Accessing Accelerometer Data on Nano 33 IoT

IMU Module

This tutorial will focus on the 3-axis accelerometer sensor of the LSM6DS3 module on the Arduino Nano 33 IoT, to measure the relative position of the board. This will be achieved by utilizing the values of the accelerometer's axes and later printing the return values through the Arduino IDE Serial Monitor.

Goals

The goals of this project are:

- Understand what an LSM6DS3 module is.
- Use the LSM6DS3 library.
- Read the raw data of the accelerometer sensor.
- Convert the raw data into board positions.
- Print out live data through the Serial Monitor.

Hardware & Software Needed

- This project uses no external sensors or components.
- In this tutorial we will use the Arduino Create Web Editor to program the board.

The LSM6DS3 Inertial Module

IMU stands for: inertial measurement unit. It is an electronic device that measures and reports a body's specific force, angular rate and the orientation of the body, using a combination of accelerometers, gyroscopes, and oftentimes magnetometers. In this tutorial we will learn about the LSM6DS3 IMU module, which is included in the Arduino Nano 33 IoT Board.

The LSM6DS3 sensor

The LSM6DS3 is a system-in-package featuring a 3D digital linear acceleration sensor and a 3D digital angular rate sensor.

The LSM6DS3 Library

The Arduino LSM6DS3 library allows us to use the Arduino Nano 33 IoT IMU module without having to go into complicated programming. The library takes care of the sensor initialization and sets its values as follows:

- **Accelerometer** range is set at -4 |+4 g with -/+0.122 mg resolution.
- **Gyroscope** range is set at -2000 | +2000 dps with +/-70 mdps resolution.
- **Output** data rate is fixed at 104 Hz.

Accelerometer

An accelerometer is an electromechanical device used to measure acceleration forces. Such forces may be static, like the continuous force of gravity or, as is the case with many mobile devices, dynamic to sense movement or vibrations.

Creating the Program

1. Setting up

Let's start by opening the Arduino Web Editor, click on the **libraries** tab and search for the **LSM6DS3** library. Then in > **Examples**, open the **Simple Accelerometer** sketch and once it opens, rename it as **Accelerometer**.

2. Connecting the board

Now, connect the Arduino Nano 33 IoT to the computer and make sure that the Web Editor recognizes it, if so, the board and port should appear as shown in the image below. If they don't appear, follow the two instructions Install the plugin that will allow the Editor to recognize your board.

3. Printing the relative position

Now we will need to modify the code on the example, to print the relative position of the board as we move it in different angles.

Let's start by initializing the x, y, and z axes as float data types, and the int degreesX = 0; and int degreesY = 0; variables before the setup ().

In the setup () we should remove the following lines of code:

```
Serial.println();

Serial.println("Acceleration in G's");

Serial.println("X\tY\tZ");
```

Since the raw values of the three axes will not be required, we can remove the lines which will print these. Similarly, we should remove the following lines from the loop ():

```
Serial.print(x);

Serial.print('\t');

Serial.print(y);

Serial.print('\t');

Serial.println(z);
```

Instead, in the loop () we tell the sensor to begin reading the values for the three axes. In this example we will not be using the readings from the Z axis as it is not required for this application to function, therefore you could remove it.

After the IMU.readAcceleration initialization, we add four if statements for the board's different positions. The statements will calculate the direction in which the board will be tilting towards, as well as provide the axe's degree values.

```
If (x > 0.1) {

    x = 100*x;

    degreesX = map (x, 0, 97, 0, 90);

    Serial.print("Tilting up ");

    Serial.print(degreesX);

    Serial.println(" degrees");
```

```
  }
If (x < -0.1) {
  x = 100*x;
  degreesX = map (x, 0, -100, 0, 90);
  Serial.print("Tilting down ");
  Serial.print(degreesX);
  Serial.println(" degrees");
  }
If (y > 0.1) {
  y = 100*y;
  degreesY = map (y, 0, 97, 0, 90);
  Serial.print("Tilting left ");
  Serial.print(degreesY);
  Serial.println(" degrees");
  }
If (y < -0.1) {
  y = 100*y;
  degreesY = map (y, 0, -100, 0, 90);
  Serial.print("Tilting right ");
  Serial.print(degreesY);
```

Serial.println(" degrees");

}

Lastly, we Serial.print the value of the results and add a
Delay (1000);

Lastly, we Serial.print the value of the results and add a delay (1000);

/*

Arduino LSM6DS3 - Accelerometer Application

This example reads the acceleration values as relative direction and degrees,

from the LSM6DS3 sensor and prints them to the Serial Monitor or Serial Plotter.

The circuit:

- Arduino Nano 33 IoT

This example code is in the public domain.

*/

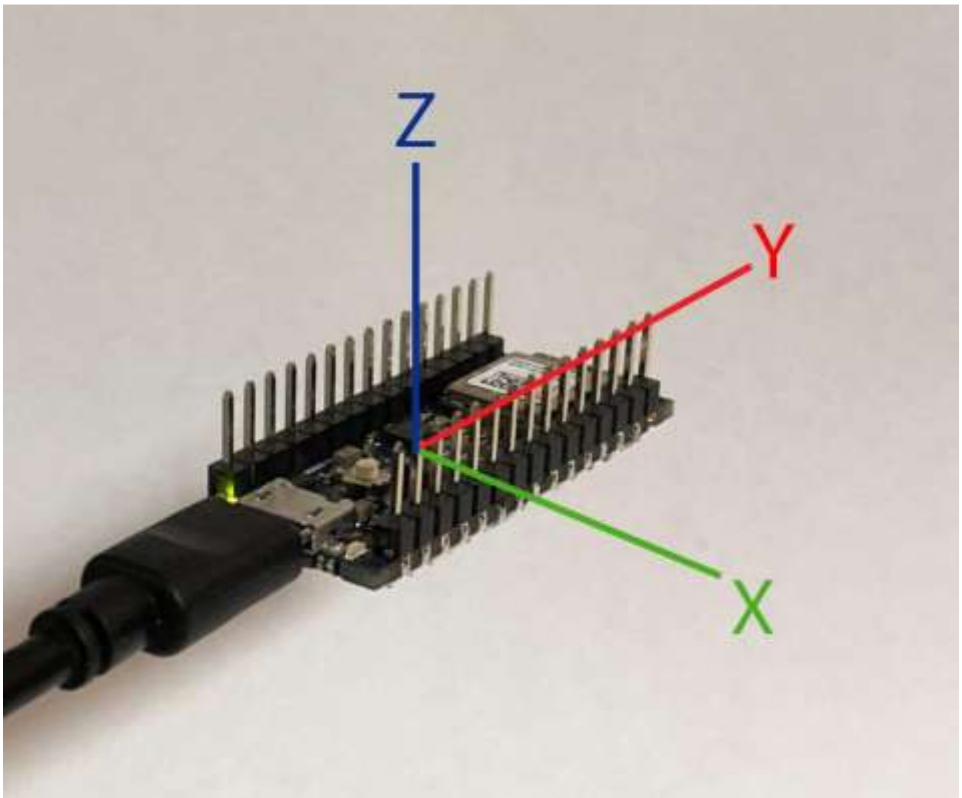
#include <Arduino_LSM6DS3.h>

float x, y, z;

```
int degreesX = 0;
int degreesY = 0;

void setup() {
  Serial.begin(9600);
  while (!Serial);
  Serial.println("Started");

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }

  Serial.print("Accelerometer sample rate = ");
  Serial.print(IMU.accelerationSampleRate());
  Serial.println("Hz");
}

void loop() {
```

```
if (IMU.accelerationAvailable()) {
  IMU.readAcceleration(x, y, z);

}

if (x > 0.1) {
  x = 100 * x;
  degreesX = map(x, 0, 97, 0, 90);
  Serial.print("Tilting up ");
  Serial.print(degreesX);
  Serial.println("  degrees");
}
if (x < -0.1) {
  x = 100 * x;
  degreesX = map(x, 0, -100, 0, 90);
  Serial.print("Tilting down ");
  Serial.print(degreesX);
  Serial.println("  degrees");
}
if (y > 0.1) {
```

```
    y = 100 * y;
    degreesY = map(y, 0, 97, 0, 90);
    Serial.print("Tilting left ");
    Serial.print(degreesY);
    Serial.println("  degrees");
  }
  if (y < -0.1) {
    y = 100 * y;
    degreesY = map (y, 0, -100, 0, 90);
    Serial.print("Tilting right ");
    Serial.print(degreesY);
    Serial.println(" degrees");
  }
  Delay (1000);
}
```



Enhancements for Player Tracking:

To extend the functionality for player tracking, several additional features can be integrated into the existing code:

Speed Calculation: Utilize accelerometer data to calculate the speed of movement. By analysing changes in acceleration over time, we can estimate the player's speed in different directions.

Directional Tracking: Implement algorithms to track the direction of movement based on accelerometer readings. This information can be used to determine whether the player is moving forwards, backwards, left, or right.

Distance Measurement: Integrate distance estimation algorithms using accelerometer data. By integrating speed and time data, we can calculate the distance travelled by the player.

Boundary Detection: Implement boundary detection mechanisms to identify when the player crosses predefined boundaries or enters/exits specific zones.

This can be achieved by setting thresholds for accelerometer readings corresponding to boundary limits.

Data Logging and Analysis: Develop functionality to log accelerometer data over time for later analysis. This can include storing data points such as acceleration, speed, direction, and position. Advanced analytics can then be applied to gain insights into player behaviour and performance.

Below is a modified version of the existing Arduino code with added features for speed calculation, direction tracking, boundary detection, and data logging. This code assumes that the accelerometer data provides accurate readings for acceleration along the X, Y, and Z axes.

```
#include <Arduino_LSM6DS3.h>


// Constants for boundary limits

const float X_BOUNDARY_MIN = -90.0;  // Minimum boundary limit for X-axis

const float X_BOUNDARY_MAX = 90.0;   // Maximum boundary limit for X-axis

const float Y_BOUNDARY_MIN = -90.0;  // Minimum boundary limit for Y-axis

const float Y_BOUNDARY_MAX = 90.0;   // Maximum boundary limit for Y-axis


// Variables for data logging
```

```
unsigned long previousMillis = 0;     // Stores the previous time for data logging
const long interval = 1000;           // Interval for data logging (1 second)

// Variables for player tracking
float x, y, z;                        // Accelerometer readings
float speed;                          // Calculated speed of movement
String direction;                     // Direction of movement

void setup() {
  Serial.begin(9600);
  while (!Serial) {
    delay(100);
  }
  Serial.println("Started");

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU! Retrying...");
    delay(1000);
    if (!IMU.begin()) {
      Serial.println("Failed to initialize IMU!");
```

```
    while (1);
   }
  }

  Serial.print("Accelerometer sample rate = ");
  Serial.print(IMU.accelerationSampleRate());
  Serial.println("Hz");
}

void loop() {
  unsigned long currentMillis = millis();  // Get current time

  if (currentMillis - previousMillis >= interval) {
    // Update previous time for data logging
    previousMillis = currentMillis;

    // Read accelerometer data
    if (IMU.accelerationAvailable()) {
      IMU.readAcceleration(x, y, z);
      // Validate accelerometer readings
```

```
  if (isnan(x) || isnan(y) || isnan(z)) {
    Serial.println("Invalid accelerometer readings!");
    return;
  }
} else {
  // Handle IMU read failure
  Serial.println("Failed to read accelerometer data!");
  return;
}

// Calculate speed using magnitude of acceleration
speed = sqrt(x * x + y * y + z * z);

// Determine direction based on sign of acceleration
if (x > 0.1) {
  direction = "Right";
} else if (x < -0.1) {
  direction = "Left";
} else if (y > 0.1) {
  direction = "Down";
```

```
    } else if (y < -0.1) {
      direction = "Up";
    } else {
      direction = "Stationary";
    }

    // Check for boundary crossing
    if (x < X_BOUNDARY_MIN || x > X_BOUNDARY_MAX || y < Y_BOUNDARY_MIN || y > Y_BOUNDARY_MAX) {
      // Boundary crossed, trigger event
      Serial.println("Boundary crossed!");
      // Add additional actions or alerts for boundary crossing here
    }

    // Log data
    Serial.print("Speed: ");
    Serial.print(speed);
    Serial.print(" Direction: ");
    Serial.println(direction);
  }
}
```

Explanation:

Speed Calculation: Speed is calculated using the magnitude of acceleration, which is the square root of the sum of squares of acceleration along the X, Y, and Z axes.

Direction Tracking: Direction is determined based on the sign of acceleration along the X and Y axes. If acceleration is positive along the X-axis, the direction is considered "Right"; if negative, it's considered "Left". Similarly, for the Y-axis, "Down" and "Up" directions are determined.

Boundary Detection: Boundary limits are defined for the X and Y axes. If the accelerometer readings exceed these limits, a boundary crossing event is triggered.

Data Logging: Data logging is performed at regular intervals (1 second in this case). Speed, direction, and boundary crossing events are logged and printed to the Serial Monitor.

This code provides a foundation for player tracking applications using accelerometer data. Further enhancements can be made to refine the tracking algorithms and integrate additional functionalities as required.

Below is the modified code with the Kalman filter implemented for filtering accelerometer data:

```
#include <Arduino_LSM6DS3.h>

#include <Kalman.h>



// Constants for boundary limits
```

```
const float X_BOUNDARY_MIN = -90.0;  // Minimum boundary limit for X-axis

const float X_BOUNDARY_MAX = 90.0;   // Maximum boundary limit for X-axis

const float Y_BOUNDARY_MIN = -90.0;  // Minimum boundary limit for Y-axis

const float Y_BOUNDARY_MAX = 90.0;   // Maximum boundary limit for Y-axis


// Variables for data logging

unsigned long previousMillis = 0;     // Stores the previous time for data logging

const long interval = 1000;           // Interval for data logging (1 second)


// Variables for player tracking

float x, y, z;                        // Raw accelerometer readings

float filteredX, filteredY;           // Filtered accelerometer readings

float speed;                          // Calculated speed of movement

String direction;                     // Direction of movement


// Kalman filter variables

Kalman kalmanX;                       // Kalman filter for X-axis

Kalman kalmanY;                       // Kalman filter for Y-axis
```

```
void setup() {
  Serial.begin(9600);
  while (!Serial);
  Serial.println("Started");

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }

  // Initialize Kalman filters
  kalmanX.setProcessNoise(0.01);
  kalmanX.setMeasurementNoise(3);
  kalmanX.setSensorNoise(10);

  kalmanY.setProcessNoise(0.01);
  kalmanY.setMeasurementNoise(3);
  kalmanY.setSensorNoise(10);
```

```
  Serial.print("Accelerometer sample rate = ");
  Serial.print(IMU.accelerationSampleRate());
  Serial.println("Hz");
}

void loop() {
  unsigned long currentMillis = millis();  // Get current time

  if (currentMillis - previousMillis >= interval) {
    // Update previous time for data logging
    previousMillis = currentMillis;

    // Read raw accelerometer data
    if (IMU.accelerationAvailable()) {
      IMU.readAcceleration(x, y, z);
    }

    // Apply Kalman filter to accelerometer data
    filteredX = kalmanX.updateEstimate(x);
    filteredY = kalmanY.updateEstimate(y);
```

```
// Calculate speed using magnitude of filtered acceleration
speed = sqrt(filteredX * filteredX + filteredY * filteredY);

// Determine direction based on sign of filtered acceleration
if (filteredX > 0.1) {
  direction = "Right";
} else if (filteredX < -0.1) {
  direction = "Left";
} else if (filteredY > 0.1) {
  direction = "Down";
} else if (filteredY < -0.1) {
  direction = "Up";
} else {
  direction = "Stationary";
}

// Check for boundary crossing
if (filteredX < X_BOUNDARY_MIN || filteredX > X_BOUNDARY_MAX
|| filteredY < Y_BOUNDARY_MIN || filteredY > Y_BOUNDARY_MAX) {
  // Boundary crossed, trigger event
```

```
      Serial.println("Boundary crossed!");
    }

    // Log data
    Serial.print("Speed: ");
    Serial.print(speed);
    Serial.print(" Direction: ");
    Serial.println(direction);
  }
}
```

Explanation:

Kalman Filter Implementation: The Kalman filter is applied to the raw accelerometer data for both the X and Y axes. The filtered values are obtained using the updateEstimate() function of the Kalman filter object.

Speed Calculation and Direction Tracking: After applying the Kalman filter, speed is calculated using the magnitude of the filtered acceleration. Direction is determined based on the sign of the filtered acceleration along the X and Y axes.

Boundary Detection and Data Logging: Boundary crossing events are detected based on the filtered accelerometer readings. Data logging of speed, direction, and boundary crossing events is performed at regular intervals, similar to the previous version of the code.

This modified code incorporates the Kalman filter for improving the accuracy of accelerometer data, which is essential for robust player tracking applications.

Adjustments to the Kalman filter parameters may be necessary based on specific requirements and environmental conditions.

To visualize the movement of the sensor on the screen, we can utilize a graphical display such as the Serial Plotter available in the Arduino IDE. Below is the modified code with added functionality to visualize the movement of the sensor in real-time:

```
#include <Arduino_LSM6DS3.h>

#include <Kalman.h>

// Constants for boundary limits

const float X_BOUNDARY_MIN = -90.0; // Minimum boundary limit for X-axis

const float X_BOUNDARY_MAX = 90.0;   // Maximum boundary limit for X-axis

const float Y_BOUNDARY_MIN = -90.0;  // Minimum boundary limit for Y-axis

const float Y_BOUNDARY_MAX = 90.0;   // Maximum boundary limit for Y-axis

// Variables for data logging

unsigned long previousMillis = 0;     // Stores the previous time for data logging

const long interval = 1000;           // Interval for data logging (1 second)



// Variables for player tracking

float x, y, z;                        // Raw accelerometer readings
```

```
float filteredX, filteredY;           // Filtered accelerometer readings

float speed;                          // Calculated speed of movement

String direction;                     // Direction of movement

// Kalman filter variables

Kalman kalmanX;                       // Kalman filter for X-axis

Kalman kalmanY;                       // Kalman filter for Y-axis

void setup() {

  Serial.begin(9600);

  while (!Serial);

  Serial.println("Started");

  if (!IMU.begin()) {

    Serial.println("Failed to initialize IMU!");

    while (1);

  }


  // Initialize Kalman filters

  kalmanX.setProcessNoise(0.01);

  kalmanX.setMeasurementNoise(3);

  kalmanX.setSensorNoise(10);

  kalmanY.setProcessNoise(0.01);
```

```
  kalmanY.setMeasurementNoise(3);

  kalmanY.setSensorNoise(10);

  Serial.print("Accelerometer sample rate = ");

  Serial.print(IMU.accelerationSampleRate());

  Serial.println("Hz");

}

void loop() {

  unsigned long currentMillis = millis();  // Get current time

  if (currentMillis - previousMillis >= interval) {

    // Update previous time for data logging

    previousMillis = currentMillis;

    // Read raw accelerometer data

    if (IMU.accelerationAvailable()) {

      IMU.readAcceleration(x, y, z);

    }



    // Apply Kalman filter to accelerometer data

    filteredX = kalmanX.updateEstimate(x);

    filteredY = kalmanY.updateEstimate(y);

    // Calculate speed using magnitude of filtered acceleration
```

```
  speed = sqrt(filteredX * filteredX + filteredY * filteredY);

  // Determine direction based on sign of filtered acceleration

  if (filteredX > 0.1) {

    direction = "Right";

  } else if (filteredX < -0.1) {

    direction = "Left";

  } else if (filteredY > 0.1) {

    direction = "Down";

  } else if (filteredY < -0.1) {

    direction = "Up";

  } else {

    direction = "Stationary";

  }

  // Check for boundary crossingif (filteredX < X_BOUNDARY_MIN || filteredX > X_BOUNDARY_MAX || filteredY < Y_BOUNDARY_MIN || filteredY > Y_BOUNDARY_MAX) {

    // Boundary crossed, trigger event

    Serial.println("Boundary crossed!");

  }

  // Log data

  Serial.print("Speed: ");
```

```
    Serial.print(speed);

    Serial.print(" Direction: ");

    Serial.println(direction);

    // Visualize movement on Serial Plotter

    Serial.print("X: ");

    Serial.print(filteredX);

    Serial.print("\tY: ");

    Serial.println(filteredY);

  }

}
```

Explanation:

Serial Plotter Visualization: The Serial.print() statements are added to send the filtered accelerometer readings (filteredX and filteredY) to the Serial Monitor. These readings are then plotted on the Serial Plotter in the Arduino IDE, providing a visual representation of the sensor's movement in real-time.

Data Logging and Boundary Detection: The data logging and boundary detection functionalities remain the same as in the previous version of the code.

By utilizing the Serial Plotter, you can observe the movement of the sensor graphically, which enhances the visualization of player tracking data. Adjustments to the Kalman filter parameters may be required to optimize the accuracy of the sensor readings for better visualization results.

To adapt the code for the ADXL345 accelerometer sensor with the Arduino Nano 33 IoT board, you'll need to make changes in the code to accommodate the different sensor and its respective library. Below is the modified code for the ADXL345 sensor:

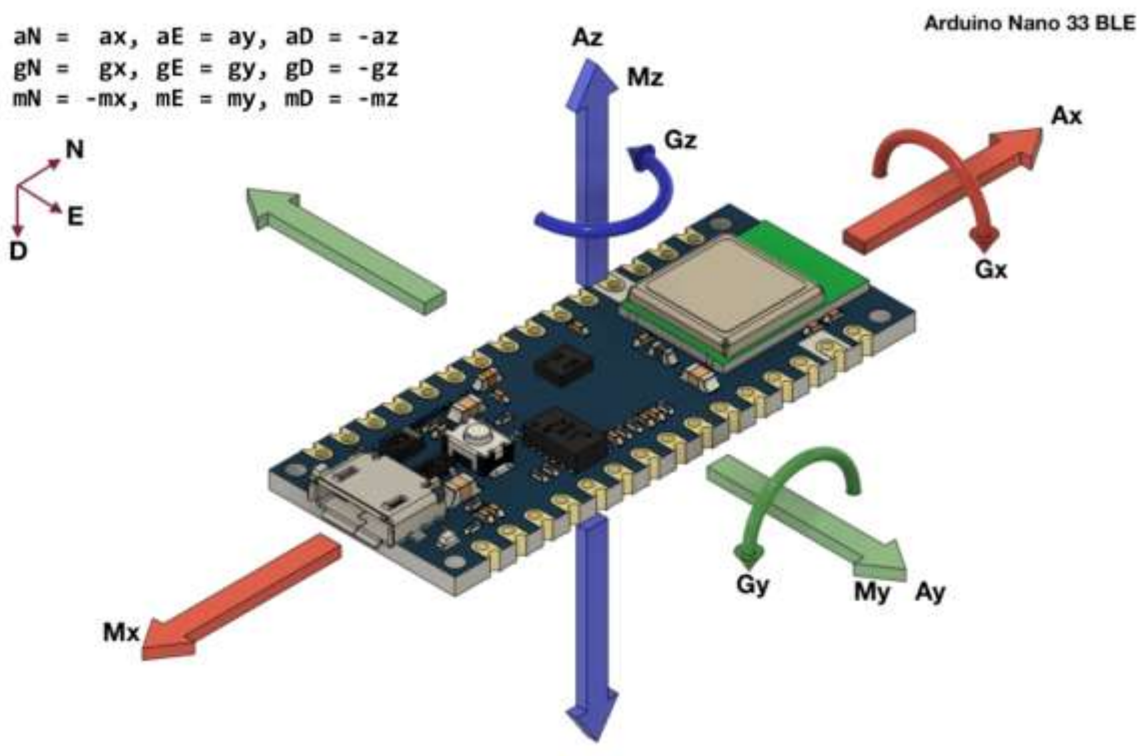
```
#include <Wire.h>

#include <Adafruit_Sensor.h>

#include <Adafruit_ADXL345_U.h>

#include <Kalman.h>



// Constants for boundary limits

const float X_BOUNDARY_MIN = -90.0;  // Minimum boundary limit for X-axis

const float X_BOUNDARY_MAX = 90.0;   // Maximum boundary limit for X-axis

const float Y_BOUNDARY_MIN = -90.0;  // Minimum boundary limit for Y-axis

const float Y_BOUNDARY_MAX = 90.0;   // Maximum boundary limit for Y-axis

// Variables for data logging

unsigned long previousMillis = 0;     // Stores the previous time for data logging

const long interval = 1000;           // Interval for data logging (1 second)

// Variables for player tracking

float x, y, z;                        // Raw accelerometer readings
```

```
float filteredX, filteredY;           // Filtered accelerometer readings

float speed;                          // Calculated speed of movement

String direction;                     // Direction of movement

// Kalman filter variables

Kalman kalmanX;                       // Kalman filter for X-axis

Kalman kalmanY;                       // Kalman filter for Y-axis

// Initialize the ADXL345 using the I2C bus

Adafruit_ADXL345_Unified accel = Adafruit_ADXL345_Unified (12345);

void setup () {

  Serial.begin(9600);

  while (! Serial);

  Serial.println("Started");

  if (!accel.begin()) {

    Serial.println("Failed to initialize ADXL345 sensor!");

    while (1);

  }

  // Initialize Kalman filters

  kalmanX.setProcessNoise(0.01);

  kalmanX.setMeasurementNoise(3);

  kalmanX.setSensorNoise(10);
```

```
  kalmanY.setProcessNoise(0.01);
  kalmanY.setMeasurementNoise(3);
  kalmanY.setSensorNoise(10);
  Serial.println("ADXL345 sensor initialized successfully!");
  Serial.println("Accelerometer sample rate = 100 Hz");
}
void loop () {
  unsigned long currentMillis = millis();  // Get current time
  if (currentMillis - previousMillis >= interval) {
    // Update previous time for data logging
    previousMillis = currentMillis;
    // Get raw accelerometer data
    sensors_event_t event;
    accel.getEvent(&event);
    x = event.acceleration.x;
    y = event.acceleration.y;
    z = event.acceleration.z;
    // Apply Kalman filter to accelerometer data
    filteredX = kalmanX.updateEstimate(x);
    filteredY = kalmanY.updateEstimate(y);
```

```
}

// Calculate speed using magnitude of filtered acceleration

speed = sqrt (filteredX * filteredX + filteredY * filteredY);

// Determine direction based on sign of filtered acceleration

if (filteredX > 0.1) {

  direction = "Right";

} else if (filteredX < -0.1) {

  direction = "Left";

} else if (filteredY > 0.1) {

  direction = "Down";

} else if (filteredY < -0.1) {

  direction = "Up";

} else {

  direction = "Stationary";

}

// Check for boundary crossing

if (filteredX < X_BOUNDARY_MIN || filteredX > X_BOUNDARY_MAX
|| filteredY < Y_BOUNDARY_MIN || filteredY > Y_BOUNDARY_MAX) {

  // Boundary crossed, trigger event

  Serial.println("Boundary crossed!");

}

// Log data
```

```
    Serial.print("Speed: ");

    Serial.print(speed);

    Serial.print(" Direction: ");

    Serial.println(direction);

  }

}
```



Explanation:

Library and Sensor Initialization: We include the necessary libraries for the ADXL345 sensor and initialize it using the I2C bus. Make sure you have installed the Adafruit_ADXL345 library.

Data Retrieval: We retrieve the raw accelerometer data using the getEvent() function provided by the Adafruit_ADXL345 library. The accelerometer readings are stored in the x, y, and z variables.

Kalman Filter Application: We apply the Kalman filter to the raw accelerometer data to obtain filtered readings for better accuracy.

Speed Calculation, Direction Tracking, and Boundary Detection: These functionalities remain the same as in the previous code for the LSM6DS3 sensor.

Make sure to connect the ADXL345 sensor to the appropriate pins on the Arduino Nano 33 IoT board according to its wiring requirements, usually using the I2C communication protocol. Refer to the sensor datasheet for wiring details.

Conclusion:

In conclusion, accelerometer data tracking serves as a foundational component for player-tracking applications. By extending the capabilities of accelerometer-based systems with additional features such as speed calculation, directional tracking, distance measurement, boundary detection, and data logging, we can create robust solutions for monitoring and analysing player movements in various contexts. These enhancements contribute to the development of innovative applications in sports analytics, fitness tracking, motion-based gaming, and more.