Institute for Aerospace Studies
UNIVERSITY OF TORONTO

AER 1513
State Estimation For Aerospace Vehicles
Assignment 1

Sandro Papais
Institute for Aerospace Studies
University of Toronto
sandro.papais@mail.utoronto.ca

In this assignment we use the batch linear-Gaussian algorithm to estimate the robot's one-dimensional position from odometry and laser measurements. The linear motion and observation models are

$$x_k = x_{k-1} + Tu_k + w_k = x_0 + T\sum_{i=1}^{k} u_i + \sum_{i=1}^{k} w_i, \tag{1a}$$

$$y_k = x_c - r_k = x_k + n_k, \tag{1b}$$

where x_k is the robot's position along the rail, u_k is the robot's speed derived from odometer, w_k is the process noise, T is the sampling period, r_k is the range to the cylinder derived from the laser rangefinder, x_c is the position of the cylinder's center along the rail, and n_k is the exteroceptive sensor noise. The measurement data is shown in Figure 1.

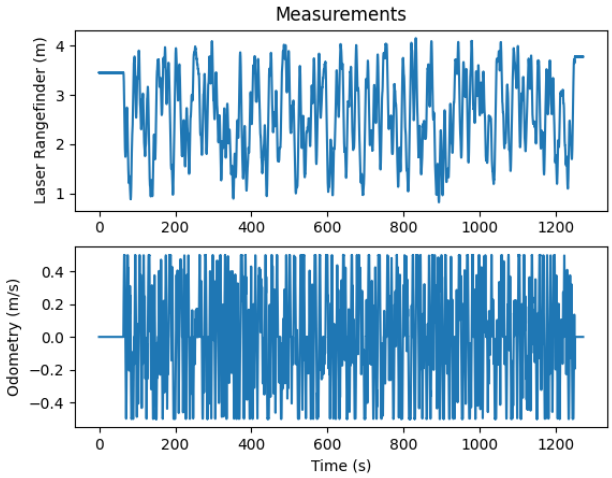

Figure 1: Robot measurement data provided over time.

1 Measurement Data Analysis

In order to perform linear-Gaussian estimation we must assume

$$w_k \sim \mathcal{N}(0, \sigma_q^2), \quad n_k \sim \mathcal{N}(0, \sigma_r^2). \tag{2}$$

Using the ground truth position information, x_k^{true}, from the Vicon motion capture system, we can rearrange the motion and observation model to solve for the process and measurement noise as

$$w_k = x_k - x_0 - T\sum_{i=1}^{k} u_i - \sum_{i=1}^{k-1} w_i, \quad n_k = x_c - r_k - x_k. \tag{3}$$

The computed motion and measurement model noise is shown in Figure 2. The mean of the measurement model and motion model noise was computed to be $1.005 \times 10^{-15}\,\mathrm{m}$ and $4.505 \times 10^{-5}\,\mathrm{m}$ respectively, which is quite small relative to the mean true position magnitude $1.780\,\mathrm{m}$. Therefore, the assumption of zero-mean measurement model noise is reasonable over small time scales. However, since the motion model noise has an additive effect this will lead to 0.45 m of error over 1000 seconds if left uncorrected.

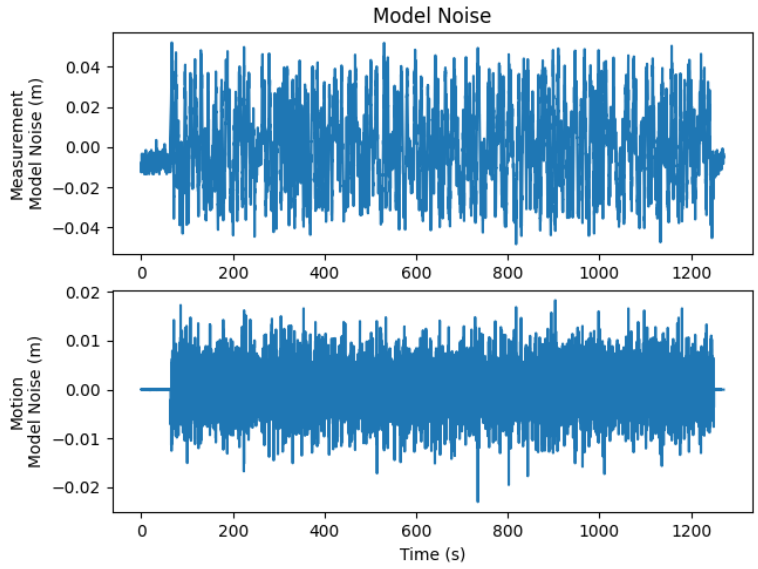

Figure 2: Computed motion and measurement model noise over time.

The assumption that the noise follows a Gaussian distribution can be qualitatively assessed by plotting the histogram of the noise against a best fit normal distribution, shown in Figure 3. We can also assess the fit using a Quantile-Quantile Plot, shown in Figure 4, which shows the fraction of points below the given value in the data against that of the best first normal distribution. Based on these plots, it is reasonable to claim that the measurement and modified process noise roughly follow a normal distribution. However, it is not a perfect fit and can be seen that the measurement model noise has a significant deviation from normal at the edges of the distribution where there is less data. We could quantitatively assess the goodness of fit using a statistical test, such as the D'Agostino and Pearson test, but this was assumed to be out of the scope of this question.

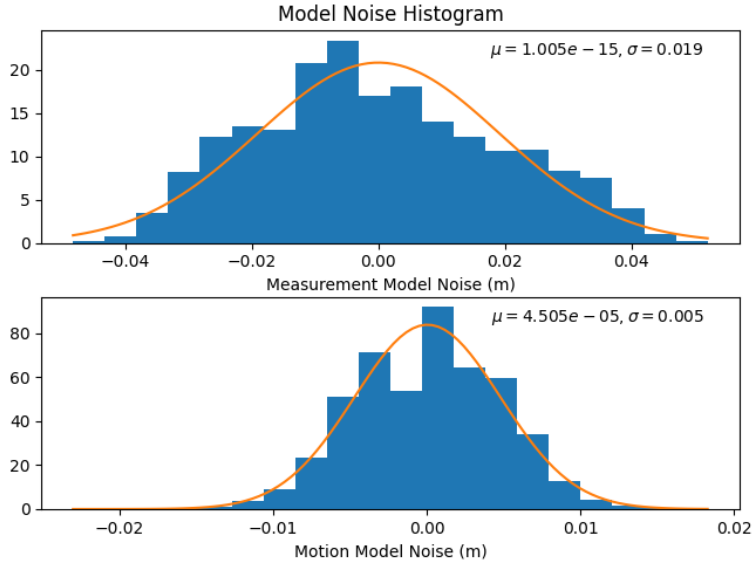

Figure 3: Histogram plot of noise compared to a best fit normal distribution.

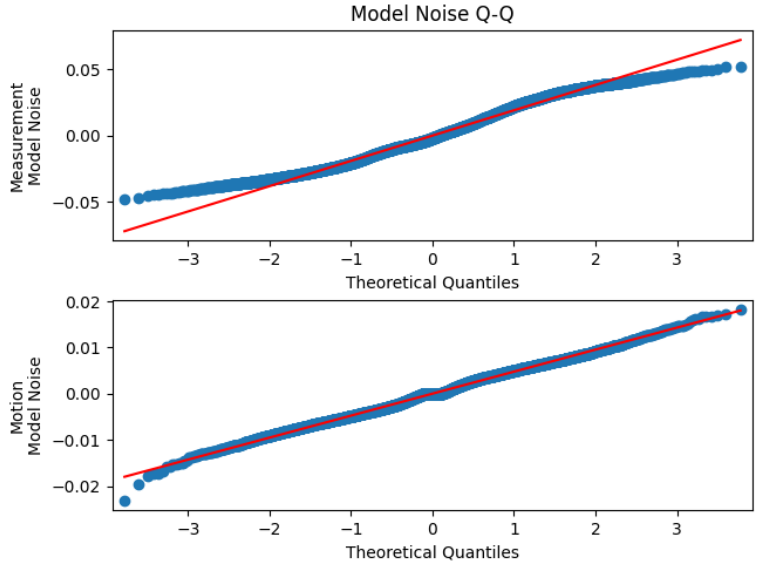

Figure 4: Quantile-Quantile plot of noise compared to a best fit normal distribution

The variances of the process model and observation model noise are computed to be

$$\sigma_q^2 = \frac{1}{K}\sum_{k=0}^{K}(w_k - \bar{w}_k)^2 = 0.000\,02\,\mathrm{m}^2, \quad \sigma_r^2 = \frac{1}{K}\sum_{k=0}^{K}(n_k - \bar{w}_k)^2 = 0.000\,37\,\mathrm{m}^2. \tag{4}$$

However, these values were later found to provide an inconsistent variance estimate in the smoother due to overconfidence in the process model. For a more consistent smoother uncertainty, the variance of the process model noise was inflated to $\sigma_q^2 = 0.002\,\mathrm{m}^2$ as determined by trial and error.

2 Batch Estimation Objective

Our goal is to find the most likely values for the state, meaning the state values that maximize the posterior probability density function given all our measurements of the state. In other words we we wish to find the state estimate as

$$\hat{x}_{0:K} = \arg\max_{x_{0:K}} p(x_{0:K}|u_{1:K}, r_{0:K}), \tag{5}$$

We can apply Baye's rule, simplify, and factor such that

$$\begin{aligned}
\hat{x}_{0:K} &= \arg\max_{x_{0:K}} \frac{p(r_{0:K}|x_{0:K}, u_{1:K})p(x_{0:K}|u_{1:K})}{p(r_{0:K}|u_{1:K})}, &(6)\\
&= \arg\max_{x_{0:K}} p(r_{0:K}|x_{0:K})p(x_{0:K}|u_{1:K}), &(7)\\
&= \arg\max_{x_{0:K}} \prod_{k=0}^{K} p(r_k|x_k)p(x_0|\check{x}_0) \prod_{k=1}^{K} p(x_k|x_{k-1}, u_k). &(8)
\end{aligned}$$

The component Gaussian probability density functions are given by

$$\begin{aligned}
p(r_k|x_k) &= \frac{1}{\sqrt{2\pi\sigma_r^2}} \exp\left(-\frac{(x_c - r_k - x_k)^2}{2\sigma_r^2}\right), &(9)\\
p(x_0|\check{x}_0) &= \frac{1}{\sqrt{2\pi\check{P}_0}} \exp\left(-\frac{(x_0 - \check{x}_0)^2}{2\check{P}_0}\right), &(10)\\
p(x_k|x_{k-1}, u_k) &= \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(-\frac{(x_k - x_{-1} - Tu_k)^2}{2\sigma_q^2}\right). &(11)
\end{aligned}$$

By substituting the probability density functions into Equation 8 and taking the negative logarithm we can pose an equivalent minimization problem of the form

$$\hat{x}_{0:K} = \arg\min_{x_{0:K}} J(x_{0:K}, u_{0:K}, r_{0:K}), \tag{12}$$

where

$$J(x_{0:K}, u_{1:K}, r_{0:K}) = \frac{(x_0 - \check{x}_0)^2}{2\check{P}_0} + \frac{1}{2\sigma_q^2}\sum_{k=1}^{K}(x_k - x_{k-1} - Tu_k)^2 + \frac{1}{2\sigma_r^2}\sum_{k=0}^{K}(x_c - r_k - x_k)^2. \tag{13}$$

We can equivalently represent the objective function in matrix form as

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{z} - \mathbf{H}\mathbf{x})^T\mathbf{W}^{-1}(\mathbf{z} - \mathbf{H}\mathbf{x}), \tag{14}$$

where

$$\mathbf{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_K \end{bmatrix}_{(K+1)\times 1}, \mathbf{z} = \begin{bmatrix} \check{x}_0 \\ Tu_1 \\ \vdots \\ Tu_K \\ x_c - r_0 \\ \vdots \\ x_c - r_K \end{bmatrix}_{2(K+1)\times 1}, \mathbf{H} = \begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \\ 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}_{2(K+1)\times(K+1)},$$

$$\mathbf{W} = \begin{bmatrix} \check{P}_0 & & & & & & \\ & \sigma_q^2 & & & & & \\ & & \ddots & & & & \\ & & & \sigma_q^2 & & & \\ & & & & \sigma_r^2 & & \\ & & & & & \ddots & \\ & & & & & & \sigma_r^2 \end{bmatrix}_{2(K+1)\times 2(K+1)}$$

3 Least Squares Position Estimate

We will now solve the weighted least squares optimization problem with the objective function given in Equation 13. We can use calculus to find minimum of the objective function as the derivative of the objective function with respect to the state variable, given by

$$\frac{\partial J(x_{0:K}, u_{0:K}, r_{0:K})}{\partial x_{0:K}} = \begin{bmatrix} \frac{\partial J(x_{0:K}, u_{0:K}, r_{0:K})}{\partial x_0} \\ \frac{\partial J(x_{0:K}, u_{0:K}, r_{0:K})}{\partial x_1} \\ \vdots \\ \frac{\partial J(x_{0:K}, u_{0:K}, r_{0:K})}{\partial x_K} \end{bmatrix} = \begin{bmatrix} \frac{x_0 - \check{x}_0}{\check{P}_0} + \frac{x_0 - x_1 + Tu_1}{\sigma_q^2} + \frac{x_0 - x_c + r_0}{\sigma_r^2} \\ \frac{(-x_0 + x_1 - Tu_1) + (x_1 - x_2 + Tu_2)}{\sigma_q^2} + \frac{x_1 - x_c + r_1}{\sigma_r^2} \\ \vdots \\ \frac{-x_{K-1} + x_K - Tu_K}{\sigma_q^2} + \frac{x_K - x_c + r_K}{\sigma_r^2} \end{bmatrix} = 0. \quad (15)$$

We can rearrange this expression as

$$\begin{bmatrix} \frac{x_0}{\check{P}_0} + \frac{x_0 - x_1}{\sigma_q^2} + \frac{x_0}{\sigma_r^2} \\ \frac{-x_0 + 2x_1 - x_2}{\sigma_q^2} + \frac{x_1}{\sigma_r^2} \\ \vdots \\ \frac{-x_{K-1} + x_K}{\sigma_q^2} + \frac{x_K}{\sigma_r^2} \end{bmatrix} = \begin{bmatrix} \frac{\check{x}_0}{\check{P}_0} - \frac{Tu_1}{\sigma_q^2} + \frac{x_c - r_0}{\sigma_r^2} \\ \frac{Tu_1 - Tu_2}{\sigma_q^2} + \frac{x_c - r_1}{\sigma_r^2} \\ \vdots \\ \frac{Tu_K}{\sigma_q^2} + \frac{x_c - r_K}{\sigma_r^2} \end{bmatrix}, \quad (16)$$

$$\begin{bmatrix} \frac{1}{\check{P}_0}+\frac{1}{\sigma_q^2}+\frac{1}{\sigma_r^2} & -\frac{1}{\sigma_q^2} & & & & \\ -\frac{1}{\sigma_q^2} & \frac{2}{\sigma_q^2}+\frac{1}{\sigma_r^2} & -\frac{1}{\sigma_q^2} & & & \\ & \ddots & \ddots & \ddots & & \\ & & -\frac{1}{\sigma_q^2} & \frac{2}{\sigma_q^2}+\frac{1}{\sigma_r^2} & -\frac{1}{\sigma_q^2} & \\ & & & -\frac{1}{\sigma_q^2} & \frac{1}{\sigma_q^2}+\frac{1}{\sigma_r^2} & \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{K-1} \\ x_K \end{bmatrix} = \begin{bmatrix} \frac{\check{x}_0}{\check{P}_0}-\frac{Tu_1}{\sigma_q^2}+\frac{x_c-r_0}{\sigma_r^2} \\ \frac{Tu_1-Tu_2}{\sigma_q^2}+\frac{x_c-r_1}{\sigma_r^2} \\ \vdots \\ \frac{Tu_{K-1}-Tu_K}{\sigma_q^2}+\frac{x_c-r_{K-1}}{\sigma_r^2} \\ \frac{Tu_K}{\sigma_q^2}+\frac{x_c-r_K}{\sigma_r^2} \end{bmatrix}. \tag{17}$$

Now we have a linear system of equations in the form of $\mathbf{Ax} = \mathbf{b}$. This system of equations can be shown to be equivalent to $(\mathbf{H}^T\mathbf{W}^{-1}\mathbf{H})\mathbf{x} = \mathbf{H}^T\mathbf{W}^{-1}\mathbf{z}$. We can solve this system of equations for the optimal position estimate, $\hat{x}_{0:K}$, as

$$\hat{\mathbf{x}} = \begin{bmatrix} \frac{1}{\check{P}_0}+\frac{1}{\sigma_q^2}+\frac{1}{\sigma_r^2} & -\frac{1}{\sigma_q^2} & & & & \\ -\frac{1}{\sigma_q^2} & \frac{2}{\sigma_q^2}+\frac{1}{\sigma_r^2} & -\frac{1}{\sigma_q^2} & & & \\ & \ddots & \ddots & \ddots & & \\ & & -\frac{1}{\sigma_q^2} & \frac{2}{\sigma_q^2}+\frac{1}{\sigma_r^2} & -\frac{1}{\sigma_q^2} & \\ & & & -\frac{1}{\sigma_q^2} & \frac{1}{\sigma_q^2}+\frac{1}{\sigma_r^2} & \end{bmatrix}^{-1} \begin{bmatrix} \frac{\check{x}_0}{\check{P}_0}-\frac{Tu_1}{\sigma_q^2}+\frac{x_c-r_0}{\sigma_r^2} \\ \frac{Tu_1-Tu_2}{\sigma_q^2}+\frac{x_c-r_1}{\sigma_r^2} \\ \vdots \\ \frac{Tu_{K-1}-Tu_K}{\sigma_q^2}+\frac{x_c-r_{K-1}}{\sigma_r^2} \\ \frac{Tu_K}{\sigma_q^2}+\frac{x_c-r_K}{\sigma_r^2} \end{bmatrix}. \tag{18}$$

4 Recursive Smoothing Method

As shown in Equation 17, in order to solve the system of equations $\mathbf{Ax} = \mathbf{b}$ we require the inverse of a large matrix, $\mathbf{A}_{(K+1)\times(K+1)}$, where $K = 12\,708$. The matrix has a symmetric tridiagonal sparsity pattern with the form

$$\mathbf{A} = \begin{bmatrix} \ddots & \ddots & & & & \\ \ddots & \ddots & \ddots & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \ddots \\ & & & & \ddots & \ddots \end{bmatrix}_{(K+1)\times(K+1)}. \tag{19}$$

This pattern allows us to solve for the inverse more efficiently than using traditional algorithms, such as Gaussian elimination, which typically run in $\mathcal{O}(n^3)$ time. Given the tridiagonal sparsity pattern, we can invert the matrix using the tridiagonal matrix algorithm (also known as the forward block tridiagonal or Thomas algorithm), Rauch-Tung-Striebel (RTS) Smoother, or Cholesky Smoother. All three of these algorithms can be shown to be equivalent. The algorithms use a single forward pass followed by a single backward substitution pass and run in $\mathcal{O}(n)$ time. Once

$\mathbf{A}^{-1}$ is found we can compute the most likely state estimate and confidence in our solution as $\hat{\mathbf{x}} = \mathbf{A}^{-1}\mathbf{b}, \hat{\mathbf{P}} = \mathbf{A}^{-1}$.

The algorithm chosen is the RTS Smoother, the psuedocode is provided in Algorithm 1. The first predicted state estimate is assigned the first measurement and the first predicted state variance is assigned the measurement noise variance. For subsequent steps during the forward pass the predicted state variance and estimate is computed for the current step based on the previous state variance, estimate, and process model (line 7 and 8). If our iterator, i, is divisible by the range measurement update interval, δ_r, then we compute the Kalman gain (line 11), otherwise it is set to zero. When range measurements are available, we use the Kalman gain to solve for the corrected state variance and estimate (line 15 and 16) using the optimal weighting of the predicted state and measurement. In the backwards pass we apply the backward substitution smoothing equation to solve for the optimal state estimate and variance (line 23 and 24).

Algorithm 1 Implementation of the Rauch-Tung-Striebel Smoother algorithm

1: $i \leftarrow 0$
2: **while** $i \leq K$ **do** ▷ Forward pass loop
3: **if** $i == 0$ **then**
4: $\check{x}_{0,f} \leftarrow y_0$
5: $\check{P}_{0,f} \leftarrow \sigma_r^2$
6: **else**
7: $\check{P}_{k,f} = \hat{P}_{k-1,f} + \sigma_q^2$
8: $\check{x}_{k,f} = \hat{x}_{k-1,f} + Tu_k$
9: **end if**
10: **if** $(i \mod \delta_r) == 0$ **then**
11: $K_k = \check{P}_{k,f}/(\check{P}_{k,f} + \sigma_r^2)$
12: **else**
13: $K_k = 0$
14: **end if**
15: $\hat{P}_{k,f} = (1 - K_k)\check{P}_{k,f}$
16: $\hat{x}_{k,f} = \check{x}_{k,f} + K_k(x_c - r_k - \check{x}_{k,f})$
17: $i \leftarrow i + 1$
18: **end while**
19: $\hat{P}_K \leftarrow \hat{P}_{K,f}$
20: $\hat{x}_K \leftarrow \hat{x}_{K,f}$
21: $i \leftarrow K$
22: **while** $i \geq 0$ **do** ▷ Backward pass loop
23: $\hat{P}_{k-1} = \hat{P}_{k-1,f} + (\hat{P}_k - \check{P}_{k,f})(\hat{P}_{k-1,f}/\check{P}_{k,f})^2$
24: $\hat{x}_{k-1} = \hat{x}_{k-1,f} + (\hat{x}_k - \check{x}_{k,f})\hat{P}_{k-1,f}/\check{P}_{k,f}$
25: $i \leftarrow i - 1$
26: **end while**

5 RTS Smoother Results

A Python script was created to solve for the robots position estimate and uncertainty at all K time steps. Four cases were considered with different range measurement update step sizes, $\delta_r = 1, 10, 100, 100$. The derivation in Section 1 to Section 3 assumed the case of $\delta_r = 1$, however the extension to the other cases is similar. For the general case motion and measurement model are

$$x_k = x_{k-1} + Tu_k + w_k, \quad k = 1, \ldots, K \tag{20a}$$
$$y_k = x_c - r_k = x_k + n_k, \quad k = \delta_r, 2\delta_r, \ldots, K. \tag{20b}$$

This gives us a new objective function

$$J(x_{0:K}, u_{1:K}, r_{\delta_r:\delta_r:K}) = \frac{(x_0 - \check{x}_0)^2}{2\check{P}_0} + \frac{1}{2\sigma_q^2}\sum_{k=1}^{K}(x_k - x_{k-1} - Tu_k)^2 + \frac{1}{2\sigma_r^2}\sum_{k=1}^{K/\delta_r}(x_c - r_{k\cdot\delta_r} - x_{k\cdot\delta_r})^2. \tag{21}$$

Our final system of equations will be the same as Equation 17, except that all terms with $\frac{1}{\sigma_r^2}$ will be removed. The process presented in Algorithm 1 is valid for all values of δ_r and was used to write the Python code found in Appendix A.

The results for the estimation error and uncertainty for the different values of δ_r are shown in Figure 5 to Figure 8. It should be noted that the out of normal distribution peak near zero error on all the histograms is a result of the initial and final portion of the trajectory where the robot is not moving. Figure 5 shows consistent estimation for $\delta_r = 1$ is achieved since the position estimation error $3\sigma_{x_k - \hat{x}_k} = 0.050$ m, which closely matches the posterior uncertainty provided by the RTS algorithm during the trajectory, $3\hat{\sigma}_{x_k} = 0.057$ m.

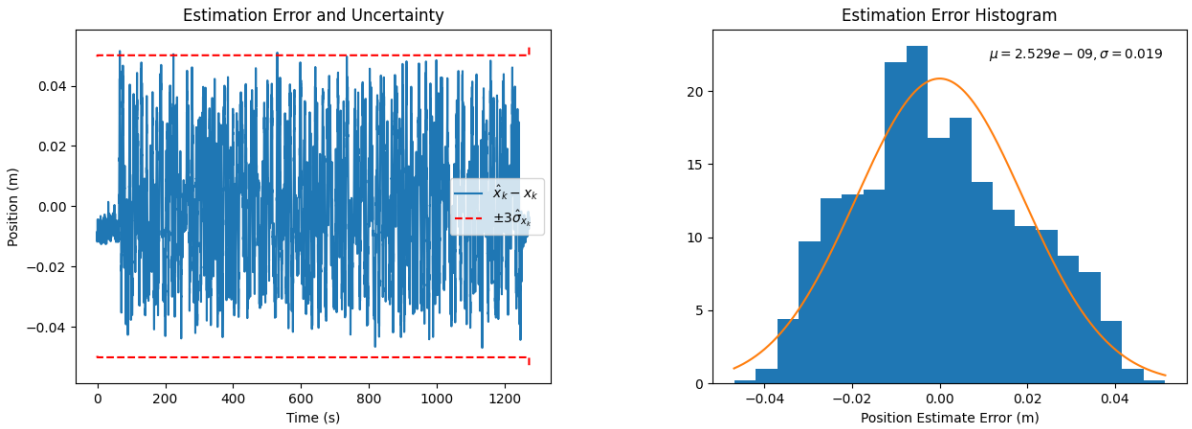

Figure 5: Position estimation error for rangefinder measurement step size of $\delta_r = 1$.

Figure 6 shows that the estimation error is similar for $\delta_r = 10$ as it was for $\delta_r = 1$, however the the posterior variance is no longer constant as is now fluctuating with a repeating period of 1 second, equal to the laser measurement time step. The value of the position estimation error $3\sigma_{x_k - \hat{x}_k} = 0.061$ m, and the posterior uncertainty is now $3\hat{\sigma}_{x_k} \in [0.056, 0.216\text{m}]$.

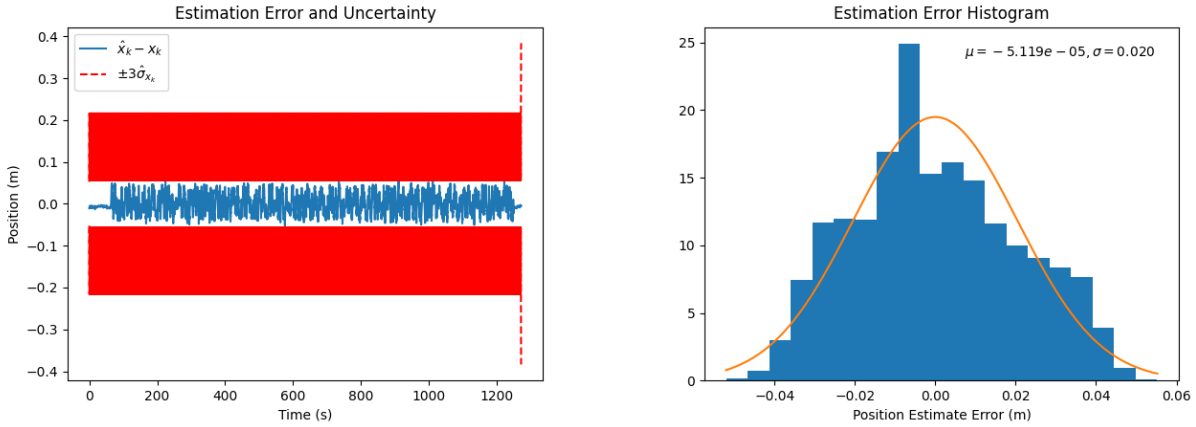

Figure 6: Position estimation error for rangefinder measurement step size of $\delta_r = 10$.

Figure 7 shows that the scenario for $\delta_r = 100$ has an increase in position estimation error $3\sigma_{x_k - \hat{x}_k} = 0.149$ m. It also shows an increase in the maximum value of the posterior uncertainty $3\hat{\sigma}_{x_k} \in [0.057, 0.672\text{m}]$.

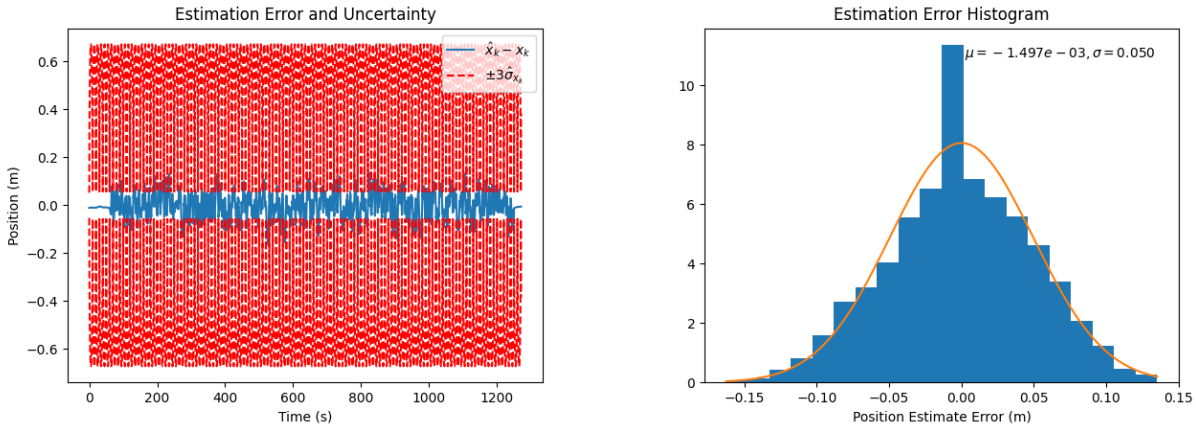

Figure 7: Position estimation error for rangefinder measurement step size of $\delta_r = 100$.

Figure 8 shows that the scenario for $\delta_r = 1000$ has an increase in position estimation error to $3\sigma_{x_k - \hat{x}_k} = 0.240$ m. It also shows an increase in the maximum value of the posterior $3\hat{\sigma}_{x_k} \in [0.057, 2.12\text{m}]$. The period of the posterior uncertainty fluctuation is now 100 seconds, equal to the range update time step, as expected.

A summary of the performance of the estimator for all four scenarios is presented in Table 1. Over each the trajectories we see that at each of the range measurement updates our uncertainty in our estimate is the smallest, while at the mid-point between measurement updates our uncertainty is the largest. This is to be expected since the range measurement provides an exteroceptive update which can correct any drift accumulated by the interoceptive odemetry measurements and the process model. In all scenarios a filter estimation error is consistent with the filter posterior variance was achieved, meaning that $x_k - \hat{x}_k$ rarely exceeds $3\hat{\sigma}_{x_k}$. As we increase the value δ_r we see increases

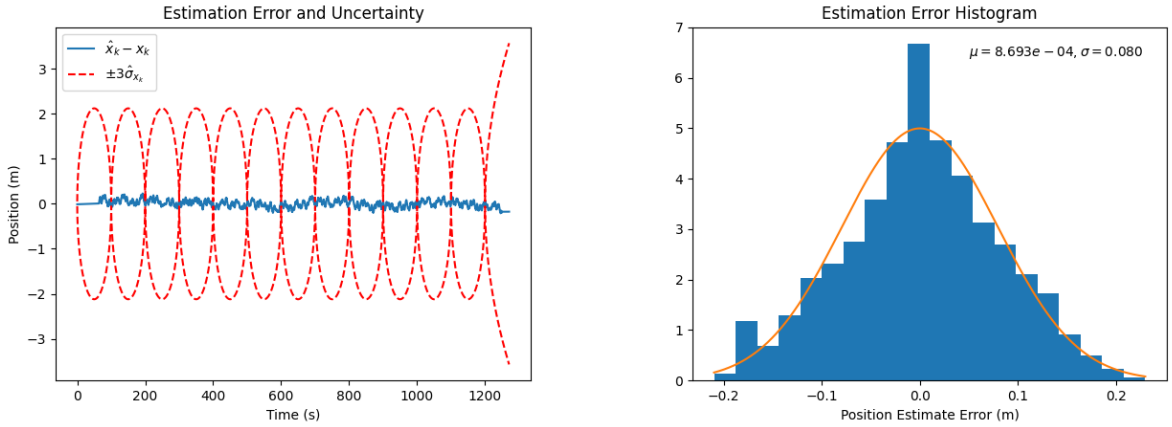

Figure 8: Position estimation error for rangefinder measurement step size of $\delta_r = 1000$.

in $3\sigma_{x_k - \hat{x}_k}$ and the upper bound of $3\hat{\sigma}_{x_k}$ over the trajectory, however the lower bound of $3\hat{\sigma}_{x_k}$ does not change. However, these results required the manual tuning of the process noise, σ_q^2, since the initial value found based on analysis of the odometery data led to an overconfident estimate. Since the real world process and model noise do not perfectly follow a zero-mean normal distribution it is expected that tuning is required for consistent results.

δ_r	$3\sigma_{x_k - \hat{x}_k}$	$3\hat{\sigma}_{x_k}$
1	0.057 m	0.050 m
10	0.061 m	[0.056, 0.216 m]
100	0.149 m	[0.057, 0.672 m]
1000	0.240 m	[0.057, 2.12 m]

Table 1: Summary of estimation performance.

A Analysis and RTS Smoother Implementation

```
from os.path import dirname, join
from os.path import dirname, join
import scipy.io as sio
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
from statsmodels.graphics.gofplots import qqplot


def read_data_mat():
    # Load .mat file
```

```
    data_dir = join(dirname(__file__), 'data')
    mat_file_name = join(data_dir, 'dataset1.mat')
    mat_contents = sio.loadmat(mat_file_name)
    return mat_contents


def compute_measurement_stats(mat_contents):
    # Unpack data and flatten to 1D
    range_meas = mat_contents['r'].flatten()
    pos_true = mat_contents['x_true'].flatten()
    pos_true = pos_true.flatten()
    t = mat_contents['t'].flatten()
    vel_meas = mat_contents['v'].flatten()
    pos_cyl = mat_contents['l'].flatten()
    # range_meas_var = mat_contents['r_var'].flatten() # unused
    # vel_meas_var = mat_contents['v_var'].flatten() # unused
    samples_count = pos_true.size
    t_step = t[1]

    # Measurement model and motion model noise
    pos_meas = pos_cyl - range_meas
    pos_meas_err = pos_meas - pos_true # measurement model noise
    pos_prop_err = np.zeros(samples_count) # motion model noise
    for i in range(1, samples_count):
        pos_prop_err[i] = pos_true[i] - pos_true[0] - t_step *
            np.sum(vel_meas[:i + 1]) - np.sum(pos_prop_err[:i])

    # Fit data and compute statistics
    pos_true_mean = np.mean(pos_true)
    meas_err_mean, meas_err_std = norm.fit(pos_meas_err)
    prop_err_mean, prop_err_std = norm.fit(pos_prop_err)
    q_proc_noise_cov = prop_err_std ** 2
    r_meas_noise_cov = meas_err_std ** 2
    print('*** PREPROCESS ***\n'
          f'Position mean = {pos_true_mean:1.3f} m\n'
          f'Measurement noise: mean={meas_err_mean:1.3e} m,
             std={meas_err_std:1.3f} m,'
          f' 3-sigma={3 * meas_err_std:1.3f}m\n'
          f'Propagation noise: mean={prop_err_mean:1.3e} m,
             std={prop_err_std:1.3f} m, '
          f'3-sigma={3 * prop_err_std:1.3f}m')

    # PLOT01: Measurements
    fig1, axs1 = plt.subplots(2, 1)
    axs1[0].plot(t, range_meas)
    axs1[0].set_ylabel('Laser Rangefinder (m)')
    axs1[0].set_title("Measurements")
```

```
axs1[1].plot(t, vel_meas)
axs1[1].set_xlabel('Time (s)')
axs1[1].set_ylabel('Odometry (m/s)')
fig1.savefig('out/meas_vs_t.png')

# PLOT02: Model Noise
fig2, axs2 = plt.subplots(2, 1)
axs2[0].plot(t, pos_meas_err)
axs2[0].set_ylabel('Measurement\nModel Noise (m)')
axs2[0].set_title("Model Noise")
axs2[1].plot(t, pos_prop_err)
axs2[1].set_xlabel('Time (s)')
axs2[1].set_ylabel('Motion\nModel Noise (m)')
fig2.tight_layout(pad=0.2)
fig2.savefig('out/model_noise_vs_t.png')

# PLOT03: Model Noise Histogram
fig3, axs3 = plt.subplots(2, 1)
n_pts_plot = 100
bins = axs3[0].hist(pos_meas_err, 20, density=1)[1]
meas_err_fit_bins = np.linspace(bins[0], bins[-1], n_pts_plot)
meas_err_fit = norm.pdf(meas_err_fit_bins, meas_err_mean,
   meas_err_std)
axs3[0].plot(meas_err_fit_bins, meas_err_fit)
axs3[0].set_xlabel('Measurement Model Noise (m)')
axs3[0].text(0.95, 0.95, fr'$\mu={meas_err_mean:1.3e},
   \sigma={meas_err_std:1.3f}$',
          horizontalalignment='right', verticalalignment='top',
             transform=axs3[0].transAxes)
axs3[0].set_title("Model Noise Histogram")
bins = axs3[1].hist(pos_prop_err, 20, density=1)[1]
prop_err_fit_bins = np.linspace(bins[0], bins[-1], n_pts_plot)
prop_err_fit = norm.pdf(prop_err_fit_bins, prop_err_mean,
   prop_err_std)
axs3[1].plot(prop_err_fit_bins, prop_err_fit)
axs3[1].set_xlabel('Motion Model Noise (m)')
axs3[1].text(0.95, 0.95, fr'$\mu={prop_err_mean:1.3e},
   \sigma={prop_err_std:1.3f}$',
          horizontalalignment='right', verticalalignment='top',
             transform=axs3[1].transAxes)
fig3.tight_layout(pad=0.2)
fig3.savefig('out/hist_model_noise.png')

# PLOT04: Model Noise Q-Q
fig4 = plt.figure()
ax4 = fig4.add_subplot(2, 1, 1)
qqplot(pos_meas_err, ax=ax4, line='s')
```

```
    ax4.set_ylabel('Measurement\nModel Noise')
    ax4.set_title("Model Noise Q-Q")
    ax4 = fig4.add_subplot(2, 1, 2)
    qqplot(pos_prop_err, ax=ax4, line='s')
    ax4.set_ylabel('Motion\nModel Noise')
    fig4.tight_layout(pad=0.2)
    fig4.savefig('out/qq_model_noise.png')

    return pos_meas, vel_meas, pos_true, t, samples_count, t_step,
        q_proc_noise_cov, r_meas_noise_cov


def rts_smoother(pos_meas, vel_meas, pos_true, t, samples_count,
    t_step, q_proc_noise_cov, r_meas_noise_cov,
              flg_debug_prop_only, flg_debug_fwd_only,
                 update_interval_indices, a_trans_mat, c_obs_mat):
    # RTS smoother initialization
    pos_est_ini = pos_meas[0]
    pos_var_ini = q_proc_noise_cov
    pos_est_pred_fwd = np.zeros(samples_count)
    pos_est_corr_fwd = np.zeros(samples_count)
    pos_est_corr = np.zeros(samples_count)
    pos_var_pred_fwd = np.zeros(samples_count)
    pos_var_corr_fwd = np.zeros(samples_count)
    pos_var_corr = np.zeros(samples_count)

    # RTS smoother forward pass
    for i in range(0, samples_count):
        if i == 0:
            pos_est_pred_fwd[0] = pos_est_ini
            pos_var_pred_fwd[0] = pos_var_ini
        else:
            pos_var_pred_fwd[i] = a_trans_mat * pos_var_corr_fwd[i - 1] *
                a_trans_mat + q_proc_noise_cov
            pos_est_pred_fwd[i] = a_trans_mat * pos_est_corr_fwd[i - 1] +
                t_step * vel_meas[i]
        if flg_debug_prop_only == 1:
            kalman_gain = 0
        elif (i % update_interval_indices) == 0:
            kalman_gain = pos_var_pred_fwd[i] * c_obs_mat / (
                  c_obs_mat * pos_var_pred_fwd[i] * c_obs_mat +
                     r_meas_noise_cov)
        else:
            kalman_gain = 0
        pos_var_corr_fwd[i] = (1 - kalman_gain * c_obs_mat) *
            pos_var_pred_fwd[i]
```

```
        pos_est_corr_fwd[i] = pos_est_pred_fwd[i] + kalman_gain *
            (pos_meas[i] - c_obs_mat * pos_est_pred_fwd[i])

    # RTS smoother backward pass
    if flg_debug_fwd_only == 1:
        pos_var_corr = pos_var_corr_fwd
        pos_est_corr = pos_est_corr_fwd
    else:
        pos_est_corr[-1] = pos_est_corr_fwd[-1]
        pos_var_corr[-1] = pos_var_corr_fwd[-1]
        for i in range(samples_count - 1, 0, -1):
            pos_est_corr[i - 1] = \
                pos_est_corr_fwd[i - 1] + (pos_var_corr_fwd[i - 1] *
                    a_trans_mat / pos_var_pred_fwd[i]) * \
                (pos_est_corr[i] - pos_est_pred_fwd[i])
            pos_var_corr[i - 1] = \
                pos_var_corr_fwd[i - 1] + (pos_var_corr_fwd[i - 1] *
                    a_trans_mat / pos_var_pred_fwd[i]) * \
                (pos_var_corr[i] - pos_var_pred_fwd[i]) *
                    (pos_var_corr_fwd[i - 1] * a_trans_mat /
                    pos_var_pred_fwd[i])

    # Post process results
    pos_est_err = pos_est_corr - pos_true
    pos_est_err_mean = np.mean(pos_est_err)
    pos_est_err_mod_avg = np.mean(np.abs(pos_est_err))
    pos_est_rmse = np.sqrt(np.sum(pos_est_err ** 2) / samples_count)
    pos_est_err_std = np.std(pos_est_err)
    pos_3sigma = 3 * np.sqrt(pos_var_corr)
    print('*** POST-PROCESS ****\n'
          f'Process model variance = {q_proc_noise_cov:1.5f}\n'
          f'Measurement model variance = {r_meas_noise_cov:1.5f}\n'
          f'Average error magnitude = {pos_est_err_mod_avg:1.3f} m\n'
          f'Root mean square error = {pos_est_rmse:1.3f} m\n'
          f'3 sigma error = {3 * pos_est_err_std:1.3f} m')

    # PLOT05: Estimation Error and Uncertainty
    fig5, ax5 = plt.subplots()
    ax5.plot(t, pos_est_err, label=r'$\hat x_k - x_k$')
    ax5.plot(t, pos_3sigma, 'r--', label=r'$\pm3\hat\sigma_{x_k}$')
    ax5.plot(t, -pos_3sigma, 'r--')
    ax5.set_xlabel('Time (s)')
    ax5.set_ylabel('Position (m)')
    ax5.set_title("Estimation Error and Uncertainty")
    ax5.legend()
    fig5.savefig(f'out/est_err_and_3std_{update_interval_indices}steps.png')
```

```
    # PLOT06: Estimation Error Histogram
    fig6, ax6 = plt.subplots()
    n_pts_plot = 1000
    bins = ax6.hist(pos_est_err, 20, density=1)[1]
    pos_est_err_fit_bins = np.linspace(bins[0], bins[-1], n_pts_plot)
    pos_est_err_fit = norm.pdf(pos_est_err_fit_bins, 0, pos_est_err_std)
    ax6.plot(pos_est_err_fit_bins, pos_est_err_fit)
    ax6.set_xlabel('Position Estimate Error (m)')
    ax6.set_title("Estimation Error Histogram")
    plt.text(0.95, 0.95, fr'$\mu={pos_est_err_mean:1.3e},
       \sigma={pos_est_err_std:1.3f}$',
           horizontalalignment='right', verticalalignment='top',
              transform=ax6.transAxes)
    fig6.savefig(f'out/est_err_hist_{update_interval_indices}steps.png')

    # PLOT07: Estimation and Uncertainty
    fig7, ax7 = plt.subplots()
    ax7.plot(t, pos_est_corr, label=r'$\hat{x_k}$')
    ax7.plot(t, pos_true, label=r'$x_k$')
    ax7.plot(t, pos_est_corr + pos_3sigma, 'r--',
       label=r'$\hat{x_k}\pm3\hat\sigma_{x_k}$')
    ax7.plot(t, pos_est_corr - pos_3sigma, 'r--')
    ax7.set_xlabel('Time (s)')
    ax7.set_ylabel('Position (m)')
    ax7.set_title("Estimation and Uncertainty")
    ax7.legend()
    fig7.savefig(f'out/est_and_tru_{update_interval_indices}steps.png')


def main():
    # Define flags
    flg_plot_show = 0 # 0, 1
    flg_debug_prop_only = 0 # 0, 1
    flg_debug_fwd_only = 0 # 0, 1
    # Define RTS smoother parameters
    update_interval_indices = 1000 # 1, 10, 100, 1000
    a_trans_mat = 1
    c_obs_mat = 1
    # Load data file
    mat_contents = read_data_mat()
    # Q1: Preprocess data for statistics
    pos_meas, vel_meas, pos_true, t, samples_count, t_step,
       q_proc_noise_cov, r_meas_noise_cov = \
       compute_measurement_stats(mat_contents)
    # Q5: Call RTS smoother on data
    q_proc_noise_cov = 0.002 # instead of using prop_err_std ** 2, we
       inflate the process noise
```

```
    rts_smoother(pos_meas, vel_meas, pos_true, t, samples_count,
        t_step, q_proc_noise_cov, r_meas_noise_cov,
                flg_debug_prop_only, flg_debug_fwd_only,
                    update_interval_indices, a_trans_mat, c_obs_mat)
    # Plotting
    if flg_plot_show == 1:
        plt.show()


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