Moving Average Filter vs Blackmann Filter

Nathan Kim

January 2025

1 Project Description

The goal of this project is to implement a noise-reduction digital signal processing algorithm optimized for embedded systems, with a focus on performance, efficiency, and practical implementation in C. The project also analyzes the trade-offs and benefits of the chosen algorithm in terms of computational complexity, accuracy, and real-world usability.

The algorithm selected for this project is the **moving average filter**, a fundamental tool in digital signal processing. Its performance will be compared against more sophisticated filters, such as the **Gaussian filter** and the **Blackman filter**, to highlight its strengths and limitations in different contexts.

The moving average filter works by averaging a specified number of points from the input signal to produce each point in the output signal. Mathematically, the filter is represented as:

$$y[n] = \frac{1}{N+M+1} \sum_{k=-N}^{M} x[n-k]$$

Mathematical Representation of the Moving Average Filter.

Here, $M+N+1$ represents the number of points included in the average, x is the input signal, and y is the resulting output signal. The moving average filter is equivalent to a **convolution with a rectangular pulse**, where the pulse has an area of one and a width corresponding to the number of samples averaged. Intuitively, the moving average filter smooths out signals by averaging multiple noisy samples, effectively reducing random variations. A simple, unweighted average is optimal in this context because all samples contribute equally to the noise, making no single sample more significant than another.

Key Strengths of the Moving Average Filter:

- Only addition and subtraction operations are needed; no multiplication is required.
- It has simple logic.
- It requires minimal memory, as seen in the code.

In the following sections, the performance of this filter will be analyzed in both the time and frequency domains. Its trade-offs, including computational complexity and smoothing effects, will be discussed, along with comparisons to more advanced filters.

2 Filter Results and Analysis

The frequency response of the moving average filter is shown below. Figure 2 demonstrates the original generated noisy signal compared to the clean signal. Figure 3 overlays the filtered signal onto the original noise signal to show the smoothing effect. Figure 4 shows the filter's magnitude response in the frequency domain, calculated as the Fourier transform of the rectangular pulse:

$$H(e^{j\omega}) = \frac{1}{N+M+1} \frac{\sin\left[\omega(M+N+1)/2\right]}{\sin(\omega/2)} e^{j\omega[(N-M)/2]}.$$

Frequency Response Equation.

2.1 Frequency Response Analysis

The frequency response, as seen in Figure 4, reveals that the moving average filter is relatively ineffective as a low-pass filter due to its slow roll-off and poor attenuation of higher frequencies. However, as the number of samples averaged increases, the filter performs better by smoothing the signal more effectively and attenuating noise, as seen in Figures 5 through 7.

Analysis Across Figures: Both the time and frequency domain results highlight the trade-offs of the moving average filter. While the filter effectively smooths signals, its frequency response shows that it is not an ideal low-pass filter due to its limited ability to attenuate high frequencies. Larger sample windows improve noise reduction (visible in Figures 6 and 7) but increase computational complexity and result in over-smoothing, which might cause the signal to lose its original characteristics.

3 Efficient Implementation on Embedded Systems

One of the key advantages of the moving average filter is its simplicity and computational efficiency, particularly in the context of embedded systems. Using overlapping calculations between consecutive output points, the filter can be implemented using a recursive approach, significantly improving its performance.

Consider the calculation of two adjacent output points, $y[50]$ and $y[51]$:

$$y[50] = x[47] + x[48] + x[49] + x[50] + x[51] + x[52] + x[53],$$

$$y[51] = x[48] + x[49] + x[50] + x[51] + x[52] + x[53] + x[54].$$

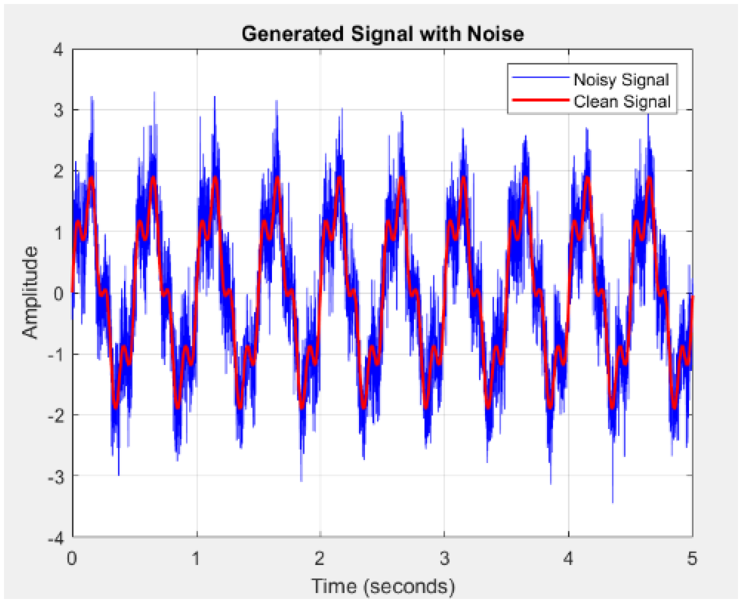


Figure 1: Generated Signal with Noise (Noisy and Clean Signals)

Observing that many terms overlap between consecutive calculations, the result can be updated recursively:

$$y[51] = y[50] + x[54] - x[47].$$

3.1 Time Complexity Analysis

The recursive implementation reduces computational overhead compared to the naive approach:

- **Non-optimized Version:** In the naive implementation, each output value requires summing M samples, and this operation is repeated N times (for N output points). This results in a time complexity of:

$$O(M \cdot N).$$

- **Optimized (Recursive) Version:** Each output point is computed using three operations: adding the incoming sample, subtracting the outgoing sample, and dividing by M. This results in a time complexity of:

$$O(N).$$

The dependency on M is eliminated because the sum is updated incrementally, making the algorithm highly efficient for large window sizes.

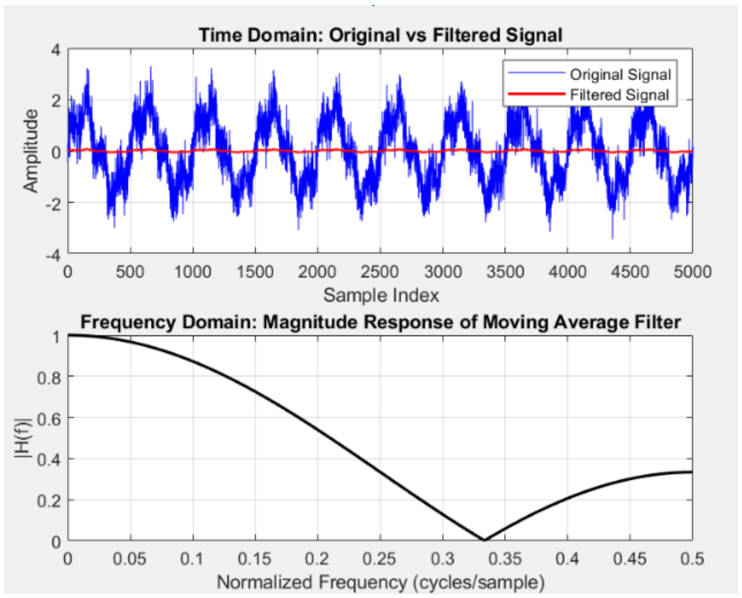


Figure 2: Time Domain (Original vs Filtered Signal) with Frequency Response for 3-Point Filter

3.2 Code Implementations

The following figures illustrate the algorithm implemented in C and MATLAB for both the naive and optimized versions:

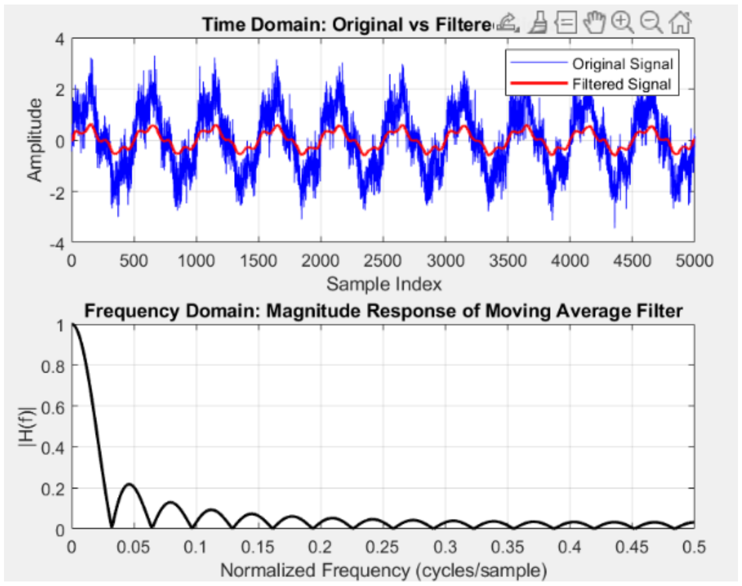


Figure 3: Time Domain (Original vs Filtered Signal) with Frequency Response for 31-Point Filter

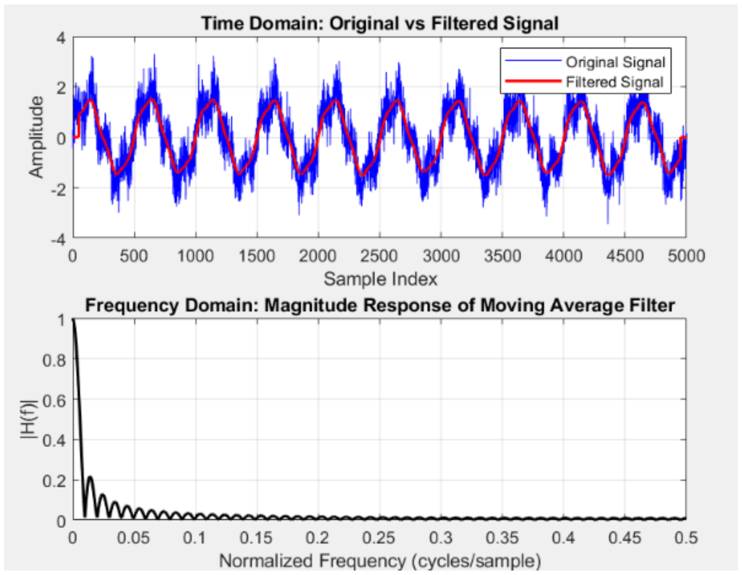
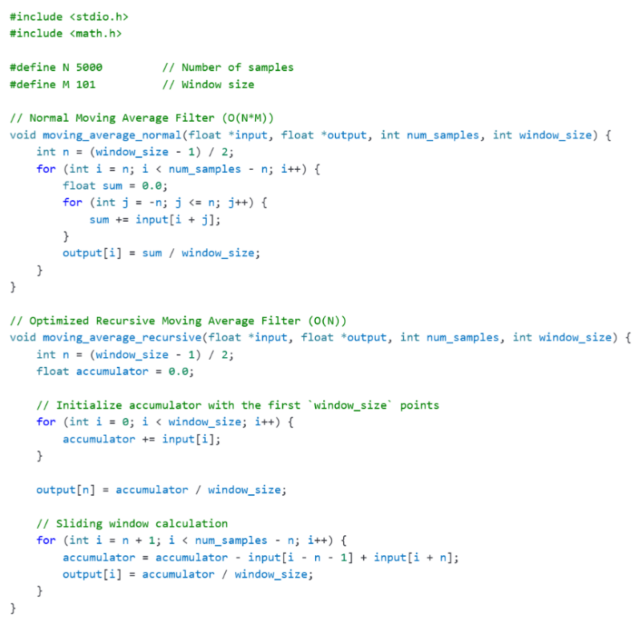
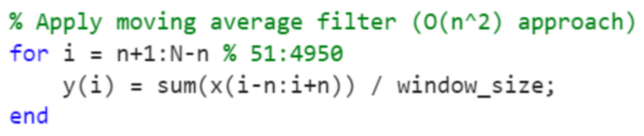


Figure 4: Time Domain (Original vs Filtered Signal) with Frequency Response for 101-Point Filter



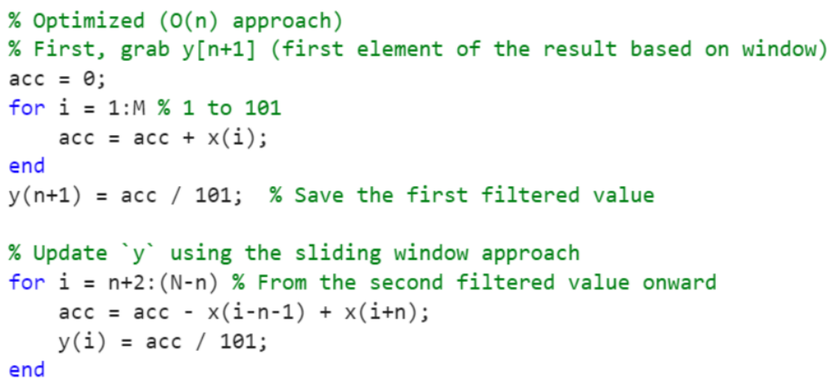
```c
#include <stdio.h>
#include <math.h>

#define N 5000         // Number of samples
#define M 101          // Window size

// Normal Moving Average Filter (O(N*M))
void moving_average_normal(float *input, float *output, int num_samples, int window_size) {
    int n = (window_size - 1) / 2;
    for (int i = n; i < num_samples - n; i++) {
        float sum = 0.0;
        for (int j = -n; j <= n; j++) {
            sum += input[i + j];
        }
        output[i] = sum / window_size;
    }
}

// Optimized Recursive Moving Average Filter (O(N))
void moving_average_recursive(float *input, float *output, int num_samples, int window_size) {
    int n = (window_size - 1) / 2;
    float accumulator = 0.0;

    // Initialize accumulator with the first `window_size` points
    for (int i = 0; i < window_size; i++) {
        accumulator += input[i];
    }

    output[n] = accumulator / window_size;

    // Sliding window calculation
    for (int i = n + 1; i < num_samples - n; i++) {
        accumulator = accumulator - input[i - n - 1] + input[i + n];
        output[i] = accumulator / window_size;
    }
}
```

Figure 5: C Implementation of the Moving Average Filter.



```matlab
% Apply moving average filter (O(n^2) approach)
for i = n+1:N-n % 51:4950
    y(i) = sum(x(i-n:i+n)) / window_size;
end
```

Figure 6: Naive Implementation in MATLAB.



```matlab
% Optimized (O(n) approach)
% First, grab y[n+1] (first element of the result based on window)
acc = 0;
for i = 1:M % 1 to 101
    acc = acc + x(i);
end
y(n+1) = acc / 101;  % Save the first filtered value

% Update `y` using the sliding window approach
for i = n+2:(N-n) % From the second filtered value onward
    acc = acc - x(i-n-1) + x(i+n);
    y(i) = acc / 101;
end
```

Figure 7: Optimized Implementation in MATLAB.

4 Comparison Between Moving Average and Blackman Filters

To better understand the trade-offs between the moving average filter and the Blackman filter, we analyze their time and frequency domain performance, computational complexities, and implementation intricacies. The table below highlights their key differences and use cases.

4.1 Performance Analysis

The time-domain plots (Figure 8) reveal that the Blackman filter preserves the signal's shape more effectively, with less oversmoothing compared to the Moving Average filter. However, this precision comes at a cost of higher computational complexity due to convolution operations.

In the frequency domain, the Blackman filter exhibits superior low-pass characteristics, with sharper roll-off and stronger sidelobe attenuation (Figure 9). The Moving Average filter, while computationally faster, has a slower roll-off and poor attenuation of high-frequency noise, making it less effective in applications requiring precise frequency-domain filtering.

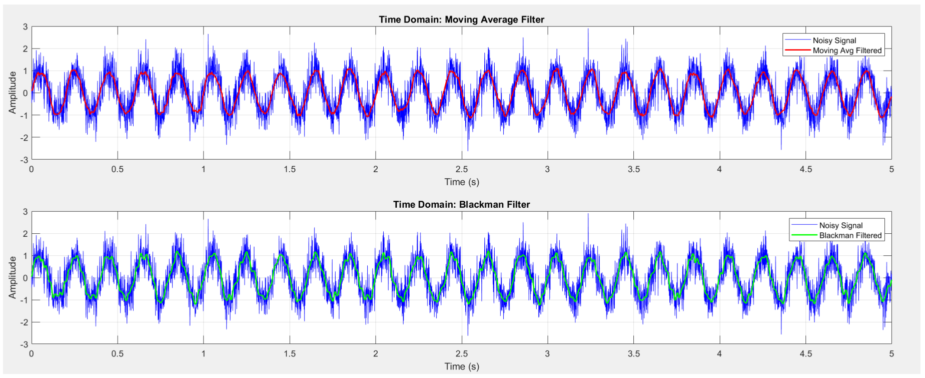


Figure 8: Time Domain: Comparison of Moving Average and Blackman Filters

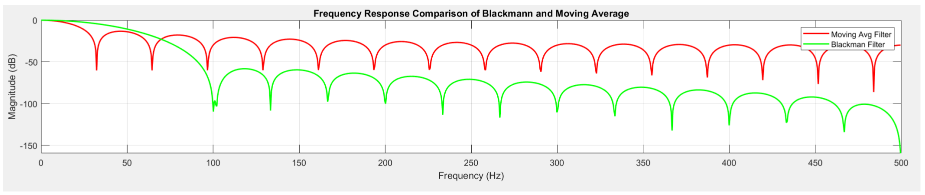


Figure 9: Frequency Response: Moving Average vs. Blackman Filter (Unzoomed)

4.2 Computational Complexity Comparison

The Moving Average filter uses a recursive algorithm, making it computationally efficient, while the Blackman filter relies on convolution, which is computationally intensive.

Moving Average Filter Pseudocode:

```
Initialize accumulator to 0
For each point:
    Update accumulator: subtract outgoing sample, add incoming sample
    Compute average: accumulator / window_size
```

Blackman Filter Pseudocode:

```
Generate Blackman coefficients: w = blackman(window_size)
Normalize coefficients: w = w / sum(w)
For each point:
    Perform convolution: filtered_signal = conv(signal, w, 'same')
```

Convolution is more computationally complex than simple addition. The Blackman filter requires $O(N \cdot M)$ time, where M is the filter length, since M samples must be summed for every output N.

4.3 Applications and Use Cases

Feature	**Moving Average Filter**	**Blackman Filter**
Computational Speed	Computationally efficient due to recursive updates; ideal for real-time applications.	Computationally slower due to convolution and weighting coefficients.
Noise Reduction	Effective for random noise but less precise.	Superior noise reduction with minimal spectral leakage.
Frequency Filtering	Poor low-pass characteristics; slow roll-off.	Excellent low-pass characteristics; sharp roll-off and strong sidelobe attenuation.
Time Domain	Oversmooths signals, losing details.	Retains finer details due to advanced weighting.
Implementation Complexity	Simple; relies on basic addition and subtraction.	Moderate; requires generating and normalizing coefficients.
Applications	Real-time, resource-constrained systems needing quick smoothing.	High-precision tasks requiring strong frequency-domain filtering.

Table 1: Comparison Between Moving Average and Blackman Filters