

Technische Dokumentation: AHRS-basiertes Rotationstracking-System

Arduino Nano 33 BLE Sense Rev2

Team Ghostbox-Videofeed

1. Einleitung und Systemarchitektur

1.1 Projektübersicht

Das vorliegende System implementiert ein hochpräzises Rotationstracking für den **Arduino Nano 33 BLE Sense Rev2**. Die Architektur basiert auf zwei spezialisierten Libraries:

- **RotationManager**: Verwaltet Sensordatenerfassung, Kalibrierung und AHRS-Fusion
- **SmoothData4D**: Implementiert Quaternion-Glättung mit ARM-optimierten Algorithmen

Das System kombiniert 9-Achsen-IMU-Daten (Beschleunigung, Gyroskop, Magnetometer) mittels Madgwick-AHRS-Algorithmus zu einer konsistenten Rotationsdarstellung im 3D-Raum.

1.2 Library-Einbindung

Die Library-Hierarchie zeigt klare Abhängigkeiten: Der `RotationManager` orchestriert alle Sensoroperationen und nutzt `SmoothData4D` für die Nachbearbeitung der Quaternionen.

arduino_main.ino:

```
#include <ArduinoBLE.h>
#include "RotationManager.h"
#include "SmoothData4D.h"
```

RotationManager.h:

```
#include <MadgwickAHRS.h>
#include <NanoBLEFlashPrefs.h>
#include "Arduino_BMI270_BMM150.h"
#include "SmoothData4D.h"
```

SmoothData4D.h:

```
#include <ReefwingAHRS.h>
#include <arm_math.h> // CMSIS-DSP für ARM-Optimierung
```

Beide Libraries verwenden ARM-CMSIS-DSP-Funktionen für optimierte Berechnungen auf der FPU des Cortex-M4-Prozessors.

2. RotationManager: Sensorfusion und AHRS

2.1 Funktionale Beschreibung

Der `RotationManager` ist das zentrale Element des Systems und erfüllt folgende Aufgaben:

1. **Initialisierung und Kalibrierung** der IMU-Sensoren (BMI270/BMM150)
2. **Sensordatenerfassung** mit 25 Hz Sample-Rate
3. **AHRS-Fusion** mittels Madgwick-Filter
4. **Adaptive Gain-Steuerung** basierend auf Bewegungszustand
5. **Magnetometer-Kompensation** mit Hard-/Soft-Iron-Korrektur

Hauptschnittstelle

```
void getCalculatedData(Quaternion& quat, float& ax, float& ay, float& az,
                      float& mx, float& my, float& mz, float& gyroMag);
```

Outputs:

- **Quaternion** (q_0, q_1, q_2, q_3): Normalisierte Rotationsdarstellung
- **Beschleunigung** (a_x, a_y, a_z): Kalibrierte Werte in g
- **Magnetfeld** (m_x, m_y, m_z): Kompensierte Werte in μT
- **Gyro-Magnitude**: Rotationsgeschwindigkeit für Zustandserkennung

2.2 Mathematische Grundlagen: AHRS-Fusion

2.2.1 Madgwick-Algorithmus

Der Madgwick-Filter fusioniert Sensor-Triplets zu einer Quaternion-Darstellung der Orientierung. Die Updategleichung lautet:

$$\mathbf{q}_{t+1} = \mathbf{q}_t + \dot{\mathbf{q}}_t \Delta t$$

wobei die Quaternion-Ableitung aus zwei Komponenten besteht:

$$\dot{\mathbf{q}}_t = \dot{\mathbf{q}}_{\omega,t} - \beta \cdot \nabla f$$

- $\dot{\mathbf{q}}_{\omega,t}$: Gyro-basierte Rotation
- $\beta \cdot \nabla f$: Korrekturgradient aus Accelerometer/Magnetometer
- β : Filter-Gain (konfigurierbar)

Gyro-Integration:

$$\dot{\mathbf{q}}_{\omega} = \frac{1}{2} \mathbf{q} \otimes \begin{bmatrix} 0 \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

Implementierung: Adaptive Gain-Anpassung

```
void RotationManager::adaptiveFilterGain(float ax, float ay, float az,
                                         float gx, float gy, float gz) {
    float acc_mag = sqrt(ax*ax + ay*ay + az*az);
    float gyro_mag = sqrt(gx*gx + gy*gy + gz*gz);

    if (gyro_mag > 200.0f) {
        filter.beta = BETA_NORM_HIGH; // 0.1
    } else if (acc_mag < ACCEL_MAG_MIN || acc_mag > ACCEL_MAG_MAX) {
        filter.beta = BETA_MIN; // 0.01
    } else {
        filter.beta = BETA_NORM_LOW; // 0.05
    }
}
```

Das System passt β dynamisch an:

- Bei schnellen Rotationen ($|\boldsymbol{\omega}| > 200^\circ/s$): Höheres β für schnellere Reaktion

- Bei externen Beschleunigungen: Niedriges β , um Gyro zu bevorzugen
- Bei Stillstand: Moderate β für Drift-Korrektur

2.2.2 Magnetometer-Kalibrierung

Magnetometer unterliegen Hard-Iron- (konstante Offsets) und Soft-Iron-Effekten (achsenabhängige Verzerrungen). Die Kompensation erfolgt in zwei Schritten:

Hard-Iron-Korrektur:

$$\begin{bmatrix} m'_x \\ m'_y \\ m'_z \end{bmatrix} = \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} - \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

Soft-Iron-Korrektur:

$$\begin{bmatrix} m''_x \\ m''_y \\ m''_z \end{bmatrix} = \mathbf{W} \cdot \begin{bmatrix} m'_x \\ m'_y \\ m'_z \end{bmatrix}$$

mit der 3×3 -Kompensationsmatrix \mathbf{W} .

Implementierung (Raum Pascal):

```
// Hard-Iron Offsets
const float MAG_HARD_IRON_OFFSET[3] = {-80.257, 37.515, 0.759};

// Soft-Iron Matrix
const float MAG_SOFT_IRON_MATRIX[3][3] = {
    {1.058, -0.017, -0.019},
    {-0.02, 1.075, 0.005},
    {0.019, 0.005, 1.113}
};

// Anwendung
float mx_soft = MAG_SOFT_IRON_MATRIX[0][0] * mx_hard +
    MAG_SOFT_IRON_MATRIX[0][1] * my_hard +
    MAG_SOFT_IRON_MATRIX[0][2] * mz_hard;
```

Validierung:

Das System überprüft die Magnetfeldstärke gegen Erwartungswerte (28–68 μT für Deutschland, Zielwert: $\approx 48 \mu T$):

```

float mag_magnitude = sqrt(mx_soft*mx_soft + my_soft*my_soft + mz_soft*mz_soft);
if (mag_magnitude < MAG_MIN || mag_magnitude > MAG_MAX) {
    magValid = false; // Werte verwerfen
}

```

2.3 Kalibrierungsprozess

Gyro-Kalibrierung

```

void RotationManager::calibrateGyro() {
    float gx_sum = 0, gy_sum = 0, gz_sum = 0;
    for (int i = 0; i < CALIBRATION_SAMPLES; i++) { // 200 Samples
        IMU.readGyroscope(data.gx, data.gy, data.gz);
        gx_sum += data.gx;
        gy_sum += data.gy;
        gz_sum += data.gz;
        delay(5);
    }
    gx_off = gx_sum / CALIBRATION_SAMPLES;
    gy_off = gy_sum / CALIBRATION_SAMPLES;
    gz_off = gz_sum / CALIBRATION_SAMPLES;
}

```

Die Kalibrierung ermittelt den Bias jeder Achse bei Stillstand:

$$\omega_{\text{off}} = \frac{1}{N} \sum_{i=1}^N \omega_i \quad \text{mit} \quad N = 200$$

Diese Offsets werden von allen zukünftigen Messungen subtrahiert.

3. SmoothData4D: Quaternion-Glättung

3.1 Funktionale Beschreibung

SmoothData4D implementiert einen exponentiellen Glättungsfilter mit folgenden Eigenschaften:

- **Adaptive Glättung** basierend auf Bewegungsgeschwindigkeit
- **Deadzone** zur Unterdrückung von Mikrovibrationen

- **Quaternion-Vorzeichen-Korrektur** (kürzester Rotationspfad)
- **ARM-CMSIS-optimiert** für Echtzeit-Performance

Hauptschnittstelle

```
void smoothQuaternion(Quaternion& _Quat, unsigned long currentTime);
```

Die Funktion modifiziert das übergebene Quaternion *in-place*.

3.2 Mathematische Grundlagen

3.2.1 Exponentielles Smoothing

Die Glättung folgt der Rekursionsformel:

$$\mathbf{q}_{\text{smooth},t} = \alpha \cdot \mathbf{q}_t + (1 - \alpha) \cdot \mathbf{q}_{\text{smooth},t-1}$$

wobei $\alpha \in [0, 1]$ den Glättungsfaktor bestimmt:

- $\alpha \rightarrow 1$: geringe Glättung, schnelle Reaktion
- $\alpha \rightarrow 0$: starke Glättung, träge Reaktion

3.2.2 Quaternion-Antipodal-Korrektur

Quaternionen besitzen die Eigenschaft $\mathbf{q} \equiv -\mathbf{q}$ (identische Rotation). Um den kürzesten Interpolationspfad zu wählen, wird das Vorzeichen korrigiert:

$$\text{sign} = \begin{cases} -1 & \text{wenn } \mathbf{q}_{\text{last}} \cdot \mathbf{q}_{\text{new}} < 0 \\ +1 & \text{sonst} \end{cases}$$

$$\mathbf{q}_{\text{corrected}} = \text{sign} \cdot \mathbf{q}_{\text{new}}$$

Implementierung (ARM-optimiert):

```
float dotProduct;
arm_dot_prod_f32(_lastQuaternions, q, 4, &dotProduct);
float sign = (dotProduct < 0.0f) ? -1.0f : 1.0f;

float q_corrected[4];
arm_scale_f32(q, sign, q_corrected, 4);
```

3.2.3 Adaptive Glättung und Deadzone

Bewegungsdetektion:

$$\Delta q = \|\mathbf{q}_{\text{corrected}} - \mathbf{q}_{\text{last}}\|$$

$$\alpha_{\text{eff}} = \begin{cases} \alpha_{\text{fast}} & \text{wenn } \Delta q > \text{MOTION_THRESHOLD} \\ \alpha & \text{sonst} \end{cases}$$

Deadzone-Implementierung:

```
if (qBetrag < DEADZONE) { // Standard: 0.003
    return; // Keine Aktualisierung
}
```

Dies unterdrückt hochfrequentes Rauschen bei Stillstand.

3.2.4 Quaternion-Normalisierung

Nach jeder Glättung muss die Einheitsquaternion-Bedingung $\|\mathbf{q}\| = 1$ wiederhergestellt werden:

$$\mathbf{q}_{\text{norm}} = \frac{\mathbf{q}}{\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}}$$

ARM-Hardware-SQRT:

```
void SmoothQuaternionData::normalizeQuaternion(float q[4]) {
    float magSquared;
    arm_dot_prod_f32(q, q, 4, &magSquared); // |q|^2

    float invMagnitude;
    arm_sqrt_f32(magSquared, &invMagnitude); // Hardware-SQRT
    invMagnitude = 1.0f / invMagnitude;

    arm_scale_f32(q, invMagnitude, q, 4); // q = q / |q|
}
```

4. Konfigurierbare Parameter

4.1 RotationManager-Parameter

| Parameter | Standardwert | Beschreibung | Auswirkung |
|-------------------|--------------|-----------------------|--|
| SAMPLE_FREQ | 25.0 Hz | AHRS-Update-Rate | Höhere Werte: bessere Auflösung, mehr CPU-Last |
| BETA_MIN | 0.01 | Minimaler Filter-Gain | Niedrig: vertraue Gyro bei externen Kräften |
| BETA_MAX | 0.3 | Maximaler Filter-Gain | Hoch: starke Accel-Korrektur bei Drift |
| BETA_NORM_LOW | 0.05 | Gain bei Stillstand | Feinabstimmung Drift-Korrektur |
| BETA_NORM_HIGH | 0.1 | Gain bei Bewegung | Balance Reaktion/Stabilität |
| ACCEL_MAG_MIN | 0.75 g | Accel-Minimum | Filtert externe Beschleunigungen |
| ACCEL_MAG_MAX | 1.25 g | Accel-Maximum | Filtert externe Beschleunigungen |
| MAG_MIN | 28 µT | Magno-Minimum | Deutschland-spezifisch |
| MAG_MAX | 68 µT | Magno-Maximum | Deutschland-spezifisch |
| MAG_SAMPLES_COUNT | 1 | Averaging-Faktor | Höher: glattere Mag-Werte, träger |

Anpassungsbeispiele:

```
// Für schnelle Bewegungen (Drohne):
BETA_NORM_HIGH = 0.15f; // Schnellere Reaktion
ACCEL_MAG_MAX = 2.0f;   // Höhere Beschleunigungen erlaubt

// Für stabilen Betrieb (Kameraständer):
BETA_NORM_LOW = 0.02f; // Weniger Drift-Korrektur
MAG_SAMPLES_COUNT = 5; // Stärkeres Mag-Averaging
```

4.2 SmoothData4D-Parameter

| Parameter | Standardwert | Beschreibung | Auswirkung |
|------------------|--------------|--------------------------------|--------------------------------------|
| alpha | 0.6 | Basis-Glättungsfaktor | Niedriger: stärkere Glättung, träger |
| FAST_ALPHA | 0.75 | Alpha bei schnellen Bewegungen | Balance Verzögerung/Stabilität |
| DEADZONE | 0.003 | Schwelle für Aktualisierung | Höher: weniger Rauschen, mehr Latenz |
| MOTION_THRESHOLD | 0.5 | Schwelle schnelle Bewegung | Bestimmt Alpha-Umschaltpunkt |

Tuning-Guidelines:

```
// Für ruhige Anwendungen (Tracking):
```

```
alpha = 0.4f;  
DEADZONE = 0.005f;
```

```
// Für reaktive Anwendungen (Gaming):
```

```
alpha = 0.8f;  
FAST_ALPHA = 0.9f;  
DEADZONE = 0.001f;
```

5. Praktische Anwendung

5.1 Vollständiges Setup-Beispiel

```
#include "RotationManager.h"
#include "SmoothData4D.h"

RotationManager sensor;
SmoothQuaternionData smoothing;

void setup() {
    sensor.init(25.0f); // 25 Hz
    smoothing.initSmoothing(25.0f, 0.6f);
}

void loop() {
    Quaternion quat;
    float ax, ay, az, mx, my, mz, gyroMag;

    sensor.getCalculatedData(quat, ax, ay, az, mx, my, mz, gyroMag);
    smoothing.smoothQuaternion(quat, millis());

    // quat enthält jetzt geglättete Rotation
    float roll, pitch, yaw;
    quaternionToEuler(quat, roll, pitch, yaw);
}
```

5.2 Performance-Optimierungen

Das System nutzt CMSIS-DSP-Funktionen für kritische Operationen:

```
// Skalarprodukt (4x schneller als Schleife)
arm_dot_prod_f32(a, b, 4, &result);

// Vektoraddition (Hardware-beschleunigt)
arm_add_f32(v1, v2, result, 4);

// Hardware-SQRT
arm_sqrt_f32(value, &sqrtValue);
```

Gemessene Laufzeiten (25 Hz):

- `ahrsMeasure()` : \approx 8 ms
- `smoothQuaternion()` : \approx 2 ms
- **Gesamte Loop:** \approx 12 ms (67% CPU-Auslastung)

6. Bekannte Limitierungen und Lösungsansätze

6.1 Yaw-Drift

Problem: Magnetometer sind in Innenräumen störanfällig (Metall, Elektronik).

Lösung:

- `magValid` -Flag verwirft ungültige Messungen
- Fallback auf 6-Achsen-IMU (`updateIMU()` statt `update()`)
- Kalibrierungsmatrizen raumspezifisch anpassen

6.2 Accel-Drift bei Beschleunigung

Problem: Externe Kräfte werden als Gravitationskomponente interpretiert.

Lösung:

```
if (acc_mag < ACCEL_MAG_MIN || acc_mag > ACCEL_MAG_MAX) {  
    filter.beta = BETA_MIN; // Vertraue Gyro  
}
```

6.3 Deadzone vs. Latenz

Trade-off: Höhere Deadzone reduziert Rauschen, erhöht aber Latenz bei kleinen Bewegungen.

Empfehlung: Adaptives Deadzone-Tuning basierend auf Anwendungsfall.

7. Zusammenfassung

Das präsentierte System kombiniert bewährte AHRS-Algorithmen mit modernen ARM-Optimierungen zu einer robusten Rotationstracking-Lösung. Die modulare Architektur ermöglicht einfache Anpassungen für verschiedene Anwendungsfälle durch Parametrierung ohne Code-Änderungen.

Kernmerkmale

- Echtzeitfähig (25 Hz mit 67% CPU)
- Adaptive Filter für verschiedene Bewegungszustände
- Robuste Magnetometer-Kompensation
- ARM-CMSIS-optimiert für maximale Performance

Anwendungsfelder

- Drohnen-Stabilisierung
- VR/AR-Tracking
- Robotik-Orientierung
- Kamera-Gimbal-Steuerung

Mathematische Zusammenfassung

Das System basiert auf zwei mathematischen Kernelementen:

1. **Madgwick-Fusion:** Kombiniert Gyro-Integration mit Accel/Mag-Korrektur durch die Gleichung:

$$\dot{\mathbf{q}}_t = \dot{\mathbf{q}}_{\omega,t} - \beta \cdot \nabla f$$

2. **Exponentielles Smoothing:** Filtert hochfrequentes Rauschen bei erhaltener Reaktivität:

$$\mathbf{q}_{\text{smooth},t} = \alpha \cdot \mathbf{q}_t + (1 - \alpha) \cdot \mathbf{q}_{\text{smooth},t-1}$$

Die Parametrierung beider Algorithmen erfolgt adaptiv basierend auf Bewegungszustand und Sensorgüte, was zu einem robusten Verhalten unter verschiedenen Einsatzbedingungen führt.