Setup Guide- Accelerometer sensor

IMU Module

This guide will explain how to use the 3-axis accelerometer sensor on the LSM6DS3 breakout board, or an Arduino board featuring an integrated LSM6DS3 (like the Arduino Nano 33 IoT). We will demonstrate how to interpret the accelerometer's axis data to determine the board's relative position. The outcomes will be displayed on the Arduino IDE Serial Monitor by printing the accelerometer's return values.

Goals

This project aims to achieve several objectives:

1. Learn about the LSM6DS3 module.
2. Utilize the LSM6DS3 library.
3. Capture the raw data from the accelerometer sensor.
4. Translate the raw data into specific board positions.
5. Display live data on the Serial Monitor.

Equipment Needed

- Arduino board (e.g., Arduino Uno, Mega, or Leonardo)
- LSM6DS3 breakout board or an Arduino board with a built-in LSM6DS3 (such as the Arduino Nano 33 IoT)
- Breadboard and jumper wires (if using a breakout board)
- USB cable to connect the Arduino to the computer

Software Requirements

- Arduino IDE (can be downloaded from Arduino Software)
-

Hardware Setup

1. **Connecting the LSM6DS3 to an Arduino:**

- For Arduino with built-in LSM6DS3 (e.g., Arduino Nano 33 IoT): No external connections are required as the sensor is built-in.

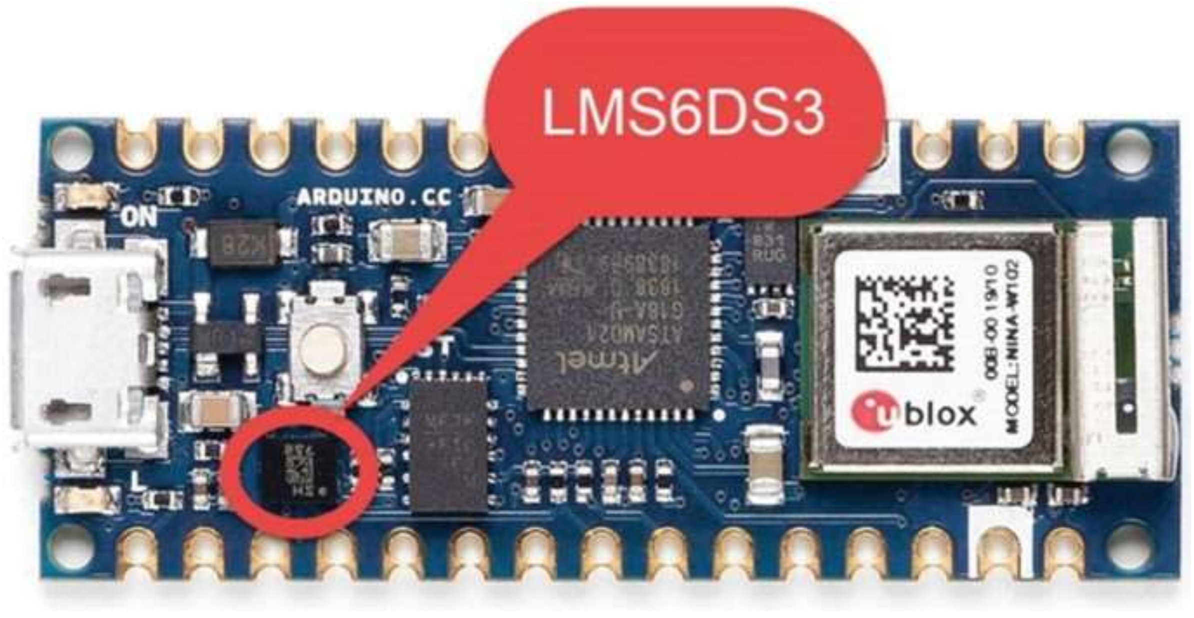
For breakout board:

- Connect VCC of LSM6DS3 to 3.3V on Arduino.
- Connect GND to GND on Arduino.
- Connect SDA (Serial Data Line) to A4 (on Uno) or the correct SDA pin of your Arduino.
- Connect SCL (Serial Clock Line) to A5 (on Uno) or the correct SCL pin of your Arduino.

2. **Secure the connections with a breadboard and jumper wires if necessary.**

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.

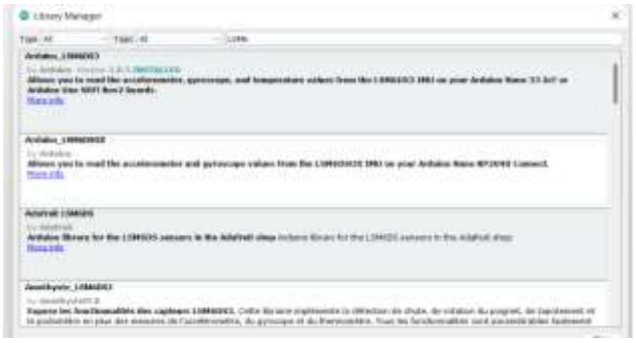
Accelerometer

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

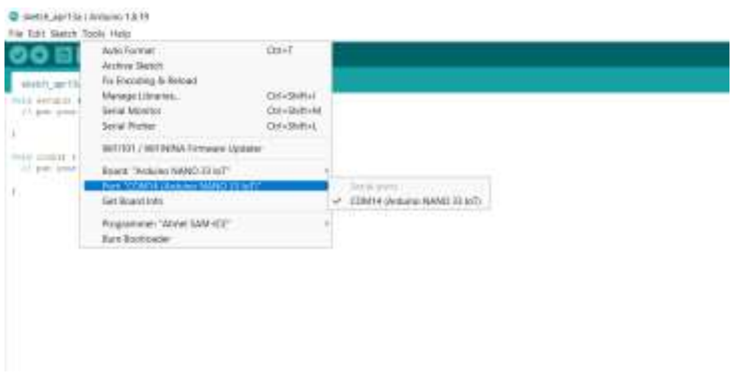
1. Setting up

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 your Arduino Nano 33 IoT to the computer. Ensure that the Arduino Web Editor successfully recognizes the device; the board and port should be displayed similarly to the example provided in the image below. If the board and port do not appear, you'll need to follow these steps: Install the plugin that enables the Editor to recognize your board.


File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino NANO 33 IoT"
Port: "COM14 (Arduino NANO 33 IoT)"
Get Board Info
Programmer: "Atmel SAM-ICE"
Burn Bootloader
Serial ports
COM14 (Arduino NANO 33 IoT)

3. Printing the relative position

To modify the example code to print the relative position of the board as it is moved at different angles, you can start by initializing the x, y, and z axes as float data types and define two integer variables, degreesX and degreesY, to store the angles. Place these variable declarations before the setup() function in your Arduino sketch:

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

```
Serial.println();

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

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

If you're looking to focus only on the angles and not the raw accelerometer values, you can simplify your code by removing any print statements that output these raw values from the loop() function. This will declutter the Serial Monitor and allow you to concentrate on the angle measurements.

Here's how you might revise your loop() function to remove unnecessary print statements:Serial.print(x);

```
Serial.print('\t');

Serial.print(y);

Serial.print('\t');

Serial.println(z);
```

In the loop() function, we'll instruct the sensor to start capturing data for the three axes. For this example, we'll skip using data from the Z axis since it's not necessary for our application. After initializing the accelerometer readings with

IMU.readAcceleration, you should add four conditional statements. These conditions will determine the board's tilt direction and calculate the degrees of tilt for each relevant axis. 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.

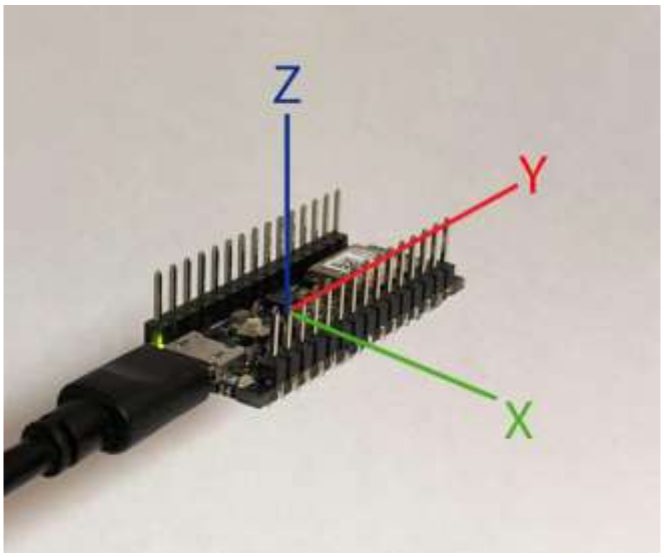
The circuit:

- 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.

GitHub→ https://github.com/Redback-Operations/redback-orion/blob/main/Accelerometer_boundary_detection.ino

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);
  }
}
```

GitHub→ https://github.com/Redback-Operations/redback-orion/blob/main/Accelerometer_KalmanFilter_code.ino

Future Deliverables of Accelerometer Sensor

Below is a comprehensive guide on setting up an Arduino accelerometer project to send alerts via email and SMS when specific sensor thresholds are met.

- **Alerting via Email and Mobile Phones**

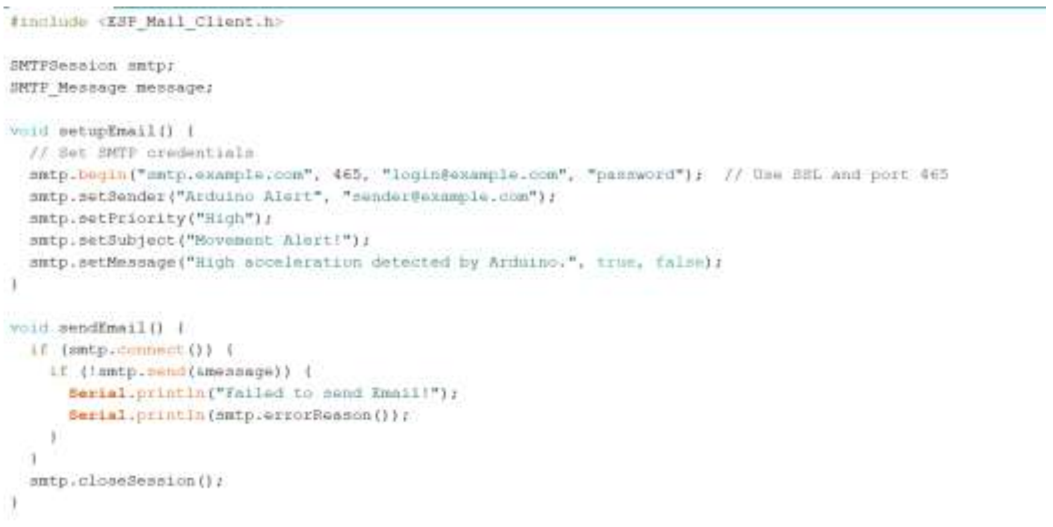
✓ **Email Alerting**

1. Objective: Send email notifications when the accelerometer detects movement exceeding predefined thresholds.
2. Required Hardware: ESP8266 or ESP32 module for internet connectivity.
3. Required Libraries: For ESP8266/ESP32, use libraries such as ESP8266WiFi and ESP_Mail_Client.
4. Steps:
    - Setup WiFi Connection:
    - Initialize the WiFi module and connect to a local network.
5. Configure SMTP Settings:
    - Set up SMTP details for sending emails through a mail server.
6. Coding:

- Integrate email sending code into the main Arduino sketch to trigger on specific accelerometer events.

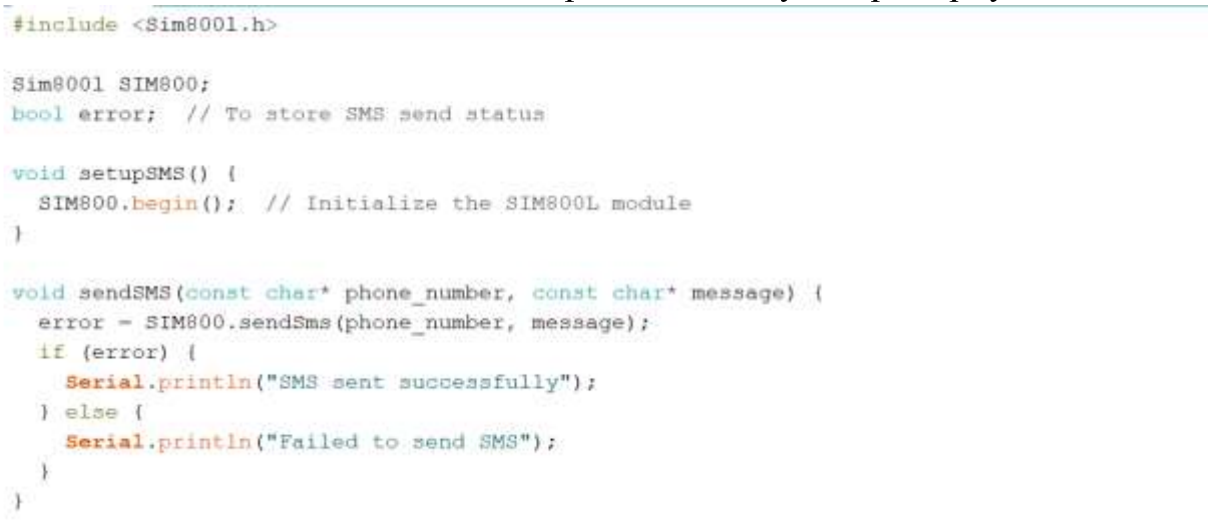
7. Testing:

- Validate by triggering alerts under controlled conditions to ensure emails are sent accurately and timely.



```
#include <ESP_Mail_Client.h>

SMTPSession smtp;
SMTP_Message message;

void setupEmail() {
  // Set SMTP credentials
  smtp.begin("smtp.example.com", 465, "login@example.com", "password");  // Use SSL and port 465
  smtp.setSender("Arduino Alert", "sender@example.com");
  smtp.setPriority("High");
  smtp.setSubject("Movement Alert!");
  smtp.setMessage("High acceleration detected by Arduino.", true, false);
}

void sendEmail() {
  if (smtp.connect()) {
    if (!smtp.send(&message)) {
      Serial.println("Failed to send Email!");
      Serial.println(smtp.errorReason());
    }
  }
  smtp.closeSession();
}
```

- **SMS Alerting**
  1. Objective: Send SMS notifications when significant motion is detected.
  2. Required Hardware: GSM module like SIM800L.
  3. Required Libraries: SIM800l library.
  4. Steps:
     - ✓ GSM Module Setup:
       - Connect the SIM800L module to the Arduino, ensuring it has a valid SIM card with SMS capabilities.
     - ✓ Coding:
       - Implement functions in the Arduino sketch to send SMS based on accelerometer readings.
     - ✓ Testing:
       - Test SMS sending functionality to make sure messages are received on a mobile phone correctly and promptly.



```
#include <Sim800l.h>

Sim800l SIM800;
bool error;  // To store SMS send status

void setupSMS() {
  SIM800.begin();  // Initialize the SIM800L module
}

void sendSMS(const char* phone_number, const char* message) {
  error = SIM800.sendSms(phone_number, message);
  if (error) {
    Serial.println("SMS sent successfully");
  } else {
    Serial.println("Failed to send SMS");
  }
}
```

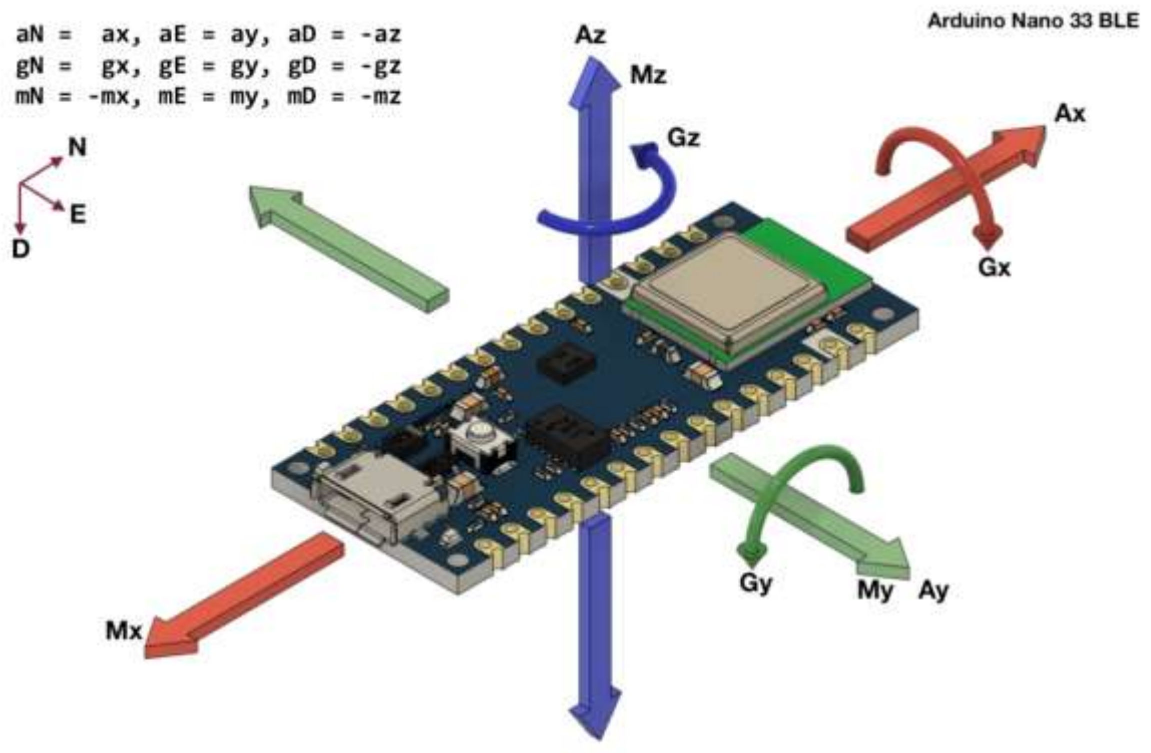
- **Integration Tips**
- **Security**: Securely handle WiFi and SMTP credentials, ensuring they are not exposed in shared or public code repositories.
- **Power Management**: Both ESP and GSM modules increase power consumption. Implement power-saving techniques, especially for battery-operated setups.
- **Scalability**: Design the alerting mechanisms to handle multiple types of alerts and destinations to enhance scalability and flexibility.

By following these guidelines and example codes, students can implement robust email and SMS alerting features in their Arduino projects. This will enable them to develop projects capable of real-time monitoring and response, useful in various applications such as home security, industrial monitoring, or personal safety devices.

Additional Tips:

- **Secure Your Credentials:** Keep your WiFi and SMTP credentials confidential, especially in shared environments.
- **Manage Power:** Devices like the ESP8266/ESP32 and SIM800L can draw significant power, particularly when transmitting. Consider power management strategies for battery-operated devices.
- **Scalability and Flexibility:** Design your system to be adaptable, allowing for easy updates to alert types and methods as needed.
- By following these simplified steps and using the example codes provided, students can effectively add email and SMS alerting functionalities to their Arduino projects, enhancing their ability to monitor and respond to significant sensor-detected events.

ADXL345 Sensor



aN = ax, aE = ay, aD = -az
gN = gx, gE = gy, gD = -gz
mN = -mx, mE = my, mD = -mz
N
E
D
Az
Mz
Gz
Arduino Nano 33 BLE
Ax
Gx
Gy
My
Ay
Mx

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.