Möbius Strip Modeling – Write-Up

By: Saishma Chennam

◈ How I Structured the Code:

The Python script is structured around an **object-oriented approach** for modularity and clarity:

1. **MobiusStrip class** encapsulates all logic:
   - Accepts radius, width, and resolution as parameters.
   - Initializes a mesh grid using numpy.meshgrid.
   - Computes 3D points (x, y, z) using the standard Möbius strip parametric equations.
   - Methods include:
     - plot() – for 3D surface visualization using matplotlib with color mapping.
     - compute_surface_area() – estimates surface area numerically.
     - compute_edge_length() – computes total edge length along the boundary.
2. **Main block (if __name__ == "__main__")**:
   - Uses default (static) parameters for polished output.
   - Includes optional input (commented) to show generalization capabilities.
   - Displays key outputs and renders a 3D visualization.

◈ How I Approximated Surface Area:

The surface area is numerically approximated using the **magnitude of the cross product** of the partial derivatives of the parametric surface:

- For a parametric surface r(u,v), the area is:

  $$A=\iint |(\partial r/\partial u)\times(\partial r/\partial v)|\, dudv$$

- I computed these partial derivatives symbolically in NumPy for:
  - $\partial r/\partial u$ and $\partial r/\partial v$
- Then, using vector cross product and norm, I integrated the resulting values using numpy.trapz.

This gives a **high-resolution numerical approximation** of the surface area.

🔷 How I Approximated Edge Length:

The edge is defined where v=±w/2

I used:

- A fine sampling of the edge path by sweeping u from 00 to 4π4\pi
- Computed (x, y, z) positions along this path
- Used np.diff() to compute segment vectors and np.linalg.norm() to sum Euclidean distances between them.

This gives the **total edge length** of the Möbius strip.

🔷 Challenges Faced:

- Ensuring the **visual accuracy** of the Möbius strip with the right combination of width and resolution was tricky. A higher resolution (n = 500) and small width (w = 0.05) helped maintain the expected thin and smooth appearance.
- Handling the surface area calculation required careful use of **partial derivatives** and correct application of **numerical integration** techniques.
- Visualizing color gradients across the 3D surface in a visually appealing way was addressed using matplotlib's colormap normalization.

Final Thoughts:

This project was a great exercise in combining **3D parametric modeling**, **numerical integration**, and **visualization**. The code is modular, human-readable, and allows both static and dynamic usage, making it ready for further extensions (e.g., animation, interactivity).