IS-311 Project Report

17-10-2023
Tingenes Internett

The A-Team
Aleksander Grimstad Mo, Joaquin Iannuzzi, William Peter Åredal

Table of Contents

Introduction

When you take a bus or a uber, or a taxi, it's always nice to know how comfortable the ride is, whether the ride is bumpy, the road has abrupt turns, or an aggressive chauffeur. This report goes through the process of creating a sensor array that measures car/bus ride comfort levels. From the initial brainstorming to the dataset visualisation. With data that encompasses a comprehensive set of metrics, aiding in the evaluation of ride comfort.

Brainstorming session

The group started to brainstorm possible project ideas right after forming. During the brainstorming session these ideas came up:

1. Self-driving car
2. Line following car
3. Echolocation through speaker and microphones
4. Sensor array that measures car/bus ride comfort levels
5. A LED light for indicating the price of power or another commodity from the web

Subsequently, a further investigation continued to find the prospects of the ideas. Some of these prospect discussions were about part requirements, practical use cases, and personal interests of the group members. After some back and forth between the ideas, the sensor array to measure car/bus ride comfort levels was chosen to be the best match. Having chosen an idea, it was possible to make a list of required components.

Hardware requests

Due to the lecturer mentioning that part-deliveries could take 1 or more weeks, it was decided that ordering parts as soon as possible was important. The first list of components and parts needed to begin on the project was made the following days after the first seminar. Underneath is the initial list of the parts and sensors required for the project:

- Arduino UNO R4
- Accelerometer to measure forces experienced across x, y and z axes
- Temperature sensor
- A microphone/decibel sensor
- Vibration sensor for sudden movements
- Wires for connecting modules to the Arduino

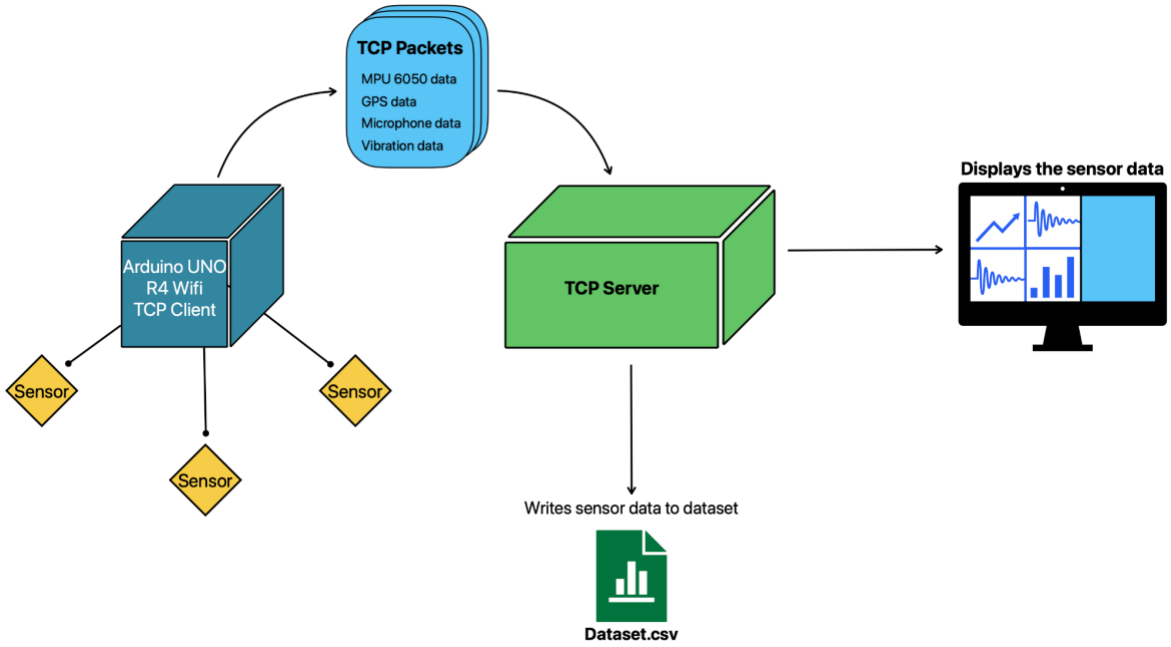
After making the component list, a request was sent to the lecturer for the part. Additionally, two Accelerometers were ordered to have one backup, because it was thought to be the most important sensor.

Til IoT sensor prosjektet som skal måle hvor komfortabel en bil/busstur er og sendingen av denne informasjonen tenker vi bruke;
- akselerometer for å måle kreftene i x, y, z aksene som passasjerer opplever.
- temperatur, fuktighet og decibel målere som ser på faktorer som kan gjøre passasjerer ukomfortable.
- vibrasjonsmåler for å prøve plukke opp om bilen/bussen kjører over hull i veien.

Liste over komponenter for prosjektet:
1x Arduino Uno (Kjell & company). 199,90kr
2x Accelerometer/Gyroscope (MPU6050). Fant til 27kr pr stk
1x Playknowlogy Liten modulpakke for Arduino (kjell & company). 299,90kr
1x Playknowlogy Komponent-kit for eksperimentering med Arduino (kjell & company). 299,90kr
- Her er vi ute etter modulene som kan måle vibrasjoner, temperatur, fuktighet, og decibel/støy

At a later date it was discovered that a GPS would enhance the system, as this would add information about the specific road sections where drive comfort is diminished. Allowing for easier identification of potholes, or poorly designed road segments. Additionally, the group discovered that the number of cables supplied in the delivered module packs were insufficient for the wiring of all modules. The calculated number of wires required for connecting all sensors was then also included in the request for the GPS module.

The system architecture



System architecture illustration

As seen in the illustration above, there are three sensors in the system: GPS, accelerometer, and microphone. These collect measurements that are sent directly to the Arduino UNO R4. The Arduino establishes a connection to the server, then sends TCP packets containing the sensor data. When the server receives the packets, it creates and writes the sensor data into a CSV file. Considering IoT terminology, the section of the system where the Arduino UNO R4 and sensors are located would be designated as being on the "edge" (*What Is Edge Computing?*, n.d.). The TCP server where data is received, stored and visualised would on the other hand be categorised as being the "cloud" in this IoT system (*What Is Cloud Computing?*, n.d.).

Network access

During the initial efforts to connect the client to the server over WI-FI resulted in discovery of some obstacles. These obstacles were related to the authentication to the university network. Therefore, the group opted for using mobile phone hotspots as a temporary substitute, ensuring the project development did not get bogged down.

Further into the project the group realised that running the system in a car/bus context would make any connection to “eduroam”, the university network, impossible. The system would thus be required to run over a portable network. Due to this, the group decided to continue using the hotspot configuration that was initially set up as a temporary substitute.

For the network protocol there were two choices, transmission control protocol (TCP) and user datagram protocol (UDP). TCP ensures there is a connection before sending the data, and ensures the data is in the correct order without errors. UDP on the other hand does not establish a connection before sending data, and does not ensure the data is in the correct order (*What Is the User Datagram Protocol (UDP)?*, n.d.). Because TCP requires more information in the packet headers to ensure data integrity, it can add a delay to the data transfer compared to UDP.

UDP offered advantages over TCP when it came to throughput. However, TCP provided a guarantee that 100% of the packets would arrive, and in the original order sent. Thus, the TCP protocol was in the end chosen as the more ideal fit for this system.

System development

In the initial phases of the project development, the group began by learning how each sensor would be connected to the Arduino. With each sensor's code residing in their own separate script. To verify that each sensor output was collected correctly, it's value(s) were outputted to the Arduino IDE terminal.

TCP client and server development

After this initial experience on how sensors were connected to the Arduino, the group began connecting multiple sensors simultaneously. Parallel to this process, development of the server and client began. Initially, this utilised the UDP protocol to simplify the setup and connection of these two. This was later swapped to utilise the TCP protocol, as this protocol ensures that no information is lost.

TCP achieves this by keeping track of the number of sent and received packages through "ack" and "seq". If any packages go missing, they will simply be sent again. This ensures the integrity of the sensor data being transmitted.

Sensor integration

Following the development and verification of the TCP client and server, work began on integrating the rest of the sensor code from the different scripts. Again, the transmission of these new sensor data values was again outputted to the server's terminal for verification. In addition to adding the code for the collection of sensor data, the group also had to modify the client and server code to specify what types of data (float, integer, string, etc.) was being transmitted.

At the end of this process, the group had made a TCP client that was able to establish a connection to the server through a three way handshake and transmit data from all sensors connected.

Pin schematic

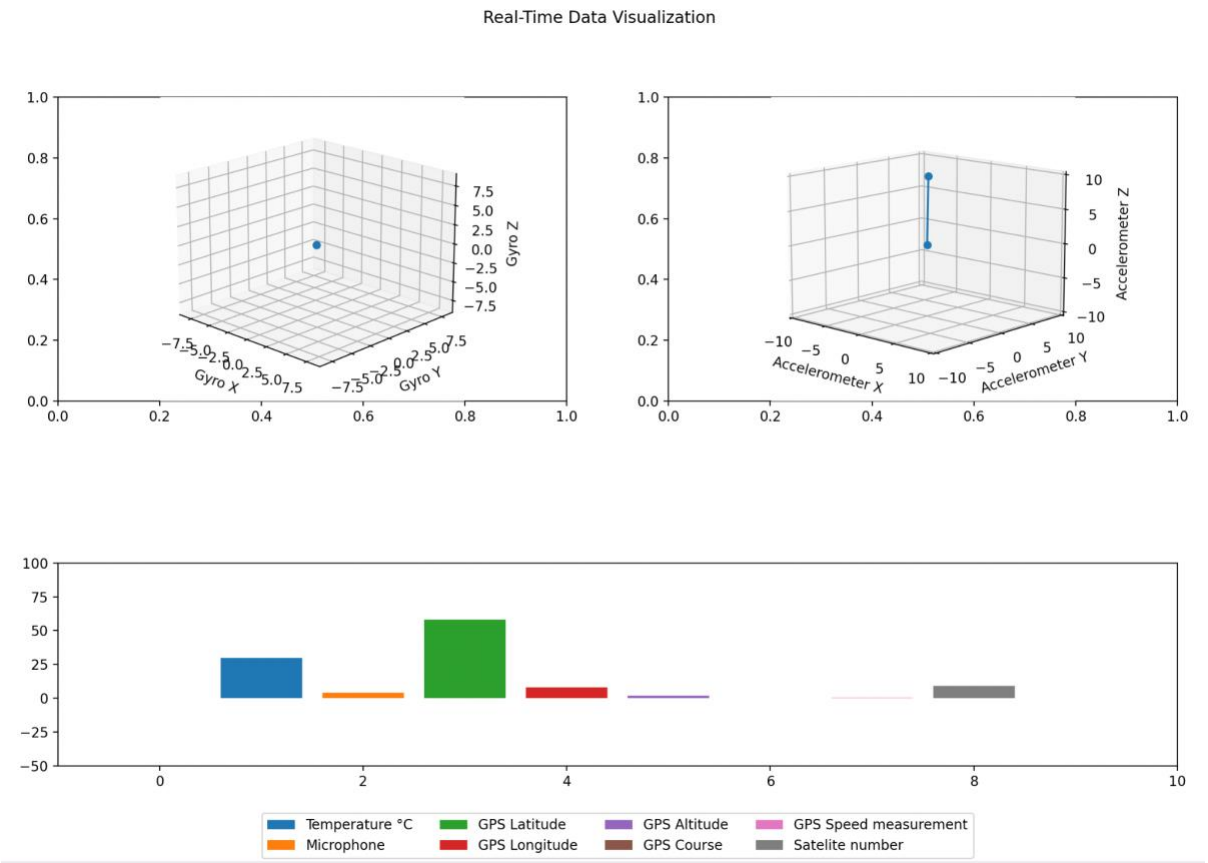
Below is a table showing the final pin schematic for the Arduino UNO R4 sensor pin connections on the sensor and its matching pin on the Arduino. Each corresponding pin connection is listed in numerical order to simplify future wiring.

Sensor module pins	Arduino pins
MPU6050: 1. VCC 2. GND 3. SCL 4. SDA	 1. 5V 2. GND 3. A5 4. A4
GPS: 1. VCC 2. GND 3. RX 4. TX	 1. 5V 2. GND 3. ~3 4. 2
Microphone: 1. + 2. G 3. A0	 1. 5V 2. GND 3. A0

Arduino-sensor pin schematic illustration

Real-time data visualisation

Finally in the system development, after having made a TCP server and client, work was undergone to visualise the sensor data in real-time on the server. The visualisation (shown below) offered helpful insights into the sensor measurements during the creation of the dataset. Allowing the group to see if sensors were returning anomalous measurements, or reasonable data.



Server Real-time data visualisation illustration

Calibration

For Arduino projects involving sensors, calibration servers as an important step to significantly enhance the precision of the collected data. Sensor modules can have inherent discrepancies in their measurements. This variance can stem from factors such as manufacturing defects, wear and tear, or damage during transportation. In this project, the group employed a systematic philosophy for the calibration of each sensor module.

A. Was the correct value known? If so, the sensor was calibrated to reflect this.
B. Was the correct value unknown, or unmeasurable? If so, the sensor was left as is. Variability in it's measurements would instead be looked at.

Most sensors landed in category A, with the microphone and temperature sensors landing in category B. To calibrate these properly, additional measurement tools would have to serve as reference point "standards". Therefore, they were left as is.

Client code

Below is the final code used by the TCP Client to connect to the sensors, receive and transmit data to the server. Where and how these key processes are performed will be explained after the code section below.

Arduino Client Code

```
#include <stdio.h>
#include <stdlib.h>

#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>

#include <WiFiS3.h>
#include <TinyGPS++.h>
#include <SoftwareSerial.h>

const char* ssid = "";      // your Wi-Fi network SSID
(network name)
const char* password = "";  // your Wi-Fi password
const char* server_ip = "";  // use ifconfig to identify mac
ip by finding 'en0', and its 'inet' ip address
const int server_port = 12345;            // the port number

const int microphonePin = A0; // Analog pin where the
microphone is connected
int microphoneValue;

WiFiClient client;  // Create a WiFiClient object

Adafruit_MPU6050 mpu;

SoftwareSerial gpsSerial(2, 3); // RXD pin 2, TXD pin 3
TinyGPSPlus gps;
```

Arduino Client Functions

```
void printValues() {
  // Print the calibrated mpu values
  Serial.print("Calibrated Acceleration X: ");
  Serial.print(calibratedAccelX);
  Serial.print(", Y: ");
  Serial.print(calibratedAccelY);
  Serial.print(", Z: ");
  Serial.print(calibratedAccelZ);
  Serial.println(" m/s^2");

  Serial.print("Calibrated Rotation X: ");
  Serial.print(calibratedGyroX);
  Serial.print(", Y: ");
  Serial.print(calibratedGyroY);
  Serial.print(", Z: ");
  Serial.print(calibratedGyroZ);
  Serial.println(" rad/s");

  Serial.print("Temperature: ");
  Serial.print(temp.temperature);
  Serial.println(" degC");

  // Print the mic values
  Serial.print("Microphone: ");
  Serial.println(microphoneValue);

  // Print the gps values
  Serial.print("Latitude: ");
  Serial.println(latitude, 6);
```

```
sensors_event_t a, g, temp;

// Calibrated offsets (replace with your own values)
float accelXOffset = 0.2;
float accelYOffset = -0.17;
float accelZOffset = 4.92335;

float gyroXOffset = -0.08;
float gyroYOffset = 0.00;
float gyroZOffset = -0.04;

float calibratedAccelX;
float calibratedAccelY;
float calibratedAccelZ;

float calibratedGyroX;
float calibratedGyroY;
float calibratedGyroZ;


// Declare global variables for GPS values
double latitude = 0.0;
double longitude = 0.0;
double altitude = 0.0;
double course = 0.0;
double speed = 0.0;
int satellites = 0;

bool areValidValues = false;


void setup() {
  Serial.begin(115200); // Initialize serial communication
  initMpu();
  initGps();
  initWifi();

}

void initMpu() {
  if (!mpu.begin()) {
    Serial.println("Failed to find MPU6050 chip");
    while (1) {
      delay(10);
    }
  }
  Serial.println("MPU6050 Found!");
  // set accelerometer range to +-8G
  mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
  // set gyro range to +- 500 deg/s
  mpu.setGyroRange(MPU6050_RANGE_500_DEG);
  // set filter bandwidth to 21 Hz
  mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);

  delay(100);
}

void initGps() {
  gpsSerial.begin(9600); // GPS module communication
}

void initWifi() {
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
}

void loop() {

  Serial.println("----------------------------");

  getMpu();
  getMic();
  getGps();

  if (areValidValues) {
    // commented out 'printValues()', only required for
debugging
    //printValues();
    sendData();
  }

  areValidValues = 0;

  delay(100);


}
```

```
  Serial.print("Longitude: ");
  Serial.println(longitude, 6);
  Serial.print("Altitude: ");
  Serial.println(altitude, 6);
  Serial.print("Course: ");
  Serial.println(course, 6);
  Serial.print("Speed: ");
  Serial.println(speed, 6);
  Serial.print("Satellites: ");
  Serial.println(satellites, 6);
  Serial.print("Valid?: ");
  Serial.println(areValidValues);
}

void getMic() {
  int mn = 1024;
  int mx = 0;

  for (int i = 0; i < 10; ++i) {
    int val = analogRead(microphonePin);
    mn = min(mn, val);
    mx = max(mx, val);
  }

  int delta = mx - mn;
  microphoneValue = delta;
}

void getMpu() {
  mpu.getEvent(&a, &g, &temp);

  // Apply offset compensation and create variables for
calibrated values
  calibratedAccelX = a.acceleration.x - accelXOffset;
  calibratedAccelY = a.acceleration.y - accelYOffset;
  calibratedAccelZ = a.acceleration.z - accelZOffset;

  calibratedGyroX = g.gyro.x - gyroXOffset;
  calibratedGyroY = g.gyro.y - gyroYOffset;
  calibratedGyroZ = g.gyro.z - gyroZOffset;
}

void getGps() {
  while (gpsSerial.available() > 0) {
    if (gps.encode(gpsSerial.read())) {

      areValidValues = gps.location.isValid() &&
gps.course.isValid() && gps.altitude.isValid() &&
gps.speed.isValid() && gps.satellites.isValid();

      if (areValidValues) {

        // Update global variables with GPS data
        latitude = gps.location.lat();
        longitude = gps.location.lng();
        altitude = gps.altitude.meters();
        course = gps.course.deg();
        speed = gps.speed.kmph();
        satellites = gps.satellites.value();

        break;
      }
    }
  }
}

// Function to send a variable as bytes to the server
template <typename T>
void sendVariableAsBytes(Client &client, T &variable) {
  byte bytes[sizeof(T)];
  memcpy(bytes, &variable, sizeof(T));
  client.write(bytes, sizeof(T));
}

void sendData() {
  if (client.connect(server_ip, server_port)) {
    Serial.println("Connected to server");

    // Send accelerometer data
    sendVariableAsBytes(client, calibratedAccelX);
    sendVariableAsBytes(client, calibratedAccelY);
    sendVariableAsBytes(client, calibratedAccelZ);

    // Send gyroscope data
    sendVariableAsBytes(client, calibratedGyroX);
    sendVariableAsBytes(client, calibratedGyroY);
    sendVariableAsBytes(client, calibratedGyroZ);

    // Send temperature data
    sendVariableAsBytes(client, temp.temperature);

    // Send microphone data
    sendVariableAsBytes(client, microphoneValue);

    // Send GPS data
    sendVariableAsBytes(client, latitude);
    sendVariableAsBytes(client, longitude);
    sendVariableAsBytes(client, altitude);
    sendVariableAsBytes(client, course);
    sendVariableAsBytes(client, speed);
    sendVariableAsBytes(client, satellites);

    client.stop();  // Disconnect from the server
  } else {
    Serial.println("Connection failed");
    delay(1000);    // Wait for 1 seconds before retrying
  }
```

Data collection and transmission

In this code segment, the client initially establishes a connection to the sensors, then the WI-FI.

```
void setup () {
  Serial.begin(115200); // Initialize serial communication
  initMpu();
  initGps();
  initWifi();

}
```

In the following code segment code section, the client utilises multiple ‘get’ functions to collect all connected sensors data. Prior to these values being sent, a check is performed through ‘areValidValues’.

```
void loop() {

  Serial.println("----------------------------");

  getMpu();
  getMic();
  getGps();

  if (areValidValues) {
    // commented out 'printValues()', only required for debugging
    //printValues();
    sendData();
  }

  areValidValues = 0;

  delay(100);


}
```

This boolean variable starts off as ‘false’, and is flipped to ‘true’ in the ‘getGps()’ function once it has collected enough GPS transmissions (NMEA 0183 sentences) for a GPS data point. NMEA 0183 is a standard for GPS receivers to communicate with each other and other navigation devices (“NMEA 0183,” 2023).

When valid, this is then converted to bytes depending on its type, and sent to the server using the ‘sendData()’ function. This function utilises ‘sendVariableAsBytes’, shown below, for the conversion to bytes, depending on the data type.

```
// Function to send a variable as bytes to the server
template <typename T>
void sendVariableAsBytes(Client &client, T &variable) {
  byte bytes[sizeof(T)];
  memcpy(bytes, &variable, sizeof(T));
  client.write(bytes, sizeof(T));
}
```

Server code

Below is the final code used by the TCP server to receive data from the client, which is in turn written to a dataset file (csv) and visualised in a plot. Where and how these key processes are achieved will be explained further after the following code section below.

Python Server Code

```
from datetime import datetime
import csv
import socket
import struct
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec


# Define the IP address and port to listen on
host = ''  # Replace with the actual IP address of your Mac (use 'ifconfic' in the terminal, and find the ip under 'en0')
port = 12345  # Replace with the desired port number

# Create a socket object
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Bind the socket to the address and port
server_socket.bind((host, port))
# Listen for incoming connections (maximum 1 connection at a time)
server_socket.listen(1)
print(f"Listening for incoming connections on {host}:{port}")


# Defines the figure for the data plot animations
gyro_measurement_limit = 8.7266
accelerometer_measurement_limit = 10

# 3D plots for data visualization
fig, axs = plt.subplots(2, 2, figsize=(10, 8))
fig.suptitle('Real-Time Data Visualization')
# Remove the second row's subplots
for ax in axs[1, :]:
    ax.remove()

# Subplots for gyro and accelerometer data
ax1 = fig.add_subplot(2, 2, 1, projection='3d')
ax2 = fig.add_subplot(2, 2, 2, projection='3d')

# Subplot for rest of data
gs = gridspec.GridSpec(2, 1, height_ratios=[2, 1])
ax3 = fig.add_subplot(gs[1])


# Defines the parameters for the dataset filename and column names
current_datetime = datetime.now().strftime('%d-%m-%Y__%H-%M')
dataset_name = f'{current_datetime}_dataset.csv'
column_headers = [
    'X_DATA', 'Y_DATA', 'Z_DATA', 'GYRO_X_DATA', 'GYRO_Y_DATA', 'GYRO_Z_DATA', # Accelerometer and Gyro data
    'TEMP_DATA', # Temperature data
    'MIC_DATA', # Microphone data
    'LATITUDE', 'LONGITUDE', 'ALTITUDE', 'COURSE', 'SPEED', 'SATELLITES' # GPS data
]
with open(dataset_name, 'w+', newline='') as csvFile:
    csv_writer = csv.writer(csvFile)

    # Checks if the file has column headers
    csvFile.seek(0)
    first_character = csvFile.read(1)
    if not first_character:
        csv_writer.writerow(column_headers)

    while True:
        # Accept incoming connection
        client_socket, client_address = server_socket.accept()
        print("------------------------")
        print(f"Accepted connection from {client_address}")

        # Receive and unpack the sensor data
        x_data = client_socket.recv(4)
        y_data = client_socket.recv(4)
        z_data = client_socket.recv(4)
        gyro_x_data = client_socket.recv(4)
        gyro_y_data = client_socket.recv(4)
        gyro_z_data = client_socket.recv(4)
        temp_data = client_socket.recv(4)

        mic_data = client_socket.recv(4)

        lat_bytes = client_socket.recv(8)
        long_bytes = client_socket.recv(8)
        alt_bytes = client_socket.recv(8)
        course_bytes = client_socket.recv(8)
        speed_bytes = client_socket.recv(8)
        sat_bytes = client_socket.recv(4)

        if (
            len(x_data) == len(y_data) == len(z_data) == len(gyro_x_data) ==
            len(gyro_y_data) == len(gyro_z_data) == len(temp_data) ==
            len(mic_data) == len(sat_bytes) == 4 and len(lat_bytes) == len(long_bytes) == len(alt_bytes) == len(course_bytes) ==
```

```
            len(speed_bytes) == 8
        ):
            x = struct.unpack('f', x_data)[0]
            y = struct.unpack('f', y_data)[0]
            z = struct.unpack('f', z_data)[0]
            gyro_x = struct.unpack('f', gyro_x_data)[0]
            gyro_y = struct.unpack('f', gyro_y_data)[0]
            gyro_z = struct.unpack('f', gyro_z_data)[0]
            temp = struct.unpack('f', temp_data)[0]

            mic = struct.unpack('I', mic_data)[0]

            lat = struct.unpack('d', lat_bytes)[0]
            long = struct.unpack('d', long_bytes)[0]
            alt = struct.unpack('d', alt_bytes)[0]
            course = struct.unpack('d', course_bytes)[0]
            speed = struct.unpack('d', speed_bytes)[0]
            sat = struct.unpack('I', sat_bytes)[0]

            # Get the current datetime with precision
            current_datetime = datetime.now().strftime('%d-%m-%Y__%H-%M-%S-%f')

            # Writes the data to a new row in the csv file
            data_row = [
                current_datetime,
                x, y, z, gyro_x, gyro_y, gyro_z, temp,
                mic,
                lat, long, alt, course, speed, sat
            ]
            csv_writer.writerow(data_row)


            # Creates the gyro and accelerometer animation
            ax1.cla()
            ax1.set_xlabel('Gyro X')
            ax1.set_ylabel('Gyro Y')
            ax1.set_zlabel('Gyro Z')
            ax1.set_xlim([-gyro_measurement_limit, gyro_measurement_limit])  # X-axis limits
            ax1.set_ylim([-gyro_measurement_limit, gyro_measurement_limit])  # Y-axis limits
            ax1.set_zlim([-gyro_measurement_limit, gyro_measurement_limit])  # Z-axis limits
            # Plot the new gyro data as an orientation from 0, 0, 0
            ax1.plot([0.0, gyro_x], [0.0, gyro_y], [0.0, gyro_z], marker='o', markersize=5)

            ax2.cla()
            ax2.set_xlabel('Accelerometer X')
            ax2.set_ylabel('Accelerometer Y')
            ax2.set_zlabel('Accelerometer Z')
            ax2.set_xlim([-accelerometer_measurement_limit, accelerometer_measurement_limit])  # X-axis limits
            ax2.set_ylim([-accelerometer_measurement_limit, accelerometer_measurement_limit])  # Y-axis limits
            ax2.set_zlim([-accelerometer_measurement_limit, accelerometer_measurement_limit])  # Z-axis limits
            # Plot the new accelerometer data as an orientation from 0, 0, 0
            ax2.plot([0.0, x], [0.0, y], [0.0, z], marker='o', markersize=5)

            # Creates the animation for the other data
            ax3.cla()
            ax3.set_xlim([-1, 10])  # Y-axis limits
            ax3.set_ylim([-50, 100])  # Y-axis limits
            ax3.bar(1, temp, label='Temperature °C')
            ax3.bar(2, mic, label='Microphone')
            ax3.bar(3, lat, label='GPS Latitude')
            ax3.bar(4, long, label='GPS Longitude')
            ax3.bar(5, alt, label='GPS Altitude')
            ax3.bar(6, course, label='GPS Course')
            ax3.bar(7, speed, label='GPS Speed measurement')
            ax3.bar(8, sat, label='Satelite number')
            ax3.legend(loc='upper center', bbox_to_anchor=(0.5, -0.2), ncol=4)

            # Display the real-time animation
            plt.pause(0.01) # shows frames with 10 milliseconds in between

            '''
            print(f"Received Acceleration X: {x}, Y: {y}, Z: {z}")
            print(f"Received Gyro X: {gyro_x}, Y: {gyro_y}, Z: {gyro_z}")
            print(f"Received Temperature: {temp}")

            print(f"Received Microphone: {mic}")

            print(f"Received Latitude: {lat}")
            print(f"Received Longitude: {long}")
            print(f"Received Altitude: {alt}")
            print(f"Received Course: {course}")
            print(f"Received Speed: {speed}")
            print(f"Received Satellites: {sat}")
            '''

        # Close the client socket
        client_socket.close()
```

Data receival and processing

In this code section, the server establishes a TCP connection through a three-way handshake. Then transmitted bytes from the client are received and unpacked. Finally, these are converted to the corresponding python data types.

```
# Accept incoming connection
client_socket, client_address = server_socket.accept()
print("-----------------------")
print(f"Accepted connection from {client_address}")

# Receive and unpack the sensor data
x_data = client_socket.recv(4)
y_data = client_socket.recv(4)
z_data = client_socket.recv(4)
gyro_x_data = client_socket.recv(4)
. . .

    x = struct.unpack('f', x_data)[0]
    y = struct.unpack('f', y_data)[0]
    z = struct.unpack('f', z_data)[0]
    gyro_x = struct.unpack('f', gyro_x_data)[0]
    . . .
```

In the following code segment, it is shown where data is written to a dataset (csv file). Values are entered into a list that then is written using the ‘csv’ library’s ‘writerow()’ function. The index column value uses the server’s current datetime, creating a time series.

```
    # Get the current datetime with precision
    current_datetime = datetime.now().strftime('%d-%m-%Y__%H-%M-%S-%f')

    # Writes the data to a new row in the csv file
    data_row = [
        current_datetime,
        x, y, z, gyro_x, gyro_y, gyro_z, temp,
        mic,
        lat, long, alt, course, speed, sat
    ]
    csv_writer.writerow(data_row)
```

Real-time data visualisation

In addition to the data being written to a dataset file, it's also visualised in real time using the library 'matplotlib'. The visualisation is achieved by establishing three subplots, represented by the variables 'ax1', 'ax2', and 'ax3'. The two first are in three dimensions (x, y and z) for visualising acceleration forces and gyro orientation data from the MPU6050. The final bottom plot is wider and is used to plot the rest of the sensor measurements in a bar plot.

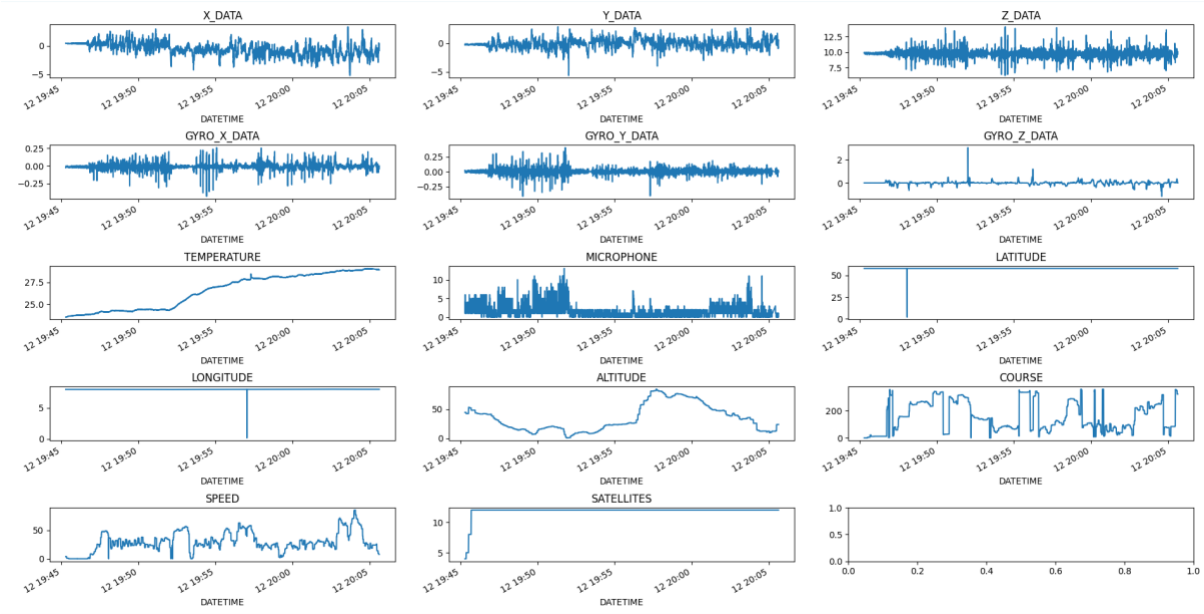
```
# 3D plots for data visualization
fig, axs = plt.subplots(2, 2, figsize=(10, 8))
fig.suptitle('Real-Time Data Visualization')

# Subplots for gyro and accelerometer data
ax1 = fig.add_subplot(2, 2, 1, projection='3d')
ax2 = fig.add_subplot(2, 2, 2, projection='3d')

# Subplot for rest of data
gs = gridspec.GridSpec(2, 1, height_ratios=[2, 1])
ax3 = plt.subplot(gs[1])

. . .

            ax1.cla()
            . . .
            # Plot the new gyro data as an orientation from 0, 0, 0
            ax1.plot([0.0, gyro_x], [0.0, gyro_y], [0.0, gyro_z], marker='o', markersize=5)

            ax2.cla()
            . . .
            # Plot the new accelerometer data as an orientation from 0, 0, 0
            ax2.plot([0.0, x], [0.0, y], [0.0, z], marker='o', markersize=5)

            # Creates the animation for the other data
            ax3.cla()
            . . .
            ax3.bar(1, temp, label='Temperature °C')
            . . .
            ax3.bar(8, sat, label='Satelite number')
            ax3.legend(loc='upper center', bbox_to_anchor=(0.5, -0.2), ncol=4)

            # Display the real-time animation
            plt.pause(0.01) # shows frames with 10 milliseconds in between
```

Dataset visualisation

After the complete system development was completed, a system test was undergone with a car ride. From this car ride, a dataset was collected. To visualise the data from the car ride, the group then wrote a simple script which plots every sensor's data. Here the dataset '12-10-2023__19-45_dataset.csv' is loaded and plotted using pandas.

```
import pandas as pd
import matplotlib.pyplot as plt

# Load the data into a DataFrame
df = pd.read_csv('12-10-2023__19-45_dataset.csv')

# Set the DATETIME column as the index
df['DATETIME'] = pd.to_datetime(df['DATETIME'], format='%d-%m-%Y__%H-%M-%S-%f')
df.set_index('DATETIME', inplace=True)

# Create subplots
fig, axs = plt.subplots(5, 3, figsize=(15, 10))

# Plot each column
df['X_DATA'].plot(ax=axs[0, 0], title='X_DATA')
df['Y_DATA'].plot(ax=axs[0, 1], title='Y_DATA')
df['Z_DATA'].plot(ax=axs[0, 2], title='Z_DATA')
df['GYRO_X_DATA'].plot(ax=axs[1, 0], title='GYRO_X_DATA')
df['GYRO_Y_DATA'].plot(ax=axs[1, 1], title='GYRO_Y_DATA')
df['GYRO_Z_DATA'].plot(ax=axs[1, 2], title='GYRO_Z_DATA')
df['TEMPERATURE'].plot(ax=axs[2, 0], title='TEMPERATURE')
df['MICROPHONE'].plot(ax=axs[2, 1], title='MICROPHONE')
df['LATITUDE'].plot(ax=axs[2, 2], title='LATITUDE')
df['LONGITUDE'].plot(ax=axs[3, 0], title='LONGITUDE')
df['ALTITUDE'].plot(ax=axs[3, 1], title='ALTITUDE')
df['COURSE'].plot(ax=axs[3, 2], title='COURSE')
df['SPEED'].plot(ax=axs[4, 0], title='SPEED')
df['SATELLITES'].plot(ax=axs[4, 1], title='SATELLITES')


# Adjust layout
plt.tight_layout()

# Show the plots
plt.show()
```

Which results in the following plot:



Dataset visualisation illustration

References

NMEA 0183. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=NMEA_0183&oldid=1161839147

What is cloud computing? | IBM. (n.d.). Retrieved October 17, 2023, from https://www.ibm.com/topics/cloud-computing

What is edge computing? | IBM. (n.d.). Retrieved October 17, 2023, from https://www.ibm.com/topics/edge-computing

What is the User Datagram Protocol (UDP)? (n.d.). Cloudflare. Retrieved October 17, 2023, from https://www.cloudflare.com/learning/ddos/glossary/user-datagram-protocol-udp/