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 **??**).
- **Data Standardization:** A unified pipeline that converts proprietary Abaqus outputs (ODB) into open scientific formats (HDF5/XDMF) for streamlined post-processing (Chapter **??**).
- **Experimental Database:** A central repository (`Exp_Data`) that stores the statistical mean dimensions of physical samples, ensuring simulations match the manufactured reality (Chapter **??**).

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 **??**).
- **Abaqus Scripting Interface (ASI):** A modular, object-oriented framework that runs inside the Abaqus kernel to procedurally generate geometries, meshes, and analysis steps (Chapter **??**).
- **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 **??**).

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 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.

2.1 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 2.1: *Data flow through the `exp_process` pipeline.*

2.2 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`

2.3 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 2.1: *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.

2.4 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
```

2.5 Chapter Organization

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

2.6 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.

2.6.1 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 2.2: *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.

2.6.2 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']`).

2.6.3 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']`).

2.6.4 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.

2.7 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.

2.7.1 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 2.3: *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.

2.7.2 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 2.4: *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": [...]}
```

2.7.3 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 2.5: *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.

2.7.4 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 2.6: *ModelOps interface.*

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

2.7.5 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 2.7: *Rebuilder interface.*

2.7.6 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 2.8: *StepSegmenter interface.*

2.7.7 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 2.9: *DataTransformer rule keys.*

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

2.8 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.

2.8.1 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 2.10: *Preprocessing config fields.*

`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 2.11: *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 **??** for syntax.

2.8.2 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 **??**.

2.8.3 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 **??**.

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.

2.8.4 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.

2.8.5 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 non-linear 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.

2.8.6 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.

2.8.7 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 **??**.

2.8.8 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.

2.9 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 **??** into an ordered, experiment-specific workflow, and exposes one method — `run()` — as the public interface.

The separation of responsibilities is strict:

- **Core** (Section **??**) — stateless algorithms, no file I/O.
- **Procedures** (Section **??**) — 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.

2.9.1 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 2.12: *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.

2.9.2 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 2.13: *`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.

2.9.3 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 2.14: *`Exp2PipelineConfig` fields.*

2.9.4 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)
```

2.10 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.

2.10.1 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 2.15: *`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.

2.10.2 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 **??**).

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 2.16: *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()`.

3
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).

3.1 Stage 1: Configuration and Orchestration

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

3.1.1 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 3.1: *Conversor module file renaming.*

3.1.2 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.

3.1.3 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.

3.1.4 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.

3.2 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 3.2: *New parameters in `OdbToNPYConverter`.*

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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`.

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 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 3.3: *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.

4

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.

4.1 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 4.1: *`element_process` module file renaming.*

4.1.1 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`.

4.1.2 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`.

4.2 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.

4.2.1 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:

$$C_{elem} = \frac{1}{N} \sum_{i=1}^{N} P_{node\_i} \tag{4.1}$$

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

4.2.2 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 4.2: *Output files from* `extractor.py`.

4.3 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.

4.3.1 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.

4.3.2 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:

$$n_x = \max\left(2,\ \left\lfloor t \cdot \frac{x_{max} - x_{min}}{\Delta_{min}} \right\rfloor\right) \tag{4.2}$$

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

4.3.3 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.

4.4 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.

4.4.1 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 4.3: *Source stress file formats accepted by `load_tension_field`.*

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

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

4.4.2 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.

4.4.3 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
...
```

4.5 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).

4.5.1 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 4.4: *`StressProcessor` constructor parameters.*

4.5.2 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.

4.5.3 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 4.5: *Output files from `stress_mapping.py`.*

4.5.4 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"
)
```

4.6 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.

4.6.1 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.

4.6.2 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 4.6: *File patterns recognised by `StressProcessorBatch.find_simulations`.*

The `S`*`N`*`_` 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`.

4.6.3 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/
      ...
```

4.6.4 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

5
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.

5.1 Library Architecture

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

5.1.1 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.

5.1.2 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.

5.1.3 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.

5.1.4 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.

5.2 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.

5.2.1 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.

5.2.2 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.

5.3 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.

5.3.1 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.

5.4 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.

5.4.1 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.

5.4.2 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.

5.5 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.

5.5.1 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.

5.5.2 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.

5.5.3 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.

5.6 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.

5.6.1 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.

5.6.2 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:

$$\text{Map}(ElementID) \rightarrow \text{SectionProperties} \tag{5.1}$$

   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).

5.6.3 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:

$$x_{min} \leq c_x \leq x_{max} \quad \text{and} \quad y_{min} \leq c_y \leq y_{max} \tag{5.2}$$

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

5.7 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.

5.7.1 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.

5.7.2 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.

5.7.3 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.

6
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.

6.1 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.

6.2 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.

6.2.1 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.

6.3 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.

6.3.1 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.

6.4 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.

6.4.1 Two-Stage Execution

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

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

6.4.2 Pipeline Integration

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

6.5 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.

6.5.1 Structure

The class groups parameters into logical domains:

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

6.6 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.

6.6.1 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.

6.6.2 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.

6.7 Simulation Processors (Business Logic)

Script: `processors.py`

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

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

6.7.1 Contour Method Processor

Class: `ContourProcessor`

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

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

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

6.7.2 Residual Stress Processor

Class: `ResidualStressProcessor` (and variants)

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

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

7 Abaqus Scripting Interface (ASI) Framework

Module Path: `src/Simulations/_modules/`

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

7.1 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.

7.1.1 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.

7.2 Design Philosophy

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

7.3 Core Definitions (Job, Step, Mesh)

Package: `_modules/core/`

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

7.3.1 Job Management

Script: `_set_job.py`

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

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

7.3.2 Analysis Step

Script: `_set_step.py`

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

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

7.3.3 Mesh Strategy

Script: `_set_mesh.py` (and submodules)

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

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

7.4 Physical Properties and Assembly

Package: `_modules/assigment/` (sic)

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

7.4.1 Material Definition

Script: `_set_material.py`

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

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

7.4.2 Section Management

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

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

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

7.4.3 Assembly Instantiation

Script: `_set_instance.py`

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

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

7.5 Geometry Construction Strategies

Package: `_modules/geometry/`

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

7.5.1 Strategy 1: T-Shape (Sim One)

Sub-package: `sim_one/`

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

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

7.5.2 Strategy 2: Milling Profile (Sim Two)

Sub-package: `sim_two/`

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

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

7.5.3 Integration: The Model Mixin

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

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

8 Contour Method Simulation Workflow

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

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

8.1 Workflow Orchestration

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

8.1.1 Phase 1: Simulation Setup and Execution

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

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

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

8.1.2 Phase 2: Result Conversion

Controlled by the `convertion_process` flag:

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

8.2 Abaqus Geometry Script (The Builder)

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

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

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

8.2.1 Execution Context and Parameter Injection

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

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

8.2.2 The `ContourAnalysis` Class

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

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

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

9 Residual Stress Analysis (RSA) Workflow

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

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

9.1 Workflow Architecture

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

9.1.1 Dependency Management

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

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

9.1.2 Geometry and Mesh Generation

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

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

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

9.1.3 Stress Mapping (The Bridge)

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

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

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

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

9.1.4 Injection and Solving

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

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

10 Residual Stress Analysis (RSA) Workflow

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

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

10.1 Workflow Orchestration

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

10.1.1 Dependency Management

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

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

10.1.2 Stress Mapping (The Bridge)

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

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

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

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

10.1.3 Injection and Solving

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

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

10.2 Abaqus Geometry Script (The RSA Builder)

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

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

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

10.2.1 The `ContourAnalysisExtended` Class

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

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

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

10.3 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 **??**.

10.3.1 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.

10.3.2 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.

10.3.3 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.

10.4 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.

10.4.1 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.

10.4.2 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 applies a scaling factor to the Young's Modulus:

$$E_{layer\_i} = \frac{E_{base}}{Scale^i} \tag{10.1}$$

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.

11 User's Guide: Step-by-Step Tutorial

11.1 Overview

This tutorial guides you through the complete Residual Stress Analysis workflow using the hybrid experimental-numerical framework.

Workflow Summary:

1. **Configuration (~20 min):** Define material properties and Design of Experiments (DOE) ranges.
2. **Contour Method Simulation (~10 min):** Reconstruct surface stresses from experimental data.
3. **Residual Stress Analysis (~10 min):** Simulate the stress redistribution during the cutting process.
4. **Visualization:** Analyze results using ParaView.

   Note: These time values are done considered one simulation with a mesh after the convergence of the result, which means 50 thousand elements.

Expected Outcome: By the end of this guide, you will have XDMF/HDF5 files representing the full 3D stress tensor field, ready for post-processing.

11.2 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 the config.
- **Python 3.10+:** Required for the pipeline orchestration.
- **Dependencies:** Install required packages:

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

- **Project Structure:** Ensure the `src/` folder matches the architecture defined in Chapter **??**.

11.3 Step 1: Simulation Configuration

Goal: Generate the master `config.json` file that controls all simulation parameters.

11.3.1 Action: Using the GUI

Navigate to the experiment data folder and launch the interface:

```
cd src/Exp_Data/s1_exp
python write_input_gui.py
```

11.3.2 Configuration Checklist

Fill in the fields as follows:

1. **Material Properties:**
   - *Elastic Modulus:* e.g., 210000 (MPa)
   - *Poisson Ratio:* e.g., 0.3
2. **Design of Experiments (DOE):** Define the search space for the sensitivity analysis.
   - *Mesh Range:* Min=0.6, Max=0.8, Step=0.1
   - *Length Range:* Min=50, Max=100, Step=25

   Note: The software creates a job for every combination (e.g., Mesh0.6-Len50, Mesh0.6-Len75...).
   Note: If you put something like Min=0.6, Max=0.6, Step=0.1, the software will break.
3. **Validation & Save:**
   - Click **"Save JSON"**: Confirm the success message.

11.3.3 Verification

Go to `src/Exp_Data/s1_exp/config/` and check if `config.json` was created. Open it with a text editor to ensure the paths (e.g., `"work_directory"`) match your machine's structure.

11.4 Step 2: Contour Method (CM) Simulation

Goal: Run the inverse calculation to reconstruct boundary conditions and solve for surface stresses.

11.4.1 Action: Execution

Run the main orchestrator from the project root:

```
python src/Simulations/cm_main.py
```

The terminal will display progress logs: `[INFO] Processing Case: Mesh-0_8--Length-50...`

11.4.2 Result Verification

Since the pipeline is automated, the system ensures data consistency before saving. Upon completion, you can locate the results in the `CM_Directory` defined in your configuration.

Generated Outputs:

1. **Data Files:** Navigate to the `/xdmf_hdf5_files/` folder. You will find the converted results for each DOE case:
   - `.xdmf` (Metadata for ParaView)
   - `.h5` (Heavy data storage)
2. **Execution Logs:** Review `summary_report.txt` to see the execution time for each simulation case.

11.5 Step 3: Residual Stress Analysis (RSA)

Goal: Map the CM results onto a new geometry (machining state) and simulate stress redistribution.

11.5.1 Action: Execution

Open `src/Simulations/rs_main.py` in your editor. Locate the `main()` call at the bottom and ensure the flags are set:

```
if __name__ == "__main__":
    # run_cma=False (Skip Step 2 if already done)
    # run_rsa=True  (Execute this step)
    main(run_cma=False, run_rsa=True)
```

Run the script:

```
python src/Simulations/rs_main.py
```

11.5.2 What is happening?

1. **Geometry Gen:** The script creates new '.inp' files for the RSA geometry.
2. **Mapping:** It uses a KDTree algorithm to interpolate stress tensors from the CM mesh to the RSA mesh.
3. **Injection:** It modifies the input files to add `*Initial Conditions, type=STRESS`.

11.5.3 Verification

Navigate to the `REA_Directory` (e.g., `C:/Simulation/Residual_Stresses_Analysis`).

- Look for the `Output` folder.
- Ensure that for each case, a `stress_input.txt` or similar CSV file exists (this proves mapping was successful).
- Check if the final `.odb` files are generated in the job folders.

11.6 Step 4: Visualizing Results

Goal: Interpret the results using ParaView.

11.6.1 Loading Data

1. Open **ParaView**.
2. File → Open → Navigate to your output folder (e.g., `CM_Directory/xdmf_hdf5_files/`).
3. Select a file, e.g., `Mesh-0_8--Length-50.xdmf`.
4. Click **"Apply"** in the left properties panel.

11.6.2 Analyzing Stresses

1. In the toolbar, locate the variable selector (usually shows "Solid Color").
2. Change it to **"S"** (Stress Tensor).
3. In the component selector (next to "S"), choose:
   - **S11:** Longitudinal Stress (typical interest for T-shapes).
   - **S33:** Normal Stress (to the cut plane).
   - **Mises:** Equivalent Von Mises stress.

11.6.3 Interpretation Hints

- **Red Zones (Positive):** Indicate Tensile Residual Stress.
- **Blue Zones (Negative):** Indicate Compressive Residual Stress.
- **Validation:** The stress perpendicular to the cut surface (S33) should be close to zero at the cut face (boundary condition enforcement).

11.7 Troubleshooting & Advanced Usage

11.7.1 Common Issues

Error: "Abaqus command not found"

The pipeline cannot find the solver. Edit `config.json` manually or via GUI and ensure `"abaqus_cmd"` points to the full path of your batch file (e.g., `C:/SIMULIA/Commands/abq2021.`

Simulation Hangs / Convergence Failure

Check the `.msg` file in the simulation directory.

- If "Too many increments": Increase `maxNumInc` in the GUI.
- If "Time increment too small": Your mesh might be too distorted. Try a larger `Mesh Size` in the DOE settings.

Missing HDF5 Files

This usually means the Abaqus Python extraction failed. Check if you have write permissions in the directory or if the disk is full.

11.7.2 Advanced: Customizing Geometry

To analyze a shape other than the standard T-part:

1. Edit `src/Exp_Data/s1_exp/mean_dim.py` to update the physical dimensions.
2. Run `attempt.py` manually once to verify the new geometry builds correctly in Abaqus CAE before running the full batch.