Contents

This page intentionally left blank.

1 Introduction

This software suite is designed to process, analyze, and visualize 3D measurement data obtained from surface scanning of physical samples, specifically T-shaped metal specimens. The toolkit addresses common challenges in metrology data processing, including noise reduction, outlier detection, surface fitting, and 3D reconstruction.

1.1 Purpose

Raw measurement data from coordinate measuring machines (CMMs) or optical scanners typically contains:

- Measurement noise and systematic errors
- Thousands of unstructured data points
- Inconsistencies between multiple measurements

This toolset provides a systematic workflow to:

1. Clean and organize raw measurement data
2. Fit mathematical surface models to point clouds
3. Generate statistical analysis and visualizations
4. Reconstruct regularized surface models for further analysis or simulation

Part I

Data Preparation & Preprocessing

2

Module A: Surface Analysis (T-Shape)

2.1 Stage 1: Data Cleaning and Organization

Script: `s1_Outline_process.py`

This stage addresses the initial ingestion of raw measurement data. Raw data typically arrives in fragmented files (separated by "bottom" and "walls") and contains systematic noise. This script consolidates these files, performs statistical averaging, and executes an automated cleaning routine.

2.1.1 How it Works

The script executes a pipeline consisting of four sequential operations:

1. **File Combination:** The script merges corresponding `_bottom` and `_wall` text files into a single entity. It inserts metadata markers (`#bottom` and `#wall`) to preserve the geometric distinction between regions.
2. **Measurement Averaging:** To minimize random noise, multiple measurements of the same sample are averaged.
    - *Note on Shapes:* If measurements differ in point count (e.g., one run has 1000 points and another 1005), the script automatically trims the data to the minimum row count found to ensure mathematical compatibility.
3. **Step Detection:** The T-shaped samples contain physical "steps". The script segments the data by analyzing the X-coordinates. A new step is registered if the X-value changes by more than **0.6%** relative to the previous point.
4. **Automated Outlier Removal (IQR):** The script cleans noise using the Interquartile Range (IQR) method. It uses a rigorous dual-projection logic: a point is kept only if it is valid in **both** the XZ projection AND the YZ projection.

2.1.2 Usage

To run the automated cleaning process, execute the script from the terminal within the project directory:

```
python s1_Outline_process.py
```

The script processes all files in the `Sample_og` folder and outputs the results to `Sample_postprocess` in JSON format.

2.1.3 Configuration and Modifiable Parameters

The script is tuned for standard T-shape measurements, but parameters can be adjusted in the code for different geometries or noise levels.

- **Cleaning Iterations:** Controlled by the `iterations` variable in the `main()` function. The default is **3 passes**. Increasing this makes the cleaning more aggressive.
- **Outlier Sensitivity (IQR Factors):** The strictness of the cleaning is defined in the `section_params` dictionary.
  - **Bottom/Wall:** Default factor is **1.2**. Lower values (e.g., 1.0) remove more points; higher values (e.g., 2.0) are more permissive.
  - **Default:** Default factor is **1.5** for unspecified sections.

  Note on Adaptability: This script is hardcoded for the file naming convention `SideX_MeasurmentY`. If the measurement strategy changes (as seen in `src/Preprocess/exp2`), the Regex patterns in `combine_bottom_wall_files` must be updated to match the new filename structure.

2.2 Stage 2: Visual Inspection and Manual Cleaning

Script: `s2_Outline_gui.py`

While Stage 1 removes statistical outliers, it cannot account for all geometric anomalies. Stage 2 provides a Graphical User Interface (GUI) for the visual inspection and manual refinement of the point cloud data.

2.2.1 How it Works

The tool is built using the `Tkinter` framework embedded with `Matplotlib` for visualization. It operates on the following principles:

1. **Smart Data Loading:** The script scans the `Sample_postprocess` directory. It uses a regular expression pattern (`Side*_Cleaned_Iter*_IQR*`) to identify files and automatically loads only the **latest iteration** (highest "Iter" number) for each measurement side.

2. **2D Projection (YZ Plane):** The visualization focuses exclusively on the YZ plane (Side View), as this is the critical cross-section for determining the profile quality of the T-shape samples.
3. **Interactive Deletion:** Users can remove erroneous points by clicking on the plot. The system calculates the Euclidean distance between the cursor click and the nearest data point:

$$d = \sqrt{(y_{point} - y_{click})^2 + (z_{point} - z_{click})^2} \tag{2.1}$$

   If $d < 0.1$, the point is flagged for removal.
4. **Non-Destructive Saving:** When data is saved, the script does not overwrite the original cleaning files. Instead, it generates a new file suffixed with `_Modified.json` (e.g., `Side1_Modified.json`) to ensure data traceability.

2.2.2 Usage

To launch the interface, execute the script from the terminal:

```
python s2_Outline_gui.py
```

Operational Steps

- **Load Data:** The tool loads files automatically on startup. Use the "Reload Files" button if new data is added during the session.
- **Navigation:** Select the desired **Side** and **Step** from the dropdown menus in the right-hand control panel.
- **Inspection:** Use the "Color by Section" checkbox to visually distinguish between "bottom" and "wall" points.
- **Cleaning:** Click directly on outlier points in the plot area to remove them. A confirmation message will appear indicating the number of points removed.
- **Save:** Click "Save Modified Data" to write the changes to disk.

2.2.3 Configuration and Parameters

Most parameters are accessible via the GUI controls, but certain core behaviors are defined in the code:

- **Deletion Threshold:** The sensitivity of the mouse click is hardcoded to **0.1 units** in the `on_click` method. If points are too dense or too sparse, this value can be adjusted in the script.
- **Directory Paths:** The script is configured to look for a relative directory named `Sample_postprocess`. If the project structure changes, the `self.input_dir` variable in the `__init__` method must be updated.

2.3 Stage 3: Mathematical Surface Fitting

Script: `s3_Plane_process.py`

Once the point cloud data is cleaned, the next step is to generate a continuous mathematical definition of the surface. This module performs regression analysis using two distinct polynomial bases to determine the optimal fit.

2.3.1 How it Works

The script processes every identified measurement side by testing polynomial degrees from 1 up to a configured maximum (default: 10). It utilizes two different mathematical approaches:

1. Standard Polynomial Fit

The standard fit in this implementation models the surface as a sum of independent powers of X and Y.

$$Z(x, y) = \sum_{k=1}^{n} (a_k x^k + b_k y^k) + c \tag{2.2}$$

The regression matrix is built by appending columns for x^k and y^k sequentially. This method is computationally lighter but assumes features are aligned with the primary axes.

2. Chebyshev Polynomial Fit

The Chebyshev method uses orthogonal polynomials of the first kind, $T_n(x)$. Unlike the standard fit, this implementation uses a **tensor product** basis, capturing cross-coupled features (xy) more effectively:

$$Z(x, y) = \sum_{i=0}^{n} \sum_{j=0}^{n} c_{ij} T_i(x_{norm}) T_j(y_{norm}) \tag{2.3}$$

Crucially, this method requires **coordinate normalization**:

- The script maps raw x, y coordinates to the range $[-1, 1]$ via the `normalize_data` function before fitting.
- This prevents numerical instability (Runge's phenomenon) at higher polynomial degrees.

Evaluation Metrics

For every degree and method, the script calculates:

- **MSE (Mean Squared Error)**: The average squared deviation.

- **RMSE (Root Mean Squared Error)**: The primary metric used to rank the quality of the fits.
- **Max Error**: The single largest deviation found on the surface.

2.3.2 Usage

Execute the script to process all cleaning results found in `Sample_postprocess`:

```
python s3_Plane_process.py
```

Process Flow:

1. **File Discovery:** The script uses `find_latest_iteration_files` to ensure only the most recently modified cleaning data (from Stage 2) is used.
2. **Iterative Fitting:** It loops from degree 1 to 10.
3. **Comparison:** It compares Standard vs. Chebyshev RMSE and logs the percentage improvement.
4. **Output:** Results are saved to `Sample_postprocess/Planes_data/` as JSON files containing the coefficients and equation strings.

2.3.3 Configuration

The behavior is controlled by variables within the `main` function:

- **Maximum Degree:**

  ```
  maximum_degree = 10
  ```

  This sets the upper limit for the polynomial complexity. Higher values may reduce RMSE but risk overfitting the surface roughness rather than the geometry.
- **Method Selection:** The `compare_standard_vs_chebyshev` function is hardcoded to run both methods. To save computation time in production, one could modify `main()` to call `experiment_method` with only a single method string ("Standard" or "Chebyshev").

2.4 Stage 4: Comparative Analysis and Averaging

Script: `s4_Choosed_plane.py`

While Stage 3 fits surfaces to individual measurements, Stage 4 performs comparative analysis to isolate specific surface features and generate a representative "master" profile for the sample batch.

2.4.1 How it Works

The script executes a two-step analytical process:

1. Tilt Correction via Surface Subtraction

Measurement data often includes a systematic "tilt" due to physical alignment during scanning. To remove this while preserving surface roughness/curvature:

1. The script loads two fitted surfaces of different degrees (e.g., Degree 5 and Degree 1).
2. It computes the **Difference Surface** (Z_{diff}) by subtracting the lower-degree surface (tilt/form) from the higher-degree surface (roughness/waviness):

$$Z_{diff}(x, y) = Z_{high\_degree}(x, y) - Z_{low\_degree}(x, y) \tag{2.4}$$

3. This operation effectively "flattens" the data relative to its general form.

2. Multi-Side Averaging

To create a robust simulation model, the tool averages the results from multiple sides (e.g., Side 1 and Side 2) into a single definition.

- Instead of averaging the raw points, the script **averages the polynomial coefficients** (P_{avg}) directly:

$$P_{avg} = \frac{1}{N} \sum_{i=1}^{N} P_{side\_i} \tag{2.5}$$

- It then recalculates the Z coordinates for the entire grid using these averaged parameters, ensuring a mathematically consistent surface.

2.4.2 Usage

Run the analysis script from the terminal:

```
python s4_Choosed_plane.py
```

Output:

- **Subtraction Files:** `SideX__subtraction.json` (The individual corrected surfaces).
- **Final Average:** `Final_Average__average_subtraction.json` (The master profile used for reconstruction).

2.4.3 Configuration

The analysis logic is defined in the `main()` function variables:

- **Degrees to Compare:**

```
degrees_to_use = [1, 5]
```

This list defines which surfaces are subtracted. The logic is always `Index 1 - Index 0` (e.g., Degree 5 minus Degree 1). This is critical for determining what frequency of surface features is retained.

- **Sides to Process:**

  ```
  sides = ["Side1", "Side2"]
  ```

  Defines which measurement sets are included in the final average. If a sample has 4 measured sides, all 4 should be listed here to improve the statistical reliability of the result.

2.5 Stage 5: Surface Reconstruction

Script: `s5_Rebuild.py`

The final stage of the preprocessing pipeline transforms the abstract mathematical models derived in Stage 4 back into a discrete, tangible geometry. This script generates a structured grid of points representing the T-shaped sample, with surface topography defined by the averaged polynomial equation.

2.5.1 Purpose and Functionality

While raw measurement data consists of unstructured scatter plots, simulation software (FEA) often requires structured meshes. This script performs three key geometric operations:

1. Geometric Definition (Shapely)

The script defines the ideal boundaries of the T-shaped sample using the `Shapely` library. It constructs the geometry by uniting two polygonal rectangles:

- **Horizontal Bar:** Defined from $(0, 0)$ to $(h_{width}, h_{thickness})$.
- **Vertical Bar:** Positioned with specific offsets relative to the horizontal bar.

The function `create_t_polygon_shapely` ensures the boundaries form a valid, continuous T-shape.

2. Regular Grid Generation

Inside these boundaries, the script generates a clean mesh. The function `generate_points_in_shape_via_` subdivides the geometry into a regular grid:

- It calculates discrete points $(x, y, 0)$ based on a configurable density (N_x, N_y).
- It automatically removes duplicate points where the vertical and horizontal bars overlap to prevent mesh errors.

3. Z-Coordinate Assignment

For every point in the generated grid, the script calculates the height (Z) using the parameters from the "Final Average" surface.

$$Z_{point} = P_{average}(x_{grid}, y_{grid}) \times \text{Scaling Factor} \tag{2.6}$$

Note on Units: The script includes a scaling operation in `update_points_with_parameters` (currently set to $\times 100$), which converts the Z-values (typically in meters from the polynomial calculation) into the desired unit (e.g., centimeters) for the output file.

2.5.2 Usage

To generate the reconstructed surface, run:

```
python s5_Rebuild.py
```

Input: The script looks for `Final_Average__5_average_subtraction.json` (hard-coded in `read_parameters_from_file`).

Output: Saves a file `Avearege__Rebuild.json` containing the structured point cloud ready for simulation import.

2.5.3 Configuration

Users can adjust the mesh density and geometry through the following parameters in the `main()` function:

- **Mesh Density Factor (`n_factor`):**

  ```
  n_factor = 1  # Multiplier for point density
  ```

  Increasing this value (e.g., to 2 or 5) proportionally increases the number of divisions (N_x, N_y) in both the horizontal and vertical bars, resulting in a finer resolution mesh.
- **Sample Dimensions:** The physical dimensions of the T-shape are imported from `Exp_Data.s1_exp.mean_dim`. To reconstruct a sample with different proportions, the dimensions in that external module must be updated.

2.6 Stage 6: Visualization and Quality Assessment

Scripts: `Plane_plot.py` and `Choosed_plot.py`

Numerical metrics like RMSE are insufficient for fully characterizing surface quality. A low error value might mask systematic geometric distortions or localized defects. This stage provides both static statistical charts and interactive 3D models to visually validate the fitted equations.

2.6.1 Static 2D Analysis (Plane_plot.py)

This script generates comprehensive diagnostic dashboards for every fitted surface found in the data directory. It leverages `Matplotlib` and `Seaborn` to create high-resolution static images.

Diagnostic Layout

For each fitted plane, the script produces a 6-panel figure (`_seaborn_layout.png`) containing:

1. **Fitted Z Scatter:** Top-down view of the mathematical surface.
2. **Original Data Overlay:** Visualizes the raw "Steps" data to verify alignment.
3. **Reconstructed Surface (T-Shape):** Uses the utility `plot_plane_reconstruction_on_ax` to render the continuous polynomial surface.
   - *Masking:* It applies a boolean mask via `create_t_shape_mask` to render only the T-shaped region, hiding the invalid rectangular bounding box.
4. **Residual Heatmap:** A spatial map of errors ($Z_{measured} - Z_{fitted}$). This highlights if errors are random (good) or clustered in specific regions (bad).
5. **Residual Histogram:** A distribution plot with Kernel Density Estimation (KDE) to verify if errors follow a normal distribution.
6. **Statistics Panel:** Displays the exact equation string, RMSE, and Max Error.

Method Comparison

If both Standard and Chebyshev fits exist for the same side/degree, the script automatically generates a **Comparison Plot**. This side-by-side analysis calculates the percentage improvement of one method over the other.

2.6.2 Interactive 3D Visualization (Choosed_plot.py)

While 2D plots are good for statistics, 3D plotting is essential for spatial understanding. This script uses the `Plotly` library to generate an interactive HTML file.

Functionality

The script creates a 2x2 grid of 3D scenes:

- **Grid Generation:** It converts the polynomial equation into a dense regular grid (default 100×100) using `generate_plane_plot_data`.
- **Layers:**
  - *Surface:* The continuous fitted mesh (opacity 0.8).
  - *Points:* The original measurement points overlaid as black dots.
- **Interactivity:** The output is saved as `3D_planes_visualization.html`. Opening this file in a web browser allows the user to rotate, zoom, and pan the models to inspect specific defects or curvature.

2.6.3 Usage and Configuration

Running the Scripts

```
# Generate static statistics/residuals
python Plane_plot.py

# Generate interactive 3D model
python Choosed_plot.py
```

Configuration

- **Z-Axis Scaling:** In `Plane_plot.py`, the visualization limits are hardcoded to focus on surface texture:

  ```
  z_min_val = -0.01
  z_max_val = 0.02
  ```

  These should be adjusted if the sample deviations exceed this range.
- **Target Files (3D):** The `Choosed_plot.py` script is currently hardcoded to look for specific file names in the `main()` function (e.g., `Side1__6_subtraction.json`, `Final_Average...`). Users must update these paths to match the specific degree or file names they wish to inspect.

3

Module B: Profile Analysis (Milling)

3.1 Stage 1: Data Ingestion and Alignment

Script: `s1_Outline_process.py` (Milling Variant)

Unlike the T-shape samples, the milling profile analysis requires the synchronization of two distinct measurement passes: Left (L) and Right (R). Due to the physical nature of the scanning process, these datasets often represent opposing traversal directions and require geometric alignment before averaging.

3.1.1 L/R Pairing and Directional Correction

The script operates on the `Sample_og/Milling Samples` directory. It identifies corresponding sample pairs by parsing folder names matching the pattern `Sample<N>L` and `Sample<N>R` (e.g., `12L` and `12R`).

The function `calc_average_and_save_json` implements a specific alignment algorithm to handle the inverted scan directions:

1. **Data Truncation:** To ensure matrix compatibility, both datasets are trimmed to the length of the shortest file ($\min(N_L, N_R)$).
2. **Array Reversal (Mirroring):** The "Left" sample is treated as the reference frame. The "Right" sample is aligned to this reference by taking its *last* N points and reversing their order. This corrects the spatial inversion caused by the bi-directional scanning:

```
# s1_Outline_process.py
paired_L = data_L[:min_len]
paired_R = data_R[::-1][:min_len]  # Reverse slice
```

3. **Composite Averaging:** The final point cloud is generated by calculating the arithmetic mean of the aligned arrays:

$$P_{avg} = \frac{P_{aligned\_L} + P_{aligned\_R}}{2} \tag{3.1}$$

3.1.2 Progressive Statistical Cleaning (IQR)

After alignment, the data is subjected to an iterative cleaning process via `run_iterative_iqr`. Unlike the single-pass approach used in some workflows, this module employs a **progressive tightening strategy** to preserve edge fidelity while removing noise.

The Interquartile Range (IQR) filter is applied sequentially with decreasing tolerance factors, defined in `iqr_params_per_iteration`:

Iteration	Factor (k)	Purpose
1	1.5	Removal of gross outliers (large spikes)
2	1.0	General noise reduction
3	0.9	Fine filtering of profile edges

Table 3.1: *Progressive IQR thresholds for milling profiles*

A point is only retained if it satisfies the condition $Q_1 - k \cdot IQR \leq x \leq Q_3 + k \cdot IQR$ simultaneously for all active dimensions (X, Y, Z).

3.2 Stage 2: Visual Inspection and Manual Refinement

Script: `s2_Outline_gui.py`

Following the automated IQR filtering, the profile data often requires a final manual pass to remove artifacts such as loose chips or measurement spikes that statistical methods failed to catch. This stage utilizes a lightweight GUI to allow operators to interactively "prune" the profile.

3.2.1 Smart Data Loading

The script is designed to automatically pick up where the previous stage left off. It scans the output directory and selects the input file based on a specific hierarchy:

1. **Priority 1:** The highest numbered iteration file (e.g., `Sample1_Iter3.json`).
2. **Priority 2:** If no iterations exist, it falls back to the average file (`Sample1_Average.json`).

This ensures that the user is always editing the most refined version of the dataset available.

3.2.2 Interaction Logic: Pixel-Based Deletion

Unlike the coordinate-based thresholding used in other modules, this viewer employs a **screen-space interaction model** for point removal. This approach is more intuitive

for visual cleaning, as it depends on what the user *sees* rather than the physical scale of the data.

When the user clicks on the plot:

1. The script transforms the physical coordinates (x, y) of all points into screen coordinates (px, py) using the matplotlib transformation `ax.transData.transform`.
2. It calculates the Euclidean distance in pixels between the mouse click and every data point:
$$d_{pix} = \sqrt{(px_{point} - px_{click})^2 + (py_{point} - py_{click})^2} \tag{3.2}$$
3. If the minimum distance is less than the threshold (configured to **10 pixels**), the closest point is deleted.

3.2.3 Usage and Output

```
python s2_Outline_gui.py
```

Controls:

- **Sample Selection:** A dropdown menu allows quick switching between samples found in the directory.
- **Visual Aids:** Users can toggle "Equal aspect" to see the true physical proportion of the profile or adjust point size/color for better visibility of defects.
- **Save:** Clicking "Save Modified Data" writes the current state to a new file suffixed with `_Modified.json`. This preserves the original automated output for traceability.

3.3 Stage 3: 1D Curve Fitting

Script: `s3_Curve_process.py`

Unlike the bivariate surface fitting in Module A, milling profiles are modeled as univariate polynomials $Z = P(x)$. This stage fits curves of varying degrees (1 to N) to the cleaned profile data, utilizing Ridge Regression to control overfitting.

3.3.1 Regularized Fitting (Ridge Regression)

To prevent Runge's phenomenon—where high-degree polynomials oscillate wildly at the edges—the script employs **Ridge Regression** (Tikhonov Regularization).

Instead of minimizing just the residual sum of squares ($||Ax - b||^2$), the algorithm minimizes:

$$J(\beta) = ||\mathbf{X}\beta - \mathbf{y}||^2 + \alpha||\beta||^2 \tag{3.3}$$

Where:

- $\mathbf{X}$ is the Vandermonde matrix of powers $[1, x, x^2, \dots]$.

- β are the polynomial coefficients.
- α (configured as `RIDGE_ALPHA`) is the penalty term.

The solution is computed via the closed-form normal equation modified for regularization:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{3.4}$$

In the script, the value of α defaults to **1.0** (set by `RIDGE_ALPHA = 1`), providing a moderate penalty to large coefficients.

3.3.2 Coordinate Normalization and Substitution

To ensure numerical stability during the fitting process, the X-coordinates are mapped to the interval $[-1, 1]$. However, the output coefficients must describe the curve in the original physical coordinate system.

The script handles this via **Linear Substitution** using Horner's method:

1. **Normalize:** Transform $x \to t$ where $t \in [-1, 1]$.
2. **Fit:** Calculate coefficients C_t for $P(t)$.
3. **Substitute:** Map back to x using the linear relationship $t = ax + b$.

$$P(x) = \sum C_{t,i}(ax + b)^i \tag{3.5}$$

The function `_compose_linear_substitution` performs this expansion mathematically, returning coefficients valid for raw input values.

3.3.3 Usage

```
python s3_Curve_process.py
```

Process Flow:

- The script searches for the most recent data file (Modified > Iteration > Average).
- It iterates from Degree 1 to `MAX_DEGREE` (default: 7).
- For each degree, it calculates metrics (MSE, RMSE, Max Error).
- **Output:** JSON files for each degree (e.g., `Sample1_Curve_deg5.json`) and a summary file comparing all degrees.

3.4 Stage 4: Comparative Analysis and Averaging

Script: `s4_Choosed_curve.py`

Similar to the surface analysis in Module A, this stage isolates specific profile features (such as roughness or waviness) by removing underlying forms (tilt or curvature) and generates a statistical average of the sample batch.

3.4.1 Tilt Correction via Curve Subtraction

To isolate the desired surface frequency, the script subtracts a lower-degree polynomial (representing form/tilt) from a higher-degree polynomial (representing the profile detail).

Re-fitting Mechanism

Unlike simple algebraic subtraction of coefficients, this script employs a numerical re-fitting approach to ensure robustness:

1. **Evaluation:** It generates a synthetic domain of x points (default range $[-10, 10]$).
2. **Difference Calculation:** For every x, it computes the difference between the two loaded polynomial models:

$$z_{diff}(x) = P_{high\_degree}(x) - P_{low\_degree}(x) \tag{3.6}$$

3. **Regression:** It fits a **new polynomial** to these difference points (x, z_{diff}) to obtain the parameters of the corrected curve.

3.4.2 Master Profile Averaging

Once the subtraction curves are generated for all samples, the script computes a "Master Profile" to represent the entire batch.

This is achieved by averaging the polynomial coefficients directly:

$$\mathbf{C}_{avg} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{C}_i \tag{3.7}$$

Where $\mathbf{C}_i$ is the vector of coefficients for sample i. This averaged parameter set is saved as `Average_Subtraction.json` and serves as the input for the final reconstruction stage.

3.4.3 Usage and Configuration

```
python s4_Choosed_curve.py
```

Configuration

The behavior is controlled by variables within the `main()` function:

- **Degrees to Subtract:**

```
deg_high = 2
deg_low = 1
```

Defines which polynomial orders are used. In this default configuration, the script removes linear tilt (Degree 1) from the quadratic form (Degree 2).

- **Sample Range:** The script iterates through a hardcoded range of samples (e.g., `range(1, 10)`). This loop must be adjusted to match the actual number of samples processed in previous stages.

Output:

- **Individual Subtractions:** `SampleX_Subtraction.json` saved in the `Choosed_curve` directory.
- **Final Average:** `Average_Subtraction.json`.

3.5 Stage 5: Surface Extrusion and Reconstruction

Script: `s5_Rebuild.py` (Milling Variant)

The final stage of the profile analysis converts the 1D mathematical curves derived in Stage 4 into full 3D surfaces. Since the milling samples are geometrically uniform along the cutting axis, the 3D model is generated via **linear extrusion**.

3.5.1 Extrusion Logic

While Module A fits a bivariate surface $Z = f(x, y)$, Module B assumes the geometry is translationally invariant along the Y-axis. The reconstruction logic generates points (x, y, z) where the height Z is solely dependent on X:

$$Z(x, y) = P_{curve}(x) \quad \forall\, y \in [0, H] \tag{3.8}$$

The function `cria_malha_extrudada` implements this by:

1. Retrieving the physical dimensions (W, H) from the `Mean_dim_workpiece` configuration.
2. Iterating through N_x divisions along the X-axis to calculate $z = P(x)$.
3. For each x, replicating the same z value across N_y divisions along the Y-axis.

3.5.2 Grid Generation and Compatibility

To ensure the output is compatible with the project's standard visualization tools (which expect 3D plane data), the script wraps the 1D logic into a standard 3D structure.

- **Mesh Density:** The script generates a regular grid defined by `Nx=100` and `Ny=100`.
- **Wrapper Function:** A helper function `calc_z_wrapper` is used to adapt the 1D evaluator to the 4-argument signature expected by the shared utility `save_plane_data_json`.

```
def calc_z_wrapper(params, x, y, degree):
    # Ignores 'y' and 'degree', evaluating only based on 'x'
    return calcula_z_da_curva(params, x)
```

- **Output Path:** The results are saved to `Sample_postprocess/Choosed_plane`. This intentional placement allows the visualization scripts (Stage 6) to load these extruded profiles as if they were standard surfaces.

3.5.3 Usage

```
python s5_Rebuild.py
```

The script processes all `*_Subtraction.json` files found in the curve directory and outputs corresponding reconstructed JSON files ready for 3D plotting or simulation import.

3.6 Stage 6: Visualization and Utilities

Scripts: `Choosed_plot_Html.py`, `Plane_plot.py`, and various utilities.

Visual validation is the final step in confirming that the extrusion process (Stage 5) has correctly transformed the 1D profile into a coherent 3D surface. This stage relies on a set of core utility modules and offers both interactive 3D rendering and quick 2D debugging tools.

3.6.1 Core Utilities

The entire pipeline relies on two foundational helper modules that handle data abstraction and plotting geometry.

- **`plane_data_utils.py`:** This module acts as the I/O layer for the project. It standardizes how measurement data is saved and loaded. Key functions include:
  - `save_plane_data_json`: Serializes fitted parameters, equations, and point residuals into a unified JSON format.
  - `extract_points_from_plane_data`: Decouples the storage format from analysis, returning NumPy arrays $(X, Y, Z_{orig}, Z_{fit})$ for easy plotting.
- **`plane_plot_utils.py`:** Provides geometric primitives for visualization. Although primarily designed for T-shape masking in Module A, it contains the logic for projecting reconstructed surfaces onto Matplotlib axes via `plot_plane_reconstruction_on_ax`.

3.6.2 Production Visualization

For final reporting and deep analysis, the workflow uses two primary scripts:

Interactive 3D Surface (`Choosed_plot_Html.py`)

This script is the 3D counterpart to the extrusion process. It scans for files ending in `_Rebuild_from_curve.json` and generates an interactive HTML dashboard using **Plotly**.

- **Visual Output:** It renders the extruded surface (Z constant along Y) overlaid with the original measurement points.
- **Interactivity:** Users can rotate and zoom to inspect edge artifacts or extrusion consistency.

Statistical Dashboard (`Plane_plot.py`)

Designed for rigorous quality control, this script uses **Seaborn** to generate static dashboards. It calculates:

- Residual Histograms (Normal distribution check).
- RMSE and Max Error metrics.
- Heatmaps of deviation ($Z_{measured} - Z_{model}$).

3.6.3 Development and Debugging Tools

Several lightweight scripts are included for rapid prototyping and checking intermediate results. These are intended for developer use rather than final reporting:

- **`Cursves_plot.py` / `copy.py`:** Simple Matplotlib scripts with hardcoded file paths. They allow developers to quickly overlay the raw points against specific fitted curves (e.g., Degree 2 vs. Degree 4) to tune fitting parameters like the Ridge alpha.
- **`Choosed_plot.py`:** A basic 2D viewer that iterates through all curves in the `Choosed_curve` directory and plots them on a single axis to compare profiles across different samples.
- **`Choosed_plot_mpl.py`:** A minimalist 3D scatter plotter using Matplotlib's `mplot3d`. It serves as a lightweight alternative to the Plotly viewer when quick, non-interactive checks are needed.

4
Data Conversion Utilities

The simulation outputs generated by Abaqus are stored in the proprietary ODB database format. To facilitate high-performance post-processing and open-source visualization (e.g., ParaView), this module provides a pipeline to convert these files into the **XDMF + HDF5** standard.

The process is divided into three stages: configuration/orchestration, raw data extraction (Python 2.7), and final binary compilation (Python 3.x).

4.1 Stage 1: Configuration and Orchestration

Scripts: `Odb_Npz_Parameters.py`, `ODB_2_XDMF.py`

The conversion process is initiated by the orchestration layer, which manages configuration loading, directory discovery, and batch processing logic.

4.1.1 Parameter Management

The class `ODB2NPYParameters` acts as the bridge between the project's global `config.json` and the conversion scripts.

- **Config Loading:** It reads the global configuration to determine the root directories for specific simulation methods (e.g., "Contour Method").
- **Output Paths:** It automatically defines the output structure, creating a `npy_files` directory within the simulation folder to store intermediate results.

4.1.2 Batch Processing

The `OdbBatchConverter` class handles the iteration over multiple simulation files:

- **Discovery:** It scans the target directory for all files ending in `.odb`.
- **Structure Inspection:** Before conversion, it can inspect the ODB to list available Steps and Instances (nodes/elements counts), helping to validate the file integrity.

- **Unicode Handling:** It includes a `safe_str_convert` utility to handle potential encoding issues when passing file paths between modern Python environments and the older Abaqus Python kernel.

4.2 Stage 2: ODB Data Extraction (Abaqus API)

Script: `Odb_Npz_Converter.py`

This script runs within the Abaqus Python 2.7 environment. Its primary function is to deconstruct the finite element model into raw NumPy arrays ('.npy'), isolating the proprietary dependency from the rest of the pipeline.

4.2.1 Geometry Extraction

The converter aggregates mesh data from specified instances into a global coordinate system.

- **Node Mapping:** Abaqus node labels are not necessarily sequential or zero-indexed. The script creates a mapping dictionary `node_mapping` to translate arbitrary Abaqus labels (e.g., Node 105) into sequential array indices (e.g., Index 0) for the output arrays.
- **Topology:** It extracts element connectivity and offsets, converting them into the flat array format required by visualization standards.

4.2.2 Field Output Processing

The method `_process_temporal_data_optimized` iterates through simulation steps and frames to extract physical results.

- **Optimization:** To manage memory usage for large models, data is processed in "chunks" and saved immediately to disk, rather than holding the entire history in RAM.
- **Invariants:** The script leverages pre-calculated Abaqus invariants (Mises, Max Principal) when available in the `bulkDataBlocks`. This avoids manual recalculation and ensures consistency with Abaqus viewer results.
- **Thresholding:** A `stress_threshold` (default 10^{-6}) is applied to filter out numerical noise. Values below this threshold are set to zero.

4.3 Stage 3: XDMF/HDF5 Compilation

Script: `Npy_2_Xdmf.py`

The final stage takes the fragmented '.npy' files generated in Stage 2 and compiles them into a single, high-performance hierarchical file format suitable for ParaView and the viewer made by the autor.

4.3.1 HDF5 Hierarchical Structure

The `Npy2XdmfConverter` builds a binary HDF5 file (`S_batch.h5`) organized by model:

- `/ModelName/geometry/`: Stores the `coordinates` and `connectivity` arrays.
- `/ModelName/topology/`: Stores `element_types` and `offsets`.
- `/ModelName/time_series/`: A nested structure storing field data (Displacement, Stress) organized by Step and Frame.

4.3.2 XDMF Wrapper Generation

To make the HDF5 data readable, the script generates an XML wrapper (`S_batch.xdmf`).

- **Temporal Grid:** It defines a 'GridType="Collection"' with 'CollectionType="Temporal"'. This links specific simulation times to the static geometry and dynamic field datasets in the HDF5 file.
- **Data Precision:** It handles precision types explicitly, defining Geometry as `Float` (Double precision) and Topology as `Int`, ensuring correct interpretation by visualization software.

5 Element and Mesh Processing

Module Path: `src/ElementProcess/`

This module handles the mapping of stress fields onto Abaqus finite element meshes. Depending on the data source, the workflow splits into two distinct paths:

1. **Analytical Workflow (3 Steps):** Generates synthetic stress fields for benchmarking.
2. **Data-Driven Workflow (2 Steps):** Maps experimental/simulation data (from HDF5) directly onto the mesh.

5.1 Workflow Orchestration

Script: `Elements_main.py`

This script primarily orchestrates the **Analytical Workflow**, serving as a pipeline for generating and mapping theoretical stress fields onto a mesh. For Data-Driven workflows (Real Data), the process typically bypasses the synthetic generation steps in favor of direct HDF5 mapping.

5.1.1 Analytical Pipeline (3 Scripts)

When running in analytical mode, the pipeline executes three discrete stages:

1. **Extraction (s1):** Parses geometry and topology from the `.inp` file.
2. **Synthetic Generation (s2):** Calculates a theoretical stress distribution (e.g., cylindrical) based on the mesh bounding box. Executed via `s2_RE_Field.py`.
3. **Interpolation (s3):** Maps the synthetic cloud points onto the element centroids. Executed via `ElementTensionInterpolator` in `s3_RE_Interpolator.py`.

5.1.2 Data-Driven Pipeline (2 Scripts)

For real simulation data (e.g., Contour Method or RSA), the workflow is condensed. Since the source data (HDF5) already contains spatial information, the generation and

interpolation steps are merged:

1. **Extraction (s1):** Same as above; prepares the target mesh centroids.
2. **Direct Mapping (s2):** Uses `s2_RE_ExnCon.py` to read the HDF5 source and map stresses directly to the target centroids using KDTree algorithms, skipping the intermediate synthetic generation.

5.2 Stage 1: Element Extraction

Script: `s1_Ele_Extractor.py`

The first stage of the pipeline converts the unstructured text data from Abaqus input files into structured tabular data (CSV/Pandas DataFrames) suitable for mathematical processing.

5.2.1 Input Parsing Logic

The script manually parses the `.inp` text format to identify geometric definitions:

- **Element Parsing:** The function `extract_elements_from_inp` scans for the `*ELEMENT` keyword. It extracts the Element ID, the Element Type (e.g., C3D8R), and the list of connected Node IDs that define the element's topology.
- **Coordinate Extraction:** A companion function (referenced as `extract_node_coordinates`) retrieves the spatial (X, Y, Z) coordinates for every node defined under the `*NODE` keyword.

5.2.2 Geometry Calculation

To facilitate field interpolation, the script calculates the **centroid** of each element.

$$C_{elem} = \frac{1}{N} \sum_{i=1}^{N} P_{node\_i} \tag{5.1}$$

Where P_{node_i} are the coordinates of the nodes connected to the element. This reduces the element to a single point in space $(X_{center}, Y_{center}, Z_{center})$.

5.2.3 Output

The processed data is saved to the output directory as a tab-separated file containing:

- Element ID
- Element Type
- Centroid Coordinates (X, Y, Z)

This file serves as the geometric basis for the subsequent stress field generation.

5.3 Stage 2: Analytical Stress Field Generation

Script: `s2_RE_Field.py`

This script generates a synthetic residual stress field for validation or initial condition setup. It creates a defined stress distribution (typically cylindrical) based on the geometric limits of the mesh.

5.3.1 Functionality

- **Mesh Generation:** Based on the coordinate limits ($x_{min}, x_{max}, \dots$) extracted from the model information, it generates a homogeneous cloud of points to represent the material volume.
- **Stress Calculation:** It computes the geometric center of the component and calculates stress components ($\sigma_{xx}, \sigma_{yy}, \dots$) analytically relative to this center. This is useful for creating benchmark fields when experimental data is unavailable.

5.4 Stage 3: Field Interpolation

Script: `s3_RE_Interpolator.py`

The final stage maps the generated or measured stress field onto the target finite element mesh. This is critical because the source data (cloud of points) rarely matches the exact centroid locations of the simulation elements.

5.4.1 Interpolation Logic

The class `ElementTensionInterpolator` manages the transfer:

- **Linear Interpolation (Primary):** Uses `scipy.interpolate.LinearNDInterpolator` to estimate stress values at element centroids based on the surrounding source points. This method respects the gradients of the field.
- **Nearest Neighbor (Fallback):** For elements falling outside the convex hull of the source data (edges/corners), the script falls back to `NearestNDInterpolator` to avoid NaN values, ensuring robust mapping for the entire mesh.

5.4.2 Output

The result is a formatted text file containing the interpolated stress tensor for every element, ready to be imported into Abaqus as an Initial Condition (Predefined Field).

5.5 Data-Driven Stress Mapping

Script: `s2_RE_ExnCon.py`

This module implements the core logic for mapping experimental or simulation data onto a target finite element mesh. Unlike analytical generation methods, this script

processes pre-calculated stress fields stored in HDF5 format and maps them to element centroids using spatial algorithms.

5.5.1 Core Processor Architecture

The `StressProcessor` class encapsulates the entire mapping workflow. It is designed to be robust against large datasets by using chunked reading and efficient neighbor search structures.

1. Simulation Discovery and Matching

The processor scans the working directory to pair mesh files with their corresponding results:

- **Mesh Identification:** Finds files matching `Mesh-*--Lenth-*.inp`.
- **Result Association:** For each mesh, it looks for a corresponding HDF5 file (e.g., `S_batch.h5`) in the specified source folder. It uses regex patterns to ensure the correct Load Case (e.g., `step_1_Load`) is extracted.

2. Spatial Mapping (KDTree)

The mapping of stress tensors from the source cloud (HDF5) to the target mesh (Element Centroids) is performed via a k-dimensional tree:

1. **Tree Construction:** A `scipy.spatial.KDTree` is built using the (x, y, z) coordinates of the source data nodes.
2. **Querying:** For each target element centroid, the tree is queried to find the nearest source node within a specified `tolerance` (default 5×10^{-2}).
3. **Tensor Assignment:** The full stress tensor components ($\sigma_{11}, \sigma_{22}, \sigma_{33}, \tau_{12}, \tau_{13}, \tau_{23}$) from the nearest neighbor are assigned to the target element.

3. Abaqus Input Generation

The final output is an ASCII file formatted for direct inclusion in Abaqus:

```
*Initial Conditions, type=STRESS, input=FileName.csv
```

The script writes the mapped stresses in the specific column order required by Abaqus for 3D elements, handling zero-padding for shear components if the source data is incomplete (e.g., 2D to 3D mapping).

5.5.2 Usage

The script can be executed directly or imported as a module:

```
from Modules_python.s2_RE_ExnCon import StressProcessor
```

```
# Initialize with tolerance configuration
processor = StressProcessor(
    base_dir="C:/Simulations",
    tolerance=0.05
)

# Run batch processing
results = processor.process_all_simulations(
    hdf5_folder="C:/Data/HDF5_Source"
)
```

.

5.6 Batch Processing Variant (Exp2 Extension)

Script: `s2_RE_ExnCon2.py`

While the base processor handles the mapping of a single simulation, this script provides a specialized extension class, `StressProcessorBatch`, designed to handle the complex directory structures found in the Profile Analysis (Exp2/Milling) workflow.

5.6.1 Inheritance and Architecture

The class inherits directly from the core `StressProcessor` (defined in `s2_RE_ExnCon.py`). This design ensures that the heavy computational tasks—HDF5 extraction, KDTree construction, and stress tensor mapping—remain identical to the base implementation, ensuring consistency across experiments.

```
class StressProcessorBatch(StressProcessor):
    # Inherits all mapping logic, overrides only discovery methods
```

5.6.2 Specialized Discovery Logic

The primary modification lies in the `find_simulations` method. Unlike the standard workflow which assumes a 1:1 relationship between mesh and result, the milling experiment often generates multiple load cases (S1, S2, etc.) for a single geometry.

- **Pattern Matching:** The script scans for input files matching both the standard pattern (`Mesh-*`) and the variant pattern (`S*_Mesh-*`) to ensure backward compatibility.
- **Case Aggregation:** It normalizes the mesh names to identify the base geometry (removing the `S*` prefix) and aggregates all corresponding HDF5 results. This allows the processor to iterate through every load case found in the source directory and map it to the correct mesh automatically.

5.6.3 Execution

This script is typically invoked when processing large batches of sensitivity analysis simulations where the mesh geometry remains constant but the boundary conditions (and thus the HDF5 results) vary.

```
processor = StressProcessorBatch(base_dir, hdf5_folder)
all_results = processor.process_all_simulations()
```

The output is organized into subfolders (e.g., `Output/S1_Mesh...`) corresponding to each specific load case identified by the batch processor.

Part II

Simulations

6
Core INP Manipulation Library

Module Path: `src/Simulations/_inp_modules/`

This module serves as the foundational library for programmatically interacting with Abaqus Input Files (`.inp`). Unlike standard Python scripts that utilize the Abaqus Object Model (AOM) inside the GUI, this library acts as a standalone parser and modifier, manipulating the ASCII input files directly. This approach offers greater flexibility, performance, and independence from the Abaqus licensing environment during the pre-processing phase.

Note: This is a core module, so the user will rarely need to modify anything here.

6.1 Library Architecture

The library is structured into four functional components that abstract the complexity of finite element definitions.

6.1.1 1. Data Structures and Parsing

Files: `dataclasses.py`, `parser.py`

Defines lightweight data structures (e.g., `Node`, `Element`, `SectionProperties`) to represent FE entities in memory. The `INPParser` class provides robust static methods to handle the case-insensitive and whitespace-variable nature of Abaqus keywords.

6.1.2 2. Geometry Extraction

File: `process.py`

Responsible for interpreting the physical model described in the input file.

- **Entity Reading:** The `ReadEntities` class iterates through the file to construct lists of nodes and elements.
- **Section Mapping:** The `SectionReader` maps geometric properties (like Shell Thickness) to specific element sets.

- **Region Filtering:** Utilities like `RegionElementExtractor` allow extracting specific subsets of elements based on bounding boxes (x_{min}, x_{max}), facilitating localized analysis.

6.1.3 3. Model Modification (The "Injector")

File: `modifier.py`

The core engine for Residual Stress Analysis, allowing the injection of external data into an existing simulation deck.

- **Stress Generation:** The `InitialStressGenerator` converts dictionary-based stress data into formatted `*Initial Conditions, type=STRESS` blocks.
- **Safe Insertion:** The `INPInserter` locates safe injection points within the file structure (e.g., placing Boundary Conditions inside the correct `*Step`) to ensure the generated file runs without syntax errors.

6.1.4 4. Execution Wrappers

File: `runners.py`

Abstracts the command-line calls to the solver. The `AbaqusJobRunner` manages the execution of jobs, handling configuration parameters like CPU cores and GPU acceleration flags automatically.

6.2 Data Structures and Configuration

Script: `dataclasses.py`

To decouple the Python logic from the specific formatting of Abaqus input files, this module defines strongly-typed data structures (using Python's `@dataclass`). These classes act as intermediate representations for the physical entities and execution configurations.

6.2.1 Finite Element Entities

- **Node:** Represents a geometric point in 3D space with an ID (label) and coordinates (x, y, z).
- **Element:** Stores the mesh topology, linking an ID (label) to a list of constituent `Node` IDs and defining the element type (e.g., C3D8R).
- **SectionProperties:** Abstract container for physical properties associated with element sets. It distinguishes between `Solid` (homogeneous) and `Shell` sections, storing critical thickness parameters needed for stress calculation integration points.

6.2.2 Execution Configuration

Class: `AbaqusJobConfig`

This class encapsulates all parameters required to launch an Abaqus solver job via the command line. It includes:

- **Resource Allocation:** Number of CPUs (`n_cpus`) and memory percentage (`memory`).
- **Environment Settings:** Path to the Abaqus executable and scratch directory usage.
- **Job Control:** Timeout limits (default 30 hours) to prevent stalled processes from hanging the pipeline indefinitely.

6.3 Robust INP Parsing

Script: `parser.py`

Abaqus Input Files (`.inp`) are ASCII-based but often contain inconsistent formatting (variations in capitalization, whitespace, and parameter ordering). The `INPParser` class provides static utility methods to handle this variability robustly.

6.3.1 Key Methods

- **`is_header(line, keyword)`:** Performs a case-insensitive check to see if a line starts with a specific keyword (e.g., `*ELEMENT`). This centralizes the logic for identifying Abaqus command blocks.
- **`get_parameter(line, key)`:** Extracts values from comma-separated key-value pairs typical of Abaqus headers.

  ```
  Input: "*ELEMENT, TYPE=C3D8R, ELSET=Set-1"
  Call: get_parameter(line, "ELSET")
  Output: "Set-1"
  ```

  It handles edge cases like spaces around the equals sign or mixed casing, ensuring reliable metadata extraction.

6.4 Input/Output Operations

Scripts: `reader.py`, `writer.py`

These modules abstract the file system interactions, providing specialized readers for the various file formats encountered in the workflow.

6.4.1 Readers

The `reader.py` module implements specific classes for data ingestion:

- **`INPReader`:** A simple wrapper around file reading that ensures consistent encoding (UTF-8) and provides line-by-line access to the input deck.

- `JSONReader`: Used to load the polynomial parameters (degree and coefficients) generated in the Preprocessing phase (Module A/B). It handles the conversion of lists back into NumPy arrays for mathematical operations.
- `StressReader`: A CSV parser designed to read external stress field definitions. It robustly handles comment lines (starting with `#`) and converts tabular stress data into a dictionary mapping Element IDs to stress vectors.

6.4.2 Writers

The `writer.py` module contains the `INPWriter` class. While currently a lightweight wrapper for writing lists of strings to disk, it centralizes file encoding handling, ensuring that the modified Input Files generated by the software are always compliant with the text format expected by the Abaqus solver.

6.5 Model Modification and Injection

Script: `modifier.py`

This module acts as the "injector" engine of the library. It is responsible for generating valid Abaqus command blocks (like Element Sets or Initial Conditions) and surgically inserting them into an existing input deck without breaking the file structure.

6.5.1 Content Generators

These classes transform raw data into Abaqus-formatted strings:

- `InitialStressGenerator`: Takes a dictionary of stress tensors and formats them into an `*Initial Conditions, type=STRESS` block. It iterates through each element and its integration points, ensuring the correct CSV format required by the solver.
- `ElsetGenerator`: Automatically creates `*Elset` definitions for the elements receiving stress. This is crucial because Abaqus applies initial conditions to specific element sets, not global IDs.
- `BCGenerator`: Generates `*Boundary` cards for displacement control. It calculates the necessary nodal displacements (e.g., to deform a mesh to match a specific shape) and formats them as `Type: Displacement/Rotation` conditions.

6.5.2 Intelligent Insertion Logic

Class: `INPInserter`

Modifying an INP file requires placing commands in specific sections (e.g., Assembly vs. Step level). The inserter implements context-aware logic:

- **`insert_initial_stresses`:** Scans the file for the first `*Step` keyword. It injects the stress block *before* the step begins, effectively setting the initial state of the simulation. It also handles merging with existing `*Predefined Fields` if present.
- **`insert_elsets`:** Locates the `*End Assembly` marker to inject element sets inside the assembly definition but outside of instance definitions, ensuring global visibility.
- **`replace_material_block`:** A robust method to update material properties. It finds a material definition by name and replaces its entire block (Elastic, Plastic, Density) while preserving the rest of the file structure. It detects the end of a material block by looking for "Block Breaker" keywords like `*Step` or `*Boundary`.

6.5.3 High-Level Writer

Class: `StressINPWriter`

This class encapsulates the entire read-modify-write cycle. It orchestrates the readers, generators, and inserters to produce a final, runnable Abaqus input file with the applied residual stress fields.

6.6 Geometry Interpretation and Processing

Script: `process.py`

While the parsing module handles the text syntax, this module is responsible for reconstructing the semantic meaning of the finite element model. It links abstract element definitions to physical properties and provides spatial filtering capabilities.

6.6.1 Entity Extraction

Class: `ReadEntities`

This class performs a linear scan of the input file to build the core mesh database:

- **Node Parsing:** Reads `*NODE` blocks to populate `Node` objects with spatial coordinates (x, y, z).
- **Element Parsing:** Reads `*ELEMENT` blocks. It extracts the element type (e.g., C3D8R) and the connectivity list (the sequence of nodes defining the element).
- **Set Extraction:** The `read_nset` method parses `*NSET` definitions, allowing the system to identify subsets of nodes referenced by boundary conditions or output requests.

6.6.2 Physical Property Mapping

Class: `SectionReader`

Abaqus defines properties (like thickness or material assignment) on "Element Sets", not on individual elements. This class resolves this indirection:

1. **Section Discovery:** It scans for `*SOLID SECTION` and `*SHELL SECTION` keywords to identify which Element Sets define the physics of the model.
2. **Property Extraction:** For shells, it extracts the defined thickness and number of integration points.
3. **Element-Level Mapping:** It iterates through the model's `*ELSET` definitions to create a direct map:

$$\text{Map}(ElementID) \rightarrow \text{SectionProperties} \tag{6.1}$$

This is critical for stress integration, as the software needs to know if an element is a thin shell (requires thickness integration) or a solid (centroid only).

6.6.3 Spatial Filtering

Classes: `RegionFilter`, `RegionElementExtractor`

For localized analysis (e.g., analyzing only the weld bead region), processing the entire mesh is inefficient. These classes implement a bounding-box filter:

- **Centroid Calculation:** For every element, the geometric center is computed based on its constituent nodes.
- **Box Logic:** Elements are retained only if their centroid falls within the specified window:

$$x_{min} \leq c_x \leq x_{max} \quad \text{and} \quad y_{min} \leq c_y \leq y_{max} \tag{6.2}$$

- **Type Filtering:** Optionally filters by element type (e.g., keeping only "C3D" elements to ignore 2D dummy elements).

6.7 Execution Wrappers

Script: `runners.py`

This module bridges the gap between the Python data structures and the external Abaqus solver. It abstracts the complexity of command-line invocation, process management, and batch reporting.

6.7.1 Single Job Execution

Class: `AbaqusJobRunner`

This class handles the execution of a single simulation job. It is designed to be robust on Windows environments where path handling can be problematic.

- **Command Construction:** The method `_build_command_string` constructs the exact CLI string required by the Abaqus driver. It handles:

```
abaqus job=JobName input="Path With Spaces.inp" cpus=4 memory=90 scra
```

It ensures all paths are correctly quoted to prevent errors.

- **Resource Management:** It translates the configuration object (from `dataclasses.py`) into solver flags, setting CPU affinity and memory limits dynamically.
- **Scratch Handling:** If enabled, it automatically creates and assigns a scratch directory for temporary solver files, keeping the main output directory clean.

6.7.2 Batch Orchestration

Class: `INPRunner`

This class manages the execution of multiple input files in sequence.

- **Process Monitoring:** It launches the Abaqus process and monitors its return code. A code of `0` implies success, while anything else triggers error logging.
- **Reporting:** After the batch completes, it generates a `summary_report.txt` containing execution times and status (OK/ERROR) for every file processed. This allows the user to quickly identify failed simulations in large batches.

7 Automation Pipeline Core

Module Path: `src/Simulations/pipeline/`

This module functions as the high-level orchestration layer for the entire simulation framework. While individual scripts (like `cm_main.py`) define the specific physics of an experiment, the `pipeline` module handles the "logistics": directory management, configuration loading, tool chaining, and execution flow control.

Architectural Stability: This module is designed as a closed, stable framework. It provides the standardized infrastructure upon which specific simulation scripts are built. **Users are typically not expected to modify these files**, as changes here affect the global behavior of all simulation types.

7.1 Core Responsibilities

The pipeline abstracts repetitive tasks into reusable components, ensuring consistency across different experimental modules (Module A, Module B, etc.).

- **Configuration Management (`config.py`):** Centralizes the loading and validation of the global `config.json`. It ensures that all paths, parameters, and flags are correctly propagated to the solvers.
- **Process Abstraction (`processors.py`, `converters.py`):** Wraps the lower-level tools (such as the Element Extractor or the ODB Converter) into unified Python classes. This allows the main simulation scripts to call complex operations via simple methods like `processor.run()`.
- **Workspace Management (`generator.py`, `clear_dir.py`):** Automates the creation of standardized directory structures (Input/Output/Post) and handles cleanup tasks, ensuring a pristine environment for each simulation batch.
- **Data Standardization (`dataclass.py`):** Defines strict data contracts for passing information between stages, reducing the risk of type errors or missing parameters during complex multi-step simulations.

7.2 Configuration Management

Script: `config.py`

The `ConfigurationManager` class serves as the single source of truth for the simulation parameters. It decouples the Python logic from the user inputs, loading settings from an external `config.json` file and populating a strictly-typed `SimulationConfig` object.

7.2.1 Loading Logic

The `load()` method implements a "fail-safe" loading strategy:

1. **JSON Parsing:** Reads the raw JSON structure.
2. **Path Resolution:** Resolves relative paths (e.g., `./CM_Simulations`) to absolute paths based on the project root, ensuring portability across different machines.
3. **Default Fallbacks:** If specific parameters (like `mesh_step` or `n_cpus`) are missing from the JSON, the manager assigns hardcoded default values (e.g., $E = 210000$ MPa, $\nu = 0.3$) to guarantee the simulation can proceed.

7.3 Workspace Hygiene

Script: `clear_dir.py`

To prevent data contamination between simulation runs—where results from a previous iteration might be mistakenly read by the current job—this script provides a robust cleaning utility.

7.3.1 Functionality

The function `ClearDirectory(target_dir)` performs a deep clean:

- **Validation:** Checks if the target directory exists to avoid errors.
- **Recursive Removal:** Iterates through the directory contents, distinguishing between files (removed via `os.unlink`) and subdirectories (removed via `shutil.rmtree`).
- **Error Handling:** Wraps deletions in try-except blocks to report specific file access errors without crashing the entire pipeline.

7.4 Results Conversion Pipeline

Script: `converters.py`

While the `Conversor` module (Part I) contains the low-level logic for handling data formats, this script acts as the high-level trigger within the automation pipeline. It orchestrates the multi-stage process of transforming proprietary Abaqus results into open formats.

7.4.1 Two-Stage Execution

The `ResultConverter` class manages the bridge between the different Python environments required for extraction:

1. **Abaqus Extraction (Python 2.7):** The method `_run_abaqus_extraction` triggers the `ODB_2_XDMF.py` script using the configured Abaqus command line. It wraps the call in an `AbaqusScriptRunner`, ensuring the proprietary ODB API is accessed correctly to dump raw data into NPY files.
2. **XDMF Compilation (Python 3.x):** Immediately after extraction, the method `_run_npy_to_xdmf` invokes the `NpyBatchToXdmfConverter`. Since this runs in the modern pipeline environment, it efficiently compiles the raw arrays into hierarchical HDF5 files ready for visualization.

7.4.2 Pipeline Integration

This converter is designed to be called at the end of a simulation workflow (e.g., inside `cm_main.py`). It takes a `method_type` (like "Contour Method") as an argument to correctly route the output files to their specific directories.

7.5 Data Standardization

Script: `dataclass.py`

To maintain robustness across the pipeline, raw dictionary data loaded from JSON is immediately converted into a strictly-typed object. The `SimulationConfig` dataclass acts as the central contract for the entire simulation workflow.

7.5.1 Structure

The class groups parameters into logical domains:

- **Paths:** Validated `Path` objects for working directories and scripts (e.g., `geometry_script`).
- **Physics:** Material properties like Elastic Modulus and Poisson's Ratio.
- **Meshing:** Range definitions (`min`, `max`, `step`) for Design of Experiments (DOE).
- **Solver:** Abaqus-specific flags such as `nlgeom` (Non-Linear Geometry) and time incrementation limits.

7.6 Automated Case Generation

Script: `generator.py`

This module implements the "Design of Experiments" (DOE) logic, automatically generating the necessary input files for a parametric study.

7.6.1 Parameter Combination

Class: `ParameterGenerator`

The method `generate_combinations` takes the ranges defined in the configuration (e.g., Mesh Size from 0.6 to 1.0, Length from 50 to 100) and produces a comprehensive list of all permutation dictionaries. It generates a unique `simulation_id` for each case (e.g., `Mesh-0_8--Length-50`) to serve as a key throughout the pipeline.

7.6.2 Geometry Fabrication

Class: `GeometryGenerator`

Once parameters are defined, this class instantiates the physical models:

1. **Environment Setup:** It serializes the parameter dictionary into a JSON string and injects it into the OS environment variables (`SIMULATION_PARAMETERS`).
2. **Abaqus CAE Execution:** It invokes the configured Abaqus Python script in `noGUI` mode. This script reads the environment variables and constructs the `.inp` file programmatically.
3. **Batch Management:** It iterates through the entire list of combinations, ensuring a dedicated directory is created and populated for every simulation case.

7.7 Simulation Processors (Business Logic)

Script: `processors.py`

> **Development Note:** This module is currently identified as the primary candidate for future refactoring. As new simulation types are added, the logic here tends to grow in complexity, and a move towards a more polymorphic architecture (e.g., specific Strategy classes for CM vs. RSA) is planned.

This script implements the specific "business logic" for modifying input files based on the experiment type. It acts as the glue between the static configuration (`config.py`) and the low-level manipulation tools (`_inp_modules`).

7.7.1 Contour Method Processor

Class: `ContourProcessor`

This class manages the application of boundary conditions derived from the surface measurements (Module A/B).

- **Data Ingestion:** It loads the polynomial coefficients (JSON) representing the cut surface. It supports both "Batch Mode" (matching JSONs to samples by name) and "Single Mode" (applying one reference surface to all simulations for sensitivity analysis).

- **Nodal Displacement Calculation:** For every node on the cut surface (identified by `nset_disp_name`), it evaluates the polynomial $Z = P(x, y)$ using the imported `calculate_z_polynomial` function. This translates the experimental roughness into simulation boundary conditions.

7.7.2 Residual Stress Processor

Class: `ResidualStressProcessor` (and variants)

Note: While sharing the same file, this logic handles the mapping of stress fields.

- **CSV Matching:** It implements a heuristic to pair Abaqus input files with their corresponding stress data (CSV/Dataframes) generated by the ElementProcess module. It attempts exact name matching first, falling back to partial containment matching if necessary.
- **Injection Workflow:** Once paired, it uses the `_inp_modules` to:
  1. Generate `*Elset` definitions for the elements receiving stress.
  2. Create the `*Initial Conditions, type=STRESS` block.
  3. Update material properties (Elastic/Plastic) based on the global configuration, ensuring the simulation runs with the correct physical parameters.

8

Abaqus Scripting Interface (ASI) Framework

Module Path: `src/Simulations/_modules/`

> **Architecture Note:** This module represents the deepest layer of the simulation infrastructure. Unlike the text-based manipulation of `_inp_modules`, this framework operates directly within the Abaqus kernel (Python 2.7), utilizing the official API ('mdb', 'part', 'assembly') to construct models programmatically.

8.1 Overview

This library provides a modular, object-oriented abstraction over the verbose Abaqus scripting interface. It is organized into functional domains to separate geometry generation, property assignment, and meshing logic.

8.1.1 Core Components

The framework is divided into four primary sub-packages:

- **Core (`_modules/core`):** Handles the standard Finite Element definitions common to all simulations. This includes Job creation, Mesh control (seeding, element types like C3D8R), and Step definitions.
- **Assignment (`_modules/assigment`):** Manages the physical properties of the model. It contains dedicated modules for Material definition, Section creation, and the instantiation of parts into the assembly.
- **Geometry (`_modules/geometry`):** Implements the "CAD" logic. It uses a Strategy pattern where each simulation type (e.g., `sim_one` for T-Shape, `sim_two` for Milling) has its own geometry generator, ensuring that new experiments can be added without modifying the core logic.

- **Setup & Utilities (`_modules/geometry_setup`, `utilitary`):** Provides high-level helpers for partitioning, datum plane creation, and boundary conditions. The utility module includes **Mixins** for logging and context management, allowing distinct classes to share common functionalities seamlessly.

8.2 Design Philosophy

The module employs a **Mixin-based architecture**. A main simulation class typically inherits from multiple specialized mixins (e.g., `ModelMixin`, `MeshMixin`), composing a complete FEA model builder from small, reusable blocks. This approach manages the complexity of the Abaqus API, making the codebase maintainable and scalable.

8.3 Core Definitions (Job, Step, Mesh)

Package: `_modules/core/`

This package establishes the fundamental Finite Element Analysis (FEA) settings that are consistent across different simulation types. It abstracts the standard Abaqus commands for job submission, time-stepping, and discretization into reusable classes.

8.3.1 Job Management

Script: `_set_job.py`

The `JobMixin` class encapsulates the creation of the analysis job within the Abaqus database (`mdb`).

- **Resource Allocation:** It translates the configuration parameters (CPUs, GPU acceleration, RAM) into the specific arguments required by `mdb.Job()`.
- **Submission:** Provides methods to submit the job programmatically and wait for completion (blocking call), which is essential for the pipeline's sequential execution.

8.3.2 Analysis Step

Script: `_set_step.py`

Defines the physics of the simulation time. The `StepMixin` typically creates a **Static, General** step (`Step-1`).

- **Non-Linearity:** Sets the `nlgeom=ON` flag to account for large deformations, which is critical for accurate residual stress redistribution.
- **Incrementation:** Configures the automatic time incrementation scheme (Initial, Minimum, and Maximum increment sizes) to ensure convergence stability.

8.3.3 Mesh Strategy

Script: `_set_mesh.py` (and submodules)

Meshing in Abaqus scripting is complex because it requires selecting regions (Cells) and assigning specific controls. This framework breaks down the mesh logic into specialized modular components:

1. **Element Type (`_set_mesh_sc8r.py`):** Enforces the use of **C3D8R** elements (8-node linear brick, reduced integration). This element type is chosen for its computational efficiency and robustness in contact/plasticity problems (hourglass control included).
2. **Global Seeding (`_set_mesh_seed.py`):** Applies the global element size target defined in the DOE configuration (e.g., 0.8 mm) to the entire part.
3. **Structured Meshing Controls:**
   - **Sweep vs. Stack (`_set_mesh_sweep.py`, `_set_mesh_stack.py`):** Defines the meshing technique. *Sweep* is used for extrudable geometries, while *Stack* direction is explicitly set to ensure layers are aligned with the cut plane (Z-axis).
   - **Biasing (`_set_mesh_bias.py`):** Allows for variable mesh density, refining the mesh near the cut surface (where stress gradients are high) and coarsening it further away to save computational cost.

8.4 Physical Properties and Assembly

Package: `_modules/assigment/` (sic)

Once the geometry is generated, it must be assigned physical properties and instantiated within the simulation assembly. This package manages the material definitions, section creations, and the hierarchical assembly process.

8.4.1 Material Definition

Script: `_set_material.py`

The `MaterialMixin` class creates the constitutive models in the Abaqus database.

- **Elasticity:** Defines the `Elastic` behavior using Young's Modulus and Poisson's Ratio provided by the global configuration.
- **Plasticity:** Optionally adds `Plastic` behavior. This is crucial for residual stress analysis, as the redistribution of stresses often induces localized yielding. The framework checks if plastic properties are defined in the config before creating this node in the material graph.

8.4.2 Section Management

Scripts: `_set_section.py`, `_set_section_assign.py`

In Abaqus, materials are referenced by "Sections", which are then assigned to geometry regions.

1. **Creation:** The `SectionMixin` creates a **Solid Homogeneous Section**. It links the previously defined material (e.g., "WORK_PIECE_MATERIAL") to this section definition.
2. **Assignment:** The `SectionAssignMixin` applies this section to the actual part geometry. It targets the **Whole Part** (Cells) to ensure the entire volume simulates the specified metal properties.

8.4.3 Assembly Instantiation

Script: `_set_instance.py`

Abaqus distinguishes between "Parts" (geometry definitions) and "Instances" (occurrences in the assembly). The solver runs on the assembly.

- **Instance Creation:** The `InstanceMixin` imports the generated Part into the Assembly module.
- **Naming Convention:** It forces the instance name to match the configuration default (e.g., `T_SHAPE_PART-1`). This strict naming is vital for the external `_inp_modules` (Part II) to correctly locate and inject boundary conditions later in the pipeline.

8.5 Geometry Construction Strategies

Package: `_modules/geometry/`

This package implements the Computer-Aided Design (CAD) logic of the framework. To support multiple experimental setups without code duplication, it employs a **Strategy Pattern**: each simulation type (e.g., T-Shape, Milling) is encapsulated in its own sub-package (`sim_one`, `sim_two`, etc.), but they all expose a consistent `ModelMixin` interface.

8.5.1 Strategy 1: T-Shape (Sim One)

Sub-package: `sim_one/`

Designed for the standard Residual Stress benchmark (Module A), this strategy generates a parametric T-shaped beam.

- **Vertex Calculation (`_get_shape.py`):** Calculates the 2D coordinates of the T-profile cross-section based on the provided dimensions (web width, flange height, etc.). It returns a closed loop of points ensuring geometric continuity.
- **Solid Extrusion (`_set_geometry.py`):** Uses the Abaqus API `BaseSolidExtrude`. It draws the calculated profile on a sketch plane and extrudes it by the specified `Length` parameter to create the 3D part.

8.5.2 Strategy 2: Milling Profile (Sim Two)

Sub-package: `sim_two/`

Designed for the Profile Analysis experiment (Module B), dealing with material removal or complex boundary conditions.

- **Geometry Logic (`_set_geometry2.py`):** Unlike the simple extrusion of Sim One, this module may handle additional geometric features or different orientation requirements specific to the milling setup.
- **Coordinate Handling (`_get_shape2.py`):** Provides the specialized vertex logic required for this specific profile, ensuring the mesh seeds align with the measurement points.

8.5.3 Integration: The Model Mixin

Files: `model_mixin.py` (in each sub-package)

The `ModelMixin` class acts as the standardized connector. Regardless of whether the underlying geometry is a T-Shape or a Milling plate, this mixin provides the main `build_model()` method that the orchestration script calls. This allows the high-level pipeline to switch between experiments simply by importing a different Mixin, without changing the execution logic.

9 Contour Method Simulation Workflow

Main Orchestrator: `src/Simulations/cm_main.py`
CAE Script: `src/Simulations/tests/attempt.py` (Default Geometry Builder)

This module implements the specific workflow for the Contour Method experiment. It serves as the master controller, utilizing the generic tools from the `pipeline` module to execute the simulation lifecycle: form generation, boundary condition application, solving, and result extraction.

9.1 Workflow Orchestration

The `main()` function in `cm_main.py` defines a linear execution path divided into two primary phases.

9.1.1 Phase 1: Simulation Setup and Execution

Controlled by the `default_process` flag, this phase builds and runs the models:

1. **Environment Preparation:** Calls `ClearDirectory` to purge previous results, ensuring data integrity.
2. **Design of Experiments (DOE):** Uses `ParameterGenerator` to create a matrix of simulation cases (varying mesh density, lengths, etc.) based on `config.json`.
3. **Geometry Generation (The "Builder"):** Instantiates `GeometryGenerator`. This generic wrapper calls the specific Abaqus Python script configured in the JSON (typically `attempt.py`). It launches Abaqus CAE in background mode (`noGUI`) to generate the base `.inp` files for every case in the DOE.
4. **Boundary Condition Injection:** Invokes `ContourProcessor`. This is the critical physics step where the surface topography data (measured in Module A/B) is calculated via polynomials and injected into the `.inp` files as nodal displacements.

5. **Batch Solving:** Uses `INPRunner` to submit all generated jobs to the Abaqus solver, managing queues and hardware resources.

9.1.2 Phase 2: Result Conversion

Controlled by the `convertion_process` flag:

- **Automated Extraction:** Triggers the `ResultConverter`. It locates the output ODB files, extracts the stress tensors, and converts them into the XDMF/HDF5 format required for the final correlation analysis.

9.2 Abaqus Geometry Script (The Builder)

Script: `src/Simulations/tests/attempt.py`

This script represents the "factory floor" of the simulation. Unlike the high-level orchestrators running in Python 3, this script executes strictly within the **Abaqus Python 2.7 kernel**. It is responsible for the procedural construction of the T-Shape geometry, meshing, and input file generation.

> **Legacy Architecture:** Unlike the modular "Mixin" approach used in newer experiments (e.g., Module B/Milling), this script employs a monolithic design. It instantiates the `ContourAnalysis` class to perform all modeling steps sequentially in a single pass.

9.2.1 Execution Context and Parameter Injection

Since Abaqus runs in a separate process, passing arguments from the main pipeline to this script requires a specific mechanism:

- **Environment Variables:** The script retrieves simulation parameters (Mesh Size, Length, Max Increment) via the `os.environ['SIMULATION_PARAMETERS']` variable, which is injected by the `GeometryGenerator` before the Abaqus process starts.
- **Path Patching:** To access the project's custom libraries (`_modules`, `Exp_Data`) from within the isolated Abaqus environment, the script dynamically appends relative paths to `sys.path` at runtime.

9.2.2 The `ContourAnalysis` Class

This class encapsulates the procedural logic to build the model from scratch.

1. **Initialization:** Receives explicit physical parameters (e.g., `mesh`, `comprimento`) and solver settings (e.g., `nlgeom`, `initialInc`). It sets up the working directory and prepares the Abaqus MDB (Model Database).

2. **Geometry Construction (T-Shape):** It sketches the T-profile based on hardcoded dimensions (Flange Width, Web Height) and extrudes it by the variable `comprimento`. This rigidity makes it specific to the "Module A" benchmark.
3. **Physical Assignment:** Creates the material (referencing `WORK_PIECE_MATERIAL`) and creates the `Solid Homogeneous Section`, assigning it to the generated part.
4. **Discretization (Meshing):** Applies the "Global Seed" based on the DOE parameter `mesh_size`. It forces the use of standard elements (C3D8R) suitable for stress analysis.
5. **Job & Input Generation:** Instead of submitting the job to the solver immediately, it generates the **Input File (.inp)**. This is a critical design choice: it hands over the control back to the Python 3 pipeline (`cm_main.py`) to perform the actual submission and result management.

10 Residual Stress Analysis (RSA) Workflow

Main Orchestrator: `src/Simulations/rs_main.py`

This module orchestrates the final stage of the experimental validation: simulating the redistribution of residual stresses after the cutting process. It is a dependency-aware workflow that links the results from the Contour Method (CMA) directly into a new Finite Element analysis.

10.1 Workflow Architecture

The `rs_main.py` script acts as a bridge between the *Inverse Calculation* (Module A/B results) and the *Forward Simulation* (Stress Redistribution). Its execution flow is designed to ensure data consistency between these two physical stages.

10.1.1 Dependency Management

Before starting the stress analysis, the script verifies the existence of the required source data.

- **CMA Execution:** The function accepts a `run_cma` flag. If set to `True`, it imports and executes the `cm_main.py` workflow first. This guarantees that the HDF5 files containing the measured stress fields are up-to-date before attempting to map them.

10.1.2 Geometry and Mesh Generation

Similar to the CM workflow, it uses the `GeometryGenerator` to build the target models.

> **Distinct Geometry:** The geometry for RSA is typically different from the CM geometry (e.g., it may represent the cut part in a relaxed state). The

script points to a specific Abaqus script (configured via `rea_directory`) to generate these specific meshes.

10.1.3 Stress Mapping (The Bridge)

This is the core differentiator of this workflow. It integrates the legacy `ElementProcess` module to perform spatial mapping:

1. **Target Node Extraction:** Calls `Nodes_main` (from `ElementProcess`) to extract the centroids of the newly generated RSA mesh. These centroids act as the "Target Points" for interpolation.
2. **Field Interpolation:** Instantiates the `StressProcessor`. It loads the source HDF5 data (from the CM output folder) and uses KDTree algorithms to map the full stress tensor onto the new RSA mesh elements.

   ```
   # Mapping source (CM) -> target (RSA)
   proc = StressProcessor(str(config.rea_directory), ...)
   proc.process_all_simulations(str(cm_hdf5_path))
   ```

   This step effectively transfers the physical state from the first simulation to the second.

10.1.4 Injection and Solving

Once the stresses are mapped and saved as CSVs, the standard pipeline takes over:

- **Injection:** The `ResidualProcessor` injects the `*Initial Conditions, type=STRESS` block into the input files.
- **Solving:** The `INPRunner` submits the jobs to Abaqus to calculate the equilibrium state (redistribution).

11 Residual Stress Analysis (RSA) Workflow

Main Orchestrator: `src/Simulations/rs_main.py`

This module orchestrates the final stage of the experimental validation: simulating the redistribution of residual stresses after the cutting process. It is a dependency-aware workflow that links the results from the Contour Method (CMA) directly into a new Finite Element analysis.

11.1 Workflow Orchestration

The `rs_main.py` script acts as a bridge between the *Inverse Calculation* (Module A/B results) and the *Forward Simulation* (Stress Redistribution). Its execution flow is designed to ensure data consistency between these two physical stages.

11.1.1 Dependency Management

Before starting the stress analysis, the script verifies the existence of the required source data.

- **CMA Execution:** The function accepts a `run_cma` flag. If set to `True`, it imports and executes the `cm_main.py` workflow first. This guarantees that the HDF5 files containing the measured stress fields are up-to-date before attempting to map them.

11.1.2 Stress Mapping (The Bridge)

This is the core differentiator of this workflow. It integrates the `ElementProcess` module to perform spatial mapping between the two distinct geometries:

1. **Target Node Extraction:** Calls `Nodes_main` (from `ElementProcess`) to extract the centroids of the newly generated RSA mesh. These centroids act as the "Target Points" for interpolation.
2. **Field Interpolation:** Instantiates the `StressProcessor`. It loads the source HDF5 data (from the CM output folder) and uses KDTree algorithms to map the full stress tensor onto the new RSA mesh elements.

   ```
   # Mapping source (CM) -> target (RSA)
   proc = StressProcessor(str(config.rea_directory), ...)
   proc.process_all_simulations(str(cm_hdf5_path))
   ```

   This step effectively transfers the physical state from the first simulation to the second.

11.1.3 Injection and Solving

Once the stresses are mapped and saved as CSVs, the standard pipeline takes over:

- **Injection:** The `ResidualProcessor` injects the `*Initial Conditions, type=STRESS` block into the input files.
- **Solving:** The `INPRunner` submits the jobs to Abaqus to calculate the equilibrium state (redistribution).

11.2 Abaqus Geometry Script (The RSA Builder)

Script: `src/Simulations/rsa/REA_Extended_refactored.py`

Just like `attempt.py` in the CM workflow, this script represents the "factory floor" for the Residual Stress Analysis. It executes strictly within the **Abaqus Python 2.7 kernel** to generate the model geometry and simulation steps.

> **Modern Architecture:** Unlike the legacy monolithic design of `attempt.py`, this script fully utilizes the **Mixin-based Framework** documented in Chapter 8. It acts as a controller that coordinates specialized workers (`MeshSetter`, `ModelChangeSetter`, etc.) rather than performing all tasks itself.

11.2.1 The `ContourAnalysisExtended` Class

This class encapsulates the logic for simulating the material removal process.

1. **Context Binding & Propagation:** Upon initialization, it creates the geometry using `GeometrySetterTwo`. Crucially, it uses the `propagate_to()` method to share the `mdb` model context with all other mixins. This ensures that the mesher, the partitioner, and the boundary condition setters all operate on the exact same assembly instance.

2. **Dynamic Partitioning (JSON Driven):** It instantiates a `PlaneGetter` to load cutting plane coordinates (ZX, ZY) from an external JSON file (generated by the Preprocess module). The `RemoveDatumSetter` then creates physical datum planes and partitions the mesh at these exact coordinates, ensuring the finite elements align perfectly with the cut path.
3. **Step Definition (The Physics of Cutting):** Unlike the static analysis of the previous module, this builder defines a multi-step sequence to mimic the experiment:
    - **Step 1 (Material-Removal):** Uses `ModelChangeSetter` to deactivate elements in the "Remove" set, simulating the physical cut.
    - **Step 2 (BC-Removal):** Deactivates the clamping boundary conditions (`BCBelowSetter`), allowing the part to relax and redistribute stresses.
    - **Step 3 (Stabilization):** A final node-release step to ensure numerical convergence.
4. **Input Generation:** Finally, it calls the `JobSetter` not to submit the job, but to generate the `.inp` file. This file is then handed back to the Python 3 orchestrator (`rs_main.py`) for stress injection and execution.