WE MechLoad Viewer

Technical Architecture and Data Flow Documentation

Table of Contents

1. Introduction & High-Level Architecture

1.1. Purpose of the Application

The **WE MechLoad Viewer** is a specialized desktop application designed for mechanical engineers to load, visualize, analyze, and export mechanical load data. It handles both time-domain and frequency-domain data, providing various plotting capabilities and export functions for further analysis in software like Ansys.

1.2. Architectural Pattern (MVC-inspired)

The application's architecture is heavily inspired by the **Model-View-Controller (MVC)** design pattern. This pattern separates the application's concerns into three interconnected components, which makes the code more organized, scalable, and easier to maintain.

[a Model-View-Controller diagram resmi]

- **Model (DataManager)**: Manages the application's data and business logic. It's responsible for reading data from files, processing it, and storing it. It knows nothing about the user interface.
- **View (UI Tabs like SingleDataTab)**: The user interface. It displays the data from the Model and sends user actions (button clicks, selections) to the Controller.
- **Controller (PlotController, ActionHandler)**: Acts as the intermediary between the Model and the View. It receives user input from the View, processes it (sometimes by querying the Model for data), and updates the View with the results.

This separation is key. For example, the DataManager can be tested independently of the UI, and the UI can be changed without affecting the data loading logic.

1.3. Key Components Overview

Here's a tree view of the main components and their roles:

```
WE MechLoad Viewer
├── main.py (Entry Point)
│
├── app/
│   ├── data_manager.py (Model)
│   │   └── Handles all file I/O and data parsing. Holds the raw data logic.
│   │
│   ├── main_window.py (Main Container)
│   │   └── Holds all UI elements and acts as the central communication hub.
│   │
│   ├── controllers/
│   │   ├── plot_controller.py (Controller)
```

```
│   │   │   └── Manages all logic related to updating plots.
│   │   └── action_handler.py (Controller)
│   │       └── Manages logic for exports and other complex actions.
│   │
│   ├── ui/ (Views)
│   │   ├── tab_single_data.py
│   │   ├── tab_interface_data.py
│   │   └── ... (Other UI tabs and components)
│   │
│   ├── plotting/
│   │   └── plotter.py (Utility)
│   │       └── Contains the actual plotting code (using Plotly).
│   │
│   └── analysis/
│       ├── data_processing.py (Utility)
│       │   └── Contains functions for filtering, windowing, etc.
│       └── ansys_exporter.py (Utility)
│           └── Contains logic to generate Ansys script files.
```

2. Core Components: The Application Backbone

2.1. main.py: The Application Entry Point

This is the script that starts everything. Its role is simple but critical:

1. **Create the QApplication instance**: The fundamental object for any PyQt app.
2. **Instantiate Core Objects**: It creates the two most important objects:
   - data_manager = DataManager()
   - main_window = MainWindow(data_manager)
3. **Establish Initial Connection**: It makes the very first signal-slot connection, which is the foundation for the entire application's data flow: data_manager.dataLoaded.connect(main_window.on_data_loaded)

4. **Show the Window**: It displays the MainWindow.
5. **Trigger Initial Data Load**: It uses QTimer.singleShot to call data_manager.load_data_from_directory(). Using a timer ensures the UI is fully rendered and responsive before the potentially blocking file dialog appears.

2.2. DataManager: The Data Engine (Model)

The DataManager is the sole authority on data loading and parsing. It is completely decoupled from the UI.

Data Loading Process (load_data_from_paths)

This is the core method. When called, it performs the following steps for each folder path it receives:

1. **Validation**: It checks for the existence of required full.pld and max.pld files. If they're missing, the folder is skipped.
2. **Read Raw Data**: It reads all full.pld files into a temporary Pandas DataFrame.
3. **Determine Domain**: It inspects the DataFrame's columns to see if it contains 'TIME' or 'FREQ', thus determining the data's domain.
4. **Domain Consistency Check**: It ensures that all folders being loaded share the same domain. If a mismatch is found, the inconsistent folder is skipped.
5. **Read Headers**: It calls _read_pld_log_file to parse the max.pld file, which contains the column headers.
6. **Assign Columns**: It uses _get_column_headers to format the headers correctly (adding 'Phase_' columns for frequency data) and assigns them to the temporary DataFrame.
7. **Tag Data**: It adds a 'DataFolder' column to the DataFrame to identify the source of the data. This is crucial for multi-folder analysis.
8. **Combine and Sort**: After processing all valid folders, it concatenates all temporary DataFrames into one final DataFrame and sorts it by the domain column ('TIME' or 'FREQ').

Emitted Signals

The DataManager communicates with the rest of the application using these signals:

- dataLoaded(pd.DataFrame, str, str): Emitted upon a successful data load. It sends the final DataFrame, the data domain ('TIME' or 'FREQ'), and the path of the first valid folder.
- dataLoadFailed(str): Emitted if no valid data could be loaded.
- comparisonDataLoaded(pd.DataFrame): Emitted when a secondary dataset for comparison has been successfully loaded.

2.3. MainWindow: The Central Hub (View-Controller)

MainWindow is the most central class. It owns all the UI elements, the controllers, and holds the application's state.

Responsibilities:

1. **UI Setup (_setup_ui)**: Initializes and arranges all the main UI components: the menu bar, the DirectoryTreeDock, and the main QTabWidget which contains all the individual view tabs.
2. **Signal Aggregation (_connect_signals)**: This is where the application is wired together. It connects signals from the UI tabs and DataManager to the slots in the controllers (PlotController, ActionHandler) and MainWindow itself. This method is the blueprint for the application's interactivity.
3. **State Management**: It holds the primary data in self.df and comparison data in

self.df_compare. By holding the state, it ensures that all controllers and views have a single source of truth for the data.

Key Slots:

- **on_data_loaded(self, data, data_domain, folder_path)**: This is arguably the most important slot in the application. When the DataManager emits its dataLoaded signal, this method is triggered. It does the following:
  1. Receives the data and stores it in self.df.
  2. Updates the window title.
  3. Populates the DirectoryTreeDock.
  4. Calls _populate_all_selectors() to fill all the dropdowns in the UI tabs with the columns and interfaces from the new data.
  5. Adjusts UI visibility based on the data (e.g., shows/hides the "Time Domain Rep." tab).
  6. Triggers an initial plot update by calling self.plot_controller.update_all_plots_from_settings().
- **on_comparison_data_loaded(self, df_compare)**: Stores the comparison DataFrame in self.df_compare and populates the relevant UI controls in the CompareDataTab.

3. The Controller Layer: Handling Logic

3.1. Role of Controllers

The controllers listen for signals from the View (UI tabs) and decide what to do. They contain the "business logic" of the application. This design prevents the UI tabs from becoming bloated with complex logic.

3.2. PlotController: The Visualization Coordinator

The PlotController's only job is to create plots. It listens for ..._parameters_changed signals from the various UI tabs.

Workflow for a Typical Slot (e.g., update_single_data_plots):

1. **Trigger**: The user changes a setting in SingleDataTab (e.g., selects a new column). SingleDataTab emits plot_parameters_changed.
2. **Activation**: update_single_data_plots slot is called.
3. **Gather Information**:
   - It accesses the main DataFrame: self.main_window.df.
   - It reads the current UI settings from SingleDataTab (e.g., self.main_window.tab_single_data.column_selector.currentText()).
   - It reads global plot settings from SettingsTab.
4. **Delegate to Plotter**: It passes the data and settings to a method in the Plotter class (e.g., self.main_window.plotter.create_single_plot(...)).
5. **Receive Figure**: The Plotter returns a Plotly figure object.

6. **Update View**: It calls the appropriate display method on the SingleDataTab to render the figure: self.main_window.tab_single_data.display_regular_plot(fig).

This cycle—**View Event -> Controller Logic -> Model Data -> View Update**—is repeated for every plot in the application.

3.3. ActionHandler: The Action & Export Manager

The ActionHandler is similar to the PlotController but handles non-plotting actions.

Key Slots:

- **handle_compare_data_selection()**: This is a simple passthrough. It receives the request from CompareDataTab and calls self.data_manager.load_comparison_data(), effectively initiating the data loading flow for the comparison dataset.
- **handle_time_domain_represent_export()**:
  1. Triggered by the "Extract Data" button in TimeDomainRepresentTab.
  2. Reads the plot data currently stored on the tab (tab.current_plot_data).
  3. Formats this data into a new Pandas DataFrame.
  4. Opens a QFileDialog to ask the user where to save the CSV file.
  5. Saves the DataFrame to the selected location.
- **handle_ansys_export()**: This is the most complex action.
  1. Triggered by the "Export to Ansys" button in PartLoadsTab.
  2. Opens a dialog to allow the user to select which "sides" to export.
  3. **Filters Data**: Creates a new DataFrame containing only the columns relevant to the selected sides.
  4. **Processes Data**: It accesses the UI settings on PartLoadsTab to see if time-slicing (apply_data_section) or a Tukey window (apply_tukey_window) should be applied. It calls these functions from data_processing.py.
  5. **Unit Conversion**: It multiplies the relevant load columns by 1000.
  6. **Delegate to Exporter**: It creates an AnsysExporter instance and calls the appropriate method (create_harmonic_template or create_transient_template) to generate the final Ansys script files.

4. The View Layer: User Interaction Tabs

Each tab is a self-contained QWidget responsible for its own layout and user inputs. Their primary role is to emit signals when the user interacts with them.

4.1. SingleDataTab

- **Purpose**: To visualize a single data column against TIME or FREQ.
- **Key Widgets**: column_selector, spectrum_checkbox, filter_checkbox.
- **Emitted Signals**:
  - plot_parameters_changed: Emitted when any control that affects the main plot or the existence of the spectrum plot is changed.

    - spectrum_parameters_changed: Emitted only when controls that *only* affect the spectrum plot's appearance (e.g., plot_type_selector) are changed. This is an optimization to avoid re-calculating the main plot unnecessarily.

4.2. InterfaceDataTab

- **Purpose**: To visualize all components (e.g., Fx, Fy, Fz) of a specific interface.
- **Key Widgets**: interface_selector.
- **Emitted Signals**: plot_parameters_changed.

4.3. PartLoadsTab

- **Purpose**: To visualize loads grouped by "side" (e.g., all forces on the "Left-Hand-Side").
- **Key Widgets**: side_filter_selector, component_selector, tukey_checkbox.
- **Emitted Signals**: plot_parameters_changed, export_to_ansys_requested.

4.4. CompareDataTab

- **Purpose**: To plot a column from the primary dataset against the same column from a secondary (comparison) dataset.
- **Key Widgets**: column_selector, compare_column_selector.
- **Emitted Signals**: plot_parameters_changed, select_compare_data_requested.

4.5. TimeDomainRepresentTab

- **Purpose**: For frequency-domain data, this tab reconstructs and visualizes the time-domain representation at a single selected frequency.
- **Key Widgets**: data_point_selector (to select the frequency).
- **Emitted Signals**: plot_parameters_changed, extract_data_requested.

4.6. SettingsTab

- **Purpose**: To control global plotting settings that affect all tabs.
- **Key Widgets**: Checkboxes for legend visibility, grid lines, etc.
- **Emitted Signals**: settings_changed. This signal is connected to the PlotController's update_all_plots_from_settings slot, causing a full refresh of every visible plot.

5. Signal & Slot Communication Maps

This section provides a visual trace of the most common user workflows.

5.1. Application Startup and Initial Data Load

```
graph TD
    A[main.py] -->|1. Creates| B(DataManager);
    A -->|2. Creates| C(MainWindow);
    A -->|3. Connects| D{dataLoaded Signal};
    C -->|4. Listens on| E[on_data_loaded Slot];
```

```
    D -.-> E;
    A -->|5. QTimer calls| F[load_data_from_directory];
    B -- owns --> F;
    F -->|6. User selects folder| G[File Dialog];
    F -->|7. Parses .pld/.log files| H[pandas DataFrame];
    F -->|8. Emits| D;
```

5.2. Plotting Workflow: SingleDataTab

```
USER ACTION: Selects 'I1 Fx' from column_selector in SingleDataTab
    |
    |
    ▼
SingleDataTab.column_selector.currentIndexChanged
    |
    ├── emits signal: plot_parameters_changed()
    |
    |
    ▼
MainWindow._connect_signals (Connection was made here)
    |
    |
    ▼
PlotController.update_single_data_plots() [SLOT]
    |
    ├── 1. Accesses MainWindow.df (The main data)
    ├── 2. Reads current column 'I1 Fx' from SingleDataTab
    ├── 3. Reads plot settings from SettingsTab
    |
    ├── 4. Calls Plotter.create_single_plot(data, column='I1 Fx', ...)
    |     |
    |     └── Returns a Plotly figure object `fig`
    |
    ├── 5. Calls SingleDataTab.display_regular_plot(fig)
    |     |
    |     └── Renders `fig` into the QWebEngineView
    |
    └── [PROCESS COMPLETE]
```

5.3. Plotting Workflow: InterfaceDataTab

This follows the same pattern as SingleDataTab, but is triggered by the interface_selector.

```
USER ACTION: Selects 'I3' from interface_selector in InterfaceDataTab
    |
    ▼
InterfaceDataTab.interface_selector.currentIndexChanged
    |
    ├── emits signal: plot_parameters_changed()
    |
    ▼
PlotController.update_interface_data_plots() [SLOT]
    |
    ├── 1. Accesses MainWindow.df
    ├── 2. Reads current interface 'I3'
    ├── 3. Finds all columns related to 'I3' (e.g., 'I3 Fx', 'I3 Fy', ...)
    ├── 4. Calls Plotter.create_interface_plot(data, interface='I3', ...)
    |
    └── Calls InterfaceDataTab.display_plot(fig)
```

5.4. Plotting Workflow: PartLoadsTab

This workflow is slightly more complex as it involves filtering by both "side" and "component".

```
USER ACTION: Selects 'LHS' from side_filter_selector
    |
    ▼
PartLoadsTab.side_filter_selector.currentIndexChanged
    |
    ├── emits signal: plot_parameters_changed()
    |
    ▼
PlotController.update_part_loads_plots() [SLOT]
    |
    ├── 1. Accesses MainWindow.df
    ├── 2. Reads side 'LHS' and component 'Mz' from PartLoadsTab
    ├── 3. Filters df to get all 'Mz' columns that also contain 'LHS'
    ├── 4. Calls Plotter.create_part_loads_plot(...)
    |
    └── Calls PartLoadsTab.display_plot(fig)
```

5.5. Data Comparison Workflow

This is a two-step process: loading the data, then plotting.

Step 1: Loading Comparison Data

```
USER ACTION: Clicks "Select Data to Compare" button in CompareDataTab
    |
    ▼
CompareDataTab.select_compare_data_requested() [SIGNAL]
    |
    ▼
ActionHandler.handle_compare_data_selection() [SLOT]
    |
    └─ Calls DataManager.load_comparison_data()
        |
        ├─ Opens File Dialog, parses files...
        |
        └─ emits comparisonDataLoaded(df_compare) [SIGNAL]
            |
            ▼
        MainWindow.on_comparison_data_loaded(df_compare) [SLOT]
            |
            ├─ Stores data in self.df_compare
            └─ Populates selectors in CompareDataTab
```

Step 2: Plotting Comparison Data
This follows the standard plotting flow, but the PlotController's update_compare_data_plots slot will access both self.main_window.df and self.main_window.df_compare.

5.6. Action Workflow: Ansys Export

```
USER ACTION: Clicks "Export to Ansys" button in PartLoadsTab
    |
    ▼
PartLoadsTab.export_to_ansys_requested() [SIGNAL]
    |
    ▼
ActionHandler.handle_ansys_export() [SLOT]
    |
    ├─ 1. Opens dialog to select "sides" for export.
    ├─ 2. Filters MainWindow.df based on selection.
    ├─ 3. Checks UI controls on PartLoadsTab (e.g., tukey_checkbox).
    |
    ├─ 4. Calls functions from data_processing.py (e.g., apply_tukey_window).
    |    |
```

```
    │    └── Returns a processed DataFrame `df_processed`.
    │
    ├── 5. Performs unit conversions on `df_processed`.
    │
    ├── 6. Creates AnsysExporter instance.
    │
    ├── 7. Calls AnsysExporter.create_transient_template(df_processed, ...).
    │    │
    │    └── Writes .dat and .inp files to disk.
    │
    └── [PROCESS COMPLETE]
```

5.7. Action Workflow: Time Domain Data Extraction

```
USER ACTION: Clicks "Extract Data" button in TimeDomainRepresentTab
    │
    ▼
TimeDomainRepresentTab.extract_data_requested() [SIGNAL]
    │
    ▼
ActionHandler.handle_time_domain_represent_export() [SLOT]
    │
    ├── 1. Reads plot data from TimeDomainRepresentTab.current_plot_data.
    ├── 2. Constructs a new DataFrame from this plot data.
    ├── 3. Opens a QFileDialog to get a save path from the user.
    │
    └── 4. Saves the new DataFrame as a .csv file.
```

6. Backend Modules: Analysis & Utilities

These modules contain pure, non-GUI logic.

6.1. data_processing.py

This is a library of functions that operate on DataFrames.

- apply_data_section: Slices a DataFrame based on a min/max time.
- apply_tukey_window: Applies a Tukey (tapered cosine) window to the data to reduce spectral leakage, which is important for FFT analysis.
- apply_low_pass_filter: Applies a Butterworth filter to remove high-frequency noise from time-domain data.

6.2. ansys_exporter.py

This class is responsible for converting a Pandas DataFrame into a format that Ansys Mechanical can understand.

- It creates .dat files which are essentially tables of load-vs-time or load-vs-frequency.
- It also creates a corresponding .inp (APDL script) file that tells Ansys how to read the .dat file and apply it as a load in a transient or harmonic analysis.

6.3. plotter.py

This class isolates the plotting library (Plotly) from the rest of the application. The PlotController tells the Plotter *what* to plot, and the Plotter handles the *how*. This design means you could swap out Plotly for Matplotlib or another library by only changing plotter.py, without touching any of the controller logic.

7. State Management

7.1. Primary Data State (self.df)

The primary loaded data is stored in MainWindow.df. This is the **single source of truth**. All plotting and analysis operations read from this DataFrame. It is considered immutable after loading; any processing (like filtering) creates a temporary copy, rather than modifying the original df.

7.2. Comparison Data State (self.df_compare)

Similarly, MainWindow.df_compare holds the secondary dataset. It's kept separate to avoid confusion with the primary data.

7.3. UI State

The state of all checkboxes, selectors, and input fields across all UI tabs is the "UI State". The controllers read this state at the moment a plot or action is requested to get the user's desired parameters.

8. Conclusion and Future Development

The WE MechLoad Viewer employs a robust, decoupled architecture based on MVC principles and the Qt signal/slot mechanism. This design promotes:

- **Maintainability**: Code is organized by function, making it easy to locate and modify logic.
- **Scalability**: New tabs, plots, or export formats can be added by creating new view/controller pairs without disrupting existing functionality.
- **Testability**: The data and controller layers can be tested independently of the GUI.

Potential Future Enhancements:

- **Unit Testing**: Implement unit tests for DataManager, PlotController, ActionHandler, and the analysis modules.
- **Asynchronous Data Loading**: For very large datasets, the data loading process could be moved to a separate QThread to prevent the GUI from freezing.
- **User-Defined Calculations**: Add a feature allowing users to create new data columns based on mathematical operations on existing columns (e.g., New_Force = I1_Fx * 2.5).
- **Plugin System**: Develop a plugin system that would allow new plot types or data exporters to be added as separate modules.