

ENAE 441 - 0101

Final Project: Extended Kalman Filter

Due on December 16th, 2025 at 12:30 PM

Dr. Martin, 09:30 AM

Vai Srivastava

December 20th, 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

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

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: $(35.297^\circ, -116.914^\circ)$
- DSN # 1: Madrid, Spain: $(40.4311^\circ, -4.248^\circ)$
- DSN # 2: Canberra, Australia: $(-35.4023^\circ, 148.9813^\circ)$

These ground stations are positioned on a spherical Earth of radius 6378.137 km , with a rotation rate of $\omega_{\mathcal{E}/\mathcal{N}} = 7.292115 \times 10^{-5} \frac{\text{rad}}{\text{s}}$ and a Local Sidereal Time of $\gamma_0 = 0^\circ$.

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:

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

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

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

where $\mathbf{R}_{\text{site},i}$ 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 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

- Express the non-linear system in continuous time state-space form, clearly defining the vectors $\mathbf{f}(X(t))$ and $\mathbf{h}(X(t))$.
- Define the linearized dynamics and measurement matrices $A(t)$ and $C(t)$.
- Show how these matrices are converted to their discrete time forms F_k and H_k . Recall F_k is the state transition matrix $\Phi(t_j, t_i)$ which requires integration.
- Define your noise matrices Q_k and R_k , and discuss their relationship to the aforementioned system of equations.
- Plot the measurements as a function of time.

Solution

Part A

First, we use the inertial Cartesian state:

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

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:

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

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 reference 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 $r = \bar{r}(t), v = \bar{v}(t), r = \|r\|$,

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

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:

$$\frac{\partial \rho}{\partial r} = u^T, \quad \frac{\partial \rho}{\partial v} = 0_{1 \times 3}$$

Range-rate row:

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

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

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

Thus,

$$C(t) = \begin{bmatrix} u^T & 0_{1 \times 3} \\ \frac{1}{\rho} v_{\text{rel}}^T (I - uu^T) & u^T \end{bmatrix}$$

Part C

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

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

The discrete linearized mapping for perturbations is:

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

Φ is obtained by integrating the variational equation alongside the state:

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

and then setting:

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

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:

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

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

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

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

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

Part E

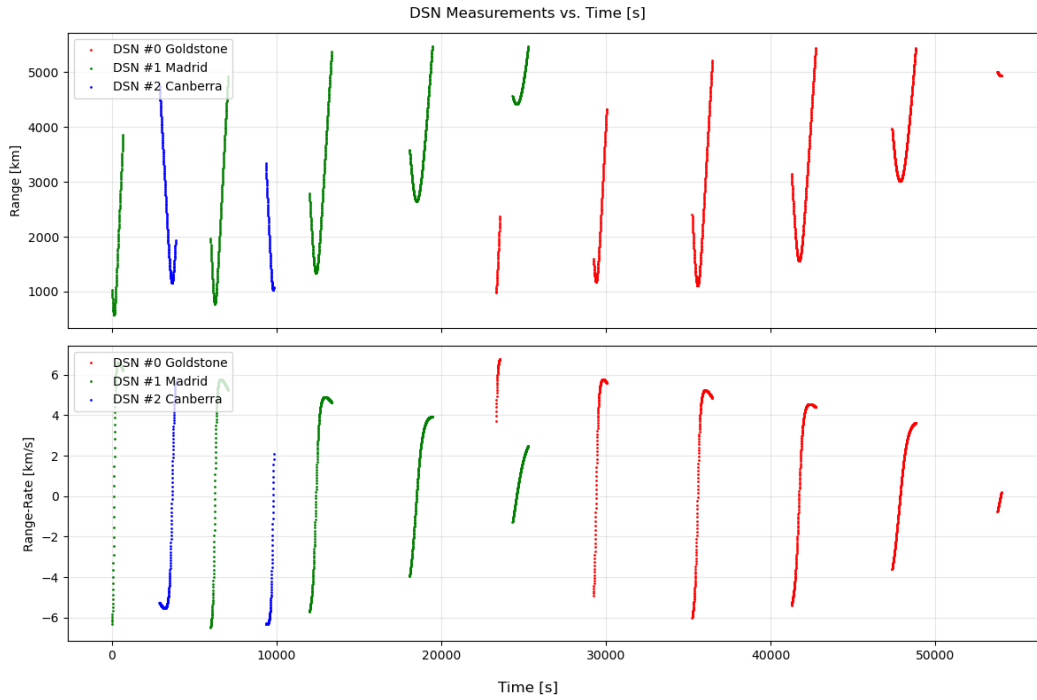


Figure 1: DSN Range [km] vs. Time [s]

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

```

1 Inputs:
2   data.t[0..N-1], data.i[0..N-1], data.rho[0..N-1], data.drho[0..N-1]
3   station lat/lon: (phi[i], lam[i]), i=0..2
4   constants: mu, RE, omegaE, gamma0
5   noise: R = diag(sig_rho^2, sig_drho^2) (in FILTER UNITS)
6           Q-model params: q_a (white accel PSD) OR set Qk = 0
7   initial: X_plus (6x1), P_plus (6x6)
8
9 Precompute:
10   for each station i:
11     R_ecef[i] = RE * [cos(phi_i)cos(lam_i), cos(phi_i)sin(lam_i), sin(phi_i)]^T
12
13 Helpers:
14   site_eci(i, t):
15     gamma = gamma0 + omegaE*t
16     R_site = R3(gamma) * R_ecef[i]
17     Rdot_site = omega x R_site # (omega = [0,0,omegaE]^T)
18     return (R_site, Rdot_site)
19
20   dynamics(X):
21     r = X[0:3], v = X[3:6]
22     rnorm = ||r||
23     rdot = v
24     vdot = -mu * r / rnorm^3
25     return [rdot; vdot]
26
27   A_matrix(X):
28     r = X[0:3]; rnorm = ||r||
29     I3 = identity(3)
30     dadr = -mu*( (1/rnorm^3)*I3 - (3/rnorm^5)*(r*r^T) )
31     return [[0, I3],
32             [dadr, 0]]
33
34   propagate_state_and_stm(X0, t0, t1):
35     Integrate coupled ODEs from t0 -> t1:
36     Xdot = dynamics(X)
37     Phidot = A_matrix(X) * Phi
38     with initial Phi(t0)=I6
39     return (X_minus, Phi) where Phi = Φ(t1,t0)
40

```

```

41  meas_and_jacobian(X, i, t):
42      (R_site, Rdot_site) = site_eci(i, t)
43      r = X[0:3]; v = X[3:6]
44      rho_vec = r - R_site
45      rho = ||rho_vec||
46      u = rho_vec / rho
47      v_rel = v - Rdot_site
48      drho = u^T * v_rel
49
50      yhat = [rho; drho]
51
52      # Jacobian H = ∂h/∂X (2x6)
53      H_rho_r = u^T
54      H_rho_v = [0 0 0]
55
56      H_drho_v = u^T
57      H_drho_r = (1/rho) * (v_rel^T * (I3 - u*u^T))
58
59      H = [[H_rho_r, H_rho_v],
60           [H_drho_r, H_drho_v]]
61      return (yhat, H)
62
63  process_noise_discrete(dt, q_a):
64      if q_a == 0:
65          return 0_(6x6)
66      else:
67          I3 = identity(3)
68          Q = q_a * [[(dt^3/3)*I3, (dt^2/2)*I3],
69                    [(dt^2/2)*I3, (dt)*I3]]
70          return Q
71
72  Main EKF:
73      t_prev = data.t[0]
74
75      for k in 0..N-1:
76          t_k = data.t[k]
77          i_k = data.i[k]
78          y_k = [data.rho[k], data.drho[k]]^T
79
80          # Predict / propagate to t_k
81          dt = t_k - t_prev
82          if dt > 0:
83              (X_minus, Phi) = propagate_state_and_stm(X_plus, t_prev, t_k)
84              Qk = process_noise_discrete(dt, q_a)
85              P_minus = Phi * P_plus * Phi^T + Qk
86          else:
87              X_minus = X_plus
88              P_minus = P_plus

```

```

89     Phi = I6
90
91     # Measurement prediction at (t_k, i_k)
92     (yhat_k, Hk) = meas_and_jacobian(X_minus, i_k, t_k)
93     nu = y_k - yhat_k # innovation (2x1)
94
95     # Gain computation
96     S = Hk * P_minus * Hk^T + R # (2x2)
97     K = P_minus * Hk^T * inv(S) # (6x2)
98
99     # Update (state + Joseph covariance)
100    X_plus = X_minus + K * nu
101    P_plus = (I6 - K*Hk)*P_minus*(I6 - K*Hk)^T + K*R*K^T
102
103    # Diagnostics
104    log_residual nu, NIS = nu^T * inv(S) * nu, etc.
105    t_prev = t_k

```

Outputs:

filtered state estimates X_{plus} and X_{minus} over time
residual history, NIS history, covariance history

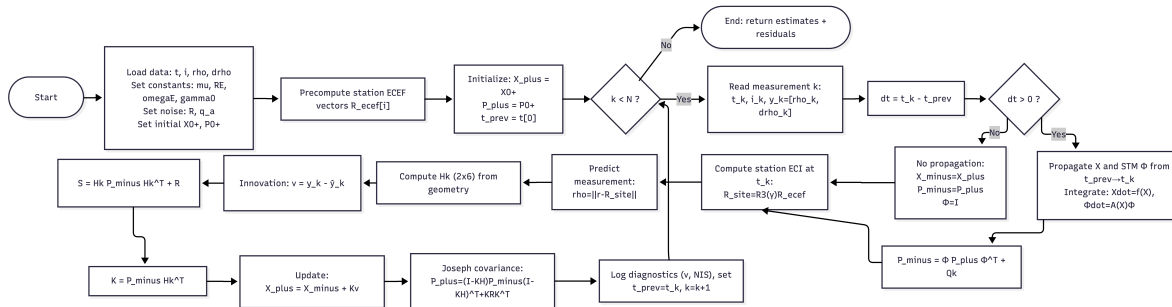


Figure 2: (Pseudo) Code Flowchart

Problem 3: Pure Prediction

- Specify your choice of \mathbf{x}_0 , P_0 , Q_0 , and R_0 . Explain your reasoning. (For \mathbf{x}_0 , express in cartesian coordinates.)
- Implement just the prediction step of the extended Kalman Filter (i.e. do not perform measurement updates).
- 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 $P_{\hat{\mathbf{x}},k}^-(t)$. Choose appropriate y-limits for your plot to maximize readability. Explain what you see and if it makes sense.

Solution

Part A

\mathbf{x}_0 is the nominal spacecraft orbit:

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

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:

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

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

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

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:

$$\sigma_\rho^2 = (1 \times 10^{-3} \text{ [km]})^2$$

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

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

```

1  X0+ =
2  [ 4.48544067e+03 -1.26177495e+03  4.48544067e+03  2.15162036e+00
3    7.55367318e+00  2.15162036e+00]
4  P0+ =
5  [[1.e+00 0.e+00 0.e+00 0.e+00 0.e+00 0.e+00]
6   [0.e+00 1.e+00 0.e+00 0.e+00 0.e+00 0.e+00]
7   [0.e+00 0.e+00 1.e+00 0.e+00 0.e+00 0.e+00]
8   [0.e+00 0.e+00 0.e+00 1.e-08 0.e+00 0.e+00]
9   [0.e+00 0.e+00 0.e+00 0.e+00 1.e-08 0.e+00]
10  [0.e+00 0.e+00 0.e+00 0.e+00 0.e+00 1.e-08]]
11  R0 =
12  [[1.e-06 0.e+00]
13   [0.e+00 1.e-10]]

```

Part B

Implementation can be viewed in the [Python files](#) embedded at the end of the report.

Part C

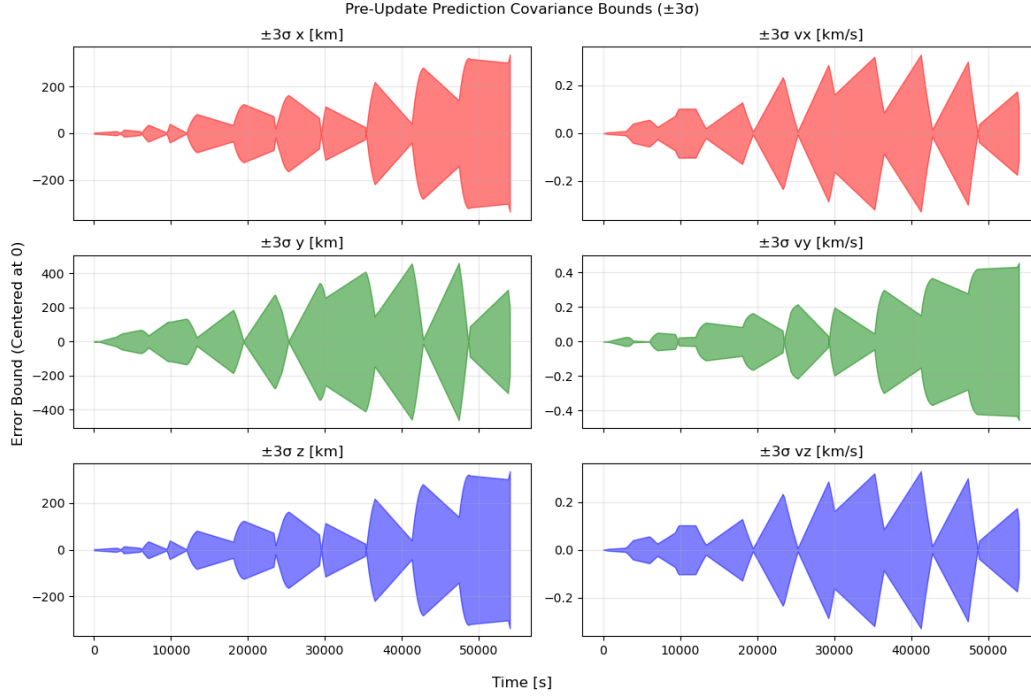


Figure 3: $\pm 3\sigma$ Bounds on Pre- Measurement Covariance $P_{\hat{x},k}^-$ vs. Time t

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

- Implement the measurement update step of the extended Kalman Filter.
- Plot the pre- and post-measurement update $\pm 3\sigma$ bounds on the same plot. Explain any differences.
- Plot the difference between the pre- and post-measurement update state estimates, $\mu_{\hat{x},k}^+(t_k) - \mu_{\hat{x},k}^-(t_k)$, inside of the pre-measurement covariance $\pm 3\sigma$ bounds $P_{\hat{x},k}^+(t_k)$.

Solution

Part A

Implementation can be viewed in the [Python files](#) embedded at the end of the report.

Part B

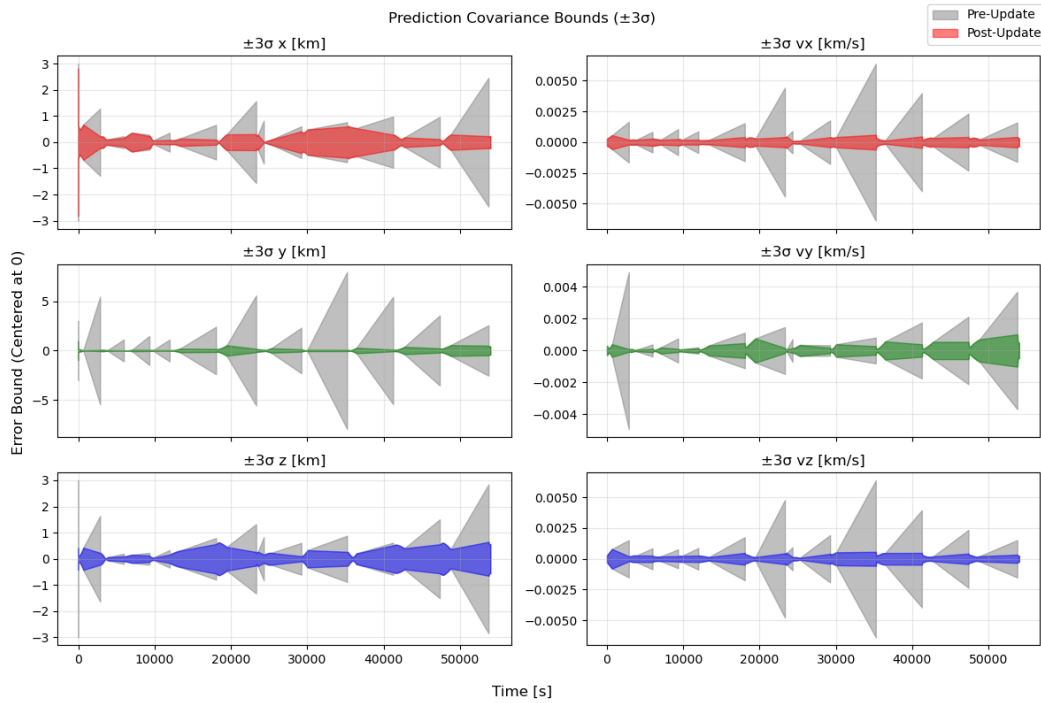


Figure 4: $\pm 3\sigma$ Bounds on Pre- and Post- Measurement Covariance

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

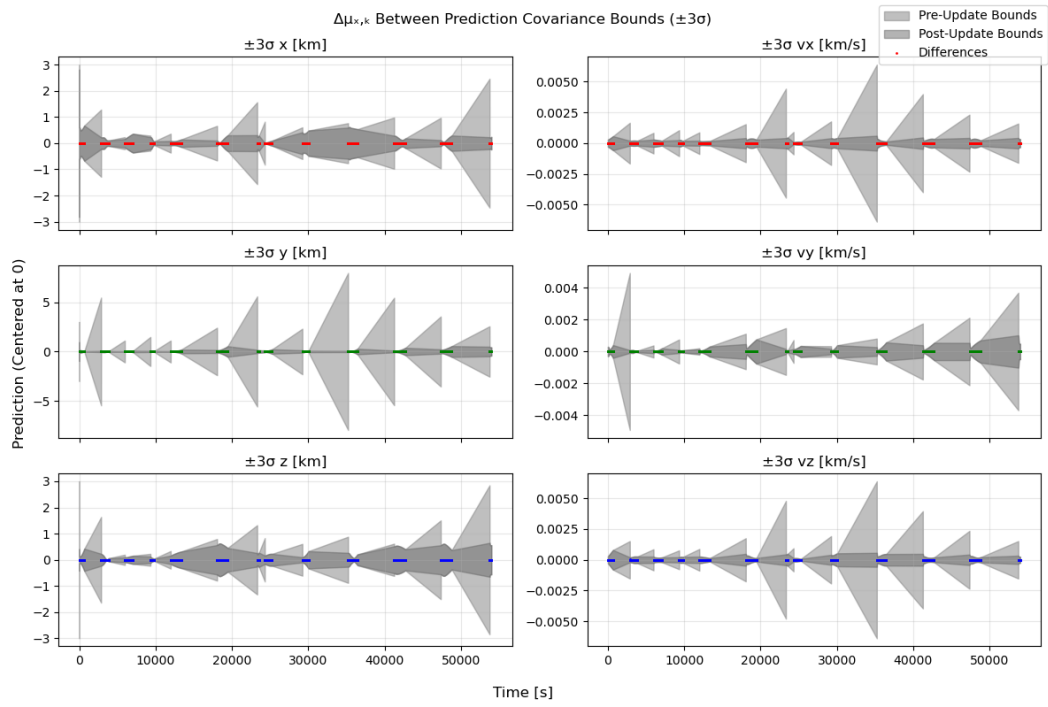


Figure 5: Difference Between Pre- and Post-Measurement Update State Estimates

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:

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

- b. Do your measurement residuals give you confidence that you estimated the state correctly? Why or why not?
- c. Plot the estimated state $\boldsymbol{\mu}_{\hat{\mathbf{x}},k}^+(t_k)$ and its associated $\pm 3\sigma$ bounds taken from $P_{\hat{\mathbf{x}},k}^+(t)$ as a function of time.
- d. Report your best state estimate at the final time-step and its associated uncertainty.

Solution

Part A

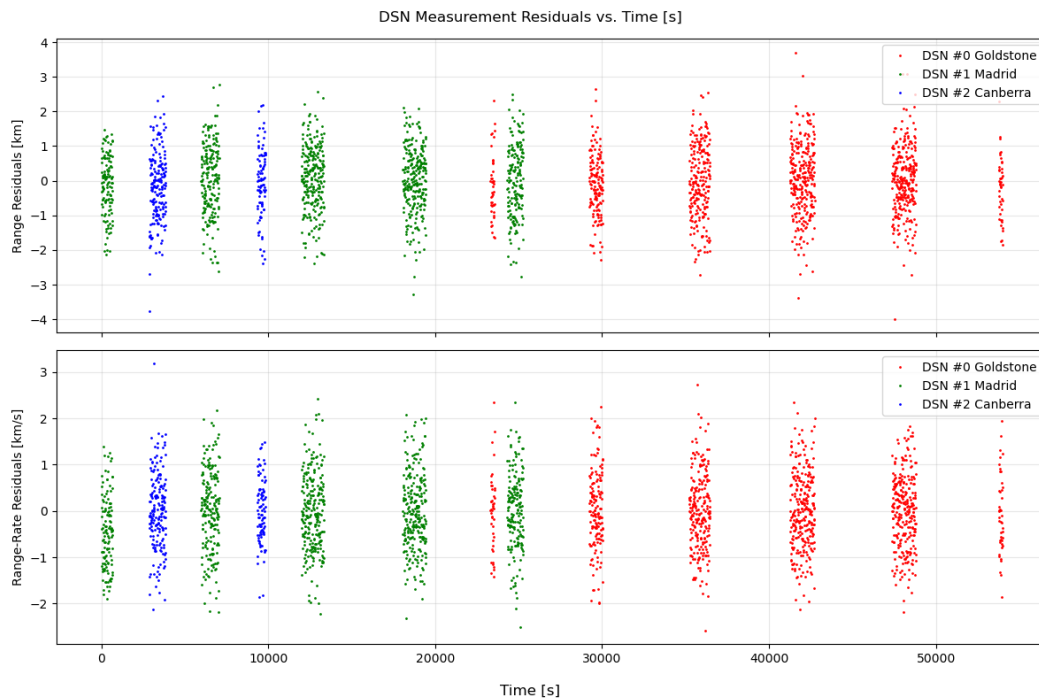


Figure 6: Post-Fit Measurement Residuals $\delta \mathbf{Y}$ vs. Time t

Part B

```

1 | Post-fit RMS range residual: 0.970 [m]
2 | Post-fit RMS range-rate residual: 0.817 [cm/s]
3 | 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:

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

$$\text{RMS range-rate residual} \approx 0.817 \frac{\text{cm}}{\text{s}}$$

- 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

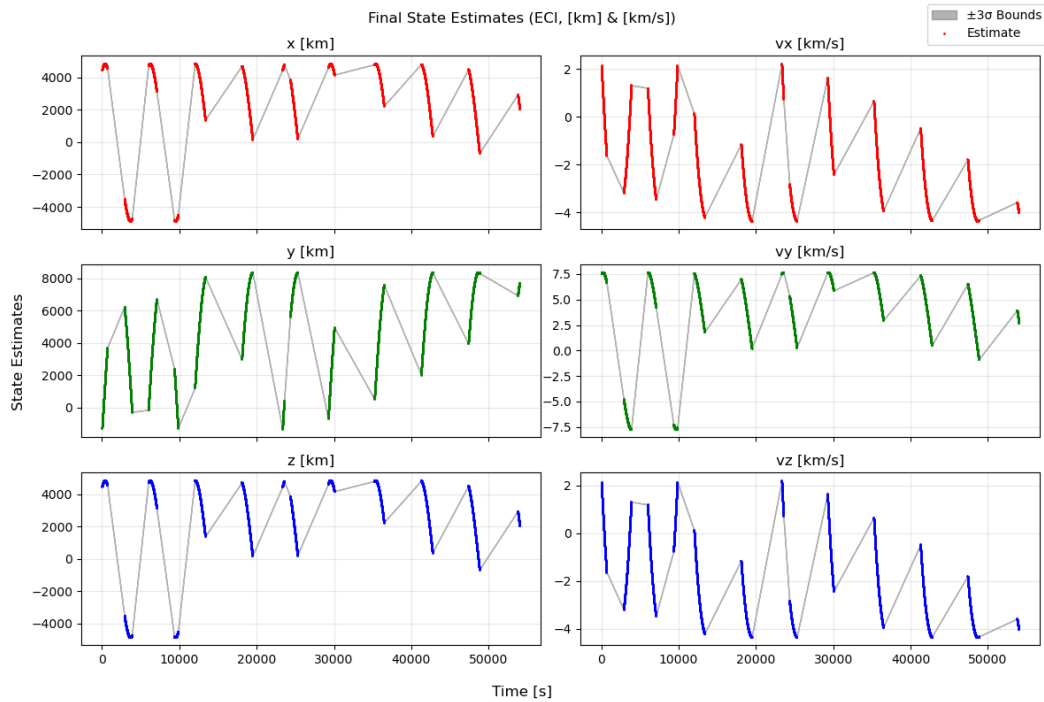


Figure 7: Estimated State $\mu_{x,k}^+$ with $\pm 3\sigma$ Bounds vs. Time t

Part D

```
1 | Final state estimate (ECI, km and km/s):x [km] : 2050.638271630 ± 0.080274911 (1σ)
2 | y [km] : 7747.569493734 ± 0.141891722 (1σ)
3 | z [km] : 2050.672230306 ± 0.186952201 (1σ)
4 | vx [km/s]: -4.010088130 ± 0.000109635 (1σ)
5 | vy [km/s]: 2.701084104 ± 0.000163016 (1σ)
6 | 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

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

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

Answer

Code

See the [full source code](#) for this project.

./src/final.py

```
1  # libs
2  import matplotlib.pyplot as plt
3  import numpy as np
4  import os
5
6  # src
7  from measurements import *
8  from helpers import *
9  import p01
10 import p03
11 import p04
12 import p05
13
14 def main() -> int:
15     # constants
16     EPS = 1e-12 # [-]
17     MU = 398600.4418 # [km^3/s^2]
18     RE = 6378.137 # [km]
19     OMEGA_E = 7.292115e-5 # [rad/s]
20     GAMMA0 = 0.0 # [rad]
21
22     # givens
23     oe = [
24         7e3, # [km]
25         0.2, # [-]
26         np.deg2rad(45), # [deg -> rad]
27         np.deg2rad(0), # [deg -> rad]
28         np.deg2rad(270), # [deg -> rad]
29         np.deg2rad(78.75) # [deg -> rad]
30     ]
31     stations = {
32         0: (np.deg2rad(35.297), np.deg2rad(-116.914)), # [lat, long]
33         1: (np.deg2rad(40.4311), np.deg2rad(-4.248)), # [lat, long]
34         2: (np.deg2rad(-35.4023), np.deg2rad(148.9813)), # [lat, long]
35     }
36     # measurement noise
37     sigma_n = [
38         1e-6, # [km^2]
39         1e-10 # [km^2/s^2]
40     ]
41     # process noise tuning
42     sigma_a = 1e-6 # [km/s^2]
43
```

```

44 # load numpy measurements
45 data = numpy_data.load("./data/Project-Measurements-Easy.npy")
46
47 #####
48 # p01 #
49 #####
50 # p01e
51 p01.e(data).savefig("./outputs/figures/s01e.png")
52
53 #####
54 # p03 #
55 #####
56 # p03a
57 X0_plus, P0_plus, R0 = p03.a(oe,  $\sigma_n$ ,  $\sigma_a$ , MU)
58 with open("./outputs/text/s03a.txt", "w", encoding="utf-8") as f:
59     f.write(f"X0+ =\n{X0_plus}\n")
60     f.write(f"P0+ =\n{P0_plus}\n")
61     f.write(f"R0 =\n{R0}\n")
62 # p03b
63 t_pred, X_minus_hist, P_minus_hist = p03.b(data, X0_plus, P0_plus,  $\sigma_a$ , MU)
64 # p03c
65 p03.c(t_pred, X_minus_hist, P_minus_hist).savefig("./outputs/figures/s03c.png")
66
67 #####
68 # p04 #
69 #####
70 # p04a
71 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,  $\sigma_a$ , MU, RE, OMEGA_E, GAMMA0)
72 # p04b
73 p04.b(t_pred, X_minus_hist, P_minus_hist, X_plus_hist,
    ↪ P_plus_hist).savefig("./outputs/figures/s04b.png")
74 # p04c
75 p04.c(t_pred, X_minus_hist, P_minus_hist, X_plus_hist,
    ↪ P_plus_hist).savefig("./outputs/figures/s04c.png")
76
77 #####
78 # p05 #
79 #####
80 # p05a
81 p05.a(data, t_pred, resid_pre_hist,
    ↪ resid_post_hist).savefig("./outputs/figures/s05a.png")
82 # p05b
83 rms_r_m, rms_rr_cms, nis_mean = p05.b(resid_post_hist, nis_pre_hist)
84 with open("./outputs/text/s05b.txt", "w", encoding="utf-8") as f:
85     f.write(f"Post-fit RMS range residual: {rms_r_m:.3f} [m]\n")
86     f.write(f"Post-fit RMS range-rate residual: {rms_rr_cms:.3f} [cm/s]\n")

```

```

87     f.write(f"Mean pre-fit NIS (df=2): {nis_mean:.3f}\n")
88     # p05c
89     p05.c(t_pred, X_plus_hist, P_plus_hist).savefig("./outputs/figures/s05c.png")
90     # p05d
91     X_final,  $\sigma$ _final, names = p05.d(X_plus_hist, P_plus_hist)
92     with open("./outputs/text/s05d.txt", "w", encoding="utf-8") as f:
93         f.write("Final state estimate (ECI, km and km/s):")
94         for name, X,  $\sigma$  in zip(names, X_final,  $\sigma$ _final):
95             f.write(f"{name:8s}: {X: .9f}  $\pm$  { $\sigma$ : .9f} (1 $\sigma$ )\n")
96
97     return 0
98
99
100 if __name__ == "__main__":
101     main()

```

./src/p01.py

```

1  # libs
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  # src
6  from measurements import *
7
8  def e(data: measurements.Measurement) -> plt.Figure:
9      t = np.asarray(data.t, dtype=float)
10     i = np.asarray(data.i, dtype=float)
11     p = np.asarray(data.p, dtype=float)
12     dp = np.asarray(data.dp, dtype=float)
13
14     labels = {0: "DSN #0 Goldstone", 1: "DSN #1 Madrid", 2: "DSN #2 Canberra"}
15     colors = {0: "r", 1: "g", 2: "b"}
16
17     order = np.argsort(t)
18     t, i, p, dp = t[order], i[order], p[order], dp[order]
19
20     fig, axs = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=False, figsize=(12, 8))
21
22     for stn in np.unique(i):
23         m = (i == stn)
24
25         # range measurements
26         axs[0].plot(t[m], p[m],
27                     linestyle="-", linewidth=0,
28                     marker=".", markersize=2,
29                     color=colors.get(int(stn)),
30                     label=labels.get(int(stn), f"DSN #{int(stn)}"))
31         axs[0].grid(True, alpha=0.3)

```

```

32     axs[0].legend(loc = "upper left")
33     axs[0].set_ylabel("Range [km]")
34
35     # range-rate measurements
36     axs[1].plot(t[m], dp[m],
37                 linestyle="-", linewidth=0,
38                 marker=".", markersize=2,
39                 color=colors.get(int(stn)),
40                 label=labels.get(int(stn), f"DSN #{int(stn)}"))
41     axs[1].grid(True, alpha=0.3)
42     axs[1].legend(loc = "upper left")
43     axs[1].set_ylabel("Range-Rate [km/s]")
44
45     fig.supxlabel("Time [s]")
46     fig.suptitle("DSN Measurements vs. Time [s]")
47     fig.tight_layout()
48
49     return fig

```

./src/p03.py

```

1  # libs
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  # src
6  from measurements import *
7  from helpers import *
8
9  def a(oe: list, σ_n: list, σ_a: float, μ: float) -> [np.ndarray, np.ndarray, float,
10 ↪ np.ndarray]:
11     # choose x0 from starting OE
12     r0, v0 = system.coe2rv(oe, μ)
13     X0_plus = np.hstack([r0, v0])
14
15     # choose P0 from starting range variance
16     σ_r, σ_v = [float(x)/σ_a for x in σ_n]
17     P0_plus = np.diag([σ_r**2]*3 + [σ_v**2]*3)
18
19     # choose R0 from measurement noise
20     R0 = np.diag(σ_n)
21
22     return [X0_plus, P0_plus, R0]
23
24 def b(data: measurements.Measurement, X0_plus: np.ndarray, P0_plus: np.ndarray, σ_a: float,
25 ↪ μ: float) -> [list, np.ndarray, np.ndarray]:
26     t_pred, X_minus_hist, P_minus_hist = propagators.kf(data, X0_plus, P0_plus, σ_a, μ)
27     return t_pred, X_minus_hist, P_minus_hist

```

```

27 def c(t_pred: list, X_minus_hist: np.ndarray, P_minus_hist: np.ndarray) -> plt.Figure:
28     σ = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_minus_hist]), 0.0, np.inf)) # Nx6
29     bounds = 3.0 * σ # Nx6
30
31     state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
32     colors = {0: "r", 1: "g", 2: "b"}
33
34     fig, axs = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=False, figsize=(12, 8))
35
36     for j in range(len(state_names)):
37         ax = axs[j%3, int(j/3)]
38         ax.fill_between(t_pred, bounds[:, j], -bounds[:, j],
39                        color = colors.get(j%3),
40                        alpha=0.5)
41         ax.grid(True, alpha=0.3)
42         ax.set_title(f"±3σ {state_names[j]}")
43
44     fig.supxlabel("Time [s]")
45     fig.supylabel("Error Bound (Centered at 0)")
46     fig.suptitle("Pre-Update Prediction Covariance Bounds (±3σ)")
47     fig.tight_layout()
48
49     return fig

```

./src/p04.py

```

1 # libs
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 # src
6 from measurements import *
7 from helpers import *
8
9 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]:
10     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)
11     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
12
13 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:
14     σ_ms = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_minus_hist]), 0.0, np.inf)) #
    ↳ Nx6
15     σ_ps = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_plus_hist]), 0.0, np.inf)) # Nx6

```

```

16 bound_ms = 3.0 *  $\sigma_{ms}$  # Nx6
17 bound_ps = 3.0 *  $\sigma_{ps}$  # Nx6
18
19 state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
20 colors = {0: "r", 1: "g", 2: "b"}
21
22 fig, axs = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=False, figsize=(12, 8))
23
24 for j in range(len(state_names)):
25     ax = axs[j%3, int(j/3)]
26     ax.fill_between(t_pred, bound_ms[:, j], -bound_ms[:, j],
27                    color = "gray",
28                    alpha=0.5)
29     ax.fill_between(t_pred, bound_ps[:, j], -bound_ps[:, j],
30                    color = colors.get(j%3),
31                    alpha=0.5)
32     ax.grid(True, alpha=0.3)
33     ax.set_title(f" $\pm 3\sigma$  {state_names[j]}")
34
35 plt.figlegend(["Pre-Update", "Post-Update"])
36 fig.supxlabel("Time [s]")
37 fig.supylabel("Error Bound (Centered at 0)")
38 fig.suptitle("Prediction Covariance Bounds ( $\pm 3\sigma$ )")
39 fig.tight_layout()
40
41 return fig
42
43 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:
44     dX = (X_plus_hist - X_minus_hist)/1e6 # Nx6
45      $\sigma_{ms}$  = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_minus_hist]), 0.0, np.inf)) #
↪ Nx6
46      $\sigma_{ps}$  = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_plus_hist]), 0.0, np.inf)) # Nx6
47     bound_ms = 3.0 *  $\sigma_{ms}$  # Nx6
48     bound_ps = 3.0 *  $\sigma_{ps}$  # Nx6
49
50 state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
51 colors = {0: "r", 1: "g", 2: "b"}
52
53 fig, axs = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=False, figsize=(12, 8))
54
55 for j in range(len(state_names)):
56     ax = axs[j%3, int(j/3)]
57     ax.fill_between(t_pred, bound_ms[:, j], -bound_ms[:, j],
58                    color = "gray",
59                    alpha=0.5)
60     ax.fill_between(t_pred, bound_ps[:, j], -bound_ps[:, j],
61                    color = "dimgray",

```

```

62         alpha=0.5)
63     ax.plot(t_pred, dX,
64             linestyle="-", linewidth=0,
65             marker=".", markersize=2,
66             color = colors.get(j%3))
67     ax.grid(True, alpha=0.3)
68     ax.set_title(f" $\pm 3\sigma$  {state_names[j]}")
69
70     plt.figlegend(["Pre-Update Bounds", "Post-Update Bounds", "Differences"])
71     fig.supxlabel("Time [s]")
72     fig.supylabel("Prediction (Centered at 0)")
73     fig.suptitle(" $\Delta\mu_{x,k}$  Between Prediction Covariance Bounds ( $\pm 3\sigma$ )")
74     fig.tight_layout()
75
76     return fig

```

./src/p05.py

```

1  # libs
2  import numpy as np
3  import scipy as sp
4  import matplotlib.pyplot as plt
5
6  # src
7  from measurements import *
8  from helpers import *
9
10 def  $\alpha$ (data: measurements.Measurement, t: list, resid_pre_hist: np.ndarray, resid_post_hist:
    ↪ np.ndarray) -> plt.Figure:
11     # extract arrays from data
12     t = np.asarray(data.t, dtype=float)
13     i = np.asarray(data.i, dtype=float)
14
15     # shift epoch so t[0] = 0
16     t0 = t.min()
17     t = t - t0
18
19     # sort by time
20     order = np.argsort(t, kind="mergesort")
21     t, i = t[order], i[order]
22
23     # shift units for display
24     res_pre_r_m = resid_pre_hist[:, 0] * 1e3 # [km] -> [m]
25     res_post_r_m = resid_post_hist[:, 0] * 1e3 # [km] -> [m]
26     res_pre_rr_cms = resid_pre_hist[:, 1] * 1e5 # [km/s] -> [cm/s]
27     res_post_rr_cms = resid_post_hist[:, 1] * 1e5 # [km/s] -> [cm/s]
28
29     # station info
30     labels = {0: "DSN #0 Goldstone", 1: "DSN #1 Madrid", 2: "DSN #2 Canberra"}

```



```

31 colors = {0: "r", 1: "g", 2: "b"}
32
33 fig, axs = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=False, figsize=(12, 8))
34
35 for stn in np.unique(i):
36     m = (i == stn)
37
38     # range residuals
39     # axs[0].plot(t[m], res_pre_r_m[m],
40     #             linestyle="-", linewidth=0,
41     #             marker=".", markersize=2,
42     #             color=colors.get(int(stn)),
43     #             label=labels.get(int(stn), f"DSN #{int(stn)}"))
44     axs[0].plot(t[m], res_post_r_m[m],
45                 linestyle="-", linewidth=0,
46                 marker=".", markersize=2,
47                 color=colors.get(int(stn)),
48                 label=labels.get(int(stn), f"DSN #{int(stn)}"))
49     axs[0].grid(True, alpha=0.3)
50     axs[0].legend(loc = "upper right")
51     axs[0].set_ylabel("Range Residuals [km]")
52
53     # range-rate residuals
54     # axs[1].plot(t[m], res_pre_rr_cms[m],
55     #             linestyle="-", linewidth=0,
56     #             marker=".", markersize=2,
57     #             color=colors.get(int(stn)),
58     #             label=labels.get(int(stn), f"DSN #{int(stn)}"))
59     axs[1].plot(t[m], res_post_rr_cms[m],
60                 linestyle="-", linewidth=0,
61                 marker=".", markersize=2,
62                 color=colors.get(int(stn)),
63                 label=labels.get(int(stn), f"DSN #{int(stn)}"))
64     axs[1].grid(True, alpha=0.3)
65     axs[1].legend(loc = "upper right")
66     axs[1].set_ylabel("Range-Rate Residuals [km/s]")
67
68     # measurement noise reference bands
69     # axs[0].fill_between(t, -3.0, +3.0,
70     #                    color = "gray",
71     #                    alpha=0.5)
72     # axs[1].fill_between(t, -3.0, +3.0,
73     #                    color = "gray",
74     #                    alpha=0.5)
75
76 fig.supxlabel("Time [s]")
77 fig.suptitle("DSN Measurement Residuals vs. Time [s]")
78 fig.tight_layout()

```

```

79
80     return fig
81
82 def b(resid_post_hist: np.ndarray, nis_pre: np.ndarray) -> [float, float, float]:
83     # shift units
84     res_post_r_m = resid_post_hist[:, 0] * 1e3 # [km] -> [m]
85     res_post_rr_cms = resid_post_hist[:, 1] * 1e5 # [km/s] -> [cm/s]
86
87     rms_r_m = np.sqrt(np.mean(res_post_r_m**2))
88     rms_rr_cms = np.sqrt(np.mean(res_post_rr_cms**2))
89     nis_mean = np.mean(nis_pre)
90
91     return rms_r_m, rms_rr_cms, nis_mean
92
93 def c(t_pred: list, X_plus_hist: np.ndarray, P_plus_hist: np.ndarray) -> plt.Figure:
94     sigma_ps = np.sqrt(np.clip(np.stack([np.diag(P) for P in P_plus_hist]), 0.0, np.inf)) # Nx6
95     bound_ps = 3.0 * sigma_ps # Nx6
96
97     state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]
98     colors = {0: "r", 1: "g", 2: "b"}
99
100     fig, axs = plt.subplots(nrows=3, ncols=2, sharex=True, sharey=False, figsize=(12, 8))
101
102     for j in range(len(state_names)):
103         ax = axs[j%3, int(j/3)]
104         ax.fill_between(t_pred, X_plus_hist[:, j] - bound_ps[:, j], X_plus_hist[:, j] +
105             ↪ bound_ps[:, j],
106             color = "dimgray",
107             alpha=0.5)
108         ax.plot(t_pred, X_plus_hist[:, j],
109             linestyle="-", linewidth=0,
110             marker=".", markersize=2,
111             color = colors.get(j%3))
112         ax.grid(True, alpha=0.3)
113         ax.set_title(f"{state_names[j]}")
114
115     plt.figlegend(["±3σ Bounds", "Estimate"])
116     fig.supxlabel("Time [s]")
117     fig.supylabel("State Estimates")
118     fig.suptitle("Final State Estimates (ECI, [km] & [km/s])")
119     fig.tight_layout()
120
121     return fig
122
123 def d(X_plus_hist: np.ndarray, P_plus_hist: np.ndarray) -> [np.ndarray, np.ndarray]:
124     X_final = X_plus_hist[-1]
125     sigma_final = np.sqrt(np.clip(np.diag(P_plus_hist[-1]), 0.0, np.inf)) # 1σ
126     state_names = ["x [km]", "y [km]", "z [km]", "vx [km/s]", "vy [km/s]", "vz [km/s]"]

```

```
126
127     return X_final,  $\sigma$ _final, state_names
```

./src/helpers/propagators.py

```
1  # libs
2  import numpy as np
3  import scipy as sp
4  import matplotlib.pyplot as plt
5
6  # src
7  from measurements import *
8  from .system import *
9
10 def kf(data: measurements.Measurement, X0_plus: np.ndarray, P0_plus: np.ndarray,  $\sigma$ _a: float,
    ↪  $\mu$ : float) -> [list, np.ndarray, np.ndarray]:
11     # extract arrays from data
12     t = np.asarray(data.t, dtype=float)
13
14     # shift epoch so t[0] = 0
15     t0 = t.min()
16     t = t - t0
17
18     # sort by time (for propagation)
19     order = np.argsort(t)
20     t = t[order]
21
22     # prediction storage
23     X_minus_hist = np.zeros((len(t), 6))
24     P_minus_hist = np.zeros((len(t), 6, 6))
25
26     X_plus = X0_plus.copy()
27     P_plus = P0_plus.copy()
28     t_prev = 0.0
29
30     for k, t_k in enumerate(t):
31         dt = t_k - t_prev
32
33         if dt > 0:
34              $\Phi$ 0 = np.eye(6)
35             y0 = np.concatenate([X_plus,  $\Phi$ 0.reshape(-1)])
36
37             sol = sp.integrate.solve_ivp(
38                 fun=lambda tt, yy: eom_with_stm(tt, yy,  $\mu$ ),
39                 t_span=(t_prev, t_k),
40                 y0=y0,
41                 t_eval=[t_k],
42                 rtol=1e-10,
43                 atol=1e-12,
```

```

44         method="DOP853",
45     )
46     y_k = sol.y[:, -1]
47     X_minus = y_k[0:6]
48      $\Phi$  = y_k[6:].reshape(6,6)
49
50     # prediction-only, so no process update:
51     Q_k = Q_discrete(dt,  $\sigma_a$ )
52     P_minus =  $\Phi$  @ P_plus @  $\Phi$ .T + Q_k
53     else:
54         # multiple measurements at same time tag: no propagation
55         X_minus = X_plus
56         P_minus = P_plus
57
58     X_minus_hist[k, :] = X_minus
59     P_minus_hist[k, :] = P_minus
60
61     # prediction-only, so no measurement update:
62     X_plus = X_minus
63     P_plus = P_minus
64     t_prev = t_k
65
66     return t, X_minus_hist, P_minus_hist
67
68 def ekf(data: measurements.Measurement, X0_plus: np.ndarray, P0_plus: np.ndarray, R0:
↳ np.ndarray, stations: dict,  $\sigma_a$ : float,  $\mu$ : 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]:
69     # extract arrays from data
70     t = np.asarray(data.t, dtype=float)
71     i = np.asarray(data.i, dtype=float)
72      $\rho$  = np.asarray(data.p, dtype=float)
73     dp = np.asarray(data.dp, dtype=float)
74
75     # station positions
76     R_ecef = site_ecef(stations, RE)
77
78     # shift epoch so t[0] = 0
79     t0 = t.min()
80     t = t - t0
81
82     # time steps
83     N = len(t)
84
85     # sort by time (for propagation)
86     order = np.argsort(t, kind="mergesort")
87     t, i = t[order], i[order]
88      $\rho$ _meas, dp_meas =  $\rho$ [order], dp[order]

```

```

89
90     # prediction storage
91     X_minus_hist = np.zeros((N, 6))
92     P_minus_hist = np.zeros((N, 6, 6))
93     X_plus_hist = np.zeros((N, 6))
94     P_plus_hist = np.zeros((N, 6, 6))
95
96     yhat_minus_hist = np.zeros((N, 2))
97     yhat_plus_hist = np.zeros((N, 2))
98     resid_pre_hist = np.zeros((N, 2)) # y - h(X_minus)
99     resid_post_hist = np.zeros((N, 2)) # y - h(X_plus)
100     nis_pre_hist = np.zeros(N)
101
102     X_plus = X0_plus.copy()
103     P_plus = P0_plus.copy()
104     t_prev = t[0]
105
106     for k, t_k in enumerate(t):
107         dt = t_k - t_prev
108
109         if dt > 0:
110             Φ0 = np.eye(6)
111             y0 = np.concatenate([X_plus, Φ0.reshape(-1)])
112
113             sol = sp.integrate.solve_ivp(
114                 fun=lambda tt, yy: eom_with_stm(tt, yy, μ),
115                 t_span=(t_prev, t_k),
116                 y0=y0,
117                 t_eval=[t_k],
118                 rtol=1e-10,
119                 atol=1e-12,
120                 method="DOP853",
121             )
122             y_k = sol.y[:, -1]
123             X_minus = y_k[0:6]
124             Φ = y_k[6:].reshape(6,6)
125
126             # process update
127             Q_k = Q_discrete(dt, σ_a)
128             P_minus = Φ @ P_plus @ Φ.T + Q_k
129         else:
130             # multiple measurements at same time tag: no propagation
131             X_minus = X_plus
132             P_minus = P_plus
133
134         # save pre-update
135         X_minus_hist[k, :] = X_minus
136         P_minus_hist[k, :, :] = P_minus

```

```

137
138     # measurement update
139     y_k = np.array([p_meas[k], dp_meas[k]])
140     yhat_minus, H_k = meas_and_jacobian(X_minus, i[k], t_k, R_ecef, OMEGA_E, GAMMA0)
141     yhat_minus_hist[k] = yhat_minus
142     resid_pre_hist[k] = y_k - yhat_minus
143
144     # innovation
145     v = y_k - yhat_minus
146     S = H_k @ P_minus @ H_k.T + R0
147     nis_pre = float(v.T @ np.linalg.inv(S) @ v)
148     nis_pre_hist[k] = nis_pre
149
150     # kalman gain
151     # K = P_minus @ H_k.T @ np.linalg.pinv(S)
152     K = P_minus @ H_k.T @ np.linalg.solve(S, np.eye(S.shape[0]))
153
154     # update state
155     X_plus = X_minus + K @ v
156
157     # joseph covariance update
158     I6 = np.eye(6)
159     P_plus = (I6 - K @ H_k) @ P_minus @ (I6 - K @ H_k).T + K @ R0 @ K.T
160
161     # save post-update
162     X_plus_hist[k, :] = X_plus
163     P_plus_hist[k, :, :] = P_plus
164
165     # post-fit measurement
166     yhat_plus, _ = meas_and_jacobian(X_plus, i[k], t_k, R_ecef, OMEGA_E, GAMMA0)
167     yhat_plus_hist[k] = yhat_plus
168     resid_post_hist[k] = y_k - yhat_plus
169
170     # advance time
171     t_prev = t_k
172
173     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

```

1 # libs
2 import numpy as np
3 import scipy as sp
4 import matplotlib.pyplot as plt
5
6 # src
7 from measurements import *
8

```

```

9  def coe2rv(oe: list, μ:float) -> [np.ndarray, np.ndarray]:
10     # a in km, angles in rad
11     a, e, i, ω, Ω, v = [float(x) for x in oe]
12
13     # Semi-latus rectum
14     p = a * (1 - e**2)
15
16     # Perifocal coordinates (PQW frame)
17     r_pf = (p / (1 + e*np.cos(v))) * np.array([np.cos(v), np.sin(v), 0.0])
18     v_pf = np.sqrt(μ/p) * np.array([-np.sin(v), e + np.cos(v), 0.0])
19
20     # Rotation from PQW -> ECI (3-1-3 sequence)
21     R_pqw_eci = sp.spatial.transform.Rotation.from_euler("ZXZ", [Ω, i, ω])
22     r = R_pqw_eci.as_matrix() @ r_pf
23     v = R_pqw_eci.as_matrix() @ v_pf
24
25     return r, v
26
27 def A_matrix(r: np.ndarray, μ: float) -> np.ndarray:
28     rnorm = np.linalg.norm(r)
29     I3 = np.eye(3)
30     dadr = -μ * (I3/(rnorm**3) - 3.0*np.outer(r, r)/(rnorm**5))
31     A = np.block([
32         [np.zeros((3,3)), I3],
33         [dadr, np.zeros((3,3))]
34     ])
35
36     return A
37
38 def Q_discrete(dt: float, σ_a: float) -> np.ndarray:
39     if dt ≤ 0.0:
40         return np.zeros((6,6))
41
42     q_a = σ_a**2
43     I3 = np.eye(3)
44
45     Q = q_a * np.block([
46         [(dt**3/3.0)*I3, (dt**2/2.0)*I3],
47         [(dt**2/2.0)*I3, (dt)*I3]
48     ])
49
50     return Q
51
52 def eom_with_stm(t: float, y: list, μ: float) -> np.ndarray:
53     # y = [r(3), v(3), Phi:flat(36)]
54     r = y[0:3]
55     v = y[3:6]
56     Φ = y[6:].reshape(6,6)

```

```

57     rnorm = np.linalg.norm(r)
58     a = -μ * r / (rnorm**3)
59
60     A = A_matrix(r, μ)
61     dΦ = A @ Φ
62
63     return np.concatenate([v, a, dΦ.reshape(-1)])
64
65
66 def site_ecef(stations: dict, RE: float) -> dict:
67     R_ecef = {}
68
69     for idx, (φ, λ) in stations.items():
70         R_ecef[idx] = RE * np.array([
71             np.cos(φ)*np.cos(λ),
72             np.cos(φ)*np.sin(λ),
73             np.sin(φ)
74         ])
75
76     return R_ecef
77
78 def site_eci(station_idx: int, t: float, R_ecef: dict, OMEGA_E: float, GAMMA0: float) ->
↪ [np.ndarray, np.ndarray]:
79     γ = GAMMA0 + OMEGA_E * t
80     ω = np.array([0.0, 0.0, OMEGA_E])
81
82     R_ecef_eci = sp.spatial.transform.Rotation.from_euler("Z", γ)
83     R_site = R_ecef_eci.as_matrix() @ R_ecef[int(station_idx)]
84
85     dR_site = np.cross(ω, R_site)
86
87     return R_site, dR_site
88
89 def meas_and_jacobian(X: list, station_idx: int, t: float, R_ecef: dict, OMEGA_E: float,
↪ GAMMA0: float) -> [np.ndarray, np.ndarray]:
90     r = X[0:3]
91     v = X[3:6]
92     R_site, dR_site = site_eci(station_idx, t, R_ecef, OMEGA_E, GAMMA0)
93
94     p_vec = r - R_site
95     ρ = np.linalg.norm(p_vec)
96     u = p_vec / ρ
97
98     v_rel = v - dR_site
99     dp = u @ v_rel
100
101     yhat = np.array([ρ, dp])
102     I3 = np.eye(3)

```



```
103
104     # H (2x6)
105     H_p_r = u.reshape(1,3)
106     H_p_v = np.zeros((1,3))
107     H_dp_r = (1.0/p) * (v_rel.reshape(1,3) @ (I3 - np.outer(u, u)))
108     H_dp_v = u.reshape(1,3)
109
110     H = np.block([
111         [H_p_r, H_p_v],
112         [H_dp_r, H_dp_v],
113     ])
114
115     return yhat, H
```

./src/measurements/measurements.py

```
1 # libs
2 from typing import NamedTuple
3 import numpy as np
4
5 class Measurement(NamedTuple):
6     t: np.ndarray
7     i: np.ndarray
8     p: np.ndarray
9     dp: np.ndarray
```

./src/measurements/numpy_data.py

```
1 # libs
2 import numpy as np
3
4 # src
5 from .measurements import *
6
7 def raw_load(file_path: str) -> np.ndarray:
8     try:
9         raw_data = np.load(file_path, allow_pickle=True)
10    except NameError:
11        print(f"Could not load data from {file_path}")
12
13    return raw_data
14
15 def parse(raw_data: np.ndarray) -> Measurement:
16     data = Measurement(raw_data[:, 0], raw_data[:, 1], raw_data[:, 2], raw_data[:, 3])
17
18     return data
19
20 def load(file_path: str) -> Measurement:
21     raw_data = raw_load(file_path)
```

```
22     data = parse(raw_data)
23
24     return data
```