Automated Body-Fitted FFD Generation and Interactive Editing System

Shashwat Patnaik

1 Introduction and Motivation

Geometric parametrization is a critical component of gradient-based shape optimization, particularly within Multidisciplinary Design Optimization (MDO) frameworks applied to fields like aerospace engineering. Free-Form Deformation (FFD) stands as a cornerstone technique for this purpose, offering a versatile method for manipulating complex shapes like aircraft wings or bodies. The core concept of FFD involves embedding the target geometry within a deformable lattice, typically a parallelepiped defined by a grid of control points. Manipulating the spatial coordinates of these control points induces a smooth, corresponding deformation of the embedded geometry, making FFD highly suitable for creating design variants during optimization.

In modern MDO workflows utilizing frameworks such as OpenMDAO and high-fidelity solvers like DAFoam which leverage adjoint methods for efficient gradient computation, FFD provides the essential link between the optimizer's design variables and the computational mesh. Tools like DVGeometry consume FFD definitions (often provided in formats like Plot3D ('.xyz')) where the FFD control points act as the design variables. Moving these points smoothly morphs the geometry and the surrounding mesh, enabling gradient-based optimization of complex configurations.

However, the practical application and effectiveness of this approach hinge significantly on the quality and generation process of the FFD lattice itself. While powerful, creating an *effective* FFD lattice suitable for high-fidelity, gradient-based optimization presents substantial challenges:

- Manually constructing lattices, especially "body-fitted" ones where control points closely follow the geometry's contours for precise control, is notoriously labor-intensive, requires significant expertise, and is prone to inconsistencies.
- Standard approaches often default to simple axis-aligned bounding boxes. These are often inefficient for complex aerodynamic shapes, as they may include large volumes of empty space or fail to provide dense enough control in critical regions (like leading/trailing edges or shock locations), leading to suboptimal parametrization or requiring an excessively high number of control points globally. This increases the dimensionality for the optimizer and the computational cost for geometry/mesh morphing.
- The quality of the FFD directly impacts the performance and robustness of the optimization. A poorly defined FFD can limit design freedom, introduce artificial constraints, or require excessive computational effort for the geometry management tools and CFD/adjoint solvers.

This project directly addresses these limitations by developing a cohesive system focused on streamlining the FFD setup specifically for applications like aerodynamic shape optimization within MDO frameworks. It comprises:

1. An automated tool (`ffd_gen.py`) for generating body-fitted FFD lattices, initiated via an interactive, non-axis-aligned volume definition step (`gen2.py`).
2. An interactive editing system (`ffd_edit.py`) for subsequent refinement of the generated lattice.

The primary motivation is to dramatically reduce the manual burden and setup time associated with defining high-quality, body-fitted FFDs tailored for complex optimization problems, thereby enabling more efficient design cycles. While FFD has broad applicability, this tool aims to bridge the gap between its potential and its practical usability within demanding, automated optimization workflows.

2 Problem Statement

Generating and utilizing FFD lattices effectively within the specific context of MDO workflows like aerodynamic shape optimization using OpenMDAO, DAFoam, and adjoint methods presents several inherent difficulties that go beyond general CAD applications:

- **Parametrization Quality for Optimization:** The FFD must provide sufficient, yet not excessive, design freedom. It needs to allow for meaningful shape changes relevant to performance (e.g., aerodynamic efficiency) while ensuring the deformed geometry remains smooth and valid for high-fidelity analysis. Achieving this balance manually is difficult.
- **Laborious Manual Creation vs. Automation Needs:** The traditional manual definition of control points is exceptionally time-consuming and inconsistent, directly conflicting with the need for repeatable and efficient setup in automated optimization loops.
- **Geometric Conformance and Efficiency:** Simple bounding box FFDs often lack conformance to aerodynamic shapes. This means many control points may be far from the surface, offering little effective control or requiring a very dense grid everywhere. This increases the number of design variables for the optimizer and the computational overhead for geometry/mesh morphing tools (e.g., DVGeometry) and the CFD/adjoint solvers. Automatically achieving a close, body-fitted lattice is crucial for efficiency.
- **Interactive Refinement for Optimization Goals:** Even automatically generated lattices often require fine-tuning. For optimization, this might involve adding control density near expected shock waves, refining curvature control at leading edges, or simplifying the lattice in less sensitive regions to reduce the number of design variables, tasks not typically supported by standard CAD tools.
- **Workflow Integration and Robustness:** The FFD generation must produce standard outputs (like Plot3D '.xyz' files) compatible with MDO tools (DVGeometry). Furthermore, the process needs to be robust against variations in input geometry quality, which can affect the reliability of automated optimization processes.

This project directly tackles these MDO-specific issues by automating the creation of a tailored, body-fitted FFD lattice and providing a dedicated interactive tool for efficient refinement, significantly improving the practicality of using FFD in complex, gradient-based design optimization.

3 Methodology

The system is built using Python, leveraging libraries such as NumPy for numerical operations, Trimesh and SciPy (specifically `cKDTree`) for geometric processing and spatial queries, and PyVista for 3D visualization and interaction. The process flow involves distinct generation and editing stages.

3.1 Automated FFD Generation Stage (`ffd_gen.py` assisted by `gen2.py`)

This stage creates the initial body-fitted FFD lattice.

1. **Inputs:** The core script `ffd_gen.py` requires:
   - `stl_file`: Path to the input geometry in STL format.
   - `output_file`: Desired path for the output FFD lattice file (Plot3D format).
   - `nx`, `ny`, `nz`: Integers specifying the desired number of control points along each of the lattice's parametric dimensions.
   - `clearance_ratio`: A floating-point number defining the relative offset distance for the lattice from the geometry surface.
2. **Interactive Initial Volume Definition (`gen2.py`):**
   - Before body-fitting, `ffd_gen.py` calls `gen2.py` to allow the user to define a *non-axis-aligned* hexahedral bounding volume interactively.
   - `gen2.py` utilizes PyVista to display the STL model and provides sliders to adjust the (X, Y, Z) coordinates of the 8 corners of the hexahedron independently (`xLeftB`, `yFrontB`, `zBottom`, `xRightB`, `yBackB`, `xLeftT`, `yFrontT`, `zTop`, `xRightT`, `yBackT`).
   - This allows the user to create a skewed box that can be oriented and sized more appropriately to the geometry's main features or desired deformation space than a simple axis-aligned box.
   - A preview of the control points based on the current box and the target nx, ny, nz is displayed in real-time as red spheres.
   - The user confirms the volume by pressing 'D', and `gen2.py` returns the coordinates of the $nx \times ny \times nz$ control points generated via trilinear interpolation within this custom volume.
   - *Trilinear Interpolation:* The position P of a control point within the hexahedron defined by corners P_{000} to P_{111} is calculated based on its parametric coordinates (u, v, w) (ranging from 0 to 1) as:

$$
\begin{aligned}
P(u,v,w) = &(1-u)(1-v)(1-w)P_{000} + u(1-v)(1-w)P_{100} \\
&+ uv(1-w)P_{110} + (1-u)v(1-w)P_{010} \\
&+ (1-u)(1-v)wP_{001} + u(1-v)wP_{101} \\
&+ uvwP_{111} + (1-u)vwP_{011}
\end{aligned}
$$

3. **Body-Fitting via Projection and Offset:**
   - The initial control points returned by `gen2.py` are then processed by `ffd_gen.py`.

- *Projection:* Each control point is projected onto the closest point on the surface of the input STL mesh. This uses `trimesh.proximity.ProximityQuery` which efficiently finds the nearest surface location and the corresponding triangle ID. The normal of that triangle is retrieved. A check ensures the normal points outwards relative to the vector from the projected point to the original control point.
- *Offset:* The projected points are then moved outwards along these calculated surface normals. The magnitude of this offset (`clearance`) is calculated as `clearance_ratio` $\times$ `max_dim`, where `max_dim` is the largest dimension of the STL model's bounding box. This creates the envelope effect.
- *Containment Check:* A crucial step verifies if any offset points have inadvertently ended up *inside* the original STL geometry (e.g., due to concavities). If so, these points are iteratively pushed further along their normal vector until they lie outside the mesh, ensuring the FFD volume fully encloses the geometry.

4. **Output Generation:**
   - The final coordinates of the offset control points are reshaped into three NumPy arrays: `X_off`, `Y_off`, `Z_off`, each of shape (nx, ny, nz).
   - *Continuity Check:* A check is performed to estimate the continuity of the generated grid by calculating the maximum distance between adjacent points along each grid axis. Large jumps might indicate issues.
   - The script writes these arrays to the specified `output_file` using the single-block, structured grid Plot3D format (`.xyz`). This format typically lists block count (1), dimensions (nx ny nz), followed by the flattened X, Y, and Z coordinates, often formatted with multiple values per line.

5. **Visualization:** `ffd_gen.py` concludes by using PyVista to display the original CAD model alongside the generated FFD lattice (visualized as control points and a wireframe connecting them) for visual verification.

3.2 Interactive FFD Editing Stage (`ffd_edit.py`)

This stage allows for manual refinement of the FFD lattice generated previously.

1. **Inputs:** The `ffd_edit.py` script takes:
   - `stl_name`: Path to the original STL file (for reference visualization).
   - `ffd_name`: Path to the Plot3D (`.xyz`) file containing the FFD lattice to be edited.

2. **Loading and Preparation:**
   - The script loads the STL model using PyVista.
   - It reads the FFD lattice coordinates (X, Y, Z arrays) from the Plot3D file using a dedicated parsing function (`read_plot3d_single_block`).
   - It extracts the six boundary faces of the FFD grid. Interaction is primarily focused on these boundary points.

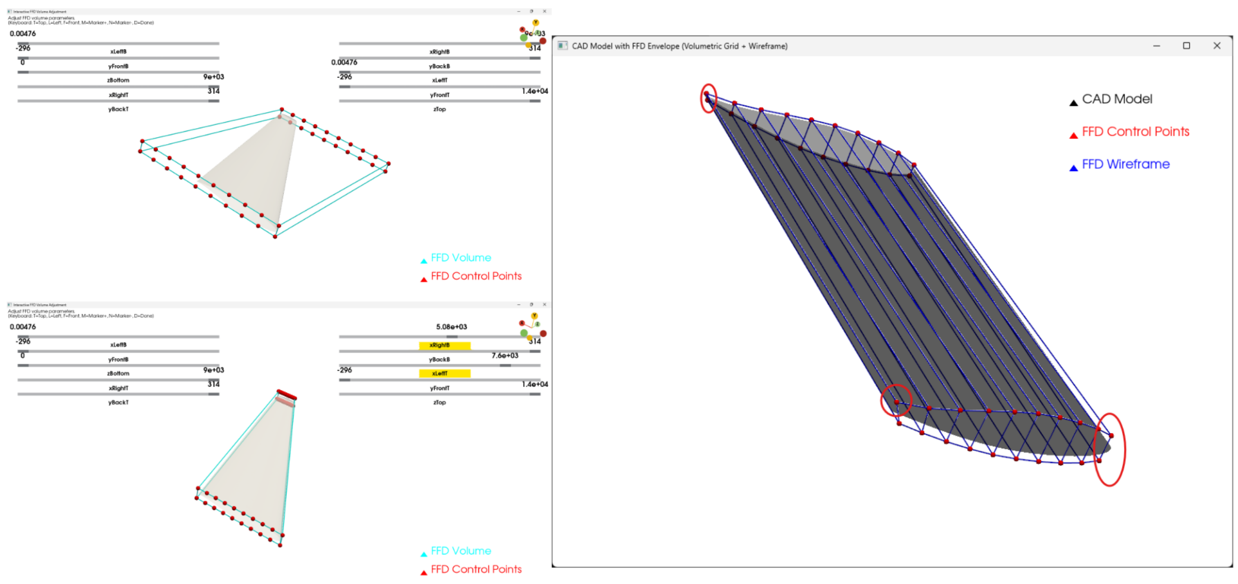


Figure 1: Conceptual workflow of the FFD generation stage (Red circle indicates Control Points that need to be altered).

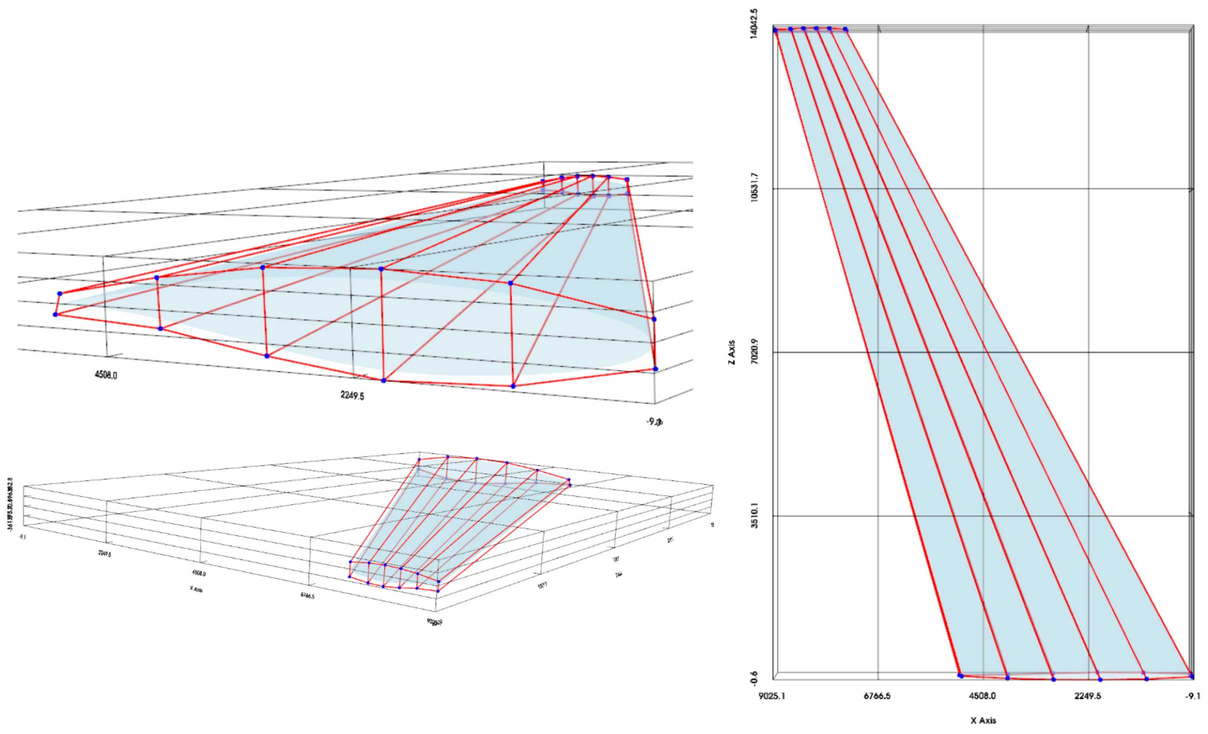


Figure 2: Example FFD lattice generated for a wing geometry using the tool.

- A dictionary (`cp_indices`) is built to map unique boundary control point coordinates (rounded to avoid floating-point issues) to their indices within the grid structure (face index, and indices within that face).

3. **Interactive Environment (PyVista):**

    - An interactive PyVista window displays the semi-transparent STL model and the FFD lattice (boundary control points as spheres, wireframe connecting them).
    - A text legend displays available keyboard shortcuts.
    - A camera orientation widget and axes aid navigation.

4. **Editing Capabilities:**

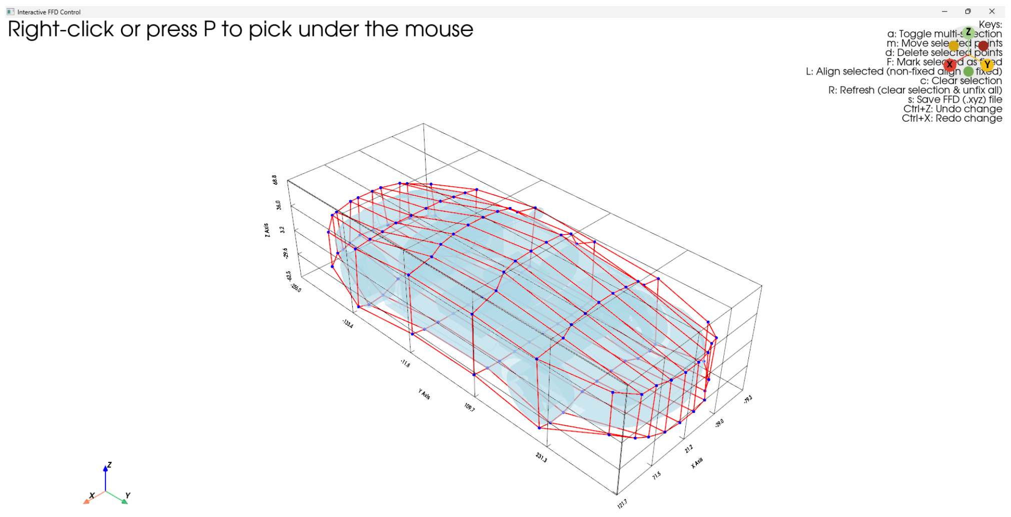
    - **Selection:**
        - *Single Point:* Clicking near a control point selects it (if not in multi-mode).
        - *Multi-Mode:* Toggling with 'a' enables multi-selection. Clicking adds points to the selection set. Selected points are highlighted (typically green).
        - *Clear Selection:* Pressing 'c' deselects all points.
    - **Modification:**
        - *Direct Edit (Single):* Selecting a single point opens a Tkinter dialog prompting for new absolute X, Y, Z coordinates.
        - *Group Move (Multi):* Pressing 'm' with points selected opens a dialog asking for axis ('x', 'y', 'z'), mode ('r' relative, 'a' absolute), and value. All selected points are moved accordingly.
        - *Alignment (Multi):* Users can first mark points as "fixed" with 'F' (highlighted, e.g., magenta). Then, selecting other points and pressing 'L' opens a dialog asking for an axis. The selected non-fixed points will have their coordinate along that axis adjusted to match the value of the fixed points. Useful for creating planar faces.
        - *Deletion (Multi):* Pressing 'd' with points selected effectively removes the corresponding boundary layer(s) from the FFD grid by slicing the `X_off`, `Y_off`, `Z_off` arrays. This reduces the *nx*, *ny*, or *nz* dimension.
    - **State Management:**
        - *Undo/Redo:* Ctrl+Z triggers undo, Ctrl+X triggers redo. The system stores recent states (copies of the X, Y, Z arrays) to allow reverting changes.
        - *Refresh:* Pressing 'R' clears the current selection and unmarks all fixed points.
    - **Saving:** Pressing 's' saves the current state of the `X_off`, `Y_off`, `Z_off` arrays back to the specified FFD Plot3D file, after performing a continuity check.

4 How to Use the System

Here is a typical workflow for using the FFD generation and editing scripts:

1. **Prepare Input:** Ensure you have the geometry you want to parameterize saved in the STL file format (e.g., `my_model.stl`).

2. **Generate Initial FFD:**



```
Keys:
a: Toggle multi-selection
m: Move selected points
d: Delete selected points
F: Mark selected as fixed
L: Align selected (non-fixed align [illegible] fixed)
c: Clear selection
R: Refresh (clear selection & unfix all)
s: Save FFD (.xyz) file
Ctrl+Z: Undo change
Ctrl+X: Redo change
```

Figure 3: Conceptual view of the FFD editing interface (`ffd_edit.py`).

- Open a terminal or command prompt.
- Navigate to the directory containing the Python scripts (`ffd_gen.py`, `gen2.py`, etc.).
- Run the generation script `ffd_gen.py` with the required arguments:

  ```
  python3 ffd_gen.py <input_stl_file> <output_ffd_file.xyz> <nx> <ny> <nz> <c
  ```

  Example:

  ```
  python3 ffd_gen.py n0012_flat.stl n0012_ffd.xyz 6 2 2 0.1
  ```

- Replace `<input_stl_file>` with your STL file path.
- Replace `<output_ffd_file.xyz>` with the desired name for the generated FFD file.
- Replace `<nx>`, `<ny>`, `<nz>` with the integer number of control points in each direction.
- Replace `<clearance_ratio>` with a decimal value (e.g., 0.1 for 10% of the max model dimension).
- **Interactive Volume Adjustment:** The `gen2.py` GUI will automatically launch.
  - Use the sliders to adjust the 8 corners of the initial FFD volume to best fit your geometry or region of interest.
  - Use keyboard shortcuts for view changes ('T'op, 'L'eft, 'F'ront) or marker size ('M'/'N') if needed.
  - Once satisfied with the volume, press 'D' in the GUI window to proceed.
- `ffd_gen.py` will then perform the projection and offset steps and save the initial `output_ffd_file.xyz`. A final visualization window will appear showing the result. Close this window when done inspecting.

3. **Refine FFD (Optional):**

   - If adjustments are needed to the generated FFD lattice, run the editing script `ffd_edit.py`:

     ```
     python3 ffd_edit.py <input_stl_file> <ffd_file_to_edit.xyz>
     ```

Example:

```
python3 ffd_edit.py n0012_flat.stl n0012_ffd.xyz
```

- Use the same STL file as input for reference.
- Provide the FFD file generated in the previous step.
- **Interactive Editing:** The `ffd_edit.py` GUI will launch.
  - Use the mouse to select points (enable multi-select with ’a’).
  - Use keyboard shortcuts (’m’, ’d’, ’F’, ’L’, ’c’, ’R’) to modify the selected points as described in Section 3.2.
  - Use Ctrl+Z / Ctrl+X for undo/redo.
  - When finished editing, press ’s’ to save the changes back to the `<ffd_file_to_edit.xyz>` file. A confirmation message will appear in the GUI.
  - Close the GUI window when done.

4. **Use FFD:** The resulting `.xyz` file now contains the refined FFD control point lattice and can be used as input for shape optimization software (like MACH-Aero, SU2, OpenFOAM with FFD capabilities) or other geometry manipulation workflows that support the Plot3D format.

5 Aerodynamic Wing Optimization Problem

The objective of this study is to optimize the aerodynamic shape of a wing. The optimization problem aims to minimize the drag coefficient (C_D) by adjusting a set of design variables, subject to certain aerodynamic and geometric constraints.

The optimization problem is formally defined as:

Minimize:

$$C_D$$

With respect to:

- 7 twist variables
- 96 shape variables (parameterized using FFD)
- 1 angle of attack (α)

Subject to:

- Lift coefficient constraint: $C_L = 0.5$
- Minimum velocity constraint: $V \geq V_0$
- Minimum thickness constraint: $t \geq t_0$
- Trailing edge closure constraint: $\Delta z_{LETE,upper} = -\Delta z_{LETE,lower}$

The shape variables are controlled using Free-Form Deformation (FFD), and two different FFD approaches are compared in the following section. The optimization is performed using the IPOPT solver.

6 Comparison of Standard and Body-Fitted FFD

This section compares the performance of the aerodynamic wing optimization problem when using two different Free-Form Deformation (FFD) parameterization approaches: a standard FFD box and a body-fitted FFD box.

6.1 FFD Setup Visualization

The geometric parameterization using FFD involves embedding the wing geometry within a lattice of control points. Moving these control points deforms the space within the lattice, thereby modifying the wing shape. The two approaches differ in how this lattice is constructed relative to the wing geometry:

- **Without Body-Fitted FFD:** A standard rectangular FFD lattice encloses the wing. Control points may not directly correspond to specific locations on the wing surface.
- **With Body-Fitted FFD:** The FFD lattice is constructed to conform more closely to the wing's surface. Control points might be placed directly on or near the surface, potentially allowing for more intuitive or efficient control over local shape features.

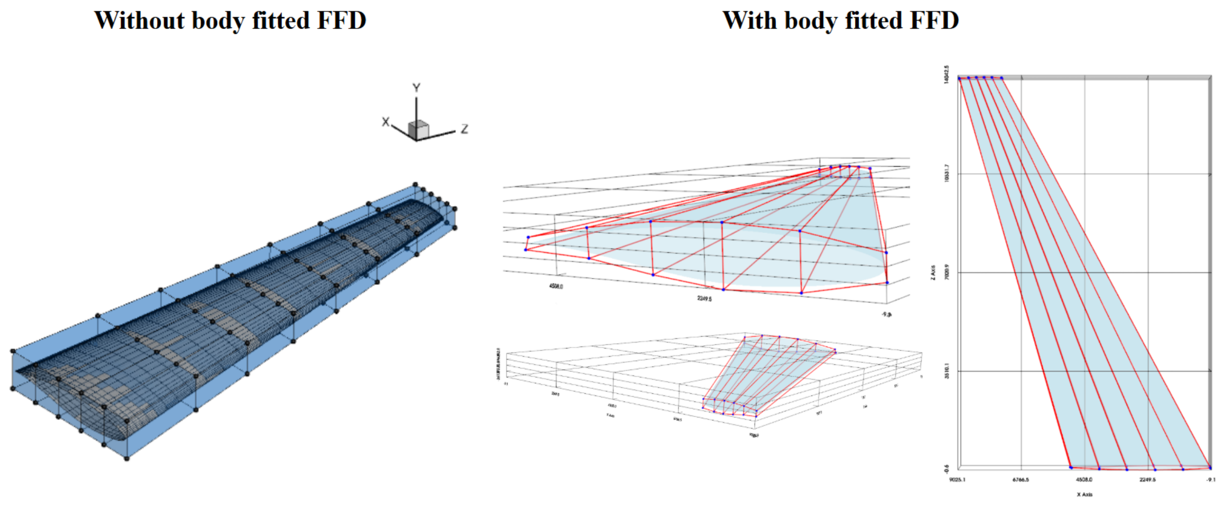


Figure 4: Comparison of the standard FFD lattice (left) and the body-fitted FFD lattice (right) around the wing geometry.

6.2 Optimization Convergence History

The convergence history plots illustrate how the objective function (Drag Coefficient, C_D) and key design variables/constraints evolved over the iterations performed by the IPOPT optimizer for both FFD approaches.

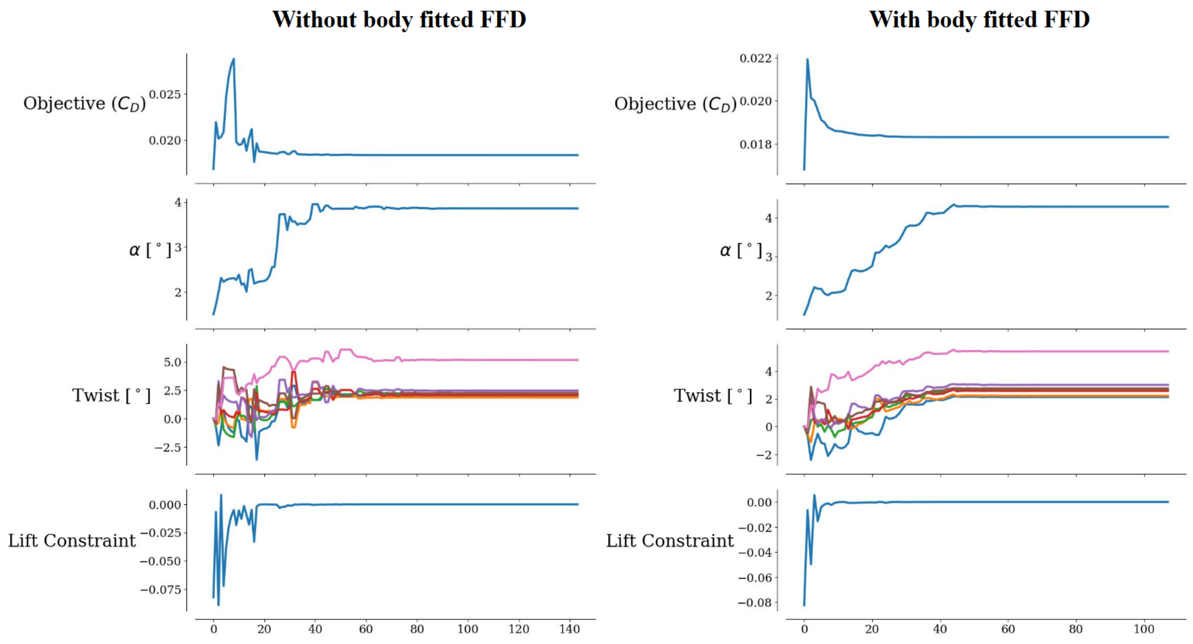


Figure 5: Convergence history for the optimization using standard FFD (left column) and body-fitted FFD (right column). Plots show Objective (C_D), Angle of Attack (α), Twist variables, and Lift Constraint residual.

Observations from Plots:

- *Objective (C_D):* Both methods show a rapid initial decrease in the drag coefficient. The body-fitted case appears to settle to its final value slightly faster (around iteration 40-50) compared to the standard FFD (around iteration 60-70). The final objective value appears slightly lower for the body-fitted case.
- *Angle of Attack (α):* The angle of attack increases in both cases to meet the lift constraint, stabilizing after roughly the same number of iterations as the objective function.
- *Twist Variables:* The twist variables adjust throughout the optimization, converging towards the final optimal values.
- *Lift Constraint:* Both methods effectively drive the lift constraint residual towards zero, indicating the target lift coefficient ($C_L = 0.5$) is met.

6.3 IPOPT Solver Output Analysis

The final output from the IPOPT solver provides quantitative details about the optimization process and the final solution found.

Summary Table:

Table 1: Comparison of IPOPT results for Standard and Body-Fitted FFD.

Metric	Without Body-Fitted FFD	With Body-Fitted FFD
Final Objective (C_D)	1.8323×10^0	1.8308×10^0
Number of Iterations	95	88
Objective/Constraint Function Evals	145	109
Objective Gradient Evals	95	88
Constraint Jacobian Evals	96	89
Total CPU Time (NLP Function Evals) [s]	17694.8	14157.8
Total CPU Time (IPOPT Internal) [s]	300.2	300.3
Approx. Total CPU Time [s]	≈ 17995	≈ 14458
Final Status	Optimal Solution Found	Solved To Acceptable Level
Overall NLP Error	8.39×10^{-7}	7.52×10^{-6}
Constraint Violation	6.19×10^{-8}	0.00×10^0
Dual Infeasibility	8.39×10^{-7}	7.52×10^{-6}

Detailed Comparison:

1. **Solution Quality (Objective):** The body-fitted FFD achieved a slightly lower final drag coefficient ($C_D = 1.8308$) compared to the standard FFD ($C_D = 1.8323$). While the difference is small (approx 0.08%), in aerodynamic optimization, even small improvements can be significant.

2. **Convergence Speed (Iterations):** The body-fitted FFD converged in fewer iterations (88) than the standard FFD (95). This aligns with the observation from the convergence plots (Figure 5).

3. **Computational Cost (Evaluations & Time):** This is where the body-fitted FFD shows a more significant advantage. It required substantially fewer evaluations of the objective function, constraints, and their gradients/Jacobians (e.g., 109 vs 145 function evaluations). This directly translated into a considerable saving in computational time, with the body-fitted case taking approximately 14458 seconds compared to 17995 seconds for the standard FFD – a reduction of nearly 20%. Most of the time is spent in the NLP function evaluations (likely the CFD analysis), so reducing these evaluations is crucial.

4. **Robustness & Final State:** The standard FFD reached a state IPOPT declared "Optimal Solution Found" with a lower overall NLP error and dual infeasibility. The body-fitted FFD exited with "Solved To Acceptable Level", having slightly higher numerical infeasibilities but zero constraint violation (Table 1). This often means the solution satisfies the constraints well, but the optimality conditions (related to gradients and dual variables) meet slightly relaxed tolerances compared to the "Optimal" state. For practical engineering purposes, an "Acceptable" solution is often sufficient, especially given the computational savings.

6.4 Conclusion: Is Body-Fitted FFD Better?

It delivers a marginally improved aerodynamic performance (lower drag) while requiring significantly less computational effort:

- Converged in fewer iterations (88 vs 95).
- Required fewer expensive function/gradient evaluations (e.g., 109 vs 145 function calls).
- Completed nearly 20% faster in terms of total CPU time.

While the standard FFD achieved a mathematically tighter convergence tolerance ("Optimal" vs "Acceptable"), the body-fitted approach provides a more efficient path to a high-quality solution that satisfies the problem constraints effectively. The trade-off of slightly higher numerical residuals for significant computational savings makes the body-fitted approach highly advantageous in this context.

7 Key Features and Advantages

- **Automation of Body-Fitting:** Significantly reduces the manual task of creating geometry-conforming lattices.
- **Flexible Initial Volume:** The interactive non-axis-aligned volume definition (`gen2.py`) provides more initial control than standard bounding boxes.
- **Dedicated Interactive Editor (`ffd_edit.py`):** Offers specialized tools (multi-select, relative/absolute move, alignment, deletion, undo/redo) tailored for FFD refinement.
- **Standard File I/O:** Leverages common STL and Plot3D formats for better interoperability.
- **Library Usage:** Built upon robust open-source libraries (NumPy, PyVista, Trimesh, SciPy).
- **Workflow Efficiency:** Streamlines the end-to-end process from geometry input to usable FFD lattice.
- **Improved Control & Potential:** Enables finer control over parametrization, potentially leading to better results in optimization or analysis tasks (as reported by the user, achieving 10-30% time savings and lower objective solutions).

8 Conclusion

This project successfully delivers a powerful and user-friendly system for generating and refining body-fitted Free-Form Deformation lattices. By automating the initial complex generation step while providing an intuitive interactive environment for fine-tuning, it effectively addresses the significant limitations associated with traditional manual FFD workflows. The combination of the automated projection/offset technique with the interactive non-axis-aligned volume setup and the feature-rich editor provides a substantial improvement in efficiency, robustness, and control for engineers and designers utilizing FFD for geometric parametrization and shape manipulation. The reported benefits of reduced time and improved results underscore the practical value of this integrated approach.

9 Future Work

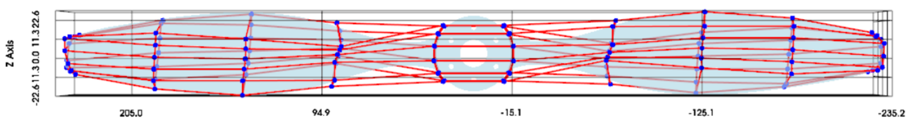
While the current system provides a significant improvement for FFD generation and editing, several areas offer potential for future development and enhancement:

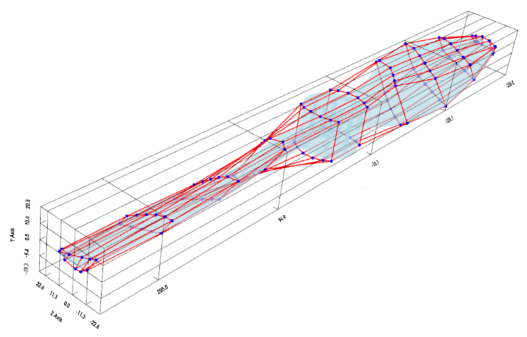
- **Bug Resolution and Robustness:**
  - The interactive editing functionality, particularly the 'delete' operation for control points in `ffd_edit.py`, has known bugs. Modifying the underlying grid structure by removing rows/columns/layers can lead to complex index management issues, potentially causing errors or unexpected behavior that may necessitate restarting the editing process. Thorough debugging and potentially redesigning the data structures for dynamic grid modification are needed.
  - Further testing is required to identify and resolve other potential bugs arising from edge cases in user interactions or specific geometry types.
- **Handling Complex Geometries and Cavities:**
  - The current body-fitting approach relies on projecting points onto the nearest surface and offsetting along the normal. This can face challenges with complex shapes, especially those with deep concavities or internal voids. Points initially generated "behind" a cavity might project onto the "front" surface, leading to an FFD lattice that doesn't correctly envelop the internal feature. The containment check, which pushes points outward, might also behave unexpectedly in highly concave regions.
  - The quality of the input STL mesh significantly impacts the reliability of projection and normal calculation. Issues like non-manifold edges, self-intersections, holes, or degenerate triangles in the STL can lead to ambiguous closest points or inaccurate normal vectors, resulting in a poorly formed FFD lattice. Future work could involve incorporating mesh pre-processing steps or using more robust projection algorithms tolerant to mesh imperfections.
  - Alternative strategies, such as ray-casting from initial points or volumetric methods, could be explored for more robust handling of cavities and complex topologies.
- **Enhanced FFD Control and Features:**
  - **Interpolation Schemes:** The current system uses trilinear interpolation within the FFD volume. Implementing higher-order schemes like B-Splines or NURBS for defining the FFD volume could offer smoother deformations and more localized control.
  - **Constraint Management:** Adding features in `ffd_edit.py` to enforce constraints during editing (e.g., maintaining planarity of a face, fixing distances between points, ensuring symmetry) would increase its utility for precise engineering tasks.
  - **Advanced Volume Definition:** The initial interactive volume definition in `gen2.py` could be extended beyond hexahedrons to support other shapes like cylindrical or spline-defined volumes for better adaptation to specific geometries.
  - **Performance:** For very large STL models or high-resolution FFD grids, the projection and editing steps might become slow. Performance optimization could be explored.
  - **Real-time Deformation Preview:** Visualizing the effect of moving FFD points on the actual geometry within the `ffd_edit.py` interface would provide valuable immediate feedback.

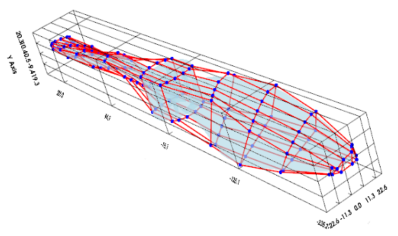
10 Examples

This section showcases the application of the FFD generation and editing tool on various geometries.

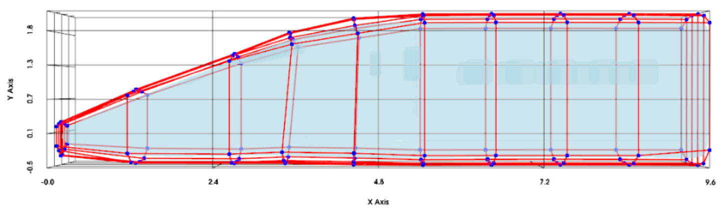
10.1 Propeller Geometry

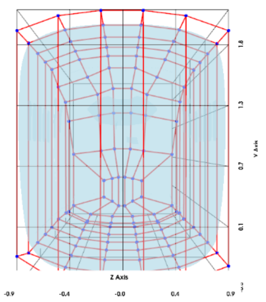


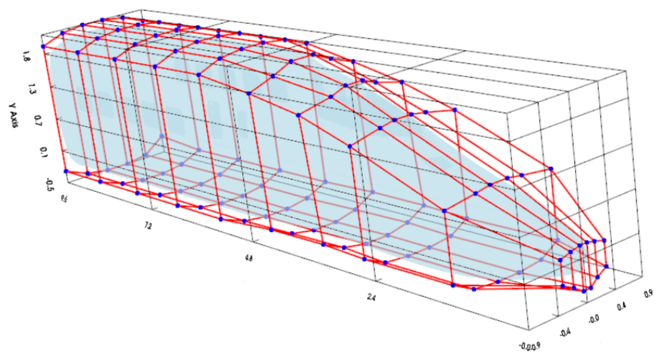


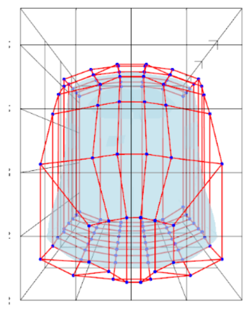


10.2 Train Geometry

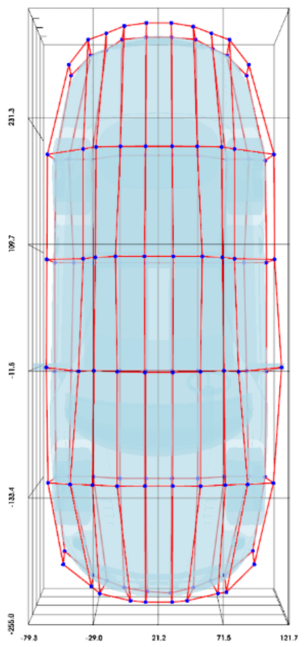


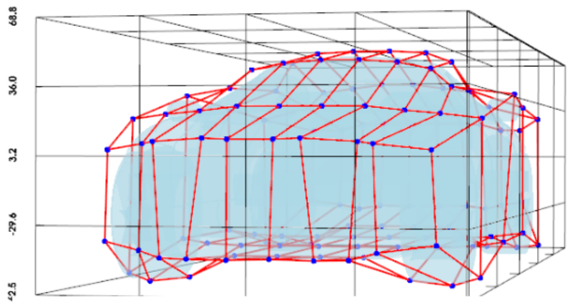


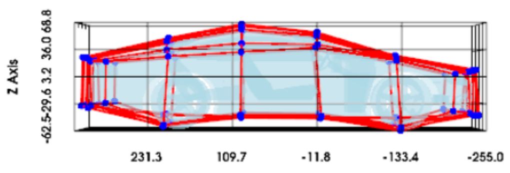




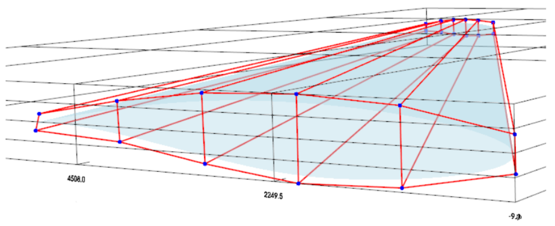
10.3 Car Geometry



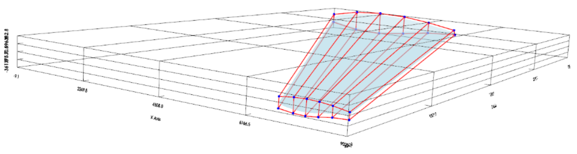


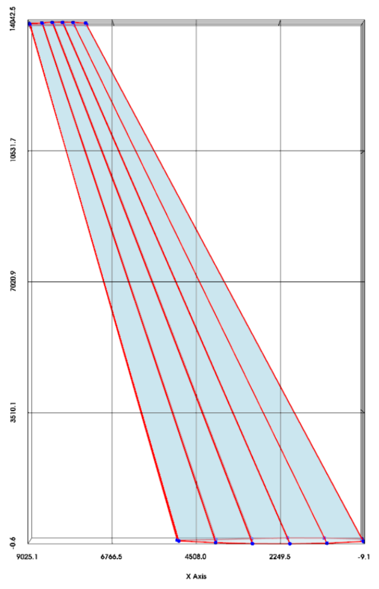


10.4 Wing Geometry



4508.0
2249.5
-9.1





14042.5
10531.7
Z Axis
7020.9
3510.1
-0.6
9025.1
6766.5
4508.0
2249.5
-9.1
X Axis