**Q2.**

Step 3 asks us to use 2 libraries to create dashboard for the Gyroscope data. Requirements are there should be a drop-down menu to choose between basic charts and statistical charts; option to select or de-select variables x, y, z; next and previous button for data navigation with a text box to input how large the navigation will be; a table along side the graph to see which data is being displayed; the interval to save data is 10s. The transition for newly added data I used the combination of cubic easing and interpolation. Cubic easing controls the rate of change during data transitions. Interpolation generates intermediate data points between current and new datasets to enable smooth updates. Combining the two, we have a visual appealing and user-friendly view. I managed to apply this to Bokeh, but Streamlit doesn’t seem to support it.

**Q3.**

**Bokeh framework.**

Data Reading Function

def get\_data() -> pd.DataFrame:

    data = pd.read\_csv(r'C:\Users\tomde\OneDrive\Documents\Deakin\Deakin-Data-Science\T1Y2\SIT225 - Data Capture Technologies\Week 6 - Visualisation - Plotly data dashboard\6.2HD\data\_gathering\data.csv',

                       header=None, names=['Timestamp', 'x', 'y', 'z'])

    return data

Function that returns a pandas dataframe with headers

Initial Data Setup

df = get\_data()

N = 100  # Number of samples to display

current\_start = max(0, len(df) - N)

last\_navigated = False  # Track if user navigated

# Axis options

axis\_options = ['x', 'y', 'z']

axis\_selection = CheckboxGroup(labels=axis\_options, active=[0, 1, 2])

selected\_axes = [axis\_options[i] for i in axis\_selection.active]

axis\_colors = {'x': 'blue', 'y': 'green', 'z': 'red'}

source = ColumnDataSource(df.iloc[current\_start:current\_start + N])

`N` is the initial number of samples to display and both the plots and the table

`current\_start` is the starting data index to slice, it also ensures the index doesn’t go below zero

`last\_navigated` is a Boolean flag to track the next and previous button

The axis options is for selecting axis on the plots.

`source` is Bokeh’s ColumnDataSource with slice dataframe based on `current\_start` and `N`.

Define table columns for Bokeh’s DataTable

columns = [

    TableColumn(field="Timestamp", title="Timestamp"),

    TableColumn(field="x", title="x"),

    TableColumn(field="y", title="y"),

    TableColumn(field="z", title="z"),]

The layout is pretty straightforward, `field` is for the data source, and `title` is the display name.

Cubic easing function for smooth transition

def cubic\_ease\_out(t: float, b: Union[float, np.array], c: Union[float, np.array], d: float) -> Union[float, np.array]:

    """Cubic easing function for smooth transition between data updates

    Args:

        t (float): current time

        b (Union[float, np.array]): starting value

        c (Union[float, np.array]): change in value

        d (float): total duration of the transition

    Returns:

        Union[float, np.array]: the eased value at time `t`

    """

    t = t / d - 1

    return c \* (pow(t, 3) + 1) + b

Function to interpolate data via cubic easing

def interpolate\_data(current: Dict[str, np.ndarray], new: Dict[str, np.ndarray], steps: int = 10) -> List[Dict[str, np.ndarray]]:

    """ Interpolate data points between current and new data using cubic easing

    Args:

        current (Dict[str, np.ndarray]): current data array x,y,z

        new (Dict[str, np.ndarray]): new data array x,y,z

        steps (int, optional): number of interpolation steps. Defaults to 10.

    Returns:

        List[Dict[str, np.ndarray]]: list of interpolated data dictionaries at each step

    """

interpolated = [] # empty list to store interpolated data dictionaries

    for i in range(steps):

        progress = i / steps # normalized progress of the interpolation at the current step

        interpolated\_step = {} # empty dictionary to hold interpolated values for each axis

        for axis in current.keys():

            interpolated\_step[axis] = cubic\_ease\_out(progress, current[axis], new[axis] - current[axis], 1)

        interpolated.append(interpolated\_step) # append the interpolated values to the list

    return interpolated

Create the DataTable

data\_table = DataTable(source=source, columns=columns, width=700, height=280)

Create the Figure

p = figure(x\_axis\_type="datetime", title="Accelerometer Data", height=400, width=700)

p.line(x='Timestamp', y='x', source=source, line\_width=2, color="blue", legend\_label="x-axis")

p.line(x='Timestamp', y='y', source=source, line\_width=2, color="green", legend\_label="y-axis")

p.line(x='Timestamp', y='z', source=source, line\_width=2, color="red", legend\_label="z-axis")

p.legend.location = 'bottom\_left'

`p` is the bokeh figure to plot the accelerometer data

`p.line` adds line glyphs to figure axis

`p.legend.location` change the location of legend

User interaction options

# Create Select widget for graph type

graph\_type = Select(title="Select graph type", value="Line", options=["Line", "Scatter", "Distribution"])

# TextInput to select the number of samples to display

sample\_input = TextInput(title="Number of samples to display", value=str(N))

# Navigation buttons for previous and next

prev\_button = Button(label="Previous", button\_type="success")

next\_button = Button(label="Next", button\_type="success")

Update Plot based on graph type selection

def update\_graph\_type():

    p.renderers = [] # Clears existing glyphs from the plot

    if graph\_type.value == "Line": # Adds line glyphs to the plot

        for axis in selected\_axes:

            p.line(x='Timestamp', y=axis, source=source, line\_width=2, color=axis\_colors[axis], legend\_label=f"{axis}-axis")

    elif graph\_type.value == "Scatter": # Adds scatter glyphs to the plot

        for axis in selected\_axes:

            p.scatter(x='Timestamp', y=axis, source=source, size=7, color=axis\_colors[axis], legend\_label=f"{axis}-axis")

    elif graph\_type.value == "Distribution": # Create histograms for each axis and adds quad glyphs as distributions to the plot

        for axis in selected\_axes:

            hist, edges = np.histogram(df[axis].iloc[current\_start:current\_start + N], bins=30)

            p.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:], fill\_color=axis\_colors[axis], line\_color=axis\_colors[axis], alpha=0.5, legend\_label=f"{axis}-axis")

    p.legend.location = 'bottom\_left'

Most of the code is self-explanatory. All 3 charts use the built-on functions from bokeh to plot. For the distribution plot, I need to create histograms for each axis and then plot.

Add Callback to Select widget (graph type selector)

graph\_type.on\_change('value', lambda attr, old, new: update\_graph\_type())

Attack the callback function to `value` property of the `graph\_type` widget

Update displayed data based on current range

def update\_data():

    global current\_start, N, df, last\_navigated, selected\_axes

    # Get the new data

    df = get\_data()

    if not last\_navigated:  # Check if user has not navigated

        current\_start = max(0, len(df) - N)  # Update to display the latest data

    new\_data\_slice = df.iloc[current\_start:current\_start + N]

    timestamps = new\_data\_slice['Timestamp'].values

    if not selected\_axes:

        # If no axes are selected, update only the Timestamp

        source.data = {'Timestamp': timestamps}

        data\_table.columns = get\_table\_columns(selected\_axes)

        update\_graph\_type()

        last\_navigated = False  # Reset navigation flag

        return

    # Get the current and new data slices

    current\_data = {axis: np.array(source.data[axis]) for axis in selected\_axes}

    new\_data = {axis: np.array(new\_data\_slice[axis]) for axis in selected\_axes}

    # Determine the minimum length

    min\_length = min(len(current\_data[selected\_axes[0]]), len(new\_data[selected\_axes[0]]))

    # Trim data to the minimum length

    for axis in selected\_axes:

        current\_data[axis] = current\_data[axis][:min\_length]

        new\_data[axis] = new\_data[axis][:min\_length]

    timestamps = timestamps[:min\_length]

    # Interpolate data over 10 steps

    interpolated\_steps = interpolate\_data(current\_data, new\_data, steps=10)

    # Apply the interpolation over time

    def apply\_interpolation(step=0):

        if step < len(interpolated\_steps):

            # Update data for selected axes and Timestamp

            data = {'Timestamp': timestamps}

            for axis in selected\_axes:

                data[axis] = interpolated\_steps[step][axis]

            source.data = data

            curdoc().add\_timeout\_callback(lambda: apply\_interpolation(step + 1), 100)  # Delay each step by 100ms

        else:

            # Ensure the final data is set correctly

            data = {'Timestamp': timestamps}

            for axis in selected\_axes:

                data[axis] = new\_data[axis]

            source.data = data

This function updates both data from table and plot.

Global variables are defined to allow for modification within the function.

`interpolated\_steps` is applied to both current and new data

`apply\_interpolation(step=0)` is a nested function that applies interpolated data to the source overtime. The interpolation step is delay at 100ms.

Callback for previous and next button

# Callback for the Previous button

def prev\_samples():

    global current\_start, N, last\_navigated

    last\_navigated = True  # Mark navigation occurred

    current\_start = max(0, current\_start - N)

    update\_data()

# Callback for the Next button

def next\_samples():

    global current\_start, N, last\_navigated

    last\_navigated = True  # Mark navigation occurred

    current\_start = min(len(df) - N, current\_start + N)

    update\_data()

Callback for the number of samples displayed

def update\_samples(attr, old, new):

    global N, current\_start

    try:

        N = int(sample\_input.value)

    except ValueError:

        N = 100

    current\_start = max(0, len(df) - N)

    update\_data()

This function is called when the value in `sample\_input` widget changes. If the value user type is not integer, then it uses the default `N`.

Callback for the axes update

def update\_axes(attr, old, new):

    global selected\_axes

    selected\_axes = [axis\_options[i] for i in axis\_selection.active]

    source.data = df.iloc[current\_start:current\_start + N][['Timestamp'] + selected\_axes].to\_dict('list')

    data\_table.columns = get\_table\_columns(selected\_axes)

    update\_graph\_type()

Attach callback to navigation buttons, input text box and the axis selection

sample\_input.on\_change('value', update\_samples)

prev\_button.on\_click(prev\_samples)

next\_button.on\_click(next\_samples)

axis\_selection.on\_change('active', update\_axes)

Layout for Bokeh and periodic callback for 2s

# Layout for Bokeh

layout = column(row(graph\_type, sample\_input), axis\_selection, p, row(prev\_button, next\_button), data\_table)

curdoc().add\_root(layout)

# Periodic callback for 2s

def periodic\_update():

    global last\_navigated

    if not last\_navigated:  # Only update if no navigation happened

        update\_data()

curdoc().add\_periodic\_callback(periodic\_update, 2000)

**Streamlit framework**

Dashboard setup

st.set\_page\_config(

    page\_title="Accelerometer dashboard",

    page\_icon="✅",

    layout="wide",

)

This configures the default setting of Streamlit app

Data reading function

def get\_data() -> pd.DataFrame:

    data = pd.read\_csv(r'C:\Users\tomde\OneDrive\Documents\Deakin\Deakin-Data-Science\T1Y2\SIT225 - Data Capture Technologies\Week 6 - Visualisation - Plotly data dashboard\6.2HD\data\_gathering\data.csv', header=None, names=['Timestamp', 'x', 'y', 'z'])

    data['Timestamp'] = pd.to\_datetime(data['Timestamp'], format='%Y%m%d%H%M%S')

    return data

Transition functions

The same as the previous dashboard. I can’t seem to make it work for Streamlit.

# Cubic easing function for smooth transition

def cubic\_ease\_out(t: float, b: Union[float, np.array], c: Union[float, np.array], d: float) -> Union[float, np.array]:

    """Cubic easing function for smooth transition between data updates

    Args:

        t (float): current time

        b (Union[float, np.array]): starting value

        c (Union[float, np.array]): change in value

        d (float): total duration of the transition

    Returns:

        Union[float, np.array]: the eased value at time `t`

    """

    t = t / d - 1

    return c \* (t\*\*3 + 1) + b

# Function to interpolate data using cubic easing

def interpolate\_data(current: Dict[str, np.ndarray], new: Dict[str, np.ndarray], steps: int = 10) -> List[Dict[str, np.ndarray]]:

    """ Interpolate data points between current and new data using cubic easing

    Args:

        current (Dict[str, np.ndarray]): current data array x,y,z

        new (Dict[str, np.ndarray]): new data array x,y,z

        steps (int, optional): number of interpolation steps. Defaults to 10.

    Returns:

        List[Dict[str, np.ndarray]]: list of interpolated data dictionaries at each step

    """

interpolated = [] # empty list to store interpolated data dictionaries

    for i in range(steps):

        progress = i / steps # normalized progress of the interpolation at the current step

        interpolated\_step = {} # empty dictionary to hold interpolated values for each axis

        for axis in current.keys():

            interpolated\_step[axis] = cubic\_ease\_out(progress, current[axis], new[axis] - current[axis], 1)

        interpolated.append(interpolated\_step) # append the interpolated values to the list

    return interpolated

Set up for the user interactions

# dashboard title

st.title("Accelerometer dashboard")

# chart selection

chart\_type = st.selectbox("Select chart type", ["Line Chart", "Scatter Plot", "Distribution Plot"])

# axis selection

axes\_options = st.multiselect("Select axis to plot", options=["x", "y", "z"], default=["x", "y", "z"])

# text box to display the number of samples to display

N = st.number\_input("Enter number of samples to display", min\_value=1, max\_value=len(get\_data()), value=100)

Navigation control

The next and previous button

# Placeholder for the graph and table

place\_holder = st.empty()

# Initialize variables for navigation

start\_index = st.session\_state.get('start\_index', len(get\_data())-50)

# Navigation buttons for previous and next

col1, col2 = st.columns([1, 1])

with col1:

    if st.button("Previous"):

        start\_index = max(0, start\_index - N)  # Go backward but don't go negative

        st.session\_state['start\_index'] = start\_index

with col2:

    if st.button("Next"):

        start\_index = min(len(get\_data()) - N, start\_index + N)  # Go forward but don't exceed the dataset length

        st.session\_state['start\_index'] = start\_index

# End index for slicing

end\_index = start\_index + N

Data Visualization Loop

while True:

    # Get data

    df = get\_data()

First, we need to retrieve the data

# Dataframe for plotting

    prev\_data = df.iloc[start\_index:end\_index-1]

    new\_data = df.iloc[start\_index:end\_index]

    # Ensure previous and new data have the same length

    min\_length = min(len(prev\_data), len(new\_data))

    prev\_data = prev\_data.tail(min\_length).reset\_index(drop=True)

    new\_data = new\_data.head(min\_length).reset\_index(drop=True)

Then, we have to prepare the data. `prev\_data` represents the current data (before update). `new\_data` is the newly retrieved data from the csv. `min\_length` ensures that both `prev\_data` and `new\_data` have the same number of rows. Then we reset the index to drop the old index.

# Prepare data for interpolation

    current = {

        'x': prev\_data['x'].values,

        'y': prev\_data['y'].values,

        'z': prev\_data['z'].values,

    }

    new = {

        'x': new\_data['x'].values,

        'y': new\_data['y'].values,

        'z': new\_data['z'].values,

    }

    # Interpolate data

    interpolated\_steps = interpolate\_data(current, new, steps=10)

Next step is data interpolation where there are 2 dictionaries for `current` and `new` data arrays for each axis. We use these 2 dictionaries to interpolate.

    with place\_holder.container():

        # Update chart with interpolated data

        col\_chart, col\_data = st.columns(2)

        if axes\_options:

            with col\_chart:

                plot\_placeholder = st.empty()

                for interpolated\_data in interpolated\_steps:

                    interpolated\_df = pd.DataFrame({

                        'Timestamp': prev\_data['Timestamp'],

                        \*\*{axis: interpolated\_data[axis] for axis in axes\_options}

                    })

                    if chart\_type == "Line Chart":

                        fig = px.line(interpolated\_df, x='Timestamp', y=axes\_options, title="Accelerometer Data - Line Chart")

                    elif chart\_type == "Scatter Plot":

                        fig = px.scatter(interpolated\_df, x='Timestamp', y=axes\_options, title="Accelerometer Data - Scatter Plot")

                    elif chart\_type == "Distribution Plot":

                        fig = px.histogram(interpolated\_df.melt(id\_vars="Timestamp", value\_vars=axes\_options),

                                        x='value', color='variable', barmode='overlay',

                                        title="Accelerometer Data - Distribution Plot")

                    plot\_placeholder.write(fig)

                    time.sleep(0.1) # small delay between frames

        else:

            st.write("Please select at least one axis to display the chart.")

        # Update dataframe

        with col\_data:

            st.markdown("Latest data based on selected data for rows {} to {}".format(start\_index, end\_index))

            table\_data = new\_data[['Timestamp'] + axes\_options]

            st.dataframe(table\_data.tail(10))

    # Update `prev\_data` for next iteration

    st.session\_state['prev\_data'] = new\_data

    # Delay before next update

    time.sleep(2)

This is where the chart and dataframe being updated. `place\_holder.container()` is a container for the chart and table. They are being displayed side by side. `axes\_options` check is to ensure at least one axis is being selected, or else no data will be displayed. `interpolated\_df` is the interpolated data used for plotting the charts. The table is being updated with the axis options. One interesting thing about streamlit is that I can use functions from both plotly or bokeh to plot the plots. As seen in the code above I used plotly to plot the line chart, scatterplot and distribution plot. To plot the distribution plot, I melt to unpivot the data frame to plot the histogram.

Contrasting Plotly with Bokeh and Streamlit.

Based on my experience, Streamlit has the most aesthetic front-end and is the easiest to use. It also supports graphs from bokeh and plotly. One downside is that it is quite hard to find a way for a smooth transition, I didn’t manage to be able to do that. Bokeh support smooth transition, but the front-end looks not very nice and the documentation is quite hard to understand. Plotly has decent documentation, but the code is a bit longer than Streamlit. If I had to choose, I would go with Streamlit for it ease of use. Streamlit and Bokeh have the display cap at whatever the default display number is, we got to set it to lower but not higher.

**Q4.**

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

**Q5.**