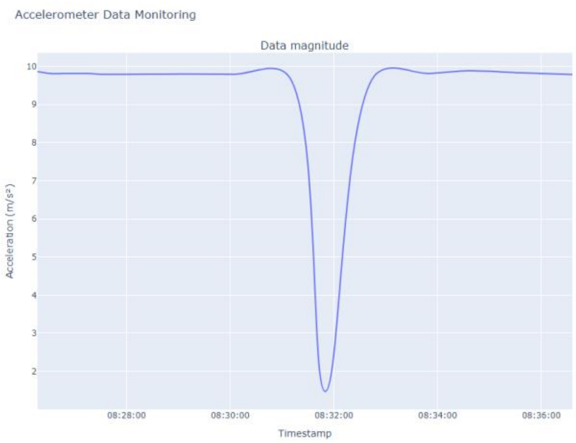
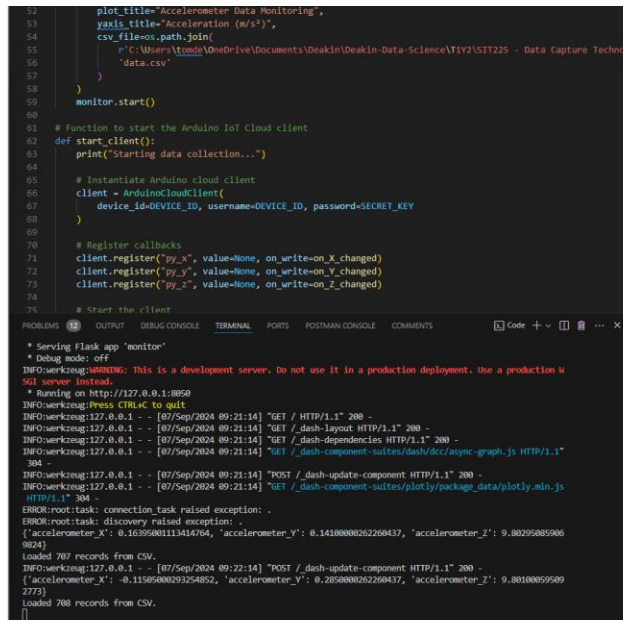
Q1.

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.

```python
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
```

```python
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
```

```python
        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.

```python
    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.

```python
    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
```

```python
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.

```python
# 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.

```python
# 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.

```python
# 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.

```python
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.