ENAE 441 - 0101

Final Project: Extended Kalman Filter

Due on December 20$^{\text{th}}$, 2025 at 09:30 AM

Dr. Martin, 09:30 AM

Vai Srivastava

December 20$^{\text{th}}$, 2025

Overview

You are working at NASA GSFC on the navigation team for an Earth orbiting science satellite launched into orbit on a Falcon-9. SpaceX designed its launch to deliver the payload into low Earth orbit with reference coordinates of

$$\boldsymbol{X}_{\text{œ}} = \begin{bmatrix} a \\ e \\ i \\ \omega \\ \Omega \\ \theta \end{bmatrix} = \begin{bmatrix} 7 \times 10^3\,\text{km} \\ 0.2 \\ 45° \\ 0° \\ 270° \\ 78.75° \end{bmatrix}$$

however it's the navigation's team responsibility to verify this orbit and/or determine any errors in its delivery. To accomplish this, the navigation team has access to range and range-rate measurements from the spacecraft provided by the Deep Space Network (DSN). The latitude and longitude (ϕ, λ) of the three DSN ground stations are provided below:

- DSN # 0: Goldstone, USA: $(35.297°, -116.914°)$
- DSN # 1: Madrid, Spain: $(40.4311°, -4.248°)$
- DSN # 2: Canberra, Australia: $(-35.4023°, 148.9813°)$

These ground stations are positioned on a spherical Earth of radius 6378.137 km , with a rotation rate of $\omega_{\mathcal{E}/\mathcal{N}} = 7.292\,115 \times 10^{-5}\,\frac{\text{rad}}{\text{s}}$ and a Local Sidereal Time of $\gamma_0 = 0°$.
`Project-Measurements-Easy.npy` contains the range ρ and range-rate $\dot{\rho}$ measurements alongside additional information. Specifically, each row in the datafile is formatted as:

$$[t, i, \rho, \dot{\rho}]$$

where t corresponds with the time the measurement was received and i corresponds to the ground station index (as labeled above).
As best as you are aware, the spacecraft's motion is governed by the following differential equation:

$$\ddot{\boldsymbol{r}} = -\frac{\boldsymbol{\mu}}{r^3} \cdot \boldsymbol{r}$$

where $\boldsymbol{r}$ is the position of the spacecraft in the inertial frame. Similarly, the measurements provided can be computed using the following equations:

$$\begin{aligned} \boldsymbol{\rho} &= \boldsymbol{r} - \boldsymbol{R}_{\text{site},i} \\ \rho &= \|\boldsymbol{r} - \boldsymbol{R}_{\text{site},i}\| \\ \dot{\rho} &= \frac{\boldsymbol{\rho}\left(\dot{\boldsymbol{r}} - \dot{\boldsymbol{R}}_{\text{site},i}\right)}{\rho} \\ \dot{\boldsymbol{R}}_{\text{site},i} &= \boldsymbol{\omega}_{\mathcal{E}/\mathcal{N}} \times \boldsymbol{R}_{\text{site},i} \end{aligned}$$

where $\boldsymbol{R}_{\text{site},i}$ is the location of the DSN station also in the inertial frame. Note that the measurements provided by the DSN have some intrinsic noise which can be modeled as Gaussian white noise. Explicitly, the noise in the range is characterized by a variance of $1\,\text{m}^2$, and by $1\,\frac{\text{cm}^2}{\text{s}^2}$ variance in the range-rate.
Using your knowledge of the spacecraft's dynamics and the measurements provided by the DSN, generate a report which culminates in an estimate of the spacecraft's state over time. To help facilitate your progress, please generate your report in the following order, answering the following intermediate questions:

Problem 1: Problem Setup

a. Express the non-linear system in continuous time state-space form, clearly defining the vectors $\boldsymbol{f}(X(t))$ and $\boldsymbol{h}(X(t))$

b. Define the linearized dynamics and measurement matrices $A(t)$ and $C(t)$.

c. Show how these matrices are converted to their discrete time forms F_k and H_k. Recall F_k is the state transition matrix $\boldsymbol{\Phi}(t_j, t_i)$ which requires integration.

d. Define your noise matrices Q_k and R_k, and discuss their relationship to the aforementioned system of equations.

e. Plot the measurements as a function of time.

Solution

Part A

First, we use the inertial Cartesian state:

$$X(t) \equiv \begin{bmatrix} r(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} \end{bmatrix}^T \in \mathbb{R}^6, \quad r, v \in \mathbb{R}^3$$

And the two-body point mass model:

$$\dot{r} = v, \quad \dot{v} = -\mu \frac{r}{\|r\|^3}$$

Thus,

$$f(X, t) = \begin{bmatrix} v \\ -\mu \frac{r}{\|r\|^3} \end{bmatrix}$$

Now, each station's ECEF position is:

$$R_{\text{ECEF},i} = R_E \begin{bmatrix} \cos(\phi_i)\cos(\lambda_i) \\ \cos(\phi_i)\sin(\lambda_i) \\ \sin(\phi_i) \end{bmatrix}$$

As Earth rotates about inertial $+\hat{z}$ with $\omega_E = \omega_{\mathcal{E}/\mathcal{N}}$, siderial angle is:

$$\gamma(t) = \gamma_0 + \omega_E t$$

Rotating to get the Site positions:

$$R_{\text{site},i} = R_3(\gamma(t))\, R_{\text{ECEF},i}, \quad \dot{R}_{\text{site},i}(t) = \omega \times R_{\text{site},i}(t), \quad \omega = \begin{bmatrix} 0 \\ 0 \\ \omega_E \end{bmatrix}$$

Now, each station's LOS vector is:

$$\rho_i(t) = r(t) - R_{\text{site},i}(t), \quad \rho(t) = \|\rho_i(t)\|, \quad \hat{\rho}(t) = \frac{\rho_i(t)}{\rho(t)}$$

Range-rate:

$$\dot{\rho}(t) = \frac{\rho_i^T \left(v - \dot{R}_{\text{site},i}\right)}{\rho} = \hat{\rho}^T \left(v - \dot{R}_{\text{site},i}\right)$$

Problem 1 continued on next page...

Measurements including noise:

$$y(t) = h(X, t, i) + v(t)\,, \quad v(t) \sim \mathcal{N}(0, R)$$

Part B

With δX as a small perturbation about a refernece trajectory $\bar{X}(t)$,

$$\delta\dot{X}(t) = A(t)\,\delta X(t)\,, \quad \delta y(t) = C(t)\,\delta X(t) + v(t)$$

$A(t)$ and $C(t)$ are Jacobians:

$$A(t) = \left.\frac{\partial f}{\partial X}\right|_{\bar{X}(t)}, \quad C(t) = \left.\frac{\partial h}{\partial X}\right|_{\bar{X}(t),t,i}$$

Dynamics Jacobian $A(t)$:
With $r = \bar{r}(t), v = \bar{v}(t), r = \|r\|$,

$$A(t) = \begin{bmatrix} \frac{\partial \dot{r}}{\partial r} & \frac{\partial \dot{r}}{\partial v} \\ \frac{\partial \dot{v}}{\partial r} & \frac{\partial \dot{v}}{\partial v} \end{bmatrix} = \begin{bmatrix} 0_{3\times3} & I_{3\times3} \\ -\mu\left(\frac{1}{r^3} I_{3\times3} - \frac{3}{r^5} r r^T\right) & 0_{3\times3} \end{bmatrix}$$

Measurement Jacobian $C(t)$:
For a measurement taken at time t from station i,

$$\rho = r - R_{\text{site},i}(t), \quad \rho = \|\rho\|, \quad u = \hat{\rho} = \rho/\rho, \quad v_{\text{rel}} = v - \dot{R}_{\text{site},i}(t)$$

Range row:

$$\frac{\partial \rho}{\partial r} = u^T, \quad \frac{\partial \rho}{\partial v} = 0_{1\times3}$$

Range-rate row:

$$\dot{\rho} = u^T v_{\text{rel}}$$

Using $\frac{\partial u}{\partial r} = \frac{1}{\rho}\left(I - u u^T\right)$ (since $u = \rho/\|\rho\|$ and ρ depends linearly on r),

$$\frac{\partial \dot{\rho}}{\partial v} = u^T, \quad \frac{\partial \dot{\rho}}{\partial r} = v_{\text{rel}}^T \frac{\partial u}{\partial r} = \frac{1}{\rho} v_{\text{rel}}^T \left(I - u u^T\right)$$

Thus,

$$C(t) = \begin{bmatrix} u^T & 0_{1\times3} \\ \frac{1}{\rho} v_{\text{rel}}^T \left(I - u u^T\right) & u^T \end{bmatrix}$$

Part C

Measurements occur at times t_k (not necessarily uniform), meaning the EKF-style discrete model is:

$$X_{k+1} = \varphi\left(X_k, t_k, t_{k+1}\right) + w_k, \quad y_k = h\left(X_k, t_k, i_k\right) + v_k$$

The discrete linearized mapping for perturbations is:

$$\delta X_{k+1} = F_k \delta X_k, \quad F_k \equiv \Phi\left(t_{k+1}, t_k\right)$$

$\boldsymbol{\Phi}$ is obtained by integrating the variational equation alongside the state:

$$\dot{\Phi}\left(t, t_k\right) = A(t)\Phi\left(t, t_k\right), \quad \Phi\left(t_k, t_k\right) = I_{6\times6},$$

and then setting:

$$F_k = \Phi\left(t_{k+1}, t_k\right)$$

At measurement time t_k and station index i_k,

$$H_k \equiv C(t_k) = \left.\frac{\partial h}{\partial X}\right|_{\bar{X}(t_k),t_k,i_k}$$

Part D

Measurement noise is simply:

$$R_k = \begin{bmatrix} \sigma_\rho^2 & 0 \\ 0\sigma_{\dot{\rho}}^2 & \end{bmatrix}$$

Process noise is used to account for unmodeled accelerations. Here we choose additive white acceleration noise as our process noise model:

$$\dot{r} = v, \quad \dot{v} = -\mu\frac{r}{\|r\|^3} + w_a(t), \quad w_a(t) \sim \mathcal{N}\left(0, Q_c\right),$$

with $Q_c = q_a I_{3\times3}$, and simplifying to:

$$\dot{X} = f(X,t) + G w_a(t), \quad G = \begin{bmatrix} 0_{3\times3} \\ I_{3\times3} \end{bmatrix}$$

Then the discrete process noise over $[t_k, t_{k+1}]$ is

$$Q_k = \int_{t_k}^{t_{k+1}} \Phi\left(t_{k+1}, \tau\right) G Q_c G^T \Phi\left(t_{k+1}, \tau\right)^T d\tau$$

Part E

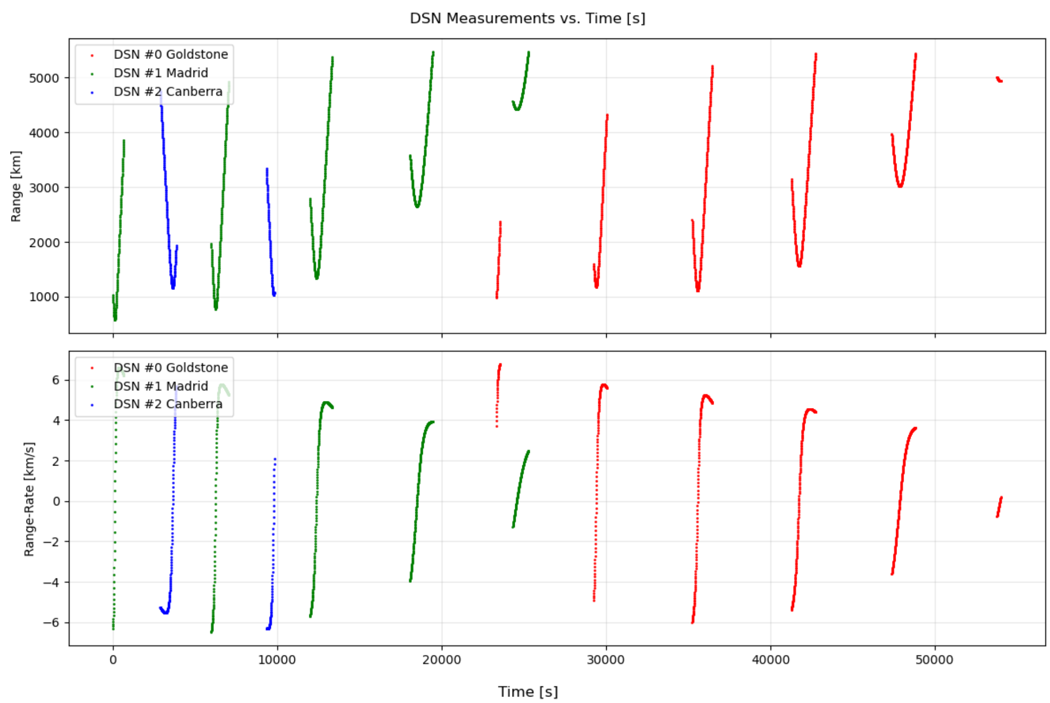

Figure 1: DSN Range [km] vs. Time [s]

Problem 2: Plan Filter Implementation

Provide pseudocode from which you will base your extended Kalman filter implementation. Highlight the major steps in your algorithm and any noteworthy modifications or subtle details required for this problem that you want the grades to be aware of. Be comprehensive, as this is what the grading team will primarily reference if the results/plots don't quite look right.

Solution

```
Inputs:
    data.t[0..N-1], data.i[0..N-1], data.rho[0..N-1], data.drho[0..N-1]
    station lat/lon: (phi[i], lam[i]), i=0..2
    constants: mu, RE, omegaE, gamma0
    noise: R = diag(sig_rho^2, sig_drho^2)   (in FILTER UNITS)
           Q-model params: q_a (white accel PSD) OR set Qk = 0
    initial: X_plus (6x1), P_plus (6x6)

Precompute:
    for each station i:
        R_ecef[i] = RE * [cos(phi_i)cos(lam_i), cos(phi_i)sin(lam_i), sin(phi_i)]^T

Helpers:
    site_eci(i, t):
        gamma = gamma0 + omegaE*t
        R_site = R3(gamma) * R_ecef[i]
        Rdot_site = omega x R_site # (omega = [0,0,omegaE]^T)
        return (R_site, Rdot_site)

    dynamics(X):
        r = X[0:3], v = X[3:6]
        rnorm = ||r||
        rdot = v
        vdot = -mu * r / rnorm^3
        return [rdot; vdot]

    A_matrix(X):
        r = X[0:3]; rnorm = ||r||
        I3 = identity(3)
        dadr = -mu*( (1/rnorm^3)*I3 - (3/rnorm^5)*(r*r^T) )
        return [[0, I3],
                [dadr, 0]]

    propagate_state_and_stm(X0, t0, t1):
        Integrate coupled ODEs from t0 -> t1:
        Xdot  = dynamics(X)
        Phidot = A_matrix(X) * Phi
        with initial Phi(t0)=I6
        return (X_minus, Phi)   where Phi = Φ(t1,t0)
```

Problem 2 continued on next page...

```
    meas_and_jacobian(X, i, t):
        (R_site, Rdot_site) = site_eci(i, t)
        r = X[0:3]; v = X[3:6]
        rho_vec = r - R_site
        rho = ||rho_vec||
        u = rho_vec / rho
        v_rel = v - Rdot_site
        drho = u^T * v_rel

        yhat = [rho; drho]

        # Jacobian H = ∂h/∂X (2x6)
        H_rho_r = u^T
        H_rho_v = [0 0 0]

        H_drho_v = u^T
        H_drho_r = (1/rho) * (v_rel^T * (I3 - u*u^T))

        H = [[H_rho_r,  H_rho_v],
             [H_drho_r, H_drho_v]]
        return (yhat, H)

    process_noise_discrete(dt, q_a):
        if q_a == 0:
            return 0_(6x6)
        else:
            I3 = identity(3)
            Q = q_a * [[(dt^3/3)*I3, (dt^2/2)*I3],
                       [(dt^2/2)*I3, (dt)*I3]]
            return Q

Main EKF:
    t_prev = data.t[0]

    for k in 0..N-1:
        t_k = data.t[k]
        i_k = data.i[k]
        y_k = [data.rho[k], data.drho[k]]^T

    # Predict / propagate to t_k
    dt = t_k - t_prev
    if dt > 0:
        (X_minus, Phi) = propagate_state_and_stm(X_plus, t_prev, t_k)
        Qk = process_noise_discrete(dt, q_a)
        P_minus = Phi * P_plus * Phi^T + Qk
    else:
        X_minus = X_plus
        P_minus = P_plus
```

Problem 2 continued on next page...

```
        Phi = I6

    # Measurement prediction at (t_k, i_k)
    (yhat_k, Hk) = meas_and_jacobian(X_minus, i_k, t_k)
    nu = y_k - yhat_k # innovation (2x1)

    # Gain computation
    S = Hk * P_minus * Hk^T + R # (2x2)
    K = P_minus * Hk^T * inv(S) # (6x2)

    # Update (state + Joseph covariance)
    X_plus = X_minus + K * nu
    P_plus = (I6 - K*Hk)*P_minus*(I6 - K*Hk)^T + K*R*K^T

    # Diagnostics
    log residual nu, NIS = nu^T * inv(S) * nu, etc.
    t_prev = t_k

Outputs:
    filtered state estimates X_plus and X_minus over time
    residual history, NIS history, covariance history
```

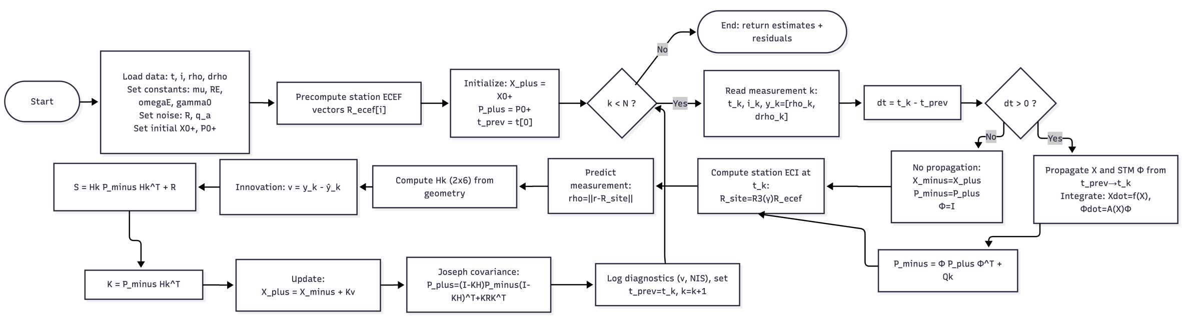

Figure 2: (Pseudo) Code Flowchart

Problem 3: Pure Prediction

a. Specify your choice of $\boldsymbol{x}_0$, P_0, Q_0, and R_0. Explain your reasoning. (For $\boldsymbol{x}_0$, express in cartesian coordinates.)

b. Implement just the prediction step of the extended Kalman Filter (i.e. do not perform measurement updates).

c. Plot the covariance of the state estimate as a function of time, centered about a mean of zero — i.e. $\pm 3\sigma$ bounds taken from $P^-_{\hat{\boldsymbol{x}},k}(t)$. Choose appropriate y-limits for your plot to maximize readability. Explain what you see and if it makes sense.

Solution

Part A

$\boldsymbol{x}_0$ is the nominal spacecraft orbit:

$$\boldsymbol{X}_0^+ \equiv \begin{bmatrix} r_0 \\ v_0 \end{bmatrix}$$

P_0 is chosen to be a diagonal covariance matrix, with the first 3 terms being the range noise uncertainty scaled by the tuning ratio:

$$\sigma_r^2 = \left(\frac{\sigma_\rho}{\sigma_a}\right)^2 = \left(\frac{1\times 10^{-6}}{1\times 10^{-6}}\,[\mathrm{km}]\right)^2$$

and the final 3 terms being the range-rate noise uncertainty scaled by the tuning ratio:

$$\sigma_v^2 = \left(\frac{\sigma_{\dot\rho}}{\sigma_a}\right)^2 = \left(\frac{1\times 10^{-10}}{1\times 10^{-6}}\left[\frac{\mathrm{km}}{\mathrm{s}}\right]\right)^2$$

Q_0 is chosen to be the first-order Wiener discretized continuous white noise acceleration model:

$$Q_k \approx q_a \begin{bmatrix} \frac{\Delta t^3}{3} I_3 & \frac{\Delta t^2}{2} I_3 \\ \frac{\Delta t^2}{2} I_3 & \Delta t I_3 \end{bmatrix}$$

R_0 is chosen to be a diagonal covariance matrix, with the first 3 terms being the range noise uncertainty:

$$\sigma_\rho^2 = \left(1\times 10^{-3}\,[\mathrm{km}]\right)^2$$

and the final 3 terms being the range-rate noise uncertainty

$$\sigma_{\dot\rho}^2 = \left(1\times 10^{-10}\left[\frac{\mathrm{km}}{\mathrm{s}}\right]\right)^2$$

```
X₀⁺ =
[ 4.48544067e+03 -1.26177495e+03  4.48544067e+03  2.15162036e+00
  7.55367318e+00  2.15162036e+00]
P₀⁺ =
[[1.e+00 0.e+00 0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 1.e+00 0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 1.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 1.e-08 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00 1.e-08 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00 0.e+00 1.e-08]]
R₀ =
[[1.e-06 0.e+00]
 [0.e+00 1.e-10]]
```

Problem 3 continued on next page...

Part B

Implementation can be viewed in the Python files embedded at the end of the report.

Part C

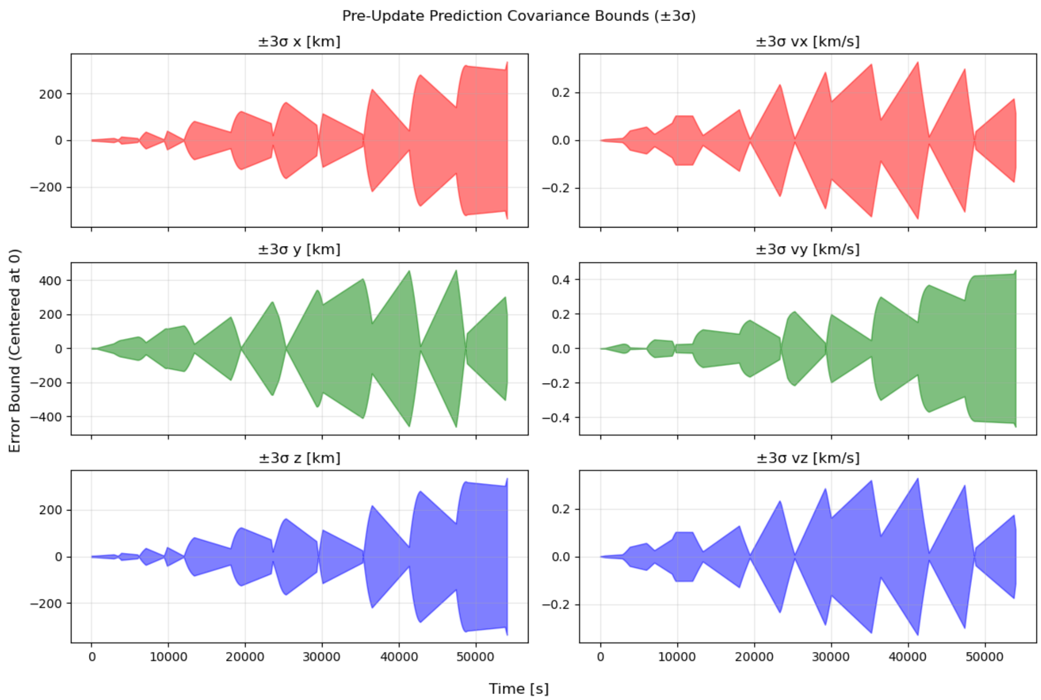

Figure 3: $\pm 3\sigma$ Bounds on Pre- Measurement Covariance $P^-_{\hat{\boldsymbol{x}},k}$ vs. Time t

The plot exhibits sawtooth behavior consistent with a white acceleration process noise model (like the Wiener acceleration we chose). This makes sense, as, without a measurement, the uncertainty can only accumulate. As well, the irregular nature of the plot (with regard to the spacing of each "wedge") is consistent with varying measurement time-gaps.

Problem 4: Measurement Updates

a. Implement the measurement update step of the extended Kalman Filter.

b. Plot the pre- and post-measurement update $\pm 3\sigma$ bounds on the same plot. Explain any differences.

c. Plot the difference between the pre- and post-measurement update state estimates, $\boldsymbol{\mu}^{+}_{\hat{\boldsymbol{x}},k}(t_k) - \boldsymbol{\mu}^{-}_{\hat{\boldsymbol{x}},k}(t_k)$, inside of the pre-measurement covariance $\pm 3\sigma$ bounds $P^{+}_{\hat{\boldsymbol{x}},k}(t_k)$.

Solution

Part A

Implementation can be viewed in the Python files embedded at the end of the report.

Part B

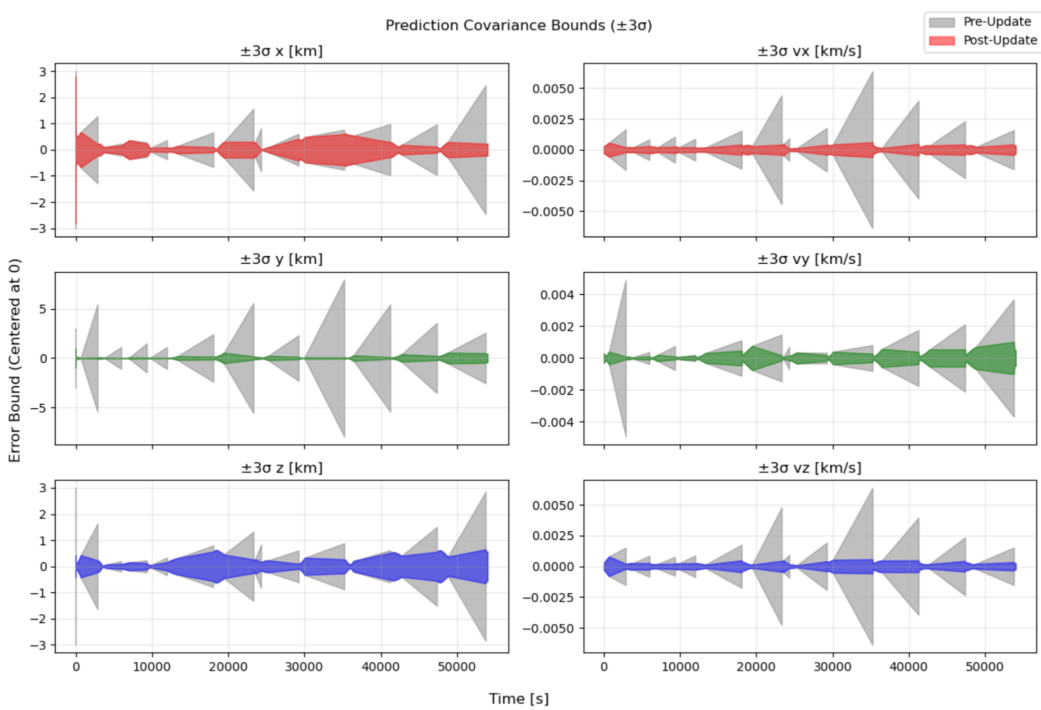

Figure 4: $\pm 3\sigma$ Bounds on Pre- and Post- Measurement Covariance

The post-update bounds are substantially tighter than the pre-update bounds at the same time, because each range/range-rate measurement injects information and reduces uncertainty (via the Kalman gain and Joseph update). The pre-update bounds grow between measurements (propagation), while the post-update bounds exhibit repeated "pull-downs" (corrections) whenever a measurement is processed. Overall, this behavior is consistent with the propagation & update cycle of an EKF.

Part C

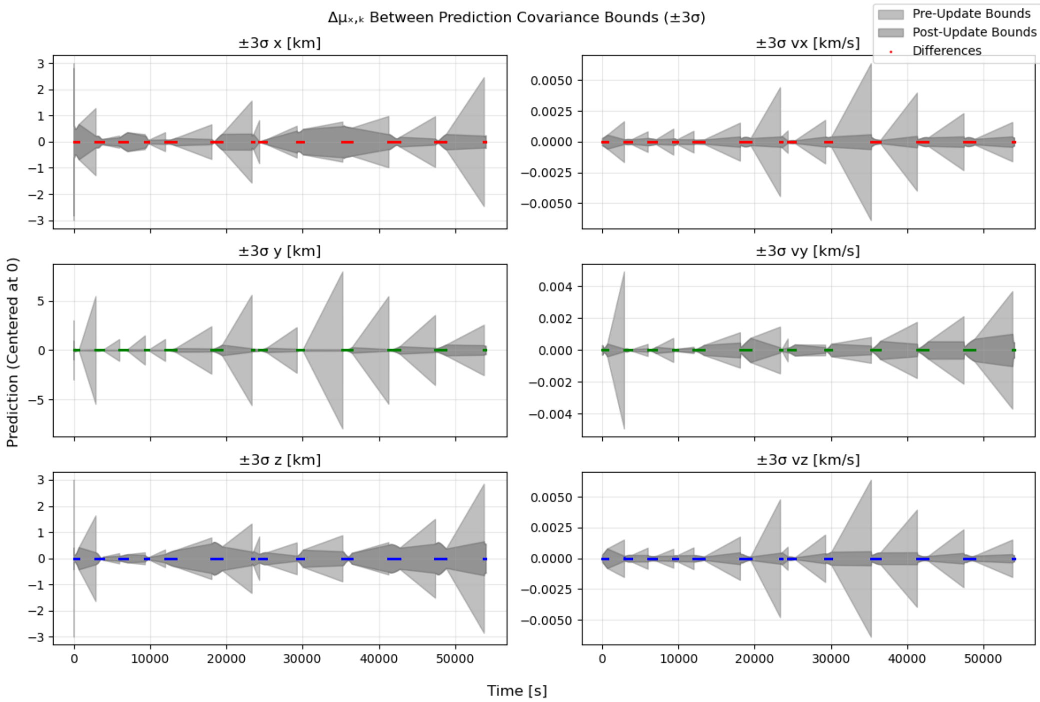

Figure 5: Difference Between Pre- and Post-Measurement Update State Estimates

Problem 5: Filter Solutions

a. With your EKF implemented, run the simulation through to completion and compute the post-fit measurement residuals as a function of time via:

$$\delta \boldsymbol{Y}(t_k) = \boldsymbol{Y}(t_k) - h\Big(\boldsymbol{\mu}^{+}_{\hat{\boldsymbol{x}},k}(t_k)\Big)$$

b. Do your measurement residuals give you confidence that you estimated the state correctly? Why or why not?

c. Plot the estimated state $\boldsymbol{\mu}^{+}_{\hat{\boldsymbol{x}},k}(t_k)$ and its associated $\pm 3\sigma$ bounds taken from $P^{+}_{\hat{\boldsymbol{x}},k}(t)$ as a function of time.

d. Report your best state estimate at the final time-step and its associated uncertainty.

Solution

Part A

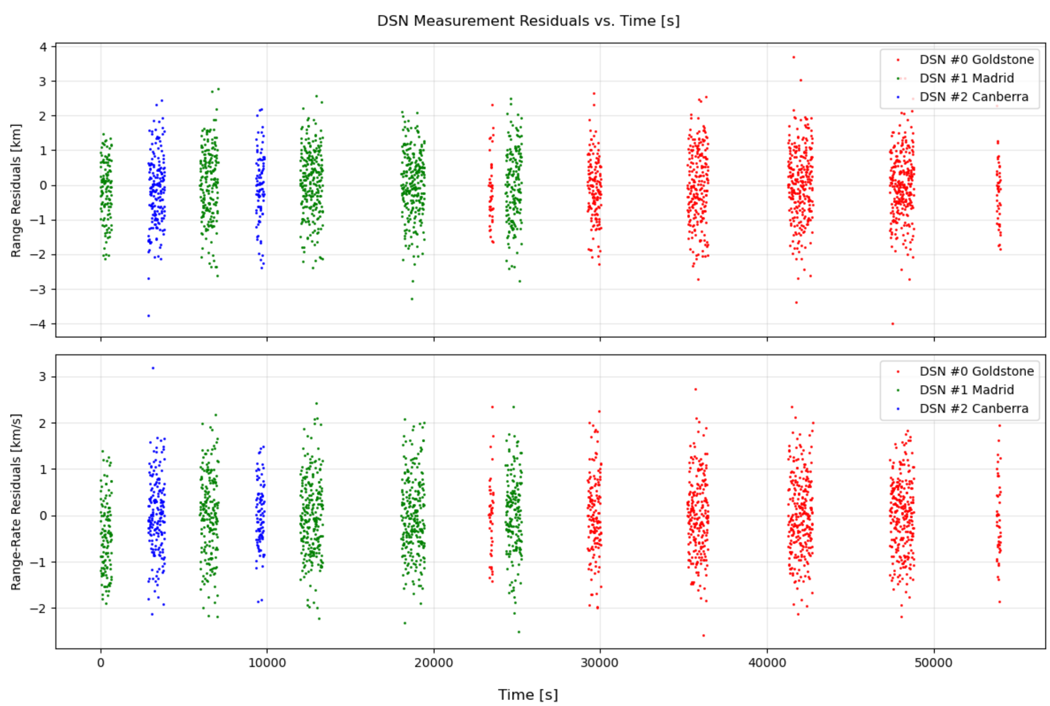

Figure 6: Post-Fit Measurement Residuals $\delta \boldsymbol{Y}$ vs. Time t

Part B

```
Post-fit RMS range residual: 0.970 [m]
Post-fit RMS range-rate residual: 0.817 [cm/s]
Mean pre-fit NIS (df=2): 1.950
```

Yes, the residuals strongly support that the estimated state is correct and that the filter is well-tuned/consistent. There are three key pieces of evidence to support this claim:

- The post-fit residuals plotted in Part A appear zero-mean and noise-like, without obvious drifts or station-dependent biases.
- Quantitatively, the post-fit RMS residuals are the same order of magnitude (and scale) of the measurement noise:

  RMS range residual $\approx 0.970\text{m}$

  RMS range-rate residual $\approx 0.817\frac{\text{cm}}{\text{s}}$
- The mean pre-fit NIS is ≈ 1.950 for $df = 2$, which is close to the expected value (≈ 2) for a statistically consistent filter with 2 measurement dimensions.

Part C

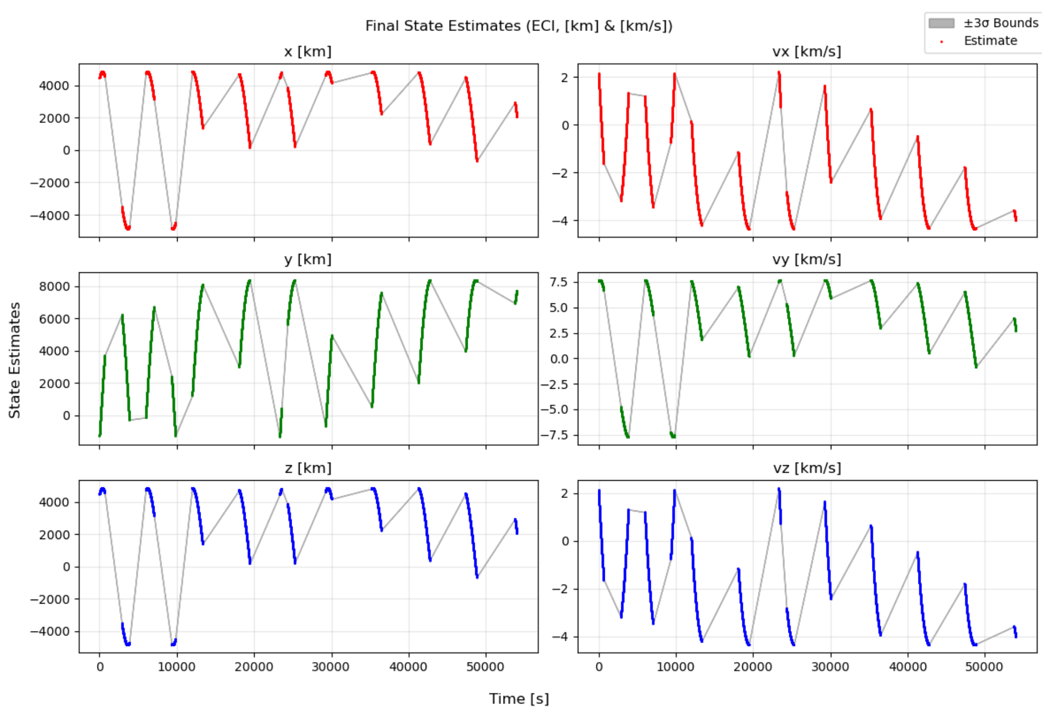

Figure 7: Estimated State $\boldsymbol{\mu}^{+}_{\hat{\boldsymbol{x}},k}$ with $\pm 3\sigma$ Bounds vs. Time t

Part D

```
Final state estimate (ECI, km and km/s):x [km]  :  2050.638271630 ±  0.080274911 (1σ)
y [km]  :  7747.569493734 ±  0.141891722 (1σ)
z [km]  :  2050.672230306 ±  0.186952201 (1σ)
vx [km/s]: -4.010088130 ±  0.000109635 (1σ)
vy [km/s]:  2.701084104 ±  0.000163016 (1σ)
vz [km/s]: -4.009092965 ±  0.000083902 (1σ)
```

Problem 6: Debugging Efforts *(Optional)*

Use this section to outline any of your debugging efforts for if things aren't going your way. This is a good place to earn some partial credit. This should be a "research" log of what experiments you performed and why. A list of guiding questions if your stuck include:

1. Consider how process noise matrix is used in the filter. What happens if there are large gaps between measurements?
2. Consider if the values used in your measurement noise matrix are appropriate. Should these values only reflect the uncertainty in the sensor?
3. If your filter is diverging, does the divergence start at the beginning or mid-way through? What possible reasons exist for either outcome?
4. If you had to define a single scalar metric to evaluate your filter's quality, what would it be, and can you use this to help you determine optimal tuning values?
5. Does plotting your best estimate in a different reference frame or element description help?

Solution

N/A?
I am honestly not sure what to put here, as I was fairly confident in my solution throughout the entirety of my time working on this project. Especially once I computed my result for Problem 5, Part B, any worry that my solution was inadequate was completely gone.
I attribute the ease of development on this project to three things:

1. Thorough structuring and architecting of my planned solution before implementation.
2. Strictly typing all variables, objects, and functions throughout my codebase (thank you Python 3.13).
3. Modularization/functional-decomposition of my codebase.

Problem 7: Challenge Orbit *(Bonus)*

Perform a second analysis with the more difficult dataset `Project-Measurements-Hard.npy`. Use

$$\begin{bmatrix} a \\ e \\ i \\ \omega \\ \Omega \\ \theta \end{bmatrix} = \begin{bmatrix} 7000\,\text{km} \\ 0.6 \\ 45° \\ 180° \\ 0° \\ 45° \end{bmatrix}$$

as your initial guess. Your analysis can amount to your debugging process, and points will be awarded based on how thoughtful your experimentation is and the quality of your solution.

Solution

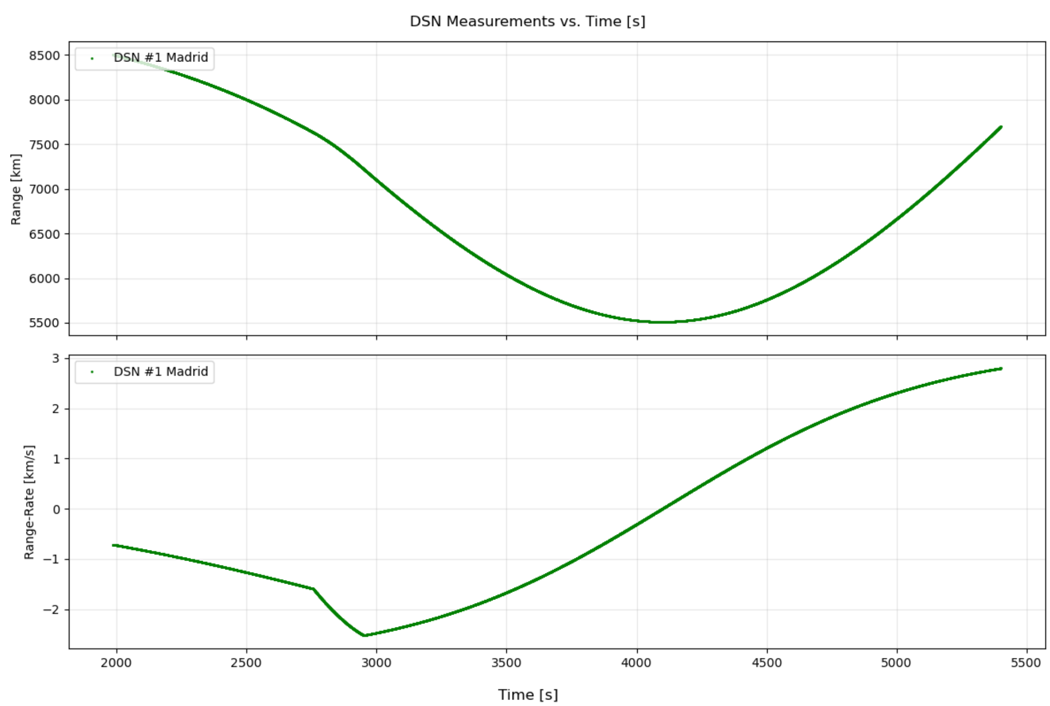

Figure 8: DSN Range [km] vs. Time [s]

```
X₀⁺ =
[ 4.48544067e+03 -1.26177495e+03  4.48544067e+03  2.15162036e+00
  7.55367318e+00  2.15162036e+00]
P₀⁺ =
[[1.e+00 0.e+00 0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 1.e+00 0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 1.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 1.e-08 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00 1.e-08 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00 0.e+00 1.e-08]]
R₀ =
```

```
[[1.e-06 0.e+00]
 [0.e+00 1.e-10]]
```

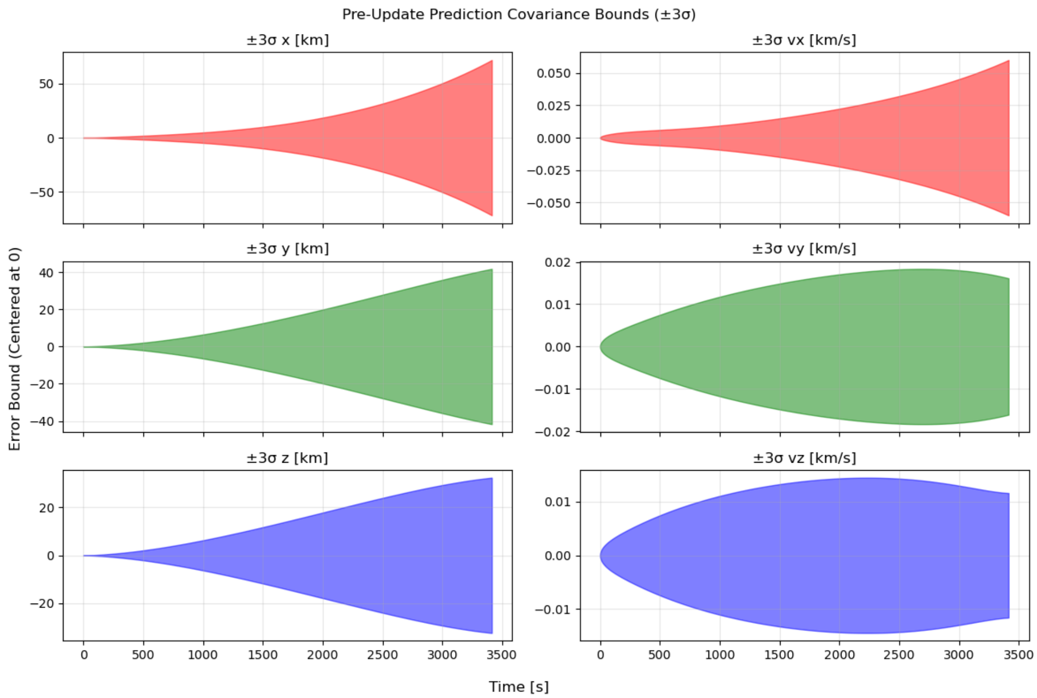

Figure 9: $\pm 3\sigma$ Bounds on Pre- Measurement Covariance $P^{-}_{\hat{\boldsymbol{x}},k}$ vs. Time t

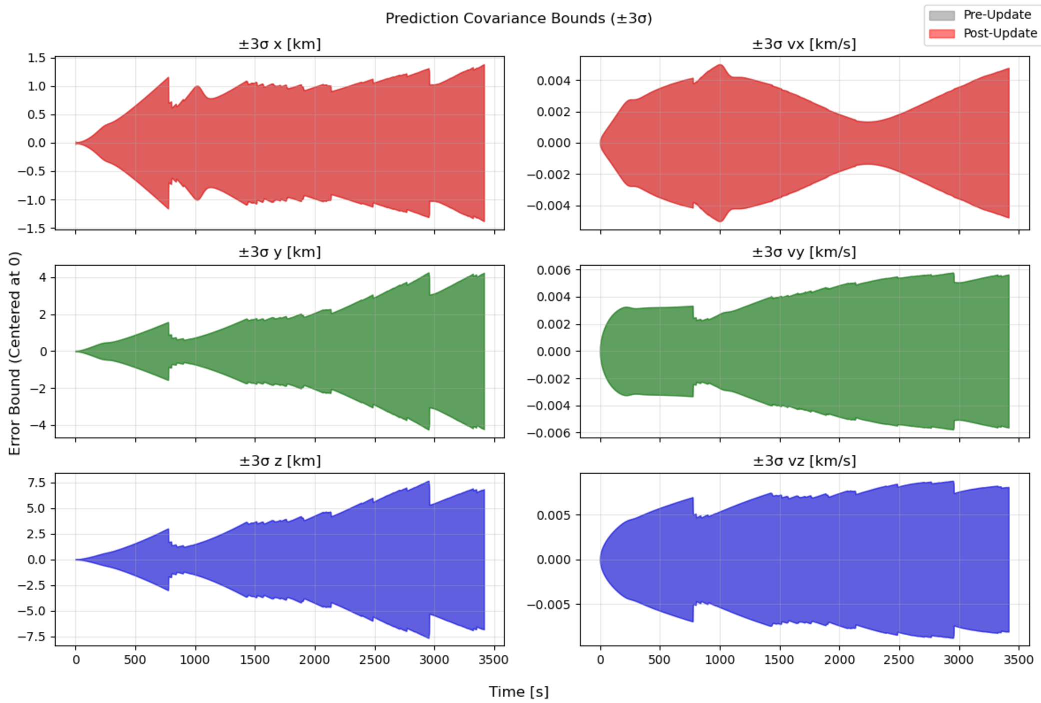

Figure 10: $\pm 3\sigma$ Bounds on Pre- and Post- Measurement Covariance

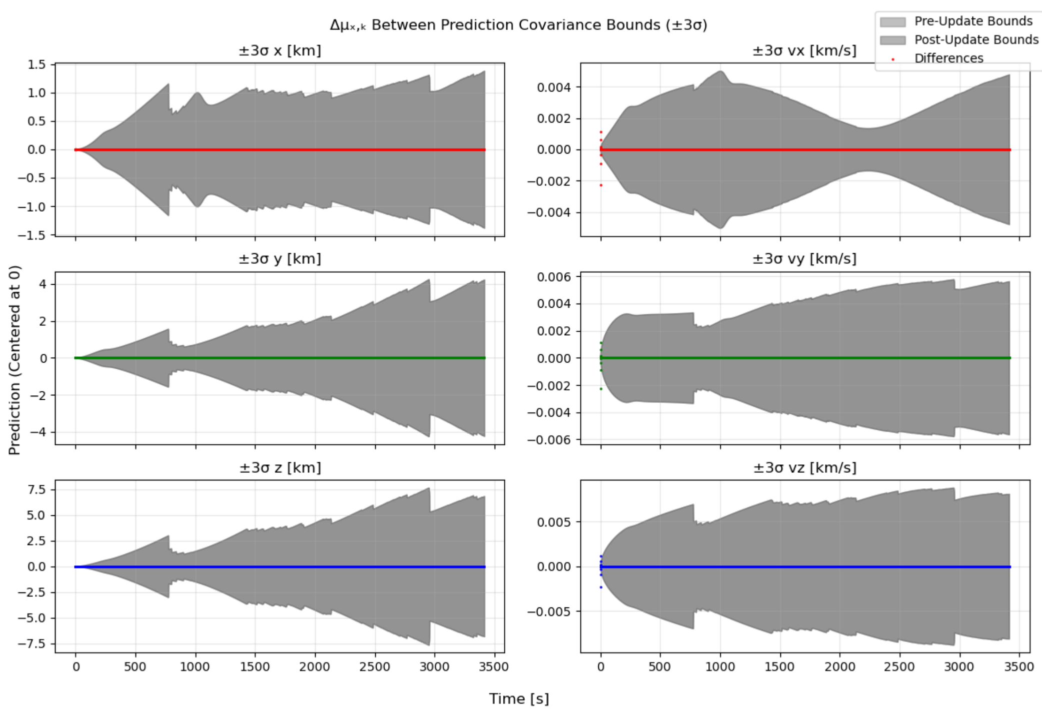

Figure 11: Difference Between Pre- and Post-Measurement Update State Estimates

Figure 12: Post-Fit Measurement Residuals $\delta\boldsymbol{Y}$ vs. Time t

```
Post-fit RMS range residual: 5277.921 [m]
Post-fit RMS range-rate residual: 2306.946 [cm/s]
Mean pre-fit NIS (df=2): 24812892.724
```

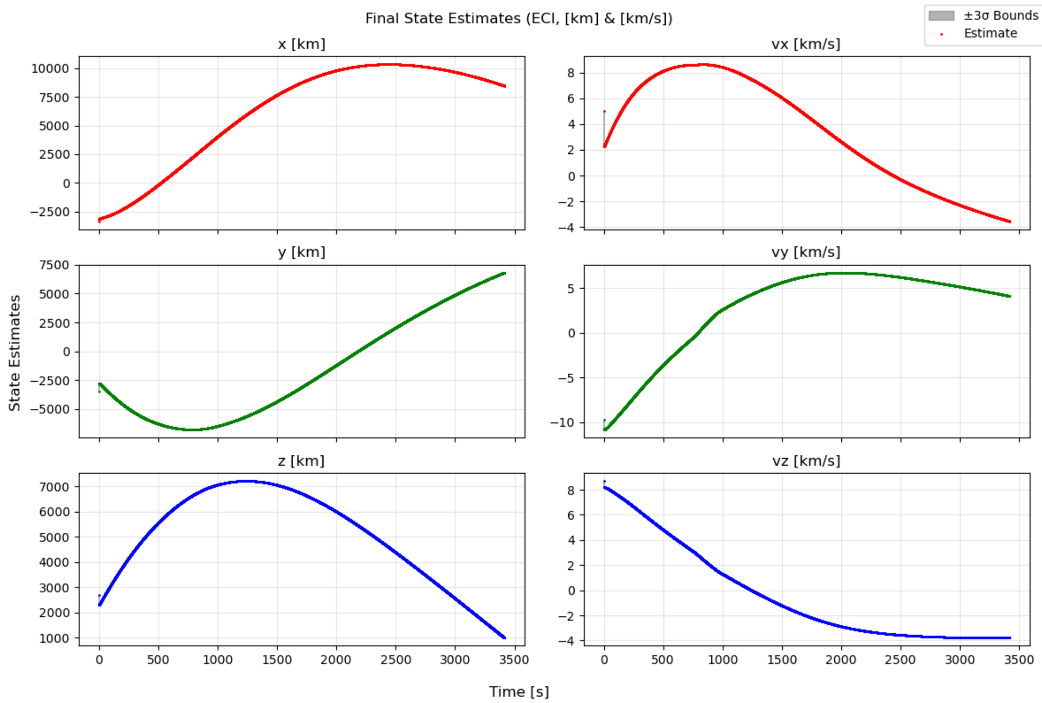

Figure 13: Estimated State $\boldsymbol{\mu}_{\hat{\boldsymbol{x}},k}^{+}$ with $\pm 3\sigma$ Bounds vs. Time t

```
Final state estimate (ECI, km and km/s):x [km]  :  8479.527850461 ±  0.460886411 (1σ)
y [km]  :  6825.434393255 ±  1.412164692 (1σ)
z [km]  :  1007.770267538 ±  2.278929774 (1σ)
vx [km/s]: -3.545201184 ±  0.001596113 (1σ)
vy [km/s]:  4.133262025 ±  0.001876156 (1σ)
vz [km/s]: -3.766752892 ±  0.002697590 (1σ)
```

./src/hard.py

```python
# libs
import matplotlib.pyplot as plt
import numpy as np
import os

# src
from measurements import *
from helpers import *
import p01
import p03
import p04
import p05

def main() -> int:
    # constants
    EPS = 1e-12 # [-]
```

```python
    MU = 398600.4418 # [km^3/s^2]
    RE = 6378.137 # [km]
    OMEGA_E = 7.292115e-5 # [rad/s]
    GAMMA0 = 0.0 # [rad]

    # givens
    oe = [
        7e3, # [km]
        0.6, # [-]
        np.deg2rad(45), # [deg -> rad]
        np.deg2rad(0), # [deg -> rad]
        np.deg2rad(170), # [deg -> rad]
        np.deg2rad(45) # [deg -> rad]
    ]
    stations = {
        0: (np.deg2rad(35.297), np.deg2rad(-116.914)), # [lat, long]
        1: (np.deg2rad(40.4311), np.deg2rad(-4.248)), # [lat, long]
        2: (np.deg2rad(-35.4023), np.deg2rad(148.9813)), # [lat, long]
    }
    # measurement noise
    σ_n = [
        1e-6, # [km^2]
        1e-10 # [km^2/s^2]
    ]
    # process noise tuning
    σ = [
        1e-2, # [km]
        1e-4 # [km/s]
    ]
    σ_a = 1e-4 # [km/s^2]

    # load numpy measurements
    data = numpy_data.load("./data/Project-Measurements-Hard.npy")

    #######
    # p01 #
    #######
    # p01e
    p01.e(data).savefig("./outputs/figures/s07-01e.png")

    #######
    # p03 #
    #######
    # p03a
    X0_plus, P0_plus, R0 = p03.a(oe, σ_n, σ, σ_a, MU)
    with open("./outputs/text/s07-03a.txt", "w", encoding="utf-8") as f:
        f.write(f"X₀⁺ =\n{X0_plus}\n")
        f.write(f"P₀⁺ =\n{P0_plus}\n")
```

```
        f.write(f"R₀ =\n{R0}")
    # p03b
    t_pred, X_minus_hist, P_minus_hist = p03.b(data, X0_plus, P0_plus, σ_a, MU)
    # p03c
    p03.c(t_pred, X_minus_hist, P_minus_hist).savefig("./outputs/figures/s07-03c.png")

    #######
    # p04 #
    #######
    # p04a
    t_pred, X_minus_hist, P_minus_hist, X_plus_hist, P_plus_hist, yhat_minus_hist, yhat_plus_hist, resid_pre_hist, resid_post_hist, nis_pre_hist = p04.a(data, X0_plus, P0_plus, R0, stations, σ_a, MU, RE, OMEGA_E, GAMMA0)
    # p04b
    p04.b(t_pred, X_minus_hist, P_minus_hist, X_plus_hist, P_plus_hist).savefig("./outputs/figures/s07-04b.png")
    # p04c
    p04.c(t_pred, X_minus_hist, P_minus_hist, X_plus_hist, P_plus_hist).savefig("./outputs/figures/s07-04c.png")

    #######
    # p05 #
    #######
    # p05a
    resid_fig = p05.a(data, t_pred, resid_pre_hist, resid_post_hist)
    resid_axs = resid_fig.get_axes()
    for ax in resid_axs:
        ax.set_yscale("log")
    resid_fig.savefig("./outputs/figures/s07-05a.png")
    # p05b
    rms_r_m, rms_rr_cms, nis_mean = p05.b(resid_post_hist, nis_pre_hist)
    with open("./outputs/text/s07-05b.txt", "w", encoding="utf-8") as f:
        f.write(f"Post-fit RMS range residual: {rms_r_m:.3f} [m]\n")
        f.write(f"Post-fit RMS range-rate residual: {rms_rr_cms:.3f} [cm/s]\n")
        f.write(f"Mean pre-fit NIS (df=2): {nis_mean:.3f}\n")
    # p05c
    p05.c(t_pred, X_plus_hist, P_plus_hist).savefig("./outputs/figures/s07-05c.png")
    # p05d
    X_final, σ_final, names = p05.d(X_plus_hist, P_plus_hist)
    with open("./outputs/text/s07-05d.txt", "w", encoding="utf-8") as f:
        f.write("Final state estimate (ECI, km and km/s):")
        for name, X, σ in zip(names, X_final, σ_final):
            f.write(f"{name:8s}: {X: .9f} ± {σ: .9f} (1σ)\n")

    return 0


if __name__ == "__main__":
```

```
    main()
```

Code

See the full source code for this project.

./src/final.py

```python
# libs
import matplotlib.pyplot as plt
import numpy as np
import os

# src
from measurements import *
from helpers import *
import p01
import p03
import p04
import p05

def main() -> int:
    # constants
    EPS = 1e-12 # [-]
    MU = 398600.4418 # [km^3/s^2]
    RE = 6378.137 # [km]
    OMEGA_E = 7.292115e-5 # [rad/s]
    GAMMA0 = 0.0 # [rad]

    # givens
    oe = [
        7e3, # [km]
        0.2, # [-]
        np.deg2rad(45), # [deg -> rad]
        np.deg2rad(0), # [deg -> rad]
        np.deg2rad(270), # [deg -> rad]
        np.deg2rad(78.75) # [deg -> rad]
    ]
    stations = {
        0: (np.deg2rad(35.297), np.deg2rad(-116.914)), # [lat, long]
        1: (np.deg2rad(40.4311), np.deg2rad(-4.248)), # [lat, long]
        2: (np.deg2rad(-35.4023), np.deg2rad(148.9813)), # [lat, long]
    }
    # measurement noise
    σ_n = [
        1e-6, # [km^2]
        1e-10 # [km^2/s^2]
    ]
    # process noise tuning
    σ_a = 1e-6 # [km/s^2]

```

```python
    # load numpy measurements
    data = numpy_data.load("./data/Project-Measurements-Easy.npy")

    #######
    # p01 #
    #######
    # p01e
    p01.e(data).savefig("./outputs/figures/s01e.png")

    #######
    # p03 #
    #######
    # p03a
    X0_plus, P0_plus, R0 = p03.a(oe, σ_n, σ_n, σ_a, MU)
    with open("./outputs/text/s03a.txt", "w", encoding="utf-8") as f:
        f.write(f"X₀⁺ =\n{X0_plus}\n")
        f.write(f"P₀⁺ =\n{P0_plus}\n")
        f.write(f"R₀ =\n{R0}")
    # p03b
    t_pred, X_minus_hist, P_minus_hist = p03.b(data, X0_plus, P0_plus, σ_a, MU)
    # p03c
    p03.c(t_pred, X_minus_hist, P_minus_hist).savefig("./outputs/figures/s03c.png")

    #######
    # p04 #
    #######
    # p04a
    t_pred, X_minus_hist, P_minus_hist, X_plus_hist, P_plus_hist, yhat_minus_hist,
    ↪ yhat_plus_hist, resid_pre_hist, resid_post_hist, nis_pre_hist = p04.a(data, X0_plus,
    ↪ P0_plus, R0, stations, σ_a, MU, RE, OMEGA_E, GAMMA0)
    # p04b
    p04.b(t_pred, X_minus_hist, P_minus_hist, X_plus_hist,
    ↪ P_plus_hist).savefig("./outputs/figures/s04b.png")
    # p04c
    p04.c(t_pred, X_minus_hist, P_minus_hist, X_plus_hist,
    ↪ P_plus_hist).savefig("./outputs/figures/s04c.png")

    #######
    # p05 #
    #######
    # p05a
    p05.a(data, t_pred, resid_pre_hist,
    ↪ resid_post_hist).savefig("./outputs/figures/s05a.png")
    # p05b
    rms_r_m, rms_rr_cms, nis_mean = p05.b(resid_post_hist, nis_pre_hist)
    with open("./outputs/text/s05b.txt", "w", encoding="utf-8") as f:
        f.write(f"Post-fit RMS range residual: {rms_r_m:.3f} [m]\n")
        f.write(f"Post-fit RMS range-rate residual: {rms_rr_cms:.3f} [cm/s]\n")
```

```python
        f.write(f"Mean pre-fit NIS (df=2): {nis_mean:.3f}\n")
    # p05c
    p05.c(t_pred, X_plus_hist, P_plus_hist).savefig("./outputs/figures/s05c.png")
    # p05d
    X_final, σ_final, names = p05.d(X_plus_hist, P_plus_hist)
    with open("./outputs/text/s05d.txt", "w", encoding="utf-8") as f:
        f.write("Final state estimate (ECI, km and km/s):")
        for name, X, σ in zip(names, X_final, σ_final):
            f.write(f"{name:8s}: {X: .9f} ± {σ: .9f} (1σ)\n")

    return 0


if __name__ == "__main__":
    main()
```

./src/p01.py

```python
# libs
import matplotlib.pyplot as plt
import numpy as np

# src
from measurements import *

def e(data: measurements.Measurement) -> plt.Figure:
    t = np.asarray(data.t, dtype=float)
    i = np.asarray(data.i, dtype=float)
    ρ = np.asarray(data.ρ, dtype=float)
    dρ = np.asarray(data.dρ, dtype=float)

    labels = {0: "DSN #0 Goldstone", 1: "DSN #1 Madrid", 2: "DSN #2 Canberra"}
    colors = {0: "r", 1: "g", 2: "b"}

    order = np.argsort(t)
    t, i, ρ, dρ = t[order], i[order], ρ[order], dρ[order]

    fig, axs = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=False, figsize=(12, 8))

    for stn in np.unique(i):
        m = (i == stn)

        # range measurements
        axs[0].plot(t[m], ρ[m],
                    linestyle="-", linewidth=0,
                    marker=".", markersize=2,
                    color=colors.get(int(stn)),
                    label=labels.get(int(stn), f"DSN #{int(stn)}"))
        axs[0].grid(True, alpha=0.3)
```

```
        axs[0].legend(loc = "upper left")
        axs[0].set_ylabel("Range [km]")

        # range-rate measurements
        axs[1].plot(t[m], dp[m],
                linestyle="-", linewidth=0,
                marker=".", markersize=2,
                color=colors.get(int(stn)),
                label=labels.get(int(stn), f"DSN #{int(stn)}"))
        axs[1].grid(True, alpha=0.3)
        axs[1].legend(loc = "upper left")
        axs[1].set_ylabel("Range-Rate [km/s]")

    fig.supxlabel("Time [s]")
    fig.suptitle("DSN Measurements vs. Time [s]")
    fig.tight_layout()

    return fig
```

./src/p03.py

```
# libs
import matplotlib.pyplot as plt
import numpy as np

# src
from measurements import *
from helpers import *

def a(oe: list, σ_n: list, σ: list, σ_a: float, μ: float) -> [np.ndarray, np.ndarray, float,
↪ np.ndarray]:
    # choose x0 from starting OE
    r0, v0 = system.coe2rv(oe, μ)
    X0_plus = np.hstack([r0, v0])

    # choose P0 from starting range variance
    σ_r, σ_v = [float(x)/σ_a for x in σ_n]
    P0_plus = np.diag([σ_r**2]*3 + [σ_v**2]*3)

    # choose R0 from measurement noise
    R0 = np.diag(σ_n)

    return [X0_plus, P0_plus, R0]

def b(data: measurements.Measurement, X0_plus: np.ndarray, P0_plus: np.ndarray, σ_a: float,
↪ μ: float) -> [list, np.ndarray, np.ndarray]:
    t_pred, X_minus_hist, P_minus_hist = propagators.kf(data, X0_plus, P0_plus, σ_a, μ)
    return t_pred, X_minus_hist, P_minus_hist

```

```python
def c(t_pred: list, X_minus_hist: np.ndarray, P_minus_hist: np.ndarray) -> plt.Figure:
    σs = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_minus_hist]), 0.0, np.inf))  # Nx6
    bounds = 3.0 * σs  # Nx6

    state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
    colors = {0: "r", 1: "g", 2: "b"}

    fig, axs = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=False, figsize=(12, 8))

    for j in range(len(state_names)):
        ax = axs[j%3, int(j/3)]
        ax.fill_between(t_pred,  bounds[:, j], -bounds[:, j],
                        color = colors.get(j%3),
                        alpha=0.5)
        ax.grid(True, alpha=0.3)
        ax.set_title(f"±3σ {state_names[j]}")

    fig.supxlabel("Time [s]")
    fig.supylabel("Error Bound (Centered at 0)")
    fig.suptitle("Pre-Update Prediction Covariance Bounds (±3σ)")
    fig.tight_layout()

    return fig
```

./src/p04.py

```python
# libs
import matplotlib.pyplot as plt
import numpy as np

# src
from measurements import *
from helpers import *

def a(data: measurements.Measurement, X0_plus: np.ndarray, P0_plus: np.ndarray, R0: np.ndarray, stations: dict, σ_a: float, μ: float, RE: float, OMEGA_E: float, GAMMA0: float) -> [list, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    t_pred, X_minus_hist, P_minus_hist, X_plus_hist, P_plus_hist, yhat_minus_hist, yhat_plus_hist, resid_pre_hist, resid_post_hist, nis_pre_hist = propagators.ekf(data, X0_plus, P0_plus, R0, stations, σ_a, μ, RE, OMEGA_E, GAMMA0)
    return t_pred, X_minus_hist, P_minus_hist, X_plus_hist, P_plus_hist, yhat_minus_hist, yhat_plus_hist, resid_pre_hist, resid_post_hist, nis_pre_hist

def b(t_pred: list, X_minus_hist: np.ndarray, P_minus_hist: np.ndarray, X_plus_hist: np.ndarray, P_plus_hist: np.ndarray) -> plt.Figure:
    σ_ms = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_minus_hist]), 0.0, np.inf))  # Nx6
    σ_ps = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_plus_hist]), 0.0, np.inf))  # Nx6
```

```
    bound_ms = 3.0 * σ_ms # Nx6
    bound_ps = 3.0 * σ_ps # Nx6

    state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
    colors = {0: "r", 1: "g", 2: "b"}

    fig, axs = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=False, figsize=(12, 8))

    for j in range(len(state_names)):
        ax = axs[j%3, int(j/3)]
        ax.fill_between(t_pred,  bound_ms[:, j], -bound_ms[:, j],
                        color = "gray",
                        alpha=0.5)
        ax.fill_between(t_pred,  bound_ps[:, j], -bound_ps[:, j],
                        color = colors.get(j%3),
                        alpha=0.5)
        ax.grid(True, alpha=0.3)
        ax.set_title(f"±3σ {state_names[j]}")

    plt.figlegend(["Pre-Update", "Post-Update"])
    fig.supxlabel("Time [s]")
    fig.supylabel("Error Bound (Centered at 0)")
    fig.suptitle("Prediction Covariance Bounds (±3σ)")
    fig.tight_layout()

    return fig

def c(t_pred: list, X_minus_hist: np.ndarray, P_minus_hist: np.ndarray, X_plus_hist:
    np.ndarray, P_plus_hist: np.ndarray) -> plt.Figure:
    dX = (X_plus_hist - X_minus_hist)/1e6 # Nx6
    σ_ms = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_minus_hist]), 0.0, np.inf))  #
        Nx6
    σ_ps = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_plus_hist]), 0.0, np.inf))  # Nx6
    bound_ms = 3.0 * σ_ms # Nx6
    bound_ps = 3.0 * σ_ps # Nx6

    state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
    colors = {0: "r", 1: "g", 2: "b"}

    fig, axs = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=False, figsize=(12, 8))

    for j in range(len(state_names)):
        ax = axs[j%3, int(j/3)]
        ax.fill_between(t_pred,  bound_ms[:, j], -bound_ms[:, j],
                        color = "gray",
                        alpha=0.5)
        ax.fill_between(t_pred,  bound_ps[:, j], -bound_ps[:, j],
                        color = "dimgray",
```

```
                        alpha=0.5)
        ax.plot(t_pred, dX,
                linestyle="-", linewidth=0,
                marker=".", markersize=2,
                color = colors.get(j%3))
        ax.grid(True, alpha=0.3)
        ax.set_title(f"±3σ {state_names[j]}")

    plt.figlegend(["Pre-Update Bounds", "Post-Update Bounds", "Differences"])
    fig.supxlabel("Time [s]")
    fig.supylabel("Prediction (Centered at 0)")
    fig.suptitle("Δμₓ,ₖ Between Prediction Covariance Bounds (±3σ)")
    fig.tight_layout()

    return fig
```

./src/p05.py

```
# libs
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

# src
from measurements import *
from helpers import *

def a(data: measurements.Measurement, t: list, resid_pre_hist: np.ndarray, resid_post_hist:
↪   np.ndarray) -> plt.Figure:
    # extract arrays from data
    t = np.asarray(data.t, dtype=float)
    i = np.asarray(data.i, dtype=float)

    # shift epoch so t[0] = 0
    t0 = t.min()
    t = t - t0

    # sort by time
    order = np.argsort(t, kind="mergesort")
    t, i = t[order], i[order]

    # shift units for display
    res_pre_r_m = resid_pre_hist[:, 0] * 1e3 # [km] -> [m]
    res_post_r_m = resid_post_hist[:, 0] * 1e3 # [km] -> [m]
    res_pre_rr_cms = resid_pre_hist[:, 1] * 1e5 # [km/s] -> [cm/s]
    res_post_rr_cms = resid_post_hist[:, 1] * 1e5 # [km/s] -> [cm/s]

    # station info
    labels = {0: "DSN #0 Goldstone", 1: "DSN #1 Madrid", 2: "DSN #2 Canberra"}
```

```
    colors = {0: "r", 1: "g", 2: "b"}

    fig, axs = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=False, figsize=(12, 8))

    for stn in np.unique(i):
        m = (i == stn)

        # range residuals
        # axs[0].plot(t[m], res_pre_r_m[m],
        #         linestyle="-", linewidth=0,
        #         marker=".", markersize=2,
        #         color=colors.get(int(stn)),
        #         label=labels.get(int(stn), f"DSN #{int(stn)}"))
        axs[0].plot(t[m], res_post_r_m[m],
                linestyle="-", linewidth=0,
                marker=".", markersize=2,
                color=colors.get(int(stn)),
                label=labels.get(int(stn), f"DSN #{int(stn)}"))
        axs[0].grid(True, alpha=0.3)
        axs[0].legend(loc = "upper right")
        axs[0].set_ylabel("Range Residuals [km]")

        # range-rate residuals
        # axs[1].plot(t[m], res_pre_rr_cms[m],
        #         linestyle="-", linewidth=0,
        #         marker=".", markersize=2,
        #         color=colors.get(int(stn)),
        #         label=labels.get(int(stn), f"DSN #{int(stn)}"))
        axs[1].plot(t[m], res_post_rr_cms[m],
                linestyle="-", linewidth=0,
                marker=".", markersize=2,
                color=colors.get(int(stn)),
                label=labels.get(int(stn), f"DSN #{int(stn)}"))
        axs[1].grid(True, alpha=0.3)
        axs[1].legend(loc = "upper right")
        axs[1].set_ylabel("Range-Rate Residuals [km/s]")

    # measurement noise reference bands
    # axs[0].fill_between(t, -3.0, +3.0,
    #                     color = "gray",
    #                     alpha=0.5)
    # axs[1].fill_between(t, -3.0, +3.0,
    #                     color = "gray",
    #                     alpha=0.5)

    fig.supxlabel("Time [s]")
    fig.suptitle("DSN Measurement Residuals vs. Time [s]")
    fig.tight_layout()
```

```python

    return fig

def b(resid_post_hist: np.ndarray, nis_pre: np.ndarray) -> [float, float, float]:
    # shift units
    res_post_r_m = resid_post_hist[:, 0] * 1e3 # [km] -> [m]
    res_post_rr_cms = resid_post_hist[:, 1] * 1e5 # [km/s] -> [cm/s]

    rms_r_m = np.sqrt(np.mean(res_post_r_m**2))
    rms_rr_cms = np.sqrt(np.mean(res_post_rr_cms**2))
    nis_mean = np.mean(nis_pre)

    return rms_r_m, rms_rr_cms, nis_mean

def c(t_pred: list, X_plus_hist: np.ndarray, P_plus_hist: np.ndarray) -> plt.Figure:
    σ_ps = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_plus_hist]), 0.0, np.inf))  # Nx6
    bound_ps = 3.0 * σ_ps # Nx6

    state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
    colors = {0: "r", 1: "g", 2: "b"}

    fig, axs = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=False, figsize=(12, 8))

    for j in range(len(state_names)):
        ax = axs[j%3, int(j/3)]
        ax.fill_between(t_pred, X_plus_hist[:, j] - bound_ps[:, j], X_plus_hist[:, j] +
        ↪  bound_ps[:, j],
                        color = "dimgray",
                        alpha=0.5)
        ax.plot(t_pred, X_plus_hist[:, j],
                linestyle="-", linewidth=0,
                marker=".", markersize=2,
                color = colors.get(j%3))
        ax.grid(True, alpha=0.3)
        ax.set_title(f"{state_names[j]}")

    plt.figlegend(["±3σ Bounds", "Estimate"])
    fig.supxlabel("Time [s]")
    fig.supylabel("State Estimates")
    fig.suptitle("Final State Estimates (ECI, [km] & [km/s])")
    fig.tight_layout()

    return fig

def d(X_plus_hist: np.ndarray, P_plus_hist: np.ndarray) -> [np.ndarray, np.ndarray]:
    X_final = X_plus_hist[-1]
    σ_final = np.sqrt(np.clip(np.diag(P_plus_hist[-1]), 0.0, np.inf)) # 1σ
    state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
```

```python

    return X_final, σ_final, state_names
```

./src/helpers/propagators.py

```python
# libs
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

# src
from measurements import *
from .system import *

def kf(data: measurements.Measurement, X0_plus: np.ndarray, P0_plus: np.ndarray, σ_a: float,
↪   μ: float) -> [list, np.ndarray, np.ndarray]:
    # extract arrays from data
    t = np.asarray(data.t, dtype=float)

    # shift epoch so t[0] = 0
    t0 = t.min()
    t = t - t0

    # sort by time (for propagation)
    order = np.argsort(t)
    t = t[order]

    # prediction storage
    X_minus_hist = np.zeros((len(t), 6))
    P_minus_hist = np.zeros((len(t), 6, 6))

    X_plus = X0_plus.copy()
    P_plus = P0_plus.copy()
    t_prev = 0.0

    for k, t_k in enumerate(t):
        dt = t_k - t_prev

        if dt > 0:
            Φ0 = np.eye(6)
            y0 = np.concatenate([X_plus, Φ0.reshape(-1)])

            sol = sp.integrate.solve_ivp(
                fun=lambda tt, yy: eom_with_stm(tt, yy, μ),
                t_span=(t_prev, t_k),
                y0=y0,
                t_eval=[t_k],
                rtol=1e-10,
                atol=1e-12,
```

```
                method="DOP853",
            )
            y_k = sol.y[:, -1]
            X_minus = y_k[0:6]
            Φ = y_k[6:].reshape(6,6)

            # prediction-only, so no process update:
            Q_k = Q_discrete(dt, σ_a)
            P_minus = Φ @ P_plus @ Φ.T + Q_k
        else:
            # multiple measurements at same time tag: no propagation
            X_minus = X_plus
            P_minus = P_plus

        X_minus_hist[k, :] = X_minus
        P_minus_hist[k, :, :] = P_minus

        # prediction-only, so no measurement update:
        X_plus = X_minus
        P_plus = P_minus
        t_prev = t_k

    return t, X_minus_hist, P_minus_hist

def ekf(data: measurements.Measurement, X0_plus: np.ndarray, P0_plus: np.ndarray, R0:
↪   np.ndarray, stations: dict, σ_a: float, μ: float, RE: float, OMEGA_E: float, GAMMA0:
↪   float) -> [list, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray,
↪   np.ndarray, np.ndarray, np.ndarray]:
    # extract arrays from data
    t = np.asarray(data.t, dtype=float)
    i = np.asarray(data.i, dtype=float)
    ρ = np.asarray(data.ρ, dtype=float)
    dρ = np.asarray(data.dρ, dtype=float)

    # station positions
    R_ecef = site_ecef(stations, RE)

    # shift epoch so t[0] = 0
    t0 = t.min()
    t = t - t0

    # time steps
    N = len(t)

    # sort by time (for propagation)
    order = np.argsort(t, kind="mergesort")
    t, i = t[order], i[order]
    ρ_meas, dρ_meas = ρ[order], dρ[order]
```

```python

    # prediction storage
    X_minus_hist = np.zeros((N, 6))
    P_minus_hist = np.zeros((N, 6, 6))
    X_plus_hist = np.zeros((N, 6))
    P_plus_hist = np.zeros((N, 6, 6))

    yhat_minus_hist = np.zeros((N, 2))
    yhat_plus_hist  = np.zeros((N, 2))
    resid_pre_hist  = np.zeros((N, 2)) # y - h(X_minus)
    resid_post_hist = np.zeros((N, 2)) # y - h(X_plus)
    nis_pre_hist = np.zeros(N)

    X_plus = X0_plus.copy()
    P_plus = P0_plus.copy()
    t_prev = t[0]

    for k, t_k in enumerate(t):
        dt = t_k - t_prev

        if dt > 0:
            Φ0 = np.eye(6)
            y0 = np.concatenate([X_plus, Φ0.reshape(-1)])

            sol = sp.integrate.solve_ivp(
                fun=lambda tt, yy: eom_with_stm(tt, yy, μ),
                t_span=(t_prev, t_k),
                y0=y0,
                t_eval=[t_k],
                rtol=1e-10,
                atol=1e-12,
                method="DOP853",
            )
            y_k = sol.y[:, -1]
            X_minus = y_k[0:6]
            Φ = y_k[6:].reshape(6,6)

            # process update
            Q_k = Q_discrete(dt, σ_a)
            P_minus = Φ @ P_plus @ Φ.T + Q_k
        else:
            # multiple measurements at same time tag: no propagation
            X_minus = X_plus
            P_minus = P_plus

        # save pre-update
        X_minus_hist[k, :] = X_minus
        P_minus_hist[k, :, :] = P_minus
```

```python

        # measurement update
        y_k = np.array([p_meas[k], dp_meas[k]])
        yhat_minus, H_k = meas_and_jacobian(X_minus, i[k], t_k, R_ecef, OMEGA_E, GAMMA0)
        yhat_minus_hist[k] = yhat_minus
        resid_pre_hist[k] = y_k - yhat_minus

        # innovation
        v = y_k - yhat_minus
        S = H_k @ P_minus @ H_k.T + R0
        nis_pre = float(v.T @ np.linalg.inv(S) @ v)
        nis_pre_hist[k] = nis_pre

        # kalman gain
        # K = P_minus @ H_k.T @ np.linalg.pinv(S)
        K = P_minus @ H_k.T @ np.linalg.solve(S, np.eye(S.shape[0]))

        # update state
        X_plus = X_minus + K @ v

        # joseph covariance update
        I6 = np.eye(6)
        P_plus = (I6 - K @ H_k) @ P_minus @ (I6 - K @ H_k).T + K @ R0 @ K.T

        # save post-update
        X_plus_hist[k, :] = X_plus
        P_plus_hist[k, :, :] = P_plus

        # post-fit measurement
        yhat_plus, _ = meas_and_jacobian(X_plus, i[k], t_k, R_ecef, OMEGA_E, GAMMA0)
        yhat_plus_hist[k] = yhat_plus
        resid_post_hist[k] = y_k - yhat_plus

        # advance time
        t_prev = t_k

    return t, X_minus_hist, P_minus_hist, X_plus_hist, P_plus_hist, yhat_minus_hist,
    ↪  yhat_plus_hist, resid_pre_hist, resid_post_hist, nis_pre_hist
```

./src/helpers/system.py

```python
# libs
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

# src
from measurements import *

```

```python
def coe2rv(oe: list, μ:float) -> [np.ndarray, np.ndarray]:
    # a in km, angles in rad
    a, e, i, ω, Ω, ν = [float(x) for x in oe]

    # Semi-latus rectum
    p = a * (1 - e**2)

    # Perifocal coordinates (PQW frame)
    r_pf = (p / (1 + e*np.cos(ν))) * np.array([np.cos(ν), np.sin(ν), 0.0])
    v_pf = np.sqrt(μ/p) * np.array([-np.sin(ν), e + np.cos(ν), 0.0])

    # Rotation from PQW -> ECI (3-1-3 sequence)
    R_pqw_eci = sp.spatial.transform.Rotation.from_euler("ZXZ", [Ω, i, ω])
    r = R_pqw_eci.as_matrix() @ r_pf
    v = R_pqw_eci.as_matrix() @ v_pf

    return r, v

def A_matrix(r: np.ndarray, μ: float) -> np.ndarray:
    rnorm = np.linalg.norm(r)
    I3 = np.eye(3)
    dadr = -μ * (I3/(rnorm**3) - 3.0*np.outer(r, r)/(rnorm**5))
    A = np.block([
        [np.zeros((3,3)), I3],
        [dadr, np.zeros((3,3))]
    ])

    return A

def Q_discrete(dt: float, σ_a: float) -> np.ndarray:
    if dt <= 0.0:
        return np.zeros((6,6))

    q_a = σ_a**2
    I3 = np.eye(3)

    Q = q_a * np.block([
        [(dt**3/3.0)*I3, (dt**2/2.0)*I3],
        [(dt**2/2.0)*I3, (dt)*I3]
    ])

    return Q

def eom_with_stm(t: float, y: list, μ: float) -> np.ndarray:
    # y = [r(3), v(3), Phi:flat(36)]
    r = y[0:3]
    v = y[3:6]
    Φ = y[6:].reshape(6,6)
```

```

    rnorm = np.linalg.norm(r)
    a = -μ * r / (rnorm**3)

    A = A_matrix(r, μ)
    dΦ = A @ Φ

    return np.concatenate([v, a, dΦ.reshape(-1)])

def site_ecef(stations: dict, RE: float) -> dict:
    R_ecef = {}

    for idx, (ϕ, λ) in stations.items():
        R_ecef[idx] = RE * np.array([
            np.cos(ϕ)*np.cos(λ),
            np.cos(ϕ)*np.sin(λ),
            np.sin(ϕ)
        ])

    return R_ecef

def site_eci(station_idx: int, t: float, R_ecef: dict, OMEGA_E: float, GAMMA0: float) -> [np.ndarray, np.ndarray]:
    γ = GAMMA0 + OMEGA_E * t
    ω = np.array([0.0, 0.0, OMEGA_E])

    R_ecef_eci = sp.spatial.transform.Rotation.from_euler("Z", γ)
    R_site = R_ecef_eci.as_matrix() @ R_ecef[int(station_idx)]

    dR_site = np.cross(ω, R_site)

    return R_site, dR_site

def meas_and_jacobian(X: list, station_idx: int, t: float, R_ecef: dict, OMEGA_E: float, GAMMA0: float) -> [np.ndarray, np.ndarray]:
    r = X[0:3]
    v = X[3:6]
    R_site, dR_site = site_eci(station_idx, t, R_ecef, OMEGA_E, GAMMA0)

    ρ_vec = r - R_site
    ρ = np.linalg.norm(ρ_vec)
    u = ρ_vec / ρ

    v_rel = v - dR_site
    dρ = u @ v_rel

    yhat = np.array([ρ, dρ])
    I3 = np.eye(3)
```

```

    # H (2x6)
    H_ρ_r = u.reshape(1,3)
    H_ρ_v = np.zeros((1,3))
    H_dρ_r = (1.0/ρ) * (v_rel.reshape(1,3) @ (I3 - np.outer(u, u)))
    H_dρ_v = u.reshape(1,3)

    H = np.block([
        [H_ρ_r,  H_ρ_v],
        [H_dρ_r, H_dρ_v],
    ])

    return yhat, H
```

./src/measurements/measurements.py

```
# libs
from typing import NamedTuple
import numpy as np

class Measurement(NamedTuple):
    t: np.ndarray
    i: np.ndarray
    ρ: np.ndarray
    dρ: np.ndarray
```

./src/measurements/numpy_data.py

```
# libs
import numpy as np

# src
from .measurements import *

def raw_load(file_path: str) -> np.ndarray:
    try:
        raw_data = np.load(file_path, allow_pickle=True)
    except NameError:
        print(f"Could not load data from {file_path}")

    return raw_data

def parse(raw_data: np.ndarray) -> Measurement:
    data = Measurement(raw_data[:, 0], raw_data[:, 1], raw_data[:, 2], raw_data[:, 3])

    return data

def load(file_path: str) -> Measurement:
    raw_data = raw_load(file_path)
```

```
    data = parse(raw_data)

    return data
```