Contents

This page intentionally left blank.

1 Introduction

This software suite constitutes a comprehensive framework for the automated analysis of Residual Stresses (RS) using the Contour Method (CM) and Finite Element Analysis (FEA). It bridges the gap between physical metrology and digital simulation, providing an end-to-end workflow that processes raw surface measurements, generates data-driven boundary conditions, and orchestrates complex simulations in Abaqus.

1.1 Context and Purpose

The evaluation of residual stresses in manufactured components—specifically via the Contour Method—requires a rigorous integration of experimental data into numerical models. The traditional workflow faces several challenges:

- **Data Noise:** Raw surface scans from CMMs or optical scanners contain high-frequency noise and outliers that must be filtered before analysis.
- **Complex Mapping:** Translating a measured 2D surface topography into 3D nodal boundary conditions for FEA is mathematically non-trivial.
- **Simulation Bottlenecks:** Setting up parametric studies (e.g., varying mesh density or cut length) in commercial solvers like Abaqus is labor-intensive and error-prone when done manually.
- **Data Fragmentation:** Results are often scattered across proprietary formats (ODB), spreadsheets, and raw text files, hindering correlation analysis.

1.2 System Architecture

To address these challenges, this framework is architected into two primary domains, managed by a central configuration core.

1.2.1 Part I: Data Preparation & Conversion

This domain handles the "Digital Twin" aspect of the samples:

- **Preprocessing:** Algorithms to clean point clouds, align measurement axes, and fit polynomial surfaces to experimental data (Chapter 3).
- **Data Standardization:** A unified pipeline that converts proprietary Abaqus outputs (ODB) into open scientific formats (HDF5/XDMF) for streamlined post-processing (Chapter 4).
- **Experimental Database:** A central repository (`Exp_Data`) that stores the statistical mean dimensions of physical samples, ensuring simulations match the manufactured reality (Chapter 3).

1.2.2 Part II: Automated Simulation Pipeline

This domain acts as the computational engine:

- **Orchestration:** A high-level Python pipeline that manages the entire lifecycle of a simulation batch—from directory creation to job submission (Chapter 7).
- **Abaqus Scripting Interface (ASI):** A modular, object-oriented framework that runs inside the Abaqus kernel to procedurally generate geometries, meshes, and analysis steps (Chapter 8).
- **Hybrid Workflows:** Specialized solvers for the **Contour Method** (calculating stress from deformation) and **Residual Stress Analysis** (simulating redistribution after material removal), including the mapping of stress fields between dissimilar meshes (Chapter 11).

1.3 Scope of this Manual

This document details the internal structure, logic, and usage of the software modules. It is intended for developers and researchers aiming to extend the framework or understand the specific algorithms used for stress reconstruction and mapping.

2

User's Guide: Step-by-Step Tutorial

2.1 Overview

This tutorial guides you through the complete Residual Stress Analysis workflow for both Experiment 1 (T-shaped specimen) and Experiment 2 (milled block). All entry points are consolidated in the `scripts/` directory.

Workflow Summary:

1. **Data Processing (~5 min):** Pre-process raw measurement files and fit polynomial models to the experimental surface/curve data.
2. **Geometry Setup (<1 min):** Configure the cutting-plane positions using the interactive GUI and save them to `data/e{n}_plane_settings.json`.
3. **Contour Method Simulation (~10 min):** Build meshes, apply boundary conditions, and extract ODB results.
4. **Residual Stress Analysis (~10 min):** Map CM stresses onto the RSA mesh, inject initial conditions, and solve.
5. **Visualization:** Inspect results using the built-in `answer_viewer` or ParaView.

> **Note:** Simulation times assume a mesh-converged run of approximately 50 000 elements.

Expected Outcome: XDMF/HDF5 files representing the full 3-D stress tensor field, ready for post-processing in the built-in viewer or ParaView.

2.2 Entry-Point Scripts

All user-facing commands live in `scripts/`:

Script	Purpose
`e1_process_run.py`	Process Exp1 surface measurements (3-D T-profile).
`e2_process_run.py`	Process Exp2 curve measurements (1-D milling).
`setup_geometry.py`	Launch the cutting-plane configurator GUI.
`e1_simulation_run.py`	Unified CM + RSA runner for Experiment 1.
`e2_simulation_run.py`	Unified CM + RSA runner for Experiment 2.

2.3 Prerequisites

Before proceeding, ensure the following environment is set up:

- **Abaqus 2021 or 2023:** Must be installed. The command `abq2021` (or equivalent) must be accessible via terminal or defined in `data/config.json`.
- **Python 3.10+:** Required for all pipeline orchestration scripts.
- **Dependencies:** Install required packages:

  ```
  pip install -r requirements.txt
  ```

- **Input data:** Raw measurement files placed in `data/input/exp1/` and `data/input/exp2/`.

2.4 Step 1: Process Experimental Data

Goal: Pre-process raw measurement files and compute polynomial surface/curve models that the simulation scripts will consume.

2.4.1 Experiment 1 --- 3-D Surface (T-profile)

```
python scripts/e1_process_run.py
```

Reads all measurement files from `data/input/exp1/`, applies the side-mirroring rule for `Side2`, fits a degree-4 2-D polynomial surface, and writes JSON results to `data/output/exp1/surface_data/`.

2.4.2 Experiment 2 --- 1-D Curves (Milled Block)

```
python scripts/e2_process_run.py
```

Reads all curve measurement files from `data/input/exp2/`, applies mirroring for side `"2"`, fits a degree-2 polynomial with Ridge regularisation (`alpha=1.0`), and writes JSON results to `data/output/exp2/curve_data/`.

2.4.3 Verification

After either script finishes, confirm that `*.json` files (including `Average.json` for Exp1) are present in the respective output directory. Use the notebooks in `notebook/` to visually inspect the fits before proceeding.

2.5 Step 2: Configure Cutting-Plane Geometry

Goal: Define the ZX and ZY cutting-plane positions and the material-removal region. These values are saved to `data/e{n}_plane_settings.json` and consumed by the CAE scripts.

2.5.1 Action: Using the GUI

```
# Experiment 1
python scripts/setup_geometry.py --e1

# Experiment 2
python scripts/setup_geometry.py --e2

# Interactive selection
python scripts/setup_geometry.py
```

The `plane_selector` window opens (`customtkinter`). The specimen outline is drawn automatically from the sample geometry file.

2.5.2 Configuration

1. **Plane ZX (Horizontal Cut):** Enter the Y position in mm. The horizontal dashed line updates in real time.
2. **Plane ZY (Vertical Cut):** Enter the X position in mm. The vertical dash-dot line updates.
3. **Removal Region:** Enter the reference point (X, Y) inside the region to be removed. The hatched rectangle shows which side will be cut away.
4. Click **SAVE JSON**: the file is written and the window closes automatically.

2.5.3 Verification

Open `data/e1_plane_settings.json` (or `e2_plane_settings.json`) in a text editor and confirm that `plane_zx`, `plane_zy`, and `remove_region` contain the expected coordinates.

2.6 Step 3: Contour Method (CM) Simulation

Goal: Generate INP files, run Abaqus jobs, extract ODB results, and convert them to XDMF/HDF5.

2.6.1 Action: Execution

Run from the project root (choose the experiment):

```
# Experiment 1 -- run full pipeline (sim + conversion)
python scripts/e1_simulation_run.py cm --all

# Experiment 2
python scripts/e2_simulation_run.py cm --all
```

Phase flags (can be combined):

Flag	Effect
`--sim`	Geometry generation + Abaqus job submission only.
`--conv`	ODB extraction + NPY-to-XDMF conversion only.
`--all`	Both simulation and conversion phases.

With no arguments the script enters **interactive mode**, presenting numbered menus for pipeline and phase selection.

2.6.2 What Happens Internally

1. `ClearDirectory` removes stale results from `cm_directory`.
2. `ParameterGenerator` builds the DOE matrix from `data/config.json`.
3. `GeometryGenerator` launches Abaqus CAE (`noGUI`) with `cma/script.py` (Exp1) or `cma2/script.py` (Exp2) per parameter set.
4. `ContourProcessor` applies polynomial boundary conditions to every mesh.
5. `INPRunner` submits all jobs.
6. `ResultConverter` extracts ODB data and converts to XDMF/HDF5.

2.6.3 Result Verification

Generated Outputs (inside `cm_directory`):

- `*.xdmf` and `*.h5` — one pair per DOE case, ready for the viewer.
- `*.odb` — Abaqus output databases.

The terminal displays progress logs such as `[INFO] Processing Case: Mesh-0_8--Length-50...`.

2.7 Step 4: Residual Stress Analysis (RSA)

Goal: Map the CMA stress field onto the RSA mesh, inject `*Initial Conditions`, solve, and convert results.

2.7.1 Action: Execution

```
# Experiment 1 -- run full RSA (includes CMA dependency)
python scripts/e1_simulation_run.py rsa --all
```

```
# Experiment 2
python scripts/e2_simulation_run.py rsa --all
```

Phase flags for RSA:

Flag	Effect
`--cma`	Re-run the CMA pipeline first (ensures fresh HDF5 source data).
`--sim`	Stress mapping + Abaqus job submission only.
`--conv`	ODB extraction + NPY-to-XDMF conversion only.
`--all`	`--cma` + `--sim` + `--conv`.

If CMA results from Step 3 are already up to date, skip the CMA re-run:

```
python scripts/e1_simulation_run.py rsa --sim --conv
```

2.7.2 What Happens Internally

1. `ClearDirectory` purges `rea_directory`.
2. `GeometryGenerator` builds RSA meshes using `rsa/script.py` (Exp1) or `rsa2/script.py` (Exp2).
3. `Nodes_main` extracts element centroids from the RSA mesh.
4. `StressProcessor` (Exp1) or `StressProcessorBatch` (Exp2) maps S_{33} from the CMA HDF5 source onto the RSA centroids via KDTree.
5. `ResidualProcessor` injects `*Initial Conditions, type=STRESS` into every `.inp`.
6. `INPRunner` submits all jobs.
7. `ResultConverter` extracts ODB data and converts to XDMF/HDF5.

2.7.3 Verification

Check `rea_directory`:

- CSV files per case confirm that stress mapping completed successfully.
- `*.h5` / `*.xdmf` files confirm successful conversion.
- Abaqus `.odb` files are present in each job sub-directory.

2.8 Step 5: Visualizing Results

Goal: Inspect the XDMF/HDF5 output using the built-in viewer or ParaView.

2.8.1 Option A: Built-in Viewer (`answer_viewer.py`)

Launch from the project root:

```
python -m simulations.gui.answer_viewer
```

1. **File → Open HDF5** — select an `.h5` file from `cm_directory` or `rea_directory`.
2. Use the **Model** combo to switch between DOE cases.

3. Select **Step**, **Frame**, and **Field** (e.g., `stress_tensor`).
4. Click **Update Visualization** or change any combo to refresh the 3-D view.

Key controls:

- **View buttons (+X/+Y/+Z, Iso 1/2):** Abaqus-style camera presets.
- **Clipping mask:** Automatically hides the removed material region using the plane settings loaded from `data/e{n}_plane_settings.json`.
- **Deformation scale:** Amplifies displacement field for visual clarity.
- **Export Image** / **Export Data:** Save a screenshot or the current field array as PNG or CSV/NPY.

2.8.2 Option B: ParaView

1. Open **ParaView**.
2. **File** → **Open** → navigate to the output folder.
3. Select a `.xdmf` file and click **Apply**.
4. In the toolbar, change the variable to `stress_tensor` and use **Split Field** to select the S_{33} component.

2.8.3 Interpretation Hints

- **Red zones (positive):** Tensile residual stress.
- **Blue zones (negative):** Compressive residual stress.
- **Validation:** S_{33} should be near zero at the cut face (boundary condition enforcement).
- **Clipping mask off:** Disable the mask in the Visualization tab to inspect the full geometry including the removed region.

2.9 Troubleshooting & Advanced Usage

2.9.1 Common Issues

Error: "Abaqus command not found"

The pipeline cannot find the solver. Open `data/config.json` in a text editor and set `"abaqus_cmd"` to the full path of your batch file, e.g. `C:/SIMULIA/Commands/abq2021.bat`.

Simulation Hangs / Convergence Failure

Inspect the `.msg` file in the job sub-directory inside `cm_directory` or `rea_directory`.

- "Too many increments": Increase `maxNumInc` in `data/config.json`.
- "Time increment too small": The mesh may be too distorted; increase the mesh-size lower bound in the DOE settings.

Missing HDF5 / XDMF Files

The Abaqus Python ODB extraction failed. Check write permissions in the output directory or available disk space. Re-run the conversion phase only:

```
python scripts/e1_simulation_run.py cm --conv
```

Stress Mapping Produces All Zeros

The CMA HDF5 path pointed to by `cm_hdf5_path` inside the RSA pipeline does not contain valid data. Re-run `cm --all` first, or pass `--cma` to the RSA runner:

```
python scripts/e1_simulation_run.py rsa --cma --sim
```

Viewer Shows No Elements

The clipping mask may be removing everything. Toggle **Apply clipping mask** off in the Visualization tab or verify that `data/e{n}_plane_settings.json` contains valid plane coordinates (regenerate with `setup_geometry.py` if needed).

2.9.2 Advanced: Running Individual Phases

Each runner supports granular control. For example, to regenerate geometry only for Exp2 RSA without re-mapping stresses:

```
python scripts/e2_simulation_run.py rsa --sim
```

2.9.3 Advanced: Customizing Geometry

To adjust the specimen dimensions:

1. Edit the sample geometry file in `data/input/exp{n}/exp{n}_sample01.py`.
2. Run the process script to re-fit the polynomial model.
3. Re-run `setup_geometry.py` to confirm that the updated outline is displayed correctly before starting simulations.
4. Run `rsa/script.py` (Exp1) or `rsa2/script.py` (Exp2) standalone in Abaqus CAE to verify the new mesh builds correctly before launching a full batch.

3 Experimental Data Processing

Module Path: `src/exp_process/`

This module implements the full preprocessing and fitting pipeline for raw experimental surface and curve measurements. It replaces both the previous `Exp_Data` and `Preprocess` modules with a more structured, object-oriented architecture designed to be extended for new experiment types with minimal changes to existing code.

Previously, `Exp_Data` handled physical dimensions and sample configuration while `Preprocess` ran separate per-experiment scripts for cleaning, fitting, and visualization. These responsibilities are now unified under `exp_process`, with the processing logic reorganized into reusable, experiment-agnostic layers. Sample dimensions and simulation configuration remain in separate modules outside `exp_process` (see `data/input/`).

The module handles two distinct experiment types, each with their own geometry and data format:

- **Experiment 1 (T-Shape):** 3D surface measurements from a T-shaped specimen, acquired in multiple passes (`bottom` and `wall` regions) across two measurement sides. Multiple measurements per side are averaged during parsing.
- **Experiment 2 (Rectangular Profile):** 1D curve measurements from a rectangular specimen, acquired in two opposing directions (Left/Right). The L/R pair is averaged and merged into a single profile during parsing, before any further processing.

3.1 Module Architecture

The module is organized into five layers, each with a well-defined responsibility. Data flows top-to-bottom through these layers, with configuration injected at the pipeline level.

1. **Parsers (`parsers/`):** Read raw `.txt` files from disk and return structured NumPy arrays. One parser class per experiment type.

2. **Core** (`core/`)**:** Stateless mathematical and geometric utilities — cleaning, fitting, meshing, transforming, segmenting, and reconstructing. These classes have no knowledge of file paths or experiment types.
3. **Procedures** (`procedures/`)**:** Orchestrate the core utilities into named pipeline stages: `preprocess`, `fitting`, `comparison`, and `validation`. Each procedure reads from and writes to disk as JSON.
4. **Pipeline** (`pipeline/`)**:** High-level entry points that chain procedures together. A user runs the pipeline; the pipeline calls the procedures in order.
5. **Utils & GUI** (`utils/`, `gui/`)**:** JSON I/O helpers and the interactive point cloud viewer used in the validation step.

```
[Raw .txt files]
     |
[ Parsers ]  <-- reading + averaging (Exp1: multi-pass; Exp2: L/R merge)
     |
[ Procedures: Preprocess ]  <-- IQR cleaning, step segmentation, JSON output
     |
[ GUI: Validation ]  <-- manual point deletion (blocking)
     |
[ Procedures: Fitting ]  <-- 1D/2D polynomial fitting, JSON output
     |
[ Procedures: Comparison ]  <-- subtraction, averaging, rebuild output
```

Figure 3.1: *Data flow through the `exp_process` pipeline.*

3.2 Dependencies

The module depends on standard scientific Python libraries. All are listed in `src/exp_process/deps_core` and `deps_gui.py`:

- **Core:** `numpy`, `shapely`, `scipy` (implicit via `numpy.linalg`)
- **GUI:** `matplotlib` (TkAgg backend), `tkinter`

3.3 How to Run

The textttscripts/ directory at the project root contains ready-made run scripts for each experiment. These handle the path setup, sample dimension reading, and pipeline configuration automatically — they are the intended entry point for day-to-day use.

Script	Purpose
`scripts/e1_process_run.py`	Full preprocessing pipeline for Experiment 1 (T-Shape, 3D surface).
`scripts/e2_process_run.py`	Full preprocessing pipeline for Experiment 2 (Rectangular, 1D curve).

Table 3.1: *Scripts in `scripts/`.*

To run the processing pipeline for either experiment, execute the corresponding script from the project root:

```
# From the project root:
python scripts/e1_process_run.py    # Experiment 1
python scripts/e2_process_run.py    # Experiment 2
```

Each script performs the following automatically:

1. Adds `src/` to `sys.path` so `exp_process` is importable without installation.
2. Reads sample dimensions from `data/input/exp`*N*`/` via `ExpProcessor.process()` and derives any required geometric transformation rules (e.g. mirror axis for Exp1 Side2).
3. Assembles the full pipeline config (texttttExp1PipelineConfig or `Exp2PipelineConfig`) with the correct input and output directories.
4. Calls `.run()` on the pipeline, which executes preprocessing, opens the validation GUI, runs fitting, and (Exp1 only) computes the averaged reference surface.

Output is written to:

- **Exp1:** `data/output/exp1/surface_data/`
- **Exp2:** `data/output/exp2/curve_data/`

What to change before running:

- **New sample file:** update the `sample_path` variable at the top of the script to point to the correct sample definition file in `data/input/exp`*N*`/`.
- **Fitting parameters:** adjust `high_degree`, `ridge_alpha`, or `fix_rules` directly in the config block. These are clearly marked in the script with section comments.
- **Output directory:** change `OUTPUT_DIR` at the top of the script; all sub-directories are derived from it automatically.

3.4 File Structure

```
src/exp_process/
    deps_core.py          # core imports
    deps_gui.py           # GUI imports
    parsers/
        _base.py          # AbstractParser
        t_shape.py        # Experiment 1 parser
        rec_shape.py      # Experiment 2 parser
    core/
        cleaner.py        # IQR outlier removal
        fitter.py         # 1D and 2D polynomial fitting
        mesher.py         # grid generation
        operations.py     # model arithmetic (subtract, average)
```

```
    rebuilder.py      # surface/curve reconstruction
    segmenter.py      # step detection
    transformer.py    # geometric corrections
procedures/
    preprocess.py     # Stage 1: clean and segment
    fitting.py        # Stage 3: polynomial fitting
    comparison.py     # Stage 4: model averaging
    validation.py     # Stage 2: GUI launcher
pipeline/
    base.py           # BasePipeline (abstract)
    surface.py        # Experiment 1 full pipeline
    curve.py          # Experiment 2 full pipeline
gui/
    viewer.py         # PointCloudViewer
utils/
    io.py             # JSON save/load utilities
```

3.5 Chapter Organization

The following sections document each layer in detail, including configuration parameters, extension points, and common modification scenarios.

3.6 Parsers

Module path: `src/exp_process/parsers/`

Parsers are responsible for reading raw `.txt` files from disk and returning structured NumPy arrays. They are the only layer with knowledge of the input directory structure and file naming conventions. All downstream code receives plain arrays and has no dependency on file paths.

Each parser implements the `AbstractParser` interface and handles one experiment type. The output of `load()` is always a `dict` mapping an identifier string to a NumPy array, so all procedures can iterate over them uniformly regardless of experiment type.

3.6.1 AbstractParser

File: `parsers/_base.py`

Defines the interface that all parsers must implement. Contains no logic.

Method	Signature	Description
`__init__`	`(input_dir: str)`	Stores the input directory path.
`load`	`(target_id: str) -> dict`	Load raw data for a given ID.
`list_ids`	`() -> list`	List all available IDs in `input_dir`.

Table 3.2: *AbstractParser interface.*

To add a new experiment type, subclass `AbstractParser` and implement both methods. No changes to procedures or pipeline are required as long as the return format is respected.

3.6.2 TShapeParser — Experiment 1

File: `parsers/t_shape.py`

Reads surface measurement data from T-shaped specimens. The input directory is expected to contain files named:

```
{SideID}_Measurment{N}_bottom.txt
{SideID}_Measurment{N}_wall.txt
```

where `N` is the measurement number (integer) and `SideID` is typically `Side1` or `Side2`. Each side has multiple measurements (repetitions), and each measurement is split into a `bottom` region and a `wall` region in separate files.

load(side_id) stacks `bottom` and `wall` arrays vertically per measurement using `np.vstack`, returning:

```
{
    "1": np.ndarray,  # measurement 1: bottom + wall
    "2": np.ndarray,  # measurement 2: bottom + wall
    ...
}
```

The merge of `bottom` and `wall` preserves region identity only implicitly through row ordering. If a region file is missing for a given measurement number, that measurement is skipped with a printed warning.

list_ids() scans `input_dir` for filenames matching the naming pattern and returns the unique side identifiers found (e.g., `['Side1', 'Side2']`).

3.6.3 RecShapeParser — Experiment 2

File: `parsers/rec_shape.py`

Reads 1D curve measurement data from rectangular specimens. The input directory is expected to contain subfolders named:

```
{sample_id}L/   <-- contains one .txt or .csv file
{sample_id}R/   <-- contains one .txt or .csv file
```

where `sample_id` is a numeric string (e.g., `1`, `2`). L and R represent measurements taken from opposing directions on the same specimen, forming a single geometric profile when combined.

load(sample_id) merges L and R as follows: R is reversed (`[::-1]`) to align acquisition direction with L, both are truncated to the shorter length, and their element-wise average is computed. The result is a single array representing the merged profile:

```
{"1": np.ndarray}  # averaged L/R profile for sample 1
```

If either folder or file is missing, the sample is skipped with a printed warning.

list_ids() scans `input_dir` for folders matching `^(\d+)[LR]$` and returns unique numeric identifiers (e.g., `['1', '2', '3']`).

3.6.4 Adding a New Parser

To support a new experiment type:

1. Create `parsers/my_type.py` with a class inheriting `AbstractParser`.
2. Implement `list_ids()` to scan the input directory.
3. Implement `load(target_id)` to return `{id: np.ndarray}`.
4. Create the corresponding pipeline entry point in `pipeline/`.

No changes to `core/` or `procedures/` are required.

3.7 Core

Module path: `src/exp_process/core/`

The core layer contains stateless utilities for all mathematical and geometric operations. No class here reads from or writes to disk, and none has knowledge of experiment types. All methods are `@staticmethod` (except `DataTransformer`, which holds configuration state). This design allows procedures to compose core utilities freely without coupling.

3.7.1 OutlierCleaner

File: `core/cleaner.py`

Removes outliers from point arrays using the Interquartile Range (IQR) method. Each axis is filtered independently using the configured factor.

Method	Description
`filter_iqr(data, factors)`	Removes points outside $[Q1 - f \cdot IQR,\ Q3 + f \cdot IQR]$ for each axis specified in `factors`. Returns the filtered array.

Table 3.3: *OutlierCleaner interface.*

`data` must be shape `(N, 3)`. `factors` is a dict mapping axis names to multipliers, e.g.:

```
OutlierCleaner.filter_iqr(points, {'x': 1.5, 'y': 1.5, 'z': 1.2})
```

Axes not present in `factors` are not filtered. If `data` is empty, it is returned unchanged.

3.7.2 Fitter

File: `core/fitter.py`

Polynomial fitting and evaluation for both 1D (Exp2 curves) and 2D (Exp1 surfaces). All methods return a model `dict` that encodes the polynomial type, degree, and coefficients, allowing evaluation to be decoupled from fitting.

Method	Description
`fit_1d_poly(x, z, degree, ...)`	Fits a 1D polynomial. Supports optional x-normalization to $[-1, 1]$ and Ridge regularization ($\lambda > 0$). Returns coefficients back-transformed to the original x scale.
`eval_1d_poly(x, model)`	Evaluates a 1D model at scalar or array `x` using `np.polyval`.
`fit_2d_poly(x, y, z, degree)`	Fits a separable 2D polynomial of the form $z = \sum_{k=1}^{d}(a_k x^k + b_k y^k) + c$. Uses `np.linalg.lstsq`.
`eval_2d_poly(x, y, model)`	Evaluates a 2D model at arrays `x`, `y`. Returns a `np.ndarray`.

Table 3.4: *Fitter interface.*

The 2D polynomial is separable: cross-terms ($x^i y^j$, $i, j > 0$) are not included. The coefficient vector layout is `[a_1, b_1, a_2, b_2, ..., c]` where `c` is the constant bias stored last. This layout is shared with `ModelOps`.

Model dict structure:

```
# 1D
{"type": "poly_1d", "degree": 4, "coeffs": [...], "norm": {...}, "fit": {...}}

# 2D
{"type": "poly_2d", "degree": 6, "coeffs": [...]}
```

3.7.3 MeshGenerator

File: `core/mesher.py`

Generates structured point grids for surface reconstruction. Used by `Rebuilder`.

Method	Description
`rectangular_grid(width, height, step)`	Regular grid over $[0, \text{width}] \times [0, \text{height}]$ with spacing `step`. Returns flat `(x, y)` arrays.
`t_shape_grid(dims, step)`	Grid of points inside a T-shaped polygon, built as the union of two rectangles (horizontal flange and vertical web). Uses Shapely for containment testing. Returns flat `(x, y)` arrays.

Table 3.5: *MeshGenerator interface.*

`t_shape_grid` expects a `dims` dict with keys `h_width`, `h_thickness`, `v_width`, `v_height`, and optionally `offset_1` for the horizontal position of the web. If `offset_1` is absent, the web is centered on the flange.

3.7.4 ModelOps

File: `core/operations.py`

Arithmetic operations on fitted model dicts. Handles coefficient alignment between models of different degrees, with separate logic for 1D and 2D coefficient layouts.

Method	Description
`subtract_coeffs(model_high, model_low)`	Subtracts `model_low` from `model_high`, padding the lower-degree model with zeros as needed. For 1D, pads from the left (high-degree terms); for 2D, pads term-by-term preserving the bias.
`average_models(models)`	Computes the element-wise mean of coefficients across a list of same-degree models. Raises `ValueError` if degrees differ.

Table 3.6: *ModelOps interface.*

`model_high` must have degree ≥ `model_low`. The returned dict preserves `type`, `degree`, and `norm` from `model_high`.

3.7.5 Rebuilder

File: `core/rebuilder.py`

Reconstructs point clouds from fitted models by evaluating them on structured grids. Composes `MeshGenerator` and `Fitter`.

Method	Description
`rebuild_surface(model, geometry_type, dims, step)`	Evaluates a 2D model on a `t_shape` or `recta` Returns an `(N, 3)` array.
`rebuild_curve_extrusion(model_1d, dims, step_x, step_y)`	Evaluates a 1D model along x and extrudes y. Returns an `(N, 3)` array.

Table 3.7: *Rebuilder interface.*

3.7.6 StepSegmenter

File: `core/segmenter.py`

Detects step discontinuities in point clouds by identifying large relative jumps in x-values after sorting. Used in the Exp1 preprocessing stage to split the merged bottom+wall cloud into individual measurement steps.

Method	Description
`find_steps(points, threshold_percent)`	Sorts points by `(x, y)`, computes relative x-differences, and splits at indices where the difference exceeds `threshold_percent` (default `0.6%`). Returns a list of arrays.

Table 3.8: *StepSegmenter interface.*

3.7.7 DataTransformer

File: `core/transformer.py`

Applies per-side geometric corrections (mirroring, inversion, coordinate offsets) to raw point arrays before fitting. Unlike the other core classes, `DataTransformer` is instantiated with a `rules` dict that maps side IDs to their transformations.

```
transformer = DataTransformer(rules={
    "Side2": {"mirror_x": True, "invert_z": True},
    "Side1": {"offset_x": 5.0}
})
corrected = transformer.apply("Side2", points)
```

Rule key	Effect
`mirror_x`	Reflects x around `mirror_ref` (defaults to `max(x)`).
`invert_z`	Negates the z (or y for 2-column arrays) coordinate.
`offset_x/y/z`	Adds a scalar offset to the respective coordinate.

Table 3.9: *DataTransformer rule keys.*

Sides with no entry in `rules` are returned unchanged. Points must have shape `(N, 2)` or `(N, 3)`.

3.8 Procedures

Module path: `src/exp_process/procedures/`

Procedures are the named, configurable pipeline stages. They orchestrate core utilities into sequences: read input files, call core methods, write JSON output. Procedures carry knowledge of experiment types, file naming, and inter-step data layout. Core classes do not.

- **Interface contract:** every procedure receives a config dataclass and returns a `{id: Path}` dict — one entry per processed item pointing to its output file. An empty dict means nothing was processed, either because no input files were found or all items failed their data checks.
- **Side effects:** each procedure creates `output_dir` on instantiation via `BasePreprocessConfig.__pos` No other global state is modified.
- **Error strategy:** the procedures never raise on per-item failures; instead they print a `[SKIP]` message and continue. A crash-level exception only occurs when a core utility itself raises (e.g. corrupt JSON, numpy shape mismatch).

Pipeline execution order (both experiments share the same sequence):

`preprocess` → `run_validation` → `fit` → `compare_exp1` (Exp1 only)

Each step reads the output directory of the previous step as its input directory.

3.8.1 Configuration Dataclasses

File: `procedures/preprocess.py`

`BasePreprocessConfig` is the shared base for all configs. It holds `input_dir` and `output_dir` as `Path` objects and creates `output_dir` on instantiation via `__post_init__`. All other config classes inherit from it.

Config	Field	Default	Description
`Exp1PreprocessConfig`	`outlier_bottom`	`1.2`	IQR factor — bottom region.
	`outlier_top`	`1.2`	IQR factor — wall region.
	`outlier_general`	`1.5`	IQR factor for final merged cloud.
	`step_threshold`	`0.6`	Relative x-jump threshold (%) for step detection.
`Exp2PreprocessConfig`	`x_col`	`0`	Column index for X in raw data array.
	`z_col`	`1`	Column index for Z in raw data array.

Table 3.10: *Preprocessing config fields.*

Remark. *`outlier_bottom` and `outlier_top` are declared in `Exp1PreprocessConfig` but are not currently consumed by `preprocess_exp1`. The refactored parser already delivers a merged bottom+wall point cloud, so only one cleaning pass (with `outlier_general`) is applied. These fields are retained for potential future use — for example, if per-region cleaning needs to be reintroduced. Do not remove them without checking whether the per-region logic has been added back.*

File: `procedures/fitting.py`

Config	Field	Default	Description
`Exp1FittingConfig`	`high_degree`	`4`	Degree for the detailed 2D surface fit.
	`fix_rules`	`None`	Transformation rules for `DataTransformer`.
`Exp2FittingConfig`	`high_degree`	`2`	Degree for the detailed 1D curve fit.
	`normalize_x`	`True`	Normalize x to $[-1, 1]$ before fitting.
	`ridge_alpha`	`1.0`	Ridge regularization factor (λ).
	`fix_rules`	`None`	Transformation rules for `DataTransformer`.

Table 3.11: *Fitting config fields.*

Where to change — fitting parameters: the most commonly tuned values are `high_degree` and `ridge_alpha`.

- Increasing `high_degree` captures finer residual shape but risks overfitting; start conservative (4 for Exp1, 2 for Exp2) and increase only when the residual clearly shows structured curvature.
- `ridge_alpha` for Exp2 only: values above `1.0` smooth the fit; values below `0.1` behave like ordinary least squares. If the fitted curve oscillates, raise this value.
- `fix_rules` applies coordinate corrections via `DataTransformer` before fitting; see Section 3.7.7 for syntax.

3.8.2 preprocess_exp1

File: `procedures/preprocess.py`

Processes all sides found in `input_dir` through the following sequence:

1. `TShapeParser.load(side_id)` — loads and stacks bottom+wall measurements for all repetitions.
2. `np.vstack` — merges all repetitions into a single point cloud per side.
3. `OutlierCleaner.filter_iqr` — removes outliers on the z axis using `outlier_general`.
4. `StepSegmenter.find_steps` — splits the cloud into measurement steps.
5. `IOUtils.save_result` — writes `{side_id}_Steps.json`.

Output structure per side:

```
{
    "id": "Side1",
    "total_steps": 12,
    "steps": [
        {"step_number": 1, "point_count": 84, "points": [...]},
        ...
    ]
}
```

Where to change:

- **Too many outliers survive cleaning:** lower `outlier_general` (e.g. `1.0`). If legitimate geometry near the edges is also being removed, raise it back and instead address the source in the parser.
- **Steps are over-split or under-split:** adjust `step_threshold`. Lower values make the segmenter more sensitive to small x-jumps (more steps); higher values require larger gaps before a new step is declared.
- **Unexpected side IDs:** `TShapeParser.list_ids` controls discovery — see Section 3.6.2.

3.8.3 preprocess_exp2

File: `procedures/preprocess.py`

Processes all samples found in `input_dir`:

1. `RecShapeParser.load(sample_id)` — loads, reverses the right-side profile, and returns the L/R averaged point cloud.
2. `IOUtils.save_result` — writes `{sample_id}_Raw.json`.

No outlier removal or segmentation occurs here for Exp2. The raw merged profile is saved as-is for manual inspection via the GUI before fitting. The averaging logic lives entirely inside `RecShapeParser` — see Section 3.6.3.

Where to change:

- **Add outlier removal before saving:** insert an `OutlierCleaner.filter_iqr` call between the parser output and `IOUtils.save_result`. Add the corresponding IQR field to `Exp2PreprocessConfig`.
- **Raw data uses different column indices:** set `x_col` and `z_col` in the config to match the measurement file layout.

3.8.4 fit_exp1

File: `procedures/fitting.py`

For each `*_Steps.json` in `input_dir`:

1. Flattens all step points from all steps into a single array.
2. Applies `DataTransformer.apply` if `fix_rules` is set.
3. Fits a 2D polynomial of degree `high_degree` — the detailed surface model.
4. Fits a degree-1 baseline (`_LOW_DEGREE = 1`) — captures global tilt only.
5. Subtracts baseline from the high-degree model via `ModelOps.subtract_coeffs`, yielding the tilt-corrected residual.
6. Writes `{side_id}.json` with the flattened model coefficients and the transformed point cloud.

The constant `_LOW_DEGREE = 1` is defined at module level in `fitting.py` and applies to both Exp1 and Exp2. Changing it affects all fitting procedures simultaneously.

Where to change:

- **Higher-order tilt correction:** change `_LOW_DEGREE` at the top of `fitting.py` to `2` or `3` if the measurement platform has a bowl-shaped drift rather than a simple tilt.
- **Coordinate corrections:** pass a `fix_rules` dict to the config. This is the correct place to flip axes or apply offsets — not inside the parser.
- **Output contains only the model, not the raw points:** the points stored are *after* `DataTransformer` is applied. If you need the untransformed points separately, add a raw-save step before the transformer call.

3.8.5 fit_exp2

File: `procedures/fitting.py`

For each `*_Raw.json` in `input_dir`:

1. Applies `DataTransformer.apply` if `fix_rules` is set.
2. Fits a 1D polynomial of degree `high_degree` via `Fitter.fit_1d_poly`, with optional x-normalization and Ridge regularization.
3. Fits a degree-1 baseline (`_LOW_DEGREE`) and subtracts it via `ModelOps.subtract_coeffs`.
4. Writes `{sample_id}.json` with the flattened model and the transformed point cloud.

Where to change:

- **Oscillating fit:** raise `ridge_alpha` (e.g. `5.0` or `10.0`). If the curve is clearly nonlinear but the fit is too flat, lower `ridge_alpha` and/or increase `high_degree` to 3.
- **Disabling normalization:** set `normalize_x = False` when the x-range of the data is already well-conditioned (e.g. values close to 0–1). With large x-ranges (e.g. millimetre data), keep `True` to avoid ill-conditioned polynomial matrices.

3.8.6 compare_exp1

File: `procedures/comparison.py`

Loads `Side1.json` and `Side2.json` from `input_dir`, computes the coefficient-wise mean via `ModelOps.average_models`, and writes `Average.json`. This is the final step of the Exp1 pipeline, producing a single reference surface from both measurement sides.

Expected input files are hard-coded by name as `Side1.json` and `Side2.json`. If either file is missing, the function returns an empty dict and prints an error — no partial average is written.

Where to change:

- **More than two sides:** replace the two hard-coded `load_json` calls with a loop over `glob("*.json")` (excluding `Average.json`) and pass the resulting list to `ModelOps.average_models`. The averaging logic itself already supports arbitrary list lengths.
- **Weighted average:** `ModelOps.average_models` currently computes a simple mean. To weight by point count, modify that method in `core/operations.py` — the procedure does not need to change.

3.8.7 run_validation

File: `procedures/validation.py`

Launches the `PointCloudViewer` GUI in a blocking Tkinter mainloop. Execution resumes only after the user closes the window. Intended to be called between preprocessing and fitting in both pipelines, allowing the researcher to inspect and manually edit individual step or raw-curve files before fitting locks in the data.

```
from exp_process.procedures.validation import run_validation
# Point at the preprocessing output directory
run_validation(output_dir="data/processed/exp1")
```

The function handles window close gracefully: it calls `plt.close("all")` before destroying the Tkinter root to prevent matplotlib backends from leaving orphaned figure threads.

Where to change:

- **Run without blocking:** `run_validation` always blocks because it calls `root.mainloop()`. If non-blocking behaviour is needed (e.g. automated testing), instantiate `PointCloudViewer` directly and manage the event loop manually.
- **Auto-loading a specific file:** pass a specific JSON filename to `PointCloudViewer` via its constructor arguments — see Section 3.10.

3.8.8 Adding a New Procedure

To add a procedure for a new experiment type:

1. Create a config dataclass inheriting from `BasePreprocessConfig` in `preprocess.py` (or a new file). Add only the fields specific to that experiment.
2. Write the procedure function. Follow the contract: accept the config, return `{id: Path}`.
3. Import and expose it in `procedures/__init__.py`.
4. Wire it into the pipeline script (`pipeline/base.py` or a dedicated pipeline class).

Do not put file I/O logic directly into core modules when adding a procedure. The division is intentional: core modules are stateless utilities; procedures own the file system layout.

3.9 Pipeline

Module path: `src/exp_process/pipeline/`

The pipeline layer is the single entry point for running the full processing sequence. It composes the procedure functions from Section 3.8 into an ordered, experiment-specific workflow, and exposes one method — `run()` — as the public interface.

The separation of responsibilities is strict:

- **Core** (Section 3.7) — stateless algorithms, no file I/O.
- **Procedures** (Section 3.8) — named steps, each reads and writes files.
- **Pipeline** — assembles steps into a sequence and manages data directories between them.

A pipeline object never calls core classes directly; it only calls procedures. This means if the internal algorithm of a step changes, only the relevant core module and possibly the procedure need to change — the pipeline stays untouched.

3.9.1 BasePipeline

File: `pipeline/base.py`

`BasePipeline` is an abstract base class (`ABC`) that defines the execution contract for all experiment pipelines. It implements one concrete method — `run()` — and declares three abstract methods that subclasses must implement.

Execution sequence in `run()`:

1. **STEP 1 — Preprocess:** calls `self._preprocess()`, which returns the output directory path for the next step.
2. **STEP 2 — Validation (GUI):** passes that directory to `run_validation()`, blocking until the user closes the viewer.
3. **STEP 3 — Fit:** calls `self._fit()`, which reads from the (now manually validated) preprocessing output.
4. **STEP 4 — Compare:** calls `self._compare()`, which defaults to a no-op; only `SurfacePipeline` overrides it.

Each step prints a labelled header (`=== STEP N ===`) to stdout so progress is visible during a run.

Abstract interface:

Method	Returns	Description
`_preprocess()`	`str`	Run preprocessing; return output directory path.
`_fit()`	`None`	Run fitting on preprocessing output.
`_compare()`	`None`	Optional post-fit comparison step. Base no-op.

Table 3.12: *Abstract methods of `BasePipeline`.*

Where to change:

- **Skip validation in automated runs:** the GUI call is hard-coded in `run()`. To skip it, override `run()` in the subclass and call the steps directly without `run_validation`. This is the recommended approach for batch or CI runs.
- **Add a new stage between fit and compare:** add a `_post_fit()` abstract method to `BasePipeline`, call it inside `run()` after `_fit()`, and provide a default no-op implementation. Subclasses can then override it as needed.

3.9.2 SurfacePipeline — Experiment 1

File: `pipeline/surface.py`

`SurfacePipeline` implements the full 4-step sequence for Exp1 (T-Shape, 3D surface). It uses `Exp1PipelineConfig` as its single constructor argument.

Config structure:

```
from exp_process.pipeline.surface import SurfacePipeline, Exp1PipelineConfig
from exp_process.procedures.preprocess import Exp1PreprocessConfig
from exp_process.procedures.fitting import Exp1FittingConfig
from exp_process.procedures.preprocess import BasePreprocessConfig

cfg = Exp1PipelineConfig(
    preprocess=Exp1PreprocessConfig(
        input_dir="data/raw/exp1",
        output_dir="data/processed/exp1/preprocess",
    ),
```

```
    fitting=Exp1FittingConfig(
        input_dir="data/processed/exp1/preprocess",
        output_dir="data/processed/exp1/fitting",
        high_degree=4,
    ),
    comparison=BasePreprocessConfig(
        input_dir="data/processed/exp1/fitting",
        output_dir="data/processed/exp1/comparison",
    ),
)

SurfacePipeline(cfg).run()
```

Config field	Type	Purpose
`preprocess`	`Exp1PreprocessConfig`	Parser, cleaning and segmentation settings.
`fitting`	`Exp1FittingConfig`	Polynomial degree and transform rules.
`comparison`	`BasePreprocessConfig`	Directories for Side1/Side2 averaging.

Table 3.13: *`Exp1PipelineConfig` fields.*

Note that `fitting.input_dir` must point to the *same* directory as `preprocess.output_dir`, and `comparison.input_dir` must match `fitting.output_dir`. These connections are not enforced automatically — the paths must be consistent in the config.

Where to change:

- **Paths are mismatched between steps:** ensure `preprocess.output_dir`, `fitting.input_dir`, `fitting.output_dir`, and `comparison.input_dir` form an unbroken chain.
- **Skip the comparison step:** override `_compare()` in a subclass and make it a no-op, or simply do not use the comparison output in downstream scripts.

3.9.3 CurvePipeline — Experiment 2

File: `pipeline/curve.py`

`CurvePipeline` implements the 3-step sequence for Exp2 (rectangular, 1D curve). The comparison step is not present — Exp2 has no two-side averaging. The `_compare()` method inherits the no-op from `BasePipeline`.

Config structure:

```
from exp_process.pipeline.curve import CurvePipeline, Exp2PipelineConfig
from exp_process.procedures.preprocess import Exp2PreprocessConfig
from exp_process.procedures.fitting import Exp2FittingConfig

cfg = Exp2PipelineConfig(
    preprocess=Exp2PreprocessConfig(
        input_dir="data/raw/exp2",
```

```
        output_dir="data/processed/exp2/preprocess",
    ),
    fitting=Exp2FittingConfig(
        input_dir="data/processed/exp2/preprocess",
        output_dir="data/processed/exp2/fitting",
        high_degree=2,
        ridge_alpha=1.0,
    ),
)

CurvePipeline(cfg).run()
```

Config field	Type	Purpose
`preprocess`	`Exp2PreprocessConfig`	Parser and column index settings.
`fitting`	`Exp2FittingConfig`	Polynomial degree, normalization, Ridge λ.

Table 3.14: *`Exp2PipelineConfig` fields.*

3.9.4 Adding a New Pipeline

To support a new experiment type:

1. Create a new file `pipeline/my_experiment.py`.
2. Define a `@dataclass` config aggregating the necessary procedure configs.
3. Subclass `BasePipeline`, implement `_preprocess()` and `_fit()`, and optionally `_compare()`.
4. Return the preprocessing output directory as a `str` from `_preprocess()` so the validation step receives the correct path.
5. Expose the new class in `pipeline/__init__.py`.

The minimum skeleton:

```
# pipeline/my_experiment.py
from .base import BasePipeline
from ..procedures.preprocess import MyPreprocessConfig, preprocess_my
from ..procedures.fitting import MyFittingConfig, fit_my

@dataclass
class MyPipelineConfig:
    preprocess: MyPreprocessConfig
    fitting: MyFittingConfig

class MyPipeline(BasePipeline):
    def __init__(self, cfg: MyPipelineConfig):
```

```
        self.cfg = cfg

    def _preprocess(self) -> str:
        preprocess_my(self.cfg.preprocess)
        return str(self.cfg.preprocess.output_dir)

    def _fit(self) -> None:
        fit_my(self.cfg.fitting)
```

3.10 Utilities and GUI

This section covers the two support modules of `exp_process`: the I/O utilities in `utils/io.py` and the interactive point cloud viewer in `gui/viewer.py`. Neither module contains domain logic — they provide the plumbing that all other layers rely on.

3.10.1 IOUtils — `utils/io.py`

File: `utils/io.py`

All JSON read and write operations in the module go through `IOUtils`. The class exists to centralise two concerns: handling NumPy types during serialisation, and providing a consistent file-naming convention for pipeline output.

NumpyEncoder

A custom `json.JSONEncoder` subclass that transparently converts NumPy scalars and arrays to native Python types before serialisation:

NumPy type	Serialised as
`np.integer`	`int`
`np.floating`	`float`
`np.ndarray`	`list` (via `.tolist()`)

Table 3.15: *`NumpyEncoder` type mapping.*

This encoder is used automatically by all `save_json` calls. It does not need to be imported or called directly — it is an implementation detail of the I/O layer.

Module-level vs. class-level functions

`io.py` exposes both module-level functions (`save_json`, `load_json`) and the same methods as static methods on `IOUtils`. The only behavioural difference is in `load_json`:

- Module-level `load_json(filepath)` — raises `FileNotFoundError` if the file does not exist.
- `IOUtils.load_json(filepath)` — returns `None` if the file does not exist (safe for pipeline use).

All procedure modules use `IOUtils` (the class), so missing files produce a `[SKIP]` rather than a crash. The module-level functions are kept for standalone scripts or external callers that prefer an exception.

IOUtils.save_result

The primary write method used by all procedures:

```
IOUtils.save_result(output_dir: Path, name: str, data: dict) -> Path
```

Constructs the output path as `output_dir / name.json`, serialises `data` with `NumpyEncoder`, creates `output_dir` if it does not exist, and returns the resolved `Path`. The `name` argument is always the item identifier without extension: `"Side1"`, `"Side1_Steps"`, `"Average"`, etc.

Where to change:

- **Different output format (e.g. CSV, HDF5):** replace `IOUtils.save_result` calls in the relevant procedure. The core and pipeline layers do not call `save_result` directly, so the change is isolated to the procedures.
- **Custom file naming convention:** the `name` argument is controlled by each procedure. Change it there — `IOUtils` does not impose a naming scheme beyond the `.json` extension.
- **Adding logging:** `save_json` already calls `logger.info` on success. To change the log level or destination, configure the `logging` module at the application entry point; no changes to `io.py` are needed.

3.10.2 PointCloudViewer — `gui/viewer.py`

File: `gui/viewer.py`

`PointCloudViewer` is a Tkinter + Matplotlib GUI for manually inspecting and editing point cloud JSON files between the preprocessing and fitting steps. It is launched exclusively through `run_validation()` (Section 3.8.7).

Layout

The window is split into two panels:

- **Left panel** — Matplotlib canvas with the full navigation toolbar (zoom, pan, save figure). Displays the currently selected step or flat point cloud, with a polynomial model overlay when coefficients are present in the file.
- **Right panel** — file/sample selector (dropdown), step list (listbox), instructions label, *Save Changes* button, and an info label showing point count.

Data types

When loading files from `input_dir`, the viewer classifies each JSON file into one of three types based on its structure:

Type	Detected when	Example files
`steps`	top-level `"steps"` key is a list	`Side1_Steps.json`
`flat`	top-level `"points"` key is a list	`01_Raw.json`
`model`	neither key present	`Side1.json`, `Average.json`

Table 3.16: *File type classification in `PointCloudViewer`.*

`model` files are loaded and shown in the dropdown but are not editable — they contain coefficient data, not raw points. The viewer overlays the fitted polynomial curve on the plot when it reads a `"coeffs"` key in the currently displayed file.

On startup, the viewer auto-selects the first `steps` file found; if none exist, it selects the first file alphabetically.

Interactive editing

The core editing interaction is click-to-delete:

1. Select a file from the dropdown.
2. Select a step from the listbox (or *All Points* for flat files).
3. Click on a point in the canvas to remove it. The point nearest to the click within a picker tolerance of 5 pixels is deleted.
4. After editing, click *Save Changes* to overwrite the JSON file on disk. The `modified` flag is set on any deletion and cleared on save.

Deletions modify `data_store` in memory. Nothing is written to disk until *Save Changes* is clicked. Closing the window without saving discards all deletions.

Model overlay

When the selected file contains a `"coeffs"` key (i.e. a fitted model), the viewer renders the polynomial curve over the point cloud. The overlay is reconstructed each time `update_plot()` is called from the stored axes limits, so it remains valid after point deletions change the scale.

This also means that if you open a `model` file (e.g. `Side1.json` after fitting), you can visually verify the fit quality without running a separate script.

Where to change:

- **Picker tolerance too tight or too loose:** the value `picker=5` is hard-coded in `_setup_figure()`. Increase it if clicks are not registering; decrease it if adjacent points are accidentally deleted.
- **Plot axes labels:** currently fixed to `"Y (mm)"` and `"Z (mm)"` in `_setup_figure()` and `update_plot()`. If the data coordinate system changes, update both locations.
- **Add undo support:** the viewer has no undo. To add it, keep a stack of point arrays in `update_current_points()` before overwriting, and add an *Undo* button that pops the stack and redraws.

- **View 3D surface data (Exp1):** the viewer currently projects onto a 2D Y/Z plane. To visualize the full 3D point cloud, replace `fig.add_subplot(111)` with `fig.add_subplot(111, projection='3d')` and adjust the scatter call — note this requires updating both `_setup_figure()` and `update_plot()`.

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

Files: `odb_npz_para.py`, `deps.py`

4.1.1 Module Restructuring

The conversor module was reorganised and all scripts renamed to follow a consistent lowercase-underscore convention:

Old name	New name
`Odb_Npz_Parameters.py`	`odb_npz_para.py`
`Odb_Npz_Converter.py`	`odb_npz_conv.py`
`Npy_2_Xdmf.py`	`npy_hdf_conv.py`
— (new)	`deps.py`

Table 4.1: *Conversor module file renaming.*

4.1.2 deps.py --- Centralised Imports

All imports for the conversor package are now consolidated in `deps.py`. This solves two problems from the old code: duplicate import blocks across files, and IDE false-positive errors from Abaqus-only symbols.

The file handles Python 2/3 compatibility explicitly:

- `pathlib.Path` is imported with a `try/except` — if the import fails (Python 2.7 Abaqus environment), `Path` is set to `None` so the rest of the code still runs.
- `scipy` and `h5py` are guarded the same way — their absence does not crash the Abaqus side, which does not need them.
- NumPy warnings about files created on Python 2 are suppressed globally here, not in each consumer module.

Where to change: add any new third-party dependency import here, with the same `try/except` guard if it is Python-3-only.

4.1.3 ODB2NPYParameters

`ODB2NPYParameters` loads `data/config.json` and returns the ODB simulation directory, the NPY output directory, and the conversion parameters dict consumed by `OdbBatchConverter`. Responsibility is unchanged from the previous version.

A diagnostic block was added to `run()`: it logs Python version, current directory, root directory, and encoding before returning. This is useful when the script runs from Abaqus where the environment can differ from the development machine.

4.1.4 OdbBatchConverter

Finds all `.odb` files in `Simulation_dir`, inspects their structure (steps, instances, frame counts), and calls `OdbToNPYConverter` for each one. One subdirectory per ODB file is created inside `NPY_Dir`.

A new helper `_convert_frame_params(begin_frame, end_frame)` was added to convert legacy `-1` sentinel values to the format expected by the new converter, keeping backward compatibility with existing configs.

Where to change:

- **Process only specific ODB files:** modify `find_odb_files` or add a name filter before the loop in `convert_batch`.
- **Change mesh type:** set `MeshType` in `config.json` under `conversion_params/Contour_Method` — `"12"` for Hexahedron, `"10"` for Tetrahedron.

4.2 Stage 2: ODB Data Extraction (Abaqus API)

Script: `odb_npz_conv.py` (runs in **Python 2.7** inside the Abaqus environment)

This script must be executed through the Abaqus Python kernel; it cannot run in a standard Python 3 environment because it imports Abaqus-internal symbols (`openOdb`, `ELEMENT_NODAL`, etc.). Its responsibility is to extract mesh and field data from an ODB and save individual `.npy` files to disk.

Parameter	Default	Description
`begin_frame`	`'-1'`	First frame to extract. `-1` = first frame of each step.
`end_frame`	`'-1'`	Last frame to extract. `-1` = last frame of each step.

Table 4.2: *New parameters in `OdbToNPYConverter`.*

4.2.1 New Constructor Parameters

Two parameters were added to `OdbToNPYConverter` that did not exist before:

This allows extracting only a window of frames instead of the full time history — useful when only the final loading state is needed. The method `_get_frame_range` translates the `-1` sentinel to the actual last index and clamps both values to the available range.

4.2.2 Geometry Extraction and Immediate Memory Release

Node mapping works as before: Abaqus labels are translated to sequential zero-based indices via a per-instance `node_mapping` dict. The key change is in memory management.

Geometry arrays (`coordinates`, `connectivity`, `element_types`, `offsets`) are now saved to disk immediately after extraction via `_save_geometry_topology`, and then explicitly deleted from `geom_data` with a `gc.collect()` call. This frees memory before the (much larger) temporal data loop begins, which is critical for high-element-count models.

4.2.3 Dual-Strategy Field Extraction

Both displacement and stress extraction now use a fallback chain of position strategies rather than a single fixed position. This was the source of silent empty arrays in the old code when Abaqus stored results at a different integration level.

Displacement — tries in order:

1. `NODAL`
2. `ELEMENT_NODAL`
3. `INTEGRATION_POINT`
4. Default (no position filter)

Stress — tries in order:

1. `ELEMENT_NODAL` (extrapolated from integration points to nodes — preferred)
2. `NODAL`
3. Default via `bulkDataBlocks` (fastest, block-level access)

For each position, the extractor first attempts `bulkDataBlocks` (faster), then falls back to iterating `field.values` individually. A warning is printed if no strategy produces data.

4.2.4 Stress Components Saved

The refactored extractor now saves all nine tensor components plus the Abaqus-computed invariants:

- `stress_tensor.npy` — shape `[n_nodes, 9]`, full symmetric tensor (S11, S22, S33, S12, S13, S23 mapped to all 9 positions)
- `von_mises.npy` — shape `[n_nodes]`
- `max_principal.npy` — shape `[n_nodes]`
- `min_principal.npy` — shape `[n_nodes]`

The invariants are read directly from Abaqus (`v.mises`, `v.maxPrincipal`, `v.minPrincipal`) rather than recomputed. This ensures consistency with the Abaqus Viewer. After averaging across all contributing values per node, the `stress_threshold` filter (default 10^{-6}) zeros out all fields at nodes below the threshold to suppress numerical noise.

Where to change:

- **Extract only the last frame:** leave both `begin_frame` and `end_frame` at `'-1'` (default behaviour).
- **Suppress noisy near-zero stress regions:** raise `stress_threshold` (e.g. `1e-3`).
- **Add a new field output (e.g. strain):** follow the same dual-strategy pattern in `_process_and_save_frame` — allocate a sum array, cache the field, accumulate and average across contributing values, apply the threshold, then save with `np.save`.

4.3 Stage 3: XDMF/HDF5 Compilation

Script: `npy_hdf_conv.py` (runs in **Python 3**)

This stage has no Abaqus dependency and runs in a standard Python 3 environment. It reads the `.npy` directories produced by Stage 2 and consolidates them into a single HDF5 file plus an XDMF wrapper for ParaView.

4.3.1 NPY2XDMFParameters --- New Separate Config Class

A new `NPY2XDMFParameters` class was added (separate from `ODB2NPYParameters` in Stage 1). It reads `data/config.json` and returns the NPY root directory, the output directory, and the options dict for `NpyBatchToXdmfConverter`. The split avoids importing any Abaqus-side code into the Python 3 environment.

4.3.2 Memory-Mapped NPY Loading

All `.npy` files are now loaded via `np.load(path, mmap_mode='r')` (internal method `_np_load`). Memory mapping means the full array is not loaded into RAM upfront; the OS pages in only the accessed regions. For large mesh models this is the primary reason Stage 3 no longer runs out of memory when processing multiple simulations.

4.3.3 HDF5 Structure and Precision

The HDF5 hierarchy is unchanged. Precision is now explicit and documented:

```
S_batch.h5
  ModelName/
    geometry/
      coordinates      [n_nodes, 3]              float64
      connectivity     [n_elem, nodes_per_elem]   int32
    topology/
      element_types    [n_elements]               uint8
      offsets          [n_elements]               int32
    time_series/
      step_1_.../
        frame_001/
          displacement      [n_nodes, 3]    float32
          stress_tensor     [n_nodes, 9]    float32
          von_mises         [n_nodes]       float32
          max_principal     [n_nodes]       float32
          min_principal     [n_nodes]       float32
```

- **Coordinates** are written as `float64` (8 bytes) to preserve mesh geometry accuracy.
- **All field data** is written as `float32` (4 bytes), halving storage size with negligible loss for visualisation.

The `nodes_per_elem` count is inferred from the connectivity array dimensions (`conn.size // element_types.size`) rather than reading a separate metadata file.

4.3.4 XDMF Attribute Type Detection

The XDMF writer now auto-detects the attribute type of each dataset, instead of hard-coding `Scalar`:

Array shape	XDMF AttributeType	Example
`[N, 3]`	`Vector`	displacement
`[N, 9]`	`Tensor6`	stress_tensor
`[N]`	`Scalar`	von_mises, principal stresses

Table 4.3: *XDMF attribute type auto-detection.*

`Tensor6` is the correct type for a symmetric 9-component stress tensor in ParaView. Using `Scalar` for this field (as the old code did) prevented ParaView from computing derived quantities like principal stress on its own.

Where to change:

- **Disable gzip compression:** set `"hdf5_compression": false` in `config.json` under `conversion_params/NPY2XDMF`. Useful when write speed matters more than file size.
- **Change output filename:** `"S_batch.h5"` is passed as `h5_filename` in `NpyBatchToXdmfConverter.co` Change it there.
- **Add a new field to XDMF:** no change needed here — the writer iterates all datasets found in each `frame_XXX` group automatically. Ensure Stage 2 saves the new `.npy` file in the correct frame directory and the attribute type table above will handle it.

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`

Module path: `src/element_process/`

The module was restructured and all scripts renamed to the lowercase-underscore convention:

Old name	New name
`Elements_main.py`	`elements_main.py`
`s1_Ele_Extractor.py`	`extractor.py`
`s2_RE_Field.py`	`field_analitic.py`
`s3_RE_Interpolator.py`	`interpolator.py`
`s2_RE_ExnCon.py`	`stress_mapping.py`
`s2_RE_ExnCon2.py`	`stress_mapping_2.py`
— (new)	`elements_plot.py`

Table 5.1: *`element_process` module file renaming.*

5.1.1 Analytical Pipeline

Used for validation or when experimental data is unavailable. Runs three stages in sequence:

1. **Extraction:** Parses geometry and topology from the `.inp` file via `extractor.py`.
2. **Synthetic Generation:** Generates a theoretical cylindrical stress field from the mesh bounding box via `field_analitic.py`.
3. **Interpolation:** Maps the synthetic point cloud onto element centroids via `interpolator.py`.

5.1.2 Data-Driven Pipeline

For real simulation data (Contour Method or RSA), the generation step is replaced by direct HDF5 mapping:

1. **Extraction:** Same as above; prepares target mesh centroids.
2. **Direct Mapping:** Reads the HDF5 source (`S_batch.h5`) and maps stresses to element centroids using KDTree via `stress_mapping.py` or `stress_mapping_2.py`.

5.2 Stage 1: Element Extraction

Script: `extractor.py`

Parses an Abaqus `.inp` file and computes element centroids, producing two output files consumed by all downstream stages.

5.2.1 Parsing Functions

- `extract_elements_from_inp(input_file)` — scans for the `*ELEMENT` keyword and returns three arrays: element IDs, element types (e.g. `C3D8R`), and the list of connected node IDs per element.
- `extract_node_coordinates(input_file)` — scans for the `*NODE` keyword and returns a `dict` mapping each node ID to its (x, y, z) coordinates, used for centroid calculation.
- `get_element_coordinates(connected_nodes, node_coords)` — averages the coordinates of connected nodes to produce the centroid $(X_{center}, Y_{center}, Z_{center})$ for each element:

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

Nodes missing from `node_coords` are skipped; elements with no valid nodes receive `NaN` centroids.

5.2.2 Output Files

`save_element_info` writes two files to `output_dir`:

`elements_data.txt` is the primary input for `interpolator.py` and `stress_mapping.py`. `element_info.txt` is used by `field_analitic.py` to read the mesh bounding box for synthetic stress generation.

File	Format	Contents
`elements_data.txt`	Tab-separated	Element ID, Type, $X/Y/Z$ centroid per row.
`element_info.txt`	Plain text	Total element count, coordinate ranges, type distribution.

Table 5.2: *Output files from* `extractor.py`.

5.3 Stage 2: Analytical Stress Field Generation

Script: `field_analitic.py`

Generates a synthetic cylindrical residual stress field for validation or benchmarking when real measurement data is unavailable. Output is `residual_stress.txt`, read by `interpolator.py` in the analytical pipeline.

5.3.1 CoordinateLimits Dataclass

A `@dataclass` was introduced to encapsulate the mesh bounding box and total node count. It replaces the previous loose variable passing:

- `CoordinateLimits.from_info_file(path)` — reads `element_info.txt` produced by `extractor.py` and populates the limits via regex. Falls back to `default_values()` if parsing fails.
- `CoordinateLimits.default_values()` — returns hardcoded limits matching the reference mesh geometry, used when no info file is found.

5.3.2 Homogeneous Mesh Generation

`generate_homogeneous_mesh(limits, target_density=15)` creates a regular grid of points filling the bounding box. The number of divisions per axis is scaled proportionally to the axis range so that node spacing is uniform across all three dimensions:

$$n_x = \max\left(2,\ \left\lfloor t \cdot \frac{x_{max} - x_{min}}{\Delta_{min}} \right\rfloor\right) \tag{5.2}$$

where t is `target_density` and Δ_{min} is the smallest of the three axis ranges.

5.3.3 Stress Calculation

`calculate_cylindrical_stress(nodes, center)` computes cylindrical coordinates (r, θ) for each point relative to the geometric center, then assigns stress components as polynomial functions of the normalised 3D distance:

$$\sigma_r = 5000 \cdot \hat{d}^2$$
$$\sigma_\theta = 5000 \cdot \hat{d}$$
$$\sigma_z = 5000 \cdot \hat{d}^{1.5}$$

where $\hat{d} = d/d_{max}$. The output file stores all six stress components (including shear terms $\tau_{rt}, \tau_{rz}, \tau_{\theta z}$) in cylindrical coordinates. `interpolator.py` converts these to Cartesian before interpolation.

5.4 Stage 3: Field Interpolation

Script: `interpolator.py`

Maps a source stress field (analytical or Abaqus-format) onto the target finite element mesh by interpolating at each element centroid.

5.4.1 Input Loading

`ElementTensionInterpolator` reads both the target mesh and the source stress field from `output_dir`. File detection is automatic: each load method first looks for a standard filename (`elements_data.txt`, `residual_stress.txt`) and, if not found, scans the directory for a suitable candidate.

`load_tension_field` supports two input formats detected automatically from the file header:

Format	Detected by	Columns
Abaqus	header contains `INITIAL CONDITIONS`	ID, S11, S22, S33, S12, S13, S23
Cylindrical	header contains `Sigma_r`	ID, X, Y, Z, R, θ, σ_r, σ_θ, σ_z, τ_{rt}, τ_{rz}, $\tau_{\theta z}$

Table 5.3: *Source stress file formats accepted by `load_tension_field`.*

For cylindrical format, the components are converted to Cartesian in-place before interpolation:

$$S_{11} = \sigma_r \cos^2\theta + \sigma_\theta \sin^2\theta$$
$$S_{12} = (\sigma_r - \sigma_\theta) \sin\theta \cos\theta$$

5.4.2 Interpolation Strategy

For each of the six stress components independently:

1. **Linear (primary):** `LinearNDInterpolator` over the source point cloud in 3D.
2. **Nearest-neighbor (fallback for NaN):** `NearestNDInterpolator` fills elements outside the convex hull of the source data.
3. **Mean (last resort):** if both interpolators fail, the component is filled with the mean value across all source points.

If fewer than 4 source points are available (minimum for 3D linear interpolation), all elements receive the mean stress vector directly.

5.4.3 Output

`generate_interpolated_tension_file` writes `interpolated_element_stresses.txt` formatted as an Abaqus initial condition:

```
*INITIAL CONDITIONS, TYPE=STRESS
element_id, S11, S22, S33, S12, S13, S23
...
```

5.5 Data-Driven Stress Mapping

Script: `stress_mapping.py`

Maps S_{33} stress from the `S_batch.h5` HDF5 source onto the target finite element mesh using a planar KDTree strategy. Used in the data-driven pipeline (Contour Method results).

5.5.1 Constructor Parameters

Parameter	Default	Description
`base_dir`	—	Directory containing `.inp` and `Output/` folders.
`tolerance`	`5e-2`	$\|z\|$ threshold for the z=0 plane extraction.
`chunk_size`	`10000`	Batch size for HDF5 read operations on large datasets.

Table 5.4: *`StressProcessor` constructor parameters.*

5.5.2 Mapping Strategy

The workflow uses a 2D KDTree in the x-y plane rather than a full 3D tree. This is intentional: the Contour Method extracts a cut-plane measurement at $z = 0$, which is then projected onto all z-planes of the mesh.

1. `read_combined_hdf5_from_folder` — reads `S_batch.h5` and extracts coordinates and `stress_tensor[:, 8]` (S_{33}, index 8 in the 9-component tensor) for the first available step/frame.
2. `extract_z0_data_by_z` — filters HDF5 nodes where $|z| <$ `tolerance` using vectorised `np.isclose`, returning the z=0 slice.
3. `create_stress_mapping_by_z` — for each unique Z plane in the mesh, builds a KDTree on (x, y) of the z=0 source data and queries the centroid coordinates of all elements in that plane. KDTrees are cached by array hash to avoid rebuilding for repeated queries.

5.5.3 Output Formats

`save_organized_data` and `create_abaqus_stress_file` produce three output files per simulation:

File	Format
`stress_mapping_by_z.json`	Full mapping with element coordinates and distances.
`stress_mapping_by_z.h5`	Same structure in HDF5 for efficient downstream reads.
`stress_input.txt`	Abaqus-format: `element_id, 0, 0, S33, 0, 0, 0` per line.

Table 5.5: *Output files from `stress_mapping.py`.*

5.5.4 Usage

```
from element_process.stress_mapping import StressProcessor

processor = StressProcessor(
    base_dir="C:/Simulations/Residual_Stresses_Analysis",
    tolerance=5e-2,
    chunk_size=10000
)
results = processor.process_all_simulations(
    hdf5_folder="C:/Simulations/Contour_Method/xdmf_hdf5_files"
)
```

5.6 Batch Processing Variant

Script: `stress_mapping_2.py`

Extends `StressProcessor` to handle directory structures where multiple load cases (`S1`, `S2`, ...) share the same mesh geometry. Typical use case: RSA sensitivity analysis batches.

5.6.1 Inheritance

`StressProcessorBatch` subclasses `StressProcessor` via a relative import:

```
from .stress_mapping import StressProcessor

class StressProcessorBatch(StressProcessor):
    # Overrides: find_simulations, process_specific_simulation,
    #            process_all_simulations
    # Inherits:  all HDF5 reading, KDTree mapping, and output methods
```

All mapping logic (HDF5 extraction, KDTree, output file generation) is inherited unchanged. Only discovery and per-case orchestration are overridden.

5.6.2 Discovery Logic

`find_simulations` searches for two filename patterns:

Pattern	Example
`Mesh-*--Length-*.inp`	`Mesh-0_6--Length-50.inp`
`S*_Mesh-*--Length-*.inp`	`S1_Mesh-0_6--Length-50.inp`

Table 5.6: *File patterns recognised by `StressProcessorBatch.find_simulations`.*

The `SN_` prefix is stripped via regex (`r"^S\d+_(.+)$"`) to obtain the normalised mesh name used as the dictionary key. Both naming conventions resolve to the same base mesh and share the same `elements_data.txt`.

5.6.3 Per-Case Output

`process_specific_simulation` iterates all S* entries found in the HDF5 file for a given mesh base and writes a dedicated subdirectory per case:

```
Output/
  Mesh-0_6--Length-50/
    S1_Mesh-0_6--Length-50/
      stress_mapping_by_z.json
      stress_mapping_by_z.h5
      stress_input.txt
    S2_Mesh-0_6--Length-50/
      ...
```

5.6.4 Usage

```
from element_process.stress_mapping_2 import StressProcessorBatch

processor = StressProcessorBatch(
    base_dir="C:/Simulations/Residual_Stresses_Analysis"
)
results = processor.process_all_simulations(
    hdf5_folder="C:/Simulations/Contour_Method/xdmf_hdf5_files"
)
```

Part I

Simulations

6
Core INP Manipulation Library

Module Path: `src/simulations/inp_process/`

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 (`abaqus_cmd`) and scratch directory usage (`use_scratch`).
- **Job Control:** Timeout limits (default 30 hours) to prevent stalled processes from hanging the pipeline indefinitely.
- **Output Mode:** `silent_mode` — if `True`, captures stdout/stderr to a log file; if `False`, streams output live. `auto_cleanup` — if `True`, automatically removes the scratch directory after a successful run.

Class: `AbaqusScriptConfig`

A companion dataclass for running standalone Abaqus Python scripts (not full solver jobs). Used by `AbaqusScriptRunner`:

- `script_name` — path to the `.py` script to execute.
- `working_dir` — working directory; UNC paths (`\\server\share`) are handled transparently via `pushd/popd`.
- `python_cmd` — Python interpreter or Abaqus Python wrapper command.
- `env` — optional dictionary of environment variables passed to the subprocess.

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:** Finds a material definition by name and replaces its entire sub-block while preserving the rest of the file. End-of-block detection is done by scanning for a predefined list of "block breaker" keywords (*Step, *Boundary, *Node, *Element, etc.).
- **insert_in_step:** Inserts arbitrary lines (e.g., boundary conditions) immediately before *End Step of a *named* step. Useful for adding per-step output requests or displacement controls after the rest of the file has been assembled.
- **fix_restart_frequency:** Scans every line and replaces frequency=0 with frequency=1 in *Restart and *Output blocks. Abaqus ignores output requests with zero frequency, and this often introduces silent errors when input files are exported from the GUI.

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 Abaqus Script Runner

Class: `AbaqusScriptRunner`

Executes standalone Abaqus Python scripts (as opposed to full solver jobs). Key detail: it handles **UNC paths** (e.g., `\\server\share`) transparently by wrapping the call in `pushd/popd`, which maps the UNC path to a temporary drive letter before execution and unmaps it afterwards.

6.7.3 Batch Orchestration

Class: `INPRunner`

Manages the execution of multiple input files in sequence.

- **Discovery:** `find_inp_files(pattern)` scans `base_dir` recursively using a glob pattern. Default pattern is `**/*_FI.inp`.
- **Success Detection:** A job is considered successful only if the return code is `0` *and* the corresponding `.odb` file was produced in the output directory. A zero return code without an ODB file is treated as a failure.
- **Reporting:** After the batch, a `summary_report.txt` is written to `simulation_logs/` containing job name, status (OK/ERROR), wall-clock time, and a truncated error message for each file.

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. **ODB Extraction (Abaqus Python 2.7):** The method `_run_abaqus_extraction(script_module)` locates `conv.py` inside the given sub-module folder (e.g., `src/simulations/cma/conv.py`) and invokes it via the Abaqus Python interpreter using `AbaqusScriptRunner`. This extracts raw field data from `.odb` files into `.npy` files.
2. **XDMF Compilation (Python 3.x):** Immediately after extraction, `_run_npy_to_xdmf` instantiates `NPY2XDMFParameters` with the `method_type` and `target_dir_key` arguments to resolve the correct source and destination directories. It then runs `NpyBatchToXdmfConverter` to compile the raw arrays into hierarchical HDF5/XDMF files.

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` string (e.g., `"Contour_Method"`) and a `target_dir_key` 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`, `polynomial_json_dir`, `polynomial_json_default`).
- **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.
- **Internal defaults:** `instance_name`, `step_name`, `nset_disp_name`, and `abaqus_cmd` have sensible defaults and rarely need to be changed from the JSON.

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`

Implements the experiment-specific logic for modifying input files. Two dedicated classes cover the two simulation workflows.

7.7.1 Contour Method Processor

Class: `ContourProcessor`

Applies polynomial boundary conditions derived from experimental surface measurements.

- **`_extract_sample_id(filename)`:** Static helper that normalises JSON filenames to a sample prefix. Recognises formats `Sample4.json`, `S4.json`, and `4.json`, all resolving to `S4`. Used to prefix the output filename (e.g., `S4_Mesh-0_60--Length-50_FI.inp`).
- **`calculate_z_robust(params, x, y, degree)`:** Static method that evaluates the polynomial $Z = P(x, y)$ at a node coordinate. Automatically detects the polynomial format:
  - **1D (univariate in Y):** `len(params) == degree + 1`.

  - **2D (bivariate in X, Y):** `len(params) == 2*degree + 1` — alternating a_k (X) and b_k (Y) terms plus a constant.

- **`run_batch`:** Supports two modes:
  - **Batch mode (default):** Matches each JSON in `polynomial_json_dir` to every base `.inp` in `cm_directory`. Files ending in `_Raw.json` are excluded.
  - **Single mode** (`use_default_single_file=True`): Applies `polynomial_json_default` to all meshes.

  For each pair, it reads the displacement node set (`nset_disp_name`; falls back to $z = 0$ nodes), generates `BCGenerator` lines, updates the material block, and fixes restart frequency.

7.7.2 Residual Stress Processor

Class: `ResidualProcessor`

Applies pre-computed residual stresses as `*Initial Conditions, type=STRESS` to the RSA mesh files.

- **Discovery:** Scans `rea_directory/Output/` for subdirectories that contain a `stress_input.txt` file (default; configurable via `csv_filename`). Each such subdirectory is a *case* (e.g., `S1_Mesh-0_60--Length-50_FI`).
- **Mesh resolution:** Strips `_FI` suffix and any `S`N`_` prefix via regex (`r"^S\d+_(.+)$"`) to obtain the base mesh name, then looks for the corresponding `.inp` in `rea_directory`.
- **Output location:** The generated `_FI.inp` file is written to `rea_directory` (not inside the case subdirectory) so all Abaqus run artefacts (`.odb`, `.sta`, `.msg`) remain together.
- **Injection workflow:** `StressReader` → `ElsetGenerator` → `InitialStressGenerator` → `INPInserter.insert_elsets` → `INPInserter.insert_initial_stresses` → `INPInserter.replace_material_block` → `INPInserter.fix_restart_frequency` → `INPWriter`.

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_process`, 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 uses a **Setter-based service architecture** centred on three building blocks:

- **`ContextMixin`:** Carries the shared Abaqus objects (`model`, `t_part`, `instance_name`, mesh size, length, etc.) and exposes `bind_context` / `propagate_to` to push those values down the full object graph automatically.
- **`LoggerMixin`:** Inherits from `ContextMixin` and attaches a per-class `setup_logger` instance on construction. All `Setter` classes inherit from this mixin.
- **`ServiceMixin` + `SERVICE_CATALOG`:** `ServiceMixin.run(catalog)` iterates a dictionary that maps attribute names (e.g., `_mat`, `_mesh`, `_job`) to their `Setter` classes. It instantiates each one, binds it to `self`, and propagates context to all of them in one call.

The concrete simulation classes inherit from **`BaseAnalysis(LoggerMixin, ServiceMixin)`**. The constructor calls `ServiceMixin.run(SERVICE_CATALOG)`, which builds all services, and then `bind_context`, which wires the shared Abaqus objects into every service. Subclasses need only implement four abstract methods (`_get_geometry_setter`, `_setup_steps`, `_setup_boundary_conditions`, `_setup_partitions`); `run_analysis()` then drives the full template sequence.

8.3 Base Class

Script: `base.py` **Class:** `BaseAnalysis(LoggerMixin, ServiceMixin)`

This is the single entry point for every concrete simulation script. It wires geometry, services, and the execution template together.

8.3.1 Constructor

`__init__(params, output_dir)` performs four sequential steps:

1. **Parameter unpacking:** Iterates `_PARAM_KEYS` and calls `setattr` to map each JSON key to the corresponding Python attribute:
2. **Geometry creation:** Calls the abstract `_get_geometry_setter()` (implemented by the subclass) and immediately calls `_geometry(depth=self.comprimento)`, which returns the `(model, t_part)` tuple and sets `self.instance_name = t_part.name + "-1"`.

JSON key	Python attribute
mesh_size	self.mesh_size
length	self.comprimento
initialInc	self.initialInc
maxInc	self.maxInc
maxNumInc	self.maxNumInc
minInc	self.minInc
nlgeom	self.nlgeom
time	self.time

Table 8.1: *_PARAM_KEYS mapping in BaseAnalysis.*

3. **Service initialisation:** `_init_services()` imports `SERVICE_CATALOG` from `_modules`, calls `ServiceMixin.run(catalog)` to instantiate and bind all setters (e.g. `self._mat`, `self._sec`, `self._inst`, `self._mesh`, `self._job`), then calls `bind_context` to propagate `model`, `t_part`, and `instance_name` to every service.

8.3.2 Template Method: `run_analysis()`

The public API for running a full simulation. Calls the following sequence regardless of subclass:

1. `first_stage()` — creates material, section, and assembly instance.
2. `_setup_partitions()` — abstract; subclass applies datum planes and mesh partitions.
3. `self._mesh.mesh()` — generates the C3D8R mesh.
4. `_setup_steps()` — abstract; subclass creates analysis steps.
5. `_setup_boundary_conditions()` — abstract; subclass applies loads and BCs.
6. `self._job.create_and_move_job()` — writes the `.inp` and moves it to `output_dir`.

Remark. *`first_stage()` hard-codes $E = 210\,000$ MPa and $\nu = 0.3$. These values are overwritten later by `INPInserter.replace_material_block` in the `pipeline` module using the values from `config.json`, so the hard-coded defaults only matter if the pipeline step is skipped.*

8.3.3 Abstract Methods (Subclass Contract)

Method	Responsibility
_get_geometry_setter()	Return the correct GeometrySetter instance.
_setup_steps()	Create Abaqus analysis steps.
_setup_boundary_conditions()	Apply loads, BCs, and constraints.
_setup_partitions()	Apply datum planes and mesh partitions.

Table 8.2: *Abstract methods that every concrete simulation class must implement.*

8.4 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.

8.4.1 Job Management

Script: `_set_job.py`

The `JobSetter` class generates the Abaqus job and delivers the final `.inp` file to the pipeline output directory.

- **Naming:** The job name is derived from the simulation parameters as `Mesh-{mesh_size}--Length-{` with dots replaced by underscores (e.g., `Mesh-0_6--Length-50`).
- **Generation:** Calls `mdb.Job(name, model)` then `job.writeInput()` to produce the `.inp` file on disk.
- **Relocation:** Moves the generated file from the Abaqus working directory to `output_dir`. If a file with the same name already exists at the destination it is removed first.

8.4.2 Analysis Step

Script: `_set_step.py`

`StepSetter.create()` configures the **Static, General** step.

- **Idempotency:** Deletes any existing step with the same name before re-creating it, so scripts can be re-run safely.
- **Incrementation:** Sets `initialInc`, `maxInc`, `maxNumInc`, `minInc` from the simulation parameters; `nlgeom` flag is passed directly.
- **Solver:** Uses `matrixSolver=ITERATIVE`.
- **Output requests:** Default `F-Output-1` and `H-Output-1` are deleted and replaced with a single custom `FieldOutputRequest` restricted to `S` (stress) and `U` (displacement) only, keeping ODB file sizes small.

8.4.3 Mesh Strategy

Script: `_set_mesh.py` (and submodules)

All meshing logic is consolidated in `MeshSetter.mesh()`, which executes four sequential steps:

1. **Delete existing mesh:** Any previous mesh is removed via `deleteMesh` before re-meshing (errors are caught silently).
2. **Global seeding:** `seedPart(size=mesh_size)` applies the DOE mesh size with `deviationFactor=0.001`.
3. **Element type:** `C3D8R` (8-node linear brick, reduced integration, STANDARD library) is assigned to all cells.
4. **Generate:** `generateMesh()` finalises the discretisation.

The submodule files (`_set_mesh_bias.py`, `_set_mesh_del.py`, `_set_mesh_sc8r.py`, `_set_mesh_seed.py`, `_set_mesh_stack.py`, `_set_mesh_sweep.py`) contain specialised helpers for bias seeding, sweep direction, and stack orientation that are composed into `MeshSetter` via the service catalog for more complex geometries.

8.5 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.

8.5.1 Material Definition

Script: `_set_material.py`

The `MaterialSetter.material(mat_name, E_Modulus, P_ratio)` method creates the constitutive model in the Abaqus database.

- **Elasticity:** Calls `model.Material(name)` then `materials[name].Elastic(table=((E, nu),))` to define isotropic linear elastic behaviour.

Remark. *The method only defines elastic properties. Plasticity is not set here; if a plasticity node is needed it must be added separately outside this setter.*

8.5.2 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:** `SectionSetter.create(mat_name, cell_set_name, section_name, homogeneous=True, thick=None, int_points=None)` creates either a `HomogeneousSolidSection` (`homogeneous=True`, default) or a `HomogeneousShellSection` with thickness and integration points (`homogeneous=False`).
2. **Assignment:** `SectionAssigner` (embedded inside `SectionSetter`) applies the section to all cells when `cell_set_name is None`, or to a named cell set otherwise.

8.5.3 Assembly Instantiation

Script: `_set_instance.py`

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

- **Idempotency:** `InstanceSetter.create()` checks whether the instance name already exists in `rootAssembly.instances` before calling `rootAssembly.Instance()`, so re-runs do not raise duplicate-key errors.
- **Naming Convention:** The instance name is resolved from `self.instance_name`, which is set in `BaseAnalysis.__init__` as `part.name + "-1"` (e.g., `T_Shape_Part-1`). This strict naming is vital for `inp_process` to correctly locate and inject boundary conditions later in the pipeline.

8.6 Geometry Construction Strategies

Package: `_modules/geometry/`

Implements the CAD logic of the framework using a **Strategy Pattern**: each simulation type has its own sub-package with a `GeometrySetter` class that exposes a single `_geometry(depth)` method returning a `(model, t_part)` tuple. `BaseAnalysis` calls this via the abstract `_get_geometry_setter()` method, so adding a new geometry requires only a new sub-package implementing that interface.

8.6.1 Strategy 1: T-Shape (`sim_one`)

Generates the standard T-profile benchmark geometry.

- **`_get_shape.py` — `ShapeGetterI`:** Returns a dictionary of T-profile dimensions (`h_width`, `h_thickness`, `v_width`, `v_height`, `offset_1`).
- **`_set_geometry.py` — `GeometrySetter`:** Draws an 8-point closed polygon on a `ConstrainedSketch` using `PolygonDrawer` and extrudes it with `BaseSolidExtrude(depth=comprime` to produce `T_Shape_Part`.

8.6.2 Strategy 2: Milling Profile (`sim_two`)

Handles the Profile Analysis (Exp2 / milling) geometry with different orientation requirements.

- **`_get_shape2.py` — `ShapeGetterII`:** Provides the specialised vertex coordinates for the milling profile.
- **`_set_geometry2.py` — `GeometrySetterTwo`:** Same sketch-and-extrude pattern as Sim One but with coordinates and model name specific to the milling setup.

8.6.3 Strategy 3: Rectangular Block (`sim_three`)

`_set_geometry3.py` — `GeometrySetterThree`: Creates a simple rectangular prism (`width` × `height` × `depth`). Used for coupon-level validation cases where no complex cross-section is needed.

8.6.4 Strategy 4: Inverted T-Shape (`sim_iv`)

`_set_geometry4.py` — `GeometrySetterIV`: Produces an inverted T cross-section (flange at the bottom, web extending upward) with a fixed 25 mm flange width and a parametric web width/height. The web is centred via offset `ofs` $= (25 -$ `width`$)/2$.

9 Simulation Workflow Pattern

The Contour Method (Chapter 10) and Residual Stress Analysis (Chapter 11) pipelines share a common two-phase orchestration pattern. This chapter describes the shared structure; the two chapters that follow document only the method-specific and experiment-specific deviations.

9.1 Two-Phase Pipeline

Every simulation pipeline script (`e{n}_cm_pipeline.py`, `e{n}_rs_pipeline.py`) exposes a `main()` function whose boolean flags control two sequential phases.

9.1.1 Phase 1 --- Simulation Setup and Execution

1. `ClearDirectory(target_directory)` — purges previous results.
2. `ParameterGenerator.generate_combinations(config)` — creates the design-of-experiments (DOE) matrix from `data/config.json`.
3. `GeometryGenerator.run_batch(params, directory)` — launches Abaqus CAE (`noGUI`) with the experiment-specific geometry script for every parameter set.
4. **BC / IC injection (method-specific)** — applies boundary conditions (CM) or initial conditions (RSA) to the generated `.inp` files. The exact processor differs per workflow; see the respective chapters.
5. `INPRunner.run_all(silent=True)` — submits every modified `.inp` to the Abaqus solver.

9.1.2 Phase 2 --- Result Conversion

- `ResultConverter.run_pipeline(method_type, script_module)` — runs ODB data extraction via the workflow's `conv.py` script, then NPY-to-XDMF compilation for open-format post-processing.

9.2 CAE Script Architecture

Each CAE script runs inside the Abaqus Python 2.7 kernel and defines a subclass of `BaseAnalysis`. The subclass must implement four abstract methods:

Abstract Method	Responsibility
`_get_geometry_setter`	Selects the geometry class (`GeometrySetter` for T-shape, `GeometrySetterTwo` for block).
`_setup_partitions`	Applies mesh partitions and, optionally, datum planes.
`_setup_steps`	Creates the analysis step sequence (`Static, General`).
`_setup_boundary_conditions`	Defines node sets and applies BCs and/or initial conditions.

Table 9.1: *Abstract methods of `BaseAnalysis`, implemented by every CAE script.*

Parameter injection: `GeometryGenerator` serialises the DOE dictionary into `os.environ["SIMULATION` each script reads it via `ParametersGetter()` and passes it to `BaseAnalysis.__init__`.

9.3 Experiment Geometry Variants

Both CM and RSA workflows exist in two experiment-specific variants. The geometry-level differences are identical across all four pipelines and are summarised once here:

Aspect	Experiment 1 (T-shape)	Experiment 2 (Block)
Geometry class	`GeometrySetter` (`sim_one`)	`GeometrySetterTwo` (`sim_two`)
Cross-section	T-shaped specimen	Rectangular / milled block
Config override	Uses default `config.geometry_script`	Overrides at runtime: `config.geometry_script = _`

Table 9.2: *Geometry variants shared by all simulation pipelines.*

10 Contour Method Simulation Workflow

Pipeline scripts:

- `src/simulations/e1_cm_pipeline.py` — Experiment 1 (T-shape)
- `src/simulations/e2_cm_pipeline.py` — Experiment 2 (block / milling)

CAE scripts:

- `src/simulations/cma/script.py` — Exp1 geometry builder
- `src/simulations/cma2/script.py` — Exp2 geometry builder

Both pipelines follow the shared two-phase pattern described in Chapter 9. The key CM-specific difference is the BC injection step (Phase 1, Step 4): displacement boundary conditions are derived from polynomial surface fits and written into the mesh `.inp` files by `ContourProcessor`.

10.1 CM-Specific Phase 1: Boundary-Condition Injection

After mesh generation (Step 3 of the shared pattern), `ContourProcessor(config).run_batch()` applies polynomial-fitted surface data as nodal displacements to every mesh, producing `_FI.inp` files. The operating mode depends on the experiment:

- **Single-file mode** (Exp1): `run_batch(use_default_single_file=True)` applies one default polynomial JSON to all meshes.
- **Batch mode** (Exp2): `run_batch()` iterates over every JSON in `polynomial_json_dir` (excluding `_Raw.json`), producing an `S`*`N`*`_Mesh-...-_FI.inp` per mesh/polynomial combination.

10.2 CM-Specific Phase 2: Result Conversion

- `ResultConverter.run_pipeline(method_type="Contour_Method", script_module="cma")` — ODB extraction via `cma/conv.py`, then NPY-to-XDMF compilation.

10.3 CM Step Topology

Both `cma/script.py` and `cma2/script.py` define a single analysis step:

- `Step-1` (Static, General) with `previous = Initial`.

10.4 Experiment Comparison

Both CAE scripts define `ContourAnalysis(BaseAnalysis)`. The per-experiment differences are confined to geometry, partitions, and boundary conditions:

Aspect	**Experiment 1 (T-shape)**	**Experiment 2 (Block)**
Geometry script	`cma/script.py`	`cma2/script.py` (overridden at runtime)
Geometry class	`GeometrySetter` (`sim_one`)	`GeometrySetterTwo` (`sim_two`)
BC mode	Single-file: one default polynomial JSON	Batch: every JSON × every mesh
CAE class	`ContourAnalysis(BaseAnalysis)`	`ContourAnalysis(BaseAnalysis)`
Partitions	`_invt_mesh_part.set()` for partition alignment	No partition call
Encastre BC	`Set_Enc`: $z \geq L$	Back face: $z \approx L$ (tolerance ± 0.01)
Displacement BC	`Set_Disp`: $z \leq 0$	`Set_Disp`: $z \leq 0$

Table 10.1: *Per-experiment differences in the Contour Method workflow.*

11 Residual Stress Analysis (RSA) Workflow

Pipeline scripts:

- `src/simulations/e1_rs_pipeline.py` — Experiment 1 (T-shape)
- `src/simulations/e2_rs_pipeline.py` — Experiment 2 (block / milling)

CAE scripts:

- `src/simulations/rsa/script.py` — Exp1 RSA builder
- `src/simulations/rsa2/script.py` — Exp2 RSA builder

Both pipelines follow the shared two-phase pattern described in Chapter 9. Unlike the Contour Method workflow, RSA has two additional concerns: a *strict dependency on CMA results* and a *stress-mapping bridge* that runs between mesh generation and job submission.

11.1 CMA Dependency

Each RSA pipeline accepts a `run_cma` flag. When `True`, the script imports and calls the corresponding CM pipeline (`e1_cm_pipeline.main()` or `e2_cm_pipeline.main()`) before starting its own workflow. This guarantees that the HDF5 files containing the measured stress fields are up to date.

11.2 RSA-Specific Phase 1: Stress Mapping and IC Injection

After mesh generation (Step 3 of the shared pattern), the RSA pipeline executes two extra sub-steps before job submission:

11.2.1 Stress Mapping (ElementProcess Bridge)

```
Nodes_main(str(config.rea_directory), use_s1=True, use_s2=False, use_s3=False
proc = StressProcessor(str(config.rea_directory), tolerance=5e-2, chunk_size=
proc.process_all_simulations(str(cm_hdf5_path))
```

`Nodes_main` extracts element centroids from the RSA mesh; `StressProcessor` maps S_{33} from the CMA HDF5 files onto those centroids using 2D KDTree (see Chapter 5.5). For Experiment 2, `StressProcessorBatch` replaces `StressProcessor` to handle the multi-case (`S1_`, `S2_`, ...) directory layout (see Chapter 5.6).

11.2.2 Initial-Condition Injection

`ResidualProcessor(config).run_batch()` injects `*Initial Conditions` blocks into every `.inp` file before solver submission.

11.3 RSA-Specific Phase 2: Result Conversion

- `ResultConverter.run_pipeline(method_type="Residual_Stresses_Analysis", script_module="rsa")` — ODB extraction via `rsa/conv.py`, then NPY-to-XDMF compilation.

  Note: `rsa2` does **not** have a `conv.py` file. Result conversion for Exp2 RSA reuses the same `rsa/conv.py` extraction script.

11.4 Multi-Step Physics

Unlike the single-step CM workflow, every RSA CAE script creates a **three-step** analysis sequence, each with `previous` chained to the prior step:

1. **Material-Removal:** `ModelChangeSetter` deactivates the "Remove" element set, simulating the physical cut.
2. **BC-Removal:** Removes the encastre BC applied below the y-cutoff ($y < h_{avg}/8$), allowing the part to deform.
3. **BC-Removal_Nodes:** Releases pinned nodal BCs at $y < 0$ and applies final node displacement constraints via `_bc_node.setI()` (Exp1) or `_bc_node.setII()` (Exp2).

11.5 Plane Settings and Extra Initialisation

Beyond the standard `BaseAnalysis` constructor, `ResidualAnalysis` performs two additional initialisation steps:

1. **Plane settings:** `_load_plane_settings()` reads the experiment's JSON file (`data/e1_plane_settin` or `data/e2_plane_settings.json`) to obtain the ZX and ZY cutting-plane coordinates. These are stored in `self.selected_planes` (consumed by `_rm_datum`) and `self.remove_region`.
2. **Extra propagation:** `self.propagate_to(_rm_datum, _rm_region, _sets)` pushes the plane coordinates and region info into the geometry-setup services.

11.6 Overridden `run_analysis()`

`ResidualAnalysis` overrides `BaseAnalysis.run_analysis()` to insert two extra calls:

- `self._sets.setI()` (or `setII()` for Exp2) — creates named element/node sets required by the BC definitions, inserted after mesh generation.
- `self.t_part.generateMesh()` — a second mesh generation after boundary conditions are applied, ensuring the final partitioned mesh is consistent.

11.7 Experiment Comparison

Both CAE scripts define `ResidualAnalysis(BaseAnalysis)`. The per-experiment differences are:

Aspect	Experiment 1 (T-shape)	Experiment 2 (Block)
Geometry script	`rsa/script.py`	`rsa2/script.py` (overridden at runtime)
Geometry class	`GeometrySetter` (`sim_one`)	`GeometrySetterTwo` (`sim_two`)
Stress mapper	`StressProcessor` (single-case)	`StressProcessorBatch` (multi-case `S1_`, `S2_`, ...)
CAE class	`ResidualAnalysis(BaseAnalysis)`	`ResidualAnalysis(BaseAnalysis)`
Plane settings	Resolved internally from `data/e1_plane_settings.json`	Explicit `plane_settings_path` argument; default `data/e2_plane_settings.json`
Step names	Inline string literals	Module-level `STEP_NAMES` tuple
Sets / BCs	`_sets.setI()`, `_bc_node.setI()`	`_sets.setII()`, `_bc_node.setII()`
Partitions	`_invt_mesh_part.set()` + datum planes	Datum planes only (no `_invt_mesh_part`)
Result conversion	`rsa/conv.py`	Reuses `rsa/conv.py` (no `rsa2/conv.py`)

Table 11.1: *Per-experiment differences in the RSA workflow.*

12 Graphical Interfaces

Module: `src/simulations/gui/`

This sub-package provides two standalone desktop applications used at different stages of the simulation workflow:

- `plane_selector.py` — cutting-plane configuration (pre-processing, **customtkinter**).
- `answer_viewer.py` — 3-D result visualization (post-processing, **PyQt5 + PyVista**).

12.1 `plane_selector.py` --- Cutting Plane Configurator

Dependencies: `customtkinter`, `matplotlib`, `tkinter.messagebox`, `processor` (project-internal).

12.1.1 Purpose

Before the RSA workflow can run, the cutting-plane coordinates (`plane_zx`, `plane_zy`) and the material-removal region must be defined. These values are saved in a JSON file (e.g. `e1_plane_settings.json` or `e2_plane_settings.json`) and consumed by the CAE scripts documented in Chapter 11.7.

`plane_selector.py` provides an interactive GUI that lets the user adjust these parameters visually and write them back to disk with a single click.

12.1.2 Architecture

The module follows the **Model–View** pattern with two classes:

12.1.3 JSON Schema

The persisted file has the following structure:

Class	Role
`PlaneSettingsModel`	JSON I/O: loads and saves `plane_zx`, `plane_zy`, and `remove_region` to/from a `.json` file. Falls back to hard-coded defaults if the file is missing or unreadable.
`ContourPlaneApp(ctk.CTk)`	Main window. Contains a sidebar with input fields and a Matplotlib canvas for the 2-D preview.

```
{
    "plane_zx": {"point": [0.0, <Y_pos>, 0.0], "normal": [0.0, 1.0, 0.0]},
    "plane_zy": {"point": [<X_pos>, 0.0, 0.0], "normal": [1.0, 0.0, 0.0]},
    "remove_region": [<ref_x>, <ref_y>]
}
```

- `plane_zx` — horizontal cut: the Y coordinate of the cutting plane. Normal $\hat{y}$.
- `plane_zy` — vertical cut: the X coordinate of the cutting plane. Normal $\hat{x}$.
- `remove_region` — a 2-D reference point that indicates *which side* of the cut is removed.

12.1.4 Geometry Detection

The application infers the specimen shape from the JSON filename:

- `e1` / `exp1` → Experiment 1 T-profile (loads `exp1_sample01.py` via `processor.ExpProcessor`).
- `e2` / `exp2` → Experiment 2 block (loads `exp2_sample.py`).

If the geometry file cannot be resolved, a 50 mm × 50 mm fallback square is drawn.

12.1.5 Visualization

`update_plot()` redraws the Matplotlib canvas:

1. Draws the specimen outline as a `Polygon` (T-profile or rectangular block).
2. Overlays a red hatched rectangle for the material-removal zone, computed from the cut positions and the reference point.
3. Renders horizontal and vertical dashed lines for the ZX and ZY planes.
4. Plots the reference removal point as a red dot.

12.1.6 Entry Point

```
from simulations.gui.plane_selector import run_gui
run_gui("data/e1_plane_settings.json")
```

`run_gui(path)` instantiates `ContourPlaneApp` and enters the Tkinter main loop. On **SAVE JSON** the settings are written and the window closes.

12.2 answer_viewer.py --- XDMF / HDF5 Result Viewer

Dependencies: PyQt5, pyvistaqt, pyvista, h5py, numpy, matplotlib, xml.etree.ElementTree.

12.2.1 Purpose

After the pipeline converts ODB results to HDF5 (via ResultConverter), the engineer needs to inspect stress fields, displacements, and deformation modes interactively. answer_viewer.py provides a desktop application that combines:

- A 3-D finite-element renderer (PyVista QtInteractor).
- A per-field colorbar and statistics panel.
- Abaqus-style view controls, animation, and clipping driven by the same plane_settings.json used in the simulation.

12.2.2 Class: XDMFHDFViewer(QMainWindow)

Internal State

Attribute	Description
h5_file	Open h5py.File handle.
xdmf_data	Parsed XDMF root element (optional metadata).
current_model / step / frame / field	Active selection in the navigation combos.
coordinates	$N_{\text{pts}} \times 3$ numpy array of node positions.
connectivity	$N_{\text{elem}} \times n_{\text{npe}}$ element connectivity.
grid	pv.UnstructuredGrid built from the above.
element_status	Binary mask (1 = active, 0 = removed) for clipping.
removal_cfg	Loaded plane_settings.json (auto-detected per model).

HDF5 Layout Expectations

The viewer reads data from an HDF5 file with the following hierarchy:

```
<model_name>/
    geometry/
        coordinates     # (N_pts, 3)
        connectivity    # (N_elem, npe)
    time_series/
        <step_name>/
            <frame_name>/
                <field_name>  # (N, ...) dataset
```

Multiple models (e.g. S1_sample_01, S2_sample_01) can coexist in a single file; the Model combo box lists them.

UI Layout

The main window is arranged as a horizontal splitter:

- **Left panel — Control tabs:**
  1. **Data tab:** View buttons (Abaqus-style $\pm X/Y/Z$, Iso 1/2, Fit, Ortho), file info, model selector, mesh info, step/frame/field combos, and "Update Visualization" button.
  2. **Visualization tab:** Colormap selection (viridis, jet, coolwarm, ...), transparency slider, clipping mask toggle, wireframe/nodes checkboxes, deformation scale slider, and orthographic projection toggle.
- **Right panel — 3-D viewport** (`pyvistaqt.QtInteractor`) with a bottom strip containing the colour legend and display-info panel (model, field, step/frame, min/max values).

12.2.3 Mesh Construction

`build_grid()` converts the HDF5 arrays into a PyVista `UnstructuredGrid`:

- 8 nodes per element → `HEXAHEDRON`.
- 4 nodes per element → `TETRA`.

The mesh is cached per model (`_cached_model`) and rebuilt only when the user switches to a different model.

12.2.4 Clipping Mask

When a model is selected, `_load_plane_settings_for_model()` auto-selects the correct JSON:

- Model name matching `S\d+_` → `e2_plane_settings.json`.
- Otherwise → `e1_plane_settings.json`.

`_build_element_mask()` then computes a bounding box from the ZX/ZY plane coordinates and the removal-region reference point. Element centroids outside the box are marked as removed (`element_status = 0`). During rendering, `extract_cells(alive_ids)` passes only the active subset to PyVista.

12.2.5 Field Loading

`load_field_data()` reads the raw HDF5 dataset and maps it to `grid.point_data` or `grid.cell_data`, handling several storage layouts:

Field	Shape	Handling
`displacement`	$(N, 3)$	Nodal or cell vectors; enables deformed-shape rendering.
`stress_tensor`	$(N, 9)$	Extracts S_{33} (index 8) as scalar for display.
Generic scalar	$(N_{\text{pts}},)$ or $(N_{\text{elem}},)$	Direct assignment.
Gauss-point	$(N_{\text{elem}} \times n_g, \ldots)$	Averaged per element before assignment.

12.2.6 Rendering

`render_pyvista()` orchestrates the final display:

1. Applies clipping if the mask checkbox is active.
2. For displacement fields, deforms the mesh by $\mathbf{u} \times s$ where s is the deformation scale slider value.
3. Adds the mesh with the selected colormap, opacity, and wireframe options.
4. Vector fields are rendered via magnitude scalars; optional glyph arrows can be toggled.
5. The camera position is preserved between field/frame changes within the same model.

12.2.7 Animation

A `QTimer` drives frame-by-frame playback:

- `next_frame()` / `prev_frame()` cycle through the `frame_combo` entries.
- Speed is configurable via `animation_speed` (ms between frames).

12.2.8 Export

- **Image:** `plotter.screenshot(path)` to PNG, JPEG, or PDF.
- **Data:** Current field array saved as CSV (`np.savetxt`) or NPY (`np.save`).

12.2.9 Entry Point

```
python -m simulations.gui.answer_viewer
```

Launches the `QApplication`, opens an empty window, and the user loads files via **File → Open HDF5**.

12.3 Legacy: Incremental Cutting Strategy

Module Path: `src/Simulations/rsa_cut/`

> **Deprecation Notice:** This module represents an experimental branch designed to validate the effects of multi-stage material removal. It is currently **deprecated** and not used in the production pipeline. The rationale for its discontinuation is detailed in Section 12.3.3.

12.3.1 Concept and Design

The scripts in this folder were developed using an earlier, monolithic class design (similar to the `attempt.py` prototype). The primary goal was to simulate the machining process not as a single instantaneous removal of volume, but as a sequence of discrete cutting steps.

Workflow Intention:

1. **Phased Removal:** Instead of removing the entire cut volume in one `Model Change` step, the simulation divides the removal region into multiple sub-sets (e.g., Layer 1, Layer 2, Layer 3).
2. **Sequential Solving:** The solver calculates the equilibrium state after removing Layer 1, propagates the stress/deformation, and then proceeds to remove Layer 2.
3. **Objective:** To verify if the stress redistribution path (history-dependent) significantly alters the final residual stress profile compared to a single-step removal.

12.3.2 Module Components

- **Builder (`REA_Extended_Cut.py`):** A comprehensive script that constructs the geometry and defines the multiple analysis steps. Unlike the modern Mixin architecture, this script handles meshing, partitioning, and step generation within a single class structure. It relies on a pre-defined JSON configuration to determine the slice planes.
- **Orchestrator (`REA_Main_Cut.py`):** The execution entry point. It mimics the logic of the main RSA workflow but directs the pipeline to use the multi-step builder. It handles the integration with the Abaqus solver execution.
- **Plane Configuration (`ContourPlaneGUI.py`):** A specific graphical interface designed to define the multiple cutting planes ($Z_1, Z_2, \ldots, Z_n$) required to slice the removal volume into discrete chunks.

12.3.3 Theoretical Limitations & Discontinuation

The project moved away from this approach due to a fundamental limitation in simulating machining processes using standard Static Implicit analysis with pre-defined meshes.

1. **Static Equivalence:** In a linear elastic (or even standard elastic-plastic) static analysis, the final state of equilibrium depends primarily on the final boundary conditions. Removing volume V in one step often yields the identical result to removing $V/2 + V/2$ in two steps, provided no complex path-dependent plasticity or contact friction is involved. The computational cost of multiple steps yielded no accuracy gain.

2. **The "Blind Cut" Problem:** A real machining process is interactive: as the tool cuts pass N, the workpiece distorts due to released stresses. The tool (moving in a rigid machine path) then cuts a different amount of material in pass $N + 1$ relative to the distorted shape.
   Simulation Limitation: In this Abaqus implementation, the "Volume to be Removed" is defined by element sets on the **undeformed** mesh. The simulation does not update the "tool path" relative to the current deformation. It simply deletes the pre-selected elements. Therefore, it fails to capture the physical phenomenon of the part distorting "away" from or "into" the cutting tool between steps.

12.4 Legacy: Multi-Material Stiffness Gradient

Module Path: `src/Simulations/rsa_e/`

> **Experimental Status:** This module represents an investigative branch designed to analyze Residual Stresses in components with heterogeneous material properties (Functionally Graded Materials or multi-layered structures). It implements a strategy where the stiffness (Young's Modulus) varies spatially across the component height.

12.4.1 Concept: Variable Stiffness Mapping

The core hypothesis of this module is that assuming a homogeneous material property (single E and ν) might introduce errors if the manufacturing process (e.g., welding or cladding) significantly alters the local stiffness. The module attempts to discretize the domain into "material stripes" with varying elastic moduli.

12.4.2 Component Architecture

1. Multi-Material Builder

Script: `REA_Extended_E.py`

This script extends the standard geometry generator to support material partitioning.

- **Class `MultiMaterialContourAnalysis`:** Inherits from the standard extended analysis but adds logic to partition the geometry vertically.
- **Dynamic Partitioning:** The method `partition_geometry_2` slices the T-shape into N horizontal layers (defined by `num_divisions`).
- **Stiffness Scaling:** The method `assign_different_materials` iterates through these partitions and creates unique Abaqus Material definitions for each. It ap-

plies a scaling factor to the Young's Modulus:

$$E_{layer\_i} = \frac{E_{base}}{Scale^i} \tag{12.1}$$

This effectively creates a gradient of stiffness along the height of the sample, allowing for the simulation of material degradation or transition zones.

2. Standalone INP Injector

Script: `Set_Creator_Inp.py`

Unlike the main pipeline which relies on the integrated `_inp_modules`, this experimental branch uses a specialized, standalone class for file manipulation.

- **Class `InpStressUpdater`:** It parses the generated '.inp' files as raw text. It identifies the target instance and manually injects:
  1. `*Elset` definitions corresponding to the stress map.
  2. `*Initial Conditions, type=STRESS` blocks derived from external CSV data.
  3. A rewritten `*Material` block that matches the specific properties required for the analysis.

3. Orchestration & Visualization

- **Orchestrator (`REA_Main_E.py`):** Similar to other main scripts, it manages the batch execution. It specifically coordinates the `Nodes_main` extraction and the 'StressProcessor' mapping before triggering the `BatchInpStressUpdater` to modify the input files.
- **Plane Selection GUI (`ContourPlaneGUI.py`):** Provides a visual interface (using `customtkinter` and `matplotlib`) for the user to define the cutting planes (ZX, ZY) and removal regions visually. This configuration is saved to a JSON file (`plane_settings.json`) which drives the partitioning logic in the builder.

Part II

Notebooks

The `notebook/` directory contains three Jupyter notebooks used for interactive inspection of intermediate and final results. They are *read-only diagnostic tools*: they do not alter any pipeline data and can be executed independently of the main workflow.

File	Experiment	Purpose
`e1_sim_plot.ipynb`	Exp1	3-D / 2-D surface reconstruction from polynomial JSON
`e2_sim_plot.ipynb`	Exp2	1-D curve-fit quality check and overlay plot
`inp_bc_viewer.ipynb`	Both	Boundary-condition displacement map from `.inp` file

13

e1_sim_plot.ipynb --- Exp1 Surface Reconstruction

File: notebook/e1_sim_plot.ipynb
Dependencies: numpy, matplotlib, exp_process.core.mesher, exp_process.core.fitter, exp_process.utils.io, processor

13.1 Purpose

This notebook reconstructs and visualises the deformed surface of the Experiment 1 T-shaped specimen from the polynomial model coefficients produced by the exp_process pipeline (Chapter 3). Its main use is a *post-fit sanity check*: it renders the fitted surface on the actual T-profile mesh so that the analyst can confirm that the polynomial represents the measured deformation correctly before using it in the simulation.

13.2 Workflow

13.2.1 Cell 1 --- Imports and Path Setup

Appends src/ to sys.path and imports the four project modules used downstream:

- MeshGenerator — creates a scattered point grid over the T-profile geometry.
- Fitter — evaluates a 2-D polynomial on the grid.
- IOUtils — JSON loader.
- ExpProcessor — reads the sample geometry file (.py) and returns dimension parameters.

13.2.2 Cell 2 --- Load Geometry and Model

1. Reads data/input/exp1/exp1_sample01.py through ExpProcessor.process() to obtain the T-profile dimensions (h_width, v_height, etc.).

2. Loads the target JSON from `data/output/exp1/surface_data/` — default `Average.json`. The file name is configurable via `model_name`.
3. Calls `MeshGenerator.t_shape_grid(t_dims, step=0.5)` to generate a 2-D scatter grid with 0.5 mm resolution over the T-profile domain.
4. Evaluates the polynomial: $z =$ `Fitter.eval_2d_poly`$(x, y,$ `model_data`$)$.

   Where to change: Modify `model_name` to inspect any other JSON in `surface_data/` (e.g. `S1_Top.json`). Adjust `step_size` to increase or decrease mesh density.

13.2.3 Cell 3 --- 3-D Surface Plot

Renders a `scatter3D` plot with:

- Axes: X (mm), Y (mm), Z (deformation amplitude).
- Colour map: `viridis`, mapped to Z values.
- Aspect ratio locked to $(X_{\text{range}}, Y_{\text{range}}, X_{\text{range}})$.

13.2.4 Cell 4 --- 2-D Heatmap

Renders a top-view scatter plot with `RdBu_r` colormap (square markers, 2 pt), giving an immediate visual of the surface topology projected onto the XY-plane.

13.3 Expected Output

Two Matplotlib figures, inline:

1. A 3-D perspective view of the reconstructed surface mapped onto the T-profile shape.
2. A 2-D heatmap showing the spatial distribution of the deformation amplitude.

14

e2_sim_plot.ipynb --- Exp2 Curve-Fit Visualisation

File: `notebook/e2_sim_plot.ipynb`
Dependencies: `numpy`, `matplotlib`, `json`, `glob` (standard library only — no project imports required)

14.1 Purpose

Experiment 2 produces 1-D polynomial fits (curves along the X axis) for each sample, stored as JSON files in `data/output/exp2/curve_data/`. This notebook provides two complementary views:

1. **Per-sample fit quality:** raw input points vs. transformed points vs. fitted polynomial.
2. **Cross-sample overlay:** all fitted curves plotted on a single axis to compare repeatability across samples.

14.2 Workflow

14.2.1 Cell 1 --- Imports and Configuration

Sets global Matplotlib style (`seaborn-v0_8-whitegrid`) and defines `DATA_DIR` pointing to `data/output/exp2/curve_data/`.

14.2.2 Cell 2 --- Helper Functions

Three utility functions defined in this cell:

Function	Description
`load_json(filename)`	Loads a JSON from `DATA_DIR`; returns `None` with a warning if the file is not found.
`extract_points(data)`	Extracts X and Z coordinate arrays from `data["points"]`. Supports both 3-column (X, Y, Z) and 2-column (X, Z) layouts.
`calculate_poly(x, coeffs)`	Evaluates the polynomial via `np.polyval(coeffs, x)` (coefficients in descending degree order).

14.2.3 Cell 3 --- Per-Sample Fit Quality Plot

`plot_sample_fit(sample_id)` loads the pair `{sample_id}_Raw.json` and `{sample_id}.json` and renders a two-panel figure:

- **Left panel:** raw points (grey) and transformed points (blue) scatter — allows inspection of the preprocessing stage.
- **Right panel:** transformed points with the fitted polynomial superimposed as a red line.

The function is called automatically for every sample discovered by scanning `DATA_DIR` for `*_Raw.json` files.

14.2.4 Cell 4 --- Overlay Comparison Plot

Iterates over all non-raw, non-step JSON files in `curve_data/` using `glob`. For each file:

1. Reads `coeffs` (or the legacy key `params`).
2. Evaluates the polynomial over $x \in [0, 40]$ mm with 200 points.
3. Plots it as a labelled line.

The resulting figure shows all fitted curves together, making sample-to-sample variability immediately visible.

> **Where to change:** Adjust the `xs` range (`np.linspace(0, 40, 200)`) to match the physical sample length. The filtering condition `if "_Raw" in filename or "_Steps" in filename` can be extended to exclude additional intermediate files.

14.3 JSON Format

Both the per-sample and overlay cells expect JSON files with at least:

```
{
    "points": [[x0, y0, z0], ...],  # raw/processed point cloud
    "coeffs": [a_n, ..., a_1, a_0]  # descending-degree polynomial coefficient
}
```

14.4 Expected Output

- One two-panel figure per detected sample (raw vs. fit).
- One overlay figure with all final polynomial curves.

15

inp_bc_viewer.ipynb --- INP Boundary-Condition Visualiser

File: notebook/inp_bc_viewer.ipynb
Dependencies: numpy, matplotlib, scipy.interpolate.griddata, re (standard library only)

15.1 Purpose

After ResidualProcessor injects the *Initial Conditions block and ContourProcessor writes the displacement boundary conditions into an .inp file, it is important to verify that the applied nodal field matches the fitted surface before submitting the job to Abaqus.

This notebook parses a generated _FI.inp file, extracts the U_3 (DoF 3, Z-direction) displacement values applied at every node, and renders them as both a 2-D contour heatmap and a 3-D surface.

15.2 Single-Cell Architecture

The entire notebook consists of one large code cell divided into three logical sections:

15.2.1 Section 1 --- Parser: parse_inp_bc_visualization()

```
nodes, displacements = parse_inp_bc_visualization(inp_path)
```

A line-by-line parser that reads the .inp file and populates two dictionaries:

Output	Content
nodes	{node_id: (x, y, z)} — all nodes from the *Node section.
displacements	{node_id: value} — boundary condition values filtered to DoF 3 only (from the *Boundary section).

The parser handles the Abaqus assembly-level naming convention (e.g. `T_SHAPE_PART-1.1234`) by splitting on `'.'` and reading only the integer part after the last dot. Comment lines (`**`) and non-numeric lines are silently skipped.

> **Parsing logic:** State flags `reading_nodes` and `reading_bcs` are toggled by keyword lines (`*Node`, `*Boundary`, `*Element`, or any other `*` keyword). Only DoF 3 entries are stored to displacements.

15.2.2 Section 2 --- Plotter: `plot_bc_surface()`

Takes the node and displacement dictionaries, aligns them (only nodes that have a boundary condition are plotted), and produces two figures:

1. **2-D Contour Heatmap:**
   - Node positions are interpolated onto a 100×100 regular grid via `scipy.interpolate.griddata` with cubic interpolation.
   - Rendered as a filled contour plot (`contourf`, 100 levels, `jet` colormap).
   - Raw BC nodes are overlaid as scatter points for comparison.
   - Colourbar label: *Applied Displacement U3* (*mm*).
2. **3-D Surface:**
   - `ax.plot_trisurf(x, y, z)` renders the displacement field directly from the unstructured node data.
   - Colourbar: U_3 (mm).

15.2.3 Section 3 --- Execution Block

```
arquivo_inp = r"C:\Simulation4\Contour_Method\s1_Mesh-2_98---Length-10_FI.inp"
plot_bc_surface(arquivo_inp)
```

> **Where to change:** Update `arquivo_inp` to point to the `_FI.inp` file you want to inspect. Any file generated by `ContourProcessor` (stored in `data/output/exp*/cm_directory/`) is compatible.

15.3 Expected Output

Two Matplotlib figures, inline:

1. A 2-D XY contour map of U_3 showing the spatial distribution of the applied displacements over the midsection cut face.
2. A 3-D triangulated surface of the same displacement field, allowing perspective inspection of the deformation magnitude and asymmetry.

Diagnostic counts are printed to stdout:

```
Reading file: s1_Mesh-2_98---Length-10_FI.inp ...
--> Nodes found: 4823
--> Boundary conditions found: 311
```