Q1.

Looking at 3 lines on a time series plot with all time data is quite hard to visualize. For that, I have converted the 3 axes x, y, z accelerometer into the Linear Accelerometer. According to Science Buddies (n.d.), the formula for Linear Accelerometer is `linear acceleration = sqrt{x^2 + y^2 + z^2}`. They also said that our phone has a magnetometer which measures the Earth’s magnetic fields and does some calculations with gravity to measure the linear acceleration. Although I could use the float directly but opt for using the above formula because we are dealing with x, y, z variables. To achieve the smoothness of I have used the build in parameter `transaction` in the `update\_layout`. The parameter ensures a smooth transaction for newly added data, and the graph doesn’t “jump” when it updates. Note that the API I presented below can work with any numeric data, not just accelerometer.

Q2.

Here is the wrapper class for Plotly Dash users to achieve smooth and real-time graph. The formula I stated in Question 1 can work not only with Accelerometer, but any kind of axes. The wrapper works by just import it to a python project. Here is an explanation on the class.

import threading

import pandas as pd

import numpy as np

from dash import Dash, dcc, html

import plotly.graph\_objs as go

from dash.dependencies import Input, Output

from plotly.subplots import make\_subplots

from datetime import datetime, timedelta

from typing import Callable, List, Dict, Optional

import csv

import time

import os

class live\_monitor:

    def \_\_init\_\_(

        self,

        data\_function: Callable[[], Dict[str, float]],

        data\_columns: List[str],

        update\_interval: int = 5000,

        plot\_title: str = "Live Data Monitoring",

        yaxis\_title: str = "Data Value",

        csv\_file: Optional[str] = None

    ) -> None:

        """Init a live\_monitor object

        Args:

            data\_function (Callable[[], Dict[str, float]]): Function that returns a dictionary of continuous data with keys of corresponding columns

            data\_columns (List[str]): List of keys expected in the data dictionary from the 'data\_function'

            update\_interval (int, optional): interval in milliseconds to update the plot. Defaults to 5000.

            plot\_title (str, optional): title of the plot. Defaults to "Live Data Monitoring".

            yaxis\_title (str, optional): y-axis title for the plot. Defaults to "Data Value".

            csv\_file (Optional[str], optional): Optional file path to save continuous data to csv. Defaults to None.

        """

        self.data\_function = data\_function

        self.data\_columns = data\_columns

        self.update\_interval = update\_interval

        self.plot\_title = plot\_title

        self.yaxis\_title = yaxis\_title

        self.csv\_file = csv\_file

        self.plot\_data: List[Dict[str, float]] = []

        self.csv\_file\_handle = None

        self.writer = None

        self.incoming\_data = []

        # If a CSV file is loaded, open and write

        if self.csv\_file:

            try:

                self.csv\_file\_handle = open(csv\_file, mode='a', newline='')

                self.writer = csv.writer(self.csv\_file\_handle)

                # Load existing CSV data into self.plot\_data

                self.load\_csv\_data()

            except FileNotFoundError:

                print(f"CSV file {self.csv\_file} not found. Starting with empty data.")

        else:

            print("No CSV file provided. Running without CSV logging.")

I first define the class and a constructor. I have defined some attributes and instance variables, which are carefully explained in the comments. I have also loaded the csv file into memory (if provided a csv file). If there isn’t a csv file, throw an exception.

    def save\_data\_to\_csv(self, row: Dict[str, float]) -> None:

        """Save a row of data to the CSV if provided"""

        if self.csv\_file and self.writer:

            # Check if the file is empty and write headers if needed

            if os.path.getsize(self.csv\_file) == 0:

                # Write the header row

                headers = ['Timestamp'] + self.data\_columns

                self.writer.writerow(headers)

            # Ensure the timestamp is the first item in the row

            timestamp = row.pop('Timestamp')  # Remove the timestamp from the dictionary

            row\_data = [timestamp] + [row[col] for col in self.data\_columns]  # Put timestamp first

            self.writer.writerow(row\_data)

            self.csv\_file\_handle.flush()

If provided an csv, if the csv is empty, then add header rows. For csv that has header row, this function saves a new row of data into the csv. It ensures that the `Timestamp` column is the first feature in the row. After the new row has been written, it will then flush to ensure that the data is stored immediately.

    def load\_csv\_data(self) -> None:

        """Load data from CSV into self.plot\_data"""

        if self.csv\_file:

            try:

                # Check if the file is empty

                if os.path.getsize(self.csv\_file) == 0:

                    print(f"CSV file {self.csv\_file} is empty. Starting with empty data.")

                    return

                df = pd.read\_csv(self.csv\_file)

                # Convert 'Timestamp' column to datetime, if present

                if 'Timestamp' in df.columns:

                    df['Timestamp'] = pd.to\_datetime(df['Timestamp'], format='%Y-%m-%d %H:%M:%S', errors='coerce')

                # Convert column listed in data\_column to numeric

                for col in self.data\_columns:

                    if col in df.columns:

                        df[col] = pd.to\_numeric(df[col], errors='coerce')

                # Update self.plot\_data by appending loaded data

                if not df.empty:

                    self.plot\_data = df.to\_dict(orient='records')

            except FileNotFoundError:

                print(f"CSV file {self.csv\_file} not found. Starting with empty data.")

            except pd.errors.EmptyDataError:

                print(f"CSV file {self.csv\_file} is empty. Starting with empty data.")

This function is optional, but when provided an csv, the function re-format the data by converting the `Timestamp` into time data type, the numeric columns into numeric. This ensures that the data is correctly formatted for plotting.

    def create\_dash\_app(self) -> Dash:

        """Create the Dash app for live monitoring"""

        app = Dash(\_\_name\_\_)

        app.layout = html.Div([

            dcc.Graph(id='live-graph'),

            dcc.Interval(id='graph-update', interval=self.update\_interval),

        ])

        @app.callback(Output('live-graph', 'figure'), [Input('graph-update', 'n\_intervals')])

        def update\_graph(n: int) -> go.Figure:

            """Updates the graph with new data and applies smoothing."""

            if len(self.plot\_data) > 0:

                df = pd.DataFrame(self.plot\_data)

                df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

                # Filter data for the past hour

                current\_time = datetime.now()

                df = df[df['Timestamp'] >= (current\_time - timedelta(hours=1))]

                # Calculate the magnitude of the chosen data column for monitoring

                df['Data'] = np.sqrt(sum(np.square(df[col]) for col in self.data\_columns))

                # Create a plot for combined data

                fig = make\_subplots(rows=1, cols=1, subplot\_titles=["Data magnitude"])

                fig.add\_trace(go.Scatter(

                    x=df['Timestamp'],

                    y=df['Data'],

                    mode='lines',

                    name='Data magnitude',

                    line=dict(shape='spline')), row=1, col=1)

                fig.update\_layout(

                    height=700,

                    title=self.plot\_title,

                    xaxis\_title="Timestamp",

                    yaxis\_title=self.yaxis\_title,

                    xaxis=dict(tickformat="%H:%M:%S"),

                    transition={

                        'duration': 2000,  # Duration of the transition in milliseconds

                        'easing': 'cubic-in-out'  # Smooth easing function

                    }

                )

                return fig

            else:

                print("Plot data is empty")

                return go.Figure()

        return app

This function creates and configs the Dash app for live monitoring. It first layout with `dcc.Graph` to display and `dcc.Interval`` to trigger periodic updates. Inside this function, there is a callback `update\_graph` function that is triggered every time the interval elapses. The function fetches data from `self.plot\_data` and perform calculations. It filters the data for the past hour, calculates the magnitude of the numeric data and creates a smooth line Plotly.

    def start\_dash(self) -> None:

        """Start the Dash server"""

        print("Starting Dash server...")

        app = self.create\_dash\_app()

        app.run\_server(debug=False)

This function runs the Dash server by calling the ` create\_dash\_app()` method.

    def main(self) -> None:

        """Main loop to fetch data from the 'data\_function' and store it"""

        while True:

            # Add sleep interval between data updates

            time.sleep(self.update\_interval / 1000)  # Convert ms to seconds

            # Get the new data

            new\_data = self.data\_function()

            new\_data['Timestamp'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

            # Append the new data to the plot data

            self.plot\_data.append(new\_data)

            # Add data to the CSV

            self.save\_data\_to\_csv(new\_data)

            print(self.plot\_data[-1])

            print(f"Loaded {len(self.plot\_data)} records from CSV.")

            # Load the data

            self.load\_csv\_data()

The main loop that runs based on the interval of data updates. With every cycle, it fetches new data using `data\_function`, add `Timestamp` to new data, save it to the csv, and append new data to plot data dictionary.

    def start(self) -> None:

        """Start the data monitoring process"""

        dash\_thread = threading.Thread(target=self.start\_dash)

        dash\_thread.start()

        # Start main loop in the current thread

        self.main()

Method to start the live monitoring by launching the Dash server

    def \_\_del\_\_(self):

        """Ensure that the CSV file is properly closed when the object is closed"""

        if self.csv\_file\_handle:

            self.csv\_file\_handle.close()

The destructor the ensures the csv file is closed when the `live\_monitor` object is deleted.

Now I will demonstrate calling the API in another python file. One thing to note is that in the `test2.py` where I didn’t have a predefined csv, the wrapper needs to save at least 2 samples for it to be able to plot. Also, in this demo, I have used the Magnetometer axes instead of Accelerometer to show that this function also works on other values.

import os

from monitor import live\_monitor

from arduino\_iot\_cloud import ArduinoCloudClient

import threading

import time

# Configuration

DEVICE\_ID = "912ead58-1ded-4c28-ab34-5ae0350d52e2"

SECRET\_KEY = "vGkeQIQVVUBZe2wDEj2#U3VFB"

x = y = z = 1.0

This is the configuration for the connection to Arduino cloud, and global variables for the axes.

# Callback functions on value of change event

def on\_X\_changed(client, value):

    global x

    x = value

    return x

def on\_Y\_changed(client, value):

    global y

    y = value

    return y

def on\_Z\_changed(client, value):

    global z

    z = value

    return z

These functions are callback functions that are triggered when new data is received from Arduino.

# Function that returns accelerometer data in dictionary

def get\_accelerometer\_data() -> dict():

    global x, y, z

    while x == 1.0 and y == 1.0 and z == 1.0:

        time.sleep(0.1)

    # print(f"x: {x}")

    # print(f"y: {y}")

    # print(f"z: {z}")

    return {

        "Magnetometer\_X": x,

        "Magnetometer\_Y": y,

        "Magnetometer\_Z": z

    }

This function is for the `data\_function` parameter in `start\_monitor` wrapper. The function returns a dictionary with keys as the axes name. There is a loop to check that none of the values are still in their initial value of 1.0 to avoid stale and duplicate data.

# Function to start the monitor

def start\_monitor():

    monitor = live\_monitor(

        data\_function=get\_accelerometer\_data,

        data\_columns=["Magnetometer\_X", "Magnetometer\_Y", "Magnetometer\_Z"],

        update\_interval=60000,

        plot\_title="Magnetometer Data Monitoring",

        yaxis\_title="Magnetometer Linear",

        csv\_file=os.path.join(

            r'C:\Users\tomde\OneDrive\Documents\Deakin\Deakin-Data-Science\T1Y2\SIT225 - Data Capture Technologies\Week 8 - Using smartphone to capture sensor data\8.2C\api',

            'data.csv'

        )

    )

    monitor.start()

Function that defines the parameters from the API class.

# Function to start the Arduino IoT Cloud client

def start\_client():

    print("Starting data collection...")

    # Instantiate Arduino cloud client

    client = ArduinoCloudClient(

        device\_id=DEVICE\_ID, username=DEVICE\_ID, password=SECRET\_KEY

    )

    # Register callbacks

    client.register("magnetometer\_X", value=None, on\_write=on\_X\_changed)

    client.register("magnetometer\_Y", value=None, on\_write=on\_Y\_changed)

    client.register("magnetometer\_Z", value=None, on\_write=on\_Z\_changed)

    # Start the client

    client.start()

This function handles the connection between Arduino Cloud and triggers the respective callbacks.

if \_\_name\_\_ == "\_\_main\_\_":

    client\_thread = threading.Thread(target=start\_client)

    monitor\_thread = threading.Thread(target=start\_monitor)

    client\_thread.start()

    monitor\_thread.start()

    client\_thread.join()

    monitor\_thread.join()

This is the main function where both the client and data monitor are in separate threads to allow them to run concurrently. This ensures that the program collects data from Arduino Cloud while plotting and saving it.

Q3.

<https://www.youtube.com/watch?v=oT57KtRFfvM>

Q4.

<https://github.com/tomadonna1/SIT225_2024T2/tree/main/Live%20smooth%20Plotly%20Dash%20update%20for%20smartphone%20accelerometer%20data>

References

Science Buddies (n.d.). *Accelerometer Technical Note*. [online] Science Buddies. Available at: <https://www.sciencebuddies.org/science-fair-projects/references/accelerometer>.