through. In addition, I think it would be fun for students to submit their code for speed challenges. See whose is the most optimized. Finally, the one thing I wish we would have tried was a DMC assignment. A real one as it applies to our project. DMC seems really useful and powerful, and I wish we would have gotten to try it out. But overall this was a really great class!

References

[1]

[2] G. B. Jeffry Parker. Asen 5070 f2012 lecture 11. `http://ccar.colorado.edu/ASEN5070/lectures.htm`.

[3] G. B. Jeffry Parker. Asen 5070 f2012 lecture 23. `http://ccar.colorado.edu/ASEN5070/lectures.htm`.

[4] B. Tapley, B. Schutz, and G. Born. *Statistical Orbit Determination*. Elsevier Acad. Press, 2004.

Batch Processor Code

```
% function Xstar0 = BatchProcess()
% Compute new xhat
% Looks for functions to return:
%   G
%   Htilde
% Inside of /Filters/BatchTools.m

clc; clear all;close all; format compact;tic
warning off MATLAB:nearlySingularMatrix

%% First, Load and Parse Observation Data
load('Observations.mat');
t_obs       = obs(:,1);
station     = obs(:,2);
rho_obs     = obs(:,3);
rhodot_obs  = obs(:,4);

%% Pre-Allocations
x       = zeros(length(obs));
y       = x;
z       = x;
xdot    = x;
ydot    = x;
zdot    = x;
Xsite1  = x;
Ysite1  = x;
Zsite1  = x;
Xsite2  = x;
Ysite2  = x;
Zsite2  = x;
Xsite3  = x;
Ysite3  = x;
Zsite3  = x;
Phi     = cell(1,length(obs));
y_res   = zeros(2,length(obs));
Xsite   = x;
Ysite   = x;
Zsite   = x;
theta   = x;


%% Initialize Variables

% Calculations for rho, rhodot. Put into bigger functiom sometime
%-------------------------------------------------
findrhostar      = @(x,y,z,Xsite,Ysite,Zsite,theta) sqrt(x^2+y^2+z^2+Xsite^2+Ysite^2
findrhodotstar   = @(x,y,z,xdot,ydot,zdot,Xsite,Ysite,Zsite,theta,theta_dot,rho) (x*
                   +(xdot*Ysite - ydot*Xsite)*sin(theta) + theta_dot*(x*Ysite - y*X
                                                          /rho;
%-------------------------------------------------

% System Constants
%-------------------------------------------------
Phi_Init    = eye(18,18);
tol         = 1e-13;
```

```
uE          = 3.986004415e14;        % m^3/s^2
J2          = 1.082626925638815e-3;  % []
Cd          = 2;                     % []
theta_dot   = 7.29211585530066e-5;   % rad/s
time        = t_obs;
%--------------------------------------------------

% Information
%--------------------------------------------------
sigma_rho   = 0.01;                  % rms std
sigma_rhodot= 0.001;                 % rms std
R           = [sigma_rho^2  ,         0        ; ...
                    0       ,    sigma_rhodot^2];
W = inv(R);
% Range only
% W(2,2)=0;
Pbar0       = diag([1e6,1e6,1e6,1e6,1e6,1e6,1e20,1e6,1e6,1e-10,1e-10,1e-10,1e6,1e6,1
%--------------------------------------------------

% Initial Conditions
%--------------------------------------------------
RV_Init     = [757700,5222607.0,4851500.0,2213.21,4678.34,-5371.30];
Station_Init= [-5127510.0 , -3794160.0 , 0.0 ,...                %101
                3860910.0 , 3238490.0  , 3898094.0 , ...         %337
                549505.0  , -1380872.0 , 6182197.0 ];            %394
Const_Init = [uE , J2 , Cd ];
%--------------------------------------------------

% Form Initialization State
%--------------------------------------------------
Xstar0 = [RV_Init , Const_Init , Station_Init , reshape(Phi_Init,1,length(Phi_Init)^
%--------------------------------------------------

% Initial xbar0, or a-priori state deviation from reference trajectory
%--------------------------------------------------
xbar0       = zeros(18,1);
%--------------------------------------------------


%% Perform Batch Loop
num_iterations = 3;
res=[];
for ii = 1:num_iterations

    % Dynamical Integration
    %--------------------------------------------------
    tol_mat     = ones(size(Xstar0)) .* tol;
    options     = odeset('RelTol',tol,'AbsTol',tol,'OutputFcn',@odetpbar);

    [time,StatePhi] = ode45(@StateDeriv_WithPhi,time,Xstar0,options);
    %--------------------------------------------------


        % Batch Processing Part
    %--------------------------------------------------
    Lam = inv(Pbar0);
    N   = Pbar0\xbar0;  % same as inv(pobar)*xbar0
```

```
% Reform Phi Matrix
%-------------------------------------------------
for jj = 1:length(time)
       Phi      = reshape(StatePhi(jj,19:end),size(Phi_Init));
       Xstar    = StatePhi(:,1:18);
       x        = Xstar(:,1);
       y        = Xstar(:,2);
       z        = Xstar(:,3);
       xdot     = Xstar(:,4);
       ydot     = Xstar(:,5);
       zdot     = Xstar(:,6);
       Xsite1   = Xstar(:,10);
       Ysite1   = Xstar(:,11);
       Zsite1   = Xstar(:,12);
       Xsite2   = Xstar(:,13);
       Ysite2   = Xstar(:,14);
       Zsite2   = Xstar(:,15);
       Xsite3   = Xstar(:,16);
       Ysite3   = Xstar(:,17);
       Zsite3   = Xstar(:,18);

%-------------------------------------------------
```

```
%     parfor jj = 1:length(rho_obs)
        theta(jj)          = theta_dot*time(jj);
        Htilde             = zeros(2,18);
        % Check Stations
        %-------------------------------------------------
        %Station 1
        if station(jj) == 101
            Xsite(jj)   = Xsite1(jj);   Ysite(jj)=Ysite1(jj);   Zsite(jj)=Zsite1(jj)
            % Find H Tilde
            %-------------------------------------------------
            Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x
            %-------------------------------------------------
            Htilde  = [Htilde , zeros(2,6)];

        end

        %Station 2
        if station(jj) == 337
            Xsite(jj)   = Xsite2(jj);   Ysite(jj)=Ysite2(jj);   Zsite(jj)=Zsite2(jj)
            % Find H Tilde
            %-------------------------------------------------
            Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x
            %-------------------------------------------------
            Htilde  = [Htilde(:,1:9) , zeros(2,3), Htilde(:,10:12),zeros(2,3)];
        end

        %Station 3
        if station(jj) == 394
            Xsite(jj)   = Xsite3(jj);   Ysite(jj)=Ysite3(jj);   Zsite(jj)=Zsite3(jj)
            % Find H Tilde
```

```
            %-------------------------------------------------------
            Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x
            %-------------------------------------------------------
            Htilde  = [Htilde(:,1:9),zeros(2,6),Htilde(:,10:12)];
        end
        %-------------------------------------------------------

        % Map To Epoch
        %-------------------------------------------------------
        H{jj}    = Htilde*Phi;
        %-------------------------------------------------------

        % Cumulate INFORMATION?? Matrix
        %-------------------------------------------------------
        Lam       = Lam + H{jj}'*W*H{jj};
        %-------------------------------------------------------

        % Put into FindG
        %-------------------------------------------------------
        rhostar   = findrhostar(x(jj),y(jj),z(jj),Xsite(jj),Ysite(jj),Zsite(jj),thet
        rhodotstar= findrhodotstar(x(jj),y(jj),z(jj),xdot(jj),ydot(jj),zdot(jj),Xsit
        %-------------------------------------------------------

        % Find Observation Deviations
        %-------------------------------------------------------
        ystar       = [rhostar;rhodotstar];
        y_res(:,jj) = [rho_obs(jj);rhodot_obs(jj)] - ystar;
        %-------------------------------------------------------


        % Cumulate SOMETING?? Matrix
        %-------------------------------------------------------
        N     = N + H{jj}'*W*y_res(:,jj);
        %-------------------------------------------------------
    end
    fprintf('RMS of rho is :  %3.5f \n',rms(y_res(1,:)))
    fprintf('RMS of rhodot is :  %3.5f \n',rms(y_res(2,:)))
    % Find New State Deviation
    %-------------------------------------------------------
    xhat0   = Lam\(N);
    %-------------------------------------------------------

    % Update Best Guess of Initial Conditions
    %-------------------------------------------------------
    Xstar0  = [Xstar0(1:18) + xhat0; (reshape(Phi_Init,length(Phi_Init)^2,1))];
    %-------------------------------------------------------

    % Update a-priori State Deviation% function Xstar0 = BatchProcess()
    %-------------------------------------------------------
    xbar0 = xbar0 - xhat0;
    %-------------------------------------------------------


    figure(1)
    subplot(num_iterations,2,2*ii-1)
    plot(y_res(1,:))
    ylabel('$\rho$ residuals')
    xlabel('observation number')
```

```
    subplot(num_iterations,2,2*ii)
    plot(y_res(2,:))
    ylabel('$\dot{\rho}$ residuals')
    xlabel('observation number')

    res = [res,[rms(y_res(1,:));rms(y_res(2,:))]];

end

fprintf('\n\nRunning Time for Batch Processor : %3.5f\n\n',toc)

% For output stuff
% table=[Xstar0(1:18),diag(inv(Lam))];
Pfinal = Phi*inv(Lam)*Phi';
state=Xstar0(1:18);
cov     = diag(inv(Lam));
change = Xstar0(1:18)-[RV_Init , Const_Init , Station_Init]';
 save('Batch.mat','state','cov','Pfinal','change');

% figure_awesome('save','eps')

% t=[Xstar0(1:18),Xstar0(1:18)-[RV_Init , Const_Init , Station_Init]', diag(inv(Lam)
```

Kalman Filter

```
% function Xstar0 = KalmanFilter()
% Compute new xhat
% Looks for functions to return:
%   G
%   Htilde
% Inside of /Filters/KalmanTools.m
%
% Notation:    t0 is a priori, like M_(t_i-1)
%              t1 is current step   M_(ti)

clc; clear all;close all; format compact;tic
warning off MATLAB:nearlySingularMatrix
res = [];

%% First, Load and Parse Observation Data
load('Observations.mat');
t_obs       = obs(:,1);
station     = obs(:,2);
rho_obs     = obs(:,3);
rhodot_obs  = obs(:,4);

%% Pre-Allocations
x       = zeros(length(obs));
y       = x;
z       = x;
xdot    = x;
ydot    = x;
zdot    = x;
Xsite1  = x;
Ysite1  = x;
Zsite1  = x;
Xsite2  = x;
```

```
Ysite2  = x;
Zsite2  = x;
Xsite3  = x;
Ysite3  = x;
Zsite3  = x;
y1   = zeros(2,length(obs));
Xsite   = x;
Ysite   = x;
Zsite   = x;
theta   = x;


%% Initialize Variables

% Calculations for rho, rhodot. Put into bigger functiom sometime
%----------------------------------------------------
findrhostar      = @(x,y,z,Xsite,Ysite,Zsite,theta) sqrt(x^2+y^2+z^2+Xsite^2+Ysite^2
findrhodotstar   = @(x,y,z,xdot,ydot,zdot,Xsite,Ysite,Zsite,theta,theta_dot,rho) (x*
                   +(xdot*Ysite - ydot*Xsite)*sin(theta) + theta_dot*(x*Ysite - y*X
                                                          /rho;
%----------------------------------------------------

% System Constants
%----------------------------------------------------
Phi_Init    = eye(18,18);
tol         = 1e-13;
uE          = 3.986004415e14;        % m^3/s^2
J2          = 1.082626925638815e-3;  % []
Cd          = 2;                     % []
theta_dot   = 7.29211585530066e-5;   % rad/s
time        = t_obs;
%----------------------------------------------------

% Information
%----------------------------------------------------
sigma_rho   = 0.01;                  % rms std
sigma_rhodot= 0.001;                 % rms std
R           = [sigma_rho^2  ,         0        ; ...
                    0       ,    sigma_rhodot^2];
W = inv(R);
Pbar0       = diag([1e6,1e6,1e6,1e6,1e6,1e6,1e20,1e6,1e6,1e-10,1e-10,1e-10,1e6,1e6,1
%----------------------------------------------------

% Initial Conditions
%----------------------------------------------------
RV_Init     = [757700,5222607.0,4851500.0,2213.21,4678.34,-5371.30];
Station_Init= [-5127510.0 , -3794160.0 , 0.0 ,...               %101
                3860910.0 , 3238490.0  , 3898094.0 , ...        %337
                549505.0  , -1380872.0 , 6182197.0 ];           %394
Const_Init = [uE , J2 , Cd ];
%----------------------------------------------------

% Form Initialization State
%----------------------------------------------------
Xstar0 = [RV_Init , Const_Init , Station_Init , reshape(Phi_Init,1,length(Phi_Init)^
%----------------------------------------------------

% Initialize Kalman Filter
```

```
%-------------------------------------------------------
xbar       = zeros(18,1);
xhat       = xbar;

Phi_tk_t0   = Phi_Init;%ones(size(Phi_Init));
%-------------------------------------------------------


%% Perform Kalman Loop
num_iterations = 3;
 for ii = 1:num_iterations
     tr=[];
     tr_J = [];
     tr_P = [];
    P(:,:,1)           = Pbar0;

    % Dynamical Integration
    %-------------------------------------------------
    tol_mat     = ones(size(Xstar0)) .* tol;
    options     = odeset('RelTol',tol,'AbsTol',tol,'OutputFcn',@odetpbar);

    [time,StatePhi] = ode45(@StateDeriv_WithPhi,time,Xstar0,options);
    %-------------------------------------------------


    % Reform Phi Matrix
    %-------------------------------------------------
    for jj = 1:length(time)
           Phi(:,:,jj)     = reshape(StatePhi(jj,19:end),size(Phi_Init));
           Xstar    = StatePhi(:,1:18);
           x        = Xstar(:,1);
           y        = Xstar(:,2);
           z        = Xstar(:,3);
           xdot     = Xstar(:,4);
           ydot     = Xstar(:,5);
           zdot     = Xstar(:,6);
           Xsite1   = Xstar(:,10);
           Ysite1   = Xstar(:,11);
           Zsite1   = Xstar(:,12);
           Xsite2   = Xstar(:,13);
           Ysite2   = Xstar(:,14);
           Zsite2   = Xstar(:,15);
           Xsite3   = Xstar(:,16);
           Ysite3   = Xstar(:,17);
           Zsite3   = Xstar(:,18);

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


    % Htilde, yi, Ki
    %-------------------------------------------------
        theta(jj)          = theta_dot*time(jj);
        Htilde             = zeros(2,18);
        % Check Stations
        %-----------------------------------------------
        %Station 1
        if station(jj) == 101
            Xsite(jj)   = Xsite1(jj);   Ysite(jj)=Ysite1(jj);   Zsite(jj)=Zsite1(jj)
```

```
    % Find H Tilde
    %-------------------------------------------------------
    Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x
    %-------------------------------------------------------
    Htilde  = [Htilde , zeros(2,6)];

end

%Station 2
if station(jj) == 337
    Xsite(jj)   = Xsite2(jj);   Ysite(jj)=Ysite2(jj);   Zsite(jj)=Zsite2(jj)
    % Find H Tilde
    %-------------------------------------------------------
    Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x
    %-------------------------------------------------------
    Htilde  = [Htilde(:,1:9) , zeros(2,3), Htilde(:,10:12),zeros(2,3)];
end

%Station 3
if station(jj) == 394
    Xsite(jj)   = Xsite3(jj);   Ysite(jj)=Ysite3(jj);   Zsite(jj)=Zsite3(jj)
    % Find H Tilde
    %-------------------------------------------------------
    Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x
    %-------------------------------------------------------
    Htilde  = [Htilde(:,1:9),zeros(2,6),Htilde(:,10:12)];
end
%-----------------------------------------------------------



% Time Update
%-----------------------------------------------------------
if jj > 1
    Phi_step= Phi(:,:,jj)/Phi(:,:,jj-1);
    xbar(:,:,jj)   = Phi_step*xhat(:,:,jj-1);
    P(:,:,jj)   = Phi_step*P(:,:,jj-1)*Phi_step';

end
%-----------------------------------------------------------

% Put into FindG
%-----------------------------------------------------------
rhostar   = findrhostar(x(jj),y(jj),z(jj),Xsite(jj),Ysite(jj),Zsite(jj),thet
rhodotstar= findrhodotstar(x(jj),y(jj),z(jj),xdot(jj),ydot(jj),zdot(jj),Xsit
%-----------------------------------------------------------

% Find Observation Deviations
%-----------------------------------------------------------
ystar       = [rhostar;rhodotstar];
y1(:,jj)    = [rho_obs(jj);rhodot_obs(jj)] - ystar;
%-----------------------------------------------------------


% Kalman Gain
%-----------------------------------------------------------
K1          = P(:,:,jj)*Htilde'*inv(Htilde*P(:,:,jj)*Htilde' + R);
%-----------------------------------------------------------
```

```
        % Measurement Update
        %-------------------------------------------------
        xhat(:,:,jj) = xbar(:,:,jj) + K1*(y1(:,jj) - Htilde*xbar(:,:,jj));
%        P(:,:,jj) = (eye(size(K1*Htilde)) - K1*Htilde)*P(:,:,jj)*(eye(size(K1*Htild
       P(:,:,jj) = (eye(size(K1*Htilde)) - K1*Htilde)*P(:,:,jj);
%        P_P(:,:,jj) = ComputeP(Htilde,P(:,:,jj),R,Xstar(jj,1:18),ystar,'potter');

%         P(:,:,jj) = ComputeP(Htilde,P(:,:,jj),R,Xstar(jj,1:18),ystar,'potter');
        %-------------------------------------------------

        tr = [tr,trace(P(1:3,1:3,jj))];
%         tr_J = [tr_J,trace(P_J(1:3,1:3,jj))];
%         tr_P = [tr_P,trace(P_J(1:3,1:3,jj))];

    end   % End observation loop


    P_full=P;
    % Update Best Guess of Initial Conditions
    %-----------------------------------------------------%
     Xstar0 = [Xstar0(1:18) + inv(Phi(:,:,end))*xhat(:,:,end); (reshape(Phi_Init,ler
     P      = inv(Phi(:,:,end))*P(:,:,end)*inv(Phi(:,:,end))';
     xbar   = xbar(:,:,end-1) - xhat(:,:,end-1);



    %-------------------------------------------------
    fprintf('RMS of rho is :  %3.5f \n',rms(y1(1,:)))
    fprintf('RMS of rhodot is :  %3.5f \n',rms(y1(2,:)))



    figure(1)
    subplot(num_iterations,2,2*ii-1)
    plot(y1(1,:))
    ylabel('$\rho$ residules')
    xlabel('Observation Number')

    subplot(num_iterations,2,2*ii)
    plot(y1(2,:))
    ylabel('$\dot{\rho}$ residules')
    xlabel('Observation Number')

    if ii == 1
        tr_J = load('Kalman.mat');
        figure(2)
        subplot(1,2,1)
        semilogy(tr2)
        xlabel('Observation Number');ylabel('Trace( $P_{xyz}$ )');title('Standard Fo
        subplot(1,2,2)
        semilogy(tr)
        xlabel('Observation Number');ylabel('Trace( $P_{xyz}$ )');title('Joseph Form
%           subplot(1,3,3)
%         semilogy(tr_P)
%         xlabel('Observation Number');ylabel('Trace( $P_{xyz}$ )');title('Potter Fo
    end
```

```
        res = [res,[rms(y1(1,:));rms(y1(2,:))]];

    end

    % epic function by Pierce Martin
    RE      = 6378136.3;      % m    Radius of earth
    plot_X_star(Xstar',RE,station)

    figure
    hold on
    for ii=1:length(time)
        if mod(ii,30) == 0
            error_ellipse(P_J(1:3,1:3,ii),[ii*20,0,0]);
        end
    end
    xlabel('X (and time)');ylabel('Y');zlabel('Z'); title('Covariance Evolution')

   fprintf('\n\nRunning Time for Kalman Filter : %3.5f\n\n',toc)

   changeK = Xstar0(1:18)-[RV_Init , Const_Init , Station_Init]';



   % Output stuff
   matrix2latex(res,'table.txt')
```

Extended Kalman

```
     % function Xstar0 = ExtendedKalmanFilter()
% Compute new xhat
% Looks for functions to return:
%   G
%   Htilde
% Inside of /Filters/KalmanTools.m
%
% Notation:    t0 is a priori, like M_(t_i-1)
%              t1 is current step   M_(ti)

clc; clear all;close all; format compact;tic
warning off MATLAB:nearlySingularMatrix
set(0,'defaulttextinterpreter','latex')

prekalman = 1;



%% First, Load and Parse Observation Data
load('Observations.mat');
t_obs       = obs(:,1);
station     = obs(:,2); station(end+1) = station(end);station(end+1) = station(end);stat
rho_obs     = obs(:,3);
rhodot_obs  = obs(:,4);

%% Pre-Allocations
x       = zeros(length(obs),1);
y       = x;
z       = x;
```

```
xdot    = x;
ydot    = x;
zdot    = x;
Xsite1  = x;
Ysite1  = x;
Zsite1  = x;
Xsite2  = x;
Ysite2  = x;
Zsite2  = x;
Xsite3  = x;
Ysite3  = x;
Zsite3  = x;
y1   = zeros(2,length(obs));
Xsite   = x;
Ysite   = x;
Zsite   = x;
theta   = x;


%% Initialize Variables

% Calculations for rho, rhodot. Put into bigger functiom sometime
%-------------------------------------------------
findrhostar      = @(x,y,z,Xsite,Ysite,Zsite,theta) sqrt(x^2+y^2+z^2+Xsite^2+Ysite^2+Zsi
findrhodotstar   = @(x,y,z,xdot,ydot,zdot,Xsite,Ysite,Zsite,theta,theta_dot,rho) (x*xdot
                   +(xdot*Ysite - ydot*Xsite)*sin(theta) + theta_dot*(x*Ysite - y*Xsite
                                                          /rho;
%-------------------------------------------------

% System Constants
%-------------------------------------------------
Phi_Init    = eye(18,18);
tol         = 1e-9;
uE          = 3.986004415e14;        % m^3/s^2
J2          = 1.082626925638815e-3;  % []
Cd          = 2;                     % []
theta_dot   = 7.29211585530066e-5;   % rad/s
time        = t_obs;
%-------------------------------------------------

% Information
%-------------------------------------------------
sigma_rho   = 0.01;                  % rms std
sigma_rhodot= 0.001;                 % rms std
R           = [sigma_rho^2  ,          0        ; ...
                    0       ,    sigma_rhodot^2];
W = inv(R);
Pbar0       = diag([1e6,1e6,1e6,1e6,1e6,1e6,1e20,1e6,1e6,1e-10,1e-10,1e-10,1e6,1e6,1e6,1
%-------------------------------------------------

% Initial Conditions
%-------------------------------------------------
RV_Init     = [757700,5222607.0,4851500.0,2213.21,4678.34,-5371.30];
Station_Init= [-5127510.0 , -3794160.0 , 0.0 ,...                %101
                3860910.0 , 3238490.0  , 3898094.0 , ...         %337
                549505.0  , -1380872.0 , 6182197.0 ];            %394
Const_Init = [uE , J2 , Cd ];
%-------------------------------------------------
```

```
% Form Initialization State
%-------------------------------------------------------
Xstar0 = [RV_Init , Const_Init , Station_Init , reshape(Phi_Init,1,length(Phi_Init)^2)]'
%-------------------------------------------------------


%% Pre-Filter
% Perform Initial Kalman Filter
%-------------------------------------------------------
if prekalman == 1
    num_iterations=3;
    cutoff_time=180*60;
    output=0;
    Xstar0=KalmanFunction(num_iterations,cutoff_time,output);
end
%-------------------------------------------------------

%% Switch to Extended Kalman Filter
 cutoff_time=68*60;

% Initialize Extended Kalman Filter
%-------------------------------------------------------
xbar       = zeros(18,1);
xhat       = zeros(18,length(time));
%-------------------------------------------------------
num_iterations = 1;

textprogressbar('EKF Progress : ');
 for ii = 1:num_iterations
    tr=[];
    P(:,:,1)          = Pbar0;

    % Dynamical Integration Tolerances
    %-------------------------------------------------
    tol_mat     = ones(size(Xstar0)) .* tol;
    options     = odeset('RelTol',tol,'AbsTol',tol);
    %-------------------------------------------------


    StatePhi = Xstar0;

    StSw = [0,0];
    for jj = 1:length(time)
        progress=100 * jj/length(time);
        textprogressbar(progress);

        % Perform Dynamical Integration
        %-----------------------------------------------
        if jj > 1
            [t,tmp] = ode45(@StateDeriv_WithPhi,[time(jj-1),time(jj)],StatePhi(:,jj-1),o
            StatePhi(:,jj) = tmp(end,:)';
        else
            StatePhi(:,jj) = Xstar0;
        end
        %-----------------------------------------------

        % Reform Phi Matrix
```

```
%-------------------------------------------------------
Phi(:,:,jj)   = reshape(StatePhi(19:end,jj),size(Phi_Init));
Xstar         = StatePhi(1:18,jj);
x(jj)         = Xstar(1);
y(jj)         = Xstar(2);
z(jj)         = Xstar(3);
xdot(jj)      = Xstar(4);
ydot(jj)      = Xstar(5);
zdot(jj)      = Xstar(6);
Xsite1(jj)    = Xstar(10);
Ysite1(jj)    = Xstar(11);
Zsite1(jj)    = Xstar(12);
Xsite2(jj)    = Xstar(13);
Ysite2(jj)    = Xstar(14);
Zsite2(jj)    = Xstar(15);
Xsite3(jj)    = Xstar(16);
Ysite3(jj)    = Xstar(17);
Zsite3(jj)    = Xstar(18);

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


% Htilde, yi, Ki
%-------------------------------------------------------
theta(jj)           = theta_dot*time(jj);
Htilde              = zeros(2,18);
% Check Stations
%-------------------------------------------------------
%Station 1
if station(jj) == 101
    Xsite(jj)    = Xsite1(jj);    Ysite(jj)=Ysite1(jj);    Zsite(jj)=Zsite1(jj);
    % Find H Tilde
    %-------------------------------------------------------
    Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x(jj),
    %-------------------------------------------------------
    Htilde  = [Htilde , zeros(2,6)];

end

%Station 2
if station(jj) == 337
    Xsite(jj)    = Xsite2(jj);    Ysite(jj)=Ysite2(jj);    Zsite(jj)=Zsite2(jj);
    % Find H Tilde
    %-------------------------------------------------------
    Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x(jj),
    %-------------------------------------------------------
    Htilde  = [Htilde(:,1:9) , zeros(2,3), Htilde(:,10:12),zeros(2,3)];
end

%Station 3
if station(jj) == 394
    Xsite(jj)    = Xsite3(jj);    Ysite(jj)=Ysite3(jj);    Zsite(jj)=Zsite3(jj);
    % Find H Tilde
    %-------------------------------------------------------
    Htilde = FindHtilde(Xsite(jj),Ysite(jj),Zsite(jj),theta(jj),theta_dot,x(jj),
    %-------------------------------------------------------
    Htilde  = [Htilde(:,1:9),zeros(2,6),Htilde(:,10:12)];
end
```

```
        %-------------------------------------------------------
        % Time Update
        %-------------------------------------------------------
        if jj > 1
%           Phi_step= Phi(:,:,jj)/Phi(:,:,jj-1);
            Phi_step    = Phi(:,:,jj);
            xbar(:,jj)  = Phi_step*xhat(:,jj-1);
            P(:,:,jj)   = Phi_step*P(:,:,jj-1)*Phi_step';
        end


        % Check Station Switchover
        if jj > 2
            if station(jj) ~= station(jj+1) || station(jj) ~= station(jj-1) || station(j
                StSw = [StSw;jj,1];
                factor = .001;
                P(:,:,jj)=P(:,:,jj).*factor;
            end
        end



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

        % Put into FindG
        %-------------------------------------------------------
        rhostar   = findrhostar(x(jj),y(jj),z(jj),Xsite(jj),Ysite(jj),Zsite(jj),theta(jj
        rhodotstar= findrhodotstar(x(jj),y(jj),z(jj),xdot(jj),ydot(jj),zdot(jj),Xsite(jj
        %-------------------------------------------------------

        % Find Observation Deviations
        %-------------------------------------------------------
        ystar       = [rhostar;rhodotstar];
        y1(:,jj)    = [rho_obs(jj);rhodot_obs(jj)] - ystar;
        %-------------------------------------------------------


        % Kalman Gain
        %-------------------------------------------------------
        K1          = P(:,:,jj)*Htilde'*inv(Htilde*P(:,:,jj)*Htilde' + R);
        %-------------------------------------------------------

        % Measurement Update
        %-------------------------------------------------------
        if time(jj) < cutoff_time
            xhat(:,jj) = zeros(18,1);% xhat(:,:,jj) =  xhat(:,:,jj); %xbar(:,:,jj) + K1*
%             xhat(:,jj) = xbar(:,jj) + K1*(y1(:,jj) - Htilde*xbar(:,jj));
        else
%             if ii < num_iterations
%                 % Update Best Guess of Initial Conditions
%                 %-----------------------------------------------
%                 Xstar0  = [Xstar0(1:18) + inv(Phi_map)*xhat(:,jj-1);(reshape(Phi_map,1
%                 P       = inv(Phi_map)*P(:,:,end)*inv(Phi_map)';
%                 xbar    = xbar(:,end-1) - xhat(:,jj-1);
%                 %-----------------------------------------------
```

```
%                 break
%             end
            if max(StSw(:,1) == jj)==1
                P(:,:,jj)=P(:,:,jj)./factor;
            end
            xhat(:,jj) = K1*y1(:,jj);
        end
%         xhat(:,:,jj) = xbar(:,:,jj) + K1*(y1(:,jj) - Htilde*xbar(:,:,jj));
        P(:,:,jj) = (eye(size(K1*Htilde)) - K1*Htilde)*P(:,:,jj)*(eye(size(K1*Htilde))-K
%         P(:,:,jj) = (eye(size(K1*Htilde)) - K1*Htilde)*P(:,:,jj);
%         P(:,:,jj) = ComputeP(Htilde,P(:,:,jj),R,Xstar,ystar,'joseph');
        %-----------------------------------------------

        % Update Nominal Trajectory
        %-----------------------------------------------
        % Update StatePhi? Or make a State PhiBar??
        StatePhi(:,jj)  = [Xstar + xhat(:,jj); reshape(eye(size(Phi(:,:,jj))),length(Phi
        %-----------------------------------------------


        tr = [tr,trace(P(1:3,1:3,jj))];


    end   % End observation loop

    % Get rid of 0,0 element in station switch array
    StSw(1,:) = [];


    %% Compute RMS of Residules
    fprintf('\n\nRMS of rho is :  %3.5f \n',rms(y1(1,:)))
    fprintf('RMS of rhodot is :  %3.5f \n',rms(y1(2,:)))
    rho_comp = y1(1,:); rhodot_comp=y1(2,:);
    rho_comp(rho_comp<mean(rho_comp)-2*std(rho_comp) | rho_comp>mean(rho_comp)+2*std(rho
    rhodot_comp(rhodot_comp<mean(rhodot_comp)-2*std(rhodot_comp) | rhodot_comp>mean(rhod
    fprintf('\n\nWithout Outliers RMS of rho is :  %3.5f \n',rms(rho_comp))
    fprintf('Without Outliers RMS of rhodot is :  %3.5f \n',rms(rhodot_comp))


    y1_sw = y1(1,StSw(:,1))';
    y2_sw = y1(2,StSw(:,1))';

    for kk=1:length(StSw(:,1))
        if StSw(kk,1)> length(rho_comp)
            StSw(kk,:) = [0,0];
        end
    end
    StSw(StSw==0)=[];

    rho_sw = rho_comp(1,StSw(:,1))';
    rhodot_sw = rhodot_comp(1,StSw(:,1))';

    %% Generate Plots
    figure(1)
    subplot(num_iterations,2,2*ii-1)
    plot(y1(1,:))
    hold on
%     plot(StSw(:,1),y1_sw,'r.','MarkerSize',20)
```

```
    ylabel('$\rho$ residules')
    xlabel('Observation Number')
%     legend('Residules','Station Switchover')

    subplot(num_iterations,2,2*ii)
    plot(y1(2,:))
    hold on
%     plot(StSw(:,1),y2_sw,'r.','MarkerSize',20)
    ylabel('$\dot{\rho}$ residules')
    xlabel('Observation Number')
%     legend('Residules','Station Switchover')


    figure(2)
    subplot(num_iterations,1,ii)
    semilogy(tr)
    xlabel('Observation Number')
    ylabel('Trace( $P_{xyz}$ )')

    figure(3)
    subplot(num_iterations,2,2*ii-1)
    plot(rho_comp(1,:))
    hold on
    plot(StSw(:,1),rho_sw,'r.','MarkerSize',20)
    ylabel('$\rho$ residules')
    xlabel('Observation Number')
    legend('Residules','Station Switchover')

    subplot(num_iterations,2,2*ii)
    plot(rhodot_comp)
    hold on
    plot(StSw(:,1),rhodot_sw,'r.','MarkerSize',20)
    ylabel('$\dot{\rho}$ residules')
    xlabel('Observation Number')
    legend('Residules','Station Switchover')

 end

 %% Generate Observation Switchover Plot
 figure
 plot(time(time<cutoff_time)/60,rho_obs(time<cutoff_time),'*r')
 hold on
 plot(time(time>cutoff_time)/60,rho_obs(time>cutoff_time),'.b')
 xlabel('time');ylabel('$\rho_{obs}$')
 legend('CKF','EKF Switch')

fprintf('\n\nRunning Time for Kalman Filter : %3.5f\n\n',toc)
```

A Derivation

```
function [A Htilde] = FindA_Htilde()

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%
% Zach Dischner-10/21/2012
%
% FindA_Htilde
```

```
%
% Purpose: Find A and Htilde matrices
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




% Use Jacobian

%% Define the State
syms r x y z xdot ydot zdot uE J2 Cd Xsite1  Ysite1  Zsite1 Xsite2 Ysite2 Zsite2
Xsite3 Ysite3 Zsite3 theta theta_dot
syms R_E r Area m rho_a Va va

X       = [x ; y ; z ; xdot ; ydot ; zdot ; uE ; J2 ; Cd ; Xsite1 ; Ysite1 ; Zsite1; Xsi

%% Define F vector

% [xdot ydot zdot] due to gravity and J2
r       = sqrt(x^2+y^2+z^2);
F_U     = [...
              -uE/r^3*x*(1-3/2*J2*(R_E/r)^2*(5*(z/r)^2-1));
              -uE/r^3*y*(1-3/2*J2*(R_E/r)^2*(5*(z/r)^2-1));
              -uE/r^3*z*(1-3/2*J2*(R_E/r)^2*(5*(z/r)^2-3));
          ];

% [xdot ydot zdot] due to atmospheric drag
Va      = [
    xdot + theta_dot*y;
    ydot - theta_dot*x;
    zdot
    ];
va      = sqrt((xdot + theta_dot*y)^2 + (ydot - theta_dot*x)^2 + zdot^2);

syms rho0 r0 H
rho_a   = rho0.*exp(-(r - r0)./H);

F_Drag = -0.5 .* Cd .* (Area./m) .* rho_a .* va .* Va;

% Assemble
F_a = F_U + F_Drag; %

% X' = F*X
F =[xdot ; ydot ; zdot ; F_a ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0];


%% Find A Matrix
A = jacobian(F,X);
% Do this outside of the function?
A=simplify(A);
[rows,cols] = size(A);
for ii = 1:length(rows)
    for jj = 1:length(cols)
        A(ii,jj) = simplify(A(ii,jj));
```

```
    end
end

matlabFunction(A,'file','A_18x18.m')



%% Find Htilde Matrix (for a generic station position)
syms Xsite Ysite Zsite
syms theta  % = theta_dot * time
% dcm = [cos -sin 0;sin cos 0; 0 0 1];
% rho = sqrt((x-Xsite*cos(theta) - Ysite*sin(theta))^2 + (y-Ysite*cos(theta) + Xsite*sin
% The Answer
rho = sqrt(x^2+y^2+z^2+Xsite^2+Ysite^2+Zsite^2-2*(x*Xsite+y*Ysite)*cos(theta)+2*(x*Ysite
% rho = @(x,y,z,Xsite,Ysite,Zsite,theta) sqrt(x^2+y^2+z^2+Xsite^2+Ysite^2+Zsite^2+2*(x*X

rhodot = (x*xdot + y*ydot + z*zdot - (xdot*Xsite + ydot*Ysite)*cos(theta) + theta_dot*(x
            +(xdot*Ysite - ydot*Xsite)*sin(theta) + theta_dot*(x*Ysite - y*Xsite)*cos(th
                                                                /rho;


obs = [rho;rhodot];

% For just single station
Single_Station_X       = [x ; y ; z ; xdot ; ydot ; zdot ; uE ; J2 ; Cd ; Xsite ; Ysite

% Add padding depending on the station.
% If station 1, add 6 columns of padding
% If station 2, add 3 columns, move 3 endmost columns after that, then add
% 3 columns
% IF station 3, put 6 columns of zeros between the 9th and (going to 15th) columms.

Htilde = jacobian(obs,Single_Station_X);
[rows,cols] = size(Htilde);
for ii = 1:length(rows)
    for jj = 1:length(cols)
        Htilde(ii,jj) = simplify(Htilde(ii,jj));
    end
end



matlabFunction(Htilde,'file','Htilde_2x12.m')



end
```