The Comparison of Extended and Unscented Kalman Filters for 6U CubeSat Low Earth Orbit Determination

Sean L. Bohne
Purdue University, West Lafayette, IN 47907, USA

Orbit determination is an ongoing field of study that is crucial in satellite tracking and space traffic management. Many estimation filtering techniques have been developed to accurately track man-made satellites and small objects that could pose significant risks to space operations. This study investigates the performance of the Extended Kalman Filter and Unscented Kalman using angles-only and radar measurements. Estimation error convergence, filter consistency, and runtime are evaluated to compare the performance of the EKF and UKF. Model mismatch is also considered by developing a high fidelity model that accounts for atmospheric drag and solar radiation pressure perturbations. The findings show that, although the UKF performs better than the EKF in terms of estimation error, the EKF is much less computationally expensive for systems with limited computational power. In space applications, computational power is often very limited, and thus, the EKF proves a viable option balancing the tradeoff between speed and accuracy.

Contents

Nomenclature

T	=	Rotation Matrix
ω_E	=	Earth Angular Velocity
Δt	=	Time Step
a	=	Semi-major Axis
e	=	Eccentricity
i	=	Inclination
Ω	=	Longitude of the Ascending Node
ω	=	Argument of Perigee
ν	=	True Anomaly
r	=	Satellite Radial Position in Orbit Plane
$\mathbf{r}$	=	Position Vector
h	=	Specific Angular Momentum
$\mathbf{f}_{Drag}$	=	Atmospheric Drag Force
$\mathbf{f}_{SRP}$	=	Solar Radiation Pressure Force
$\mathbf{u}$	=	Unit Vector
ρ	=	Range Measurement
$\dot{\rho}$	=	Range-rate Measurement
El	=	Elevation Angle Measurement
Az	=	Azimuth Angle Measurement
(X, Y, Z)	=	Position in ECI
(x, y, z)	=	Position in ECF
σ	=	Standard Deviation
v	=	Measurement Noise
$\hat{(*)}$	=	Estimate
P	=	Covariance Matrix
$\mathbf{f}(*)$	=	Nonlinear Dynamics Function
F	=	Dynamics Jacobian
W	=	Innovations Covariance Matrix
C	=	Cross Covariance Matrix
K	=	Kalman Gain Matrix
R	=	Measurement Noise Covariance

$\mathbf{z}$	=	Measurement
$\boldsymbol{\mathcal{X}}$	=	State Sigma Points
$\mathbf{m}$	=	Mean
$\boldsymbol{\mathcal{Z}}$	=	Measurement Sigma Points
$\mathrm{P}_{xz,k}$	=	Cross Covariance Matrix
$\mathrm{P}_{z,k}$	=	Innovations Covariance Matrix
diag $(*)$	=	Diagonal Matrix
Subscripts		
0	=	Initial
k	=	Iteration Index
sat	=	Satellite
rel	=	Relative
pf	=	Perifocal
t	=	Topocentric
Subscripts		
T	=	Transpose
$-$	=	*a priori*
$+$	=	*a posteriori*

I. Introduction

Orbit determination is a critical area of study for many applications in space flight ranging from space debris management to satellite tracking and traffic management. Various filtering algorithms have been developed in order to obtain accurate estimates of satellite states. A study conducted in 2012 compared the performance of the EKF, gaussian mixture model (GMM), and particle filter (PF) for simple two-body gravity dynamics [1]. It was found that the particle filter performed the best with the lowest estimation error and covariance. However, the study only considered a simplified dynamics model and did not consider perturbation forces such as drag and solar radiation pressure. This study seeks to extend the work done in [1] by analyzing the effects of model mismatch between the continuous-discrete EKF and UKF when considering atmospheric drag and solar radiation pressure perturbations.

II. Problem Formulation

This section outlines the coordinate transformations used to simulate the low and high fidelity dynamics. To accurately model the system dynamics and measurements, a set of coordinate frames and transformations must first be established.

A. 3-1-3 Euler Angle Rotation

Any coordinate frame rotation can be represented as a sequence of three rotations. This study will use the 3-1-3 Euler angle representation, although any sequence is valid. This rotation can be described as the following:

$$T(\phi,\theta,\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

where ψ, θ, and ϕ are Euler angles representing a 3-1-3 rotation sequence.

B. Earth Centered Inertial Frame

The Earth-Centered Inertial (ECI) frame, denoted by (X, Y, Z), is a non-rotating inertial coordinate system centered at the origin of the Earth where the x-axis is aligned the vernal equinox, the z-axis is aligned with Earth's North Pole, and the y-axis completes the right-handed system [2]. The satellite dynamics are simulated in ECI and can be transformed to alternate frames such as the Earth Centered, Earth Fixed (ECEF) system when necessary.

C. Earth Centered Earth Fixed Frame

The ECEF frame, denoted by (x, y, z), is a coordinate system whose axes rotate along with the Earth [2]. In this frame, a point fixed on Earth keeps the same coordinates over time because the coordinate system turns with the

planet's rotation. A particularly important coordinate transformation is that between ECI and ECEF and the coordinate transformation matrix can be formulated as follows:

$$T_{ECI}^{ECEF} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

where θ is the amount Earth has rotated, in radians, after a given time interval and can be calculated as:

$$\theta = \omega_E * \Delta t \tag{3}$$

where $\omega_E = \frac{2\pi}{86164} = 7.292123 * 10^{-5} \frac{\text{rad}}{\text{sec}}$ is Earth's angular velocity and Δt is the time interval. Note that this model is a simplified version of Earth's rotation and does not account for disturbances including precession and nutation. Converting from ECEF to ECI can be determined using the following:

$$T_{ECEF}^{ECI} = \left(T_{ECI}^{ECEF}\right)^T \tag{4}$$

where the inverse transformation is simply the transpose because the columns of the rotation matrix are orthonormal.

D. Topocentric Frame

The topocentric frame, denoted by $(x, y, z)_t$, is a coordinate system attached to the surface of the Earth, with the x-axis pointing eastward, the y-axis pointing northward, and the z-axis pointing in the local vertical, which is described in the Coordinate Systems and Time section of [2]. The transformation from longitude and latitude coordinates can be calculated via:

$$T_t = \begin{bmatrix} -\sin\lambda & \cos\lambda & 0 \\ -\sin\phi\cos\lambda & -\sin\phi\sin\lambda & \cos\phi \\ \cos\phi\cos\lambda & \cos\phi\sin\lambda & \sin\phi \end{bmatrix} \tag{5}$$

where λ and ϕ are latitude and longitude coordinates on Earth's surface respectively. Following the topocentric transformation, a set of spherical coordinates can be determined using the following [2]:

$$\begin{aligned} \sin(\text{El}) &= \frac{z_t}{r_t} \\ \tan(Az) &= \frac{x_t}{y_t} \end{aligned} \tag{6}$$

El and Az denote the elevation, and azimuth respectively. Representing the topocentric coordinates in this form provides a method to model range, range-rate, and angles-only measurement models and will be discussed in further detail in Section III.A.

E. Keplerian Orbital Elements to ECI

This section will describe the transformation of keplerian orbital elements to ECI for trajectory simulation. Keplerian orbital elements use six parameters to fully describe a satellite's orbit under a two-body gravity model. These parameters are summarized in Table 2

Table 2 Classical Keplerian Orbital Elements

Element	Symbol	Description
Semi-major axis	a	Defines the size of the orbit; half the longest axis of the orbital ellipse.
Eccentricity	e	Describes how circular or elliptical the orbit is.
Inclination	i	Angle between the orbital and equatorial plane.
Longitude of the Ascending Node	Ω	Angle from the vernal equinox to the ascending node, measured in the equatorial plane.
Argument of Perigee	ω	Angle between the ascending node and perigee measured in the orbital plane.
True Anomaly	ν	Angle between perigee and the satellite's current position.

In order to convert keplerian orbital elements to ECI, the radial position and velocity must determined in the perifocal frame, which is centered at the focus of the orbit with the x-axis pointing toward periapsis, the z-axis pointing toward the angular momentum vector, and the y-axis completing the right-handed system. The satellite's radial position in the perifocal frame is given by the following:

$$r = \frac{a\left(1 - e^2\right)}{1 + e\cos\nu} \tag{7}$$

When vectorized, the position vector can be described as:

$$\mathbf{r}_{pf} = \begin{bmatrix} r\cos\nu \\ r\sin\nu \\ 0 \end{bmatrix} \tag{8}$$

The satellite's specific angular momentum is given by the follow expression:

$$h = \sqrt{\mu a\left(1 - e^2\right)} \tag{9}$$

where $\mu_E = 3.986 * 10^5 \text{km}^3\text{s}^{-2}$ is Earth's gravitational parameter. Using this result, one can obtain the perifocal radial and angular velocities:

$$\dot{r} = \frac{he \sin \nu}{a\left(1 - e^2\right)} \tag{10}$$

$$\dot{\theta} = \frac{h}{r} \tag{11}$$

Vectorizing the satellite's velocity in the perifocal frame gives the following result:

$$\dot{\mathbf{r}}_{pf} = \begin{bmatrix} \dot{r}\cos\nu - \dot{\theta}\sin\nu \\ \dot{r}\sin\nu + \dot{\theta}\cos\nu \\ 0 \end{bmatrix} \tag{12}$$

Applying Equation 1, where $\phi = \omega$, $\theta = i$, and $\psi = \Omega$, converts the position and velocity from the perifocal frame to ECI and is described via the following:

$$\mathbf{r}_{ECI} = T\left(\omega, i, \Omega\right)\mathbf{r}_{pf} \tag{13}$$

$$\dot{\mathbf{r}}_{ECI} = T\left(\omega, i, \Omega\right)\mathbf{v}_{pf} \tag{14}$$

The initial ECI state vector based on the given keplerian orbital elements is a combination of the ECI position and velocity vectors and is written as:

$$\mathbf{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} & z & \dot{z} \end{bmatrix}^T \tag{15}$$

where (x, y, z) and $(\dot{x}, \dot{y}, \dot{z})$ are the satellite's initial position and velocity in ECI respectively.

F. Dynamics Model

This section discusses the dynamics model used to simulate truth data.

In a simple two-body orbit, the only force acting on a satellite is gravity, which constantly pulls the satellite inward toward the mass it is orbiting. Using this simple model, the differential equation for the satellite motion is as follows [2]:

$$\ddot{\mathbf{r}} = -\frac{\mu\mathbf{r}}{r^3} \tag{16}$$

where $\mathbf{r}$ represents the satellite position vector in ECI. This study will include atmospheric drag and solar radiation pressure perturbation forces and are summarized under the Nongravitational Perturbations section in [2]. Atmospheric drag, as described by [2] is modeled as:

$$\mathbf{f}_{Drag} = -\frac{1}{2}\rho\left(\frac{C_D A}{m}\right)\dot{r}_r\dot{\mathbf{r}}_r \tag{17}$$

where C_D is the drag coefficient, A is the frontal area, m is mass, V_r is the magnitude of velocity relative to the atmosphere, $\dot{\mathbf{r}}_r$ is the corresponding velocity vector, and ρ is density. With values taken from [3], the physical parameters for the 6U CubeSat simulation are as follows: $A = 0.085\text{m}^2$, $m = 10\text{kg}$, and $C_d = 2.1$. Tapley et. al [2] indicates that density can be described by the simple exponential density model $\rho = \rho_0 e^{-\beta(h-h_0)}$ where ρ_0 and h_0 are the reference density and altitude taken at sea level, $\beta = 7.8\text{km}^{-1}$ is a scaling factor [4], and h is the distance from Earth's surface to the satellite. The relative velocity vector must account for Earth's rotation and is determined by the following:

$$\dot{\mathbf{r}}_{rel} = \dot{\mathbf{r}}_{sat} - \boldsymbol{\omega}_E \times \mathbf{r}_{sat} \tag{18}$$

where $\dot{\mathbf{r}}_{sat}$ is the satellite velocity in ECI and $\mathbf{r}_{sat}$ is the satellite position in ECI. Additionally, solar pressure radiation is described by the following equation [2]:

$$\mathbf{f}_{SRP} = -P\frac{\nu A}{m}C_R\mathbf{u} \tag{19}$$

P is the momentum flux from the sun, with a value of approximately 4.56×10^{-6} N/m^2, ν is the eclipse factor with $\nu = 0$ if the satellite is in Earth's shadow and $\nu = 1$ when exposed to the sun, C_R is the reflectivity coefficient and is approximately one, and $\mathbf{u}$ is the unit vector pointing toward the sun [2]. For simplicity, this study assumes a constant value of $\nu = 1$ and a simplified unit vector, $\mathbf{u}$, which takes the form:

$$\mathbf{u} = [\cos\theta \quad \sin\theta \quad 0]^T \tag{20}$$

where $\theta = \omega_{sun}\Delta t$, with $\omega_{Sun} = \frac{2\pi}{365.25*24*3600}$ rad/s and Δt representing the elapsed time. This simplified unit vector approximates the satellite to sun vector as equivalent to the Earth to sun vector, which is a reasonable assumption given the large distance between the bodies.

By summing the forces, the force model used for the high fidelity dynamics simulation is obtained and summarized in Equation 21.

$$\ddot{\mathbf{r}} = -\frac{\mu\mathbf{r}}{r^3} + \mathbf{f}_{Drag} + \mathbf{f}_{SRP} \tag{21}$$

This model will be used to simulate truth data and to analyze the effects of model mismatch.

III. Methodology

The following section outlines the measurement model used to simulate measurement data and the low and high fidelity models to simulate the satellite dynamics.

A. Measurement Models

1. Range and Range-Rate Model

Range and range-rate measurements are gathered through radar instruments and can provide information on a satellite's relative radial distance and velocity to the instrument. The range and range-rate models given in this section are derived under Observations in [2]. From simulation, a satellite's position is determined in ECI. However, the satellite's position must be converted to ECEF using Equation 2 to determine the relative distance and velocity between the satellite and the ground station. Once the satellite's position is represented in ECEF, the following equation can be used to determine the range measurements [2]:

$$\rho = \sqrt{(x - x_I)^2 + (y - y_I)^2 + (z - z_I)^2} + v_\rho \tag{22}$$

where ρ is the range measurement and can be thought of as the relative position between the ground station and satellite, (x, y, z) is the satellite position in ECEF, (x_I, y_I, z_I) is the instrument position in ECEF, and $v_\rho \sim \mathcal{N}\left(0, \sigma_\rho^2\right)$ is zero-mean Gaussian white-noise with covariance σ_ρ^2 associated with the measurement. Similarly, range-rate measurements can be calculated by converting the radar instrument's position from ECEF to ECI using Equation 4 and determining its velocity in ECI via the following:

$$\dot{\mathbf{r}}_{gs} = \boldsymbol{\omega}_E \times \mathbf{r}_{gs} \tag{23}$$

where $\mathbf{r}_{gs}$ is the ground station position in ECI and $\boldsymbol{\omega}_E$ is Earth's angular velocity vector pointing in the Z direction. Using the following equation, one can obtain the range-rate measurement [2]:

$$\dot{\rho} = \frac{\left((X - X_I)\left(\dot{X} - \dot{X}_I\right) + (Y - Y_I)\left(\dot{Y} - \dot{Y}_I\right) + (Z - Z_I)\left(\dot{Z} - \dot{Z}_I\right)\right)}{\rho} + v_{\dot{\rho}} \tag{24}$$

where (X, Y, Z) and $\left(\dot{X}, \dot{Y}, \dot{Z}\right)$ are the satellite position and velocity in ECI respectively, $(X, Y, Z)_I$ and $\left(\dot{X}, \dot{Y}, \dot{Z}\right)_I$ are the radar instrument position and velocity in ECI respectively, and $v_{\dot{\rho}} \sim \mathcal{N}\left(0, \sigma_{\dot{\rho}}^2\right)$ is zero-mean Gaussian white-noise with covariance σ_ρ^2 associated with the measurement.

2. Angles-Only

Angles-only measurements can be gathered optically via telescope and provide the angular position of a satellite and the process is also documented by [2]. Given a set of range measurements given by Equation 22, angles-only measurements can be calculated by converting range measurements to the topocentric frame using Equation 5 and is shown by the following:

$$\mathbf{r}_t = T_t \boldsymbol{\rho} \tag{25}$$

where $\mathbf{r}_t$, also denoted as (x_t, y_t, z_t), is the range measurement vector in the topocentric frame and $\boldsymbol{\rho}$ is the range measurement vector. Using Equation 6, the elevation and azimuth angles can be calculated and the corresponding measurement noises can be added to account for uncertainties:

$$\begin{aligned} \text{El} &= \text{El} + v_{\text{El}} \\ \text{Az} &= \text{Az} + v_{\text{Az}} \end{aligned} \tag{26}$$

where $v_{\text{El}} \sim \mathcal{N}\left(0, \sigma_{\text{El}}^2\right)$ and $v_{\text{Az}} \sim \mathcal{N}\left(0, \sigma_{\text{Az}}^2\right)$ are zero-mean Gaussian white-noise corresponding to the elevation and azimuth measurements.

3. Measurement Limits

In real systems, measurements may not available due to various reasons including cloud coverage and time periods when the satellite is not within the range of the measurement device. In simulation, this was done by setting elevation limits where $0 < \text{El} < 90°$. There were issues during implementation with covariance matrices becoming singular and all entries yielding "NaN" results which could not be resolved. However, the code found in the Appendix contains an outline for further investigation and can be found in the commented code inside the "Meas_Func" function. This study, therefore, assumes continuous measurement availability. Additionally, the ground station is located at longitude 0° and latitude 0°

B. Continuous-Discrete EKF Algorithm

This section will outline the continuous-discrete EKF algorithm used in this study. The state vector consists of six states including satellite position and velocity as described in Equation 15. The algorithm consists of three steps including: propagation, gain calculation, and update. The filter is initialized by generating a random sample from the initial probability density which is used to both propagate the state deterministically through the model described in Equation 21 and initialize the EKF filter. Additionally, the process noise matrix, Q, is scaled by the measurement time

step, Δt, to integrate noise over the time interval.

1. Propagation

Using the low fidelity model described in Equation 16, the propagation step deterministically maps the previous state and covariance forward in time to obtain the *a priori* estimates. In addition, small process noises are added to the state and covariance after each propagation step to account for model mismatch. This step is described mathematically as follows:

$$\dot{\hat{x}}\left(t\right) = \mathbf{f}\left(\hat{\mathbf{x}}\left(t\right), t\right) \tag{27}$$

$$\dot{P}\left(t\right) = F\left(\hat{\mathbf{x}}, t\right) P\left(t\right) + P\left(t\right) F^{T}\left(\hat{\mathbf{x}}\left(t\right), t\right) + G Q_{s}\left(t\right) G^{T} \tag{28}$$

$\mathbf{f}\left(\hat{\mathbf{x}}\left(t\right), t\right)$ is the nonlinear dynamics function mapping the previous state estimate, $F\left(\hat{\mathbf{x}}\left(t\right), t\right)$ is the Jacobian, which was found analytically, of the nonlinear dynamics evaluated at the propagated state estimate, $P\left(t\right)$ is the covariance matrix, G is a matrix mapping the process noise to the state, and Q_s is the power spectral density.

2. Gain Calculation

Following the propagation step, the EKF determines the optimal kalman gain based on the innovations and cross covariance gains for the *linearized* process at each time step k. The calculations are summarized below:

$$W_k = H_k P_k^{-} H_k^{T} + R_k \tag{29}$$

$$C_k = P_k^{-} H_k^{T} \tag{30}$$

$$K_k = C_k W_k^{-1} \tag{31}$$

W_k is the innovations gain, H_k is the linearized measurement model, determined using central differencing for angles-only measurements and analytically for range and range-rate measurements, R_k is the measurement noise covariance, C_k is the cross covariance gain, and K_k is the Kalman gain. The minus superscripts denotes *a priori* estimates.

3. Update

The update step uses the gains calculated in the previous step to update the state and covariance estimates based on the provided measurement to obtain the *a posteriori* estimates. This step is summarized mathematically as follows:

$$\hat{\mathbf{z}}_k = H_k \hat{\mathbf{x}}_k^- \tag{32}$$

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + K_k \left(\mathbf{z}_k - \hat{\mathbf{z}}_k\right) \tag{33}$$

$$P_k^+ = P_k^- - C_k K_k^T - K_k C_k^T + K_k W_k K_k^T \tag{34}$$

$\hat{\mathbf{z}}_k$ is the predicted measurement based on the *a priori* state estimate and $\mathbf{z}_k$ is the measurement received by the ground station. The plus superscripts denote *a posteriori* estimates. After updating the state estimate, the EKF is reinitialized with the *a posteriori* estimates and continues to the next time step.

C. UKF Algorithm

The UKF, similarly to the continuous-discrete EKF, approximates the nonlinearities in a system's dynamics and measurements. However, unlike the EKF which uses linearization about a point to obtain the update gains, the UKF uses the unscented or scaled unscented transform. This study will use the scaled unscented transform. The three main steps remain the same as the EKF and include: propagation, gain calculation, and update. Additionally, the filter is initialized in the same way as the EKF as described in section III.B. Similarly to the EKF, the process noise covariance, Q, is scaled by the measurement time step, Δt, to integrate noise over the time interval.

1. Propagation

The UKF begins by forming sigma points, which are selected based on the *a posteriori* mean and covariance at the previous time step. The sigma points are calculated as follows:

$$\begin{aligned}
\boldsymbol{\mathcal{X}}_{k-1}^{(0)} &= \mathbf{m}_{k-1}^+ \\
\boldsymbol{\mathcal{X}}_{k-1}^{(i)} &= \mathbf{m}_{k-1}^+ + \sqrt{n+\lambda}\left[\sqrt{P_{k-1}^+}\right]_i \quad i = 1, ..., n \\
\boldsymbol{\mathcal{X}}_{k-1}^{(i+n)} &= \mathbf{m}_{k-1}^+ - \sqrt{n+\lambda}\left[\sqrt{P_{k-1}^+}\right]_i \quad i = 1, ..., n
\end{aligned} \tag{35}$$

where $\boldsymbol{\mathcal{X}}^{(0)}$ is the central sigma point, n is the state dimension, $\mathbf{m}_{k-1}^+$ is the *a posteriori* state mean at the previous time step, P_{k-1}^+ is the *a posteriori* covariance at the previous time step, n is the state dimension, and $\lambda = \alpha^2 (n + \kappa) - n$.

Here, α is a scaling term between 0 and 1, inclusive, which determines the spread of the the sigma points around the central sigma point. Generally, $\kappa \geq 0$ is chosen to guarantee a positive semi-definite covariance matrix, although the actual value does not impact performance too much and is commonly chosen to be $\kappa = 0$. For this study, the weighting parameters were chosen as $\alpha = 0.001$ and $\kappa = 0$ Next, the mean and covariance weights are computed as follows:

$$\begin{aligned} w_0^{(m)} &= \frac{\lambda}{n+\lambda} \\ w_0^{(c)} &= \frac{\lambda}{n+\lambda} + \left(1 - \alpha^2 + \beta\right) \\ w_i^{(m)} &= \frac{1}{2\left(n+\lambda\right)} \qquad\qquad i = 1, ..., 2n \\ w_i^{(c)} &= \frac{1}{2\left(n+\lambda\right)} \qquad\qquad i = 1, ..., 2n \end{aligned} \tag{36}$$

β is another scaling term that accounts higher order moments such as kurtosis. For a Gaussian prior distribution, $\beta = 2$ is optimal. Therefore, this study uses $\beta = 2$ because the uncertainty is assumed to be Gaussian. Once the sigma points are formed and the weights are calculated, the sigma points are transformed through nonlinear dynamics model:

$$\boldsymbol{\mathcal{X}}_k^{(i)} = \mathbf{f}\left(\boldsymbol{\mathcal{X}}_{k-1}^{(i)}\right), \quad i = 0, ..., 2n \tag{37}$$

Using the transformed sigma points, the predicted *a priori* mean $\mathbf{m}_k^-$ and covariance P_k^- can be calculated:

$$\begin{aligned} \mathbf{m}_k^- &= \sum_{i=0}^{2n} w_i^{(m)} \boldsymbol{\mathcal{X}}_k^{(i)} \\ P_k^- &= \sum_{i=0}^{2n} w_i^{(c)} \left(\boldsymbol{\mathcal{X}}_k^{(i)} - \mathbf{m}_k^-\right)\left(\boldsymbol{\mathcal{X}}_k^{(i)} - \mathbf{m}_k^-\right)^T + P_{w,k-1} \end{aligned} \tag{38}$$

where $P_{w,k-1}$, similar to the EKF, is the process noise covariance.

2. Gain Calculation

Similarly to the propagation step, the UKF uses the *a priori* mean and covariance to form a new set of sigma points and weights:

$$\begin{aligned} \boldsymbol{\mathcal{X}}_{k-1}^{-(0)} &= \mathbf{m}_k^- \\ \boldsymbol{\mathcal{X}}_k^{-(i)} &= \mathbf{m}_k^- + \sqrt{n+\lambda}\left[\sqrt{P_k^-}\right]_i \quad i = 1, ..., n \\ \boldsymbol{\mathcal{X}}_k^{-(i+n)} &= \mathbf{m}_k^- - \sqrt{n+\lambda}\left[\sqrt{P_k^-}\right]_i \quad i = 1, ..., n \end{aligned} \tag{39}$$

The mean and covariance weights are calculated exactly as they were in the propagation step:

$$\begin{aligned} w_0^{(m)} &= \frac{\lambda}{n+\lambda} \\ w_0^{(c)} &= \frac{\lambda}{n+\lambda} + \left(1 - \alpha^2 + \beta\right) \\ w_i^{(m)} &= \frac{1}{2\left(n+\lambda\right)} \qquad\qquad i = 1, ..., 2n \\ w_i^{(c)} &= \frac{1}{2\left(n+\lambda\right)} \qquad\qquad i = 1, ..., 2n \end{aligned} \tag{40}$$

The new sigma points are transformed through the measurement mode:

$$\boldsymbol{\mathcal{Z}}_k^{(i)} = \mathbf{h}\left(\boldsymbol{\mathcal{X}}_k^{-(i)}\right), \quad i = 0, ..., 2n \tag{41}$$

Using the transformed sigma points, the predicted measurement $\hat{\mathbf{z}}_k$, covariance, $P_{z,k}$, and cross covariance, $P_{xz,k}$ can be calculated:

$$\begin{aligned} \hat{\mathbf{z}}_k &= \sum_{i=0}^{2n} w_i^{(m)} \boldsymbol{\mathcal{Z}}_k^{(i)} \\ P_{z,k} &= \sum_{i=0}^{2n} w_i^{(c)} \left(\boldsymbol{\mathcal{Z}}_k^{(i)} - \mathbf{m}_k^-\right)\left(\boldsymbol{\mathcal{Z}}_k^{(i)} - \mathbf{m}_k^-\right)^T + R_k \\ P_{xz,k} &= \sum_{i=0}^{2n} w_i^{(c)} \left(\boldsymbol{\mathcal{X}}_k^{(i)} - \mathbf{m}_k^-\right)\left(\boldsymbol{\mathcal{Z}}_k^{(i)} - \mathbf{m}_k^-\right)^T + R_k \end{aligned} \tag{42}$$

The Kalman gain is computed in the same way as the EKF a shown below:

$$K_k = P_{xz,k} P_{z,k}^{-1} \tag{43}$$

3. Update

Using the Kalman gain, the *a posteriori* mean and covariance and can be obtained:

$$\begin{aligned} \mathbf{m}_k^+ &= \mathbf{m}_k^- + K_k\left(\mathbf{z}_k - \hat{\mathbf{z}}_k\right) \\ P_k^+ &= P_k^- - P_{xz,k} K_k^T - K_k P_{xz,k}^T + K_k P_{z,k} K_k^T \end{aligned} \tag{44}$$

Following the update step, the *a posteriori* estimates are reinitialized as the initial conditions for the next time step after the arrival of a new measurement.

D. Simulation Initial Conditions and Parameters

This section summarizes the noise parameters and initial conditions used in the simulation over 100 Monte Carlo trials and 5 orbital periods for each measurement model. Reasonable angles, range, and Doppler measurement noise parameters were taken from [5] under the section Satellite Tracking and Observation Models. However, the range-rate standard deviation caused the filters to diverge significantly and larger values were used to account for this behavior. The measurement frequency, Δt, was chosen to save computational time while being fast enough to resemble real measurement devices [6]. Initial states for the truth and filter simulations were sampled from the initial probability density. Earth's radius is assumed to be a uniform 6378 km.

- $a = r_E + 1000 = 7378$ km
- $e = 0.01$
- $i = \frac{45*\pi}{180}$ rad
- $\Omega = 0$ rad
- $\omega = 0$ rad
- $\nu = 0$ rad
- $\sigma_\rho = 15$ m
- $\sigma_{\dot{\rho}} = 0.4$ mm/sec
- $\sigma_{\text{El}} = \frac{50\pi}{180*3600}$ rad
- $\sigma_{\text{Az}} = \frac{50\pi}{180*3600}$ rad
- $\sigma_{v_x} = 1.5$ mm/sec
- $\sigma_{v_y} = 1$ mm/sec
- $\sigma_{v_z} = 1$ mm/sec
- $\Delta t = 10$ sec
- $P_0 = \text{diag}\left(\begin{bmatrix} 0.05 & 1*10^{-7} & 0.02 & 1*10^{-8} & 0.02 & 1*10^{-8} \end{bmatrix}\right)$

IV. Results and Discussion

This section will compare the EKF and UKF performance under three cases using angles-only measurements, range and range-rate measurements, and all four measurement techniques. Performance will be evaluated by filter convergence, consistency, and computational speed. Estimation errors for each filter and the corresponding 3σ bounds are plotted. Average squared normalized estimation error squared (ASNEES) plots are provided to demonstrate filter consistency. In addition, *a priori* and *a posteriori* estimates are plotted together in each figure.

A. Angles-Only Discussion

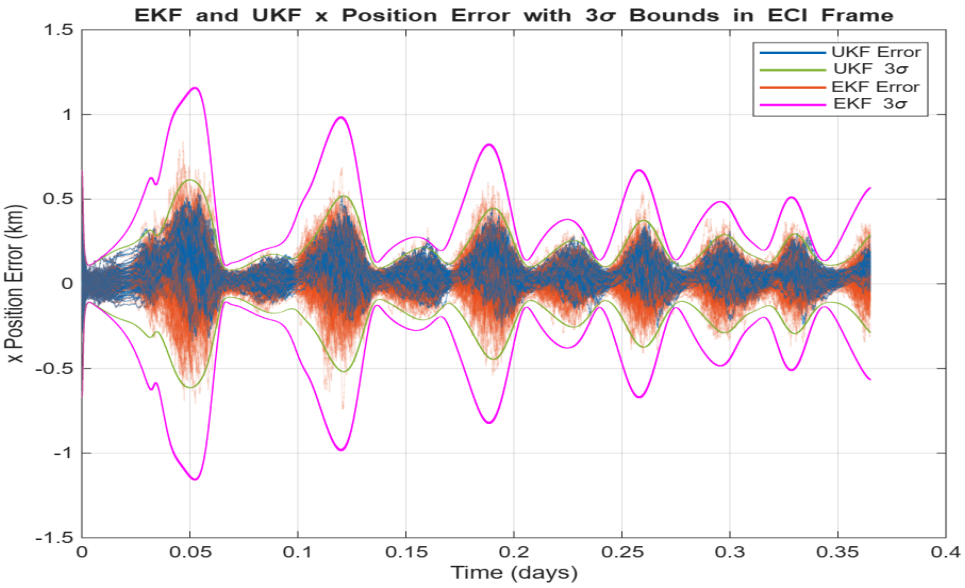

Fig. 1 EKF and UKF X position estimation error.

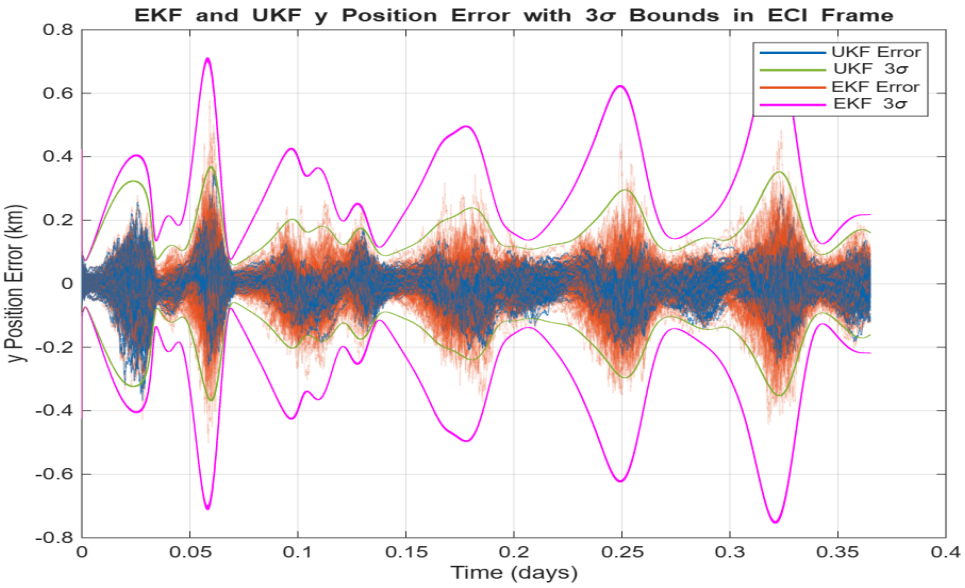

Fig. 2 EKF and UKF Y position estimation error.

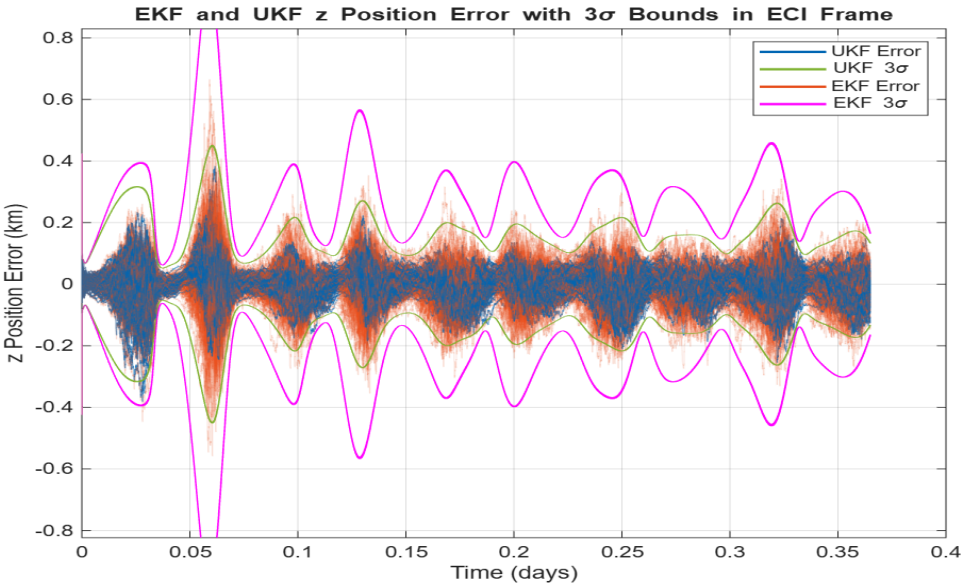

Fig. 3 EKF and UKF Z position estimation error.

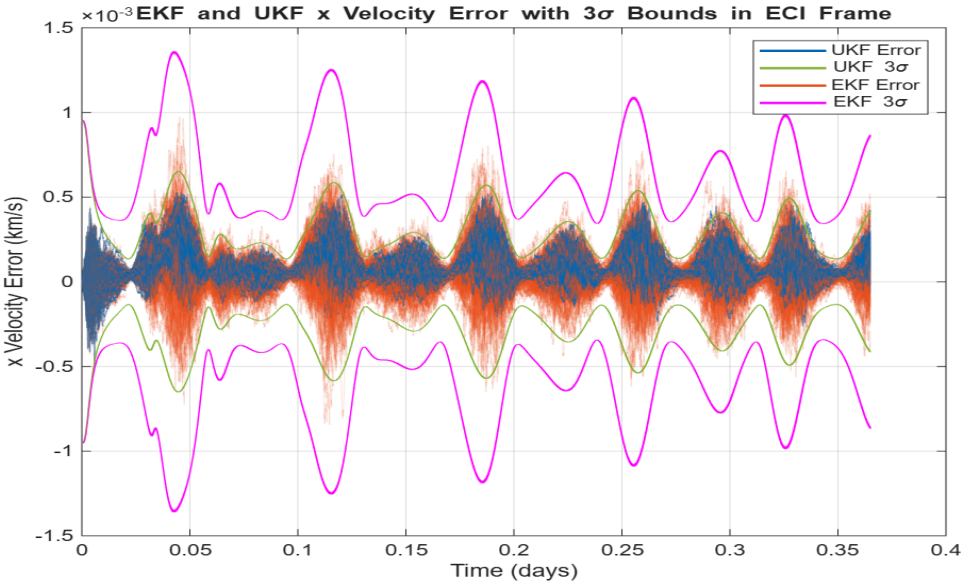

Fig. 4 EKF and UKF X velocity estimation error.

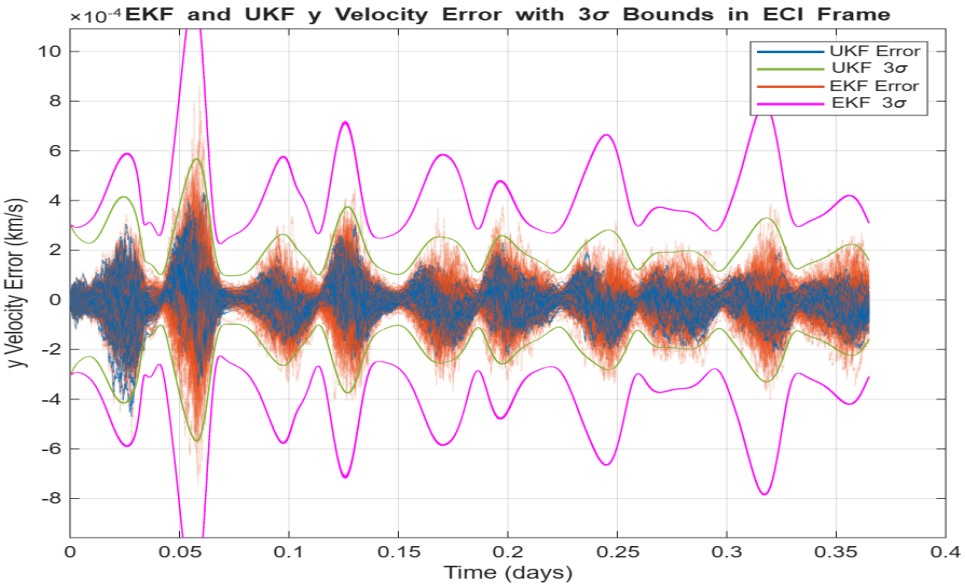

Fig. 5 EKF and UKF Y velocity estimation error.

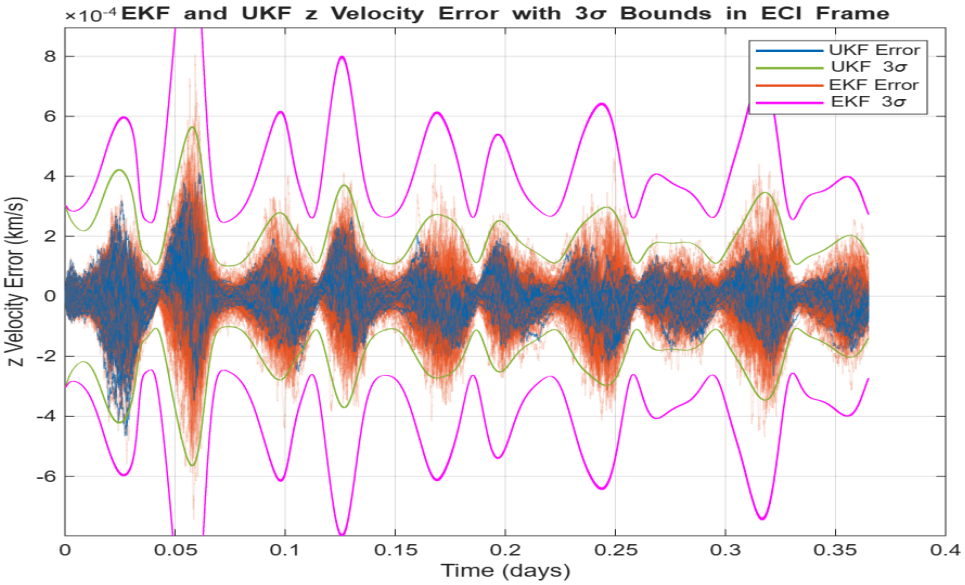

Fig. 6 EKF and UKF Z velocity estimation error.

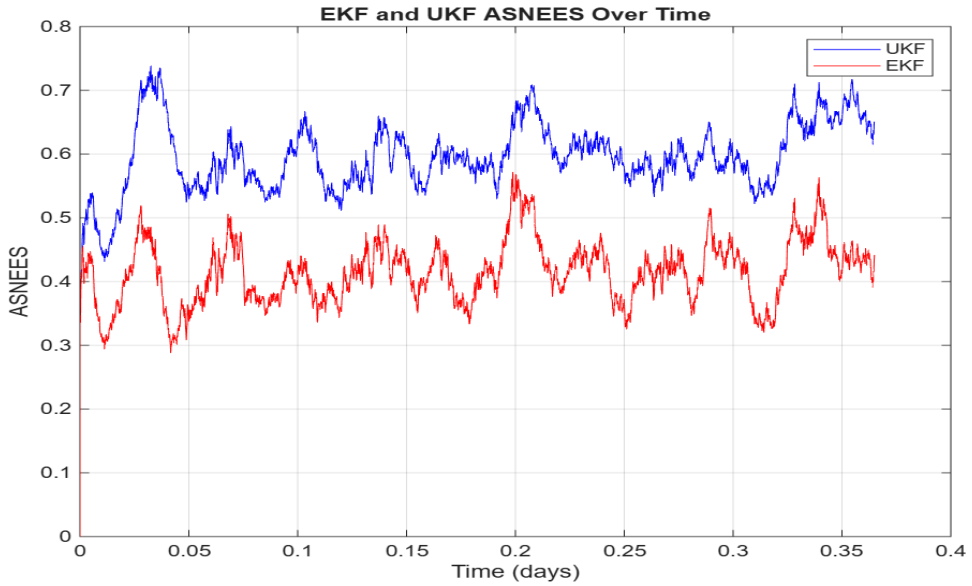

Fig. 7 ASNEES values over time. Values near one indicate a consistent filter.

Figures 1 - 6 show that both the EKF and UKF perform well in estimating the position and velocity errors and stay within the 3σ bounds for most of the simulation. However, the UKF has some trials where the estimation error falls outside the 3σ bounds for a short period, but quickly decrease and can be seen in Figure 2, Figure 3, Figure 5, and Figure 6. Both the EKF and UKF show similar behavior in the estimation error, but the UKF is consistently more accurate in its estimate. This is expected because the UKF more accurately captures the true probability density through the propagation of sigma points. Figure 7 shows that the UKF is also more consistent over the time period. However, the EKF shows its strength in computational speed, which was 6.04 times faster than the UKF on average over the 100 Monte Carlo simulations.

B. Range and Range-Rate Discussion

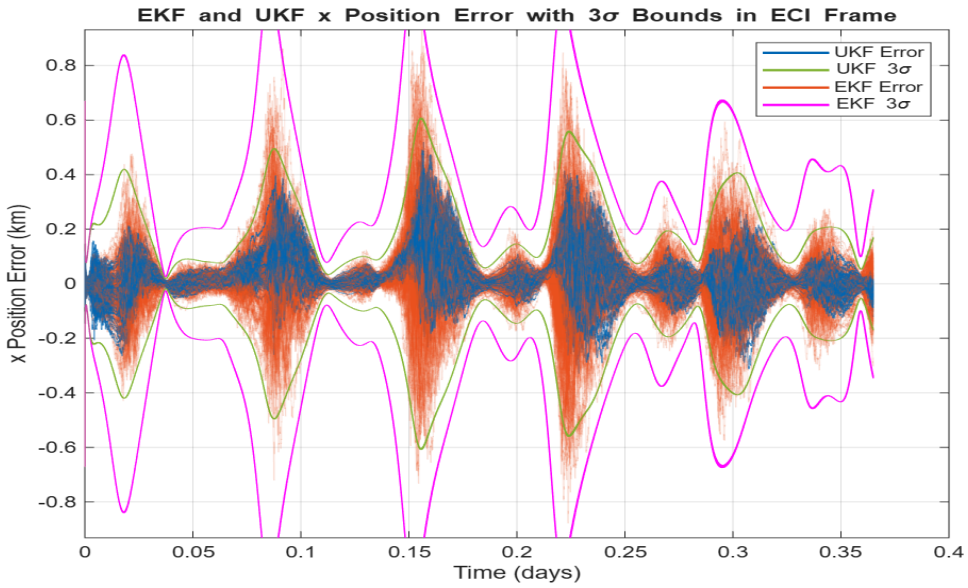

Fig. 8 EKF and UKF X position estimation error.

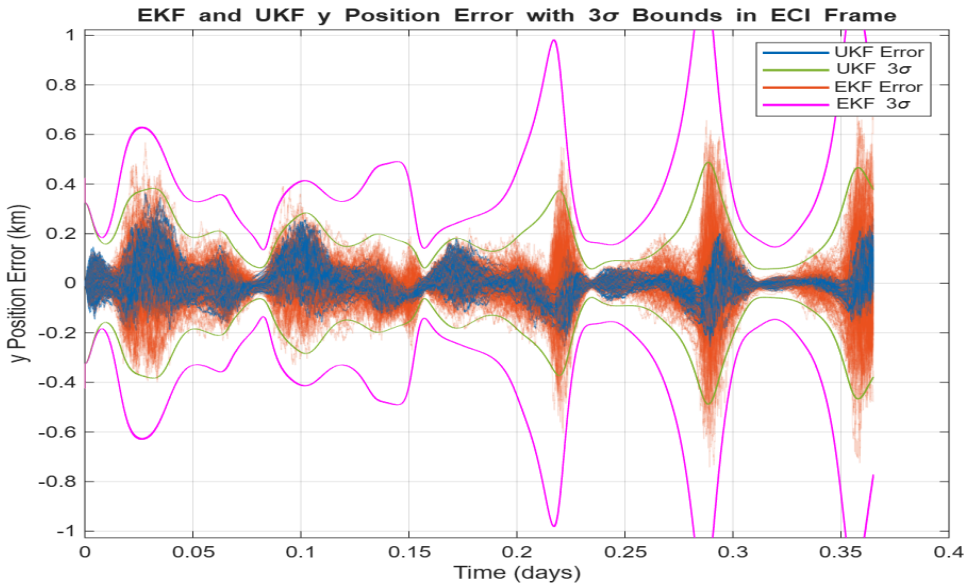

Fig. 9 EKF and UKF X position estimation error.

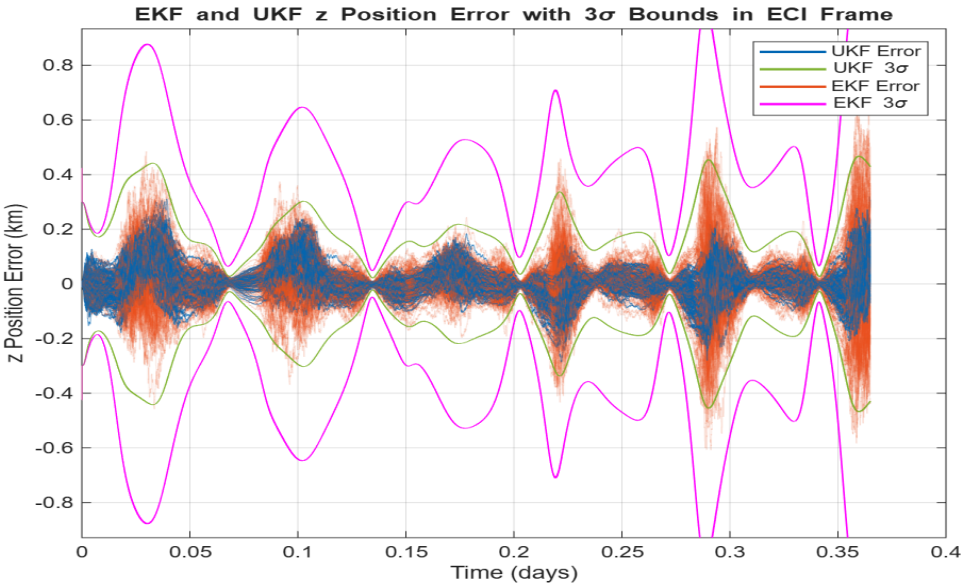

Fig. 10 EKF and UKF Z position estimation error.

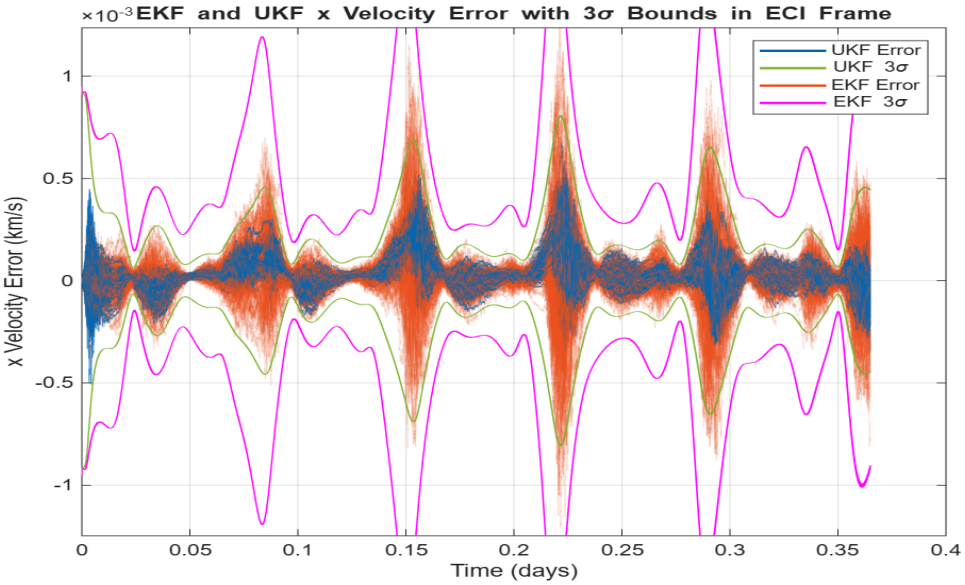

Fig. 11 EKF and UKF X velocity estimation error.

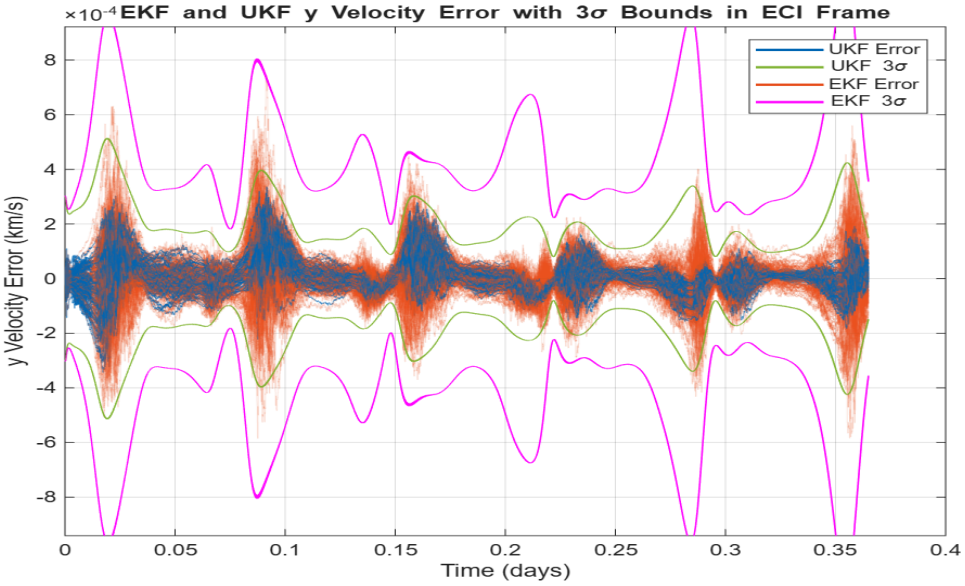

Fig. 12 EKF and UKF Y velocity estimation error.

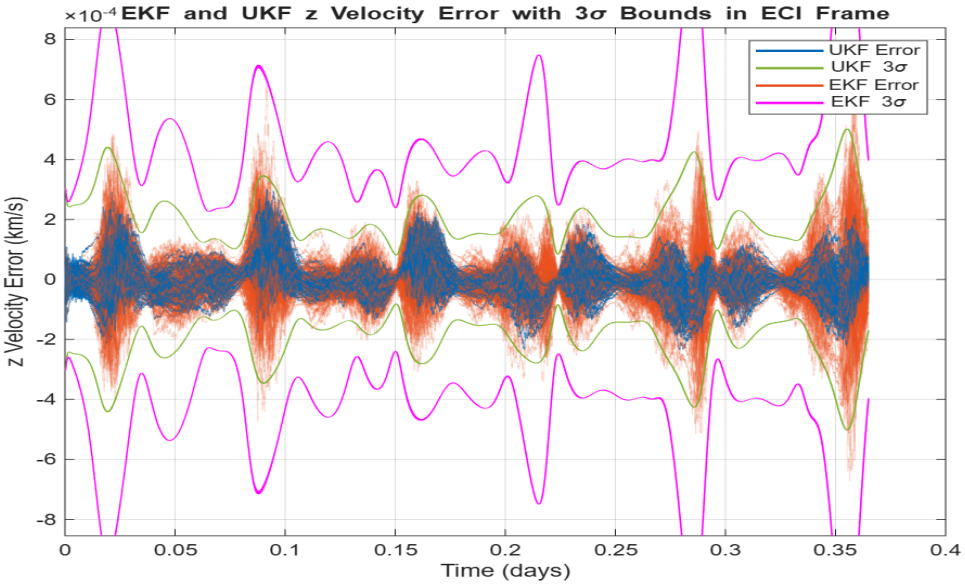

Fig. 13 EKF and UKF Z velocity estimation error.

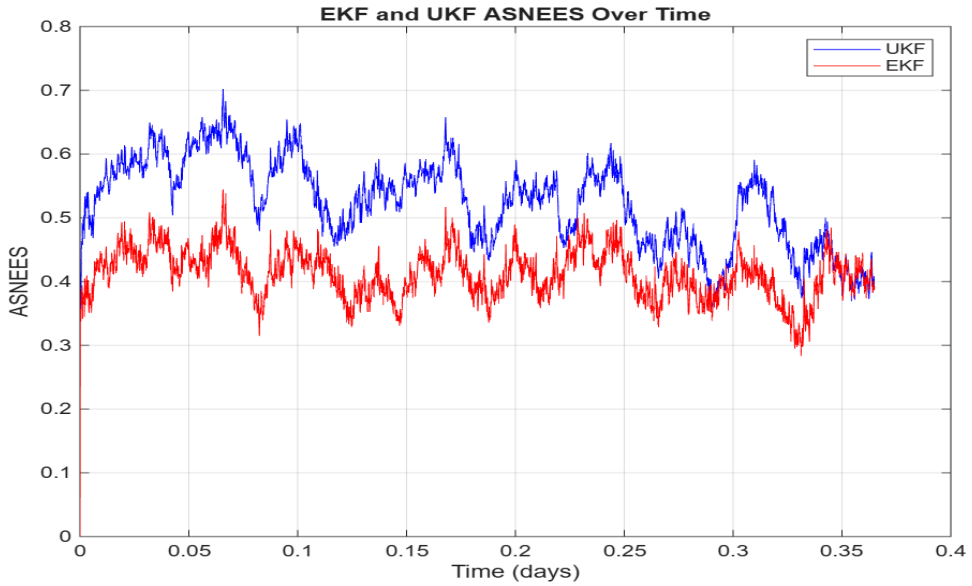

Fig. 14 ASNEES values over time. Values near one indicate a consistent filter.

Figures 8 - 13 show that, using range and range-rate measurements, the UKF performs better in terms of estimation error compared to the EKF. In contrast to the angles-only measurements, both filters don't seem to exceed the 3σ bounds. The EKF and UKF show a similar pattern in their estimation, but with the EKF showing larger uncertainty and estimation error. The estimation error is similar to that of the angles-only measurements, with both methods estimating the state well. In terms of consistency, the EKF performance is similar between both measurement models. However, the UKF seems to be more pessimistic near the end of the simulation using range and range-rate measurements, as shown in Figure 14 and Figure 7. When comparing computation speed, the EKF was 5.89 times faster on average compared to the UKF over the 100 Monte Carlo trials, again showing that the EKF's power lies in its fast computation speed.

C. Combined Measurement Discussion

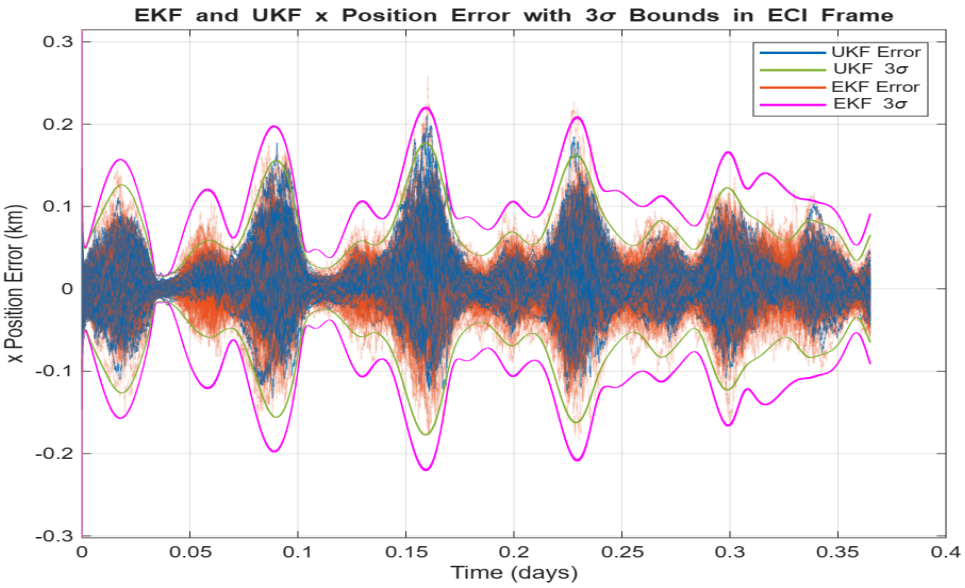

Fig. 15 EKF and UKF X position estimation error

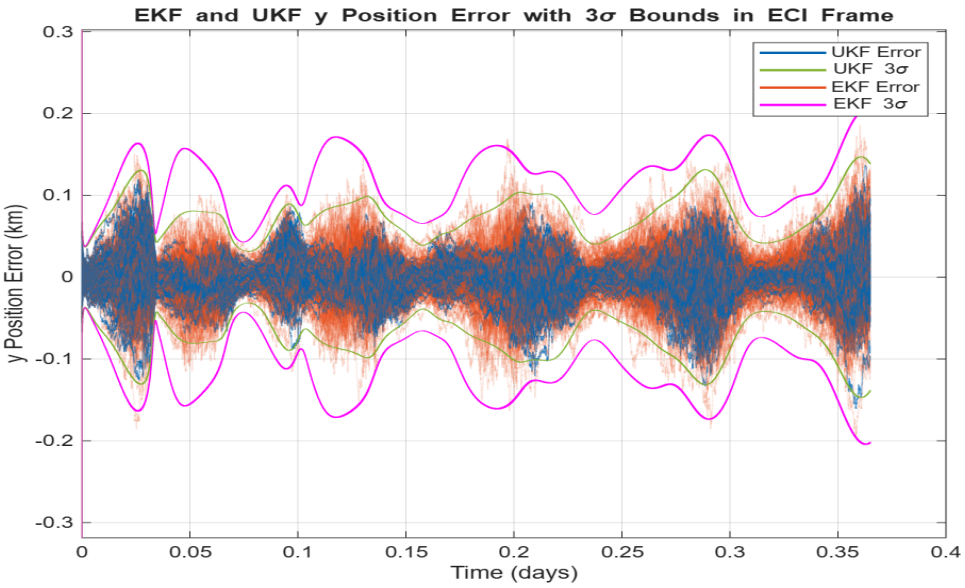

Fig. 16 EKF and UKF X position estimation error

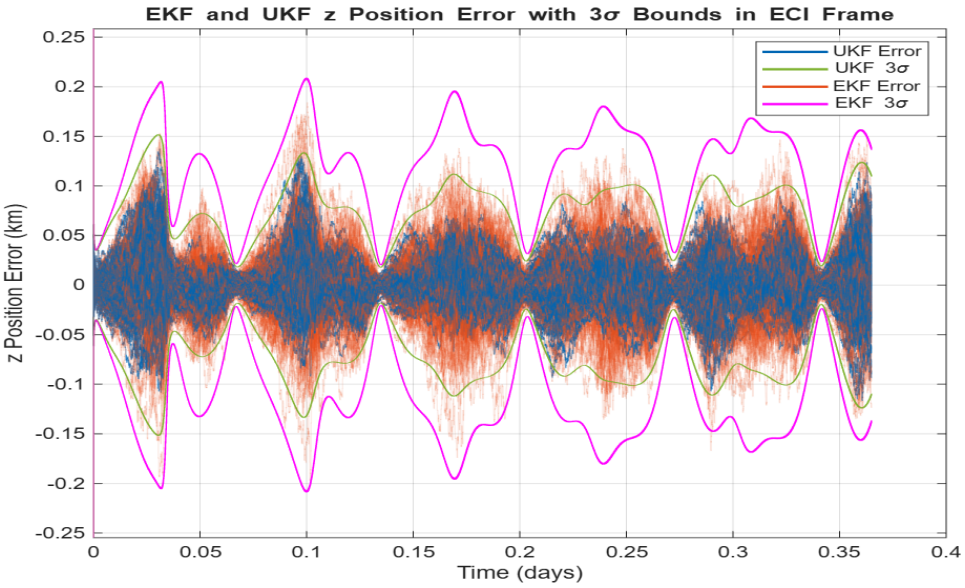

Fig. 17 EKF and UKF Z position estimation error

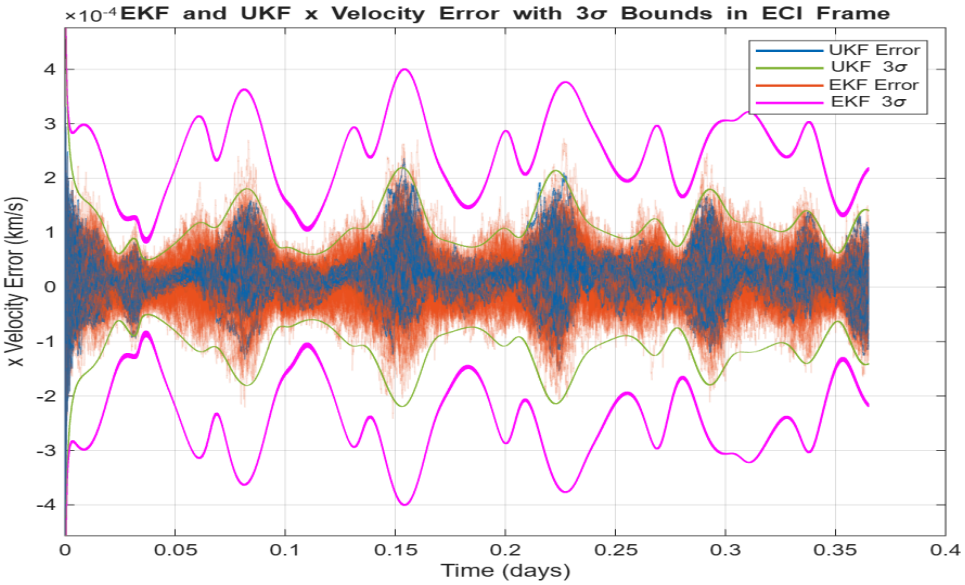

Fig. 18 EKF and UKF X velocity estimation error

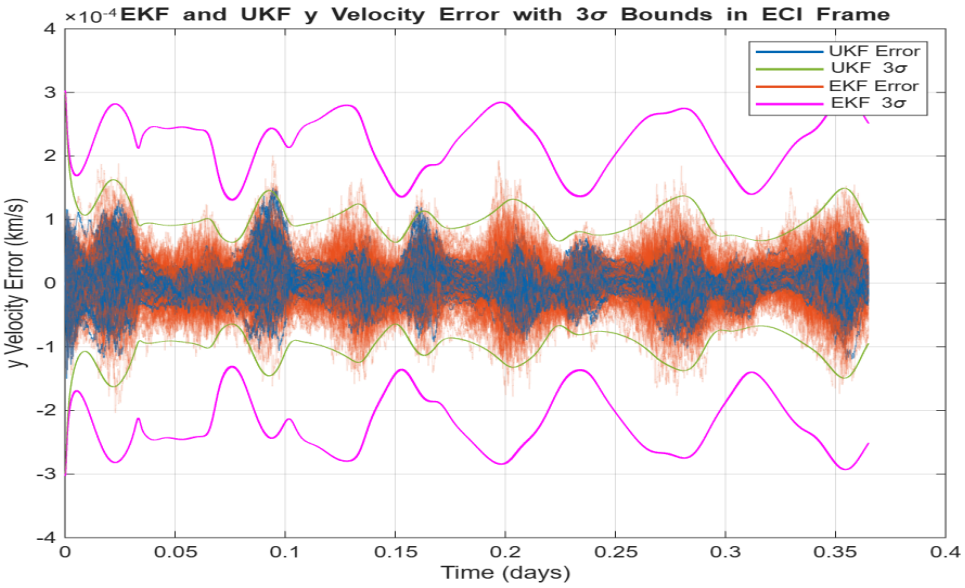

Fig. 19 EKF and UKF Y velocity estimation error

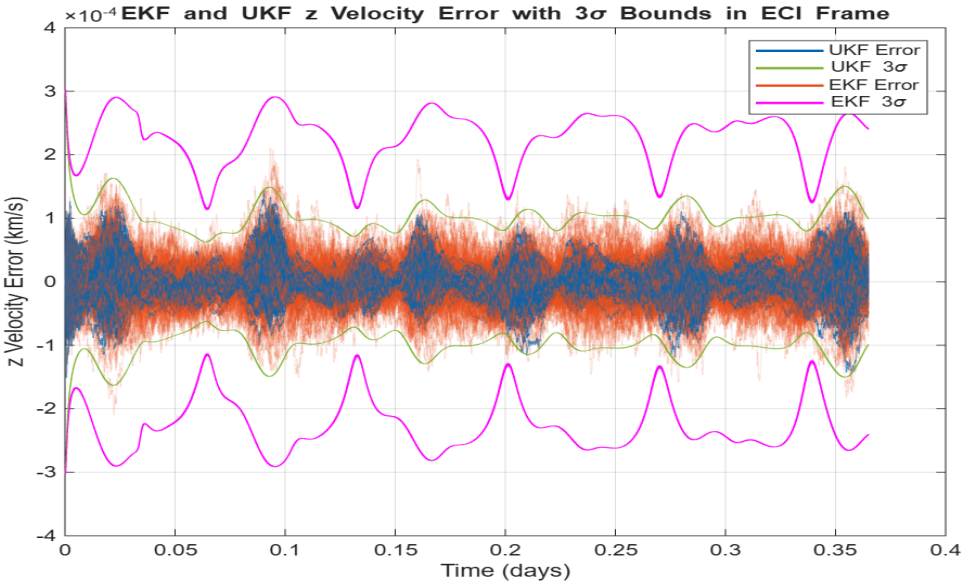

Fig. 20 EKF and UKF Z velocity estimation error

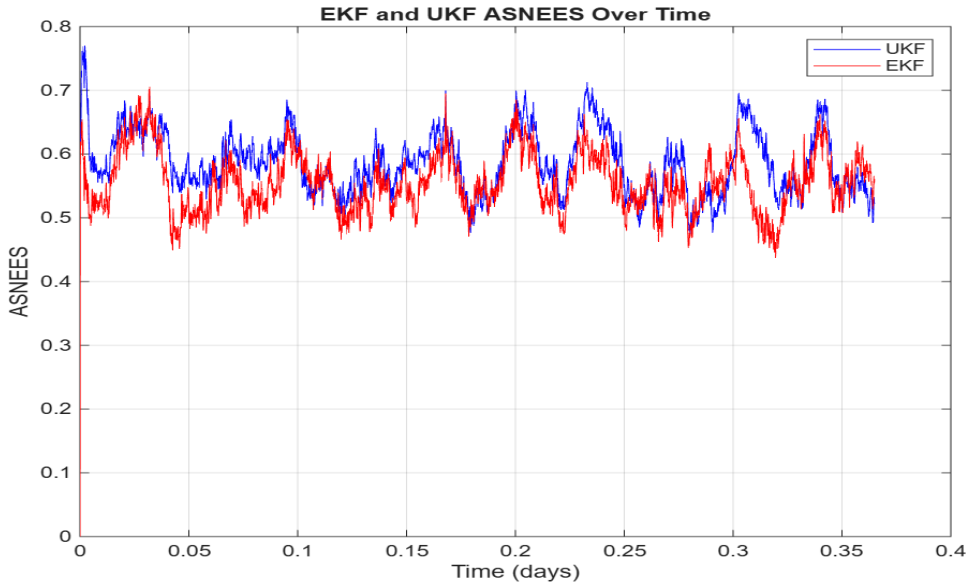

Fig. 21 ASNEES values over time. Values near one indicate a consistent filter.

Figures 15 - 20 again show that the UKF's estimation error is lower compared to the EKF using combined measurements. However, the EKF and UKF perform noticeably more similar compared to the angles-only and range and range-rate models. There are also significant periods where the UKF exceeds the 3σ bounds, which is most prominent in Figure 15. This is expected because the majority of the solar radiation pressure force is in the x direction during the simulation. In contrast, the EKF stays within its 3σ for all trials and states. In terms of pure estimation performance, the combined measurement model performs significantly better compared to the angles-only and range and range-rate models as expected. Figure 21 shows that both filters are almost equally consistent over the simulation time period, with both filters being pessimistic as the ASNEES values are below one. When comparing computation time, the EKF was 5.52 times faster on average compared to the UKF over the 100 Monte Carlo trials.

V. Conclusion

The results show that the UKF performs better and yields lower estimation errors compared to the EKF for all measurement models. However, the EKF provided similar estimation performance with an average computation time almost six times faster than the UKF over the 100 Monte Carlo trials. When comparing the angles-only and range and range-rate measurement models, both filters showed similar performance in their estimation error. However, the combined measurement model indicated significantly better performance as expected because it provides improved observability of the satellite. The UKF was more consistent with the angles-only and range and range-rate models, but showed similar consistency with the EKF when using combined measurements as shown by ASNEES plots. This study shows the power of the EKF in its ability to not only accurately estimate satellite states under model mismatch, it is also robust and is much less computationally expensive compared to the UKF in LEO applications. Satellites are constrained

by the computational power available on-board, and as a result, the EKF is the superior option when applied to space systems because it provides comparable estimation performance at almost six times the speed.

Appendix

Listing 1 Orbit Determination Code

```
clc; clear; close all;
rng(1)
% setting meas_model to 1 uses optical measurements (angles only) and 2
% uses radar (range and range-rate). If set to 3, the script runs with
% optical and radar measurement models.
meas_model = 3;

% select number of Monte Carlo trials
MC_trials = 100;

%% Initialization
% Initialize keplerian orbital elements
param.mu_Earth = 3.986*10^5; % Earth gravitational parameter (km^3/s^2)
r_Earth = 6378; % radius of Earth (km)

h_alt = 1000; % satellite orbit altitude at periapsis (km)
a =  r_Earth + h_alt; % semi major axis (km)
e = 0.01; % eccentricity
incl = 45*pi/180; % inclination (rad)
right_ascen = 0*pi/180; % longitude of the ascending node (rad)
arg_perigee = 0*pi/180; % argument of perigee (rad)
true_anom = 0*pi/180; % true anomoly (rad)

r_sat = a*(1 - e^2)/(1 + e*cos(true_anom)); % satellite initial radial position from
    center of Earth (km)
r_sat_pf = [r_sat*cos(true_anom); r_sat*sin(true_anom); 0]; % satellite perifocal
    position (km)

% transform satellite perifocal position to ECI (km)
r_sat_ECI = DCM3(right_ascen)*DCM1(incl)*DCM3(arg_perigee)*r_sat_pf;
```

```
29
30 % determine satellite perifocal velocity (km/s)
31 h = sqrt(param.mu_Earth*a*(1 - e^2)); % specific angular momentum (km^2/s)
32 rdot_sat = h*e*sin(true_anom)/(a*(1 - e^2)); % radial velocity (km/s)
33 thetadot_sat = h/r_sat; % angular velocity (km/s)
34 v_sat = [rdot_sat*cos(true_anom) - thetadot_sat*sin(true_anom); rdot_sat*sin(true_anom
       ) + thetadot_sat*cos(true_anom); 0]; % satellite velocity in perifocal frame (km/s)
35
36 % transform satellite perifocal velocity to ECI (km)
37 v_sat_ECI = DCM3(right_ascen)*DCM1(incl)*DCM3(arg_perigee)*v_sat;
38
39 % initial satellite states
40 x_sat = r_sat_ECI(1); % satellite initial x position (km)
41 y_sat = r_sat_ECI(2); % satellite initial y position (km)
42 z_sat = r_sat_ECI(3); % satellite initial z position (km)
43
44 vx_sat = v_sat_ECI(1); % satellite initial x velocity (km/s)
45 vy_sat = v_sat_ECI(2); % satellite initial x velocity (km/s)
46 vz_sat = v_sat_ECI(3); % satellite initial x velocity (km/s)
47
48 num_orbits = 5; % number of orbit period simulations
49 v0 = sqrt(2*param.mu_Earth/r_sat - param.mu_Earth/a); % satellite initial velocity (km
       /s)
50 T = num_orbits*2*pi*sqrt(a^3/param.mu_Earth); % orbit period (s)
51 x0_sat = [x_sat; vx_sat; y_sat; vy_sat; z_sat; vz_sat]; % satellite initial state in
       ECI [x; xdot; y; ydot; z; zdot]
52
53 % initialize ground station location
54 r_gs = r_Earth; % ground station distance from center of Earth (km)
55 lat_gs = 0*pi/180; % ground station latitude position (rad)
56 lon_gs = 0*pi/180; % ground station longitude position (rad)
57
58 % ground station measurement limits
59 el_lim_low = 0*pi/180; % elevation limit lower bound (rad)
60 el_lim_high = 90*pi/180; % elevation limit upper bound (rad)
```

```
61
62 % ground station position in ECEF coordinates
63 r_gs_ECEF = [r_gs*cos(lat_gs)*cos(lon_gs);
64                  r_gs*cos(lat_gs)*sin(lon_gs);
65                  r_gs*sin(lat_gs)];
66
67 dt = 10; % measurement time step (s)
68 t_meas = 0:dt:T; % measurement times (s)
69 t_comb = reshape([t_meas/60/60/24; t_meas/60/60/24],[],1); % plot time in days (days)
       for combined array
70 t_ASNEES = t_meas/60/60/24; % plot time in days for ASNEES (days)
71 num_time_steps = length(t_meas); % number of measurement time steps
72
73 n_sat = length(x0_sat); % state dimension
74
75 % determine measurement dimension
76 if meas_model == 1
77     n_meas = 2;
78 elseif meas_model == 2
79     n_meas = 2;
80 else
81     n_meas = 4;
82 end
83
84 % process noise parameters
85 sig_vx = 1.5e-6; % x velocity process noise standard deviation (km/s)
86 sig_vy = 1e-6; % y velocity process noise standard deviation (km/s)
87 sig_vz = 1e-6; % z velocity process noise standard deviation (km/s)
88 G = [0 0 0; 1 0 0; 0 0 0; 0 1 0; 0 0 0; 0 0 1]; % process noise mapping matrix
89 Q = dt*G*diag([sig_vx^2, sig_vy^2, sig_vz^2])*G';
90
91 % measurement noise parameters
92 sig_range = 0.05; % range measurement noise standard deviation (km)
93 sig_range_rate = 5e-5; % range-rate measurement noise standard deviation (km/s)
94 sig_el = 10*pi/(180*3600); % elevation measurement noise standard deviation (rad)
```

```
sig_az = 10*pi/(180*3600); % azimuth measurement noise standard deviation (rad)

% choose measurement noise covariance based on measurement model
if meas_model == 1
    R = diag([sig_el^2, sig_az^2]);
elseif meas_model == 2
    R = diag([sig_range^2, sig_range_rate^2]);
else
    R = diag([sig_range^2, sig_range_rate^2, sig_el^2, sig_az^2]);
end

% satellite physical parameters based on 6U CubeSat
A = 0.085; % 1U cubesat frontal area (m^2)
m = 10; % 1U cubesat mass (kg)
param.area_mass = A/m; % 1U cubesat area to mass ratio for drag and radiation forces

% drag and solar pressure force parameters
param.rho_ref = 1.225; % reference density (kg/m^3)
param.dens_scale = 1/7.8; % km density scaling parameter
param.h0 = r_Earth; % reference altitude (km)
param.CD = 2.13; % 1U cubesat drag coefficient
param.P = 4.56*10^-6; % momentum flux from the sun from Tapley
param.nu = 1; % eclipse factor (assume satellite is always exposed to sun)
param.CR = 1; % reflectivity coefficient approximately 1 according to Tapley

%% True Orbit and Measurement Simulation
% simulate dynamics
opts = odeset('RelTol',1e-12,'AbsTol',1e-12);

% preallocate measurement variable memory
z_k_range = NaN(1, num_time_steps);
z_k_range_rate = NaN(1, num_time_steps);
z_k_el = NaN(1, num_time_steps);
z_k_az = NaN(1, num_time_steps);

```

```
% preallocate measurement to ECI conversion
x_meas = NaN(1, num_time_steps);
y_meas = NaN(1, num_time_steps);
z_meas = NaN(1, num_time_steps);
r_rel_ECEF = NaN(3, num_time_steps);

w_Earth = 7.292115*10^-5; % Earth angular velocity (rad/s)
T_t = ECEF2Topo(lat_gs, lon_gs); % ground station position in topocentric frame

z = NaN(n_meas, num_time_steps); % preallocate combined measurements
Z_UKF = NaN(n_meas, num_time_steps); % preallocate UKF measurements
Xkm = NaN(n_sat, 2*n_sat + 1); % preallocate sigma point propogation

%% EKF and UKF loop
% initialize SUT weighting parameters
alpha = 1e-3;
beta = 2;
kappa = 0;

% parameters to determine output vectors in ODE_Low_Fidelity function
% EKF = 1 and UKF = 2
EKF = 1;
UKF = 2;

x_high_fidel = NaN(n_sat, num_time_steps,MC_trials); % preallocate simulated dynamics

P0 = diag([0.05, 10^-7, 0.02, 10^-8, 0.02, 10^-8]); % initial covariance matrix
Pkm_hist_UKF = NaN(n_sat,n_sat,num_time_steps,MC_trials); % initialize priori
    covariance matrix
Pkp_hist_UKF = NaN(n_sat,n_sat,num_time_steps,MC_trials); % initialize posterior
    covariance matrix
Pkm_hist_EKF = NaN(n_sat,n_sat,num_time_steps,MC_trials); % initialize priori
    covariance matrix
Pkp_hist_EKF = NaN(n_sat,n_sat,num_time_steps,MC_trials); % initialize posterior
    covariance matrix
```

```
161
162 % initialize UKF a priori and posterior states and covariances
163 xkm_hist_UKF = NaN(n_sat,num_time_steps,MC_trials); % initialize priori state estimate
164 xkp_hist_UKF = NaN(n_sat,num_time_steps,MC_trials); % initialize posterior state
        estimate
165 xkm_hist_EKF = NaN(n_sat,num_time_steps,MC_trials); % initialize priori state estimate
166 xkp_hist_EKF = NaN(n_sat,num_time_steps,MC_trials); % initialize posterior state
        estimate
167
168 % initialize runtime analysis variables
169 EKF_tot_time = 0;
170 UKF_tot_time = 0;
171
172 % initialize innovations variable
173 inno = NaN(n_meas,num_time_steps,MC_trials);
174
175 % start monte carlo trials
176 for i = 1:MC_trials
177     Pkm_hist_EKF(:,:,1,i) = P0; % initial covariance matrix
178     Pkp_hist_EKF(:,:,1,i) = P0; % initial covariance matrix
179     Pkm_hist_UKF(:,:,1,i) = P0; % initial covariance matrix
180     Pkp_hist_UKF(:,:,1,i) = P0; % initial covariance matrix
181
182     sig_initial = chol(Pkm_hist_UKF(:,:,1,i), 'lower'); % initial state standard
            deviation
183
184     x0 = x0_sat + sig_initial*randn(n_sat,1);
185     fprintf('Trial␣%d:␣Initial␣state␣error␣=␣[%.3f,␣%.3f,␣%.3f]␣km\n', ...
186         i, x0(1)-x0_sat(1), x0(3)-x0_sat(3), x0(5)-x0_sat(5));
187     xkm_hist_EKF(:,1,i) = x0; % initial state estimate
188     xkp_hist_EKF(:,1,i) = x0; % initial state estimate
189     xkm_hist_UKF(:,1,i) = x0; % initial state estimate
190     xkp_hist_UKF(:,1,i) = x0; % initial state estimate
191
192     % initialize filter recursion
```

```
193     xkm1_EKF = xkm_hist_EKF(:,1,i);
194     Pkm1_EKF = Pkm_hist_EKF(:,:,1,i);
195     tkm1_EKF = 0;
196     xkm1_UKF = xkm_hist_UKF(:,1,i);
197     Pkm1_UKF = Pkm_hist_UKF(:,:,1,i);
198     tkm1_UKF = 0;
199
200     % sample measurement and process noise
201     w_vx_i = sig_vx*randn(1,num_time_steps);
202     w_vy_i = sig_vy*randn(1,num_time_steps);
203     w_vz_i = sig_vz*randn(1,num_time_steps);
204
205     v_range_i = sig_range*randn(1,num_time_steps);
206     v_range_rate_i = sig_range_rate*randn(1,num_time_steps);
207     v_el_i = sig_el*randn(1,num_time_steps);
208     v_az_i = sig_az*randn(1,num_time_steps);
209
210     % combine process noise and measurement noise
211     w = [w_vx_i; w_vy_i; w_vz_i];
212     v = [v_range_i; v_range_rate_i; v_el_i; v_az_i];
213
214     x_high_fidel(:,1,i) = x0; % initial high fidelity state
215
216     % simulate true dynamics for current monte carlo trial
217     [t, x_intermed] = ode45(@ODE_High_Fidelity, t_meas, x_high_fidel(:,1,i), opts,
            param, w_Earth);
218     x_high_fidel(:,:,i) = x_intermed.';
219
220     for j = 1:num_time_steps
221         z(:,j) = meas_func(x_high_fidel(:,j,i), r_gs_ECEF, t_meas(j), T_t, w_Earth,
                meas_model, el_lim_low, el_lim_high, v(:,j));
222     end
223
224     % run EKF loop for current monte carlo trial
225     tic
```

```
    for k = 2:num_time_steps
        % check for positive definiteness
        if any(eig(Pkm1_EKF) < 1e-12) || ~isequal(Pkm1_EKF, Pkm1_EKF')
            fprintf(['Warning: Covariance matrix not positive ' ...
                     'definite! Stopping the program.\n'])
            fprintf('Failure after %i iterations.\n',k - 1)
            break
        end

        % get current time and measurement
        tk = t_meas(k);
        tspan = [tkm1_EKF tk];
        zk = z(:,k);

        % propogate step
        [~, xhat] = ode45(@ODE_Low_Fidelity, tspan, [xkm1_EKF; reshape(Pkm1_EKF,n_sat
            ^2,1)], opts, param.mu_Earth, Q, n_sat, EKF);
        xkm_EKF = xhat(end,1:n_sat)';
        Pkm_EKF = reshape(xhat(end,n_sat + 1:end),n_sat,n_sat);

        % skip update step if measurement unavailable
        if any(isnan(zk))
            % store values
            xkm_hist_EKF(:,k,i) = xkm_EKF;
            xkp_hist_EKF(:,k,i) = xkm_EKF;
            Pkm_hist_EKF(:,:,k,i) = Pkm_EKF;
            Pkp_hist_EKF(:,:,k,i) = Pkm_EKF;

            % Update recursion variables
            xkm1_EKF = xkm_EKF;
            Pkm1_EKF = (Pkm_EKF + Pkm_EKF')/2;
            tkm1_EKF = tk;

            continue
        end
```

```
260
261         % update step
262         H_k = Meas_Jacobian(xkm_EKF, r_gs_ECEF, tk, w_Earth, T_t, el_lim_low,
                el_lim_high, meas_model);
263         zhat_k = meas_func(xkm_EKF, r_gs_ECEF, tk, T_t, w_Earth, meas_model,
                el_lim_low, el_lim_high);
264         W_k = H_k*Pkm_EKF*H_k.' + R; % innovation covariance gain
265         C_k = Pkm_EKF*H_k.'; % cross covariance gain
266         K_k = C_k/W_k; % kalman gain
267
268         xkp_EKF = xkm_EKF + K_k*(zk - zhat_k); % update state estimate
269         Pkp_EKF = Pkm_EKF - C_k*K_k.' - K_k*C_k.' + K_k*W_k*K_k.'; % update covariance
270         Pkp_EKF = (Pkp_EKF + Pkp_EKF')/2; % enforce symmetry
271
272         % store values
273         xkm_hist_EKF(:,k,i) = xkm_EKF;
274         xkp_hist_EKF(:,k,i) = xkp_EKF;
275         Pkm_hist_EKF(:,:,k,i) = Pkm_EKF;
276         Pkp_hist_EKF(:,:,k,i) = Pkp_EKF;
277
278         % recurse
279         xkm1_EKF = xkp_EKF;
280         Pkm1_EKF = Pkp_EKF;
281         tkm1_EKF = tk;
282     end
283
284     EKF_time = toc;
285
286     % check if measurement is available
287     if ~any(isnan(zk))
288         % display runtimes for each trial and determine running total
289         EKF_tot_time = EKF_tot_time + EKF_time;
290         disp(['EKF elapsed time is ' num2str(EKF_time) ' seconds for Monte Carlo trial
                ' num2str(i)]);
291     end
```

```
292
293     % run UKF loop for current monte carlo trial
294     tic
295     for k = 2:num_time_steps
296         % check for positive definiteness
297         if any(eig(Pkm1_UKF) < 1e-12) || ~isequal(Pkm1_UKF, Pkm1_UKF')
298             fprintf(['Warning: Covariance matrix not positive ' ...
299                      'definite! Stopping the program.\n'])
300             fprintf('Failure after %i iterations.\n',k - 1)
301             break
302         end
303
304         % get current time and measurement
305         tk = t_meas(k);
306         tspan = [tkm1_UKF tk];
307         zk = z(:,k);
308
309         % form sigma points and calculate weights
310         [sig_points, wm, wc] = SigPoints(Pkm1_UKF, xkm1_UKF, alpha, beta, kappa, n_sat
                );
311
312         % propogate sigma points
313         for j = 1:(2*n_sat + 1)
314             [~, Xhat] = ode45(@ODE_Low_Fidelity, tspan, sig_points(:,j), opts, param.
                    mu_Earth, Q, n_sat, UKF);
315             Xkm(:,j) = Xhat(end,:).';
316         end
317
318         % calculate SUT propogated mean estimate
319         xkm_UKF = wm(1)*Xkm(:,1);
320
321         for j = 2:(2*n_sat + 1)
322             xkm_UKF = xkm_UKF + wm(j)*Xkm(:,j);
323         end
324
```

```
        % calculate SUT propogated covariance estimate
        Pkm_UKF = wc(1)*(Xkm(:,1) - xkm_UKF)*(Xkm(:,1) - xkm_UKF)' + Q;

        for j = 2:(2*n_sat + 1)
            Pkm_UKF = Pkm_UKF + wc(j)*(Xkm(:,j) - xkm_UKF)*(Xkm(:,j) - xkm_UKF)';
        end

        % skip update step if measurement unavailable
        if any(isnan(zk))
            % store values
            xkm_hist_UKF(:,k,i) = xkm_UKF;
            xkp_hist_UKF(:,k,i) = xkm_UKF;
            Pkm_hist_UKF(:,:,k,i) = (Pkm_UKF + Pkm_UKF')/2;
            Pkp_hist_UKF(:,:,k,i) = (Pkm_UKF + Pkm_UKF')/2;

            % Update recursion variables
            xkm1_UKF = xkm_UKF;
            Pkm1_UKF = (Pkm_UKF + Pkm_UKF')/2;
            tkm1_UKF = tk;

            continue
        end

        % update sigma points and weights
        [sig_points, wm, wc] = SigPoints(Pkm_UKF, xkm_UKF, alpha, beta, kappa, n_sat);

        for l = 1:(2*n_sat + 1)
            Z_UKF(:,l) = meas_func(sig_points(:,l), r_gs_ECEF, tk, T_t, w_Earth,
                meas_model, el_lim_low, el_lim_high);
        end

        % calculate predicted measurement mean
        zhat_k_UKF = wm(1)*Z_UKF(:,1);

        for l = 2:(2*n_sat + 1)
```

```
            zhat_k_UKF = zhat_k_UKF + wm(l)*Z_UKF(:,l);
        end

        % calculate predicted measurement covariance and cross covariance
        Pzzk_UKF = wc(1)*(Z_UKF(:,1) - zhat_k_UKF)*(Z_UKF(:,1) - zhat_k_UKF)' + R;
        Pxzk_UKF = wc(1)*(sig_points(:,1) - xkm_UKF)*(Z_UKF(:,1) - zhat_k_UKF)';

        for l = 2:(2*n_sat + 1)
            Pzzk_UKF = Pzzk_UKF + wc(l)*(Z_UKF(:,l) - zhat_k_UKF)*(Z_UKF(:,l) -
                zhat_k_UKF)';
            Pxzk_UKF = Pxzk_UKF + wc(l)*(sig_points(:,l) - xkm_UKF)*(Z_UKF(:,l) -
                zhat_k_UKF)';
        end

        % compute Kalman gain and update state mean and covariance
        K_k_UKF = Pxzk_UKF/Pzzk_UKF;
        xkp_UKF = xkm_UKF + K_k_UKF*(zk - zhat_k_UKF);
        Pkp_UKF = Pkm_UKF - Pxzk_UKF*K_k_UKF' - K_k_UKF*Pxzk_UKF' + K_k_UKF*Pzzk_UKF*
            K_k_UKF';

        Pkp_UKF = (Pkp_UKF + Pkp_UKF')/2; % enforce symmetry

        % store values
        xkm_hist_UKF(:,k,i) = xkm_UKF;
        xkp_hist_UKF(:,k,i) = xkp_UKF;
        Pkm_hist_UKF(:,:,k,i) = Pkm_UKF;
        Pkp_hist_UKF(:,:,k,i) = Pkp_UKF;
        inno(:,k,i) = zk - zhat_k_UKF;

        % recurse
        xkm1_UKF = xkp_UKF;
        Pkm1_UKF = Pkp_UKF;
        tkm1_UKF = tk;
    end

```

```
391     UKF_time = toc;
392
393     % check if measurement is available
394     if ~any(isnan(zk))
395         % display runtimes for each trial and determine running total
396         UKF_tot_time = UKF_tot_time + UKF_time;
397         disp(['UKF elapsed time is ' num2str(UKF_time) ' seconds for Monte Carlo trial
                ' num2str(i)]);
398     end
399 end
400
401 % displayed total time elapsed for each filter
402 disp('------------------------------------------------------------------------')
403 disp(['EKF total run time is ' num2str(EKF_tot_time) ' seconds across ' num2str(
        MC_trials) ' Monte Carlo trial']);
404 disp(['UKF total elapsed time is ' num2str(UKF_tot_time) ' seconds across ' num2str(
        MC_trials) ' Monte Carlo trial']);
405
406 % display average time elapsed for each Monte Carlo trial
407 disp('------------------------------------------------------------------------')
408 disp(['EKF average run time is ' num2str(EKF_tot_time/MC_trials) ' seconds across '
        num2str(MC_trials) ' Monte Carlo trial']);
409 disp(['UKF average run time is ' num2str(UKF_tot_time/MC_trials) ' seconds across '
        num2str(MC_trials) ' Monte Carlo trial']);
410 disp('------------------------------------------------------------------------')
411
412 % display how much faster EKF is compared to UKF on average
413 disp(['EKF is ' num2str(UKF_tot_time/EKF_tot_time) ' times faster on average compared
        to UKF over ' num2str(MC_trials) ' Monte Carlo trials']);
414 disp('------------------------------------------------------------------------')
415
416 % initialize EKF and UKF combined error and sigma arrays for plotting
417 x_pos_error_comb_EKF = NaN(MC_trials,2*num_time_steps);
418 y_pos_error_comb_EKF = NaN(MC_trials,2*num_time_steps);
419 z_pos_error_comb_EKF = NaN(MC_trials,2*num_time_steps);
```

```
x_vel_error_comb_EKF = NaN(MC_trials,2*num_time_steps);
y_vel_error_comb_EKF = NaN(MC_trials,2*num_time_steps);
z_vel_error_comb_EKF = NaN(MC_trials,2*num_time_steps);

x_pos_sig_comb_EKF = NaN(MC_trials,2*num_time_steps);
y_pos_sig_comb_EKF = NaN(MC_trials,2*num_time_steps);
z_pos_sig_comb_EKF = NaN(MC_trials,2*num_time_steps);
x_vel_sig_comb_EKF = NaN(MC_trials,2*num_time_steps);
y_vel_sig_comb_EKF = NaN(MC_trials,2*num_time_steps);
z_vel_sig_comb_EKF = NaN(MC_trials,2*num_time_steps);

x_pos_error_comb_UKF = NaN(MC_trials,2*num_time_steps);
y_pos_error_comb_UKF = NaN(MC_trials,2*num_time_steps);
z_pos_error_comb_UKF = NaN(MC_trials,2*num_time_steps);
x_vel_error_comb_UKF = NaN(MC_trials,2*num_time_steps);
y_vel_error_comb_UKF = NaN(MC_trials,2*num_time_steps);
z_vel_error_comb_UKF = NaN(MC_trials,2*num_time_steps);

x_pos_sig_comb_UKF = NaN(MC_trials,2*num_time_steps);
y_pos_sig_comb_UKF = NaN(MC_trials,2*num_time_steps);
z_pos_sig_comb_UKF = NaN(MC_trials,2*num_time_steps);
x_vel_sig_comb_UKF = NaN(MC_trials,2*num_time_steps);
y_vel_sig_comb_UKF = NaN(MC_trials,2*num_time_steps);
z_vel_sig_comb_UKF = NaN(MC_trials,2*num_time_steps);

% calculate ASNEES values and plot estimation errors with 3 sigma bounds
for i = 1:MC_trials
    % EKF and UKF position priori and posterior errors
    xkp_pos_error_EKF = squeeze(x_high_fidel(1,:,i) - xkp_hist_EKF(1,:,i));
    ykp_pos_error_EKF = squeeze(x_high_fidel(3,:,i) - xkp_hist_EKF(3,:,i));
    zkp_pos_error_EKF = squeeze(x_high_fidel(5,:,i) - xkp_hist_EKF(5,:,i));
    xkm_pos_error_EKF = squeeze(x_high_fidel(1,:,i) - xkm_hist_EKF(1,:,i));
    ykm_pos_error_EKF = squeeze(x_high_fidel(3,:,i) - xkm_hist_EKF(3,:,i));
    zkm_pos_error_EKF = squeeze(x_high_fidel(5,:,i) - xkm_hist_EKF(5,:,i));

```

```
    xkp_pos_error_UKF = squeeze(x_high_fidel(1,:,i) - xkp_hist_UKF(1,:,i));
    ykp_pos_error_UKF = squeeze(x_high_fidel(3,:,i) - xkp_hist_UKF(3,:,i));
    zkp_pos_error_UKF = squeeze(x_high_fidel(5,:,i) - xkp_hist_UKF(5,:,i));
    xkm_pos_error_UKF = squeeze(x_high_fidel(1,:,i) - xkm_hist_UKF(1,:,i));
    ykm_pos_error_UKF = squeeze(x_high_fidel(3,:,i) - xkm_hist_UKF(3,:,i));
    zkm_pos_error_UKF = squeeze(x_high_fidel(5,:,i) - xkm_hist_UKF(5,:,i));

    % EKF and UKF velocity priori and posterior errors
    xkp_vel_error_EKF = squeeze(x_high_fidel(2,:,i) - xkp_hist_EKF(2,:,i));
    ykp_vel_error_EKF = squeeze(x_high_fidel(4,:,i) - xkp_hist_EKF(4,:,i));
    zkp_vel_error_EKF = squeeze(x_high_fidel(6,:,i) - xkp_hist_EKF(6,:,i));
    xkm_vel_error_EKF = squeeze(x_high_fidel(2,:,i) - xkm_hist_EKF(2,:,i));
    ykm_vel_error_EKF = squeeze(x_high_fidel(4,:,i) - xkm_hist_EKF(4,:,i));
    zkm_vel_error_EKF = squeeze(x_high_fidel(6,:,i) - xkm_hist_EKF(6,:,i));

    xkp_vel_error_UKF = squeeze(x_high_fidel(2,:,i) - xkp_hist_UKF(2,:,i));
    ykp_vel_error_UKF = squeeze(x_high_fidel(4,:,i) - xkp_hist_UKF(4,:,i));
    zkp_vel_error_UKF = squeeze(x_high_fidel(6,:,i) - xkp_hist_UKF(6,:,i));
    xkm_vel_error_UKF = squeeze(x_high_fidel(2,:,i) - xkm_hist_UKF(2,:,i));
    ykm_vel_error_UKF = squeeze(x_high_fidel(4,:,i) - xkm_hist_UKF(4,:,i));
    zkm_vel_error_UKF = squeeze(x_high_fidel(6,:,i) - xkm_hist_UKF(6,:,i));

    % EKF and UKF position priori and posterior standard deviation bounds
    xkp_pos_sig_EKF = squeeze(sqrt(Pkp_hist_EKF(1,1,:,i)))';
    ykp_pos_sig_EKF = squeeze(sqrt(Pkp_hist_EKF(3,3,:,i)))';
    zkp_pos_sig_EKF = squeeze(sqrt(Pkp_hist_EKF(5,5,:,i)))';
    xkm_pos_sig_EKF = squeeze(sqrt(Pkm_hist_EKF(1,1,:,i)))';
    ykm_pos_sig_EKF = squeeze(sqrt(Pkm_hist_EKF(3,3,:,i)))';
    zkm_pos_sig_EKF = squeeze(sqrt(Pkm_hist_EKF(5,5,:,i)))';

    xkp_pos_sig_UKF = squeeze(sqrt(Pkp_hist_UKF(1,1,:,i)))';
    ykp_pos_sig_UKF = squeeze(sqrt(Pkp_hist_UKF(3,3,:,i)))';
    zkp_pos_sig_UKF = squeeze(sqrt(Pkp_hist_UKF(5,5,:,i)))';
    xkm_pos_sig_UKF = squeeze(sqrt(Pkm_hist_UKF(1,1,:,i)))';
    ykm_pos_sig_UKF = squeeze(sqrt(Pkm_hist_UKF(3,3,:,i)))';
```

```
    zkm_pos_sig_UKF = squeeze(sqrt(Pkm_hist_UKF(5,5,:,i)))';

    % EKF and UKF velocity priori and posterior standard deviation bounds
    xkp_vel_sig_EKF = squeeze(sqrt(Pkp_hist_EKF(2,2,:,i)))';
    ykp_vel_sig_EKF = squeeze(sqrt(Pkp_hist_EKF(4,4,:,i)))';
    zkp_vel_sig_EKF = squeeze(sqrt(Pkp_hist_EKF(6,6,:,i)))';
    xkm_vel_sig_EKF = squeeze(sqrt(Pkm_hist_EKF(2,2,:,i)))';
    ykm_vel_sig_EKF = squeeze(sqrt(Pkm_hist_EKF(4,4,:,i)))';
    zkm_vel_sig_EKF = squeeze(sqrt(Pkm_hist_EKF(6,6,:,i)))';

    xkp_vel_sig_UKF = squeeze(sqrt(Pkp_hist_UKF(2,2,:,i)))';
    ykp_vel_sig_UKF = squeeze(sqrt(Pkp_hist_UKF(4,4,:,i)))';
    zkp_vel_sig_UKF = squeeze(sqrt(Pkp_hist_UKF(6,6,:,i)))';
    xkm_vel_sig_UKF = squeeze(sqrt(Pkm_hist_UKF(2,2,:,i)))';
    ykm_vel_sig_UKF = squeeze(sqrt(Pkm_hist_UKF(4,4,:,i)))';
    zkm_vel_sig_UKF = squeeze(sqrt(Pkm_hist_UKF(6,6,:,i)))';

    for j = 1:num_time_steps
        % standard deviation combined arrays
        x_pos_sig_comb_EKF(i,2*j - 1) = xkm_pos_sig_EKF(:,j);
        x_pos_sig_comb_EKF(i,2*j) = xkp_pos_sig_EKF(:,j);
        y_pos_sig_comb_EKF(i,2*j - 1) = ykm_pos_sig_EKF(:,j);
        y_pos_sig_comb_EKF(i,2*j) = ykp_pos_sig_EKF(:,j);
        z_pos_sig_comb_EKF(i,2*j - 1) = zkm_pos_sig_EKF(:,j);
        z_pos_sig_comb_EKF(i,2*j) = zkp_pos_sig_EKF(:,j);
        x_vel_sig_comb_EKF(i,2*j - 1) = xkm_vel_sig_EKF(:,j);
        x_vel_sig_comb_EKF(i,2*j) = xkp_vel_sig_EKF(:,j);
        y_vel_sig_comb_EKF(i,2*j - 1) = ykm_vel_sig_EKF(:,j);
        y_vel_sig_comb_EKF(i,2*j) = ykp_vel_sig_EKF(:,j);
        z_vel_sig_comb_EKF(i,2*j - 1) = zkm_vel_sig_EKF(:,j);
        z_vel_sig_comb_EKF(i,2*j) = zkp_vel_sig_EKF(:,j);

        x_pos_sig_comb_UKF(i,2*j - 1) = xkm_pos_sig_UKF(:,j);
        x_pos_sig_comb_UKF(i,2*j) = xkp_pos_sig_UKF(:,j);
        y_pos_sig_comb_UKF(i,2*j - 1) = ykm_pos_sig_UKF(:,j);
```

```
525         y_pos_sig_comb_UKF(i,2*j) = ykp_pos_sig_UKF(:,j);
526         z_pos_sig_comb_UKF(i,2*j - 1) = zkm_pos_sig_UKF(:,j);
527         z_pos_sig_comb_UKF(i,2*j) = zkp_pos_sig_UKF(:,j);
528         x_vel_sig_comb_UKF(i,2*j - 1) = xkm_vel_sig_UKF(:,j);
529         x_vel_sig_comb_UKF(i,2*j) = xkp_vel_sig_UKF(:,j);
530         y_vel_sig_comb_UKF(i,2*j - 1) = ykm_vel_sig_UKF(:,j);
531         y_vel_sig_comb_UKF(i,2*j) = ykp_vel_sig_UKF(:,j);
532         z_vel_sig_comb_UKF(i,2*j - 1) = zkm_vel_sig_UKF(:,j);
533         z_vel_sig_comb_UKF(i,2*j) = zkp_vel_sig_UKF(:,j);
534
535         % state estimation error combined arrays
536         x_pos_error_comb_EKF(i,2*j - 1) = xkm_pos_error_EKF(:,j);
537         x_pos_error_comb_EKF(i,2*j) = xkp_pos_error_EKF(:,j);
538         y_pos_error_comb_EKF(i,2*j - 1) = ykm_pos_error_EKF(:,j);
539         y_pos_error_comb_EKF(i,2*j) = ykp_pos_error_EKF(:,j);
540         z_pos_error_comb_EKF(i,2*j - 1) = zkm_pos_error_EKF(:,j);
541         z_pos_error_comb_EKF(i,2*j) = zkp_pos_error_EKF(:,j);
542         x_vel_error_comb_EKF(i,2*j - 1) = xkm_vel_error_EKF(:,j);
543         x_vel_error_comb_EKF(i,2*j) = xkp_vel_error_EKF(:,j);
544         y_vel_error_comb_EKF(i,2*j - 1) = ykm_vel_error_EKF(:,j);
545         y_vel_error_comb_EKF(i,2*j) = ykp_vel_error_EKF(:,j);
546         z_vel_error_comb_EKF(i,2*j - 1) = zkm_vel_error_EKF(:,j);
547         z_vel_error_comb_EKF(i,2*j) = zkp_vel_error_EKF(:,j);
548
549         x_pos_error_comb_UKF(i,2*j - 1) = xkm_pos_error_UKF(:,j);
550         x_pos_error_comb_UKF(i,2*j) = xkp_pos_error_UKF(:,j);
551         y_pos_error_comb_UKF(i,2*j - 1) = ykm_pos_error_UKF(:,j);
552         y_pos_error_comb_UKF(i,2*j) = ykp_pos_error_UKF(:,j);
553         z_pos_error_comb_UKF(i,2*j - 1) = zkm_pos_error_UKF(:,j);
554         z_pos_error_comb_UKF(i,2*j) = zkp_pos_error_UKF(:,j);
555         x_vel_error_comb_UKF(i,2*j - 1) = xkm_vel_error_UKF(:,j);
556         x_vel_error_comb_UKF(i,2*j) = xkp_vel_error_UKF(:,j);
557         y_vel_error_comb_UKF(i,2*j - 1) = ykm_vel_error_UKF(:,j);
558         y_vel_error_comb_UKF(i,2*j) = ykp_vel_error_UKF(:,j);
559         z_vel_error_comb_UKF(i,2*j - 1) = zkm_vel_error_UKF(:,j);
```

```
        z_vel_error_comb_UKF(i,2*j) = zkp_vel_error_UKF(:,j);
    end
end

% color arrays for plotting
color_EKF_error = [0.9, 0.3, 0.1, 0.1];
color_UKF_error = [0.0, 0.4, 0.7, 0.4];
color_EKF_sigma = [1.0, 0.0, 1.0, 0.1];
color_UKF_sigma = [0.5, 0.7, 0.2, 0.1];

% color arrays for legend
color_EKF_error_legend = [0.9, 0.3, 0.1, 1];
color_UKF_error_legend = [0.0, 0.4, 0.7, 1];
color_EKF_sigma_legend = [1.0, 0.0, 1.0, 1];
color_UKF_sigma_legend = [0.5, 0.7, 0.2, 1];

% plot position errors with 3 sigma bounds
figure(1)
for i = 1:MC_trials
    plot(t_comb,x_pos_error_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_error)
    hold on
    plot(t_comb,3*x_pos_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,-3*x_pos_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,x_pos_error_comb_EKF(i,:),'LineWidth',0.1,'Color',color_EKF_error)
    plot(t_comb,3*x_pos_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
    plot(t_comb,-3*x_pos_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
end
title('EKF␣and␣UKF␣x␣Position␣Error␣with␣3\sigma␣Bounds␣in␣ECI␣Frame')
xlabel('Time␣(days)')
ylabel('x␣Position␣Error␣(km)')
h1 = plot(NaN,NaN,'Color',color_UKF_error_legend,'LineWidth',1.5);
h2 = plot(NaN,NaN,'Color',color_UKF_sigma_legend,'LineWidth',1.5);
h3 = plot(NaN,NaN,'Color',color_EKF_error_legend,'LineWidth',1.5);
h4 = plot(NaN,NaN,'Color',color_EKF_sigma_legend,'LineWidth',1.5);
```

```
legend([h1 h2 h3 h4],'UKF Error','UKF 3\sigma','EKF Error','EKF 3\sigma','Location','
    northeast')
grid on

figure(2)
for i = 1:MC_trials
    plot(t_comb,y_pos_error_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_error)
    hold on
    plot(t_comb,3*y_pos_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,-3*y_pos_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,y_pos_error_comb_EKF(i,:),'LineWidth',0.1,'Color',color_EKF_error)
    plot(t_comb,3*y_pos_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
    plot(t_comb,-3*y_pos_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
end
title('EKF and UKF y Position Error with 3\sigma Bounds in ECI Frame')
xlabel('Time (days)')
ylabel('y Position Error (km)')
h1 = plot(NaN,NaN,'Color',color_UKF_error_legend,'LineWidth',1.5);
h2 = plot(NaN,NaN,'Color',color_UKF_sigma_legend,'LineWidth',1.5);
h3 = plot(NaN,NaN,'Color',color_EKF_error_legend,'LineWidth',1.5);
h4 = plot(NaN,NaN,'Color',color_EKF_sigma_legend,'LineWidth',1.5);
legend([h1 h2 h3 h4],'UKF Error','UKF 3\sigma','EKF Error','EKF 3\sigma','Location','
    northeast')
grid on

figure(3)
for i = 1:MC_trials
    plot(t_comb,z_pos_error_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_error)
    hold on
    plot(t_comb,3*z_pos_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,-3*z_pos_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,z_pos_error_comb_EKF(i,:),'LineWidth',0.1,'Color',color_EKF_error)
    plot(t_comb,3*z_pos_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
    plot(t_comb,-3*z_pos_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
end
```

```
title('EKF and UKF z Position Error with 3\sigma Bounds in ECI Frame')
xlabel('Time (days)')
ylabel('z Position Error (km)')
h1 = plot(NaN,NaN,'Color',color_UKF_error_legend,'LineWidth',1.5);
h2 = plot(NaN,NaN,'Color',color_UKF_sigma_legend,'LineWidth',1.5);
h3 = plot(NaN,NaN,'Color',color_EKF_error_legend,'LineWidth',1.5);
h4 = plot(NaN,NaN,'Color',color_EKF_sigma_legend,'LineWidth',1.5);
legend([h1 h2 h3 h4],'UKF Error','UKF 3\sigma','EKF Error','EKF 3\sigma','Location','
    northeast')
grid on

% plot velocity errors and 3 sigma bounds
figure(4)
for i = 1:MC_trials
    plot(t_comb,x_vel_error_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_error)
    hold on
    plot(t_comb,3*x_vel_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,-3*x_vel_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,x_vel_error_comb_EKF(i,:),'LineWidth',0.1,'Color',color_EKF_error)
    plot(t_comb,3*x_vel_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
    plot(t_comb,-3*x_vel_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
end
title('EKF and UKF x Velocity Error with 3\sigma Bounds in ECI Frame')
xlabel('Time (days)')
ylabel('x Velocity Error (km/s)')
h1 = plot(NaN,NaN,'Color',color_UKF_error_legend,'LineWidth',1.5);
h2 = plot(NaN,NaN,'Color',color_UKF_sigma_legend,'LineWidth',1.5);
h3 = plot(NaN,NaN,'Color',color_EKF_error_legend,'LineWidth',1.5);
h4 = plot(NaN,NaN,'Color',color_EKF_sigma_legend,'LineWidth',1.5);
legend([h1 h2 h3 h4],'UKF Error','UKF 3\sigma','EKF Error','EKF 3\sigma','Location','
    northeast')
grid on

figure(5)
for i = 1:MC_trials
```

```
    plot(t_comb,y_vel_error_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_error)
    hold on
    plot(t_comb,3*y_vel_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,-3*y_vel_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,y_vel_error_comb_EKF(i,:),'LineWidth',0.1,'Color',color_EKF_error)
    plot(t_comb,3*y_vel_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
    plot(t_comb,-3*y_vel_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
end
title('EKF and UKF y Velocity Error with 3\sigma Bounds in ECI Frame')
xlabel('Time (days)')
ylabel('y Velocity Error (km/s)')
h1 = plot(NaN,NaN,'Color',color_UKF_error_legend,'LineWidth',1.5);
h2 = plot(NaN,NaN,'Color',color_UKF_sigma_legend,'LineWidth',1.5);
h3 = plot(NaN,NaN,'Color',color_EKF_error_legend,'LineWidth',1.5);
h4 = plot(NaN,NaN,'Color',color_EKF_sigma_legend,'LineWidth',1.5);
legend([h1 h2 h3 h4],'UKF Error','UKF 3\sigma','EKF Error','EKF 3\sigma','Location','
    northeast')
grid on

figure(6)
for i = 1:MC_trials
    plot(t_comb,z_vel_error_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_error)
    hold on
    plot(t_comb,3*z_vel_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,-3*z_vel_sig_comb_UKF(i,:),'LineWidth',0.1,'Color',color_UKF_sigma)
    plot(t_comb,z_vel_error_comb_EKF(i,:),'LineWidth',0.1,'Color',color_EKF_error)
    plot(t_comb,3*z_vel_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
    plot(t_comb,-3*z_vel_sig_comb_EKF(i,:),'LineWidth',0.1,'color',color_EKF_sigma)
end
title('EKF and UKF z Velocity Error with 3\sigma Bounds in ECI Frame')
xlabel('Time (days)')
ylabel('z Velocity Error (km/s)')
h1 = plot(NaN,NaN,'Color',color_UKF_error_legend,'LineWidth',1.5);
h2 = plot(NaN,NaN,'Color',color_UKF_sigma_legend,'LineWidth',1.5);
h3 = plot(NaN,NaN,'Color',color_EKF_error_legend,'LineWidth',1.5);
```

```
h4 = plot(NaN,NaN,'Color',color_EKF_sigma_legend,'LineWidth',1.5);
legend([h1 h2 h3 h4],'UKF Error','UKF 3\sigma','EKF Error','EKF 3\sigma','Location','
    northeast')
grid on

% preallocate NEES and ASNEES values
NEES_EKF = NaN(1,MC_trials);
NEES_UKF = NaN(1,MC_trials);
ASNEES_EKF = NaN(1,num_time_steps);
ASNEES_UKF = NaN(1,num_time_steps);

% calculate NEES and ASNEES values for consistency analysis
for j = 1:num_time_steps
    for k = 1:MC_trials
        Pk_UKF = squeeze(Pkp_hist_UKF(:,:,j,k));
        ek_UKF = x_high_fidel(:,j,k) - xkp_hist_UKF(:,j,k);

        Pk_EKF = squeeze(Pkp_hist_EKF(:,:,j,k));
        ek_EKF = x_high_fidel(:,j,k) - xkp_hist_EKF(:,j,k);

        NEES_EKF(k) = ek_EKF'/Pk_EKF*ek_EKF;
        NEES_UKF(k) = ek_UKF'/Pk_UKF*ek_UKF;
    end
    ASNEES_EKF(j) = 1/(n_sat*MC_trials)*sum(NEES_EKF);
    ASNEES_UKF(j) = 1/(n_sat*MC_trials)*sum(NEES_UKF);
end

% plot ASNEES over time
figure(7)
plot(t_ASNEES,ASNEES_UKF,'b')
hold on
plot(t_ASNEES,ASNEES_EKF,'r')
title('EKF and UKF ASNEES Over Time')
xlabel('Time (days)')
ylabel('ASNEES')
```

```
728 legend('UKF', 'EKF', 'Location', 'northeast')
729 grid on
730
731 % plot measurements, high fidelity simulation, and UKF posterior position
732 % estimates
733 figure(8)
734 plot3(x_meas, y_meas, z_meas)
735 hold on
736 plot3(x_high_fidel(1,:), x_high_fidel(3,:), x_high_fidel(5,:))
737 plot3(xkp_hist_UKF(1,:), xkp_hist_UKF(3,:), xkp_hist_UKF(5,:))
738 title('Satellite Position and Measurements in ECI Frame')
739 xlabel('x (km)')
740 ylabel('y (km)')
741 zlabel('z (km)')
742 legend('Measurements','True Orbit','UKF Estimate')
743 axis equal
744 grid on
745
746 %% Function Definitions
747 % high fidelity dynamics function
748 function dxdt = ODE_High_Fidelity(t, x, param, w_Earth)
749     % extract parameters from param struct
750     mu = param.mu_Earth;
751     A_m = param.area_mass;
752     dens_scale = param.dens_scale;
753     h0 = param.h0;
754     rho_ref = param.rho_ref;
755     CD = param.CD;
756     P = param.P;
757     nu = param.nu;
758     CR = param.CR;
759
760     % store states to meaningful variables
761     x_pos = x(1);
762     xdot = x(2);
```

```
763     y_pos = x(3);
764     ydot = x(4);
765     z_pos = x(5);
766     zdot = x(6);
767 
768     % determine radial distance and relative velocity to atmosphere
769     r = sqrt(x_pos^2 + y_pos^2 + z_pos^2);
770     v_rel = [xdot; ydot; zdot] - cross([0; 0; w_Earth], [x_pos; y_pos; z_pos]);
771     v_mag = sqrt(v_rel(1)^2 + v_rel(2)^2 + v_rel(3)^2);
772 
773     % calculate drag and solar pressure forces
774     dens = rho_ref*exp(-dens_scale*(r - h0));
775     drag = 0.5*dens*CD*A_m*v_mag*v_rel*1000; % multiply by 1000 to convert from km to
        m
776     solar = -P*nu*A_m*CR*simple_sun_vec(t);
777 
778     dxdt(1) = xdot;
779     dxdt(2) = -mu*x_pos/r^3 - drag(1) - solar(1);
780     dxdt(3) = ydot;
781     dxdt(4) = -mu*y_pos/r^3 - drag(2) - solar(2);
782     dxdt(5) = zdot;
783     dxdt(6) = -mu*z_pos/r^3 - drag(3) - solar(3);
784     dxdt = dxdt';
785 end
786 
787 function sun_vec = simple_sun_vec(t)
788     years2days = 365.25; % days in a year
789     omega_sun = 2*pi / (years2days*24*3600); % rotation around sun (rad/s)
790 
791     theta = omega_sun*t; % rotation around sun (rad)
792     sun_vec = [cos(theta); sin(theta); 0];
793 end
794 
795 % low fidelity dynamics function for model mismatch
796 function dxdt = ODE_Low_Fidelity(t, x, mu, Q, n_sat, EKF_UKF)
```

```
797     % store states to meaningful variables
798     x_pos = x(1);
799     xdot = x(2);
800     y_pos = x(3);
801     ydot = x(4);
802     z_pos = x(5);
803     zdot = x(6);
804 
805     r = sqrt(x_pos^2 + y_pos^2 + z_pos^2);
806     dxdt(1) = xdot;
807     dxdt(2) = -mu*x_pos/r^3;
808     dxdt(3) = ydot;
809     dxdt(4) = -mu*y_pos/r^3;
810     dxdt(5) = zdot;
811     dxdt(6) = -mu*z_pos/r^3;
812     dxdt = dxdt';
813 
814     if EKF_UKF == 1
815         F = Dynamics_Jacobian(x(1:n_sat),mu);
816         P = reshape(x(n_sat + 1:end),n_sat,n_sat);
817         Pdot = F*P + P*F' + Q;
818         dxdt = [dxdt; reshape(Pdot,n_sat^2,1)];
819     end
820 end
821 
822 % dynamics jacobian matrix calculation
823 function F = Dynamics_Jacobian(state, mu)
824     state = state(:);
825     % store states to meaningful variables
826     x = state(1);
827     xdot = state(2);
828     y = state(3);
829     ydot = state(4);
830     z = state(5);
831     zdot = state(6);
```

```
832
833     r = sqrt(x^2 + y^2 + z^2);
834
835     F = [            0             1               0             0               0
                    0;
836          -mu*(1/r^3 - 3*x^2/r^5)   0          mu*3*x*y/r^5        0          mu*3*x*z/r
             ^5         0;
837                      0             0               0             1               0
                                     0;
838               mu*3*x*y/r^5         0    -mu*(1/r^3 - 3*y^2/r^5)   0          mu*3*y*z/r
                  ^5         0;
839                      0             0               0             0               0
                                     1;
840               mu*3*x*z/r^5         0          mu*3*y*z/r^5        0    -mu*(1/r^3 - 3*z
                  ^2/r^5)    0];
841 end
842
843 function H = Meas_Jacobian(x_sat, r_gs_ECEF, t, w_Earth, T_t, el_low, el_high,
    meas_model)
844     % extract satellite position and velocity
845     r_sat = [x_sat(1); x_sat(3); x_sat(5)];
846     v_sat = [x_sat(2); x_sat(4); x_sat(6)];
847
848     % ground station position and velocity in ECI
849     r_gs_ECI = ECEF2ECI(r_gs_ECEF, t, w_Earth);
850     v_gs_ECI = cross([0; 0; w_Earth], r_gs_ECI);
851
852     % relative position and velocity in ECI
853     r_rel = r_sat - r_gs_ECI;
854     v_rel = v_sat - v_gs_ECI;
855
856     r_rel_ECEF = ECI2ECEF(r_rel, t, w_Earth);
857
858     rho = norm(r_rel);
859     rho_dot = r_rel'*v_rel/rho;
```

```
860
861     % convert relative ECEF position to topocentric frame [east, north, up]
862     r_t_vec = T_t*r_rel_ECEF;
863
864     x_t = r_t_vec(1); % east
865     y_t = r_t_vec(2); % north
866     z_t = r_t_vec(3); % up
867     r_t = norm(r_t_vec);
868     r_xy = sqrt(x_t^2 + y_t^2);
869
870     if meas_model == 1
871         % Use numerical differentiation for angular measurements
872         eps = 1e-9;
873         n_state = length(x_sat);
874         n_meas = 2;
875         H = zeros(n_meas, n_state);
876
877         % calculate derivate using central differencing
878         for i = 1:n_state
879             x_pert_plus = x_sat;
880             x_pert_minus = x_sat;
881             x_pert_plus(i) = x_pert_plus(i) + eps;
882             x_pert_minus(i) = x_pert_minus(i) - eps;
883             z_pert_plus = meas_func(x_pert_plus, r_gs_ECEF, t, T_t, w_Earth, 1, el_low
                , el_high);
884             z_pert_minus = meas_func(x_pert_minus, r_gs_ECEF, t, T_t, w_Earth, 1,
                el_low, el_high);
885
886             % Handle angle wrapping for azimuth
887             dz = z_pert_plus - z_pert_minus;
888             if abs(dz(2)) > pi
889                 dz(2) = dz(2) - sign(dz(2))*2*pi;
890             end
891
892             H(:,i) = dz/(2*eps);
```

```
        end
    elseif meas_model == 2
        H = [               r_rel(1)/rho                      0
            r_rel(2)/rho                      0                      r_rel(3)/rho
                                  0;
             (v_rel(1) - rho_dot*(r_rel(1)/rho))/rho   r_rel(1)/rho   (v_rel(2) -
                rho_dot*(r_rel(2)/rho))/rho   r_rel(2)/rho   (v_rel(3) - rho_dot*(
                r_rel(3)/rho))/rho   r_rel(3)/rho];
    else
        % Use numerical differentiation for angular measurements
        eps = 1e-9;
        n_state = length(x_sat);
        n_meas = 2;
        H_angles = zeros(n_meas, n_state);

        % calculate derivate using central differencing
        for i = 1:n_state
            x_pert_plus = x_sat;
            x_pert_minus = x_sat;
            x_pert_plus(i) = x_pert_plus(i) + eps;
            x_pert_minus(i) = x_pert_minus(i) - eps;
            z_pert_plus = meas_func(x_pert_plus, r_gs_ECEF, t, T_t, w_Earth, 1, el_low
                , el_high);
            z_pert_minus = meas_func(x_pert_minus, r_gs_ECEF, t, T_t, w_Earth, 1,
                el_low, el_high);

            % Handle angle wrapping for azimuth
            dz = z_pert_plus - z_pert_minus;
            if abs(dz(2)) > pi
                dz(2) = dz(2) - sign(dz(2))*2*pi;
            end

            H_angles(:,i) = dz/(2*eps);
        end

```

```
        H_range_rate = [               r_rel(1)/rho                        0
                                    r_rel(2)/rho                        0
                                    r_rel(3)/rho                        0;
                          (v_rel(1) - rho_dot*(r_rel(1)/rho))/rho   r_rel(1)/rho   (
                              v_rel(2) - rho_dot*(r_rel(2)/rho))/rho   r_rel(2)/rho   (
                              v_rel(3) - rho_dot*(r_rel(3)/rho))/rho   r_rel(3)/rho];
        H = [H_range_rate; H_angles];
    end
end

% measurement model with logic to select chosen model
function z = meas_func(x_sat, r_antenna, t, T_t, w_Earth, meas_model, low, high, v)
    % handle if noise not given
    if nargin < 9
        v_range_i = 0;
        v_range_rate_i = 0;
        v_el_i = 0;
        v_az_i = 0;
    else
        % extract measurement noise terms
        v_range_i = v(1);
        v_range_rate_i = v(2);
        v_el_i = v(3);
        v_az_i = v(4);
    end

    % convert satellite position from ECI to ECEF
    r_sat_ECI = [x_sat(1); x_sat(3); x_sat(5)];
    r_sat_ECEF = ECI2ECEF(r_sat_ECI, t, w_Earth);

    v_sat_ECI = [x_sat(2); x_sat(4); x_sat(6)];

    % calculate relative velocity in ECI
    r_antenna_ECI = ECEF2ECI(r_antenna, t, w_Earth);
    v_antenna_ECI = cross([0; 0; w_Earth], r_antenna_ECI);
```

```
953
954     % calculate relative positon and velocity of satellite to ground station
955     r_rel_ECEF = r_sat_ECEF - r_antenna;
956     r_rel_ECI = r_sat_ECI - r_antenna_ECI;
957     r_rel = norm(r_rel_ECEF);
958     v_rel_ECI = v_sat_ECI - v_antenna_ECI;
959
960     % convert relative ECEF position to topocentric frame [east, north, up]
961     r_t_vec = T_t*r_rel_ECEF;
962
963     x_t = r_t_vec(1); % east
964     y_t = r_t_vec(2); % north
965     z_t = r_t_vec(3); % up
966     r_t = norm(r_t_vec);
967
968     % calculate range and angle measurements
969     z_k_el = asin(z_t/r_t);
970
971     % set measurement limits
972     % if z_k_el < low || z_k_el > high
973     %     z_k_range = NaN;
974     %     z_k_range_rate = NaN;
975     %     z_k_el = NaN;
976     %     z_k_az = NaN;
977     % else
978         z_k_range = r_rel + v_range_i;
979         z_k_range_rate = r_rel_ECI'*v_rel_ECI/r_rel + v_range_rate_i;
980         z_k_el = asin(z_t/r_t) + v_el_i;
981         z_k_az = wrapTo2Pi(atan2(x_t,y_t) + v_az_i);
982     % end
983
984     % combine measurements based on chosen measurement model
985     if meas_model == 1
986         z = [z_k_el; z_k_az];
987     elseif meas_model == 2
```

```
        z = [z_k_range; z_k_range_rate];
    else
        z = [z_k_range; z_k_range_rate; z_k_el; z_k_az];
    end
end

% calculates sigma points and mean and covariance weights for SUT
function [sig_points, wm, wc] = SigPoints(P, m, alpha, beta, kappa, n)
    % SUT weighting parameters
    lambda = alpha^2*(n + kappa) - n; % weighting parameter
    S = chol(P,'lower'); % square root factor of the covariance matrix

    % calculate sigma points
    sig_points = zeros(n,2*n + 1);
    sig_points(:,1) = m;

    for i = 2:(n + 1)
        sig_points(:,i) = m + sqrt(n + lambda)*S(:,i - 1);
        sig_points(:,i + n) = m - sqrt(n + lambda)*S(:,i - 1);
    end

    % determine weighting terms
    w_0m = lambda/(n + lambda);
    w_0c = lambda/(n + lambda) + (1 - alpha^2 + beta);
    w_im = 1/(2*(n + lambda));
    w_ic = 1/(2*(n + lambda));

    % combine weighting terms into corresponding arrays
    wm = [w_0m, repmat(w_im, 1, 2*n)];
    wc = [w_0c, repmat(w_ic, 1, 2*n)];
end

% ECEF to topocentric frame [x_t = east, y_t = north, z_t = up]
function T = ECEF2Topo(phi, lambda)
    C = @(angle) cos(angle);
```

```
    S = @(angle) sin(angle);
    T = [    -S(lambda),          C(lambda),              0;
         -S(phi)*C(lambda),   -S(phi)*S(lambda),     C(phi);
          C(phi)*C(lambda),     C(phi)*S(lambda),    S(phi)];
end

% simple ECEF to ECI coordinate transformation assuming Earth spins at
% constant angular velocity
function r_ECEF = ECI2ECEF(r_ECI, t, w)
    theta = w*t; % simple Earth rotation
    Rz = [cos(theta),   -sin(theta),   0;
          sin(theta),    cos(theta),   0;
              0,             0,        1];
    r_ECEF = Rz*r_ECI;
end

function r_ECI = ECEF2ECI(r_ECEF, t, omegaE)
    theta = omegaE*t;
    Rz = [cos(theta),   -sin(theta),   0;
          sin(theta),    cos(theta),   0;
              0,             0,        1];
    r_ECI = Rz'*r_ECEF;
end

% rotation matrix about axis 1 for 3-1-3 sequence
function R = DCM1(angle)
    R = [1       0              0;
         0   cos(angle)   -sin(angle);
         0   sin(angle)    cos(angle)];
end

% rotation matrix about axis 3 for 3-1-3 sequence
function R = DCM3(angle)
    R = [cos(angle)   -sin(angle)   0;
         sin(angle)    cos(angle)   0;
```

```
                0               0          1];
end
```

References

[1] Mashiku, A., Garrison, J. L., and Carpenter, J. R., "Statistical orbit determination using the particle filter for incorporating Non-Gaussian uncertainties - NASA technical reports server (NTRS)," , Aug 2012. URL https://ntrs.nasa.gov/citations/20120016941.

[2] Tapley, B. D., Schutz, B. E., and Born, G. H., *Statistical Orbit Determination*, Elsevier: Academic Press, 2004.

[3] Maldonado, J. C., Jenkin, A. B., and McVey, J. P., "Probabilistic assessment of disposal orbit lifetime for CubeSat rideshares on resonant decaying geosynchronous transfer orbits," *Journal of Space Safety Engineering*, Vol. 11, No. 3, 2024, pp. 403–410. https://doi.org/https://doi.org/10.1016/j.jsse.2024.07.009, URL https://www.sciencedirect.com/science/article/pii/S2468896724001137, space Debris: Update to The State of the Art.

[4] Montenbruck, O., and Gill, E., *Satellite Tracking and ObservationModels*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 193–232. https://doi.org/10.1007/978-3-642-58351-3_6, URL https://doi.org/10.1007/978-3-642-58351-3_6.

[5] Montenbruck, O., and Gill, E., *Satellite orbits*, 2000. https://doi.org/10.1007/978-3-642-58351-3.

[6] , ????