AER1513 Assignment 3 Report

Kevin Hu

December 2023

1 Introduction

Here we use the Batch Gauss-Newton method to estimate the sensor head's position and orientation using both the IMU and stereo camera measurements.

Based on the histograms, the assumption of zero-mean Gaussian noise is reasonable. There may be some error in the position and orientation of the final result and the resulting uncertainties due to the fact that the v_l and v_r pixel errors are shifted to left, and all the errors seem to be higher at zero than a normal Gaussian. Due to the variable time step T_k, the variance $\mathbf{Q_k}$ is:

$$\mathbf{Q_k} = T_k^2 \begin{bmatrix} 0.051302^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.045550^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.028137^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.095125^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.130393^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.417991^2 \end{bmatrix}$$

$$\mathbf{Q_k} = T_k^2 \begin{bmatrix} 0.002632 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.002075 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.000792 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.009049 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.017002 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.174716 \end{bmatrix}$$

The variance $\mathbf{R_k^j}$ is:

$$\mathbf{R_k^j} = \begin{bmatrix} 6.164784^2 & 0 & 0 & 0 \\ 0 & 11.395366^2 & 0 & 0 \\ 0 & 0 & 6.477907^2 & 0 \\ 0 & 0 & 0 & 11.511221^2 \end{bmatrix}$$

$$\mathbf{R_k^j} = \begin{bmatrix} 38.004562 & 0 & 0 & 0 \\ 0 & 129.854366 & 0 & 0 \\ 0 & 0 & 41.963279 & 0 \\ 0 & 0 & 0 & 132.508209 \end{bmatrix}$$

Here we try to use the motion model and part of the observation model (part without the stereo camera) in the Lecture 11 slides about point cloud tracking (pg 23-27). We use it in conjunction with the stereo camera observation model outlined in the assignment paper. We also use the Batch Map algorithm outlined in the Lecture 11 slides (pg 34-45) for state estimation. An expression for the batch nonlinear least-squares objective function that is minimized using the Gauss-Newton method can be summarized as follows:

Define matrix $\mathbf{D}^\top = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$ and vector $\mathbf{t} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ which are used to select the rotation matrix and translation vector from the 4 by 4 pose matrix.

The motion error vector is:

$$\mathbf{e_v}(\mathbf{x}) = \begin{bmatrix} ln(\hat{\mathbf{T}}_{k1}\mathbf{T}_{\mathbf{k1}}^{\mathbf{-1}})^\vee \\ ln(exp(\Delta t_{k1+1}\varpi_{\mathbf{k1+1}}^\wedge)\hat{\mathbf{T}}_{\mathbf{k1}}\mathbf{T}_{\mathbf{k1+1}}^{\mathbf{-1}})^\vee \\ ln(exp(\Delta t_{k1+2}\varpi_{\mathbf{k1+2}}^\wedge)\hat{\mathbf{T}}_{\mathbf{k1+1}}\mathbf{T}_{\mathbf{k1+2}}^{\mathbf{-1}})^\vee \\ \vdots \\ ln(exp(\Delta t_{k2}\varpi_{\mathbf{k2}}^\wedge)\hat{\mathbf{T}}_{\mathbf{k2-1}}\mathbf{T}_{\mathbf{k2}}^{\mathbf{-1}})^\vee \end{bmatrix}$$

The measurement error term is:

$$\begin{aligned} \mathbf{e_{y,jk}}(\mathbf{x}) &= \mathbf{y_{jk}} - \mathbf{g}(\mathbf{p_{ck}^{pjck}}) \\ &= \mathbf{y_{jk}} - \mathbf{g}(\mathbf{C_{cv}}(\mathbf{D}^\top\mathbf{T_k}\mathbf{D}(\rho_{\mathbf{i}}^{\mathbf{pji}} - \mathbf{D}^\top\mathbf{T_k}\mathbf{t}) - \rho_{\mathbf{v}}^{\mathbf{cv}})) \end{aligned}$$

where

$$\mathbf{p_{ck}^{pjck}} = \begin{bmatrix} x \\ y \\ x \end{bmatrix} = \mathbf{C_{cv}}(\mathbf{D}^\top\mathbf{T_k}\mathbf{D}(\rho_{\mathbf{i}}^{\mathbf{pji}} - \mathbf{D}^\top\mathbf{T_k}\mathbf{t}) - \rho_{\mathbf{v}}^{\mathbf{cv}})$$

The term

$$\mathbf{g}(\mathbf{p_{ck}^{pjck}}) = (\mathbf{t}^\top\mathbf{p_{ck}^{pjck}})^{-\mathbf{1}} \begin{bmatrix} f_u & 0 & 0 \\ 0 & f_v & 0 \\ f_u & 0 & 0 \\ 0 & f_v & 0 \end{bmatrix} \mathbf{p_{ck}^{pjck}} - \begin{bmatrix} 0 \\ 0 \\ f_u b \\ 0 \end{bmatrix} + \begin{bmatrix} c_u \\ c_v \\ c_u \\ c_v \end{bmatrix}$$

The measurement error vector for time step k can be written as:

$$\mathbf{e_{y,k}}(\mathbf{x}) = \begin{bmatrix} \mathbf{e_{y,1k}}(\mathbf{x}) \\ \vdots \\ \mathbf{e_{y,Mk}}(\mathbf{x}) \end{bmatrix}$$

The measurement error covariance for time step k can be written as: $\mathbf{R_k} = \mathbf{diag}(\mathbf{R_{1k}}, \mathbf{R_{2k}}, \ldots, \mathbf{R_{Mk}})$ where $\mathbf{R_{jk}}$ was calculated earlier.

The measurement error covariance vector is:

$$\mathbf{e_y}(\mathbf{x}) = \begin{bmatrix} \mathbf{e_{y,k1}}(\mathbf{x}) \\ \mathbf{e_{y,k1+1}}(\mathbf{x}) \\ \vdots \\ \mathbf{e_{y,k2}}(\mathbf{x}) \end{bmatrix}$$

The objective function error vector is:

$$\mathbf{e}(\mathbf{x}) = \begin{bmatrix} \mathbf{e_v}(\mathbf{x}) \\ \mathbf{e_y}(\mathbf{x}) \end{bmatrix}$$

The objective function error covariance is:

$$\mathbf{W} = \mathbf{diag}(\check{\mathbf{P}}_{\mathbf{k1}}, \mathbf{Q_{k1+1}}, \mathbf{Q_{k1+2}}, \ldots, \mathbf{Q_{k2}}, \mathbf{R_{k1}}, \mathbf{R_{k1+1}}, \ldots, \mathbf{R_{k2}})$$

The inverse objective function error covariance is:

$$\mathbf{W^{-1}} = \mathbf{diag}(\check{\mathbf{P}}^{-1}_{\mathbf{k1}}, \mathbf{Q^{-1}_{k1+1}}, \mathbf{Q^{-1}_{k1+2}}, \ldots, \mathbf{Q^{-1}_{k2}}, \mathbf{R^{-1}_{k1}}, \mathbf{R^{-1}_{k1+1}}, \ldots, \mathbf{R^{-1}_{k2}})$$

The final expression of the batch nonlinear least-squares objective function is:

$$\mathbf{J}(\mathbf{x}) = \frac{\mathbf{1}}{\mathbf{2}}\mathbf{e}^\top \mathbf{W^{-1}} \mathbf{e}$$

To set up the Gauss-Newton algorithm, the error terms need to be linearized about an operating point for each pose $\mathbf{T_{op,k}}$ which is the current trajectory guess that will be iteratively improved. $\mathbf{T_k} = \mathbf{exp}(\epsilon_{\mathbf{k}}^\wedge)\mathbf{T_{op,k}}$, where ϵ_k is the perturbation to the current guess that is optimized at each iteration. The short hand $\mathbf{x_{op}} = (\mathbf{T_{op,k1}}, \mathbf{T_{op,k1+1}}, \ldots, \mathbf{T_{op,k2}})$ will be used as the operating point of the entire trajectory.

For the first linearized input error we have:

$$\mathbf{e_{v,k1}}(\mathbf{x}) = \mathbf{e_{v,k1}}(\mathbf{x_{op}}) - \mathcal{J}(-\mathbf{e_{v,k1}}(\mathbf{x_{op}}))^{\mathbf{-1}}\epsilon_{\mathbf{k1}}$$

where the matrix $\mathbf{E_{k1}} = \mathcal{J}(-\mathbf{e_{v,k1}}(\mathbf{x_{op}}))^{\mathbf{-1}}$.

For the later linearized input errors we have:

$$\mathbf{e_{v,k}}(\mathbf{x}) = \mathbf{e_{v,k}}(\mathbf{x_{op}}) + \mathscr{J}(-\mathbf{e_{v,k}}(\mathbf{x_{op}}))^{-\mathbf{1}}\mathbf{Ad}(\mathbf{T_{op,k}}\mathbf{T_{op,k-1}^{-1}})\epsilon_{\mathbf{k-1}} - \mathscr{J}(-\mathbf{e_{v,k}}(\mathbf{x_{op}}))^{-\mathbf{1}}\epsilon_{\mathbf{k}}$$

where the matrix $\mathbf{F_{k-1}} = \mathscr{J}(-\mathbf{e_{v,k}}(\mathbf{x_{op}}))^{-\mathbf{1}}\mathbf{Ad}(\mathbf{T_{op,k}}\mathbf{T_{op,k-1}^{-1}})$ and the matrix $\mathbf{E_k} = \mathscr{J}(-\mathbf{e_{v,k}}(\mathbf{x_{op}}))^{-\mathbf{1}}$.

Next we linearize the measurement errors. First we linearize the position of the point on the ground:

$$\begin{aligned}
\mathbf{p_{ck}^{pjck}} &= \mathbf{C_{cv}}(\mathbf{D}^\top\mathbf{T_k}\mathbf{D}(\rho_{\mathbf{i}}^{\mathbf{pji}} - \mathbf{D}^\top\mathbf{T_k}\mathbf{t}) - \rho_{\mathbf{v}}^{\mathbf{cv}}) \\
&\approx \mathbf{C_{cv}}(\mathbf{D}^\top(\mathbf{1} + \epsilon_{\mathbf{k}}^\wedge)\mathbf{T_{op,k}}\mathbf{D}(\rho_{\mathbf{i}}^{\mathbf{pji}} - \mathbf{D}^\top(\mathbf{1} + \epsilon_{\mathbf{k}}^\wedge)\mathbf{T_{op,k}}\mathbf{t}) - \rho_{\mathbf{v}}^{\mathbf{cv}}) \\
&= \mathbf{C_{cv}}((\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{D} + \mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{D})(\rho_{\mathbf{i}}^{\mathbf{pji}} - \mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{t} - \mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{t}) - \rho_{\mathbf{v}}^{\mathbf{cv}}) \\
&= \mathbf{C_{cv}}(\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{D}\rho_{\mathbf{i}}^{\mathbf{pji}} - \mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{t} - \mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{t} \\
&\qquad + \mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{D}\rho_{\mathbf{i}}^{\mathbf{pji}} - \mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{t} - \mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{t} - \rho_{\mathbf{v}}^{\mathbf{cv}}) \\
&\approx \mathbf{p_{ck}^{pjck}}(\mathbf{T_{op,k}}) + \mathbf{C_{cv}}(-\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{t} + \mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{D}\rho_{\mathbf{i}}^{\mathbf{pji}} - \mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{t}) \\
&= \mathbf{p_{ck}^{pjck}}(\mathbf{T_{op,k}}) + \mathbf{C_{cv}}(-\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top(\mathbf{T_{op,k}}\mathbf{t})^{\odot} + \mathbf{D}^\top(\mathbf{T_{op,k}}\mathbf{D}\rho_{\mathbf{i}}^{\mathbf{pji}})^{\odot} \\
&\qquad - \mathbf{D}^\top(\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{t})^{\odot})\epsilon_{\mathbf{k}}
\end{aligned}$$

Where the $\mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\epsilon_{\mathbf{k}}^\wedge\mathbf{T_{op,k}}\mathbf{t}$ term was assumed to be negligible since it is quadratic in $\epsilon_k^\wedge$.

Let $\mathbf{P_{ck}}$ denote the expression:

$$\mathbf{P_{ck}} = \mathbf{C_{cv}}(-\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top(\mathbf{T_{op,k}}\mathbf{t})^{\odot} + \mathbf{D}^\top(\mathbf{T_{op,k}}\mathbf{D}\rho_{\mathbf{i}}^{\mathbf{pji}})^{\odot} - \mathbf{D}^\top(\mathbf{T_{op,k}}\mathbf{D}\mathbf{D}^\top\mathbf{T_{op,k}}\mathbf{t})^{\odot})$$

As a result:

$$\mathbf{p_{ck}^{pjck}} \approx \mathbf{p_{ck}^{pjck}}(\mathbf{T_{op,k}}) + \mathbf{P_{ck}}\epsilon_{\mathbf{k}}$$

Next, we linearize the stereo camera model by taking the Jacobian of the model $\mathbf{g}(\mathbf{p_{ck}^{pjck}}) = \mathbf{g}(\begin{bmatrix} x \\ y \\ z \end{bmatrix})$ with respect to the variables x, y and z. Let the Jacobian of the stereo camera model be referred to as the matrix $\mathbf{S}$.

$$\mathbf{S} = \begin{bmatrix} \frac{\partial g_1}{\partial x} & \frac{\partial g_1}{\partial y} & \frac{\partial g_1}{\partial z} \\ \frac{\partial g_2}{\partial x} & \frac{\partial g_2}{\partial y} & \frac{\partial g_2}{\partial z} \\ \frac{\partial g_3}{\partial x} & \frac{\partial g_3}{\partial y} & \frac{\partial g_3}{\partial z} \\ \frac{\partial g_4}{\partial x} & \frac{\partial g_4}{\partial y} & \frac{\partial g_4}{\partial z} \end{bmatrix}$$

$$\mathbf{S} = \begin{bmatrix} \frac{f_u}{z} & 0 & \frac{-f_u x}{z^2} \\ 0 & \frac{f_v}{z} & \frac{-f_v y}{z^2} \\ \frac{f_u}{z} & 0 & \frac{-f_u x}{z^2} + \frac{f_u b}{z^2} \\ 0 & \frac{f_v}{z} & \frac{-f_v y}{z^2} \end{bmatrix}$$

Due to the chain rule we are able to relate the perturbation's affect on x, y, z coordinates of the point on the floor with what the stereo camera model sees. Take $\mathbf{p_{ck}^{pjck}}(\mathbf{T_{op,k}})$ as the operating point for the stereo camera model with perturbation $\mathbf{P_{ck}}\epsilon_\mathbf{k}$. Then the linearized error term for one measurement will be:

$$\mathbf{e_{y,jk}}(\mathbf{x}) = \mathbf{y_{jk}} - (\mathbf{g}(\mathbf{p_{ck}^{pjck}}(\mathbf{T_{op,k}})) + \mathbf{S}|_{\mathbf{p_{ck}^{pjck}}(\mathbf{T_{op,k}})}\mathbf{P_{ck}}\epsilon_\mathbf{k})$$

Let measurement error at the operating point be $\mathbf{e_{y,jk}}(\mathbf{x_{op}}) = \mathbf{y_{jk}} - \mathbf{g}(\mathbf{p_{ck}^{pjck}}(\mathbf{T_{op,k}}))$ and the matrix $\mathbf{G_{jk}} = \mathbf{S}|_{\mathbf{p_{ck}^{pjck}}(\mathbf{T_{op,k}})}\mathbf{P_{ck}}$. We can stack all the point measurements at time k together so that $\mathbf{e_{y,k}}(\mathbf{x}) \approx \mathbf{e_{y,k}}(\mathbf{x_{op}}) - \mathbf{G_k}\epsilon_\mathbf{k}$ where $\mathbf{e_{y,k}}(\mathbf{x}) = \begin{bmatrix} \mathbf{e_{y,1k}}(\mathbf{x}) \\ \mathbf{e_{y,2k}}(\mathbf{x}) \\ \vdots \\ \mathbf{e_{y,Mk}}(\mathbf{x}) \end{bmatrix}$,

$\mathbf{e_{y,k}}(\mathbf{x_{op}}) = \begin{bmatrix} \mathbf{e_{y,1k}}(\mathbf{x_{op}}) \\ \mathbf{e_{y,2k}}(\mathbf{x_{op}}) \\ \vdots \\ \mathbf{e_{y,Mk}}(\mathbf{x_{op}}) \end{bmatrix}$ and $\mathbf{G_k} = \begin{bmatrix} \mathbf{G_{1k}} \\ \mathbf{G_{2k}} \\ \vdots \\ \mathbf{G_{Mk}} \end{bmatrix}$.

To set up the Gauss-Newton update for this problem, we define the following quantities:

$$\delta\mathbf{x} = \begin{bmatrix} \epsilon_{k1} \\ \epsilon_{k1+1} \\ \vdots \\ \epsilon_{k2} \end{bmatrix}, \mathbf{H} = \begin{bmatrix} \mathbf{E_{k1}} & & & & \\ -\mathbf{F_{k1}} & \mathbf{E_{k1+1}} & & & \\ & -\mathbf{F_{k1+1}} & \ddots & & \\ & & \ddots & \mathbf{E_{k2-1}} & \\ & & & -\mathbf{F_{k2-1}} & \mathbf{E_{k2}} \\ \mathbf{G_{k1}} & & & & \\ & \mathbf{G_{k1+1}} & & & \\ & & \mathbf{G_{k1+2}} & & \\ & & & \ddots & \\ & & & & \mathbf{G_{k2}} \end{bmatrix}, \mathbf{e}(\mathbf{x_{op}}) = \begin{bmatrix} \mathbf{e_{v,k1}}(\mathbf{x_{op}}) \\ \mathbf{e_{v,k1+1}}(\mathbf{x_{op}}) \\ \vdots \\ \mathbf{e_{v,k2-1}}(\mathbf{x_{op}}) \\ \mathbf{e_{v,k2}}(\mathbf{x_{op}}) \\ \mathbf{e_{y,k1}}(\mathbf{x_{op}}) \\ \mathbf{e_{y,k1+1}}(\mathbf{x_{op}}) \\ \vdots \\ \mathbf{e_{y,k2}}(\mathbf{x_{op}}) \end{bmatrix}$$

and $\mathbf{W} = \mathbf{diag}(\check{\mathbf{P}}_{\mathbf{k1}}, \mathbf{Q}_{\mathbf{k1+1}}, \mathbf{Q}_{\mathbf{k1+2}}, \dots, \mathbf{Q}_{\mathbf{k2}}, \mathbf{R}_{\mathbf{k1}}, \mathbf{R}_{\mathbf{k1+1}}, \dots, \mathbf{R}_{\mathbf{k2}})$.

The quadratic in terms of the perturbation is then $\mathbf{J}(\mathbf{x}) \approx \mathbf{J}(\mathbf{x}_{\mathbf{op}}) - \mathbf{b}^\top \delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^\top \mathbf{A}\delta\mathbf{x}$ where $\mathbf{A} = \mathbf{H}^\top\mathbf{W}^{-1}\mathbf{H}, \mathbf{b} = \mathbf{H}^\top\mathbf{W}^{-1}\mathbf{e}(\mathbf{x}_{\mathbf{op}})$ and $\mathbf{A}$ is a block tridiagonal matrix. Minimizing with respect to $\delta\mathbf{x}$, we have $\mathbf{A}\delta\mathbf{x}^* = \mathbf{b}$ for the optimal perturbation $\delta\mathbf{x}^* = \begin{bmatrix} \epsilon^*_{k1} \\ \epsilon^*_{k1+1} \\ \vdots \\ \epsilon^*_{k2} \end{bmatrix}$. Once we have the optimal perturbation we update the operating point using the update $\mathbf{T}_{\mathbf{op,k}} \leftarrow \mathbf{exp}(\epsilon^{*\wedge}_{\mathbf{k}})\mathbf{T}_{\mathbf{op,k}}$, which guarantees that $\mathbf{T}_{\mathbf{op,k}}$ stays in SE(3). Finally we can iterate the entire scheme to convergence.

2 Results

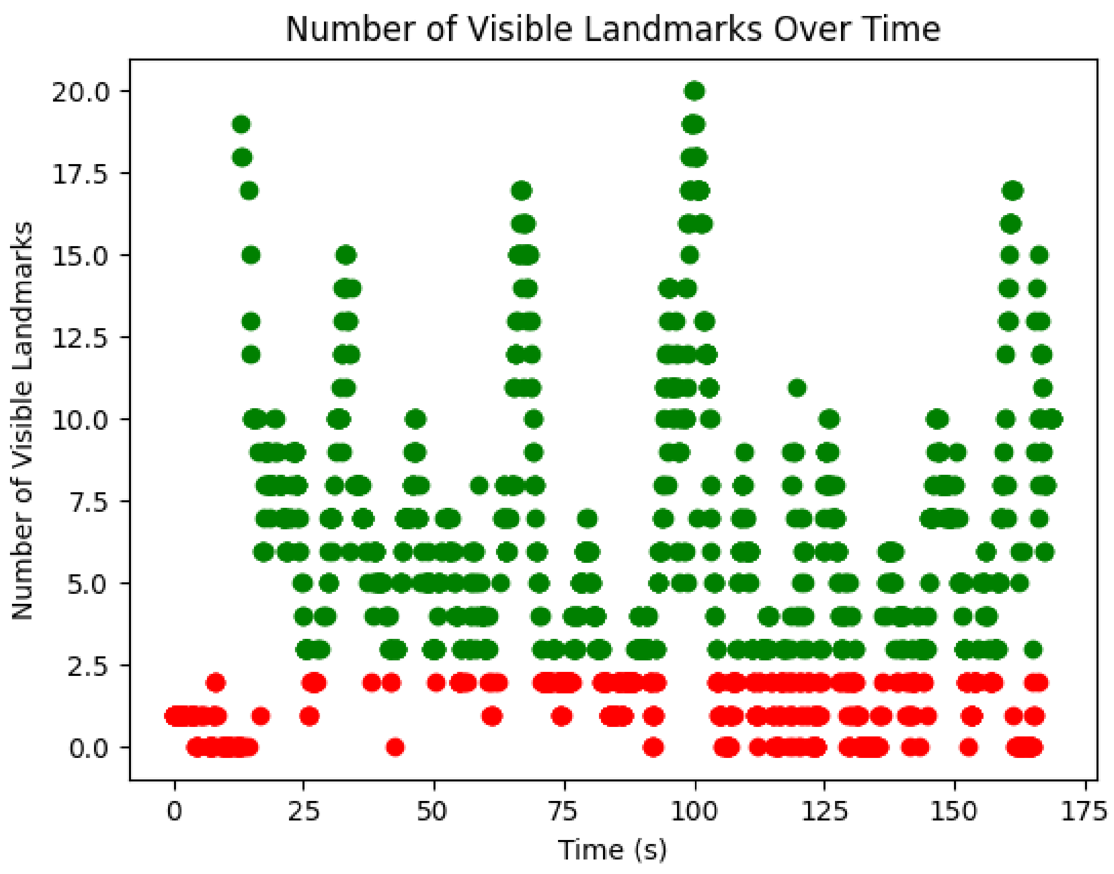
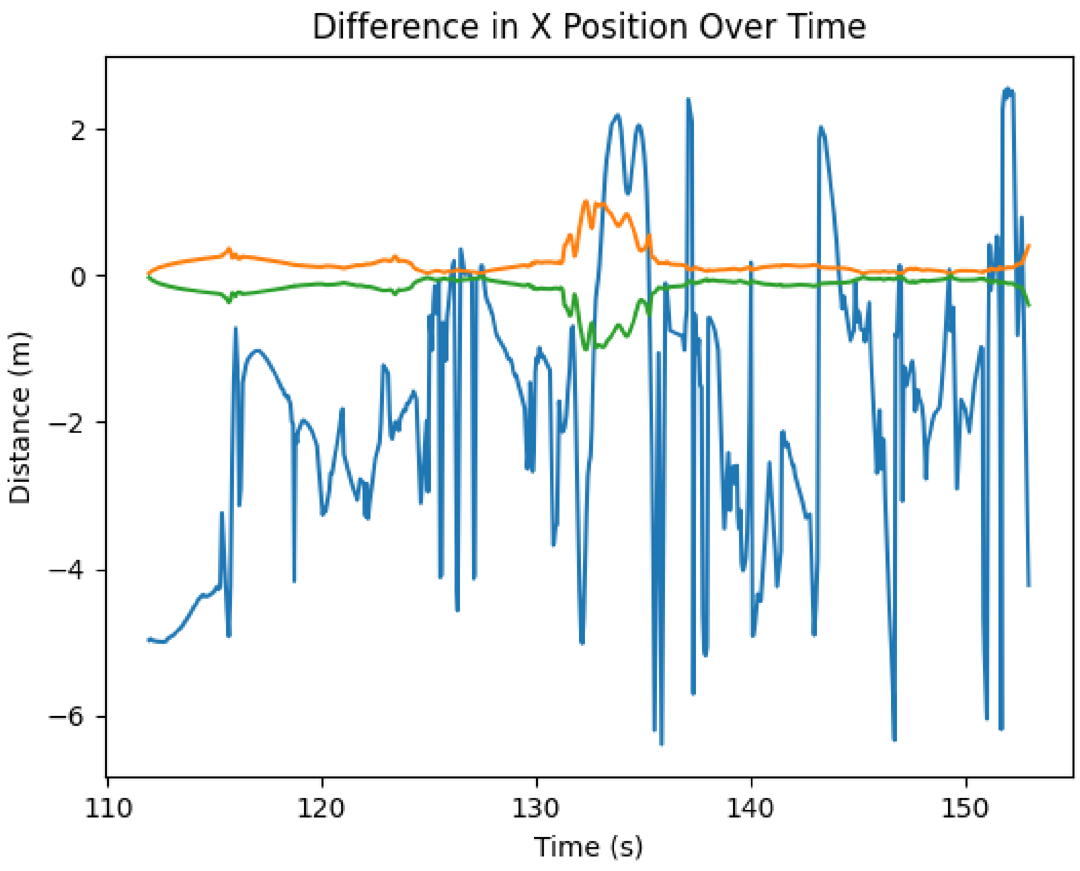
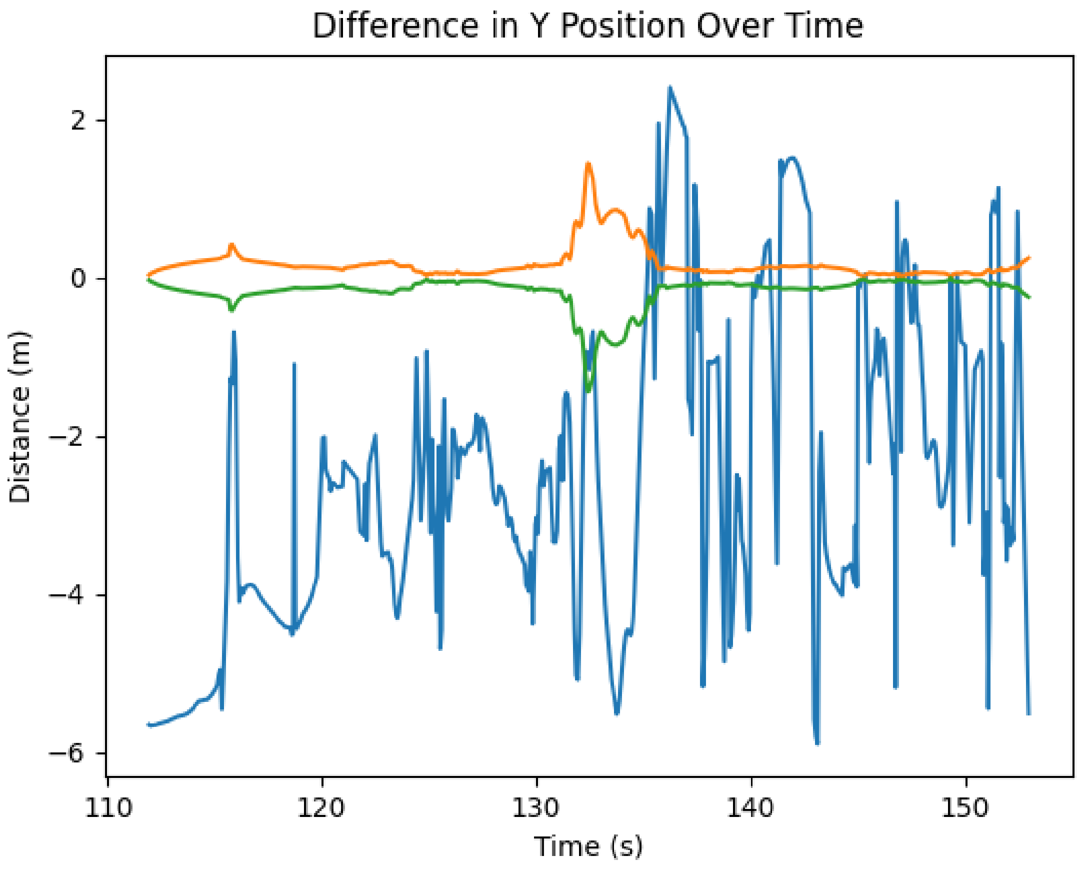
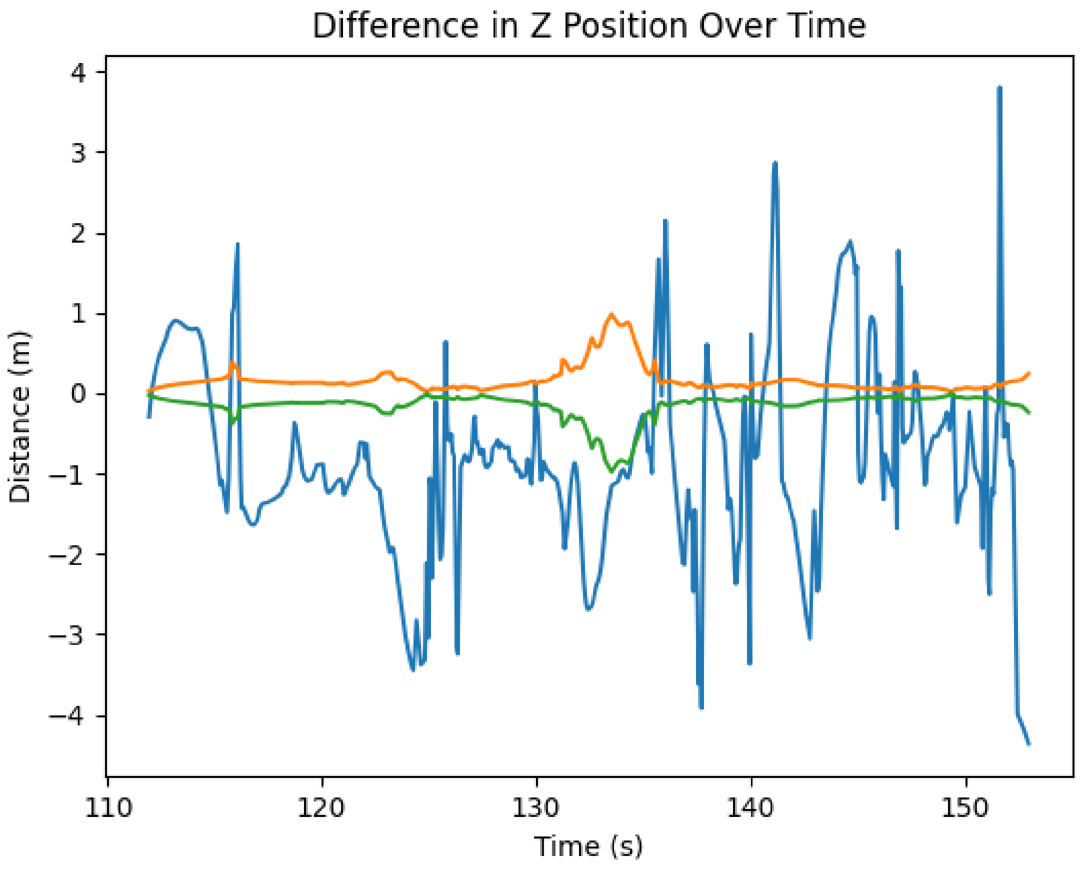


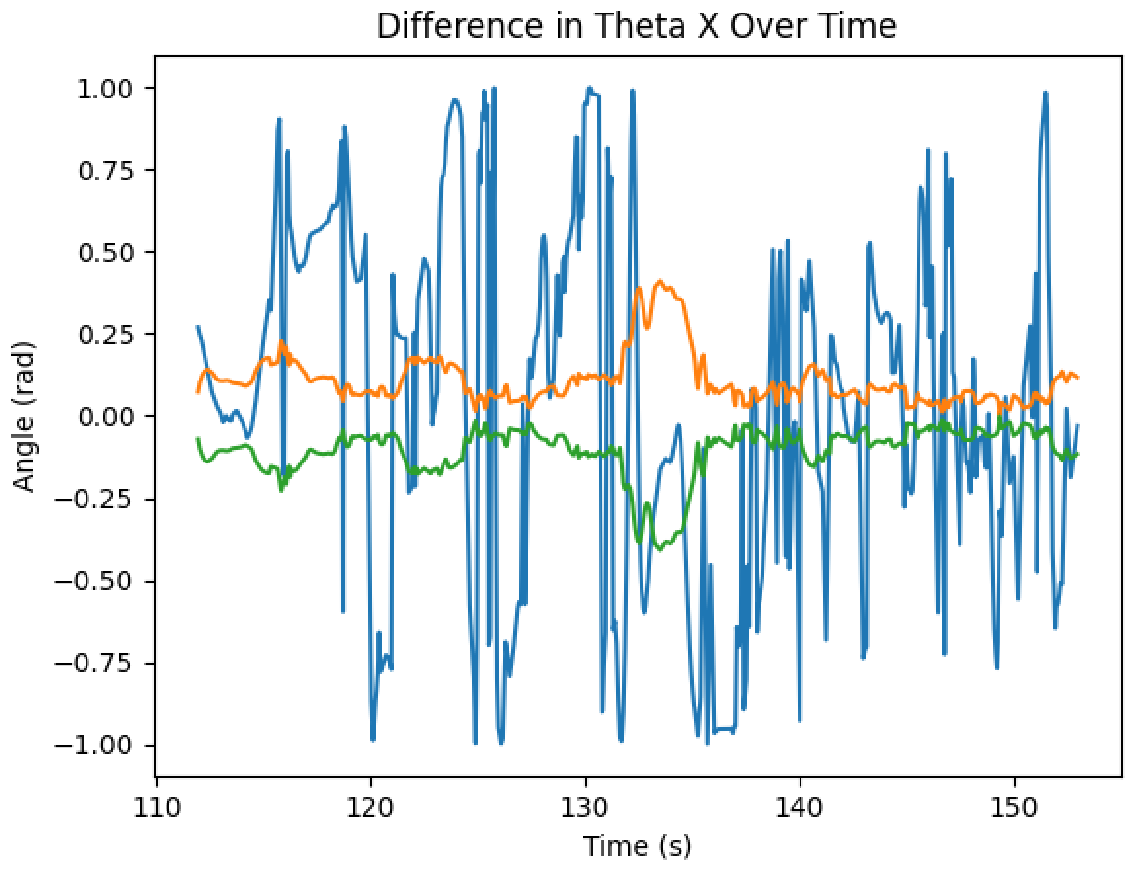
Figure 1: Number of Visible Landmarks at Each Time Step, Mk vs tk

3 Results for Batch from k1=1215 to k2=1714

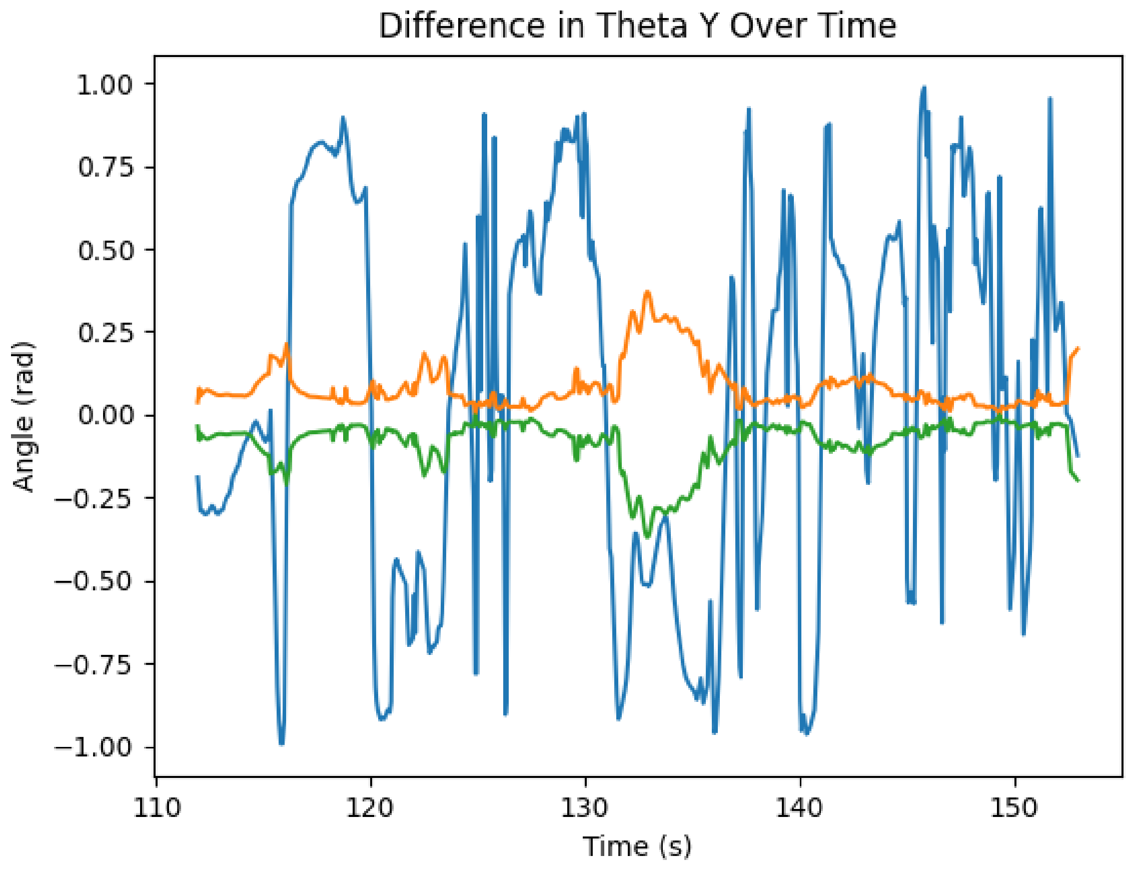


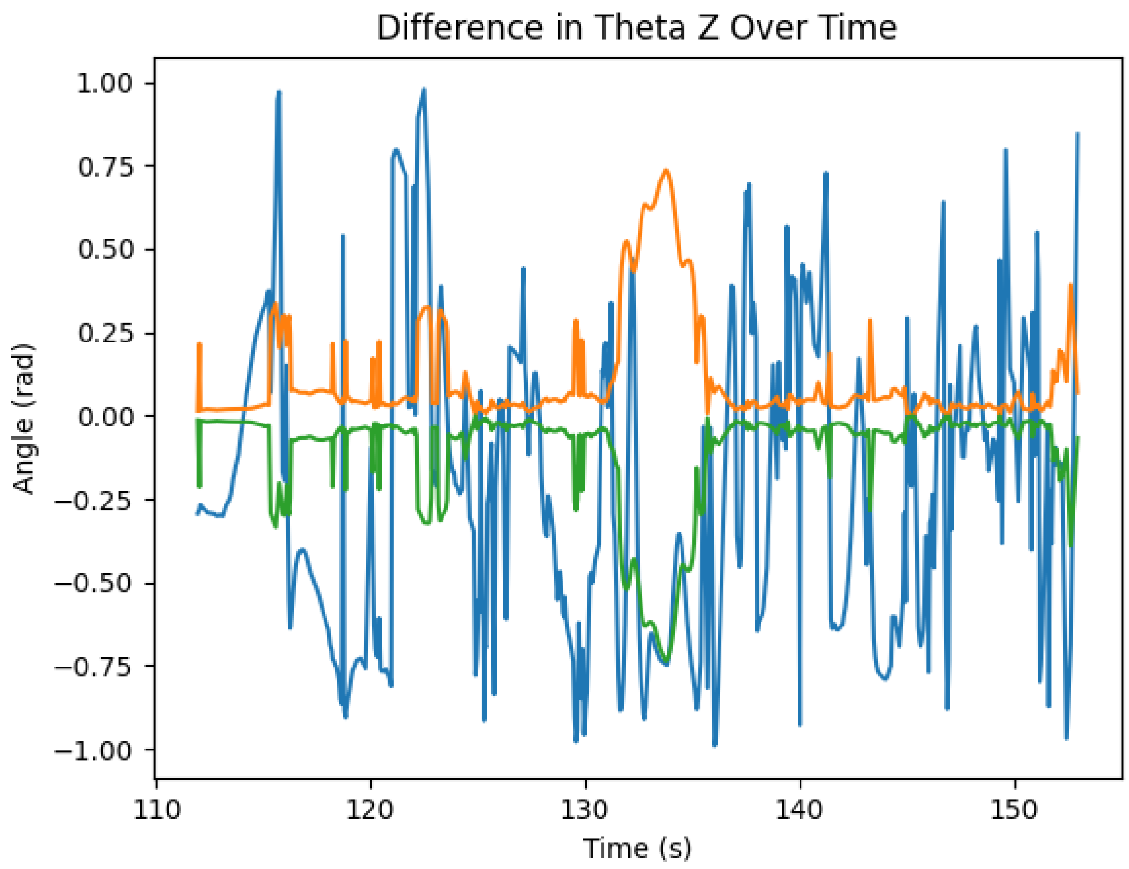

Difference in Y Position Over Time
Distance (m)
Time (s)
2
0
−2
−4
−6
110
120
130
140
150


Difference in Z Position Over Time
Distance (m)
Time (s)
4
3
2
1
0
−1
−2
−3
−4
110
120
130
140
150

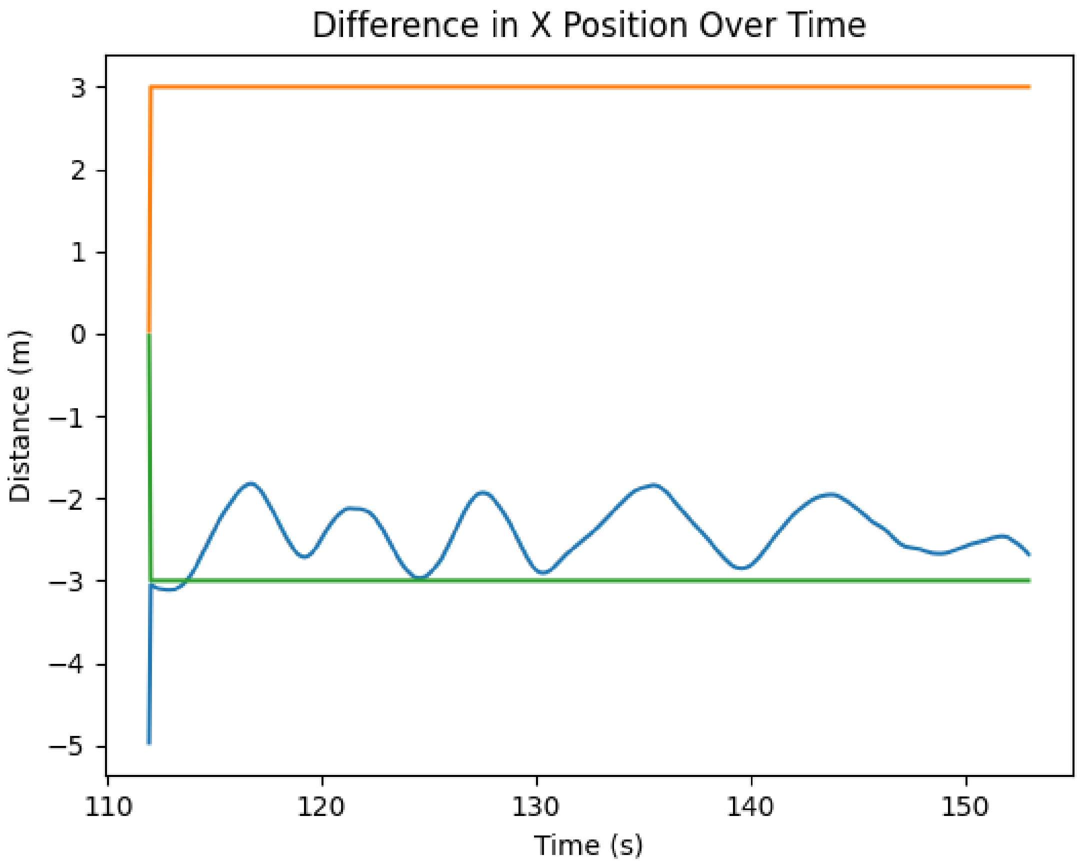


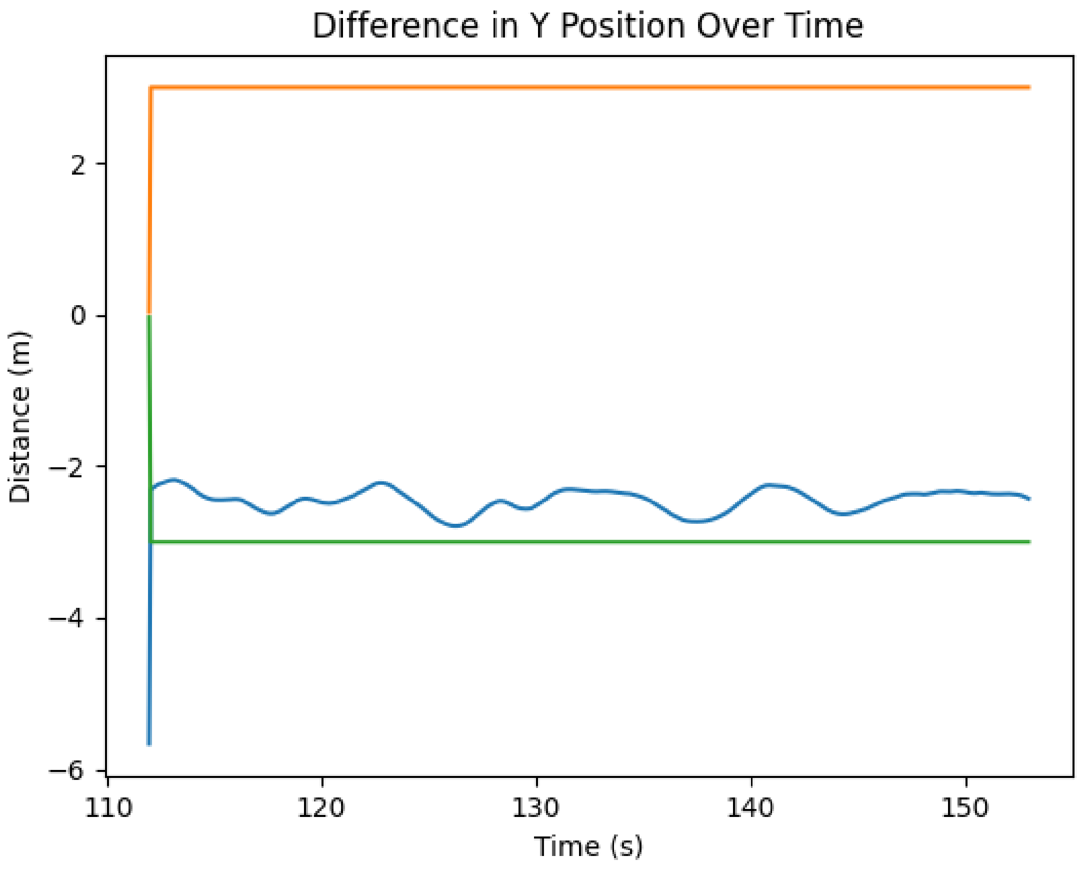
Difference in Theta X Over Time
Angle (rad)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
110
120
130
140
150
Time (s)

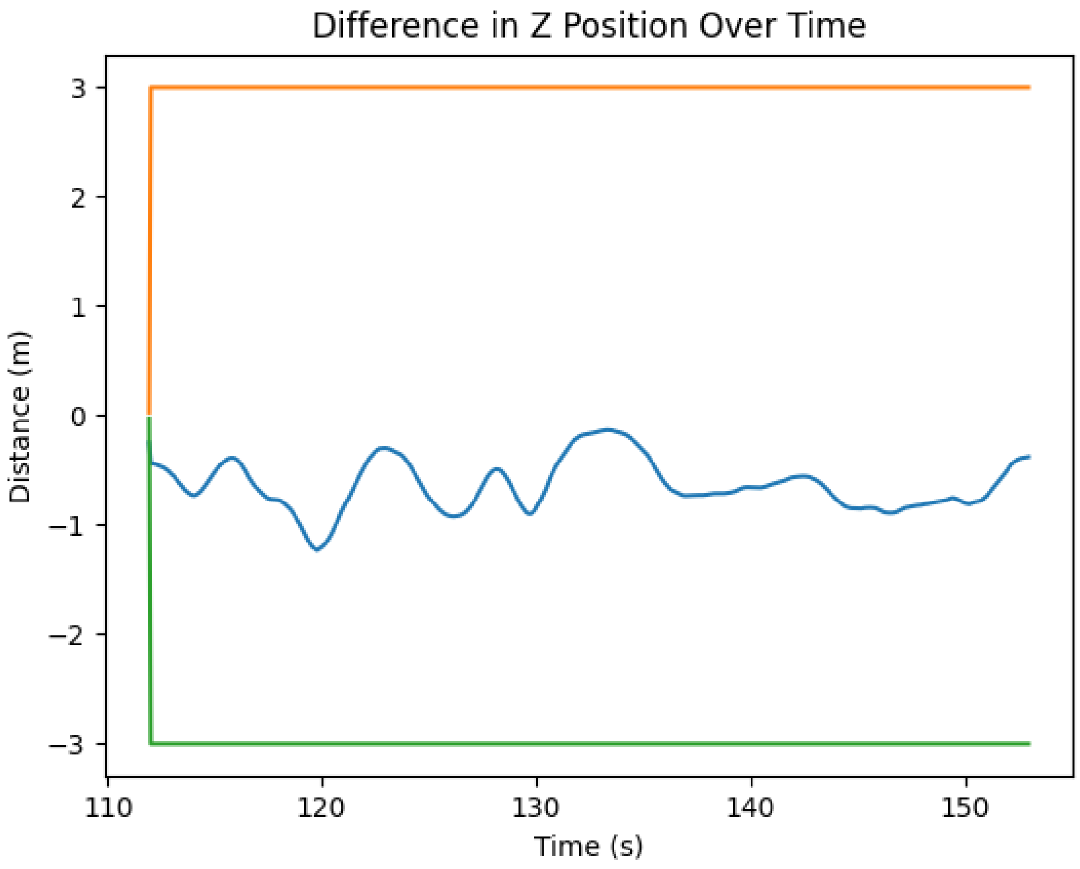

Difference in Theta Y Over Time
Angle (rad)
Time (s)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
110
120
130
140
150


Difference in Theta Z Over Time
Angle (rad)
Time (s)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
110
120
130
140
150

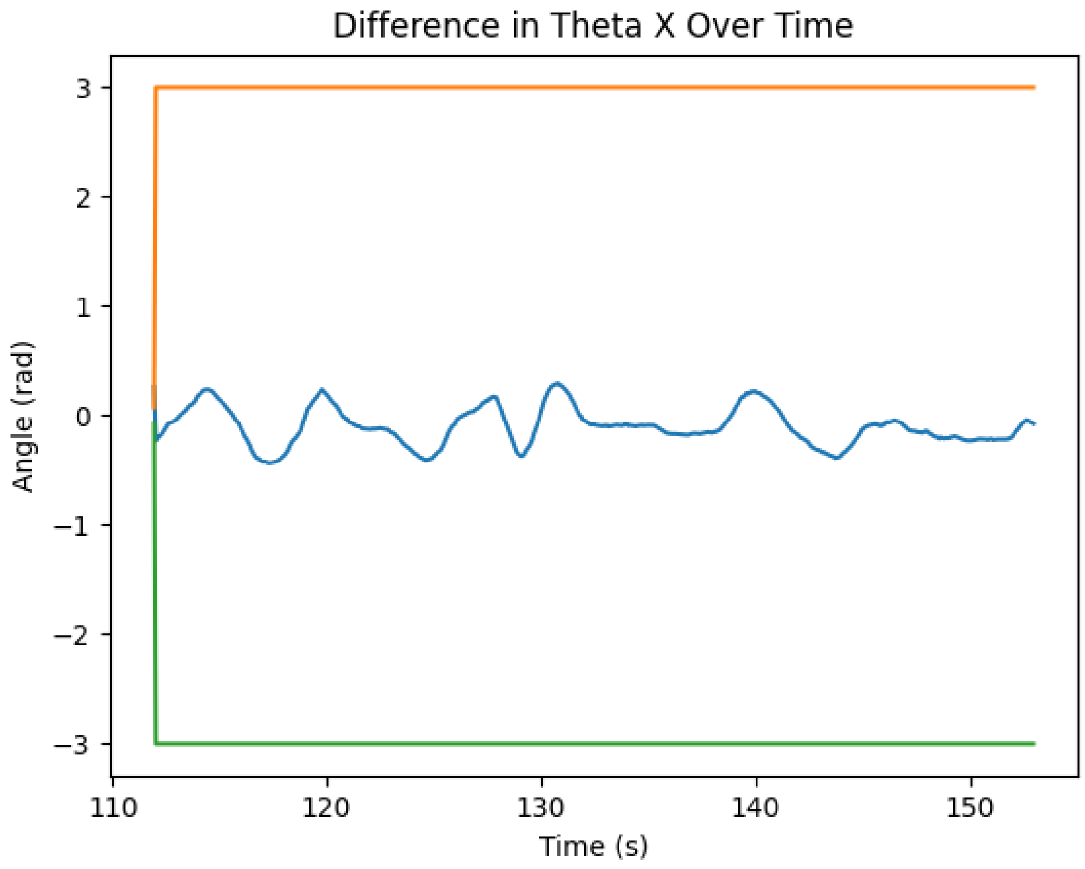
4 Results for Sliding Window with Kappa = 50 from k1=1215 to k2=1714

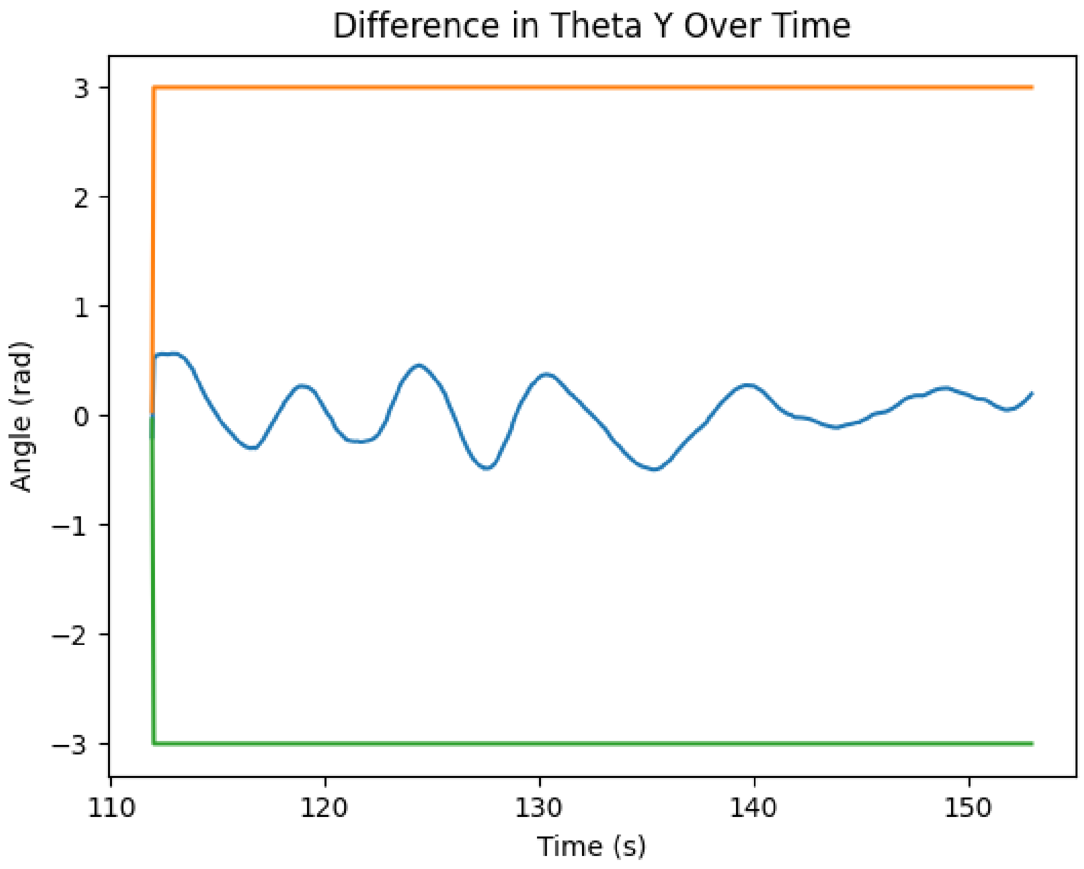


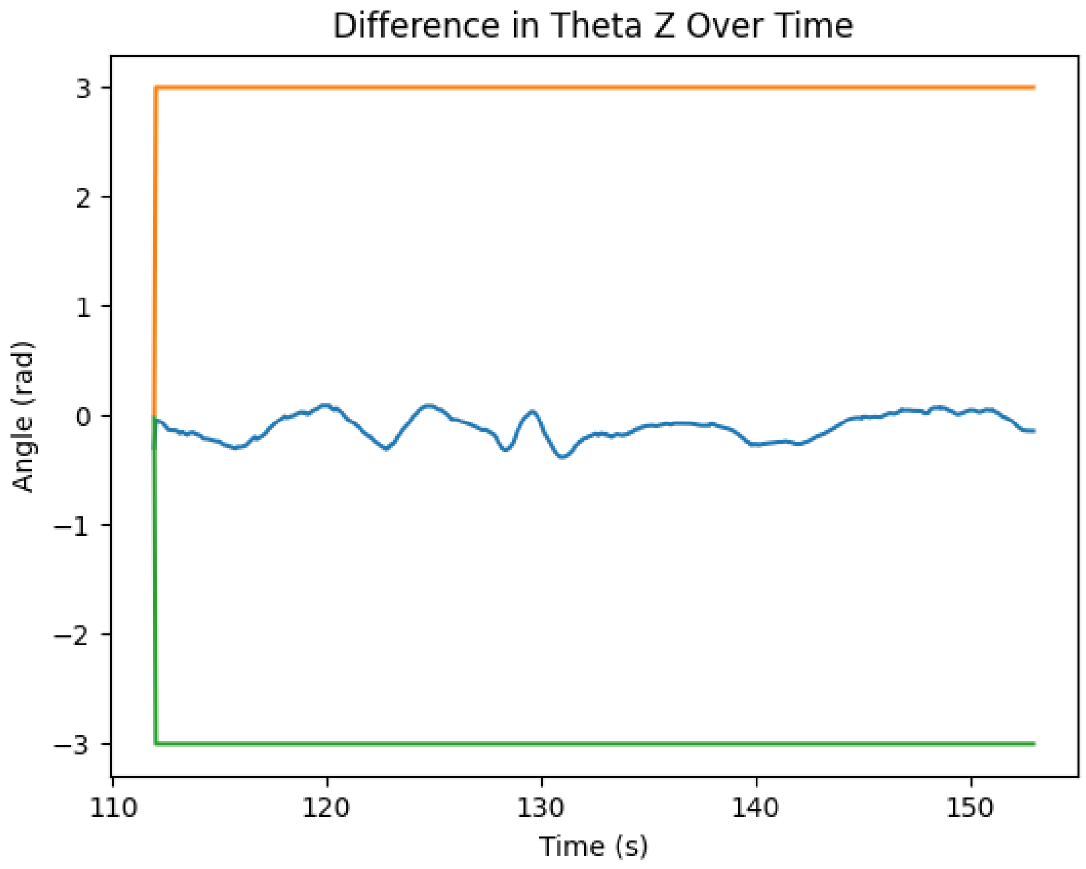

Difference in Y Position Over Time
Distance (m)
Time (s)
2
0
−2
−4
−6
110
120
130
140
150



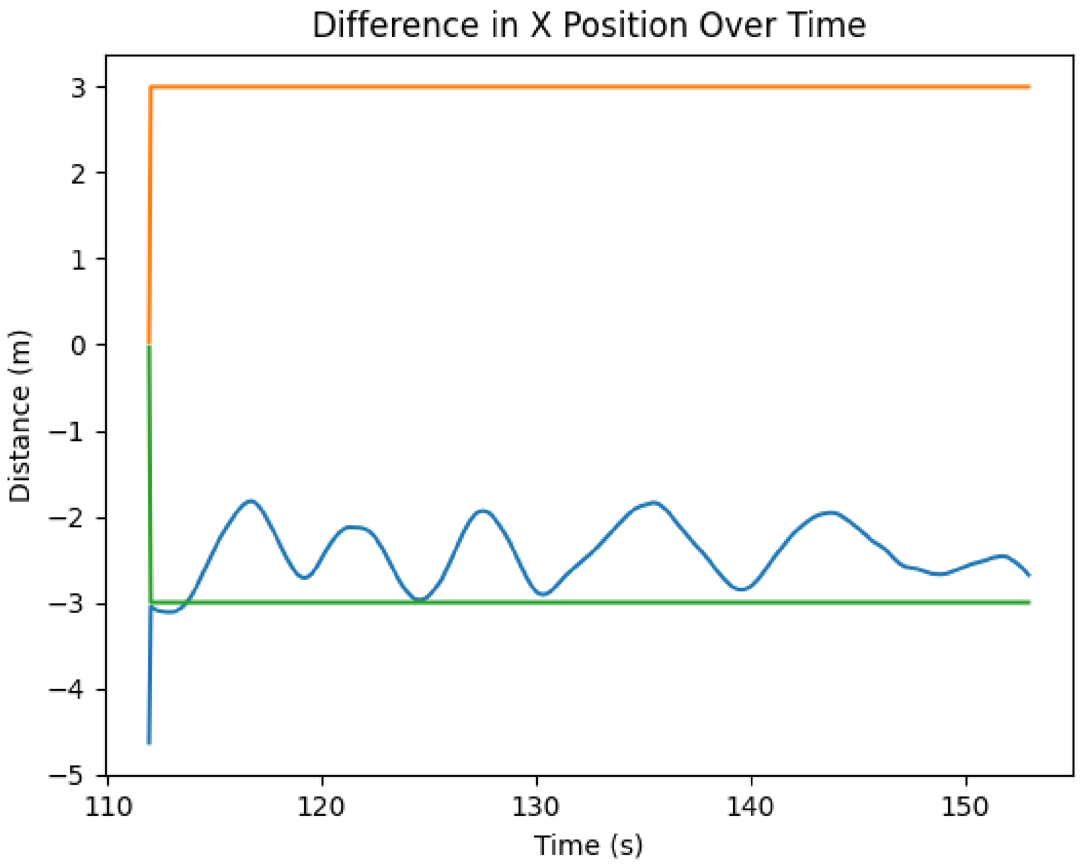
Difference in Z Position Over Time
Distance (m)
Time (s)
3
2
1
0
−1
−2
−3
110
120
130
140
150

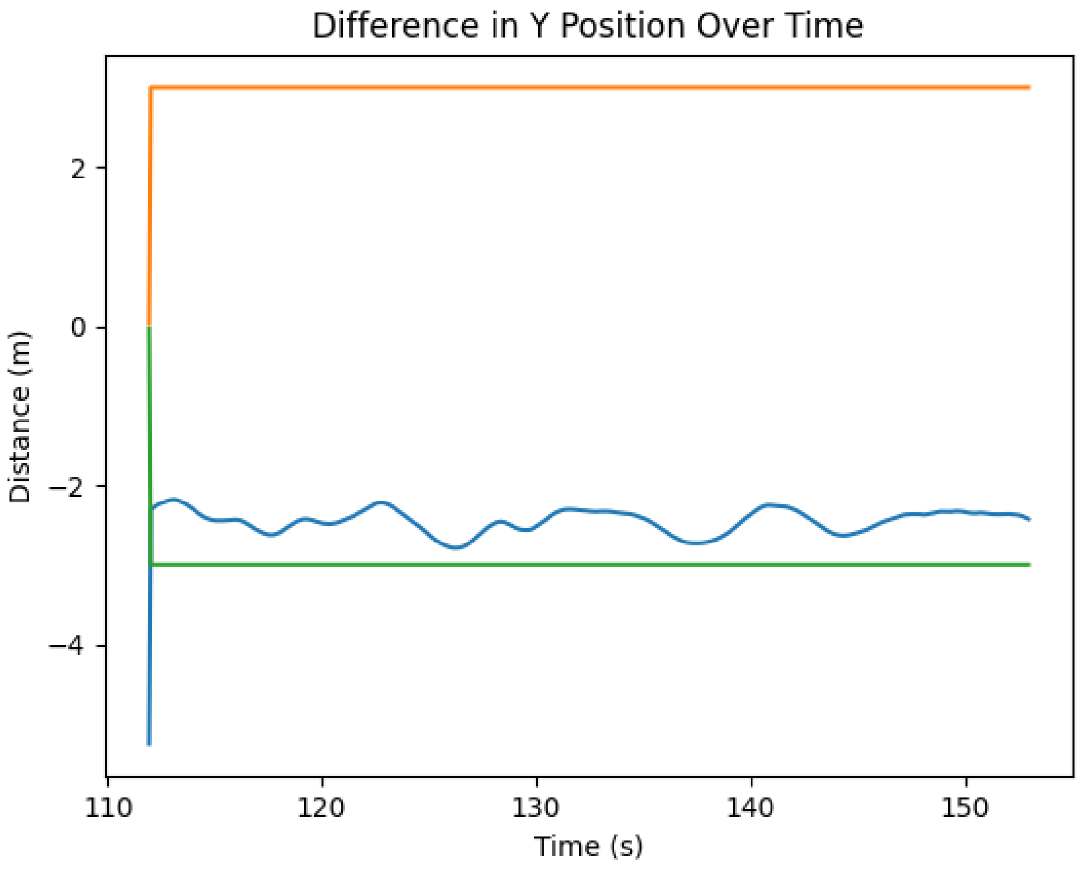

Difference in Theta X Over Time
Angle (rad)
Time (s)
3
2
1
0
−1
−2
−3
110
120
130
140
150

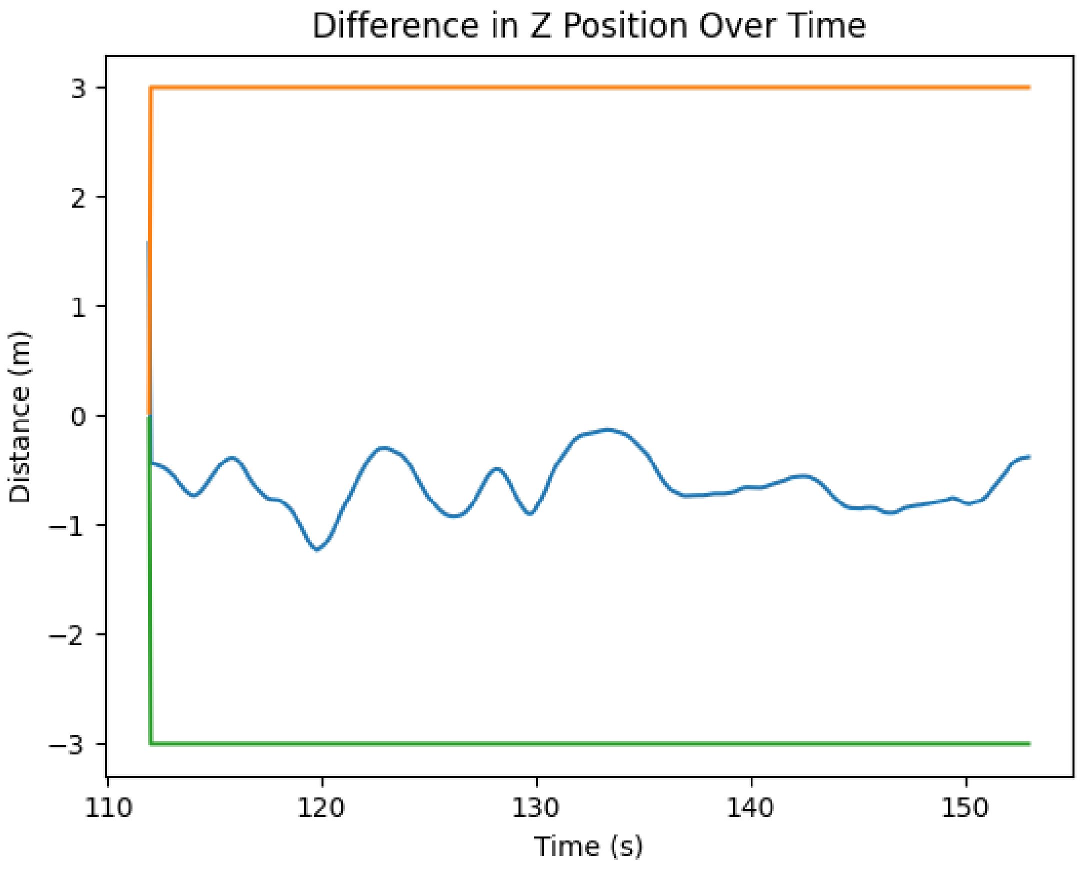

Difference in Theta Y Over Time
Angle (rad)
Time (s)
3
2
1
0
−1
−2
−3
110
120
130
140
150

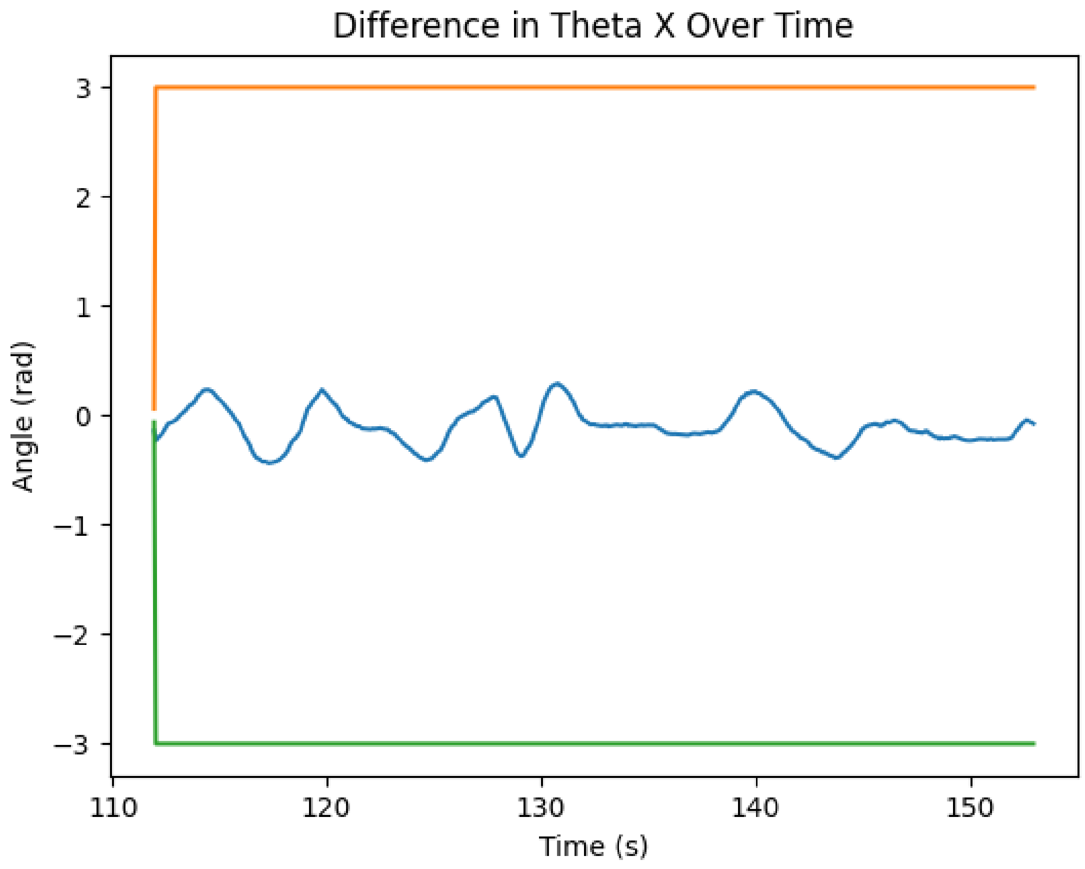

Difference in Theta Z Over Time
Angle (rad)
3
2
1
0
−1
−2
−3
110
120
130
140
150
Time (s)

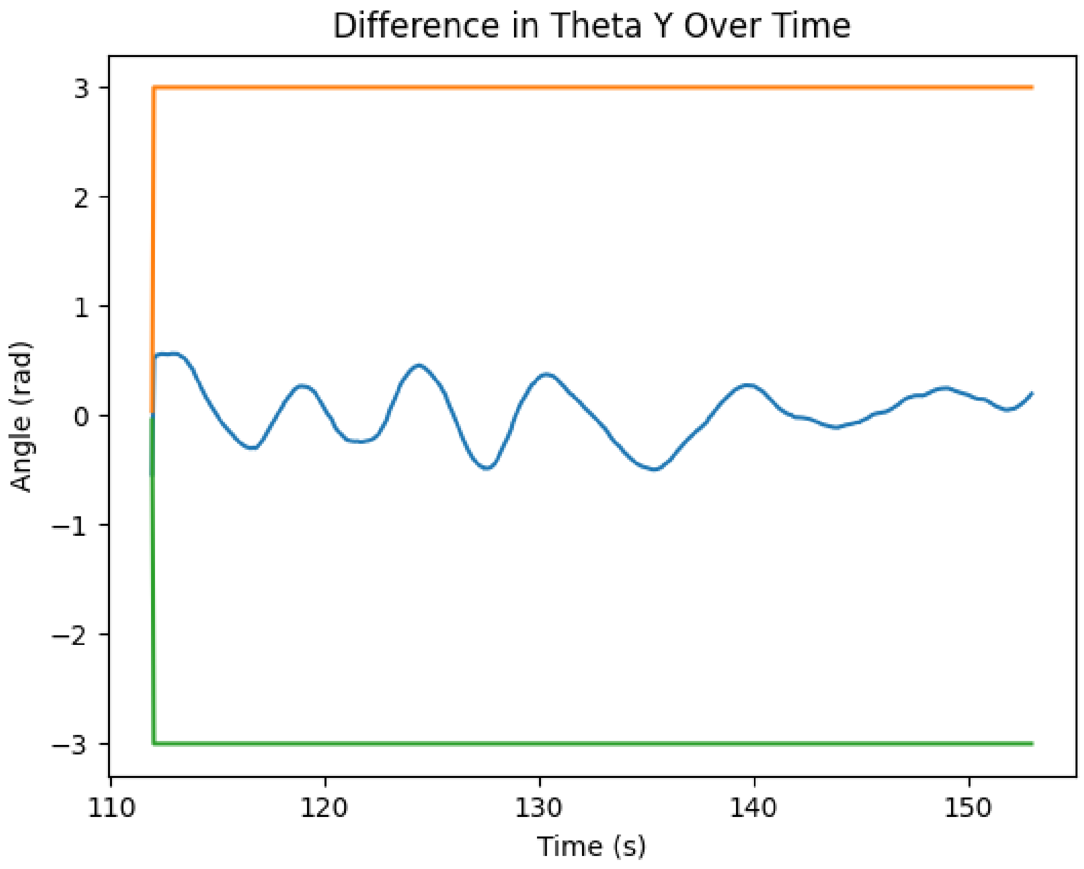
5 Results for Sliding Window with Kappa = 10 from k1=1215 to k2=1714

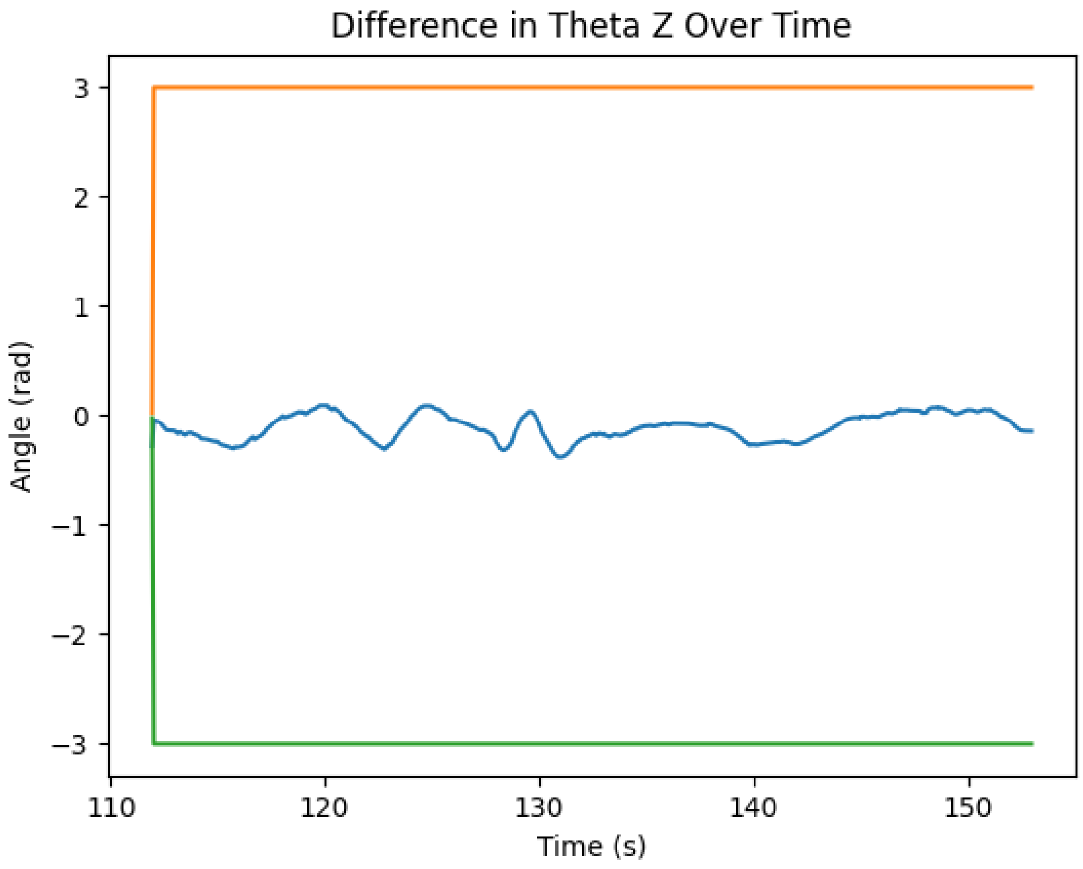



Difference in Y Position Over Time
Distance (m)
2
0
−2
−4
110
120
130
140
150
Time (s)


Difference in Z Position Over Time
Distance (m)
Time (s)
3
2
1
0
−1
−2
−3
110
120
130
140
150


Difference in Theta X Over Time
Angle (rad)
Time (s)
3
2
1
0
−1
−2
−3
110
120
130
140
150


Difference in Theta Y Over Time
Angle (rad)
Time (s)
3
2
1
0
−1
−2
−3
110
120
130
140
150



Sliding Window cases are more accurate and smoother than full batch, however both seem to be displaced from the ground truth by a set distance. It could be due to the stereo camera errors not being perfectly Gaussian and biased toward one side as mentioned in the Introduction or due to a bug in the code. Sliding window requires more computational effort as even though the individual batch iterations are shorter the number of batches causes the computation time to be higher. Some trends in the error plots and the Number of Visible Landmarks Over Time plot is that a lot of the areas where there are less than three visible landmarks (around 110s and 140s), the error is larger and the plot is noisier especially for the Sliding Window plots.

6 Appendix

```python
import numpy as np
from numpy.linalg import inv
import scipy.io
from scipy.linalg import block_diag, cholesky, solve_triangular, lu_factor,
lu_solve
from scipy.spatial.transform import Rotation
from tqdm import tqdm
import matplotlib.pyplot as plt
import pylgmath.se3.operations as lg
import pylgmath.so3.operations as l
import sys

# Load MATLAB file
mat_data = scipy.io.loadmat('dataset3.mat')

# Access data in the dictionary
# The keys in the dictionary correspond to variable names in the MATLAB file
theta_vk_i = mat_data['theta_vk_i']
r_i_vk_i = mat_data['r_i_vk_i']
t = mat_data['t']
w_vk_vk_i = mat_data['w_vk_vk_i']
w_var = mat_data['w_var']
v_vk_vk_i = mat_data['v_vk_vk_i']
v_var = mat_data['v_var']
rho_i_pj_i = mat_data['rho_i_pj_i']
y_k_j = mat_data['y_k_j']
y_var = mat_data['y_var']
C_c_v = mat_data['C_c_v']
rho_v_c_v = mat_data['rho_v_c_v']
fu = mat_data['fu'].item()
fv = mat_data['fv'].item()
cu = mat_data['cu'].item()
cv = mat_data['cv'].item()
b = mat_data['b'].item()

# Question 4
M_k = np.sum((y_k_j[0,:,:] != -1), axis=1)
t_k = t[0,:]
gt3_mask = M_k>=3
lt3_mask = M_k<3
M_k_3 = M_k[gt3_mask]
M_k_lt3 = M_k[lt3_mask]
t_k_3 = t_k[gt3_mask]
t_k_lt3 = t_k[lt3_mask]
plt.figure()
plt.scatter(t_k_lt3, M_k_lt3, color='red')
plt.scatter(t_k_3, M_k_3, color='green')
plt.title("Number of Visible Landmarks Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Number of Visible Landmarks")
plt.show()

# Question 5a
# Batch
def batch(k1,k2,T_check_k1):

    if np.all(np.eye(4) == T_check_k1):
        rot_init = theta_vk_i[:,k1]
        x_init = r_i_vk_i[:,k1]
        C_init = Rotation.from_rotvec(rot_init)

        T_check_k1_ = np.eye(4)
        T_check_k1_[:3, :3] = C_init.as_matrix()
        T_check_k1_[:3, 3] = x_init
    else:
        T_check_k1_ = T_check_k1.copy()

    D = np.array([[1,0,0],[0,1,0],[0,0,1],[0,0,0]])
    t_ = np.array([[0],[0],[0],[1]])
    xi_var = np.vstack((v_var,w_var))
    Qk = np.diag(xi_var.reshape(6,))
    Rjk_inv = inv(np.diag(y_var.reshape(4,)))
    Qks = np.repeat(Qk[np.newaxis, :], (k2-k1), axis = 0)

    T_op_init = np.eye(4)
    x_op = np.repeat(T_op_init[np.newaxis, :], (k2-k1+1), axis = 0)
    xi_k = np.vstack([v_vk_vk_i[:,k1+1:k2+1],w_vk_vk_i[:,k1+1:k2+1]])
    t_k1_k2 = t[0,k1:(k2+1)]
    delta_tk = t_k1_k2[1:] - t_k1_k2[:-1]

    Qks = np.transpose(np.transpose(Qks, axes=(1,2,0))*delta_tk, axes=(2,0,1))
    Qks_inv = inv(Qks)

    xi_k = xi_k.T[:,:,np.newaxis]
    delta_xi_k = np.transpose(np.transpose(xi_k, axes=(1,2,0))*delta_tk,
axes=(2,0,1))
    Xi_k = lg.vec2tran(delta_xi_k)

    for iteration in tqdm(range(10)):
        e_v0_xop = lg.tran2vec(T_check_k1_@inv(x_op[0,:,:]))
        E_0 = lg.vec2jacinv(-e_v0_xop)
        x_op_k = x_op[1:,:,:]
        x_op_k_1 = x_op[:-1,:,:]
        e_v_xop = lg.tran2vec(Xi_k@x_op_k_1@inv(x_op_k))
        E_k = lg.vec2jacinv(-e_v_xop)
        F_k_1 = -E_k@lg.tranAd(x_op_k@inv(x_op_k_1))
        y_mask = (y_k_j[0,k1:(k2+1),:] != -1)
        G = []
        R_inv = []
        e_y_xop = []
        for k in range(k2-k1+1):
            T_op = x_op[k,:,:]
            G_k = []
            e_y_k_xop = []
            for j in range(rho_i_pj_i.shape[1]):
                if y_mask[k,j] == True:
                    rho_i = rho_i_pj_i[:,j].reshape(3,1)
                    term_1 = -(D.T)@T_op@D@(D.T)@circledot(T_op@t_)
                    term_2 = (D.T)@circledot(T_op@D@rho_i)
                    term_3 = -(D.T)@circledot(T_op@D@(D.T)@T_op@t_)
                    P_ck = C_c_v@(term_1+term_2+term_3)

                    def f(T_):
                        p_T = C_c_v@(((D.T)@T_@D@(rho_i-((D.T)@T_@t_))) -
rho_v_c_v)
                        g_p_T = g(p_T)
                        return g_p_T

                    # Used numerical approximation of Jacobian since it seems to
work better
                    def numerical_jacobian(func, T_num, epsilon):

                        jac = np.zeros((4, 6))
                        epsilon_zero = np.zeros((6,1))

                        for col in range(6):
                            T_perturbed = T_num.copy()
                            epsilon_preturbed = epsilon_zero.copy()
                            epsilon_i = epsilon[col,0]
                            epsilon_preturbed[col,0] = epsilon_i
                            T_perturbed = lg.vec2tran(epsilon_preturbed)@T_num

                            f_perturbed_val = func(T_perturbed)
                            f_val = func(T_num)
                            jac[:, col] = ((f_perturbed_val - f_val) /
epsilon_i).reshape(4,)
                        return jac

                    if iteration == 0:
                        epsilon_it_0 = 1e-6*np.ones((6,1))
                        G_jk_numerical = numerical_jacobian(f, T_op, epsilon_it_0)
                    else:
                        G_jk_numerical = numerical_jacobian(f, T_op, epsilon_star)

                    p_Top = C_c_v@(((D.T)@T_op@D@(rho_i-((D.T)@T_op@t_))) -
rho_v_c_v)
                    g_p = g(p_Top)
                    S_p = S(p_Top)
                    G_jk = S_p@P_ck
                G_k.append(G_jk_numerical)
                e_y_jk_xop = y_k_j[:,k,j].reshape(4,1) - g_p
                e_y_k_xop.append(e_y_jk_xop)
                R_inv.append(Rjk_inv)

            if len(G_k) != 0:
                G_k = np.vstack(G_k)
            else:
                G_k = np.empty((0,6))

            G.append(G_k)

            if len(e_y_k_xop) != 0:
                e_y_k_xop = np.vstack(e_y_k_xop)
                e_y_xop.append(e_y_k_xop)

        if len(e_y_xop) != 0:
            e_y_xop = np.vstack(e_y_xop)

        E_k_list = [E_0] + E_k.tolist()
        E_matrix = block_diag(*E_k_list)
        F_matrix = block_diag(*(F_k_1.tolist()))
        F_matrix = np.block([[np.zeros((6,F_matrix.shape[0])), np.zeros((6,6))],
[F_matrix, np.zeros((F_matrix.shape[0],6))]])
        G_matrix = block_diag(*G)
        H_top = E_matrix + F_matrix
        H = np.vstack((H_top,G_matrix))

        e_v_xop_list = [e_v0_xop] + e_v_xop.tolist()
        e_v_xop = np.vstack(e_v_xop_list)
        if len(e_y_xop) != 0:
            e_xop = np.vstack((e_v_xop,e_y_xop))
        else:
            e_xop = e_v_xop

        P_check_k1_inv = inv((t[0,k1] - t[0,(k1-1)])*Qk)
        Qk_inv_list = [P_check_k1_inv] + Qks_inv.tolist()
        W_inv_list = Qk_inv_list + R_inv
        W_inv = block_diag(*W_inv_list)

        A = H.T@W_inv@H
        b_vec = H.T@W_inv@e_xop

        L = cholesky(A,lower=True)
        d_vec = solve_triangular(L,b_vec,lower=True)
        delta_x_star = solve_triangular(L.T,d_vec,lower=False)
        epsilon_star = delta_x_star.reshape((-1,6,1))
        x_op = lg.vec2tran(epsilon_star)@x_op

    lu, piv = lu_factor(A)
    A_inv = lu_solve((lu, piv), np.eye(A.shape[0]))
    cov = np.diag(A_inv)

    return x_op, cov

def circledot(v):
    D = np.array([[1,0,0],[0,1,0],[0,0,1],[0,0,0]])
    neg_rho_hat = -l.hat((D.T)@v)
    I = np.ones_like(neg_rho_hat)
    Z = np.zeros((1,6))
    inter = np.concatenate((I,neg_rho_hat), axis=1)
    result = np.concatenate((inter,Z), axis=0)
    return result

def S(p):
    return np.array([[fu/p[2,0], 0, -fu*p[0,0]/(p[2,0]**2)],[0, fv/p[2,0],
-fv*p[1,0]/(p[2,0]**2)],[fu/p[2,0],0,(-fu*p[0,0] + fu*b)/(p[2,0]**2)],[0,
fv/p[2,0], -fv*p[1,0]/(p[2,0]**2)]])

def g(p):
    return np.array([[fu*p[0,0]/p[2,0] + cu],[fv*p[1,0]/p[2,0] +
cv],[fu*(p[0,0]-b)/p[2,0] + cu],[fv*p[1,0]/p[2,0] + cv]])

def plot_graphs(k1_val,k2_val,traj_pred,cov,method,kappa):
    if method == "Sliding_Window":
        tag = f"{method}_kappa_{kappa}"
    else:
        tag = f"{method}"

    x_diff = traj_pred[:,0,3] - r_i_vk_i[0,k1_val:(k2_val+1)]
    y_diff = traj_pred[:,1,3] - r_i_vk_i[1,k1_val:(k2_val+1)]
    z_diff = traj_pred[:,2,3] - r_i_vk_i[2,k1_val:(k2_val+1)]

    rot_vec = theta_vk_i[:,k1_val:(k2_val+1)]
    C_true = Rotation.from_rotvec(rot_vec.T)

    identity = np.eye(3)
    identities = np.repeat(identity[np.newaxis, :], (k2_val-k1_val+1), axis = 0)
    theta_diff_crosses = identities -
traj_pred[:,:3,:3]@np.transpose((C_true.as_matrix()),(0,2,1))
    theta_x_diff = theta_diff_crosses[:,1,2]
    theta_y_diff = theta_diff_crosses[:,2,0]
    theta_z_diff = theta_diff_crosses[:,0,1]

    stds = np.sqrt(cov)
    epsilon_stds = stds.reshape(-1,6,1)
    T_stds = lg.vec2tran(epsilon_stds)
    x_stds = T_stds[:,0,3]
    y_stds = T_stds[:,1,3]
    z_stds = T_stds[:,2,3]
```

```
C_stds = T_stds[:,:3,:3]
r_vec_stds = Rotation.from_matrix(C_stds).as_rotvec()
theta_x_stds = r_vec_stds[:,0]
theta_y_stds = r_vec_stds[:,1]
theta_z_stds = r_vec_stds[:,2]

t_k1_k2 = t[0,k1_val:(k2_val+1)].T

plt.figure()
plt.plot(t_k1_k2,x_diff,label = "delta r_x")
plt.plot(t_k1_k2,3*x_stds,label = "+3 sigma r_x")
plt.plot(t_k1_k2,-3*x_stds,label = "-3 sigma r_x")
plt.title("Difference in X Position Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Distance (m)")
file_name = f"k1_{k1_val}_k2_{k2_val}_x_{tag}"
plt.savefig(file_name)

plt.figure()
plt.plot(t_k1_k2,y_diff,label = "delta r_y")
plt.plot(t_k1_k2,3*y_stds,label = "+3 sigma r_y")
plt.plot(t_k1_k2,-3*y_stds,label = "-3 sigma r_y")
plt.title("Difference in Y Position Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Distance (m)")
file_name = f"k1_{k1_val}_k2_{k2_val}_y_{tag}"
plt.savefig(file_name)

plt.figure()
plt.plot(t_k1_k2,z_diff,label = "delta r_z")
plt.plot(t_k1_k2,3*z_stds,label = "+3 sigma r_z")
plt.plot(t_k1_k2,-3*z_stds,label = "-3 sigma r_z")
plt.title("Difference in Z Position Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Distance (m)")
file_name = f"k1_{k1_val}_k2_{k2_val}_z_{tag}"
plt.savefig(file_name)

plt.figure()
plt.plot(t_k1_k2,theta_x_diff,label = "delta theta_x")
plt.plot(t_k1_k2,3*theta_x_stds,label = "+3 sigma theta_x")
plt.plot(t_k1_k2,-3*theta_x_stds,label = "-3 sigma theta_x")
plt.title("Difference in Theta X Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Angle (rad)")
file_name = f"k1_{k1_val}_k2_{k2_val}_theta_x_{tag}"
plt.savefig(file_name)

plt.figure()
```

```
    plt.plot(t_k1_k2,theta_y_diff,label = "delta theta_y")
    plt.plot(t_k1_k2,3*theta_y_stds,label = "+3 sigma theta_y")
    plt.plot(t_k1_k2,-3*theta_y_stds,label = "-3 sigma theta_y")
    plt.title("Difference in Theta Y Over Time")
    plt.xlabel("Time (s)")
    plt.ylabel("Angle (rad)")
    file_name = f"k1_{k1_val}_k2_{k2_val}_theta_y_{tag}"
    plt.savefig(file_name)

    plt.figure()
    plt.plot(t_k1_k2,theta_z_diff,label = "delta theta_z")
    plt.plot(t_k1_k2,3*theta_z_stds,label = "+3 sigma theta_z")
    plt.plot(t_k1_k2,-3*theta_z_stds,label = "-3 sigma theta_z")
    plt.title("Difference in Theta Z Over Time")
    plt.xlabel("Time (s)")
    plt.ylabel("Angle (rad)")
    file_name = f"k1_{k1_val}_k2_{k2_val}_theta_z_{tag}"
    plt.savefig(file_name)

    return

#Question 5a Batch
k1_val = 1215
k2_val = 1714
traj_pred, cov = batch(k1_val,k2_val,np.eye(4))
plot_graphs(k1_val,k2_val,traj_pred,cov,"Batch",None)

#Question 5b Sliding Window

def sliding_window(k1_val, k2_val, kappa):
    I = np.eye(4)
    traj_final = np.repeat(I[np.newaxis, :], (k2_val-k1_val+1), axis = 0)
    cov_final = np.ones((6*(k2_val-k1_val+1),))
    for i in range(k2_val-k1_val+1):
        k1_slide = k1_val + i
        k2_slide = k1_slide + kappa
        if k1_slide == k1_val:
            traj_pred, cov = batch(k1_slide,k2_slide,np.eye(4))
            traj_i = traj_pred[i,:,:]
            traj_final[i,:,:] = traj_i
            cov_final[i:i+6] = cov[:6]
        else:
            batch(k1_slide,k2_slide,traj_i)
    plot_graphs(k1_val,k2_val,traj_final,cov_final,"Sliding_Window",kappa)
    return

k1_val = 1215
k2_val = 1714
kappa = 50
```

```
sliding_window(k1_val, k2_val, kappa)

#Question 5c Sliding Window

k1_val = 1215
k2_val = 1714
kappa = 10

sliding_window(k1_val, k2_val, kappa)
```