Analysis of the results obtained by two different methods of measuring the performance of pneumatic cylinders

Ghinea Mihalache, Rosca Bogdan, Barbosu Dan-Alexandru

Faculty of Industrial Engineering and Robotics, Dept of Robots and Manufacturing Systems

National University of Science and Technology Politehnica Bucharest, Romania

Abstract

Pneumatic cylinders harness the power of the compressed air to generate linear motion, which can be used in many industrial applications. Due to their high-speed operation and air pressure changes, they are naturally prone to produce vibrations. If these vibrations are not considered, they can have direct implications for the surrounding machinery, equipment or the overall structure, making it indispensable to measure and analyze them, in order to come up with solutions against their effect. This study investigates the acceleration generated by a pneumatic cylinder using two different types of sensors, in order to understand their strengths and limitations associated with the vibration control problem..

Keywords: pneumatic cylinder, vibrations, linear sensor, accelerometer, mqtt

1. Introduction

Pneumatic systems use compressed air as a source of energy. Due to their efficiency, simplicity, and affordability, they have become widely used in various industrial applications. These systems harness the force generated by air pressure to perform different operations.

Vibrations have always been a common problem in pneumatic systems. Prolonged exposure to vibrations can lead to various issues, such as component damage and decreased system performance.

Efficiency is the most important pillar in any industrial system, but vibrations can reduce this optimal performance. In the context of pneumatics, vibrations can lead to system oscillations, affecting the precision and consistency of processes. For example, on a production line where precise accuracy is critical for creating a quality product, vibration-induced movements can affect the movements, decreasing the product's value. Furthermore, the movements of actuators can be affected, making them unpredictable. Lack of control can lead to decreased performance and safety risks in an industrial setting.

Despite the challenges, vibrations can be used to our advantage. Industrial maintenance is a challenge, especially unplanned maintenance, which can lead to significant financial losses. Vibrations act as a diagnostic tool for the current state of the system. By analyzing these vibrations, information about the current condition of the components can be gathered. Implementing vibration analysis systems alongside vibration control techniques helps create predictive maintenance techniques.

1.1. Inputs

1.1.1. How does a pneumatic cylinder work

Pneumatic cylinders are capable of providing a robust and efficient way to transform the energy of compressed air into linear motion. The core principle consists of a cylindrical casing that encloses a piston which is connected to an external rod. Introducing compressed air on either side of the piston creates a difference of pressure between the two cavities, causing the rod to extend or retract.

1.1.2. Used sensors

- LDS(Linear Displacement Sensor) Aventic SM6: measures the change in position of an object along one axis.
- MPU6050 accelerometer: measures the rate of change in position, the acceleration.

1.2 Our goals

In this study, we aim to find a better solution for collecting data on the vibrations of a pneumatically actuated system.

2.0. System setup

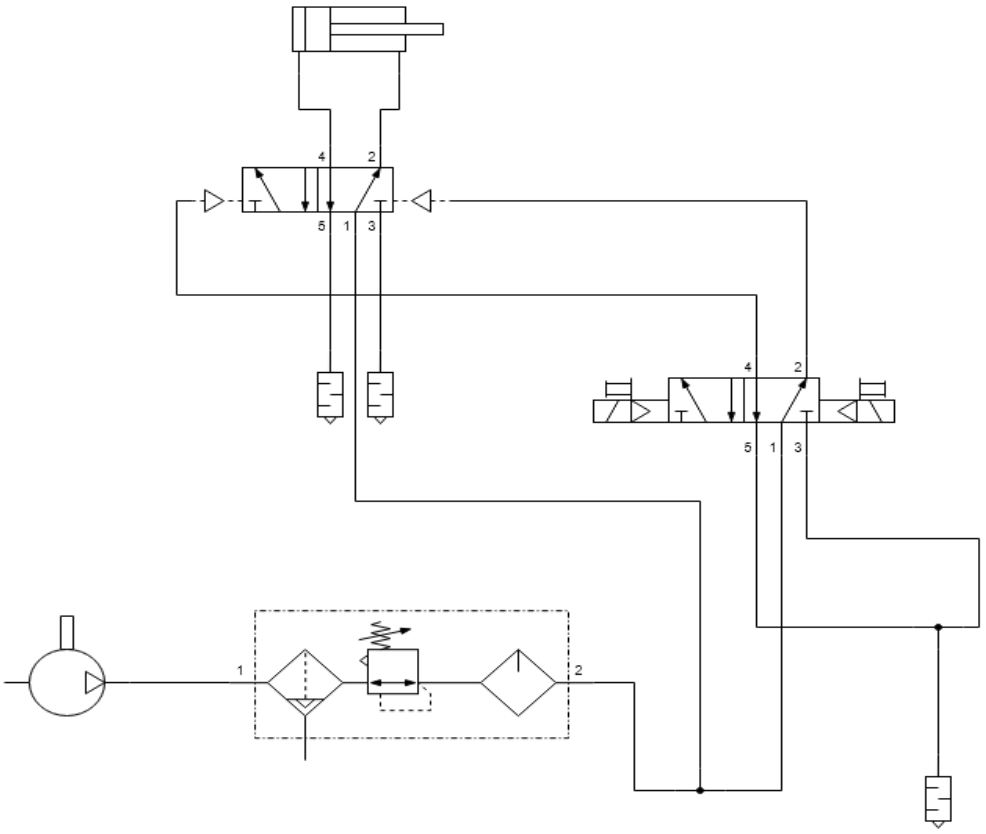
2.1. Software setup

In order to program the MPU6050 accelerometer we used the Arduino IDE(Integrated Development Environment). We used MQTT to transmit the data wirelessly. To process the data we used Node-red along with Grafana.

- Node-red is a visual programming tool designed for the interaction between multiple devices, online services.
- Grafana is an open-source analytics and visualization platform
- Arduino IDE is an application which allows the user to code and upload the program to the SOC microcontroller.

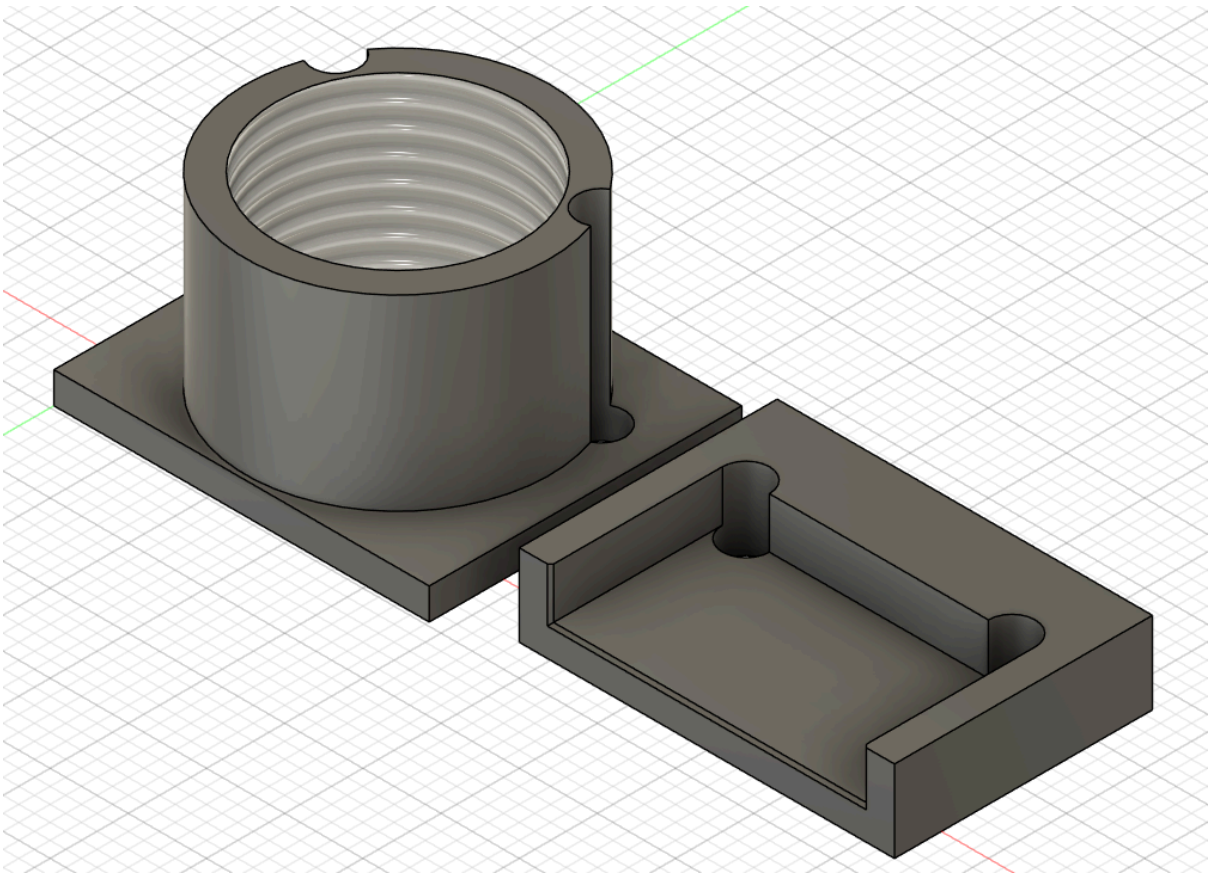
2.2. Stand setup

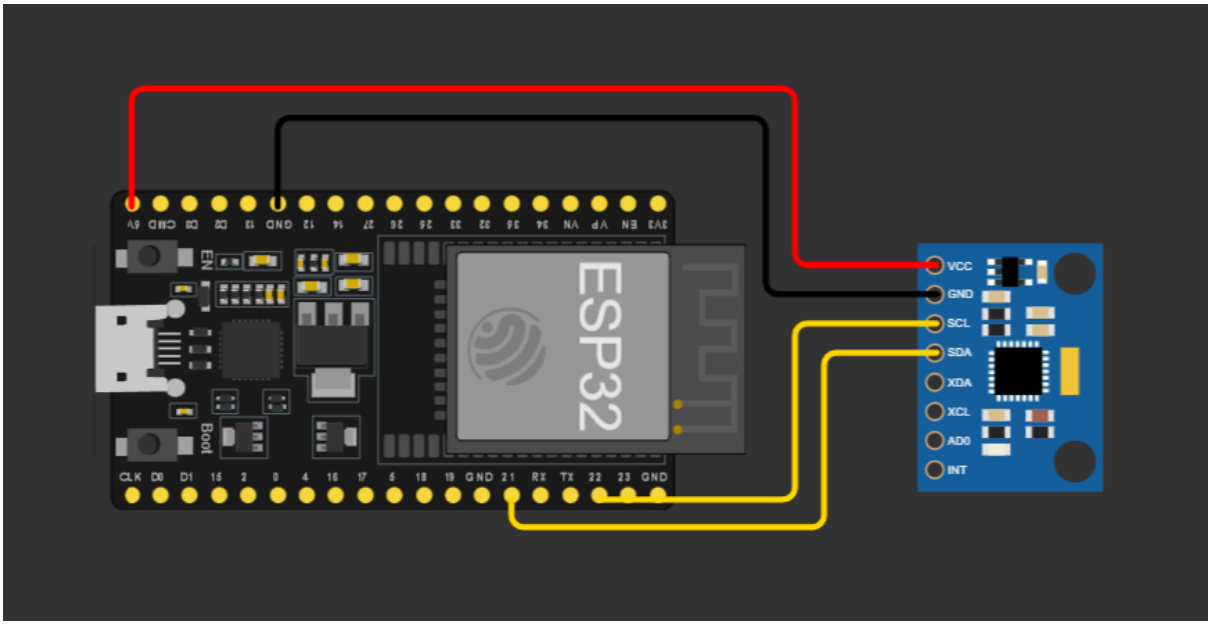
The below figure represents the pneumatic scheme of the stand,



2.3. System setup

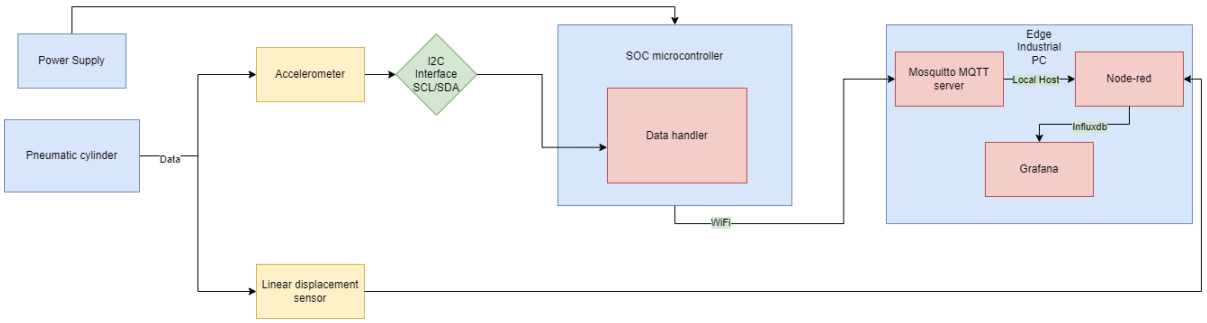
In order to mount the accelerometer on the tip of the cylinder rod, we designed a 3D model of a support





The connection between the accelerometer and the SOC microcontroller is made with an I2C(Inter-Integrated Circuit) connection, that is a serial communication protocol that uses only two wires SCL(Serial Clock Line), SDA(Serial Data Line) to connect different devices. The SCL synchronizes the data sent through the SDA by pulsing.

3.0. Implementation



The above figure(x) represents the basic principle of the system.

- The blue blocks represent the devices used.
- The red blocks represent the applications used.
- The yellow blocks represent the sensors.
- The green blocks represent the way data is shared between devices.

3.1. MQTT

Mqtt(Message Queuing Telemetry Transport) is a lightweight, publish/subscribe network protocol designed for connecting remote devices with a small code footprint and minimal network bandwidth, making it a cornerstone of Internet of Things(IoT) implementations. A central broker acts as a message hub, in the publish/subscribe model. Devices publish data to a designated topic on the broker, while others subscribe to a specific topic.

To enable efficient communication between IoT devices, it was necessary to create an MQTT server, acting as a central intermediary, managing the exchange of messages between connected devices.

Mosquitto is an open-source project and the ease of use made it the perfect solution for the role of an MQTT server for the broker.

We implemented the Mosquitto server on an Emerson Rxi2-LP Edge industrial PC, running in a docker container. We used the local IP of the device with the default port, which is 1883.

3.2. SOC(System On a Chip) microcontroller

We used an ESP-32 SOC microcontroller in order to power and process the data from the MPU6050 accelerometer. The key feature about this particular board is the capability of using WiFi. This opens the possibility to connect to the mqtt server and send the data forward to Node-Red.

4.0. Programming

The system uses distinct programming environments: Arduino IDE for the accelerometer and Node-Red for the LDS.

4.1. ESP32 program

. The source code in this IDE is written in C or C++.

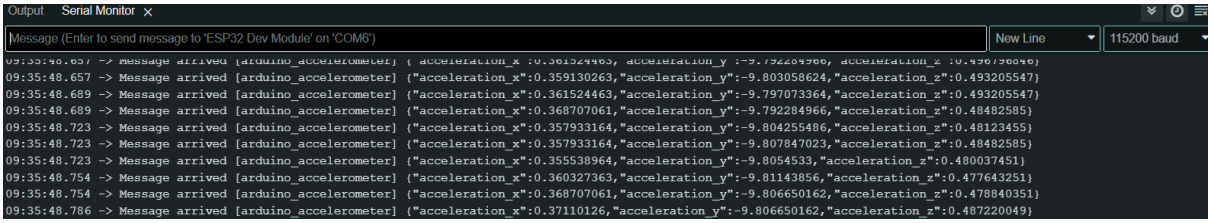
- In section S1 the variables and objects which are to be used are defined.
- S2: The Serial Monitor is initialized and ensures that the monitor is established before running the actual code. It checks if the MPU6050 accelerometer is detected by the system, printing a confirmation message in case of succes, after which the accelerometer range and bandwidth are set, otherwise it enters an infinite loop showing that it failed the connection. In the case of everything working well, it calls the setup_wifi() function, to connect to WiFi, sets up the MQTT client.
- S3: void setup_wifi() begins with showing a descriptive message to which WiFi is trying to connect, after which attempts to connect to the network, waits in a loop until the connection is established, then displays the succes in connecting.
- S4: void callback() is called when a message is received from the mosquitto server, printing the respective topic and message.
- S5: void reconnect() attempts the connection to the mosquitto server, succeeding in this action, it subscribes to all topics. Failing this action prints out a fail message.
- S6: void loop() connects to the mosquitto server, read the sensor data from the accelerometer, publishing the results to the MQTT broker in a JSON format.

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>
#include <ArduinoJson.h>

const char* ssID = ""; // WiFi name
const char* password = ""; // WiFi password
const char* mqttServer = ""; // server-ip
const int mqttPort = 1883; // default MQTT port

WiFiClient espClient; // wifi client object
PubSubClient client(espClient); // mqtt client object
Adafruit_MPU6050 mpu; // accelerometer object
```

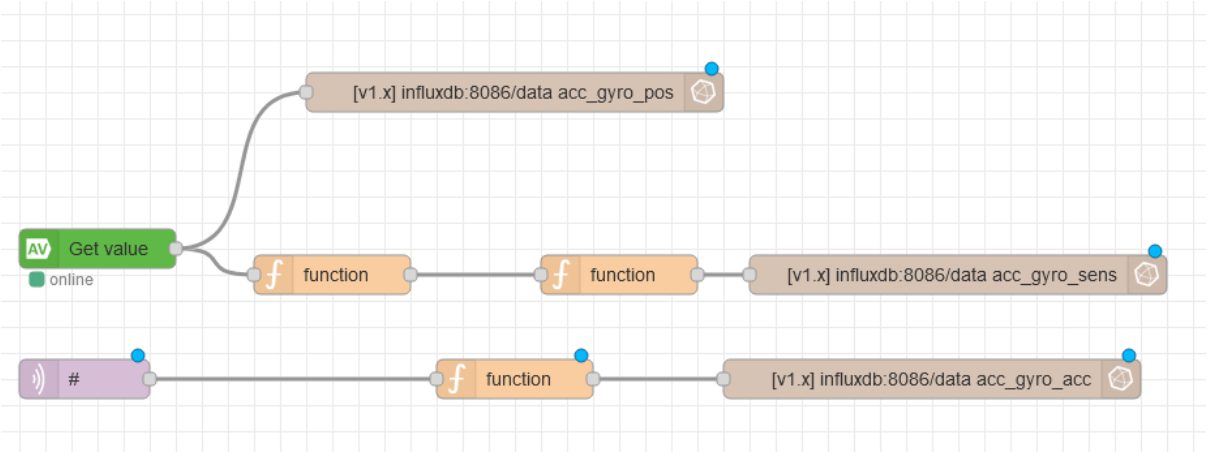
```
void setupWiFi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssID);

  WiFi.begin(ssID, password); // connect to wifi

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
```

```
 void reconnect() {  // connect to mqtt
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect("ESP32Client")) {
    Serial.println("connected");
    client.subscribe("#"); // subscribe to the topic to receive messages
    } else {
    Serial.print("failed");
    Serial.println(" try again in 5 seconds");
    delay(5000);
    }
  }
 }
void setup() {
  Serial.begin(115200);
  while (!Serial) // wait for the serial monitor to open
    delay(10);

  // Try to initialize!
  if (!mpu.begin()) { // check if the accelerometer is seen by the software
    while (1) {
    Serial.println("Failed to find MPU6050 chip");
    delay(5000);
    }
  }
  Serial.println("MPU6050 Found!"); // accelerometer found

  mpu.setAccelerometerRange(MPU6050_RANGE_2_G); // accelerometer range
  mpu.setGyroRange(MPU6050_RANGE_500_DEG); // accelerometer gyro range
  mpu.setFilterBandwidth(MPU6050_BAND_5_HZ); // accelerometer bandwidth

  delay(100);

  setupWiFi(); // connect to wifi
  client.setServer(mqttServer, mqttPort); // Set MQTT server and port
  client.setCallback(callback); // set callback function
}
```

```
86  void loop() {
87    if (!client.connected()) { // if it's not connected, connect
88      reconnect();
89    }
90    client.loop(); // ensuring the mqtt stays connected
91
92    sensors_event_t a, g, temp;
93    mpu.getEvent(&a, &g, &temp);
94
95    // create a JSON object
96    StaticJsonDocument<200> doc;
97    doc["acceleration_x"] = a.acceleration.x;
98    doc["acceleration_y"] = a.acceleration.y;
99    doc["acceleration_z"] = a.acceleration.z;
100   doc["gyro_x"] = g.gyro.x;
101   doc["gyro_y"] = g.gyro.y;
102   doc["gyro_z"] = g.gyro.z;
103   doc["temperature"] = temp.temperature;
104
105   // serialize the JSON object to a string
106   char jsonBuffer[512];
107   serializeJson(doc, jsonBuffer);
108
109   client.publish("arduino_accelerometer", jsonBuffer); // send the data to the broker
110
111   delay(1);
112 }
```



4.2. Node red



Linear sensor

Speed

```
 // Get the incoming message containing the position value

// Define the sensor output range and corresponding cylinder lengths
const sensorMinValue = 4050;
const sensorMaxValue = 29834;
const contractedLength = 130; // mm
const extendedLength = 500; // mm

if(msg.payload < 4050)
return null;

// Calculate the linear position based on sensor output (assuming linear relationship)
let position = contractedLength + ((msg.payload - sensorMinValue) / (sensorMaxValue - sensorMinValue)) * (extendedLength - contractedLength);

// Function to calculate speed (assuming you have previous position data and timestamps)
function calculateSpeed(previousPosition, previousTime) {
  if (!previousPosition || !previousTime) {
        // Handle initial state or missing data
        return 0;
  }

    const deltaTime = (new Date().getTime() - previousTime) / 1000; // Time difference in seconds (assuming time is in milliseconds)
  const distance = position - previousPosition; // Distance traveled in mm

  // Check for very small time differences to avoid division by zero
  if (Math.abs(deltaTime) < 0.001) {
        return 0;
```

```
  }

  const speed = distance / deltaTime; // Speed in mm/s

  return speed;
}

// Retrieve previous position and time data (modify based on your storage mechanism)
let previousPosition = context.get("previousPosition");
let previousTime = context.get("previousTime");

// Calculate speed
const speed = calculateSpeed(previousPosition, previousTime);

// Update previous position and time for future calculations
context.set("previousPosition", position);
context.set("previousTime", new Date().getTime());

// Set the output message payload with the calculated speed
msg.payload = speed;
msg.topic = "speed_linear_sensor"
// Send the message with the speed value
return msg;
```

Acceleration

```
// Retrieve previous speed and time data (modify based on your storage mechanism)
let previousSpeed = context.get("previousSpeed");
let previousTime = context.get("previousTime");

// Check for initial state or missing data
if (!previousSpeed || !previousTime) {
  // Set acceleration to 0 initially
  context.set("previousSpeed", msg.payload);
  context.set("previousTime", new Date().getTime());
  msg.payload = 0;
  return null;
}

const currentTime = new Date().getTime(); // Current time (assuming time is in milliseconds)

// Calculate time difference in seconds
const deltaTime = (currentTime - previousTime) / 1000;

// Check for very small time differences to avoid division by zero
if (Math.abs(deltaTime) < 0.001) {
```

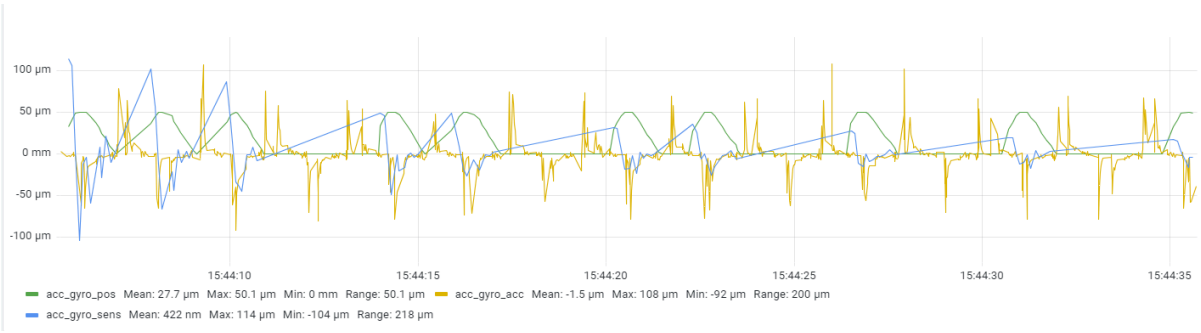
```
  msg.payload = 0;
  return msg;
}

// Calculate acceleration (assuming speed is in mm/s)
const acceleration = (msg.payload - previousSpeed) / deltaTime; // mm/s^2

// Update previous speed and time for future calculations
context.set("previousSpeed", msg.payload);
// context.set("previousTime", currentTime);

// Set the output message payload with the calculated acceleration
msg.payload = acceleration/1000;
msg.topic = "linear_sensor";
// Send the message with the acceleration value
return msg;
```

Accelerometer

```
        var lastTime = context.get("lastTime");
var lastSpeed = context.get("lastSpeed") || 0


// Parse the JSON string (assuming it's in msg.payload)
let jsonData;
try {
  jsonData = JSON.parse(msg.payload);
} catch (error) {
  node.error("Error parsing JSON: " + error.message, msg); // Send error to debug panel
  return msg; // Return the original message in case of error
}

// Extract acceleration_z value (adjust property name if necessary)
const accelerationZ = jsonData.acceleration_z;

// Update the message payload with just the acceleration_z value
msg.payload = -accelerationZ;
msg.topic = "arduino_acceleration"
// msg.payload = lastSpeed + msg.payload * ((new Date().getTime() - lastTime) / 1000.0);
// // msg.payload = (new Date().getTime() - lastTime)
// context.set("lastTime", new Date().getTime())
// context.set("lastSpeed", msg.payload)
// Send the modified message
return msg;
```

Results

The first graphic shows the displacement of the rod in time, given by the linear position sensor. Second graphic shows the graphic of acceleration measured by the accelerometer. The third graphic is the acceleration from the displacement.



Conclusion

While both sensors demonstrate similar levels of precision, the linear position sensor shows a higher accuracy. The accelerometer's lower accuracy is due to the higher polling rate, that introduces noise and sampling errors. A downside of the accelereromter can be the time that has to be calibrated before start reading the data, while the linear position sensor doesn't need that.

By looking at each sensor strong side they can have different applications, depending on the needings. The linear position sensor, can be ideal for precise tracking. This make it suitable for established, predictable system behavior. The accelerometer excels

at detecting rapid changes in motion, vibrations. This functionality is valuable for monitoring dynamic systems.