Course Project Report

Introduction to Statistical Signal Processing

Varun Shastry (2023112005)

May 8, 2025

Contents

Problem 1: De-noising the ECG signal with reference

The problem statement states that we are given two ECG signals corrupted with noise, termed Signal1 and Signal2. We're also provided the corresponding secondary source noise references, Noise1 and Noise2. We work under the assumption that the desired signals, which are the original ECG signals (not directly available) are noise-free and stationary in nature.

We have to implement various adaptive filters of order 4, and perform noise removal.

(a) Implementing the Adaptive Filters

I've implemented the Steepest Descent, the Least Mean Squares (LMS) and the Recursive Least Squares filters, all of order 4.

The problem we're trying to solve here falls under the case of **noise removal provided a reference**. The general block diagram for such a case is as in Figure 1.1 below.

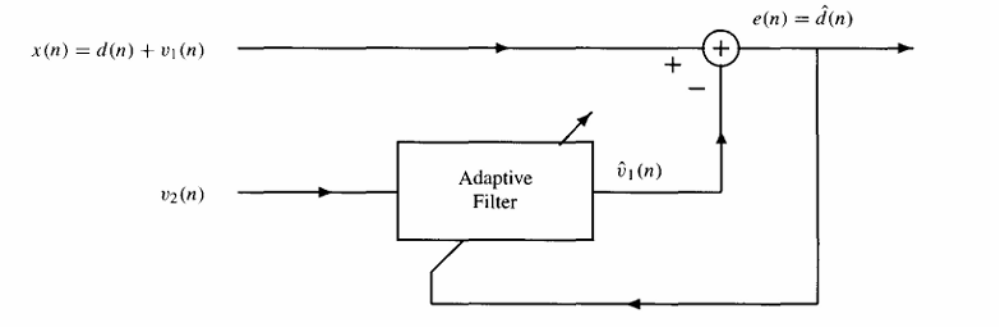


Figure 1.1: Generic Noise Cancellation with reference-Adaptive Filter block diagram

From the block diagram we clearly understand that we have a desired signal $d(n)$ corrupted with some noise $v_1(n)$, and we also have another reference for the same source of noise, providing us $v_2(n)$. Now there is some statistical relation between $v_1(n)$ and $v_2(n)$ because they come from the same source, so we use this to our advantage to de-noise the input $x(n)$. Thus, the adaptive filter here tries to **estimate the noise in the primary input and makes attempts to cancel it**. Thus, the "error" that's obtained is an estimate of the desired signal, which is the noise-free signal. The term "error" here is misleading in the nature that it's usually used to denote something as *unwanted*, while the de-noised signal here is the actually wanted signal!

What's the error $e(n)$ actually over here? As already mentioned, it's an estimate of the desired signal, with the goal being to minimise the mean-square error using the adaptive filter. This intuitively means that we have our signal power and noise power both in the error signal $e(n)$. In other words, the output of the adaptive filter is the minimum mean-square estimate of $v_1(n)$. And, if there is no information about $d(n)$ in the reference signal $v_2(n)$, then the best that the adaptive filter can do to minimize $\mathbb{E}\left[|e(n)|^2\right]$ is to remove the part of $e(n)$ that may be estimated from $v_2(n)$, which is $v_1(n)$. Since the output of the adaptive filter is the minimum mean-square estimate of $v_1(n)$, it follows that $e(n)$ is the minimum mean-square estimate of $d(n)$. So we're **minimising the noise power!**

With the problem framework understood, let's move to the implementation of the adaptive filters, the steepest descent adaptive filter (SDAF), the least mean squares filter (LMS), and the recursive least squares (RLS), all of order 4.

i. Steepest Descent Adaptive Filter

Given an objective, or a cost function, the method of Steepest Descent is an iterative procedure that is used to find extrema of nonlinear functions! So now it becomes an **optimisation problem**! This algorithm is primarily of theoretical interest and finds little use in adaptive filtering applications, for the sole fact that we need the gradient vector, which is often estimated (making it a stochastic descent). Note that here, we've only been provided with a single realization, and not multiple realizations of the signal, so we could average over them and get an **ensemble average**! Since this isn't possible, we have to depend on the time-average approximation.

The way the Steepest Descent approach works is that we have the gradient computed using some method, and we have a step-size that we use, so that we can use it in the update equation to move down the loss surface as we want to minimize the loss. Typically, the gradient is an estimate, and thus the gradients obtained are **noisy** and hence this is where the term **stochastic** comes from. The term stochastic comes from the fact that the subset of the actual value taken, creates a noisy estimate.

So in the case of the SDAF (Steepest Descent Adaptive Filter), we compute the gradient using the entire dataset at each iteration. This approach is also known as batch gradient descent, as it relies on full statistical estimates like the autocorrelation matrix and cross-correlation vector computed from all available data. While this method is computationally expensive, it typically yields the most accurate results in terms of convergence and steady-state error, since it uses complete second-order statistics. This is consistent with my implementation of the SDAF filter, which follows the steepest descent principle for minimizing the mean-square error.

The script I've implemented is given below. The inputs the function takes are the corrupted signal $x(n)$, the noise reference $v_2(n)$, the filter order which is 4, so we'd have 5 taps, the step size and the maximum number of iterations that we want to go for, because ideally this can be infinite.

It is important to choose the step-size carefully to ensure that the algorithm converges smoothly toward the minimum of the loss function. If the step-size is too large, the updates may overshoot, causing the solution to oscillate around or even diverge away from the minimum, leading to instability in the filter output. On the other hand, if it is too small, we might converge very slowly! We already know in a stationary sense that the convergence condition is when the step-size is bounded between 0 and $2/\lambda_{max}$, where λ_{max} is the maximum eigenvalue in the autocorrelation matrix of the input to the adaptive filter.

$$0 < \mu < \frac{2}{\lambda_{max}}$$

The inputs to the adaptive filter in both cases are $v_2(n)$ and $w_2(n)$, so we need to find the estimated autocorrelation matrix for them.

Let $\mathbf{v}_2(n)$ be the reference noise signal, and let p be the filter order. For each time index n, define the vector:

$$\mathbf{x}_n = \begin{bmatrix} v_2(n) \\ v_2(n+1) \\ \vdots \\ v_2(n+p) \end{bmatrix} \in \mathbb{R}^{p+1}$$

The time-averaged autocorrelation matrix $\mathbf{R}_{v_2} \in \mathbb{R}^{(p+1)\times(p+1)}$ is then computed as:

```
Step Size Bounds:
Signal 1 (Noise v2): 0 < mu < 1.696438
Signal 2 (Noise w2): 0 < mu < 0.387291
```

Figure 1.2: Step-size bounds

$$\mathbf{R}_{v_2} = \frac{1}{N}\sum_{n=0}^{N-1} \mathbf{x}_n \mathbf{x}_n^\top$$

where $N = \text{length}(v_2) - p$, and $\mathbf{x}_n\mathbf{x}_n^\top$ is the outer product of $\mathbf{x}_n$ with itself. This matrix captures the correlation between each pair of lags in the signal! Similarly, this is done for $w_2(n)$. The step size bounds thus obtained are given below:

We go for a tighter bound, which is $0 < \mu < 0.38$.

Throughout the code, I've chosen the step-size to be a convenient value of 0.01. Coming to the SDAF filter implementation again, the function outputs the filtered signal, the history of the filter taps and the moving-average MSE history as well, as the iterations go by, these quantities are found and stored.

The goal of the SDAF, again, is to filter out the noise $v_1(n)$ from $x(n)$ using the reference noise signals, where the filter estimates the noise contribution from $v_2(n)$. The output of the filter at time n is given by:

$$\hat{v_1}(n) = \sum_{k=0}^{p} w_k v_2(n-k)$$

In vector notation, the filter output becomes:

$$\hat{v_1}(n) = \mathbf{w}^T \mathbf{v}_2(n)$$

where

$$\mathbf{w} = [w_0, w_1, \ldots, w_p]^T, \quad \mathbf{v}_2(n) = [v_2(n), v_2(n-1), \ldots, v_2(n-p)]^T$$

This vector $\mathbf{v}_2(n)$ is the **delay line** that holds the current and past p samples of the second noise reference signal $v_2(n)$. The purpose of the delay line is to create a set of weighted coefficients to estimate the noise, which can then be subtracted from the corrupted signal.

The error signal is defined as the difference between the corrupted signal and the noise estimate:

$$e(n) = x(n) - \hat{v_1}(n) = d(n) + v_1(n) - \hat{v_1}(n)$$

The goal is to minimize the expected squared error:

$$J(\mathbf{w}) = \mathbb{E}[e^2(n)] = \mathbb{E}[(x(n) - \hat{v_1}(n))^2]$$

which leads to the best estimate of $v_1(n)$ using $v_2(n)$.

To minimize $J(\mathbf{w})$, we use the steepest descent update rule, which is:

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \mu \nabla J(\mathbf{w}^{(i)})$$

and the gradient of the cost function with respect to the weights is:

$$\nabla J(\mathbf{w}) = -2\mathbf{R}_{v_2 v_2}\mathbf{w} + 2\mathbf{r}_{v_2 x}$$

where

$$\mathbf{R}_{v_2v_2} = \mathbb{E}[\mathbf{v}_2(n)\mathbf{v}_2^T(n)], \quad \mathbf{r}_{v_2x} = \mathbb{E}[\mathbf{v}_2(n)x(n)]$$

These represent the autocorrelation matrix of the reference noise signal and the cross-correlation vector between the reference noise signal and the corrupted signal, respectively.

Since the true expectations are unknown, we compute the time-averaged values over the available data:

$$\mathbf{R}_{v_2v_2} \approx \frac{1}{N}\sum_{n=0}^{N-1}\mathbf{x}_2(n)\mathbf{x}_2^T(n)$$

$$\mathbf{r}_{v_2x} \approx \frac{1}{N}\sum_{n=0}^{N-1}\mathbf{x}_2(n)x(n)$$

The weights are updated iteratively using the gradient descent algorithm:

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \frac{\mu}{N}\mathbf{X}_2^T\mathbf{e}$$

where $\mathbf{X}_2$ is the matrix formed by stacking $\mathbf{x}_2(n)$ for all n, and $\mathbf{e}$ is the error vector.

Once the filter weights converge, the estimated noise is:

$$\hat{v_1}(n) = \mathbf{x}_2(n)\mathbf{w}$$

and the filtered signal, which is the clean ECG estimate, is:

$$\hat{d}(n) = x(n) - \hat{v_1}(n)$$

Thus, the SDAF filter utilizes both the primary noise reference $v_1(n)$ and the secondary noise reference $v_2(n)$ to remove noise from the ECG signal $x(n)$, with the delay line playing a crucial role in the adaptation process.

The script for the SDAF filter is provided below:

Figure 1.3: SDAF script

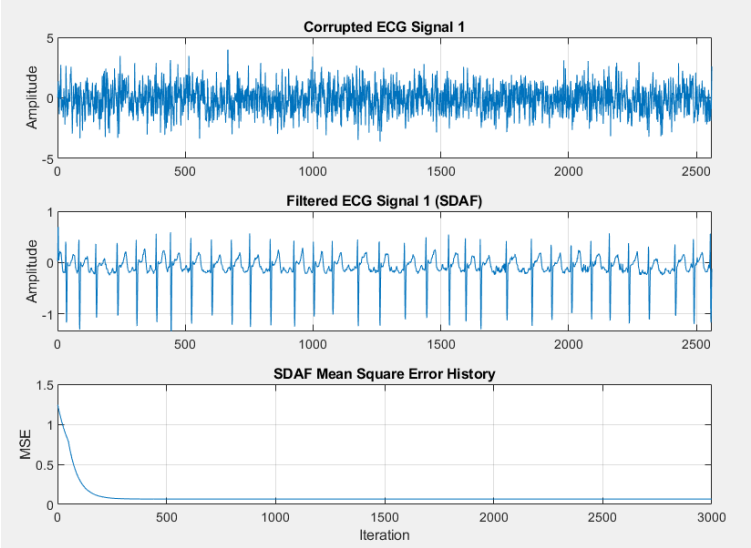


Figure 1.4: SDAF output for first signal

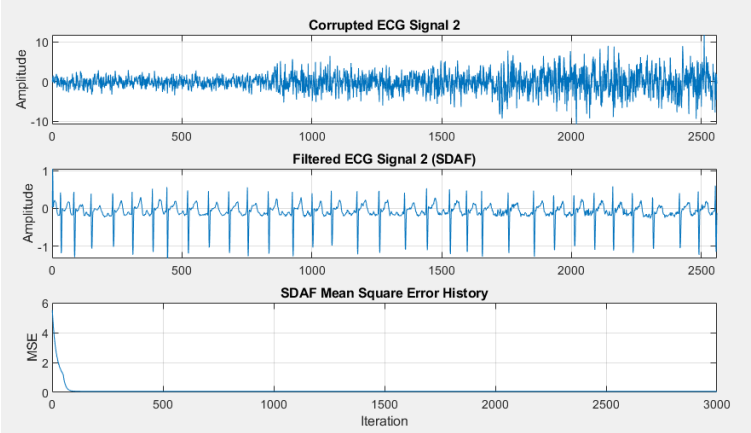


Figure 1.5: SDAF output for second signal

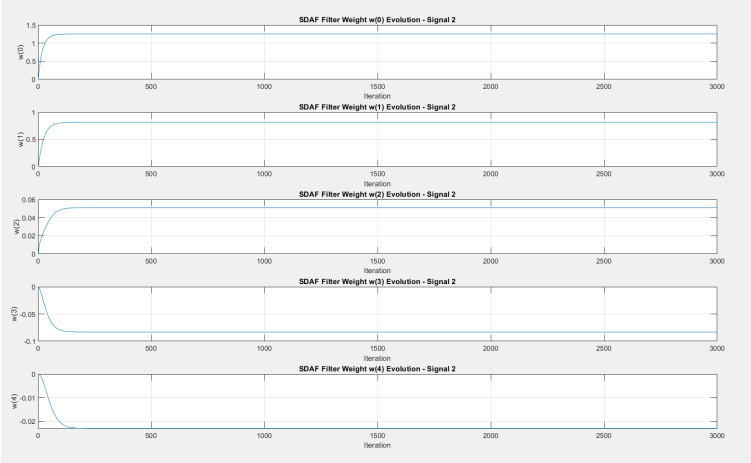


Figure 1.6: Update of the filter taps for signal1

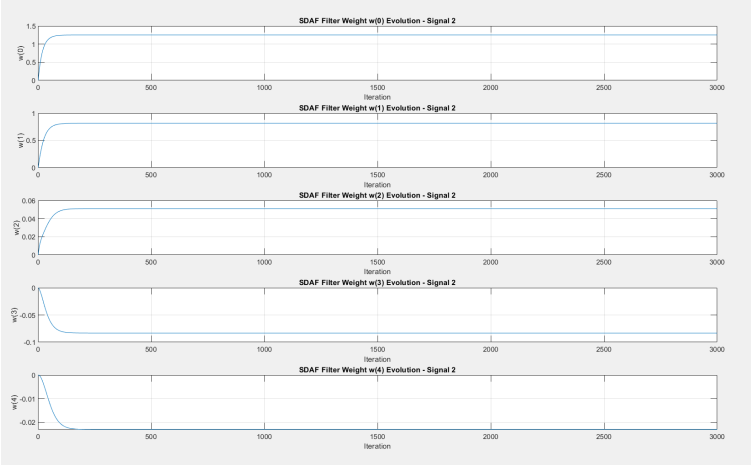


Figure 1.7: Update of the filter taps for signal2

Although more computationally intensive than LMS and RLS, the SDAF is indeed.. the best estimate of the gradient. We are also aware that in the stationary sense, the filter also converges to the Wiener filter, which is the optimum filter in terms of minimizing the MSE.

ii. Least Mean Squares (LMS)

Typically, in practical situations, an estimate of the gradient $\mathbb{E}[e(n)\mathbf{x}(\mathbf{n})]$ is considered, due to non-availability of data, or otherwise.

The Least Mean Squares (LMS) algorithm is a widely used adaptive filtering technique that updates filter weights iteratively to minimize the mean square error between a desired signal and the filter's output. Unlike the Steepest Descent Adaptive Filter (SDAF), which uses *full statistical information* (e.g., autocorrelation matrices), LMS is a **stochastic gradient descent method**, i.e. it uses **instantaneous estimates** of the gradient, making it **computationally very efficient** compared to the SDAF or RLS counterparts and suitable for real-time applications.

The computational ease of the LMS algorithm comes at a cost! the gradient is estimated using instantaneous error samples, which introduces *higher gradient noise*. Consequently, in the case of SDAF since the filter gradient is a much better estimate, we have the freedom to be more aggressive in terms of the convergence, while in the case of the LMS, fine-tuning the step size becomes an issue because if it is too large, we might diverge altogether, and not converge at all! The LMS does converge in the mean-sense to the Wiener-Hopf, and never directly converges to it because of noisy gradient.

So we expect the LMS filter to have a slower convergence rate than the SDAF filter, and have a relatively higher MSE value compared to SDAF!

Let's come to the theory and implementation of the LMS filter.

Let:

- $d(n)$: the primary signal (desired signal + noise)
- $v_2(n)$: the reference noise signal
- $x(n)$: vector of delayed versions of $v_2(n)$ at time n
- $\mathbf{w}(n)$: filter coefficient vector at time n

The objective is to estimate the noise in $d(n)$ using the reference $v_2(n)$, and subtract it to recover the original signal. The LMS algorithm minimizes the cost function:

$$\text{MSE} = \mathbb{E}[e^2(n)]$$

where the error signal is defined as:

$$e(n) = d(n) - \mathbf{w}^T(n) \cdot x(n)$$

The algorithm steps involve:

1. **Initialization**:
   - Initialize the weights $\mathbf{w}(0)$ to zeros.
   - Set up arrays to store the filtered signal, MSE history, and weight updates.
2. **Delay Line Construction**: For each time step n, construct a delay line vector $x(n)$ using the reference signal:
   $$x(n) = [v_2(n), v_2(n-1), \ldots, v_2(n-p)]^T$$
   where p is the filter order.
3. **Noise Estimation**: Estimate the noise as:
   $$\hat{y}(n) = \mathbf{w}^T(n) \cdot x(n)$$
4. **Error Calculation**: Compute the estimation error:
   $$e(n) = d(n) - \hat{y}(n)$$
   This error is also used as the cleaned signal output.
5. **Weight Update Rule**: Update the filter weights using the LMS rule:
   $$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu \cdot e(n) \cdot x(n)$$
   where μ is the learning rate or step size.
6. **MSE Tracking**: The mean squared error is tracked using a cumulative average:
   $$\text{MSE}(n) = \frac{1}{n} \sum_{i=1}^{n} e^2(i)$$

Thus the LMS algorithm provides a computationally efficient, real-time method for adaptive noise cancellation. However, because it relies on instantaneous gradient estimates, it introduces higher gradient noise compared to batch methods like Steepest Descent. As a result, its convergence can be slower and noisier, although it remains effective in many practical scenarios.

The script for the LMS filter is given below:

```m
function [filtered_signal, mse_history, w_history] = lms_filter(primary_signal, reference_signal, filter_order, mu, max_iterations)
    % LMS_FILTER Least Mean Squares adaptive filter implementation
    %   [FILTERED_SIGNAL, MSE_HISTORY, W_HISTORY] = LMS_FILTER(PRIMARY_SIGNAL,
    %   REFERENCE_SIGNAL, FILTER_ORDER, MU, MAX_ITERATIONS)
    %
    %   Inputs:
    %       primary_signal   - The corrupted signal (desired + noise)
    %       reference_signal - The reference noise signal
    %       filter_order     - Filter order (number of filter coefficients + 1)
    %       mu               - Step size (learning rate)
    %       max_iterations   - Maximum number of iterations
    %
    %   Outputs:
    %       filtered_signal  - The filtered output signal
    %       mse_history      - Mean squared error at each iteration
    %       w_history        - Filter weights at each iteration

    w = zeros(filter_order + 1, 1);

    % output arrays
    signal_length = length(primary_signal);
    filtered_signal = zeros(signal_length, 1);
    mse_history = zeros(min(max_iterations, signal_length), 1);
    w_history = zeros(filter_order + 1, min(max_iterations, signal_length));

    % for MSE
    error_sum = 0;

    for n = 1:min(max_iterations, signal_length)
        % ref signal
        if n <= filter_order
            % zeropad
            x = [reference_signal(1:n); zeros(filter_order + 1 - n, 1)];
        else
            x = reference_signal(n:-1:n-filter_order);
        end

        y_hat = w' * x;
        e = primary_signal(n) - y_hat;

        % error signal (cleaned signal)
        filtered_signal(n) = e;
        w = w + mu * e * x; % LMS weight update
        % cumulative error sum
        error_sum = error_sum + e^2;
        mse_history(n) = error_sum / n;

        % store filter
        w_history(:, n) = w;
    end
```

Figure 1.8: LMS Filter Script

The plots obtained for the filtered output and the learning curve as a function of the iterations are provided below.

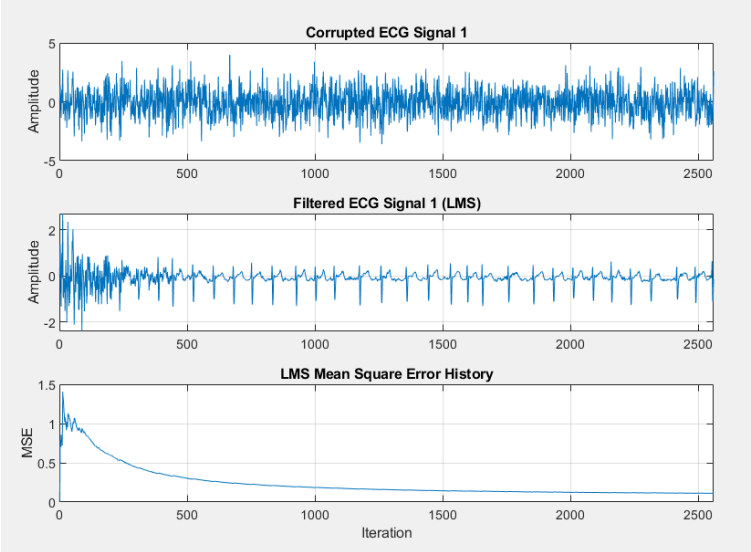


Figure 1.9: LMS output for first signal

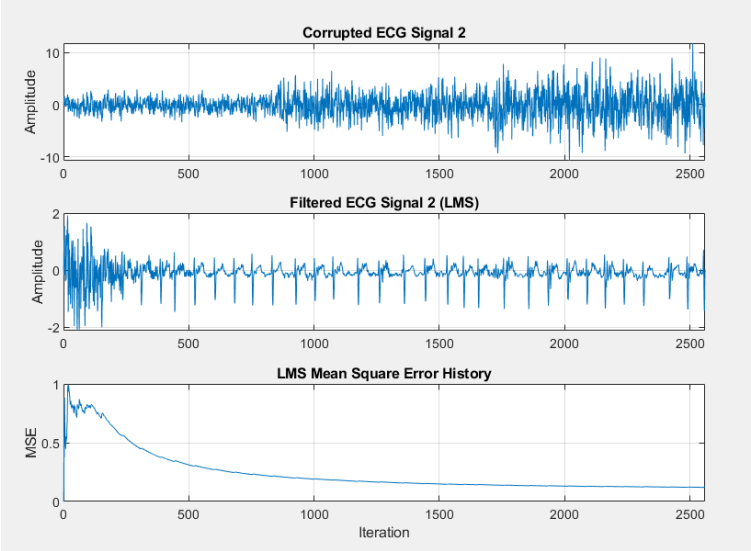


Figure 1.10: LMS output for second signal

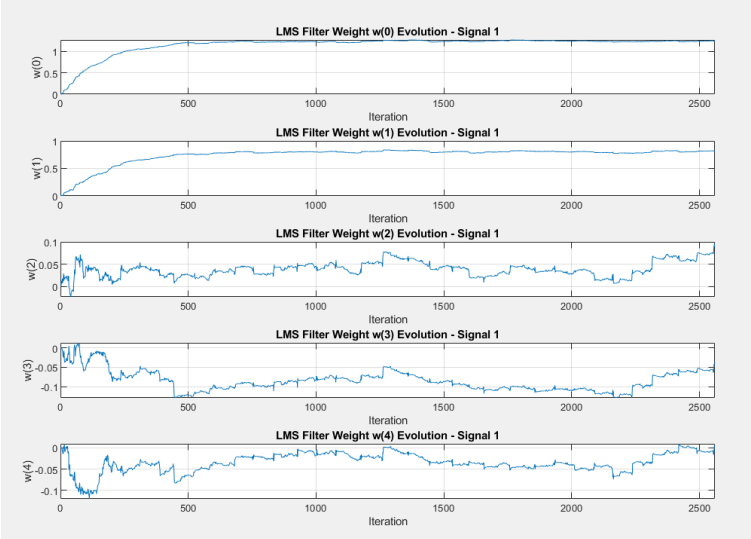


Figure 1.11: LMS update of the filter taps for signal1

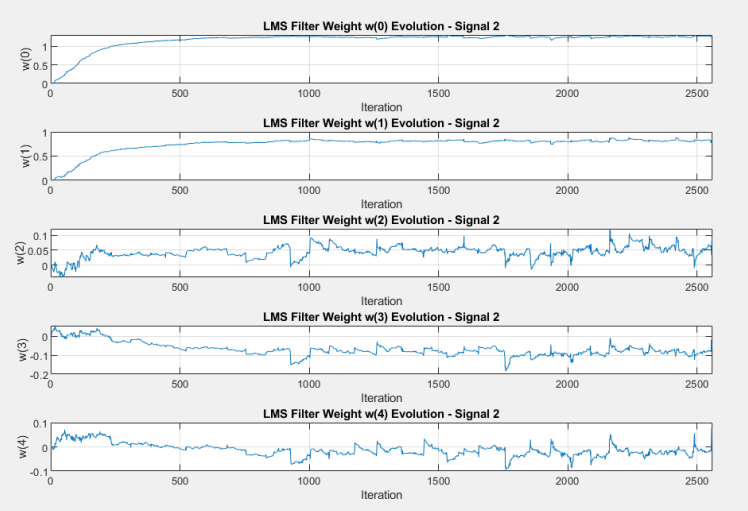


Figure 1.12: LMS update of the filter taps for signal2

Clearly, the high gradient noise causes the filter to error slowly, and with a higher MSE.

iii. Recursive Least Squares (RLS)

The Recursive Least Squares algorithm is an adaptive filtering method that aims to **minimize the total squared error** between the desired signal and the filter output, by recursively updating its coefficients. Unlike LMS, which uses instantaneous error for updates, RLS uses all past data (with an exponential weighting) to make more accurate and faster adjustments. This results in rapid convergence, even under changing signal conditions, making it highly effective in non-stationary environments. However, this performance comes at the cost of increased computational complexity, due to matrix operations at each iteration.

NOTE that the time complexity of the Steepest Descent Adaptive Filter (SDAF) is given by $\mathcal{O}(N \cdot p \cdot I)$, with I iterations and where N is the number of samples, and p is the filter order, since each gradient update uses the entire dataset and involves a computation of cost $\mathcal{O}(N \cdot p)$ per iteration. In contrast, the Least Mean Squares (LMS) algorithm has a significantly lower computational complexity of $\mathcal{O}(p)$ per iteration, since it updates the weights using only the current data sample. The Recursive Least Squares (RLS) algorithm, which we see later, has a higher complexity of $\mathcal{O}(p^2)$ per iteration due to the matrix operations involved in its update rules. Note that here $N >> p$, so as we knew intuitively, SDAF is computationally most expensive, while next comes RLS and then LMS.

Let's come to the theory and implementation of the RLS filter. Given a primary signal $d(n)$ and a reference noise signal $v_2(n)$, the RLS filter operates using the following steps at each time step n:

1. **Construct the input vector** from the reference signal using a tapped delay line of length $p+1$:

$$\mathbf{x}(n) = \begin{bmatrix} v_2(n) & v_2(n-1) & \cdots & v_2(n-p) \end{bmatrix}^T$$

Zero-padding is used during initial iterations if $n < p$.

2. **Compute the gain vector**:

$$\mathbf{z}(n) = \mathbf{P}(n-1)\mathbf{x}(n)$$

$$\mathbf{g}(n) = \frac{\mathbf{z}(n)}{\lambda + \mathbf{x}^T(n)\mathbf{z}(n)}$$

where $\mathbf{P}(n)$ is the inverse correlation matrix and λ is the forgetting factor ($0 < \lambda \leq 1$), which controls the weight of past errors.

3. **Estimate the output and error**:

$$\hat{y}(n) = \mathbf{w}^T(n-1)\mathbf{x}(n)$$

$$e(n) = d(n) - \hat{y}(n)$$

Here, $\hat{y}(n)$ is the estimate of the noise component. Subtracting it from $d(n)$ yields the denoised signal.

4. **Update the filter weights**:

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{g}(n)e(n)$$

5. **Update the inverse correlation matrix**:

$$\mathbf{P}(n) = \frac{1}{\lambda}\left[\mathbf{P}(n-1) - \mathbf{g}(n)\mathbf{x}^T(n)\mathbf{P}(n-1)\right]$$

6. **Track the performance**: The mean squared error (MSE) is computed using a cumulative average:

$$\text{MSE}(n) = \frac{1}{n}\sum_{k=1}^{n} e^2(k)$$

The RLS algorithm provides superior performance in terms of convergence and steady-state error, particularly in non-stationary environments, but at the cost of increased complexity, i.e. $\mathcal{O}(p^2)$ per iteration.

The script for the RLS filter is given below.

```
function [filtered_signal, mse_history, w_history] = rls_filter(primary_signal, reference_signal, filter_order, lambda, delta, max_iterations)
    % RLS_FILTER Recursive Least Squares adaptive filter implementation
    %   [FILTERED_SIGNAL, MSE_HISTORY, W_HISTORY] = RLS_FILTER(PRIMARY_SIGNAL,
    %   REFERENCE_SIGNAL, FILTER_ORDER, LAMBDA, DELTA, MAX_ITERATIONS)
    %
    %   Inputs:
    %       primary_signal   - The corrupted signal (desired + noise)
    %       reference_signal - The reference noise signal
    %       filter_order     - Filter order (number of filter coefficients + 1)
    %       lambda           - Forgetting factor
    %       delta            - Initial value for P(0) matrix (regularization parameter)
    %       max_iterations   - Maximum number of iterations
    %
    %   Outputs:
    %       filtered_signal  - The filtered output signal
    %       mse_history      - Mean squared error at each iteration
    %       w_history        - Filter weights at each iteration

    w = zeros(filter_order + 1, 1);

    % inverse correlation matrix
    P = (1/delta) * eye(filter_order + 1); % P(0) = δ[^-1]I

    signal_length = length(primary_signal);
    filtered_signal = zeros(signal_length, 1);
    mse_history = zeros(min(max_iterations, signal_length), 1);
    w_history = zeros(filter_order + 1, min(max_iterations, signal_length));

    % Initialize error sum
    error_sum = 0;

    for n = 1:min(max_iterations, signal_length)
        % current input vector
        if n <= filter_order
            % Pad with zeros for the initial samples
            x = [reference_signal(1:n); zeros(filter_order + 1 - n, 1)];
        else
            x = reference_signal(n:-1:n-filter_order);
        end

        % gain vector
        z = P * x; % z(n) = P(n-1)x(n)
        g = z / (lambda + x' * z); % g(n) = z(n) / (λ + x'(n)z(n))

        % output of the adaptive filter
        y_hat = w' * x; % (estimated noise)
        e = primary_signal(n) - y_hat; % apriori error
        w = w + g * e;

        % Update the inverse correlation matrix
        P = (1/lambda) * (P - g * x' * P); % P(n) = (1/λ)[P(n-1) - g(n)z'(n)]

        % error signal (cleaned signal)
        filtered_signal(n) = e;

        % cumulative error sum
        error_sum = error_sum + e^2;
        mse_history(n) = error_sum / n;  % Time-averaged MSE
        w_history(:, n) = w;
    end
end
```

Figure 1.13: RLS Filter Script

The filter update graph and the filter outputs plots are provided below:

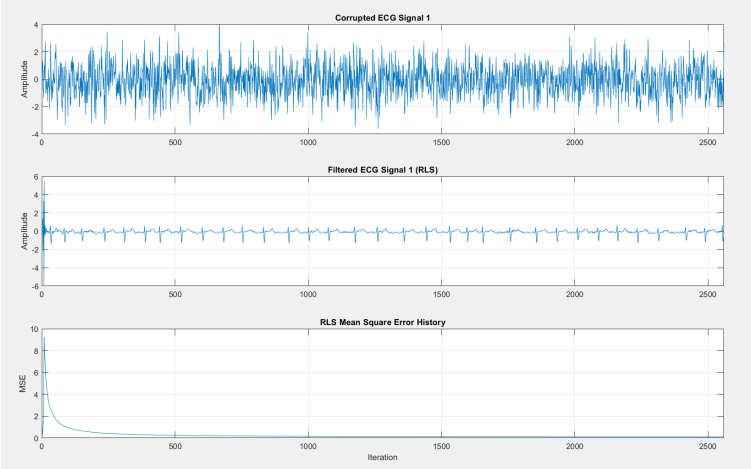


Figure 1.14: RLS Filter output for signal1

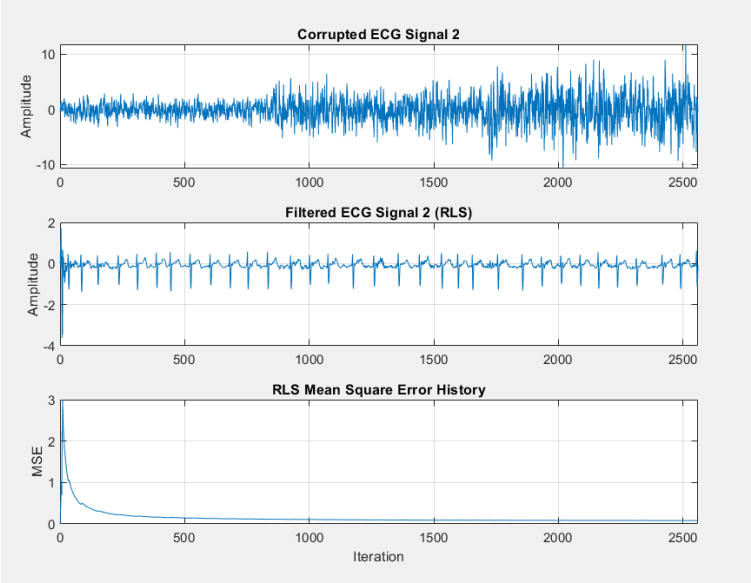


Figure 1.15: RLS Filter output for signal2

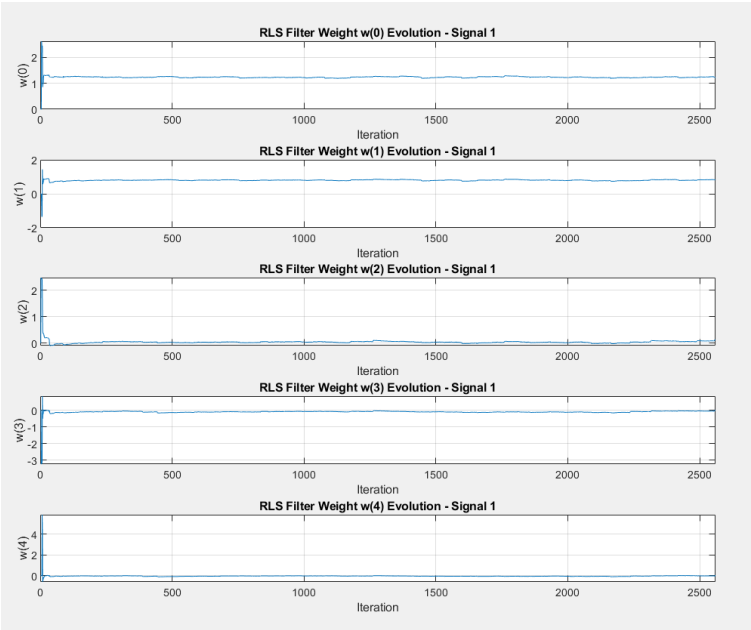


Figure 1.16: RLS Filter update for signal1

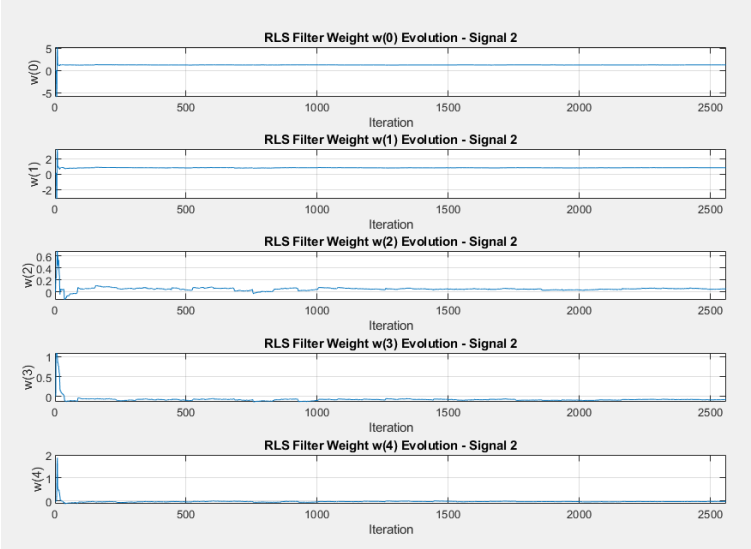


Figure 1.17: RLS Filter update for signal2

The convergence is thus indeed faster than LMS, but obviously slower than the SDAF. The error could depend w.r.t LMS, but we can see that it is more than the SDAF since the SDAF has the best gradient estimate out of the three!

This completes the section of implementing the three filters. We now look at the performance, convergence and stationarity v/s non-stationarity comparisons between the three filters.

(b) Relative performance in terms of the noise removal and convergence

As discussed quite of it above, let's first plot the "error" in the three cases, which is the filter outputs obtained using the converged/final filter coefficients.

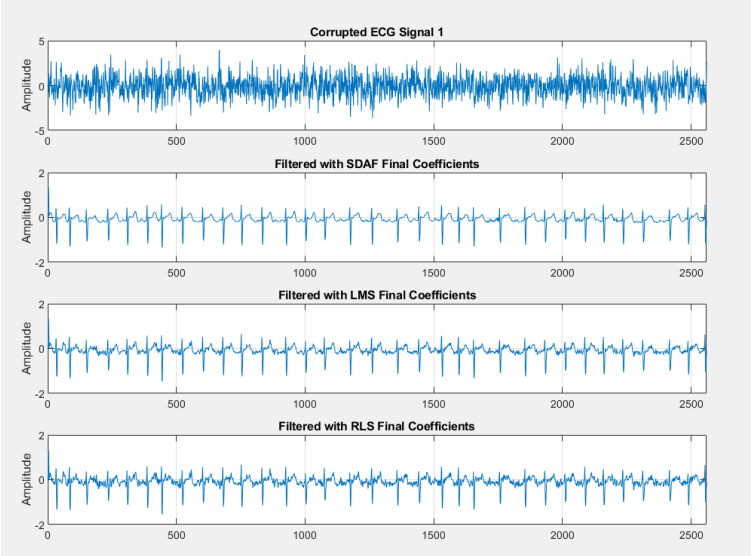


Figure 1.18: Final filter outputs for Signal1

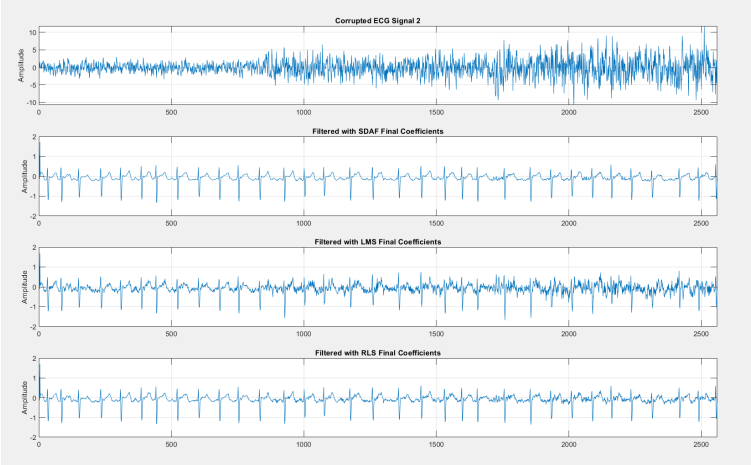


Figure 1.19: Final filter outputs for Signal2

Although not quantitative, it is worth mentioning that the filter outputs above show that for Signal1, it's not very clear in the error among LMS and RLS, that which one is better, but to pick one, LMS seems to have a tad-bit better error. A reason for the RLS having higher error could be that the forgetting factor isn't strong enough, because when the forgetting factor is made 0.997 for example, the RLS does perform better than the LMS filter.

Other reasons as to why the SDAF has the best convergence (in the stationary sense) compared to the others is already explained above, we're sacrificing with the computational complexity for a better gradient estimate, while for the LMS we see that we're using instantaneous values, hence the complexity is linear, while the complexity is in the middle for the RLS filter, giving us faster convergence than the LMS but could be at a cost of error. In the case of non-stationarity, we'd obviously pick either the LMS or the RLS depending on various factors like how fast we need the convergence to be, (then RLS would be preferred using an exponentially decaying window), and so on and so forth!

(c) Identifying non-stationarity set

Right off the bat, looking at the signals itself, made me feel the noise statistics are changing like a "step function" in case2, which made me incline towards the fact that set2 is non-stationary, although this is highly quantitative, not a hard-reasoning. In adaptive filters dealing with non-stationary statistics, the error surface becomes dynamic rather than fixed, with its minimum point continuously shifting as signal statistics evolve over time. This fundamental characteristic creates a persistent challenge where the filter must constantly chase a moving target rather than converge to a stable solution. The adaptive algorithm experiences an unavoidable lag between when statistical changes occur and when the filter can respond, introducing tracking error that simply doesn't exist in stationary environments. This tracking error, combined with the need to repeatedly re-estimate changing statistics and the inherent gradient noise in adaptation methods, inevitably produces higher Mean Square Error compared to stationary scenarios where filters can achieve and maintain optimal performance at a fixed point.

One attempt towards a solution I can think of, is that we might need to use a variable step-size that also depends on the statistics, because we need to inculcate the statistics somewhere for such scenarios!

```
Signal 1 Final MSE Values:
SDAF: 0.066673
LMS: 0.112564
RLS: 0.101620

Signal 2 Final MSE Values:
SDAF: 0.067105
LMS: 0.121917
RLS: 0.082123

Final Filter Application MSE for Signal 1:
SDAF: 0.067444
LMS: 0.070339
RLS: 0.074640

Final Filter Application MSE for Signal 2:
SDAF: 0.068021
LMS: 0.086684
RLS: 0.069420
```

Figure 1.20: Errors in the learning curve, and using the final coefficients

We can see that the error in the case of Signal2 is a tad-bit worse than Signal1, which makes me incline towards the fact that the set in case2 is non-stationary.

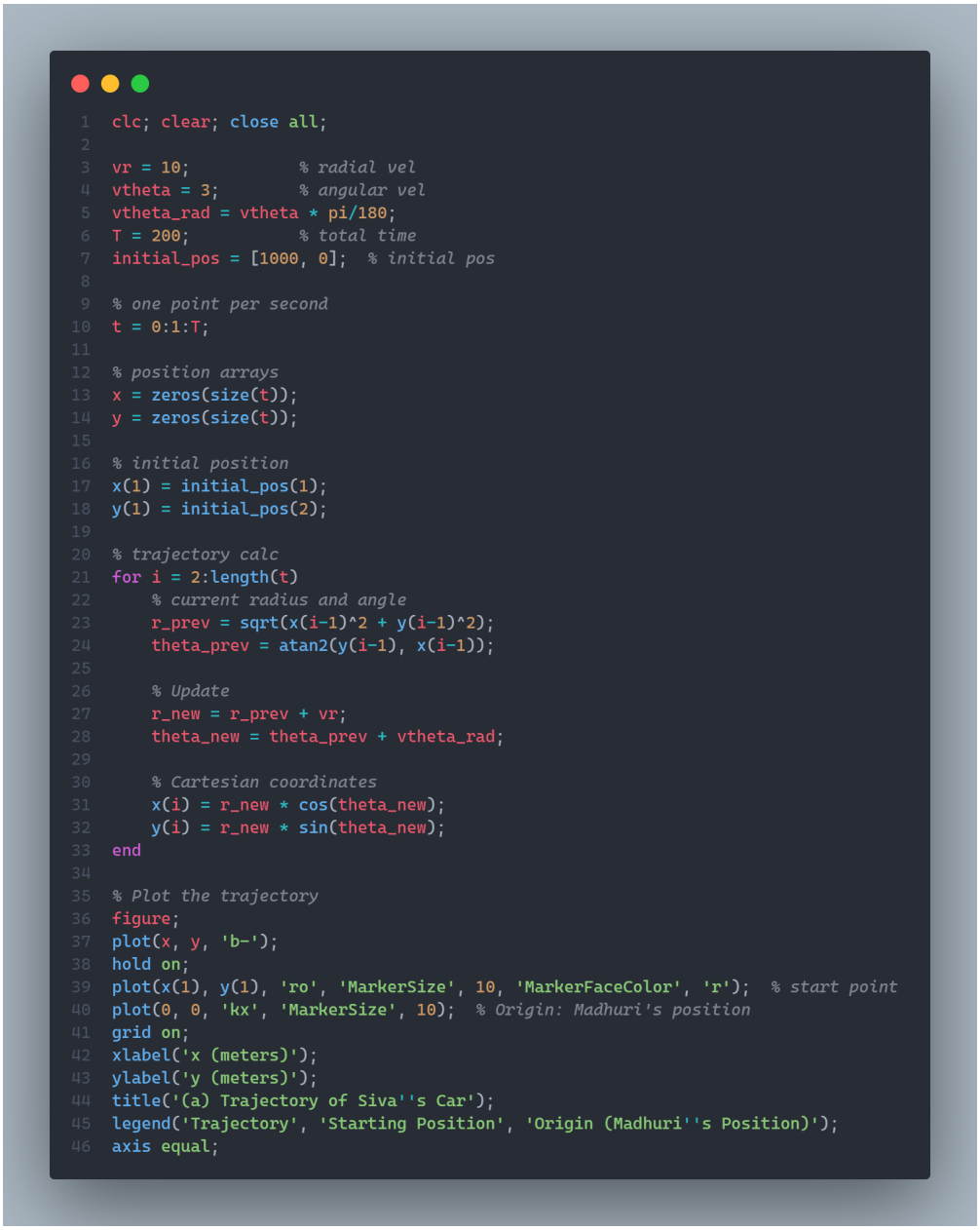
Problem 2: Tracking a Moving Vehicle using Kalman Filtering

In this problem statement, we simulate and track the motion of a car driven by Siva, who follows a spiral-like trajectory in the 2D plane. His motion is defined by constant radial velocity and angular velocity, leading to a deterministic but nonlinear path. A second observer, Madhuri, located at the origin, uses noisy range and angle measurements from sensors to estimate Siva's position and motion parameters, including radial and angular velocities, over time.

We approach this tracking problem using a Kalman Filter, which allows Madhuri to recursively estimate the car's state (range, angle, and their time derivatives) in the presence of measurement noise. The report outlines the modeling of the system dynamics, measurement model, Kalman Filter implementation, and performance analysis under different conditions, including the use of additional sensors.

(a) Trajectory of Siva's car

Siva begins at the point $[1000, 0]$ meters in the x-y plane and drives with a constant radial velocity of $v_r = 10\,\text{m/s}$ and angular velocity of $v_\theta = 3^\circ/\text{s}$. As a result, his path forms an outward-moving spiral. The following MATLAB code simulates and plots his position over a duration of 200 seconds.



```
clc; clear; close all;

vr = 10;            % radial vel
vtheta = 3;         % angular vel
vtheta_rad = vtheta * pi/180;
T = 200;            % total time
initial_pos = [1000, 0];  % initial pos

% one point per second
t = 0:1:T;

% position arrays
x = zeros(size(t));
y = zeros(size(t));

% initial position
x(1) = initial_pos(1);
y(1) = initial_pos(2);

% trajectory calc
for i = 2:length(t)
    % current radius and angle
    r_prev = sqrt(x(i-1)^2 + y(i-1)^2);
    theta_prev = atan2(y(i-1), x(i-1));

    % Update
    r_new = r_prev + vr;
    theta_new = theta_prev + vtheta_rad;

    % Cartesian coordinates
    x(i) = r_new * cos(theta_new);
    y(i) = r_new * sin(theta_new);
end

% Plot the trajectory
figure;
plot(x, y, 'b-');
hold on;
plot(x(1), y(1), 'ro', 'MarkerSize', 10, 'MarkerFaceColor', 'r');  % start point
plot(0, 0, 'kx', 'MarkerSize', 10);  % Origin: Madhuri's position
grid on;
xlabel('x (meters)');
ylabel('y (meters)');
title('(a) Trajectory of Siva''s Car');
legend('Trajectory', 'Starting Position', 'Origin (Madhuri''s Position)');
axis equal;
```

Figure 2.1: MATLAB script for trajectory plotting.

The obtained plot is provided below.

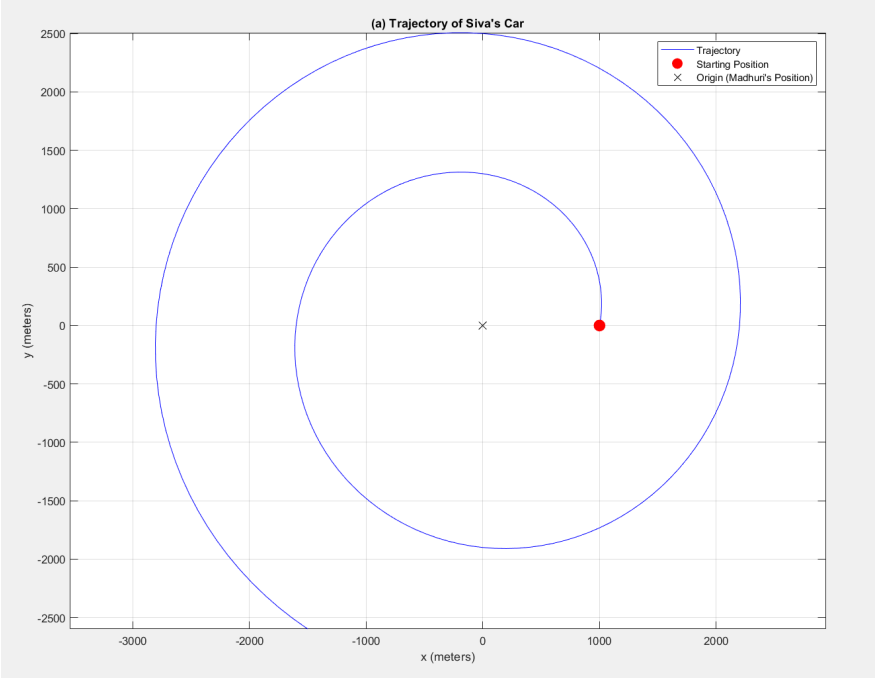


Figure 2.2: Siva's trajectory

(b) Madhuri's State-Space Model Formulation

In order to track Siva's car using the available sensor data, we need to formulate the problem in a state-space framework. This will require the definition of the state variables that fully describe the system, the state transition model that governs how these variables evolve over time, and the measurement model that relates the sensor readings to the state variables. By correctly formulating these equations, Madhuri can implement a Kalman filter to optimally estimate the car's position and velocity despite noisy sensor readings!

i. State Vector Definition

Firstly, we need to define the **state vector** that contains all the parameters *we want to track*. As mentioned in the document, we need to track the range (which is the distance from the origin), angle, radial velocity, and angular velocity, thus our state vector is:

$$\mathbf{x}_k = \begin{bmatrix} r_k \\ \theta_k \\ v_{r,k} \\ v_{\theta,k} \end{bmatrix}$$

Where:

$$r_k = \text{range} \tag{2.1}$$
$$\theta_k = \text{angle} \tag{2.2}$$
$$v_{r,k} = \text{radial velocity} \tag{2.3}$$
$$v_{\theta,k} = \text{angular velocity} \tag{2.4}$$

ii. State Transition Equation

Since Siva drives with constant radial and angular velocities, the state transition equation can be written as:

$$\mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{w}_k$$

Where the state transition matrix $\mathbf{A}_k$ is:

$$\mathbf{A}_k = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With $\Delta t = 1$ second (the sampling interval). This matrix updates the state as follows:

$$r_{k+1} = r_k + v_{r,k} \cdot \Delta t \tag{2.5}$$
$$\theta_{k+1} = \theta_k + v_{\theta,k} \cdot \Delta t \tag{2.6}$$
$$v_{r,k+1} = v_{r,k} \tag{2.7}$$
$$v_{\theta,k+1} = v_{\theta,k} \tag{2.8}$$

The term $\mathbf{w}_k$ represents the process noise with covariance matrix $\mathbf{Q}_w$, which accounts for any uncertainties in the motion model.

iii. Measurement Equation

Madhuri has two sensors measuring range and angle, so the measurement vector is:

$$\mathbf{z}_k = \begin{bmatrix} r_k^m \\ \theta_k^m \end{bmatrix}$$

Where r_k^m and θ_k^m are the measured range and angle respectively.
The measurement equation relates the measurements to the state:

$$\mathbf{y}_k = \mathbf{C}_k \mathbf{x}_k + \mathbf{v}_k$$

Where the observation matrix $\mathbf{C}_k$ is:

$$\mathbf{C}_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

This matrix extracts the range and angle components from the state vector for comparison with the measurements.

The term $\mathbf{v}_k$ represents the measurement noise with covariance matrix:

$$\mathbf{Q}_v = \begin{bmatrix} 2500 & 0 \\ 0 & 16 \end{bmatrix}$$

Based on the given sensor variances: 2500 for the range sensor and 16 for the angle sensor. The sensors are uncorrelated, hence the zero off-diagonal terms.

The error covariance matrix $\mathbf{P}_k$ evolves over time and will be estimated as part of the Kalman filter implementation in subsequent parts.

(c) Kalman Filter Equations

The Kalman filter, as we already know from the course, provides an optimal framework for estimating the state of Siva's car based on noisy measurements. It operates as a recursive estimator that combines predictions from the dynamic model with corrections from sensor measurements. Here, we will present the Kalman filter equations that Madhuri should implement for tracking Siva's car, taking into account both the state transition model and the measurement model defined earlier.

State Equation:

$$\mathbf{x}(n) = \mathbf{A}(n-1)\mathbf{x}(n-1) + \mathbf{w}(n) \tag{2.9}$$

$$\begin{bmatrix} r(n) \\ \theta(n) \\ v_r(n) \\ v_\theta(n) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r(n-1) \\ \theta(n-1) \\ v_r(n-1) \\ v_\theta(n-1) \end{bmatrix} \tag{2.10}$$

Observation Equation:

$$\mathbf{y}(n) = \mathbf{C}(n)\mathbf{x}(n) + \mathbf{v}(n) \tag{2.11}$$

$$\begin{bmatrix} r^m(n) \\ \theta^m(n) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} r(n) \\ \theta(n) \\ v_r(n) \\ v_\theta(n) \end{bmatrix} + \mathbf{v}(n) \tag{2.12}$$

with $\mathbf{Q}_v(n) = \begin{bmatrix} 2500 & 0 \\ 0 & 16 \end{bmatrix}$

Computation: For $n = 1, 2, \ldots$ compute

$$\hat{\mathbf{x}}(n|n-1) = \mathbf{A}(n-1)\hat{\mathbf{x}}(n-1|n-1) \tag{2.13}$$

$$\mathbf{P}(n|n-1) = \mathbf{A}(n-1)\mathbf{P}(n-1|n-1)\mathbf{A}^T(n-1) + \mathbf{Q}_w(n) \tag{2.14}$$

$$\mathbf{K}(n) = \mathbf{P}(n|n-1)\mathbf{C}^T(n)[\mathbf{C}(n)\mathbf{P}(n|n-1)\mathbf{C}^T(n) + \mathbf{Q}_v(n)]^{-1} \tag{2.15}$$

$$\hat{\mathbf{x}}(n|n) = \hat{\mathbf{x}}(n|n-1) + \mathbf{K}(n)[\mathbf{y}(n) - \mathbf{C}(n)\hat{\mathbf{x}}(n|n-1)] \tag{2.16}$$

$$\mathbf{P}(n|n) = [\mathbf{I} - \mathbf{K}(n)\mathbf{C}(n)]\mathbf{P}(n|n-1) \tag{2.17}$$

These equations implement the recursive Kalman filter algorithm, which starts with initial state estimates and proceeds sequentially through time to provide optimal state estimates based on the available measurements.

(d) State Estimation with Kalman Filter

We now apply the Kalman Filter to estimate the position and velocity of a moving object using noisy range and angle data provided in the `range1` and `angle1` files. We assume the filter starts with an initial state vector $[500, 0, 0, 0]^T$ and an initial error covariance matrix diag($[1000, 10, 50, 10]$). We have to plot the estimated trajectory and velocities over time. Additionally, we have to study the effect of varying the initial error covariance matrix on filter performance.

Using the data from `angle1.mat`, it's given to us that the data is in degrees, but the trigonometric functions in MATLAB use radians as their default arguments, so we need to convert the data to radians. **Most importantly**, we need to change the noise variance factor 16 which is in degree-squared to radian squared too.

So we simply implement the Kalman filter on the polar co-ordinates and then convert it into the Cartesian co-ordinates post filtering, to plot the estimated trajectory and the radial and angular velocities!

The part of the script concerned with it is given below:

```
% Kalman Filter Implementation for tracking Siva's car
close all; clear all; clc;
load('range1.mat');
load('angle1.mat');
angle1_rad = deg2rad(angle1);
dt = 1;

% Initial state and covariance
x = [500; 0; 0; 0]; % [r; theta; vr; vtheta]
P = diag([1000, 10, 50, 10]);

% Process noise covariance (zero for constant velocities)
Q_w = zeros(4, 4);

% Measurement noise covariance
angle_var_rad = (deg2rad(4))^2;
Q_v = diag([2500, angle_var_rad]);

% State transition matrix
A = [1, 0, dt, 0;
     0, 1, 0, dt;
     0, 0, 1, 0;
     0, 0, 0, 1];

% Observation matrix
C = [1, 0, 0, 0;
     0, 1, 0, 0];

% Storage for results
x_history = zeros(4, length(range1));
x_history(:, 1) = x;

% Kalman filter loop
for n = 2:length(range1)
    % prediction
    x_pred = A * x;
    P_pred = A * P * A' + Q_w;

    % update
    y = [range1(n); angle1_rad(n)];
    y_diff = y - C * x_pred;
    S = C * P_pred * C' + Q_v;
    K = P_pred * C' * inv(S);

    x = x_pred + K * y_diff;
    P = (eye(4) - K * C) * P_pred;

    % store results
    x_history(:, n) = x;
end

% Convert to Cartesian coordinates for plots
x_cart = x_history(1,:) .* cos(x_history(2,:));
y_cart = x_history(1,:) .* sin(x_history(2,:));

% Plot estimated trajectory
figure;
plot(x_cart, y_cart);
hold on;
plot(0, 0, 'ko', 'MarkerFaceColor', 'k');  % Origin (Madhuri's position)
xlabel('X position (m)');
ylabel('Y position (m)');
title('Estimated Trajectory of Siva''s Car');
legend('Estimated Trajectory', 'Origin (Madhuri''s Position)');
axis equal;
grid on;

% Plot estimated velocities
figure;
subplot(2,1,1);
plot(0:length(range1)-1, x_history(3,:));
ylabel('Radial velocity (m/s)');
title('Estimated Radial Velocity');
grid on;

subplot(2,1,2);
plot(0:length(range1)-1, rad2deg(x_history(4,:))); % Convert to deg/s
ylabel('Angular velocity (deg/s)');
xlabel('Time (s)');
title('Estimated Angular Velocity');
grid on;
```

Figure 2.3: Script for estimating trajectories using Kalman filtering

This gives us the estimated plots as provided below. I've also included the initial given estimates of state, `[500,0,0,0]` and the error covariance matrix, `diag([1000, 10, 50, 10])`

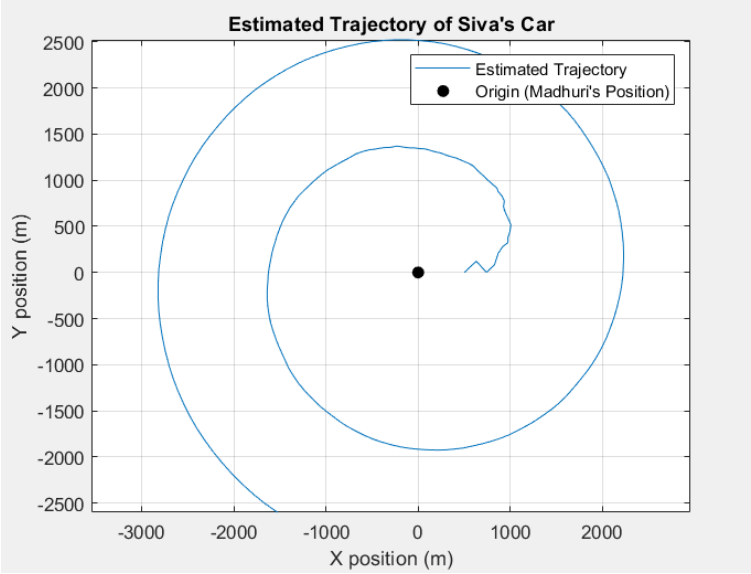


Figure 2.4: Estimated Trajectory

Similarly, I've plotted the estimated radial and angular velocities too below.

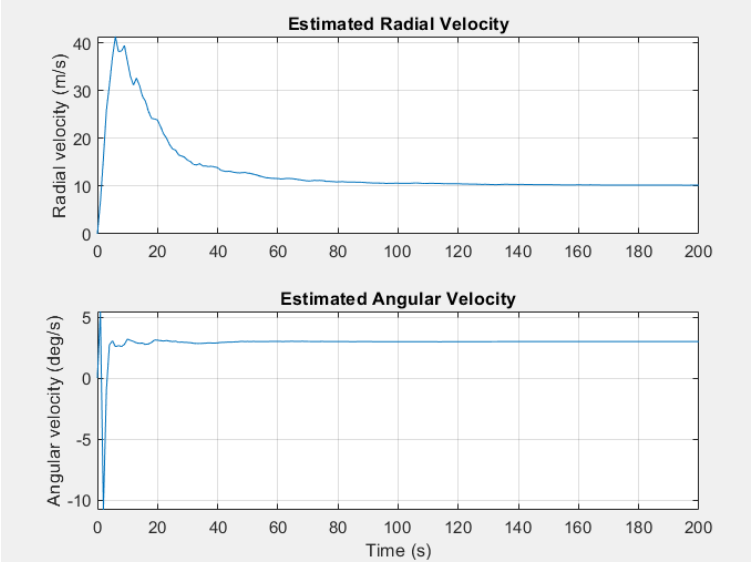


Figure 2.5: Estimated Velocities

We see that the velocities do converge to the true value, and thus can safely say that the trajectory too does converge to the true path, although I haven't included the true path in Figure 2.4.

We now try to vary the error covariance matrix initialization by scaling the given diagonal matrix itself. Let's try to understand this intuitively!

What does the Kalman filter do? We know that the Kalman filter, in it's update stage effectively weights the observations and the predictions. With time, if our Kalman filter is performing well we expect it to make the predictions stronger and stronger, and depend less on the incoming observations, while using it to make the prediction itself stronger!

This makes the convergence also depend on the initial state that we start filtering with! This very concept creates the two cases, one wherein say we use the erroneous estimate as given to us, and another wherein we use the true state itself as the initial estimate.

In the first case, we expect that since the *error covariance matrix is high*, and our estimate of the state is also not good enough, we rely on the observations initially, which leads to **increase of the convergence rate**, (note that the weight associated with the measurements is the Kalman Gain).

Simply enough, initially since our prediction is highly erroneous we try to weigh the observations more, and later as the prediction gets better and better we ofcourse reduce the weight for the observation in the update stage! This gives the intuition that larger the error-covariance matrix, faster is the convergence rate!

This can also be seen in the plots! I've plotted the convergence of the parameter-Radial velocity with iterations for three scales of the given covariance matrix, i.e. `[0.1, 1, 10]` and also the variation of the Kalman gain with the iterations below.

The script for this part is provided below.

```
% effect of changing initial covariance
cov_scales = [0.1, 1, 10];
colors = {'r', 'g', 'b'};
labels = cell(length(cov_scales), 1);

figure;
hold on;
for i = 1:length(cov_scales)
    scale = cov_scales(i);
    P_init = diag([1000, 10, 50, 10]) * scale;

    % Reset state and initialize covariance with new scale
    x_alt = [500; 0; 0; 0];
    P_alt = P_init;

    % store alternative state history
    x_alt_history = zeros(4, length(range1));
    x_alt_history(:, 1) = x_alt;

    % Run Kalman filter with new covariance
    for n = 2:length(range1)
        % prediction
        x_pred = A * x_alt;
        P_pred = A * P_alt * A' + Q_m;

        % update
        y = [range1(n); angle1_rad(n)];
        y_diff = y - C * x_pred;

        S = C * P_pred * C' + Q_v;
        K = P_pred * C' / S;

        x_alt = x_pred + K * y_diff;
        P_alt = (eye(4) - K * C) * P_pred;

        % store results
        x_alt_history(:, n) = x_alt;
    end

    % Cartesian coordinates
    x_alt_cart = x_alt_history(1,:) .* cos(x_alt_history(2,:));
    y_alt_cart = x_alt_history(1,:) .* sin(x_alt_history(2,:));

    % Plot trajectory with different covariance scale
    plot(x_alt_cart, y_alt_cart, colors{i});
    labels{i} = ['Scale = ', num2str(scale)];
end

plot(0, 0, 'ko', 'MarkerFaceColor', 'k');  % Origin
grid on;
xlabel('X position (m)');
ylabel('Y position (m)');
title('Effect of Initial Covariance on Estimated Trajectory');
legend([labels, ['Origin']]);
axis equal;

% Plot velocity convergence with different covariances
figure;
subplot(2,1,1);
hold on;
for i = 1:length(cov_scales)
    scale = cov_scales(i);
    P_init = diag([1000, 10, 50, 10]) * scale;

    % Reset state and initialize covariance
    x_alt = [500; 0; 0; 0];
    P_alt = P_init;

    % store alternative results
    x_alt_history = zeros(4, length(range1));
    x_alt_history(:, 1) = x_alt;

    % Run Kalman filter with new covariance
    for n = 2:length(range1)
        % prediction
        x_pred = A * x_alt;
        P_pred = A * P_alt * A' + Q_m;

        % update
        y = [range1(n); angle1_rad(n)];
        y_diff = y - C * x_pred;
        S = C * P_pred * C' + Q_v;
        K = P_pred * C' / S;

        x_alt = x_pred + K * y_diff;
        P_alt = (eye(4) - K * C) * P_pred;

        % store results
        x_alt_history(:, n) = x_alt;
    end

    plot(0:length(range1)-1, x_alt_history(3,:), colors{i});
end
grid on;
xlabel('Time (seconds)');
ylabel('Radial Velocity (m/s)');
title('Effect of Initial Covariance on Radial Velocity Estimate');
legend(labels);

% Run another time to collect weights
% for Kalman gain
K_vr_range_history = zeros(length(cov_scales), length(range1));

for i = 1:length(cov_scales)
    scale = cov_scales(i);
    P_init = diag([1000, 10, 50, 10]) * scale;

    % Reset state and initialize covariance with new scale
    x_alt = [500; 0; 0; 0];
    P_alt = P_init;

    % Run Kalman filter with new covariance and collect coefficients
    for n = 2:length(range1)
        % Prediction step
        x_pred = A * x_alt;
        P_pred = A * P_alt * A' + Q_m;

        % Measurement update
        y = [range1(n); angle1_rad(n)];
        y_diff = y - C * x_pred;

        S = C * P_pred * C' + Q_v;
        K = P_pred * C' / S;
        IKC = eye(4) - K * C;

        % Store K and I-KC values for radial velocity
        K_vr_range_history(i,n) = K(3,1);  % Effect of range measurement on vr

        x_alt = x_pred + K * y_diff;
        P_alt = IKC * P_pred;
    end
end

% Plot Kalman gain for radial velocity
subplot(2,1,2);
hold on;
for i = 1:length(cov_scales)
    plot(0:length(range1)-1, K_vr_range_history(i,:), colors{i});
end
grid on;
xlabel('Time (seconds)');
ylabel('K_{v_r,r}');
title('Kalman Gain: Range Measurement → Radial Velocity State');
legend(labels);
```

Figure 2.6: Script for varied error covariance matrix

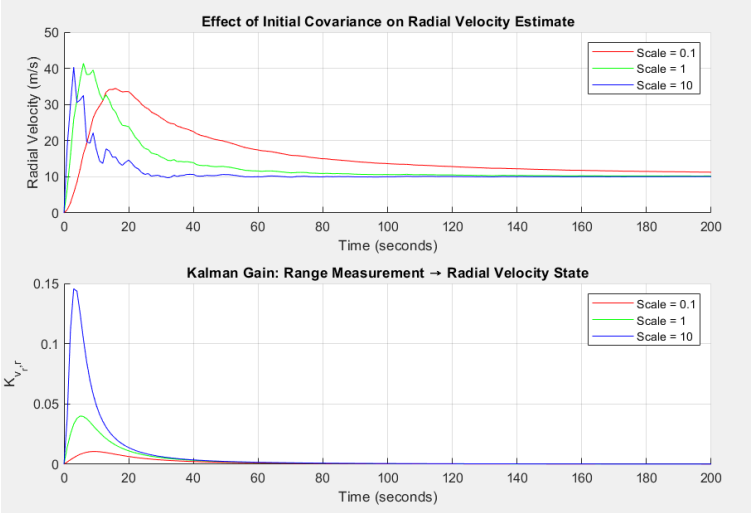


Figure 2.7: Effect of initial covariance matrix on the radial velocity, with the Kalman gain

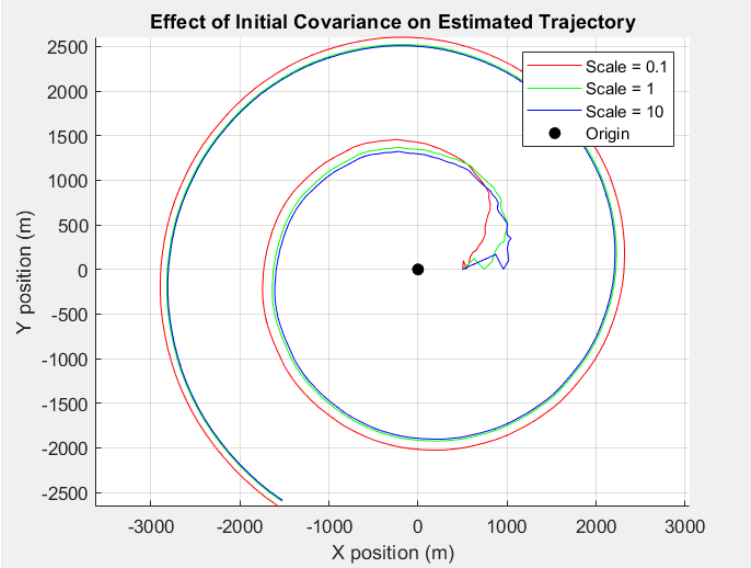


Figure 2.8: Effect of initial covariance matrix on the trajectory

We can clearly see that the dependence on the observations increases heavily for the case of higher covariance matrix, thus it converges faster.

The second case is slightly different compared to this! Since our error covariance matrix is high, we initially expect the Kalman gain to rise faster and fall compared to the ones with lesser error covariance! This is the case if we have a better estimate of the initial state. In this case, I've considered the actual true state as the initial state. This is hypothetically a good initialization point for the state estimate.

Now, the more the covariance, higher it goes up and then goes down as mentioned already, but here it slows down the convergence for higher error covariance because we're ideally supposed to trust the estimates *right off the bat* and not the observations! But the higher error covariance still supports the observations initially, so the convergence becomes slower.

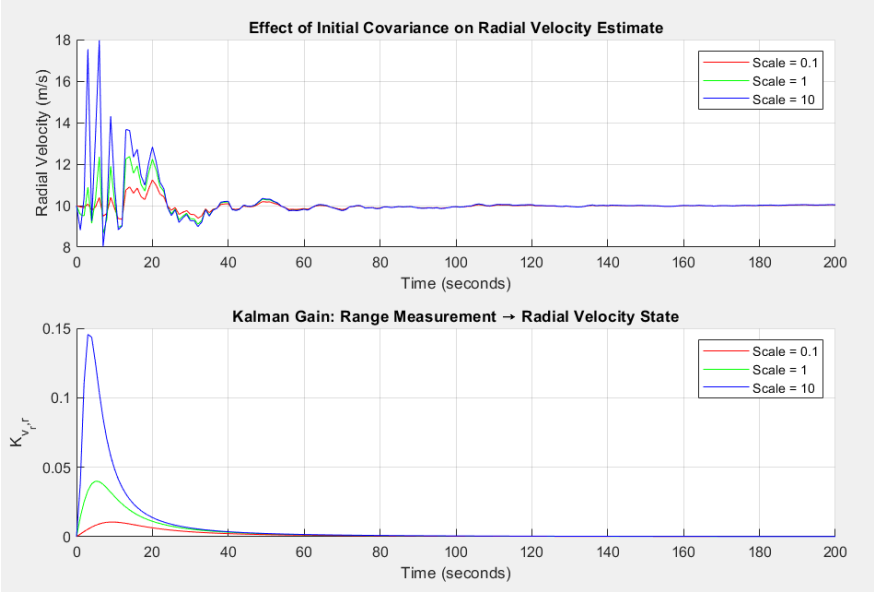


Figure 2.9: Effect of initial covariance matrix on the radial velocity, with a better state estimate

This supports our intuition as we've discussed above!

(e) Usage of Independent Sensors for better(?) estimate

We're now given another interesting scenario that Madhuri, in an attempt to improve her estimation accuracy, has purchased another set of range and angle sensors with same error characteristics but independent of existing sensors.

From intuition, we can guess that yes, using two independent sets of sensors with the same error characteristics will indeed improve estimation accuracy. Independent sensors provide independent measurements of the same quantity (range and angle), each with its own random noise. So, when we average or combine these independent measurements, the error tends to "cancel" (not really fully cancel out though) and give a better accurate estimate.

Let's consider the case of dependent sensors. In the extreme case where both sensors produce identical outputs, including identical noise, combining their measurements is equivalent to scaling a single sensor's output. This results in doubling both the signal and the noise, which means the signal to noise ratio (SNR) remains unchanged. In such a case, using both sensors provides no advantage over simply amplifying the output of one. That is, you could have as well amplified a single sensor output!

Now consider **independent sensors** that aim to measure the same underlying quantity. Suppose each sensor observes:

$$y_1 = a + n_1, \quad y_2 = a + n_2$$

where a is the true value and n_1, n_2 are independent noise terms with equal variance.

By averaging the two measurements:

$$\frac{y_1 + y_2}{2} = a + \frac{n_1 + n_2}{2}$$

Since the noise terms are independent and have the same variance σ^2, the variance of their

average becomes:

$$\text{Var}\left(\frac{n_1 + n_2}{2}\right) = \frac{\sigma^2}{2}$$

Thus, the **noise variance is halved** (or the noise power is halved! SNR is better!), which means the estimate of a is **more accurate**. This is the key advantage of using **independent sensors**: combining their measurements **reduces uncertainty**.

In contrast, if the sensors were dependent, then the noise variance doesn't reduce that much And hence gain in less!

This can be justified by the plots, by using the set of sensors given by data `range2.mat` and `angle2.mat`.

(f) Implementing dual set of sensors for better estimation

As mentioned above, intuitionally we feel that the estimate does get better. Let's verify this. In combining the sensor data, basically our observation vector $\mathbf{y}$ is now a 4x1 vector, with the observation model matrix $\mathbf{C}$ being 4x4 in dimension.

In the script, I've plotted the diagonal entries of a covariance matrix, which is the variance (or uncertainty) associated with each individual state variable over time. Most of the script is repetitive to compare with the single sensor case.

The script for the single sensor calculations is provided below:

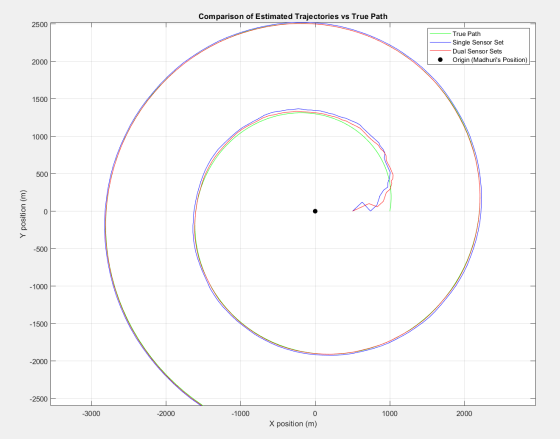
The plots are provided below.

```
% single sensor vs dual sensors
clear all; close all; clc;

% Load sensor data
load('range1.mat');
load('angle1.mat');
load('range2.mat');
load('angle2.mat');

angle1_rad = deg2rad(angle1);
angle2_rad = deg2rad(angle2);
dt = 1;

%% Calculate True Path
vr = 10;            % radial vel
vtheta = 3;         % angular vel
vtheta_rad = vtheta * pi/180;
T = 200;            % total time
initial_pos = [1000, 0];  % initial pos

% one point per second
t = 0:1:T;

% position arrays
x_true = zeros(size(t));
y_true = zeros(size(t));

% initial position
x_true(1) = initial_pos(1);
y_true(1) = initial_pos(2);

% trajectory calc
for i = 2:length(t)
    % current radius and angle
    r_prev = sqrt(x_true(i-1)^2 + y_true(i-1)^2);
    theta_prev = atan2(y_true(i-1), x_true(i-1));

    % Update
    r_new = r_prev + vr;
    theta_new = theta_prev + vtheta_rad;

    % Cartesian coordinates
    x_true(i) = r_new * cos(theta_new);
    y_true(i) = r_new * sin(theta_new);
end

% Calculate true state (r, theta, vr, vtheta) for each time point
r_true = sqrt(x_true.^2 + y_true.^2);
theta_true = atan2(y_true, x_true);
vr_true = ones(size(t)) * vr;
vtheta_true = ones(size(t)) * vtheta_rad;

%% Single Sensor Implementation
% Initial state and covariance
x_single = [500; 0; 0; 0]; % [r; theta; vr; vtheta]
P_single = diag([1000, 10, 50, 10]);

% Process noise covariance
Q_w = zeros(4, 4);

% Measurement noise covariance
angle_var_rad = (deg2rad(4))^2; % Convert to radians^2
Q_v_single = diag([2500, angle_var_rad]);

% State transition matrix
A = [1, 0, dt, 0;
     0, 1, 0, dt;
     0, 0, 1, 0;
     0, 0, 0, 1];

% Observation matrix for single sensor
C_single = [1, 0, 0, 0;
            0, 1, 0, 0];

% Storage for results
x_history_single = zeros(4, length(range1));
x_history_single(:, 1) = x_single;
P_diag_history_single = zeros(4, length(range1));
P_diag_history_single(:, 1) = diag(P_single);

% Kalman filter loop - Single sensor
for n = 2:length(range1)
    % prediction
    x_pred = A * x_single;
    P_pred = A * P_single * A' + Q_w;

    % update
    y = [range1(n); angle1_rad(n)];
    y_diff = y - C_single * x_pred;
    S = C_single * P_pred * C_single' + Q_v_single;
    K = P_pred * C_single' / S;

    x_single = x_pred + K * y_diff;
    P_single = (eye(4) - K * C_single) * P_pred;

    % Store results
    x_history_single(:, n) = x_single;
    P_diag_history_single(:, n) = diag(P_single);
end
```

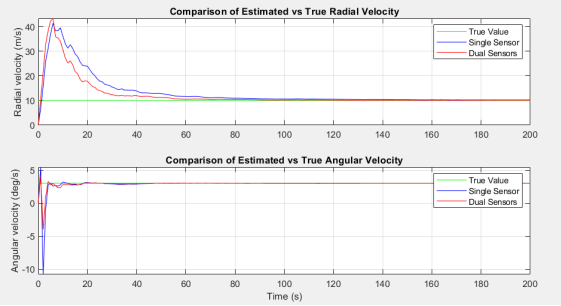
Figure 2.10: Script for single sensor

```
%% Dual Sensor Implementation
% Initial state and covariance
x_dual = [500; 0; 0; 0]; % [r; theta; vr; vtheta]
P_dual = diag([1000, 10, 50, 10]);

% Measurement noise covariance for both sensor sets
Q_v_dual = diag([2500, angle_var_rad, 2500, angle_var_rad]);

% Observation matrix for both sensor sets
C_dual = [1, 0, 0, 0;
          0, 1, 0, 0;
          1, 0, 0, 0;
          0, 1, 0, 0];


x_history_dual = zeros(4, length(range1));
x_history_dual(:, 1) = x_dual;
P_diag_history_dual = zeros(4, length(range1));
P_diag_history_dual(:, 1) = diag(P_dual);

% Kalman filter loop - Dual sensors
for n = 2:length(range1)
    % Prediction
    x_pred = A * x_dual;
    P_pred = A * P_dual * A' + Q_w;

    % update with both sensor sets
    y = [range1(n); angle1_rad(n); range2(n); angle2_rad(n)];
    y_diff = y - C_dual * x_pred;
    S = C_dual * P_pred * C_dual' + Q_v_dual;
    K = P_pred * C_dual' / S;

    x_dual = x_pred + K * y_diff;
    P_dual = (eye(4) - K * C_dual) * P_pred;

    % Store results
    x_history_dual(:, n) = x_dual;
    P_diag_history_dual(:, n) = diag(P_dual);
end

% Convert to Cartesian for plotting
x_cart_single = x_history_single(1,:) .* cos(x_history_single(2,:));
y_cart_single = x_history_single(1,:) .* sin(x_history_single(2,:));

x_cart_dual = x_history_dual(1,:) .* cos(x_history_dual(2,:));
y_cart_dual = x_history_dual(1,:) .* sin(x_history_dual(2,:));

% Error Calculation compared to true path
% For single sensor
error_r_single = abs(x_history_single(1,:) - r_true);
error_theta_single = abs(wrapToPi(x_history_single(2,:) - theta_true));
error_vr_single = abs(x_history_single(3,:) - vr_true);
error_vtheta_single = abs(x_history_single(4,:) - vtheta_true);

error_pos_single = sqrt((x_cart_single - x_true).^2 + (y_cart_single - y_true).^2);

% For dual sensors
error_r_dual = abs(x_history_dual(1,:) - r_true);
error_theta_dual = abs(wrapToPi(x_history_dual(2,:) - theta_true));
error_vr_dual = abs(x_history_dual(3,:) - vr_true);
error_vtheta_dual = abs(x_history_dual(4,:) - vtheta_true);

error_pos_dual = sqrt((x_cart_dual - x_true).^2 + (y_cart_dual - y_true).^2);

% Calculate RMS errors
rmse_pos_single = sqrt(mean(error_pos_single.^2));
rmse_r_single = sqrt(mean(error_r_single.^2));
rmse_theta_single = sqrt(mean(error_theta_single.^2));
rmse_vr_single = sqrt(mean(error_vr_single.^2));
rmse_vtheta_single = sqrt(mean(error_vtheta_single.^2));

rmse_pos_dual = sqrt(mean(error_pos_dual.^2));
rmse_r_dual = sqrt(mean(error_r_dual.^2));
rmse_theta_dual = sqrt(mean(error_theta_dual.^2));
rmse_vr_dual = sqrt(mean(error_vr_dual.^2));
rmse_vtheta_dual = sqrt(mean(error_vtheta_dual.^2));

% Calculate MAE
mae_pos_single = mean(error_pos_single);
mae_r_single = mean(error_r_single);
mae_theta_single = mean(error_theta_single);
mae_vr_single = mean(error_vr_single);
mae_vtheta_single = mean(error_vtheta_single);

mae_pos_dual = mean(error_pos_dual);
mae_r_dual = mean(error_r_dual);
mae_theta_dual = mean(error_theta_dual);
mae_vr_dual = mean(error_vr_dual);
mae_vtheta_dual = mean(error_vtheta_dual);
```

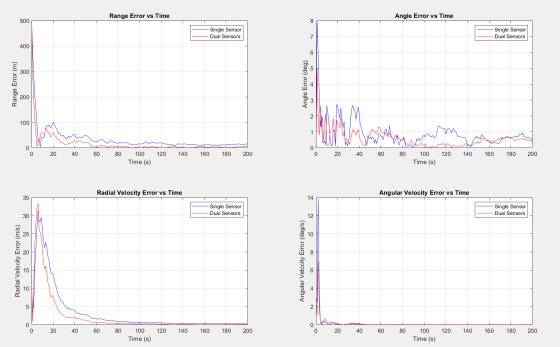
Figure 2.11: Script for dual sensors with error computation



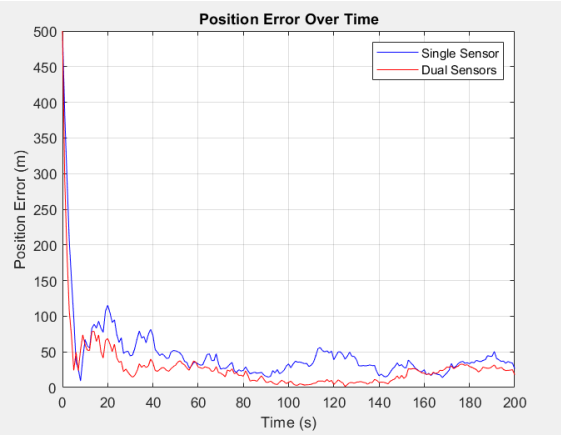
(a) Trajectories



(b) True vs Estimated



(c) Errors over time



(d) Position error over time

Figure 2.12: Comparing single and dual sensor cases

In all these plots we can see that the case of the dual sensor is better like our intuition since it provides a better and a more accurate estimate of the true state by reducing the noise power/variance.

```
Mean Absolute Error (MAE):
                        Single Sensor | Dual Sensors
Position Error (m):             46.27 |       27.82
Range Error (m):                32.64 |       16.15
Angle Error (deg):               0.86 |        0.57
Radial Vel Error (m/s):          3.29 |        2.30
Angular Vel Error (deg/s):       0.15 |        0.10

Steady-state position error (last 50 time steps):
Single Sensor: 30.98 m
Dual Sensors: 25.23 m
```

Figure 2.13: Terminal output showing MAE for the two cases.

For additional presentation, I've also added the mean absolute errors comparison for each state and we observe that the estimation using dual sensors is better!