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.

https://github.com/tomadonna1/SIT225_2024T2/tree/main/HD%20Task%20Data%20dashboard%20alternative%20to%20Plotly