Gabriel Devereux

Email: gabjol@me.com Phone: +61 487 049 009 Website: https://infinityvision.tv

INFINITY LIMITED

Analysing IMU Noise:

Gyroscope Noise Simulation and Allan Deviation Analysis

This document demonstrates an understanding of gyroscope noise characteristics by simulating Angle Random Walk (ARW), Rate Random Walk (RRW), and Bias Instability (BI) using a Python script. It also includes computing Allan Deviation and Euler integration in C++ and MATLAB, then analysing noise characteristics using MATLAB.

Table of Contents

Gabriel Devereux

Email: gabjol@me.com Phone: +61 487 049 009 Website: https://infinityvision.tv

INFINITY LIMITED

1. Basic simulation of ARW, RRW and BI in python.

The script 1GenerateGyroValues.py simulates the Angle Random Walk (ARW), Rate Random Walk (RRW), and Bias Instability (BI) characteristics of a MEMS gyroscope under static conditions. It generates 100,000 XYZ gyroscope samples at a frequency of 100 Hz over a 1000-second interval. The resulting simulated data, representing gyroscope readings in three axes (X, Y, Z), is written to a CSV file (gyro_data.csv) for further analysis.

1.1 Simulating ARW

Angle Random Walk (ARW) represents the accumulation of random noise (specifically, Gaussian white noise) in angular velocity measurements over time. When these noisy rate measurements are integrated, they **drift** in the calculated angle.

The greater the noise in the rate measurements, the greater the error accumulation over time, leading to more considerable drift in the angle estimate.

```
    # Noise generation parameters
    self.arw_std = 0.01  # Angle random walk deviation (rad/s)
```

This line adds a normally distributed random value to each axis rate measurement with a specified standard deviation (arw_std, set to 0.01 multiplied by time for drift) with a noise magnitude scaled by sqrt(self.T_s).

```
  def apply_rate_random_walk(self, rate_rw, rrw_std, t):
    # Apply Rate Random Walk (RRW) as a random drift in the rate
    rrw_std = rrw_std * (1 + 0.000005 * t)  # Control Standard / Sim Drift
    rate_rw += np.random.normal(0, rrw_std) * np.sqrt(self.T_s)
    return rate_rw
# Generate ARW noise for all axes
    arw_noise_x, arw_noise_y, arw_noise_z = self.generate_angle_random_walk()
```

1.2 Simulating RRW

Rate Random Walk (RRW) simulates random, slow variations (drift) in the gyroscope's rate measurements. This drift accumulates over time, introducing **inaccuracies** in the angular velocity readings; when these rate measurements are integrated over time to calculate angles, RRW results in drift in the angle estimates.

Standard deviation (rrw_std) increases gradually as time (t) progresses, representing growing drift. At each time step, a new RRW noise value is generated using a **Gaussian distribution** (np.random.normal(0, rrw_std)) and scaled by the square root of the sampling period (sqrt(self.T_s)).

```
  def apply_rate_random_walk(self, rate_rw, rrw_std, t):
    # Apply Rate Random Walk
    rrw_std = rrw_std * (1 + 0.000005 * t)
    rate_rw += np.random.normal(0, rrw_std) * np.sqrt(self.T_s)
```

Gabriel Devereux

Email: gabjol@me.com Phone: +61 487 049 009 Website: https://infinityvision.tv

1.3 Simulating BI

Bias Instability (BI) introduces slow, random drift into the gyroscope's bias over time. The standard deviation for the bias instability (bias_instability_std) gradually increases as time (t) progresses, representing slowly increasing drift in the bias.

At each time step, a new bias drift value is generated using a Gaussian distribution (np.random.normal(0, bias_instability_std)) and scaled by the square root of the sampling period (sqrt(self.T_s)). The resulting noise is added to the current bias (bias)

```
def apply_bias_instability(self, bias, bias_instability_std, t):
    # Apply Bias Instability
    bias_instability_std = bias_instability_std * (1 + 0.0000025 * t)
    bias += np.random.normal(0, bias_instability_std) * np.sqrt(self.T_s)
    return bias
```

2. Eulers integration and allan variance/deviation in C++ and MATLAB

The Main.cpp script in the 2AllanDeviation_CPP folder, along with the MATLAB script Mat2AllanDeviation_PlotRWBI.m, analyses XYZ angular acceleration data by first importing gyro_data.csv. The rate data is Euler-integrated to compute the gyroscope's orientation in radians. The resulting gyroscope angle data is then passed to the Allan Variance function, which calculates the square root of the variance for noise analysis.

I will break down the CPP script. I wrote it in C++ to deepen my understanding of Allan Variance Analysis in a lower-level language. I wrote the MATLAB script to compare outputs due to rounding and normalisation errors.

2.1 Eulers integration and the choice of equation in C++

For a real-time application using Allan Variance Analysis in Kalman filtration, I was exploring the use of the equation:

$$\theta_{\{n+1\}} = \theta_n + \omega_n \cdot \Delta t$$

While this equation works well for real-time applications, when dealing with a large dataset, it requires summing all the angular rate measurements up to the current time step to compute the accumulated angle. In this case, the equation becomes:

$$\theta_n = \theta_0 + \sum_{i=0}^{n} \omega_i \cdot T_s$$

Gabriel Devereux

Email: gabjol@me.com Phone: +61 487 049 009 Website: https://infinityvision.tv

For the bellow C++ function, θ_0 is assumed to be 0. The line sum += x[i] * T_s; performs the Euler integration by adding the current angular velocity (from the array x[i]) multiplied by the sampling time T_s to the cumulative sum.

```
// Perform Euler integration on angular velocities.
void euler_integration(const std::vector<double>& x) {
    theta_x.clear();
    double sum = 0;
    for (size_t i = 0; i < x.size(); ++i) {
        sum += x[i] * T_s;
        theta_x.push_back(sum);
    }
}
```

2.2 Allan Variance Analysis and root(variance) deviation

In this instance, I took this equation from Mathworks;

$$\sigma^2(\tau) = \frac{1}{2\tau^2(L-2m)} \sum_{k=1}^{L-2m} (\theta_{k+2m} - 2\theta_{k+m} + \theta_k)^2$$

```
// Calculate Allan Variance and then root for deviation.
void allan_deviation() {
    size_t L = theta_x.size();

    size_t max_m = L / 3;  // Adjust max_m as needed
```

In this line, size_t max_m = L / 3, max_m sets the maximum value of m for which Allan Variance is computed. m represents the number of time steps (or averaging factor) used to calculate the time constant. It's set to one-third of the dataset length (L) to ensure an extensive analysis range.

```
    allanvar.clear();
    tau.clear();


    for (size_t m = 1; m < max_m; ++m) {
```

This loop for size_t m = 1; m < max_m; ++m, iterates through different values of m, where each m corresponds to a different time constant. The value of m controls the size of the intervals used.

```
        long double sum_sq_diff = 0;

        for (size_t k = 0; k < L - 2 * m; ++k) {
            double diff = theta_x[k + 2 * m] - 2 * theta_x[k + m] +
theta_x[k];
            sum_sq_diff += diff * diff;
        }

        // Calculate tau directly
        double tau_m = m * T_s;
```

Gabriel Devereux

Email: gabjol@me.com Phone: +61 487 049 009 Website: https://infinityvision.tv

```
        long double variance = sum_sq_diff / (2.0 * tau_m * tau_m *
(static_cast<double>(L) - 2.0 * static_cast<double>(m)));



        allanvar.push_back(variance);
```

The inner loop: for (size_t k = 0; k < L - 2 * m; ++k); represents the summation in the Allan Variance formula where double diff = theta_x[k + 2 * m] - 2 * theta_x[k + m] + theta_x[k]; sum_sq_diff += diff * diff; is $(\theta_{k+2m} - 2\theta_{k+m} + \theta_k)^2$. The sum of these squared differences is accumulated in sum_sq_diff.

```
        tau.push_back(tau_m);  // Store tau


    // Calculate Allan deviation
    deviation.clear();
    for (double var : allanvar) {
        deviation.push_back(sqrt(var));
    }
```

2.3 Scaling and normalisation issues in MATLAB and C++

While implementing the Allan Variance script in both MATLAB and C++, I observed significant differences in the outputs, primarily due to the handling of particular computations. After thorough troubleshooting, two main issues were identified:

1. **Numerical Precision and Underflow in C++:** In C++, I encountered a problem with precision limitations when calculating the variance due to the large denominator in the formula:

$$\text{variance} = \frac{\text{sum\_sq\_diff}}{2\tau^2(L - 2m)}$$

   The narrowing of intermediate values exacerbated this issue before being assigned to a long double. Specifically, the line:

```
long double variance = sum_sq_diff / (2.0 * tau_m * tau_m *
(static_cast<double>(L) - 2.0 * static_cast<double>(m)));
```

   It caused numerical underflows, leading to the computed values rounding to zero. The precision loss before casting the final result to long double resulted in inaccurate variance calculations.

2. **Approximation of Tau in MATLAB:** The second issue stemmed from how MATLAB handles the tau array during the Angle Random Walk (ARW) computation. In MATLAB, the logarithmic scaling of the tau array excludes $\tau_1 = 1$, which caused discrepancies when calculating the ARW $\tau = 1$. To resolve this, I approximated the Allan Deviation for τ_1 .

3. Plotting noise characteristics in MATLAB

In this instance, I took the following equations from Mathworks;

Gabriel Devereux

Email: gabjol@me.com Phone: +61 487 049 009 Website: https://infinityvision.tv

The MATLAB script Mat2AllanDeviation_PlotRWBI.m computes the Allan deviation from angular rates and subsequently analyses key noise characteristics, including ARW, RRW and BI. The script Mat3PlotDevRWBIForCPP.m imports the output .csv file generated by the C++ script and visualises the deviation and the corresponding noise characteristics.

3.1 Plotting ARW for X from deviation data

This section of the script calculates the **Angle Random Walk (ARW)** in degrees per root hour by finding the Allan Deviation at $\tau = 1$ second, or the closest available value. The ARW is converted from radians per root second to degrees per root hour. A reference line is plotted to represent how ARW noise decays over increasing averaging times, following the expected relationship of $\frac{N}{\sqrt{\tau}}$, with a label indicating the ARW noise level at $\tau = 1$.

```
% Compute Angle Random Walk in degrees per root hour
tau_1 = 1; % tau = 1 second
        y_tau1 = sigma(tau == tau_1);
if isempty(y_tau1)
        y_tau1 = sigma(ARW_idx);
end
ARW_deg_per_rt_hr = y_tau1 * 60; % Convert to degrees per root hour

% Plot ARW reference line
lineN = N ./ sqrt(tau);
loglog(tau, lineN, '--r', 'LineWidth', 1.5);
text(tau_1, N, 'N', 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right');
```

3.2 Plotting RRW for X from deviation data

This section of the script identifies and plots the **Rate Random Walk (RRW)**. The script first determines the slope (0.5) associated with RRW and locates the closest match in the Allan Deviation data. Using this point, it calculates the y-intercept of the RRW line in the log-log plot. The RRW coefficient, K, is computed from the slope and intercept. A reference line is then plotted to visually represent how RRW noise increases with increasing averaging times, following the relationship $K \cdot \sqrt{\tau}/3$

```
%% Rate Random Walk (RRW)
slope = 0.5;
[~, RRW_idx] = min(abs(dlogadev - slope));

% Find the y-intercept of the line for Rate Random Walk
b = logadev(RRW_idx) - slope * logtau(RRW_idx);

% Determine the rate random walk coefficient from the line
logK = slope * log10(3) + b;
```

```
K = 10^logK;

% Plot RRW reference line
lineK = K .* sqrt(tau / 3);
loglog(tau, lineK, '--m', 'LineWidth', 1.5);
text(3, K, 'K', 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'left');
```

3.3 Plotting BI for x from deviation data

This section computes and plots the **Bias Instability (BI)** from the Allan Deviation data. The script first identifies the slope of **0** (corresponding to constant bias instability noise) and locates the closest match in the Allan Deviation data. It then calculates the y-intercept of the bias instability line on the log-log plot.

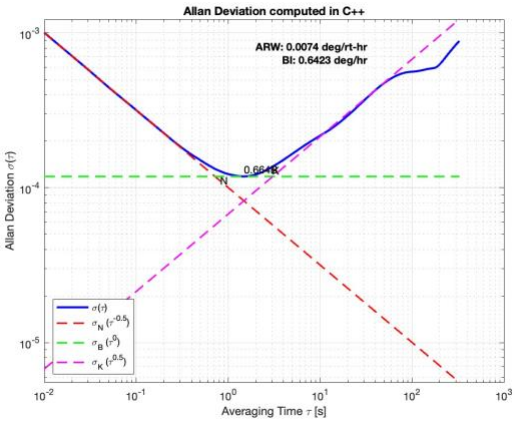
The **Bias Instability coefficient**, B, is derived from the y-intercept using the scaling factor 0.664 from $\sqrt{\frac{2\log(2)}{\pi}}$. The **BI** value is then converted to degrees per hour for easier interpretation.

```
%% Bias Instability (BI)
slope = 0;
[~, BI_idx] = min(abs(dlogadev - slope));

% Find the y-intercept of the line for Bias Instability
b = logadev(BI_idx) - slope * logtau(BI_idx);

% Determine the bias instability coefficient from the line
scfB = 664;
logB = b - log10(scfB);
B = 10^logB;

% Compute Bias Instability in degrees per hour
BI_tau = tau(BI_idx);
BI_y_value = sigma(BI_idx);
BI_deg_per_hr = (BI_y_value / 0.664) * 3600; % Convert to degrees per hour

% Plot Bias Instability reference line
lineB = B * scfB * ones(size(tau));
loglog(tau, lineB, '--g', 'LineWidth', 1.5);
text(BI_tau, scfB*B, '0.664B', 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'left');
```

3.4 Output plots

Gabriel Devereux

Email: gabjol@me.com Phone: +61 487 049 009 Website: https://infinityvision.tv



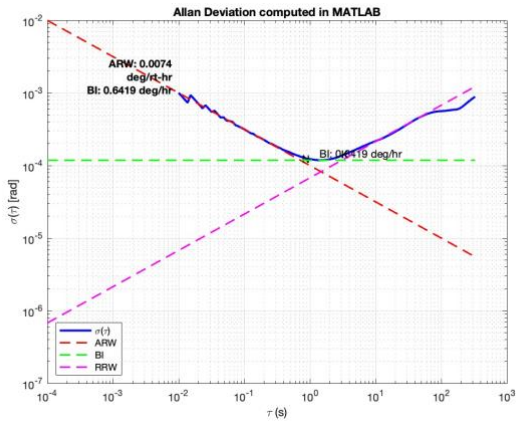


Fig 1 (left) is the plot of the CPP script, Fig (2) (right) is the plot of computing deviation in MATLAB. Note the rounding errors in BI.

4. Notes

4.1 Why not use MATLAB for everything?

MATLAB offers a rich library of predefined functions that could have made this process faster and more straightforward. However, in pursuing a deeper understanding of this statistical method, the challenge of writing the implementation in lower-level languages and the necessary troubleshooting provided an exciting and educational experience.

4.2 How an understanding of DSP makes IMU analysis easily approachable

My previous work with custom OFDM systems, where I developed an in-depth understanding of FFTs (IDFTs and DFTs), significantly simplified my approach to Allan Variance Analysis and the analysis of IMUs (Inertial Measurement Units). Both processes involve interpreting time-domain discrete data, though from different perspectives. In OFDM, my focus was on analysing time-domain data to extract frequency components, while in Allan Variance, the goal is to understand how noise varies over different time intervals. The mathematical principles required for handling discrete samples and summations are closely aligned in both techniques.

4.3 How to use (where to place the .csv's!)

- Run: python3 1GenerateGyroValues.py
- Copy (duplicate): gyro_data.csv to 2AllanDeviation_CPP/cmake-build-debug
- Run: Main.CPP
- Move: 2AllanDeviation_CPP/cmake-build-debug/allan_deviation_output.csv to the directory of the MATLAB script.
- Run: Mat3PlotDevRWBIForCPP.m
- *Optional* Run: Mat2AllanDeviation_PlotRWBI.m with gyro_data.csv in the same directory.

Gabriel Devereux

Email: gabjol@me.com Phone: +61 487 049 009 Website: https://infinityvision.tv

INFINITY
LIMITED