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. It works by accelerate at the beginning (ease-in), and the decelerate at the end (ease-out). Interpolation calculates the intermediate values between two known values. By calculating the intermedia points, it provides a smoother curve between points. Say we have data point moving from 10 to 30, cubic easing when interpolation generates a series of intermediate values (12, 18, 25) based on our cubic easing function. The points are gradually adjusting, making the updated data feel animated. I managed to apply this to Bokeh, but Streamlit doesn't seem to support it.

Q3.

Bokeh framework.

```
class AccelerometerDataApp:
    def __init__(self, data_path: str):
        """Initiate the application with a csv file path

        Args:
            data_path (str): path to csv for the accelerometer data
        """
        self.data_path = data_path
        self.N = 100  # Initial number of samples to display
        self.df = self.get_data()
        self.current_start = max(0, len(self.df) - self.N)
        self.last_navigated = False  # Track if user navigated

        self.axis_options = ['x', 'y', 'z']
        self.axis_colors = {'x': 'blue', 'y': 'green', 'z': 'red'}
        self.selected_axes = ['x', 'y', 'z']  # Default axes selected

        # Initialize ColumnDataSource
        self.source = ColumnDataSource(
            self.df.iloc[self.current_start:self.current_start +
self.N][['Timestamp'] + self.selected_axes]
        )
```

```
        # Initialize DataTable
        self.columns = self.get_table_columns(self.selected_axes)
        self.data_table = DataTable(source=self.source, columns=self.columns,
width=700, height=280)

        # Initialize Figure
        self.p = figure(x_axis_type="datetime", title="Accelerometer Data",
height=400, width=700)

        # Initialize Widgets
        self.graph_type = Select(
            title="Select graph type",
            value="Line",
            options=["Line", "Scatter", "Distribution"]
        )
        self.sample_input = TextInput(title="Number of samples to display",
value=str(self.N))
        self.prev_button = Button(label="Previous", button_type="success")
        self.next_button = Button(label="Next", button_type="success")
        self.axis_selection = CheckboxGroup(labels=self.axis_options, active=[0,
1, 2])

        # Create Layout
        self.layout = self.create_layout()

        # Attach Callbacks
        self.graph_type.on_change('value', self.on_graph_type_change)
        self.sample_input.on_change('value', self.on_sample_input_change)
        self.prev_button.on_click(self.on_prev_button_click)
        self.next_button.on_click(self.on_next_button_click)
        self.axis_selection.on_change('active', self.on_axis_selection_change)

        # Initial Graph Rendering
        self.update_graph_type()

        # Add Layout to Document
        curdoc().add_root(self.layout)

        # Periodic Callback to update data every 2 seconds (update continuously
for 2s)
        curdoc().add_periodic_callback(self.periodic_update, 2000)
```

This is the constructor of the app by setting up the layout, loading the data, attaching the callbacks for interactivity.

```
    def get_data(self) -> pd.DataFrame:
```

```
        """Read data from csv

        Returns:
            pd.DataFrame: dataframe containing accelerometer data
        """
        data = pd.read_csv(
            self.data_path,
            header=None,
            names=['Timestamp', 'x', 'y', 'z'],
            parse_dates=['Timestamp']
        )
        return data
```

This function reads data from the csv file and return the dataframe with titles and correct data type.

```
    def get_table_columns(self, selected_axes) -> List[TableColumn]:
        """Generate table columns for the DataTable based on the selected axes

        Args:
            selected_axes (_type_): list of current selected axes

        Returns:
            List[TableColumn]: list of TableColumn objects for the DataTable
        """
        columns = [TableColumn(field="Timestamp", title="Timestamp")]
        for axis in selected_axes:
            columns.append(TableColumn(field=axis, title=axis.upper()))
        return columns
```

This function generates a list of Bokeh TableColumn objects based on the selected axes. Each axis has its own column, and the Timestamp is included by default.

```
    def cubic_ease_out(self, t: float, b: Union[float, np.array], c: Union[float,
np.array], d: float) -> Union[float, np.array]:
        """Cubic easing for smooth transactions between data updates

        Args:
            t (float): current time step
            b (Union[float, np.array]): starting value
            c (Union[float, np.array]): change in value
            d (float): 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
```

Perform cubic easing interpolation for smooth transaction between data updates.

```
    def interpolate_data(self, 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 via cubic easing

        Args:
            current (Dict[str, np.ndarray]): current data (x,y,z)
            new (Dict[str, np.ndarray]): new data (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 = []
        for i in range(steps):
            progress = i / steps
            interpolated_step = {}
            for axis in current.keys():
                interpolated_step[axis] = self.cubic_ease_out(
                    progress,
                    current[axis],
                    new[axis] - current[axis],
                    1
                )
            interpolated.append(interpolated_step)
        return interpolated
```

This method interpolates data points between the current and new data arrays via cubing easing function.

```
    def create_layout(self):
        """Create layout for the app with widgets, plots, and tables

        Returns:
            _type_: the layout with its components
        """
        return column(
            row(self.graph_type, self.sample_input),
            self.axis_selection,
            self.p,
            row(self.prev_button, self.next_button),
            self.data_table
        )
```

This function creates the Bokeh layout with the widgets (Select, TextInput, Buttons), the plot for visualization, DataTable for displaying data.

```
def update_graph_type(self):
        """Update plot based on selected graph type (line, scatter, distribution)
        """
        self.p.renderers = []  # Clear existing glyphs

        if self.graph_type.value == "Line":
            for axis in self.selected_axes:
                self.p.line(
                    x='Timestamp',
                    y=axis,
                    source=self.source,
                    line_width=2,
                    color=self.axis_colors[axis],
                    legend_label=f"{axis.upper()}-axis"
                )
        elif self.graph_type.value == "Scatter":
            for axis in self.selected_axes:
                self.p.scatter(
                    x='Timestamp',
                    y=axis,
                    source=self.source,
                    size=7,
                    color=self.axis_colors[axis],
                    legend_label=f"{axis.upper()}-axis"
                )
        elif self.graph_type.value == "Distribution":
            for axis in self.selected_axes:
                hist, edges = np.histogram(
                    self.df[axis].iloc[self.current_start:self.current_start +
self.N],
                    bins=30
                )
                self.p.quad(
                    top=hist,
                    bottom=0,
                    left=edges[:-1],
                    right=edges[1:],
                    fill_color=self.axis_colors[axis],
                    line_color=self.axis_colors[axis],
                    alpha=0.5,
                    legend_label=f"{axis.upper()}-axis"
                )
```

```
        self.p.legend.location = 'bottom_left'
        self.p.legend.click_policy = "hide"
```

This method updates the plot based on the selected graph (default is line plot). When a graph is selected, it clears the data from the previous graph and plot the select graph.

```
    def update_data(self):
        """Update the data source and apply interpolation when data is updated
        """
        self.df = self.get_data()

        if not self.last_navigated:
            self.current_start = max(0, len(self.df) - self.N)

        new_data_slice = self.df.iloc[self.current_start:self.current_start +
self.N]
        timestamps = new_data_slice['Timestamp'].values

        if not self.selected_axes:
            # If no axes are selected, update only the Timestamp
            self.source.data = {'Timestamp': timestamps}
            self.data_table.columns = self.get_table_columns(self.selected_axes)
            self.update_graph_type()
            self.last_navigated = False  # Reset navigation flag
            return

        # Get the current and new data slices
        current_data = {axis: np.array(self.source.data[axis]) for axis in
self.selected_axes}
        new_data = {axis: np.array(new_data_slice[axis]) for axis in
self.selected_axes}

        # Determine the minimum length
        min_length = min(len(current_data[self.selected_axes[0]]),
len(new_data[self.selected_axes[0]]))

        # Trim data to the minimum length
        for axis in self.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 = self.interpolate_data(current_data, new_data,
steps=10)
```

```
        # Apply the interpolation over time
        def apply_interpolation(step=0):
            if step < len(interpolated_steps):
                data = {'Timestamp': timestamps}
                for axis in self.selected_axes:
                    data[axis] = interpolated_steps[step][axis]
                self.source.data = data
                curdoc().add_timeout_callback(lambda: apply_interpolation(step +
1), 100)  # 100ms delay
            else:
                # Ensure the final data is set correctly
                data = {'Timestamp': timestamps}
                for axis in self.selected_axes:
                    data[axis] = new_data[axis]
                self.source.data = data

        apply_interpolation()

        # Update the table columns
        self.data_table.columns = self.get_table_columns(self.selected_axes)

        # Update the plot
        self.update_graph_type()

        self.last_navigated = False  # Reset navigation flag
```

This function refreshes the data by reading the latest data from the csv file. It then applies the interpolation for a smooth transition between old and new data. The method does this by updating the Bokeh `ColumnDataSource` to reflect the changes in data and rendered it on the graph and the DataTable.

```
    def on_prev_button_click(self):
        """Callback for when previous button is clicked
        """
        self.last_navigated = True  # Mark navigation
        self.current_start = max(0, self.current_start - self.N)
        self.update_data()

    def on_next_button_click(self):
        """Callback for when next button is clicked
        """
        self.last_navigated = True  # Mark navigation
        self.current_start = min(len(self.df) - self.N, self.current_start +
self.N)
        self.update_data()
```

Call back methods for previous and next button when clicked, which allow the user to navigate to the next or previous set of samples.

```
    def on_axis_selection_change(self, attr, old, new):
        """Callback when the axes selection changes

        Args:
            attr (_type_): the attribute that changed
            old (_type_): the old value of the attribute
            new (_type_): the new value of the attribute
        """
        self.selected_axes = [self.axis_options[i] for i in
self.axis_selection.active]
        if self.selected_axes:
            self.source.data = self.df.iloc[self.current_start:self.current_start
+ self.N][['Timestamp'] + self.selected_axes].to_dict('list')
        else:
            self.source.data = {'Timestamp':
self.df.iloc[self.current_start:self.current_start + self.N]['Timestamp'].values}
        self.data_table.columns = self.get_table_columns(self.selected_axes)
        self.update_graph_type()
        self.last_navigated = True  # Indicate user interaction
        self.update_data()
```

Callback method that triggers when the user selects or deselects the axes for plotting. This then be updated on the graph and the table.

```
    def periodic_update(self):
        """Periodic callback to update data every 2 seconds. Continuous data
update
        """
        if not self.last_navigated:
            self.update_data()
```

Continuous update the data for every 2 seconds.

```
app = AccelerometerDataApp(r'C:\Users\tomde\OneDrive\Documents\Deakin uni\Deakin-
Data-Science\T1Y2\SIT225 - Data Capture Technologies\Week 6 - Visualisation -
Plotly data dashboard\6.2HD\data_gathering\data.csv')
```

Call the class, which does everything similar to that of an API.

Streamlit framework

```python
class AccelerometerAPI:
    def __init__(self, csv_path: str, update_interval: int = 5):
        """Initialize the API with the CSV file path and update interval"""
        self.csv_path = csv_path
        self.update_interval = update_interval
        self.N = 200 # Default number of samples to display
        self.data = pd.DataFrame()  # Empty DataFrame to start
```

Initialize the class API. The constructor consists of instances for csv file path and update interval for live data updates.

```python
    def get_data(self) -> pd.DataFrame:
        """Read accelerometer data from CSV file and handle live updates."""
        data = pd.read_csv(self.csv_path, header=None, names=['Timestamp', 'x', 'y', 'z'])
        data['Timestamp'] = pd.to_datetime(data['Timestamp'], format='%Y%m%d%H%M%S')
        return data
```

This method reads the data from the csv file and return it as a dataframe.

```python
    def update_dashboard(self):
        """Function to continuously update the dashboard with live data"""

        # Set up Streamlit page config
        st.set_page_config(
            page_title="Accelerometer Dashboard",
            page_icon="☑",
            layout="wide",
        )

        st.title("Accelerometer Dashboard")

        # Chart and axis selection
        chart_type = st.selectbox("Select chart type", ["Line Chart", "Scatter Plot", "Distribution Plot"])
        axes_options = st.multiselect("Select axis to plot", options=["x", "y", "z"], default=["x", "y", "z"])
        N = st.number_input("Enter number of samples to display", min_value=1, max_value=1000, value=self.N)

        # Placeholder for chart and table
        place_holder = st.empty()

        while True:
            self.data = self.get_data()  # get live update from CSV

            # Determine the subset of data to display
```

```
            start_index = max(0, len(self.data) - N)
            df_subset = self.data.iloc[start_index:]

            # Container that organizes the chart and table
            with place_holder.container():
                # Chart update
                col_chart, col_data = st.columns(2)
                if axes_options: # generate chart based on what user selected
                    with col_chart:
                        if chart_type == "Line Chart":
                            fig = px.line(df_subset, x='Timestamp',
y=axes_options, title="Accelerometer Data - Line Chart")
                        elif chart_type == "Scatter Plot":
                            fig = px.scatter(df_subset, x='Timestamp',
y=axes_options, title="Accelerometer Data - Scatter Plot")
                        elif chart_type == "Distribution Plot":
                            fig =
px.histogram(df_subset.melt(id_vars="Timestamp", value_vars=axes_options),
                                                x='value', color='variable',
barmode='overlay',
                                                title="Accelerometer Data -
Distribution Plot")
                        st.write(fig)

                    # Data table update
                    with col_data:
                        st.markdown(f"Latest data from rows {start_index} to
{len(self.data)}")
                        st.dataframe(df_subset[['Timestamp'] + axes_options])
                else:
                    st.write("Please select at least one axis to display the
chart.")


                time.sleep(self.update_interval)
                st.rerun()  # This forces the Streamlit app to refresh the data
```

This method continuously updates the dashboard with live data from the csv. The dashboard has options for selecting chart types, axes to plot, and the sample of data to display. 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. 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.

```
if __name__ == "__main__":
    api = AccelerometerAPI(csv_path=r'C:\Users\tomde\OneDrive\Documents\Deakin uni\Deakin-Data-Science\T1Y2\SIT225 - Data Capture Technologies\Week 6 - Visualisation - Plotly data dashboard\6.2HD\data_gathering\data.csv')
    api.update_dashboard()
```

We just need to call the class to use it.

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.

I have put the two dashboards set up in an API. Currently it is not generalized for any data, but the developers can use it as a base to display their live data.

Q4.

https://www.youtube.com/watch?v=PGCUwvj6WUs

Q5.

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