AER1513 State Estimation

Assignment 3 Report

"Starry Night Dataset"

Qilong (Jerry) Cheng 1003834103

October 29, 2025

Question 1

Based on the histogram distribution graphs, the assumption made that noise is zero-mean Gaussian distribution is not true for all noises but holds true in most input variables.

First, observing the process noise from the IMU, the translation and rotational noise at all three directions do assemble a Gaussian zero-mean distribution. And the estimated covariance might even be less confident than it should be, since the peak of both translation and rotational noise all exceed the peak of the Gaussian distribution.

However, for the stereo camera model, This assumption does not hold true: both u_l and u_r can be assumed under Gaussian noise with zero-mean since they fit under the Guassian curve very nicely. But both v_l and v_r are tail heavy and have a clear bias to the lower axis with an offset of about -7 pixels. Nonetheless, for our problme and application, zero-mean Gaussian assumption could still be reasonable as the offset is not too bad based on the histogram. During the experiment, considering the given covariance may not be the most accuracte. we could inflate the given numeber by 1.2 times or 1.5 times and compare the results to get an ideal estimated covariance.

$$\mathbf{Q}_k = \begin{bmatrix} \sigma_{v_x}^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_{v_y}^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_{v_z}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{w_x}^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_{w_y}^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_{w_z}^2 \end{bmatrix} T_K^2 = \begin{bmatrix} 0.0026 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.0021 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.00079 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.0090 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.017 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.17 \end{bmatrix} T_K^2 \tag{1}$$

$$\mathbf{R}_k^j = \begin{bmatrix} \sigma_{u_l}^2 & 0 & 0 & 0 \\ 0 & \sigma_{v_l}^2 & 0 & 0 \\ 0 & 0 & \sigma_{u_r}^2 & 0 \\ 0 & 0 & 0 & \sigma_{u_l}^2 \end{bmatrix} = \begin{bmatrix} 37.98 & 0 & 0 & 0 \\ 0 & 129.84 & 0 & 0 \\ 0 & 0 & 41.95 & 0 \\ 0 & 0 & 0 & 132.49 \end{bmatrix} \tag{2}$$

In addition to the above covariance, in the actual implementation, the motion model's initialization set the initial covariance to be all zeros:

$$\mathbf{P}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3}$$

Question 2

We first combine the translation and rotational matrix into a single pose matrix as the state. The modified state variable at each time-step after stacking the two matrix would look like:

$$\mathbf{T}_k = \mathbf{T}_{v_k,i} = \begin{bmatrix} \boldsymbol{C}_{v_k,i} & \boldsymbol{C}_{v_k,i}\boldsymbol{t}_i^{v_k,i} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4}$$

Given time-step is from k_1 to k_2, the total time passed will be: $(k_2 - k_1)$

Thus, the total state vector we try to estimate would become:

$$\boldsymbol{x}_{k_1:k_2} = \begin{bmatrix} \boldsymbol{T}_{v_{k_1},i} \\ \vdots \\ \boldsymbol{T}_{v_{k_2},i} \end{bmatrix} \tag{5}$$

The input state can be expressed as:

$$\boldsymbol{\varpi} = \begin{bmatrix} \boldsymbol{\nu}_{v_k}^{iv_k} \\ \boldsymbol{\omega}_{v_k}^{iv_k} \end{bmatrix} \tag{6}$$

Thus, the total input from the given time-steps k_1 to k_2 can be expressed as:

$$\boldsymbol{v} = \begin{bmatrix} \check{\boldsymbol{T}}_{k_1} \\ \boldsymbol{\varpi}_{k_1+1} \\ \vdots \\ \boldsymbol{\varpi}_{k_2} \end{bmatrix} \tag{7}$$

Given that M_k is the total observed landmarks at the time-step k, the total measurement vector can be written as:

$$\mathbf{y} = \begin{bmatrix} \mathbf{y}_{k_1}^{1} \\ \vdots \\ \mathbf{y}_{k_1}^{M_{k_1}} \\ \vdots \\ \mathbf{y}_{k_2}^{1} \\ \vdots \\ \mathbf{y}_{k_2}^{M_{k_2}} \end{bmatrix} \tag{8}$$

Thus, the motion model can be rewritten as:

$$\mathbf{T}_k = \mathbf{\Xi}_k \mathbf{T}_{k-1} = \exp(\Delta t_k \boldsymbol{\varpi}_k)\mathbf{T}_{k-1} \tag{9}$$

And the observation model can be written as:

$$\mathbf{y}_k^j = \frac{1}{z_{jk}} \mathbf{M} \mathbf{T}_{cv} \mathbf{T}_k \mathbf{p}_i^{p_j,i} \tag{10}$$

where $\mathbf{T}_{cv}$ is the transform between the IMU and the camera, $\mathbf{M}$ is the stereo camera intrinsic matrix.

$$\mathbf{M} = \begin{bmatrix} f_u & 0 & c_u & 0 \\ 0 & f_v & c_v & 0 \\ f_u & 0 & c_u & -f_u b \\ 0 & f_v & c_v & 0 \end{bmatrix} \tag{11}$$

Now, the errors can be defined as:

- **Motion Model Error Term** is composed by two terms: one is the error of the initial guess, and the later pose estimation error terms

$$\mathbf{e}_{v,k}(\mathbf{x}) = \begin{cases} \ln(\check{\mathbf{T}}_{k_1} \mathbf{T}_{k_1}^{-1})^\vee & k = k_1 \\ \ln(\mathbf{\Xi}_k \mathbf{T}_{k-1} \mathbf{T}_k^{-1})^\vee & k = (k_1+1)\dots k_2 \end{cases} \tag{12}$$

where

$$\mathbf{\Xi}_k = \exp\left(\Delta t_k \varpi_k^\wedge\right) \tag{13}$$

- **Measurement Model Error Term**:

$$\mathbf{e}_{y,jk}(\mathbf{x}) = \mathbf{y}_k^j - \overline{\mathbf{g}}(\mathbf{p}_{ck}^{p_j,i}) \tag{14}$$
$$= \mathbf{y}_k^j - \overline{\mathbf{g}}(\mathbf{D}^T \mathbf{T}_{cv} \mathbf{T}_k \mathbf{p}_i^{p_j,i}) \tag{15}$$

where $\overline{\mathbf{g}}$ is the nominal observation model and $\mathbf{p}_{ck}^{p_j,ck}$ is the points that are projected into the rectified images of an axis-aligned stereo camera.

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{16}$$

$$\mathbf{T}_{cv} = \begin{bmatrix} \mathbf{C}_{cv} & -\mathbf{C}_{cv}\rho_v^{cv} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{17}$$

$$\mathbf{p}_i^{p_j,i} = \begin{bmatrix} \rho i^{p_j,i} \\ 1 \end{bmatrix} \tag{18}$$

Now, based on the Bayesian point of view, we will exam the noise properties of the errors: Given that the true pose variable can be drawn from the prior:

$$\mathbf{T}_k = \exp(\delta\boldsymbol{\xi}_k^\wedge)\check{\mathbf{T}} \tag{19}$$

where

$$\delta\boldsymbol{\xi}_k \sim \mathcal{N}(\mathbf{0}, \check{\mathbf{P}}_k) \tag{20}$$

The first input error can be expressed as:

$$\mathbf{e}_{v,k_1}(\mathbf{x}) = \ln(\check{\mathbf{T}}_{k_1}\mathbf{T}_{k_1}^{-1})^\vee \tag{21}$$
$$= \ln(\check{\mathbf{T}}_{k_1}\mathbf{T}_{k_1}^{-1}\exp(-\delta\boldsymbol{\xi}_0^\wedge)) \tag{22}$$
$$= -\delta\boldsymbol{\xi}_0 \tag{23}$$

so that:

$$\mathbf{e}_{v,k_1}(\mathbf{x}) \sim \mathcal{N}(\mathbf{0}, \check{\mathbf{P}}_{k_1}) \tag{24}$$

For the measurement model, we consider that

$$\mathbf{e}_{y,jk}(\mathbf{x}) = \mathbf{y}_k^j - \overline{\mathbf{g}}(\mathbf{p}_{ck}^{p_j,i}) \tag{25}$$
$$= \mathbf{y}_k^j - \overline{\mathbf{g}}(\mathbf{D}^T \mathbf{T}_{cv} \mathbf{T}_k \mathbf{p}_i^{p_j,i}) \tag{26}$$
$$= \mathbf{n}_{jk} \tag{27}$$

Thus,

$$\mathbf{e}_{y,jk}(\mathbf{x}) \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k^j) \tag{28}$$

Given the above error properties, we can conclude that:

$$J_{v,k}(\mathbf{x}) = \begin{cases} \frac{1}{2}\mathbf{e}_{v,k_1}(\mathbf{x})^T \check{\mathbf{P}}_{k_1}^{-1} \mathbf{e}_{v,k_1}(\mathbf{x}) & k = k_1 \\ \frac{1}{2}\mathbf{e}_{v,k}(\mathbf{x})^T \mathbf{Q}_k^{-1} \mathbf{e}_{v,k}(\mathbf{x}) & k = (k_1+1)\ldots k_2 \end{cases} \tag{29}$$

$$J_{y,k}(\mathbf{x}) = \frac{1}{2}\mathbf{e}_{y,k}(\mathbf{x})^T \mathbf{R}_k^{-1} \mathbf{e}_{y,k}(\mathbf{x}) \tag{30}$$

where we stack all the M points together in the measurement model:

$$\mathbf{e}_{y,k}(\mathbf{x}) = \begin{bmatrix} \mathbf{e}_{y,1k}(\mathbf{x}) \\ \mathbf{e}_{y,2k}(\mathbf{x}) \\ \vdots \\ \mathbf{e}_{y,Mk}(\mathbf{x}) \end{bmatrix} \tag{31}$$

and

$$\mathbf{R}_k = \text{diag}(\mathbf{R}_{1k}, \mathbf{R}_{2k}, \ldots, \mathbf{R}_{Mk}) \tag{32}$$

Finally, we stack all the motion model and measurement model cost function together into one single objective function:

$$J(\mathbf{x}_{k_1:k_2}) = \sum_{k=k_1}^{k_2} (J_{v,k}(\mathbf{x}) + J_{y,k}(\mathbf{x})) \tag{33}$$

$$= \frac{1}{2}(\mathbf{e}_{k_1:k_2})^T \mathbf{T}^{-1}(\mathbf{e}_{k_1:k_2}) \tag{34}$$

where

$$\mathbf{e}(\mathbf{x}_{k_1:k_2}) = \left[\begin{array}{c} \mathbf{e}_{v,k_1}(\mathbf{x}) \\ \mathbf{e}_{v,k_1+1}(\mathbf{x}) \\ \vdots \\ \mathbf{e}_{v,k_2}(\mathbf{x}) \\ \hline \mathbf{e}_{y,k_1}(\mathbf{x}) \\ \mathbf{e}_{y,k_1+1}(\mathbf{x}) \\ \vdots \\ \mathbf{e}_{y,k_2}(\mathbf{x}) \end{array}\right] \begin{array}{l} \left.\vphantom{\begin{array}{c} a \\ a \\ \vdots \\ a \end{array}}\right\} \text{Input Error} \\ \\ \left.\vphantom{\begin{array}{c} a \\ a \\ \vdots \\ a \end{array}}\right\} \text{Measurement Error} \end{array} \tag{35}$$

and the inverse of the covariance matrix is:

$$\mathbf{T}^{-1} = \mathbf{W}^{-1} = \text{diag}(\check{\mathbf{P}}_{k_1}^{-1}, \mathbf{Q}_{k_1}^{-1}, \mathbf{Q}_{k_1+1}^{-1}, \ldots, \mathbf{Q}_{k_2}^{-1}, \mathbf{R}_{k_1}^{-1}, \mathbf{R}_{k_1+1}^{-1}, \ldots, \mathbf{R}_{k_2}^{-1}) \tag{36}$$

Question 3

First, we will linearize the error terms derived from Question 2. Assume the initial trajectory guess is $\mathbf{T}_{op,k}$, the small perturbation is $\boldsymbol{\epsilon}_k^{\wedge}$, then we get:

$$\mathbf{T}_k = \exp(\boldsymbol{\epsilon}_k^{\wedge})\mathbf{T}_{op,k} \tag{37}$$

we will use shorthand here:

$$\mathbf{x}_{op} = \begin{bmatrix} \mathbf{T}_{op,k_1} \\ \mathbf{T}_{op,k_1+1} \\ \vdots \\ \mathbf{T}_{op,k_2} \end{bmatrix} \tag{38}$$

The **first input error** is:

$$\mathbf{e}_{v,k_1}(\mathbf{x}) = \ln(\check{\mathbf{T}}_{k_1}\mathbf{T}_{k_1}^{-1})^{\vee} = \ln(\check{\mathbf{T}}_{k_1}\mathbf{T}_{op,k_1}^{-1}\exp(-\boldsymbol{\epsilon}_{k_1}^{\wedge}))^{\vee} \approx \mathbf{e}_{v,k_1}(\mathbf{x}_{op}) - \boldsymbol{\epsilon}_{k_1} \tag{39}$$

where

$$\mathbf{e}_{v,k_1}(\mathbf{x}_{op}) = \ln(\mathbf{T}_{k_1}\mathbf{T}_{op,k_1}^{-1})^{\vee} \tag{40}$$

For the **later input errors**, we have:

$$
\begin{aligned}
\mathbf{e}_{v,k}(\mathbf{x}) &= \ln(\mathbf{\Xi}_k \mathbf{T}_{k-1}\mathbf{T}_k^{-1})^\vee && (41)\\
&= \ln(\mathbf{\Xi}_k \mathbf{T}_{op,k-1}\mathbf{T}_{op,k}^{-1}\exp(-\boldsymbol{\epsilon}_k^\wedge))^\vee && (42)\\
&= \ln(\underbrace{\mathbf{\Xi}_k \mathbf{T}_{op,k-1}\mathbf{T}_{op,k}^{-1}}_{\exp(\mathbf{e}_{v,k}(\mathbf{x}_{op})^\wedge)}\exp((Ad(\mathbf{T}_{op,k}\mathbf{T}_{op,k-1})^{-1})\boldsymbol{\epsilon}_{k-1})^\wedge \times \exp(-\boldsymbol{\epsilon}_k^\wedge))^\vee && (43)\\
&\approx \mathbf{e}_{v,k}(\mathbf{x}_{op}) + \underbrace{Ad(\mathbf{T}_{op,k}\mathbf{T}_{op,k-1}^{-1})}_{\mathbf{F}_{k-1}}\boldsymbol{\epsilon}_{k-1} - \boldsymbol{\epsilon}_k && (44)
\end{aligned}
$$

where

$$\mathbf{e}_{v,k}(\mathbf{x}_{op}) = \ln(\mathbf{\Xi}_k \mathbf{T}_{op,k-1}\mathbf{T}_{op,k-1}^{-1})^\vee \tag{45}$$

is the error evaluated a the operating point.

For the **measurement error**:

$$
\begin{aligned}
\mathbf{e}_{y,jk}(\mathbf{x}) &= \mathbf{y}_{jk} - \overline{\mathbf{g}}(\mathbf{D}^T\mathbf{T}_{cv}\mathbf{T}_k\mathbf{p}_i^{p_j,i}) && (46)\\
&= \mathbf{y}_{jk} - \overline{\mathbf{g}}(\mathbf{D}^T\mathbf{T}_{cv}\exp(\boldsymbol{\epsilon}_k^\wedge)\mathbf{T}_{op,k}\mathbf{p}_i^{p_j,i}) && (47)\\
&\approx \mathbf{y}_{jk} - \overline{\mathbf{g}}(\mathbf{D}^T\mathbf{T}_{cv}(\mathbf{1}+\boldsymbol{\epsilon}_k^\wedge)\mathbf{T}_{op,k}\mathbf{p}_i^{p_j,i}) && (48)\\
&= \mathbf{y}_{jk} - \overline{\mathbf{g}}(\mathbf{D}^T\mathbf{T}_{cv}\mathbf{T}_{op,k}\mathbf{p}_i^{p_j,i} + (\mathbf{D}^T\mathbf{T}_{cv}(\mathbf{T}_{op,k}\mathbf{p}_i^{p_j,i})^\odot)\boldsymbol{\epsilon}_k) && (49)\\
&= \underbrace{\mathbf{y}_{jk} - \overline{\mathbf{g}}(\mathbf{D}^T\mathbf{T}_{cv}\mathbf{T}_{op,k}\mathbf{p}_i^{p_j,i})}_{\mathbf{e}_{y,jk}(\mathbf{x}_{op})} - \underbrace{\left.\frac{\partial\overline{\mathbf{g}}}{\partial\mathbf{z}}\right|_{\mathbf{z}=\mathbf{D}^T\mathbf{T}_{cv}\mathbf{T}_{op,k}\mathbf{p}_i^{p_j,i}}(\mathbf{D}^T\mathbf{T}_{cv}(\mathbf{T}_{op,k}\mathbf{p}_i^{p_j,i})^\odot)}_{\mathbf{G}_{jk}}\boldsymbol{\epsilon}_k && (50)
\end{aligned}
$$

where the first derivative for the observation function is:

$$\frac{\partial\overline{\mathbf{g}}}{\partial\mathbf{z}} = \begin{bmatrix} \frac{f_u}{z} & 0 & \frac{f_u x}{z^2} \\ 0 & \frac{f_v}{z} & \frac{f_v y}{z^2} \\ \frac{f_u}{z} & 0 & \frac{f_u(x-b)}{z^2} \\ 0 & \frac{f_v}{z} & \frac{f_v y}{z^2} \end{bmatrix} \tag{51}$$

where

$$\mathbf{z} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{52}$$

We can stack all the measurement error at time k together:

$$\mathbf{e}_{y,k}(\mathbf{x}) \approx \mathbf{e}_{y,k}(\mathbf{x}_{op}) - \mathbf{G}_k\epsilon_k \tag{53}$$

where

$$\mathbf{e}_{y,k}(\mathbf{x}) = \begin{bmatrix} \mathbf{e}_{y,1}(\mathbf{x}) \\ \mathbf{e}_{y,2}(\mathbf{x}) \\ \vdots \\ \mathbf{e}_{y,M_k}(\mathbf{x}) \end{bmatrix} \tag{54}$$

$$\mathbf{e}_{y,k}(\mathbf{x_{op}}) = \begin{bmatrix} \mathbf{e}_{y,1}(\mathbf{x_{op}}) \\ \mathbf{e}_{y,2}(\mathbf{x_{op}}) \\ \vdots \\ \mathbf{e}_{y,M_k}(\mathbf{x_{op}}) \end{bmatrix} \tag{55}$$

$$\mathbf{G}_k = \begin{bmatrix} \mathbf{G}_{1,k} \\ \mathbf{G}_{2,k} \\ \vdots \\ \mathbf{G}_{M,k} \end{bmatrix} \tag{56}$$

Next, we will insert the above functions into the objective function to complete the Gauss-Newton Derivation:

$$\delta\mathbf{x} = \begin{bmatrix} \boldsymbol{\epsilon}_0 \\ \boldsymbol{\epsilon}_1 \\ \vdots \\ \boldsymbol{\epsilon}_K \end{bmatrix} \tag{57}$$

$$\mathbf{e}(\mathbf{x}_{op}) = \left[\begin{array}{c} \mathbf{e}_{v,k_1}(\mathbf{x}_{op}) \\ \mathbf{e}_{v,k_1+1}(\mathbf{x}_{op}) \\ \vdots \\ \mathbf{e}_{v,k_2}(\mathbf{x}_{op}) \\ \hline \mathbf{e}_{y,k_1}(\mathbf{x}_{op}) \\ \mathbf{e}_{y,k_1+1}(\mathbf{x}_{op}) \\ \vdots \\ \mathbf{e}_{y,k_2}(\mathbf{x}_{op}) \end{array}\right] \tag{58}$$

$$\mathbf{H} = \left[\begin{array}{ccccc} \mathbf{1} & & & & \\ -\mathbf{F}_{k_1} & \mathbf{1} & & & \\ & -\mathbf{F}_{k_1+1} & & & \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & -\mathbf{F}_{k_2-1} & \mathbf{1} \\ \hline \mathbf{G}_{k_1} & & & & \\ & \mathbf{G}_{k_1+1} & & & \\ & & \mathbf{G}_{k_1+2} & & \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & & \mathbf{G}_{k_2} \end{array}\right] \tag{59}$$

and

$$\mathbf{T} = \mathbf{W} = \mathrm{diag}(\hat{\mathbf{P}}_{k1}, \mathbf{Q}_{k_1+1}, \ldots, \mathbf{Q}_{k_2}, \mathbf{R}_{k_1}, \ldots, \mathbf{R}_{k_1+1}, \ldots, \mathbf{R}_{k_2}) \tag{60}$$

$$\mathbf{J}(\mathbf{x}) \approx \mathbf{J}(\mathbf{x}_{op}) - \mathbf{b}^T\delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^T\mathbf{A}\delta\mathbf{x} \tag{61}$$

We take the derivative of the cost function, and can obtain:

$$\mathbf{A}\delta\mathbf{x}^* = \mathbf{b}$$

$$\delta\mathbf{x}^* = \begin{bmatrix} \boldsymbol{\epsilon}^*_{k_1} \\ \boldsymbol{\epsilon}^*_{k_1+1} \\ \vdots \\ \boldsymbol{\epsilon}^*_{k_2} \end{bmatrix} \tag{62}$$

$$\mathbf{A} = \mathbf{H}^T\mathbf{W}^{-1}\mathbf{H}, \quad \mathbf{b} = \mathbf{H}^T\mathbf{W}^{-1}\mathbf{e}(\mathbf{x}_{op}) \tag{63}$$

Finally, we update our operating point through the original perturbation scheme:

$$\mathbf{T}_{op,k} \leftarrow \exp(\boldsymbol{\epsilon}^{*\wedge}_k)\mathbf{T}_{op,k} \tag{64}$$

This will iterate till it converges.

Question 4

Plot for number of visible landmarks:

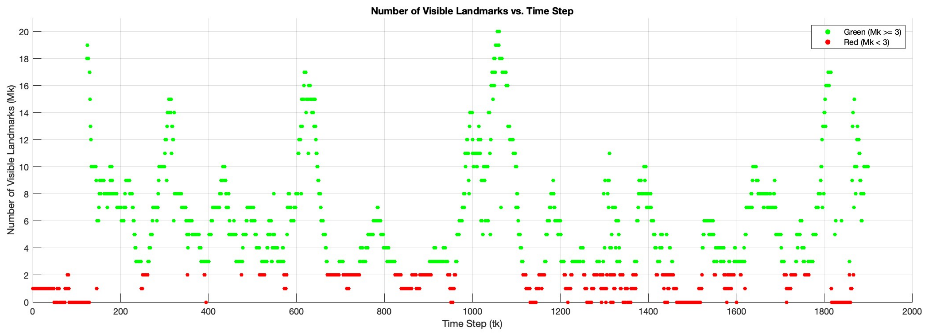


Figure 1: Scatter plot for the visible landmarks for all the timesteps (in total 1700 timesteps). When the dots are in red, meaning that the number of visible landmarks is less than 3. When the dots are in green, meaning that the number of visible landmarks is at least 3.

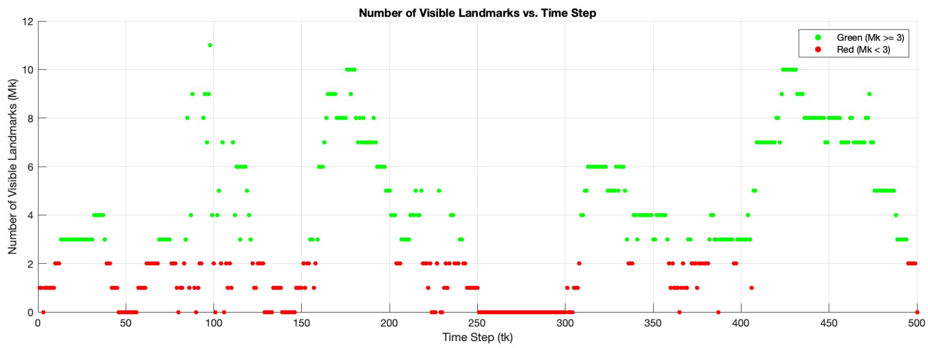


Figure 2: Scatter plot for the visible landmarks for timesteps between k_1 and k_2 (in total 500 timesteps). When the dots are in red, meaning that the number of visible landmarks is less than 3. When the dots are in green, meaning that the number of visible landmarks is at least 3.

Question 5

This section starts with the implementation of the batch estimation derived from above. The algorithm stacked the inputs and states from timestep $k_1 = 1215$ to $k_2 = 1714$ and iterated using the Gauss-Newton method for optimization.

Comments and Conclusions:

By running the above experiments of batch estimation and the sliding window approach, I am able to make the below observations:

- **Runtime:** Speed-wise, the smaller the window size is, the faster the algorithm will run. This is due to the smaller sized matrix there need to be constructed and manipulated. Especially the A matrix where the size is determined by $(6 \times N)^2$. The larger the size of the A matrix, the longer it will take for the Guass-Newton optimization to find the optimal solution.

- **Accuracy:** However, in terms of accuracy, the batch estimation in comparison is a lot more accurate. Both in comparision to the amount of error, and the covariance. In other words, batch estimation, or sliding window with a relatively large window size, will produce more accurate results – with less error and more confidence in the covariance (smaller in value). This can be seen from the plots above too. Among the three experiments, batch estimation has the smallest error, then followed by when window size equals 50. This can be explained considering that batch estimation is taking the whole time steps into account and optimize for the best pose all together, while the sliding window is more like a "locally optimized" solution.

- **Uncertainties:** Based on the plots, the uncertainties is largely influenced by the number of observable landmarks. The fewer observable or no observable landmarks there is, the more uncertain the estimates will be. This conclusion can be seen by comparing the plots from Question 4 directly to the error plots in Question 5 – noting that the timesteps between 1450 to 1500, where there is no observable landmarks, the estimation covariance is the largest in the error plots.

- **Efficiency:** Overall, the sliding window is surprisingly efficient and accurate compare to the batch estimation. Even when the window size decreased to 2, the error for the translations still remain all under 0.2m offset, and the error in the rotations are remain under 0.1rad. Especially considering the fact that it can basically run online if the window size is small enough. Also, when the window size equals to 1, this practically becomes a Kalman Filter. Hence, we can conclude that, this localization algorithm can be used as sliding window, if speed is the priority. And if the accuracy is the priority, like some extrinsic calibration problems, it is recommended to use batch estimation, as it can produce the most accurate results.

- **Covariance Estimate:** It is worth noting however that the uncertainty estimate for the sliding window is over-confident. The 3-sigma bounds do not cover up the estimated error plots as it should. However, the error plots for the batch estimation is perfectly bounded. It is suspected that this problem is occuring due to a poor initial covariance estimate. It is believed that the provided variance for the motion model and measurement model is calculated based on the provided ground-truth and the estimates using the batch estimation. Since sliding window does not take the entire pose into account, it can be over-confident and does not propagate the uncertainties as it should. Proposed way to correct this is to increase the provided covariance values so that the error plots will be within the 3-sigma bounds.

Experiment Results:

The results can be seen as follows:

Batch Estimation

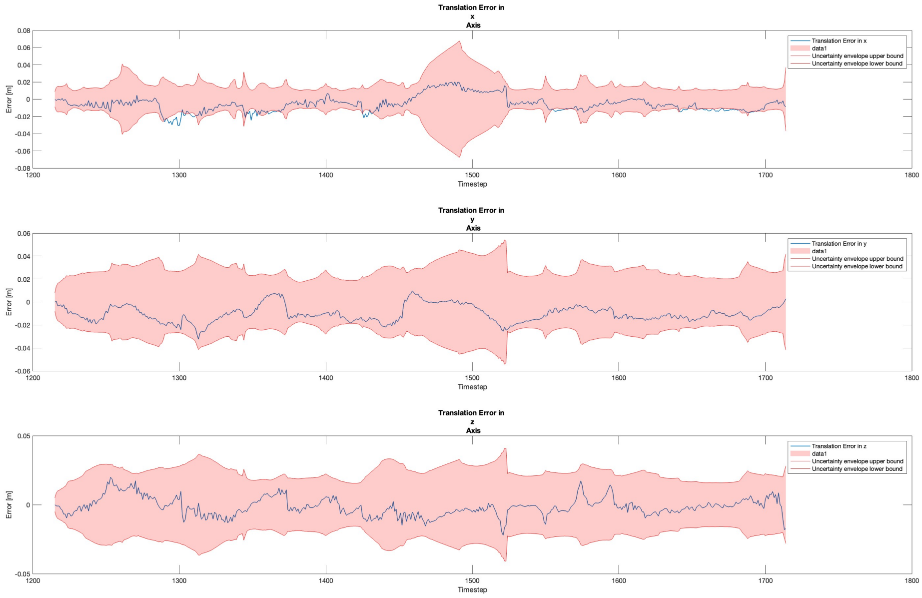


Figure 3: The batch estimation error for the translational states

Figure 4: The batch estimation error for the translational states

Sliding Window Estimation - window size = 50

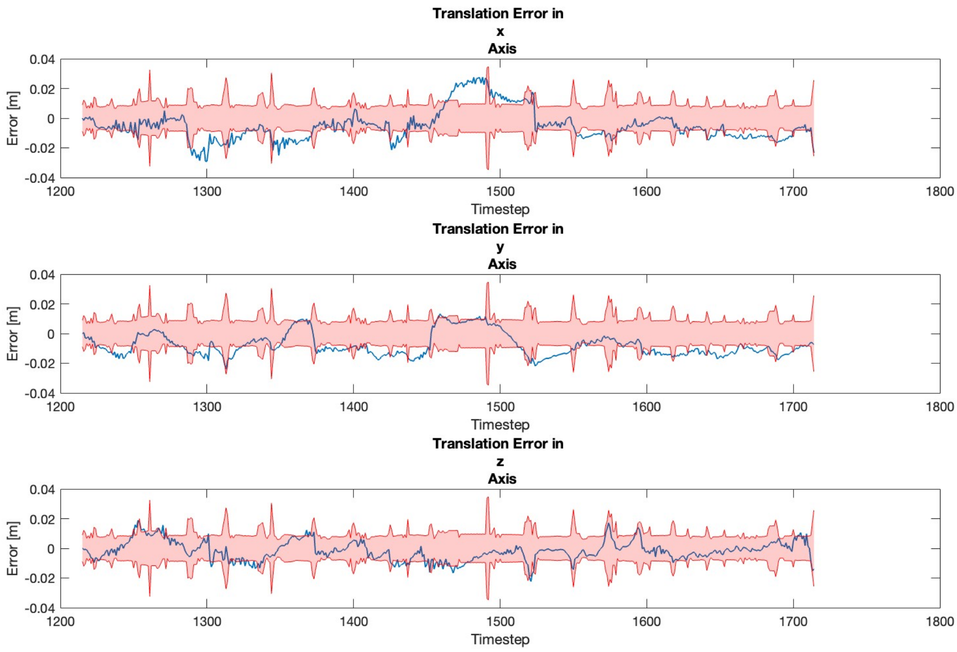


Figure 5: Sliding window estimation error for the translational states given window size of 50

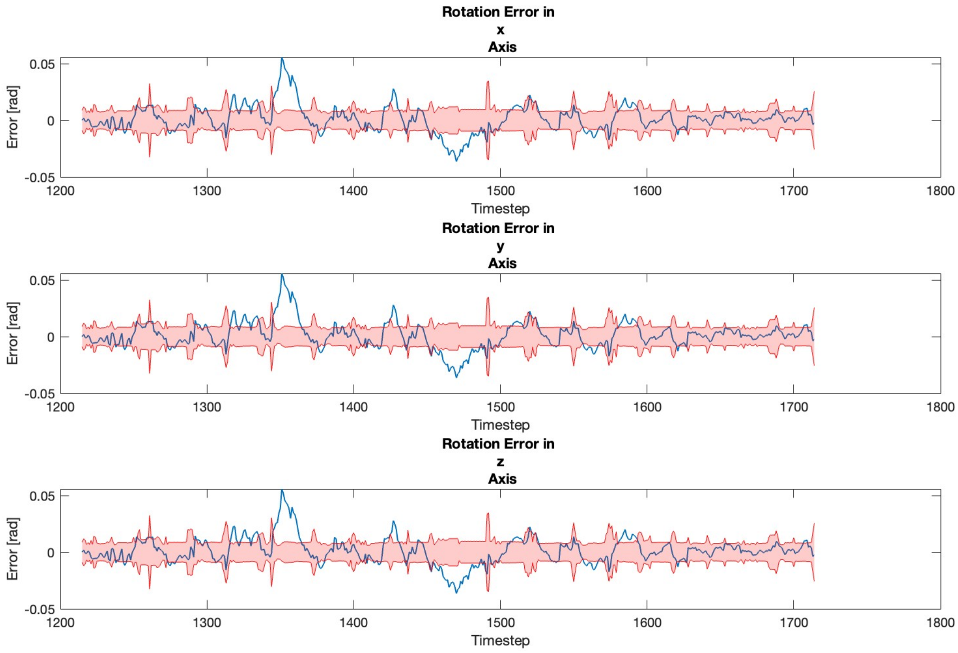


Figure 6: Sliding window estimation error for the rotational states given window size of 50

Sliding Window Estimation - window size = 10

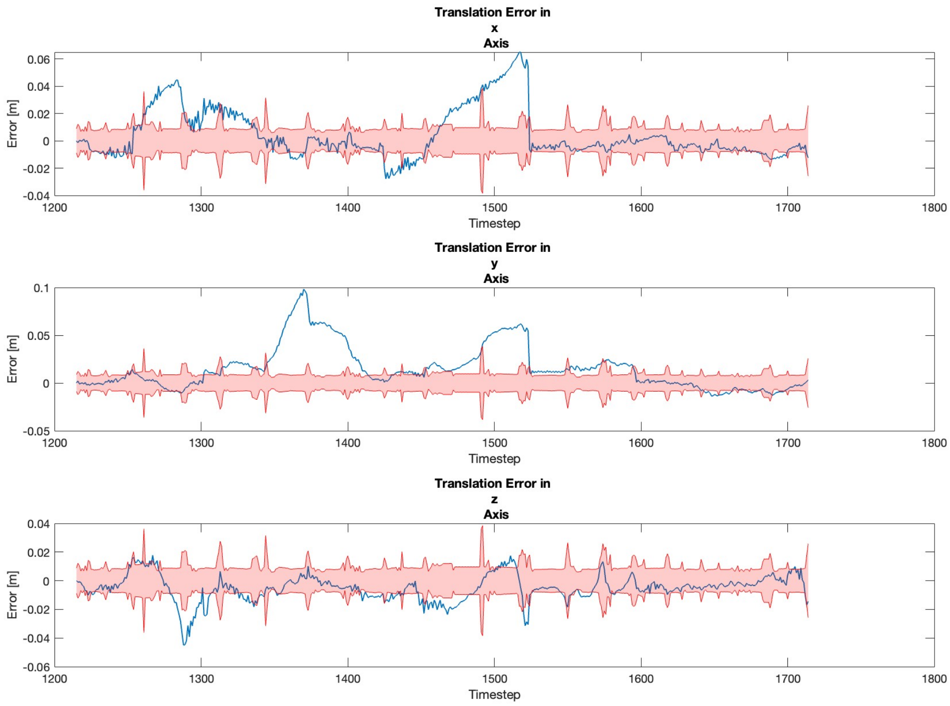


Figure 7: Sliding window estimation error for the translational states given window size of 10

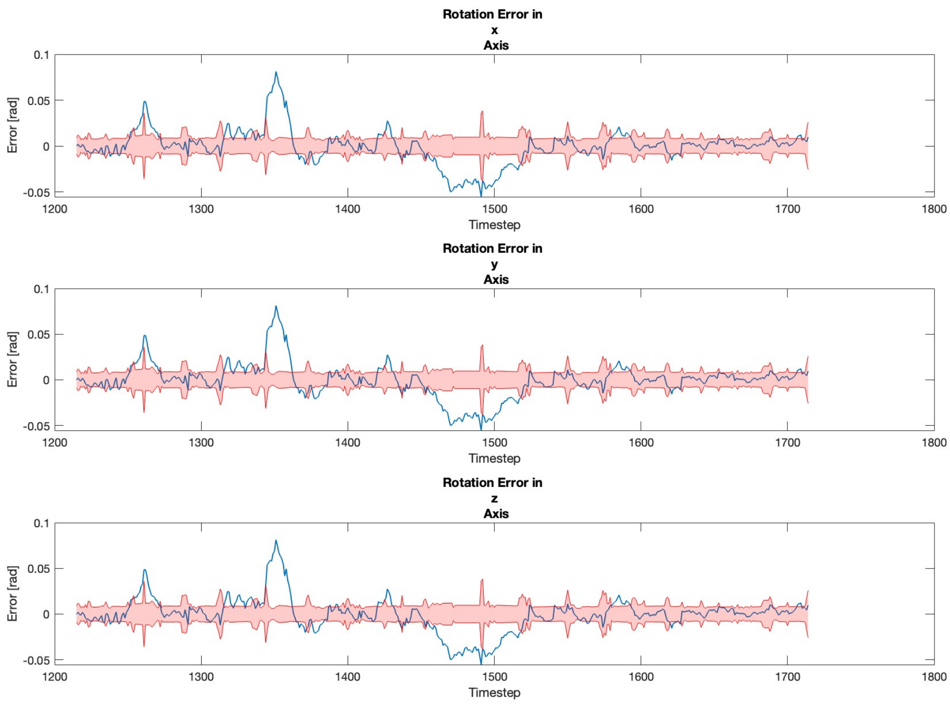


Figure 8: Sliding window estimation error for the rotational states given window size of 10

Sliding Window Estimation - window size = 2

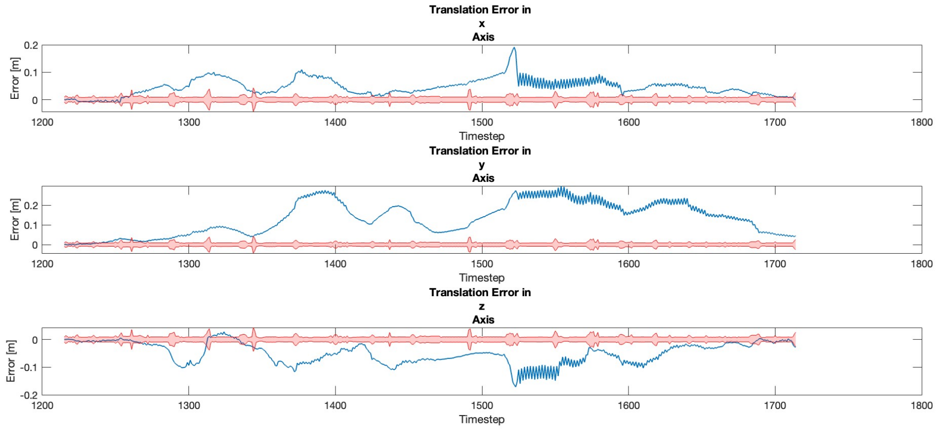


Figure 9: Sliding window estimation error for the translational states given window size of 2

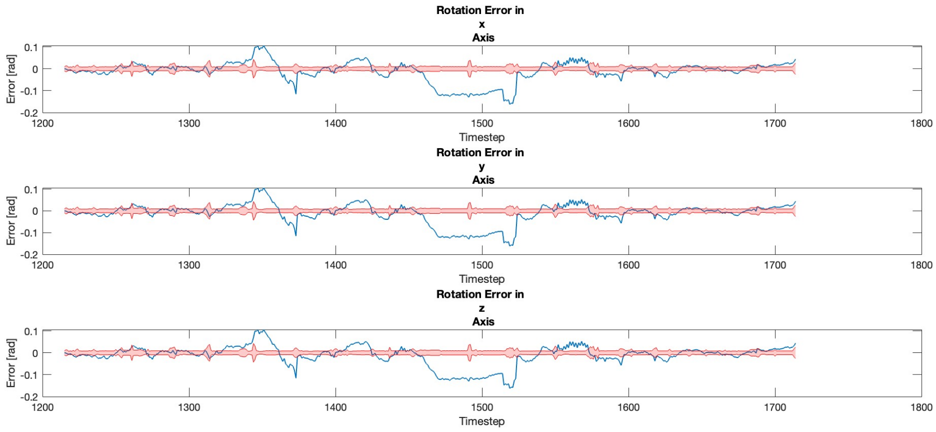


Figure 10: Sliding window estimation error for the rotational states given window size of 2

Appendix

Robot estimated trajectory plots vs ground truth

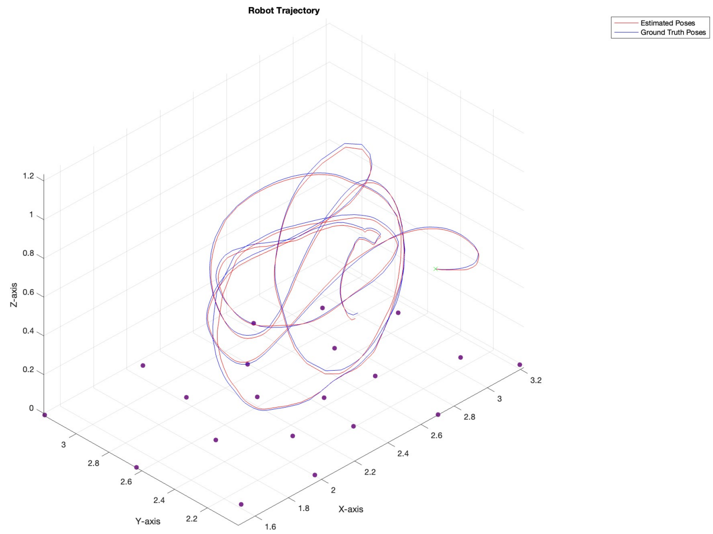


Figure 11: Estimated pose of the robot using batch estimation vs the ground truth pose of the robot. Red is the estimated pose, blue is the ground truth

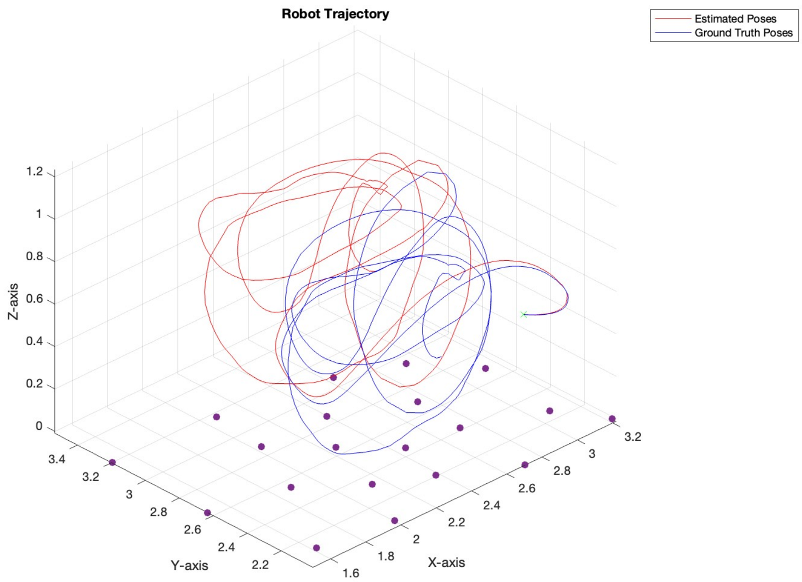


Figure 12: Estimated pose of the robot using dead reckoning vs the ground truth pose of the robot. Red is the estimated pose, blue is the ground truth

Ground truth plots with rotation representation

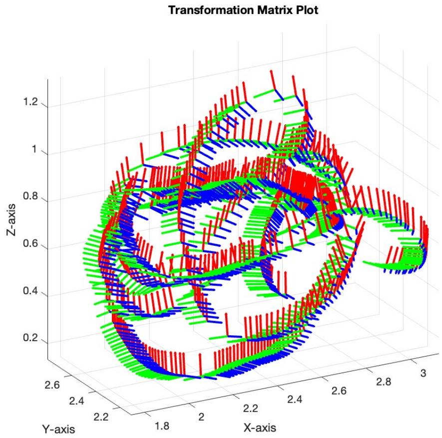
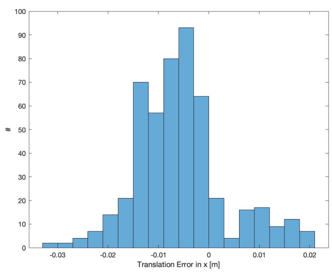


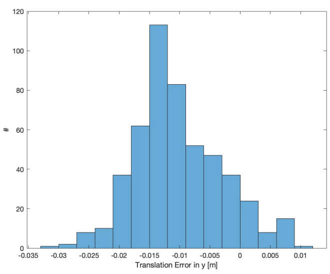
Figure 13: Ground truth plot of the robot pose. Red axis is the x-direction, blue is the y-direction and green is the z-direction

Histogram of the errors for batch estimation

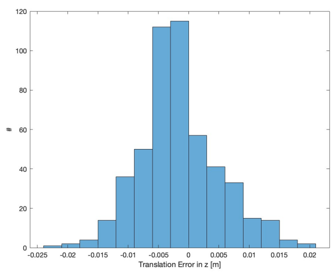
Based on the histogram of the errors, we can conclude that the Gaussian noise assumption holds valid. However, it is worth noting that for the rotational error in the x direction, there is a slight biases to the lower side. We could increase the provided covariance to increase our uncertainty bound.



(a) Translational error in x

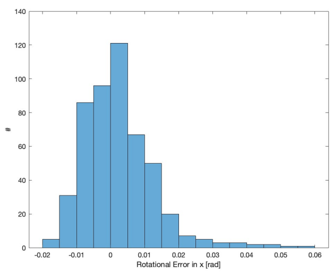


(b) Translational error in y

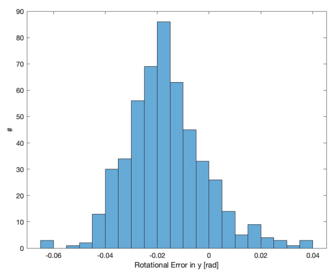


(c) Translational error in z

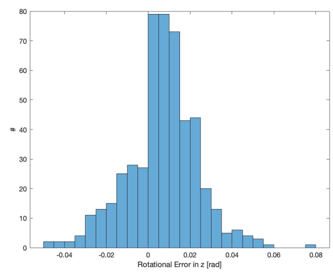
Figure 14: General caption for all three figures



(a) Rotational error in x



(b) Rotational error in y



(c) Rotational error in z

Figure 15: General caption for all three figures

Source Code:

1. Visiable Landmark Plots (Question 4)

```
load dataset3.mat
who

k1 = 1215;
k2 = 1714;

y_k_j = y_k_j(:, k1:k2, :);

%% Question 4:

% Initialize variables to store the number of visible landmarks and colors
numVisibleLandmarks = zeros(1, size(y_k_j, 2));
colors = cell(1, size(y_k_j, 2));

% Loop through each timestep
for t = 1:size(y_k_j, 2)
    % Extract the measurements at the current timestep
    measurements = squeeze(y_k_j(:, t, :));

    % Count the number of visible landmarks at the current timestep
    numVisible = sum(measurements(1, :) ~= -1); % Count values not equal to -1

    % Store the count and determine the color
    numVisibleLandmarks(t) = numVisible;
    if numVisible >= 3
        colors{t} = 'g'; % Green for at least three visible landmarks
    else
        colors{t} = 'r'; % Red otherwise
    end
end


% Create a plot of the number of visible landmarks vs. timestep
tk = 1:size(y_k_j, 2);
Mk = numVisibleLandmarks;

% Initialize colors
colors = cell(1, length(tk));
for t = 1:length(tk)
    if Mk(t) >= 3
        colors{t} = 'g'; % Green for at least three visible landmarks
    else
        colors{t} = 'r'; % Red otherwise
    end
end

% Create scatter plots for green and red dots
greenDots = scatter(tk(Mk >= 3), Mk(Mk >= 3), 20, 'g', 'filled');
hold on;
redDots = scatter(tk(Mk < 3), Mk(Mk < 3), 20, 'r', 'filled');
hold off;

% Create a custom legend
legend([greenDots, redDots], {'Green (Mk >= 3)', 'Red (Mk < 3)'});

xlabel('Time Step (tk)');
ylabel('Number of Visible Landmarks (Mk)');
title('Number of Visible Landmarks vs. Time Step');
```

```
% Customize plot appearance
grid on;
ylim([0, max(Mk) + 1]);
```

1. Batch Estimation: (Question 5(a))

main.m

```
clear all;
clc;

load dataset3.mat;
whos;

%% Some Basic Constants Here:
k1 = 1215;
k2 = 1714;
maxIterations = 10;

% K = k2 - k1 + 1;  % Total number of time steps
N = size(y_k_j, 3);  % Number of landmarks
K = size(t,2);
K_total = K;

%% Ground Truth:
T_i_vk = repmat(eye(4), [1, 1, K_total]);
T_vk_i = repmat(eye(4), [1, 1, K_total]);

for k = 1:K
    C_vk_i = vec2rot(theta_vk_i(:,k));
    C_i_vk = inv(C_vk_i);
    T_i_vk(:,:,k) = [C_vk_i, r_i_vk_i(:,k); 0,0,0,1];
    T_vk_i(:,:,k) = inv(T_i_vk(:,:,k));
end

T_gt = T_vk_i;

%% Initialization: -- using Dead Reckoning
T_op = repmat(eye(4), [1, 1, K_total]);
T_op(:,:,k1) = T_vk_i(:,:,k1);
T_op(:,:,1) = T_vk_i(:,:,1);
checkT0 = T_vk_i(:,:,1);

for k = k1+1:k2
    delta_t = t(k) - t(k-1);
    omega_k = [-v_vk_vk_i(:,k-1); -w_vk_vk_i(:,k-1)]; % input v and w
    xi_k = expm(delta_t * wedge(omega_k)); % added transformation matrix after v, w
        -- hamburger symbol
    T_op(:,:,k) = xi_k*T_op(:,:,k-1);
    T_op_i(:,:,k) = inv(T_op(:,:,k));
end

%% Measurement Matrix
D_mat = [1 0 0 0; 0 1 0 0; 0 0 1 0];
D = D_mat';
T_cv = [C_c_v, -C_c_v * rho_v_c_v; zeros(1, 3), 1]; % Define T_cv matrix (
    Transformation from vehicle to camera)

%% Test if T_op = T_gt
% T_op = T_gt;

```

```
%% MAIN LOOP:
for iteration = 1:maxIterations
    e_v = cell(K, 1);
    F = cell(K, 1);
    Q = cell(K, 1);
    e_y = cell(K, 1);
    G = cell(K, 1);
    R = cell(K, 1);
    A_mat = [];
    b_mat = [];
    delta_x_star = [];

    %% Motion Model Error:
    [e_v, F, Q] = calculateMotionModelError(T_op, T_gt, v_vk_vk_i, w_vk_vk_i, v_var,
        w_var, t, k1, k2);

    %% Measurement Model Error:
    [e_y, G, R] = calculateMeasurementModelError(T_op, T_gt, y_k_j, rho_i_pj_i, D,
        T_cv, fu, fv, cu, cv, b, y_var, k1, k2, N);

    %% Formulate H, W, A, b and e: (H'*inv(W)*H) * x = (H'*inv(W)*e)
    [A_mat, b_mat, H, W_inv, e_stack, e_v_stack, e_y_stack] = calculateNewAB(e_v, e_y
        , F, G, Q, R, k1, k2);

    %% Optimization Solver -- using Chol. Decomp.
    [T_op, delta_x_star] = optimizeAndUpdate(A_mat, b_mat, T_op, k1, k2);
    eps = norm(delta_x_star);

    %% Check if the condition is met:
    % plot_error(T_op, T_gt, A_mat, k1, k2); % plot the translational and rotational
        errors:
    fprintf('The current iteration is: %d, and error is at %f ......\n \n', iteration
        , eps); % Print the current iteration and error
    if eps < 10^-3
        disp("The pose estimation successfully converges! ")
        break;
    end
end

%% End of the MAIN LOOP

%% Plot the errors:
plot_error_batch(T_op, T_gt, A_mat, k1, k2);
```

calculateMotionModelError.m

```
    function [e_v, F, Q] = calculateMotionModelError(T_op, T_gt, v_vk_vk_i, w_vk_vk_i
        , v_var, w_var, t, k1, k2)
    % Initialize motion model error, Jacobian matrix, and covariance matrix for all
        timesteps
    K = size(T_op, 3);  % Assuming T_op is 3D with the third dimension being
        timesteps
    e_v = cell(K, 1);
    F = cell(K, 1);
    Q = cell(K, 1);
    checkT0 = T_gt(:,:,k1);
    k_start = 1215;
    k_end = 1714;
    T_op(:,:,k_start) = T_gt(:,:,k_start);

    for k = k1:k2
        if k == k_start
            % First input error
```

```
            delta_t = t(k) - t(k-1);
            T_diff = checkT0 * inv(T_op(:,:,k1)); % checkT0 is taken from the ground
                truth
            e_v{k} = vee(logm(T_diff)); % vee operator to convert matrix to vector

            F{k} = adjoint_SE3(T_op(:,:,k) * inv(checkT0)); % Adjoint of the relative
                 transformation -- linearized
            Q{k} = diag([ 1./((v_var)*delta_t^2); 1./((w_var)*delta_t^2) ]);  %
                Constructing 6x6 covariance matrix for each timestep

        else

            delta_t = t(k) - t(k-1);
            omega_k = [-v_vk_vk_i(:,k-1); -w_vk_vk_i(:,k-1)];
            xi_k = expm(delta_t * wedge(omega_k));
            e_v{k} = vee(logm(xi_k * T_op(:,:,k-1) * inv(T_op(:,:,k)))); % Later
                input errors

            F{k} = adjoint_SE3(T_op(:,:,k) * inv(T_op(:,:,k-1))); % Adjoint of the
                relative transformation -- linearized
            Q{k} = diag([ 1./((v_var)*delta_t^2); 1./((w_var)*delta_t^2) ]);  %
                Constructing 6x6 covariance matrix for each timestep
        end
    end
end
```

calculateMeasurementModelError.m

```
    function [e_y, G, R] = calculateMeasurementModelError(T_op, T_gt, y_k_j,
        rho_i_pj_i, D, T_cv, fu, fv, cu, cv, b, y_var, k1, k2, N)
    % Initialize measurement model error, Jacobian matrix, and covariance matrix for
        all timesteps

    K = size(T_op, 3);  % Assuming T_op is 3D with the third dimension being
        timesteps
    e_y = cell(K, 1); % Initialize measurement model error as a cell array, one cell
        for each timestep
    G = cell(K, 1); % Initialize Jacobian matrix G as a cell array, one cell for each
         timestep
    R = cell(K, 1);

    M = [fu, 0,  cu, 0;
         0,  fv, cv, 0;
         fu, 0,  cu, -fu * b;
         0,  fv, cv, 0]; % camera intrinsic matrix

    for k = k1:k2
        e_y_k = [];  % Initialize error for this timestep
        G_k = [];    % Initialize Jacobian for this timestep
        R_k = [];

        for j = 1:N % N = 20
            if all(y_k_j(:, k, j) ~= -1)  % Check if the landmark is observed

                % Coordinate transform:
                p_i_pj_i = [rho_i_pj_i(:, j); 1]; % Compute transformed point z in
                    the camera frame
                p_j_c =  T_cv * T_op(:,:,k) * p_i_pj_i;

                % e_y_kj:
                y_k_j_pred = (M*p_j_c)./p_j_c(3);
                e_y_kj = y_k_j(:, k, j) - y_k_j_pred; % Calculate the error
                e_y_k = [e_y_k; e_y_kj];  % Stack errors for all observed landmarks
```

```

                % Jacobian
                z = D' * T_cv * T_op(:,:,k) * p_i_pj_i;
                J_g = jacobianG(z, fu, fv, cu, cv, b);    % Compute the derivative of
                     the observation model g at z
                G_jk = J_g * D' * T_cv * odot(T_op(:,:,k) * p_i_pj_i); % Compute G_jk
                G_k = [G_k; G_jk];        % Stack Jacobians for all observed
                    landmarks

                % Covariance:
                R_k_j = diag([1./y_var]); % Calculate the covariance
                R_k = blkdiag(R_k, R_k_j);
            end
        end

        % Store the errors and stacked Jacobians for this time step in the cell
            arrays
        e_y{k} = e_y_k;
        G{k} = G_k;
        R{k} = R_k;
    end

end


function J_g = jacobianG(z, fu, fv, cu, cv, b)
    x = z(1);
    y = z(2);
    z_val = z(3);

    J_g = [fu/z_val, 0, -fu*x/z_val^2;
           0, fv/z_val, -fv*y/z_val^2;
           fu/z_val, 0, -fu*(x-b)/z_val^2;
           0, fv/z_val, -fv*y/z_val^2];
end

function pixel_coord = estimatePixelLocation(p_i_pj_i, T_cv, T_k, fu, fv, cu, cv, b)
    % Define D matrix (for projection)
    D = [1, 0, 0, 0;
         0, 1, 0, 0;
         0, 0, 1, 0];

    % Transform the landmark position to the camera frame
    z = D * T_cv * T_k * [p_i_pj_i; 1];

    % Use the camera intrinsic parameters to project onto pixel coordinates
    pixel_coord = cameraIntrinsic(z, fu, fv, cu, cv, b);
end

function pixel_coord = cameraIntrinsic(z, fu, fv, cu, cv, b)
    % Form the 3D point in homogeneous coordinates
    p = [z; 1];

    % Camera intrinsic matrix for the stereo camera
    M = [fu, 0,  cu, 0;
         0,  fv, cv, 0;
         fu, 0,  cu, -fu * b;
         0,  fv, cv, 0];

    % Project the point to pixel coordinates in the stereo image
    pixel_coordinates = M * p;
    pixel_coord = [pixel_coordinates(1) / z(3);  % x-coordinate in left image
```

```
                    pixel_coordinates(2) / z(3);  % y-coordinate in left image
                    pixel_coordinates(3) / z(3);  % x-coordinate in right image
                    pixel_coordinates(4) / z(3)]; % y-coordinate in right image
end
```

calculateNewAB.m

```
    function [A, b, H, W_inv, e_stack, e_v_stack, e_y_stack] = calculateNewAB(e_v,
        e_y, F, G, Q, R, k1, k2)

    %% Calculate W and e
    % Stack all errors from e_v and e_y and covariance matrix W
    e_stack = [];
    e_v_stack = [];
    e_y_stack = [];
    W_stack = [];
    Q_stack = [];
    R_stack = [];
    empty_error = 0;

    for k = k1:k2
        e_v_stack = [e_v_stack; e_v{k}];
        Q_stack = blkdiag(Q_stack, Q{k});

        if isempty(e_y{k})
            % e_y_stack = [e_y_stack; zeros(0,1)];
            % R_stack = blkdiag(R_stack, zeros(0,4));
            empty_error = empty_error+1 ;
        else
            e_y_stack = [e_y_stack; e_y{k}];
            R_stack = blkdiag(R_stack, R{k});
        end
    end
    e_stack = [e_v_stack; e_y_stack];
    W_stack = blkdiag(Q_stack, R_stack);
    % fprintf('The motion model error is %f \n', norm(e_v_stack));
    % fprintf('The measurement model error is %f \n', norm(e_y_stack));
    e = e_stack;
    W_inv = W_stack;

    %% Calculate H

    total_e_v_size = size(e_v_stack, 1); % Determine the total size of H
    total_e_y_size = size(e_y_stack, 1);
    H_size = total_e_v_size + total_e_y_size;
    H_v = zeros(total_e_v_size, total_e_v_size); % Preallocate H
    idx = 1; % Initialize index for filling H

    %% H_v calculation:
    for k = k1:k2
        size_e_v_k = size(e_v{k}, 1); % Size of the current e_v and e_y
        H_v(idx:idx+5, idx:idx+5) = eye(size_e_v_k); % Fill in the blocks for the
            motion model
        if k > k1
            H_v(idx:idx+5, idx-6:idx-1) = -F{k-1};
        end
        idx = idx + 6; % Update index
    end

    total_e_y_size = 0;

    for k = k1:k2
        if ~isempty(e_y{k})
```

```
            total_e_y_size = total_e_y_size + size(e_y{k}, 1);
        end
    end

    idx = 1; % Initialize the index for filling H_y
    empty_meas = 0;

    %% H_y calculation:
    H_y = zeros(total_e_y_size, 6 * (k2 - k1 + 1)); % Preallocate H_y with the correct size

    % Loop through each timestep
    for k = k1:k2
        size_e_y_k = size(e_y{k}, 1);
        if isempty(G{k})       % Check if the measurement at this timestep is empty
            empty_meas = empty_meas + 1; % If G{k} is empty, no need to fill H_y for this timestep
        else
            col_idx_start = 6 * (k - k1) + 1; % If G{k} is not empty, fill the corresponding part of H_y % Calculate the column index range for G{k} in H_y
            col_idx_end = col_idx_start + 5;
            H_y(idx:idx+size_e_y_k-1, col_idx_start:col_idx_end) = G{k};
        end
        idx = idx + size_e_y_k; % Update the row index for the next timestep
    end

    H = [H_v; H_y]; % Stack H_v and H_y together
    H_T_W_inv = H' * W_inv;

    A = H_T_W_inv * H; % Compute A and b
    b = H_T_W_inv * e;
end
```

optimizeAndUpdate.m

```
    function [T_op, delta_x_star] = optimizeAndUpdate(A, b, T_op, k1, k2)
    % Optimization Solver using Cholesky Decomposition

    if isequal(A, A') && all(eig(A) > 0)
        % Perform Cholesky decomposition
        L = chol(A, 'lower');

        % Solve for delta_x using forward and backward substitution
        y = L \ b;           % Forward substitution
        delta_x_star = L' \ y; % Backward substitution
    else
        % If A is not symmetric positive definite, fall back to another solver
        % warning('Matrix A is not symmetric positive definite. Using pinv for solving.');
        delta_x_star = pinv(A) * b;
    end

    % Update the operating point
    for k = (k1+1):k2
        % Extract perturbation for timestep k
        eps_k_star = delta_x_star((k - k1) * 6 + 1:(k - k1) * 6 + 6);

        % Update T_op using the perturbation
        T_op(:,:,k) = expm(wedge(eps_k_star)) * T_op(:,:,k);
    end
end
```

plot_error_batch.m

```
function plot_error_batch(T_op, T_gt, A, k1, k2)
% Initialize error arrays
rot_err = zeros(k2-k1+1, 3);
trans_err = zeros(k2-k1+1, 3);
% A = A(k1*6:(k2+1)*6-1, k1*6:(k2+1)*6-1);

% Calculate errors
for k = k1:k2
    C_gt = T_gt(1:3,1:3,k);
    C_op = T_op(1:3,1:3,k);

    r_gt = -C_gt' * T_gt(1:3,4,k);
    r_op = -C_op' * T_op(1:3,4,k);

    rot_err(k-k1+1, :) = get_inv_cross_op(eye(3) - C_op * C_gt');
    trans_err(k-k1+1, :) = r_op - r_gt;
end

% Print average errors
fprintf('Avg Rot Err: %f\n', mean(abs(rot_err), 'all'));
fprintf('Avg Trans Err: %f\n', mean(abs(trans_err), 'all'));

% Calculate variances
var = diag(inv(A));
var_tx = var(1:6:end);
var_ty = var(2:6:end);
var_tz = var(3:6:end);
var_rx = var(4:6:end);
var_ry = var(5:6:end);
var_rz = var(6:6:end);

% Time vector
t = k1:k2;

%% Histogram
figure;
histogram(trans_err(:, 1)', 'DisplayName', 'Translation Error in x');
ylabel('#');
xlabel('Translation Error in x [m]');

figure;
histogram(trans_err(:, 2)', 'DisplayName', 'Translation Error in y');
ylabel('#');
xlabel('Translation Error in y [m]');

figure;
histogram(trans_err(:, 3)', 'DisplayName', 'Translation Error in z');
ylabel('#');
xlabel('Translation Error in z [m]');

figure;
histogram(rot_err(:, 1)', 'DisplayName', 'Rotational Error in x');
ylabel('#');
xlabel('Rotational Error in x [rad]');

figure;
histogram(rot_err(:, 2)', 'DisplayName', 'Rotational Error in y');
ylabel('#');
xlabel('Rotational Error in y [rad]');

figure;
```

```
    histogram(rot_err(:, 3)', 'DisplayName', 'Rotational Error in z');
    ylabel('#');
    xlabel('Rotational Error in z [rad]');


    %% Plotting translational error
    figure;
    axis = ["x", "y", "z"];
    % for i = 1:3
    subplot(3, 1, 1);
    plot(t, trans_err(:, 1), 'DisplayName', 'Translation Error in x','LineWidth',1);
    hold on;
    % fill_between(t, -3 * sqrt(var_tx(i,:)), +3 * sqrt(var_tx(i,:)), '#FF9848');
    fill([t'; flipud(t')], [+3 * sqrt(var_tx); flipud(-3 * sqrt(var_tx))], 'r', '
        FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_tx), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty
        envelope upper bound');
    hold on;
    plot(t, -3 * sqrt(var_tx), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty
        envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [m]');
    title(['Translation Error in ' axis(1) ' Axis']);
    legend;

    subplot(3, 1, 2);
    plot(t, trans_err(:, 2), 'DisplayName', 'Translation Error in y','LineWidth',1);
    hold on;
    % fill_between(t, -3 * sqrt(var_tx(i,:)), +3 * sqrt(var_tx(i,:)), '#FF9848');
    fill([t'; flipud(t')], [+3 * sqrt(var_ty); flipud(-3 * sqrt(var_ty))], 'r', '
        FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_ty), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty
        envelope upper bound');
    hold on;
    plot(t, -3 * sqrt(var_ty), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty
        envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [m]');
    title(['Translation Error in ' axis(2) ' Axis']);
    legend;


    subplot(3, 1, 3);
    plot(t, trans_err(:, 3), 'DisplayName', 'Translation Error in z','LineWidth',1);
    hold on;
    % fill_between(t, -3 * sqrt(var_tx(i,:)), +3 * sqrt(var_tx(i,:)), '#FF9848');
    fill([t'; flipud(t')], [+3 * sqrt(var_tz); flipud(-3 * sqrt(var_tz))], 'r', '
        FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_tz), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty
        envelope upper bound');
    hold on;
    plot(t, -3 * sqrt(var_tz), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty
        envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [m]');
    title(['Translation Error in ' axis(3) ' Axis']);
    legend;


    %% Plotting rotational error
    figure;

```

```
    % X Axis
    subplot(3, 1, 1);
    plot(t, rot_err(:, 1), 'DisplayName', 'Rotation Error', 'LineWidth',1);
    hold on;
    fill([t'; flipud(t')], [+3 * sqrt(var_rx); flipud(-3 * sqrt(var_rx))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_rx), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    plot(t, -3 * sqrt(var_rx), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [rad]');
    title(['Rotation Error in ' axis(1) ' Axis']);
    legend;

    % Y Axis
    subplot(3, 1, 2);
    plot(t, rot_err(:, 1), 'DisplayName', 'Rotation Error', 'LineWidth',1);
    hold on;
    fill([t'; flipud(t')], [+3 * sqrt(var_ry); flipud(-3 * sqrt(var_ry))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_ry), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    plot(t, -3 * sqrt(var_ry), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [rad]');
    title(['Rotation Error in ' axis(2) ' Axis']);
    legend;

    % Z Axis
    subplot(3, 1, 3);
    plot(t, rot_err(:, 1), 'DisplayName', 'Rotation Error', 'LineWidth',1);
    hold on;
    fill([t'; flipud(t')], [+3 * sqrt(var_rz); flipud(-3 * sqrt(var_rz))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_rz), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    plot(t, -3 * sqrt(var_rz), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [rad]');
    title(['Rotation Error in ' axis(3) ' Axis']);
    legend;



end
```

3. Sliding Windows: (Question 5(b)(c))

```
clear all;
clc;

load dataset3.mat;
whos;


%% Some Basic Constants Here:
maxIterations = 10;
N = size(y_k_j, 3);  % Number of landmarks
```

```matlab
K = size(t,2);
K_total = K;

% k1 = 1215;
% k2 = 1714;
k_start = 1215;
k_end = 1714;

k1 = k_start;
k2 = k_end;


%% Ground Truth:
T_i_vk = repmat(eye(4), [1, 1, K_total]);
T_vk_i = repmat(eye(4), [1, 1, K_total]);

for k = 1:K
    C_vk_i = vec2rot(theta_vk_i(:,k));
    C_i_vk = inv(C_vk_i);
    T_i_vk(:,:,k) = [C_vk_i, r_i_vk_i(:,k); 0,0,0,1];
    T_vk_i(:,:,k) = inv(T_i_vk(:,:,k));
end

T_gt = T_vk_i;

%% Initialization: -- using Dead Reckoning
T_op = repmat(eye(4), [1, 1, K_total]);
T_op(:,:,k_start) = T_vk_i(:,:,k_start);
T_op(:,:,1) = T_vk_i(:,:,1);
checkT0 = T_vk_i(:,:,1);

for k = k1+1:k2
    delta_t = t(k) - t(k-1);
    omega_k = [-v_vk_vk_i(:,k-1); -w_vk_vk_i(:,k-1)]; % input v and w
    xi_k = expm(delta_t * wedge(omega_k)); % added transformation matrix after v, w
        -- hamburger symbol
    T_op(:,:,k) = xi_k*T_op(:,:,k-1);
end

T_est = T_op;
covariance = zeros(6,500); % initialize the covariance variable

% visualize_T_op(T_op(:,:,k_start:k_end));

%% Measurement Matrix
D_mat = [1 0 0 0; 0 1 0 0; 0 0 1 0];
D = D_mat';
T_cv = [C_c_v, -C_c_v * rho_v_c_v; zeros(1, 3), 1]; % Define T_cv matrix (
    Transformation from vehicle to camera)

%% MAIN LOOP for Sliding Windows:
window_size = 10;
k2 = k1 + window_size - 1;
iteration = 0;

% Initialize the estimation:
[T_op, var, eps] = batch_estimation(T_op, T_gt, N, v_vk_vk_i, w_vk_vk_i, y_k_j, v_var
    , rho_i_pj_i, w_var, y_var, t, k1, k2, D, T_cv, fu, fv, cu, cv, b, K);

% assign the estimated value to storage
T_est(:,:,k1) = T_op(:,:,k1);
% covariance(:, 1:window_size) = var;
```

```
var_tx = [];
var_ty = [];
var_tz = [];
var_rx = [];
var_ry = [];
var_rz = [];

while k1 ~= k_end+1

    T_op = dead_reckoning(k1, k2, w_vk_vk_i, v_vk_vk_i, t, T_est, T_op);
    [T_op, var, eps] = batch_estimation(T_op, T_gt, N, v_vk_vk_i, w_vk_vk_i, y_k_j,
        v_var, rho_i_pj_i, w_var, y_var, t, k1, k2, D, T_cv, fu, fv, cu, cv, b, K);

    var_tx = [var_tx, var(1,1)];
    var_ty = [var_ty, var(1,1)];
    var_tz = [var_tz, var(1,1)];
    var_rx = [var_rx, var(1,1)];
    var_ry = [var_ry, var(1,1)];
    var_rz = [var_rz, var(1,1)];

    k1 = k1+1;
    k2 = k1+window_size-1;
    iteration = iteration + 1;

    T_est(:,:,k1) = T_op(:,:,k1);

    fprintf("This is the timestep %d, and the error is: %f \n Itereation %d \n\n", k1
        , eps, iteration);


end

%% End of the MAIN LOOP

%% Plot the errors:
k1 = k_start;
k2 = k_end;
plot_error(T_est, T_gt, var_tx, var_ty, var_tz, var_rx, var_ry, var_rz, k1, k2);

% visualize_T_op(T_est(:,:,k_start:k_end));
% visualize_T_op(T_gt(:,:,k_start:k_end));
```

dead_reckoning.m

```
function T_op = dead_reckoning(k1, k2, w_vk_vk_i, v_vk_vk_i, t, T_est, T_op)

    T_op(:,:,k1) = T_est(:,:,k1);
    k_end = 1900;

    for k = k1+1:k_end
        delta_t = t(k) - t(k-1);
        omega_k = [-v_vk_vk_i(:,k-1); -w_vk_vk_i(:,k-1)]; % input v and w
        xi_k = expm(delta_t * wedge(omega_k)); % added transformation matrix after v,
            w -- hamburger symbol
        T_op(:,:,k) = xi_k*T_op(:,:,k-1);
    end

end
```

batch_estimation.m

```
function [T_op, var_stack, eps] = batch_estimation(T_op, T_gt, N, v_vk_vk_i,
    w_vk_vk_i, y_k_j, v_var, rho_i_pj_i, w_var, y_var, t, k1, k2, D, T_cv, fu, fv, cu
    , cv, b, K)
```

```
%% MAIN LOOP:
    for iteration = 1:50
        e_v = cell(K, 1);
        F = cell(K, 1);
        Q = cell(K, 1);
        e_y = cell(K, 1);
        G = cell(K, 1);
        R = cell(K, 1);
        A_mat = [];
        b_mat = [];
        delta_x_star = [];

        %% Motion Model Error:
        [e_v, F, Q] = calculateMotionModelError(T_op, T_gt, v_vk_vk_i, w_vk_vk_i,
            v_var, w_var, t, k1, k2);

        %% Measurement Model Error:
        [e_y, G, R] = calculateMeasurementModelError(T_op, T_gt, y_k_j, rho_i_pj_i, D
            , T_cv, fu, fv, cu, cv, b, y_var, k1, k2, N);

        %% Formulate H, W, A, b and e: (H'*inv(W)*H) * x = (H'*inv(W)*e)
        [A_mat, b_mat, H, W_inv, e_stack, e_v_stack, e_y_stack] = calculateNewAB(e_v,
             e_y, F, G, Q, R, k1, k2);

        %% Optimization Solver -- using Chol. Decomp.
        [T_op, delta_x_star] = optimizeAndUpdate(A_mat, b_mat, T_op, k1, k2);
        eps = norm(delta_x_star);

        fprintf('The current iteration is: %d, and error is at %f ......\n \n',
            iteration, eps); % Print the current iteration and error
        if eps < 5*10^-4
            break;
        end

    end

    %% End of the MAIN LOOP

    % Calculate variances
    var = diag(inv(A_mat));
    var_tx = var(1:6:end);
    var_ty = var(2:6:end);
    var_tz = var(3:6:end);
    var_rx = var(4:6:end);
    var_ry = var(5:6:end);
    var_rz = var(6:6:end);
    var_stack = [var_tx,var_ty,var_tz,var_rx,var_ry,var_rz]';

end
```

plot_error.m

```
    function plot_error(T_op, T_gt, var_tx, var_ty, var_tz, var_rx, var_ry, var_rz,
        k1, k2)
    % Initialize error arrays
    rot_err = zeros(k2-k1+1, 3);
    trans_err = zeros(k2-k1+1, 3);

    % Calculate errors
    for k = k1:k2
        C_gt = T_gt(1:3,1:3,k);
        C_op = T_op(1:3,1:3,k);
```

```
    r_gt = -C_gt' * T_gt(1:3,4,k);
    r_op = -C_op' * T_op(1:3,4,k);

    rot_err(k-k1+1, :) = get_inv_cross_op(eye(3) - C_op * C_gt');
    trans_err(k-k1+1, :) = r_op - r_gt;
end

% Print average errors
fprintf('Avg Rot Err: %f\n', mean(abs(rot_err), 'all'));
fprintf('Avg Trans Err: %f\n', mean(abs(trans_err), 'all'));

  % Time vector
t = k1:k2;
t = t';
var_tx = var_tx';
var_ty = var_ty';
var_tz = var_tz';

var_rx = var_rx';
var_ry = var_ry';
var_rz = var_rz';

%% Histogram
% figure;
% histogram(trans_err(:, 1)', 'DisplayName', 'Translation Error in x');
% ylabel('#');
% xlabel('Translation Error in x [m]');
%
% figure;
% histogram(trans_err(:, 2)', 'DisplayName', 'Translation Error in y');
% ylabel('#');
% xlabel('Translation Error in y [m]');
%
% figure;
% histogram(trans_err(:, 3)', 'DisplayName', 'Translation Error in z');
% ylabel('#');
% xlabel('Translation Error in z [m]');
%
% figure;
% histogram(rot_err(:, 1)', 'DisplayName', 'Rotational Error in x');
% ylabel('#');
% xlabel('Rotational Error in x [rad]');
%
% figure;
% histogram(rot_err(:, 2)', 'DisplayName', 'Rotational Error in y');
% ylabel('#');
% xlabel('Rotational Error in y [rad]');
%
% figure;
% histogram(rot_err(:, 3)', 'DisplayName', 'Rotational Error in z');
% ylabel('#');
% xlabel('Rotational Error in z [rad]');


%% Plotting translational error
figure;
axis = ["x", "y", "z"];
% for i = 1:3
subplot(3, 1, 1);
plot(t, trans_err(:, 1), 'DisplayName', 'Translation Error in x','LineWidth',1);
hold on;
```

```
    % fill_between(t, -3 * sqrt(var_tx(i,:)), +3 * sqrt(var_tx(i,:)), '#FF9848');
    fill([t; flipud(t)], [+3 * sqrt(var_tx); flipud(-3 * sqrt(var_tx))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_tx), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    hold on;
    plot(t, -3 * sqrt(var_tx), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [m]');
    title(['Translation Error in ' axis(1) ' Axis']);
    % legend;

    subplot(3, 1, 2);
    plot(t, trans_err(:, 2), 'DisplayName', 'Translation Error in y','LineWidth',1);
    hold on;
    % fill_between(t, -3 * sqrt(var_tx(i,:)), +3 * sqrt(var_tx(i,:)), '#FF9848');
    fill([t; flipud(t)], [+3 * sqrt(var_ty); flipud(-3 * sqrt(var_ty))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_ty), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    hold on;
    plot(t, -3 * sqrt(var_ty), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [m]');
    title(['Translation Error in ' axis(2) ' Axis']);
    % legend;


    subplot(3, 1, 3);
    plot(t, trans_err(:, 3), 'DisplayName', 'Translation Error in z','LineWidth',1);
    hold on;
    % fill_between(t, -3 * sqrt(var_tx(i,:)), +3 * sqrt(var_tx(i,:)), '#FF9848');
    fill([t; flipud(t)], [+3 * sqrt(var_tz); flipud(-3 * sqrt(var_tz))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_tz), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    hold on;
    plot(t, -3 * sqrt(var_tz), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [m]');
    title(['Translation Error in ' axis(3) ' Axis']);
    % legend;


    %% Plotting rotational error
    figure;

    % X Axis
    subplot(3, 1, 1);
    plot(t, rot_err(:, 1), 'DisplayName', 'Rotation Error', 'LineWidth',1);
    hold on;
    fill([t; flipud(t)], [+3 * sqrt(var_rx); flipud(-3 * sqrt(var_rx))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_rx), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    plot(t, -3 * sqrt(var_rx), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [rad]');
```

```
    title(['Rotation Error in ' axis(1) ' Axis']);
    % legend;

    % Y Axis
    subplot(3, 1, 2);
    plot(t, rot_err(:, 1), 'DisplayName', 'Rotation Error', 'LineWidth',1);
    hold on;
    fill([t; flipud(t)], [+3 * sqrt(var_ry); flipud(-3 * sqrt(var_ry))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_ry), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    plot(t, -3 * sqrt(var_ry), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [rad]');
    title(['Rotation Error in ' axis(2) ' Axis']);
    % legend;

    % Z Axis
    subplot(3, 1, 3);
    plot(t, rot_err(:, 1), 'DisplayName', 'Rotation Error', 'LineWidth',1);
    hold on;
    fill([t; flipud(t)], [+3 * sqrt(var_rz); flipud(-3 * sqrt(var_rz))], 'r', 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    plot(t, +3 * sqrt(var_rz), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope upper bound');
    plot(t, -3 * sqrt(var_rz), 'r-', 'LineWidth', 0.5, 'DisplayName', 'Uncertainty envelope lower bound');
    xlabel('Timestep');
    ylabel('Error [rad]');
    title(['Rotation Error in ' axis(3) ' Axis']);
    % legend;
end
```

Helper Functions:

vee.m

```
    function vec = vee(mat)
    if all(size(mat) == [4, 4]) % Case for SE(3)
        % Extract translational and rotational components
        v = mat(1:3, 4);
        w_mat = mat(1:3, 1:3);

        % Convert skew-symmetric part to a vector
        w = [w_mat(3, 2); w_mat(1, 3); w_mat(2, 1)];

        % Construct the 6x1 vector
        vec = [v; w];
    elseif all(size(mat) == [3, 3]) % Case for SO(3)
        % Convert skew-symmetric part to a vector
        vec = [mat(3, 2); mat(1, 3); mat(2, 1)];
    else
        error('Input matrix must be 3x3 or 4x4.');
    end
end
```

wedge.m

```
function mat = wedge(vec)
    if numel(vec) == 6  % Case for SE(3)
        % Extract translational and rotational components
```

```
        v = vec(1:3);  % Translational part
        w = vec(4:6);  % Rotational part

        % Create skew-symmetric matrix for w
        w_mat = [  0    -w(3)   w(2);
                  w(3)    0    -w(1);
                 -w(2)   w(1)    0 ];

        % Construct the 4x4 matrix
        mat = [w_mat, v; 0 0 0 0];
    elseif numel(vec) == 3  % Case for SO(3)
        % Input vector is a rotational part only
        w = vec;

        % Create skew-symmetric matrix for w
        mat = [  0    -w(3)   w(2);
                w(3)    0    -w(1);
               -w(2)   w(1)    0 ];
    else
        error('Input vector must have 3 or 6 elements.');
    end
end
```

plot_point.m

```
function plot_point(p_i_pj_i)

% p_i_pj_i = [2.7163; 2.4089; -0.0063; 1.0000];

% figure;
    hold on; grid on;
    plot3(p_i_pj_i(1), p_i_pj_i(2), p_i_pj_i(3), 'ro'); % Plot point as red circle
    hold on;
    grid on;
    axis equal; % Equal scaling
    xlabel('X-axis');
    ylabel('Y-axis');
    zlabel('Z-axis');
    view(3); % Isometric view
    title('3D Point Plot');

end
```

plot_T.m

```
function plot_T(T)
    % Ensure T is 4x4
    assert(all(size(T) == [4, 4]), 'Transformation matrix must be 4x4.');

    % Origin of the frame
    origin = T(1:3, 4);

    % Directions of the axes
    x_dir = T(1:3, 1);
    y_dir = T(1:3, 2);
    z_dir = T(1:3, 3);

    % Length of the axes arrows
    arrow_length = 0.1;

    hold on;
    grid on;
    axis equal;
```

```

    % Draw the axes
    quiver3(origin(1), origin(2), origin(3), arrow_length * x_dir(1), arrow_length *
        x_dir(2), arrow_length * x_dir(3), 'r', 'LineWidth', 2);
    quiver3(origin(1), origin(2), origin(3), arrow_length * y_dir(1), arrow_length *
        y_dir(2), arrow_length * y_dir(3), 'g', 'LineWidth', 2);
    quiver3(origin(1), origin(2), origin(3), arrow_length * z_dir(1), arrow_length *
        z_dir(2), arrow_length * z_dir(3), 'b', 'LineWidth', 2);

    xlabel('X-axis');
    ylabel('Y-axis');
    zlabel('Z-axis');
    view(3); % Isometric view
    title('Transformation Matrix Plot');
    hold off;
end
```

visualize_T_op.m

```
function visualize_T_op(T_op)

    k1 = 1215;
    k2 = 1714;
    K = size(T_op, 3);

    % get the transform from the inertia frame to the world frame
    for k = 1:K
        T_op(:,:,k) = inv(T_op(:,:,k));
    end


    rho_i_pj_i = [  1.61623639865093 2.11272672466273 -0.00738089473018551;
                    1.49900226256324 2.63254201551433 -0.00890945489038785;
                    1.50405785354131 3.19782554510961 -0.010249707757167;
                    2.06928872799326 3.17089614620574 -0.0106652108768971;
                    2.03108908849073 2.86780494312817 -0.0104548607925801;
                    1.8894364567693 2.54492856275833 -0.0104619386756643;
                    2.003889043779 2.05592623901308 -0.0103011134542387;
                    2.14626541109236 2.33471316651202 -0.010424517756221;
                    2.24885308692535 2.65488913580768 -0.00991525293125263;
                    2.39682628591312 2.86322869637284 -0.0107792334036271;
                    2.62792567642345 3.06172995061156 -0.00403525128637706;
                    2.92554969506537 2.9422116535639 -0.00835303226486714;
                    3.13092932442008 2.68821616288485 -0.00807754196541544;
                    2.74578526874736 2.68692081134543 -0.008404649388443 7;
                    2.44750390666386 2.44955069329397 -0.00942125234078448;
                    2.71627847013919 2.40894738671446 -0.00625649261914804;
                    2.37941412737242 2.2013713396879 -0.00761348165331085;
                    2.70311282041333 2.01308209755378 -0.00937463142134117;
                    3.22033852129903 2.03797339793336 -0.00863483007169972;
                    3.07833180801349 2.25481112395613 -0.00692203964571911]';


    % Extract the translation part of each transformation matrix
    positions = squeeze(T_op(1:3, 4, :));

    % Plot the trajectory of the robot
    figure; hold on; grid on;
    plot3(positions(1, :), positions(2, :), positions(3, :), 'r-'); % Red line

    hold on;
    % plot3(positions(1, k1), positions(2, k1), positions(3, k1), 'bx'); % starting
        point
```

```
    plot3(positions(1, 1), positions(2, 1), positions(3, 1), 'gx'); % starting point

    % Plot the features:
    scatter3(rho_i_pj_i(1,:), rho_i_pj_i(2,:), rho_i_pj_i(3,:), 'filled');

    % Set the view to isometric
    view(3); % Isometric view

    axis equal;

    % Label the axes
    xlabel('X-axis');
    ylabel('Y-axis');
    zlabel('Z-axis');

    % Title and legend
    title('Robot Trajectory');
    legend('Trajectory');

    hold off;
end
```