Task 1 – The intersector

Development Process

The task involved two areas unfamiliar to me: **Python CLI development** and **OpenCASCADE**.
I began by building a minimal CLI application that used OpenCASCADE, progressively adding functionality until I could perform the intersection operation.

To accelerate learning, I relied extensively on **AI assistance**, using it to explore, test, and refine approaches. Once I achieved a functional prototype and gained confidence with both technologies, I created a clean environment and developed the official version, focusing on clean architecture, maintainability, and extensibility.

Tools and Libraries

Conda

A reliable environment management tool. It was essential for installing the **OpenCASCADE** library, which is not easily managed by other tools.

Poetry

A modern packaging and dependency management system.
It manages Python dependencies (except `pythonocc-core`), handles versioning and metadata, and builds the distribution packages (`.whl` and `.tar.gz`).
I selected Poetry because it is user-friendly, PEP 517/518 and PEP 621 compliant, and combines multiple development features in one tool.
Alternatives include **Hatch** and **PDM**.

Click

A CLI management framework for Python. It provides argument parsing, help messages, subcommands, error handling, and validation, enabling clean and maintainable CLI applications.
An alternative is **Typer**, a newer library with similar goals.

Rich

A library for formatting console output.
Although optional, it enhances CLI applications by allowing styled messages, progress bars, and live updates. I used it mainly to improve readability and user experience.

Ruff

A modern all-in-one tool for linting, formatting, import sorting, and docstring checking.
It offers automatic fixes and excellent speed. While I am more familiar with **Pylint**, Ruff's simplicity and performance make it an attractive choice.

Black

A complementary formatter used alongside Ruff to ensure consistent and automatic code formatting.

Pytest

A lightweight and widely used testing framework that integrates easily with CI/CD pipelines and helps maintain clear and organized test structures.

Project Structure

The project follows a **standard layout**:
The main directory includes configuration, metadata, and documentation files.
The `src/` folder contains the main package, and the `tests/` folder holds test files corresponding to the main modules.

Main Package

The entry point is `cli.py`, which handles all CLI commands.
Other modules are organized into subfolders according to functionality — primarily `operations` and `utils`.

Folder `operations`

Contains the application's geometric operations. Currently, it includes only the **intersect** operation.

Folder `utils`

Includes reusable helper functions:

- **file_handler.py** – File I/O operations
- **parsing.py** – Input parsing
- **visualize.py** – Shape visualization
- **logger.py** – Logging setup

`__init__.py` Files

Although not required under PEP 420, I included them in every folder for readability and to prevent naming conflicts.

Versioning

The project follows **PEP 440** for versioning and was released as **1.0.0**, representing its first stable version.
Poetry manages version validation and automatically applies it to the generated packages.
Tools like **Commitizen** can automate versioning based on commit messages, though they are more suitable for larger projects.

Version Control and Repository Content

The project uses **Git** for version control.
Generated files are excluded via `.gitignore`, except for **poetry.lock**, which ensures reproducible environments.

A standard branching model is followed:

- **main** – stable releases
- **develop** – active development
- **feature/*** – isolated new features merged into `develop` after review

In this single-developer demo, I worked directly on the `develop` branch for simplicity.

Logging

Logging serves both users and developers.
The application uses **Rich** for styled console messages and the Python **logging** module for structured logs.
Currently, both output to the console for simplicity.
A more advanced setup would include file logging for user mode and configurable log levels via a dedicated configuration file.

Documentation

Two README files are included:

- **USER_GUIDE.md** – Describes installation, usage, and user instructions.
- **README.md** – Aimed at developers, explaining environment setup and project structure.

Maintaining both files from the beginning of a project helps ensure clarity for all contributors.

Licensing

The project is distributed under the **MIT License**, allowing free use and modification, including commercial applications.
The license text is included in the `LICENSE` file.

A separate **NOTICE** file lists third-party packages and their respective licenses, ensuring compliance and transparency.

Visualization

After performing an intersection, the program visualizes the original shape and the intersected section if applicable.
Although this feature adds complexity and reduces performance, it was implemented for experimentation and demonstration purposes.
A future improvement could include a flag to enable or disable visualization.

Use of AI

AI tools were heavily used throughout development, primarily for code generation. But very carefully and with a lot of babysitting.

Some basic prompts:

- Give me a simple CLI application that uses OpenCASCADE.
- Give me a function that read a stp file (same for write)
- How to create a plane defined by point normal form.
- How to intersect a shape with a plane
- Give me unittest for that method