Foldable Robotics Manufacturing Workflow

RAS 557 Foldable Robotics

This notebook implements the complete manufacturing workflow for laser-cut laminate fabrication.

All dimensions are in CENTIMETERS (cm)

Workflow Overview

1. Read EXACT DXF geometry (body outline and hinge slot shapes)
2. Build five-layer laminate structure (rigid-adhesive-flexible-adhesive-rigid)
3. Generate manufacturing files for laser cutting

1. Setup and Imports

```
# Install required packages if needed
# !pip install shapely matplotlib ezdxf numpy --break-system-packages -q
```

```
import os
import math
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon as MplPolygon
from matplotlib.collections import PatchCollection
import ezdxf
import foldable_robotics.dxf as frd
import foldable_robotics as fr
import foldable_robotics.manufacturing as frm
from foldable_robotics.layer import Layer
from foldable_robotics.laminate import Laminate
import foldable_robotics.parts.castellated_hinge2 as frc
from shapely.geometry import Polygon, MultiPolygon, box, Point, LineString
from shapely.ops import unary_union
from shapely import affinity

%matplotlib inline
plt.rcParams['figure.figsize'] = [14, 8]
plt.rcParams['figure.dpi'] = 100

print("Imports successful!")
```

```
Imports successful!
```

Import Libraries and Configure Plotting

In this section, we outline the essential procedures for importing necessary libraries and configuring the plotting environment for data visualization and geometric analysis.

1. **Core Python Libraries:**
    - `os` and `math`: These libraries provide fundamental functionality related to filesystem operations and mathematical computations, respectively.
    - `numpy as np`: This library is employed for performing advanced numerical operations, which are crucial in processing data efficiently.
2. **Visualization Tools:**
    - `matplotlib.pyplot as plt`: This powerful visualization library is utilized for creating a wide array of plots and visual representations of data.
    - `Polygon and PatchCollection`: These constructs from `matplotlib` are specifically designed for drawing polygonal shapes, enhancing our graphical capabilities.
3. **CAD and Geometry Libraries:**
    - `ezdxf`: This library enables the reading and writing of DXF files, facilitating interactions with CAD data.
    - **Shapely Classes**:
        - Entities such as `Polygon`, `MultiPolygon`, `box`, `Point`, and `LineString` are essential for geometric modeling and spatial analysis.
        - The `unary_union` function is utilized for merging multiple geometric shapes into a singular entity, streamlining complex geometric operations.
        - The `affinity` module is also available for executing geometric transforms, such as translation, scaling, and rotation, should the need arise in our analyses.
4. **Plotting Configuration:**
    - The setup includes activation of inline plotting within the notebook, alongside configuration settings for default figure dimensions and DPI (dots per inch) to ensure clarity in visualizations.
    - A confirmation message will be printed, verifying that all libraries have been successfully imported and are ready for further use.

By following these steps, we ensure a robust foundation for subsequent data analysis and visualization tasks.

2. Configuration Parameters

All dimensions are in centimeters (cm).

```
class Config:
    """Manufacturing parameters - ALL DIMENSIONS IN CENTIMETERS"""
```

```
    # Laminate structure
    NUM_LAYERS = 5  # rigid-adhesive-flexible-adhesive-rigid
    IS_ADHESIVE = [False, True, False, True, False]

    # Manufacturing tolerances
    KERF = 0.005                # cm (0.05mm laser kerf)
    SUPPORT_WIDTH = 0.2         # cm (2mm)

    # Alignment features
    JIG_DIAMETER = 0.5          # cm (5mm alignment holes)
    JIG_SPACING = 1.0           # cm (spacing from edges)

print("Configuration loaded!")
print(f"  Laminate layers: {Config.NUM_LAYERS}")
print(f"  Laser kerf: {Config.KERF} cm ({Config.KERF*10:.3f} mm)")
print(f"  Alignment hole diameter: {Config.JIG_DIAMETER} cm")
```

```
Configuration loaded!
  Laminate layers: 5
  Laser kerf: 0.005 cm (0.050 mm)
  Alignment hole diameter: 0.5 cm
```

Define global manufacturing configuration (Config)

This block defines a `Config` class that holds all key manufacturing parameters (in **centimeters**):

- **Laminate structure**
  - `NUM_LAYERS = 5` : total number of layers (rigid–adhesive–flexible–adhesive–rigid).
  - `IS_ADHESIVE = [False, True, False, True, False]` : Boolean flags indicating which layers are adhesive.
- **Manufacturing tolerances**
  - `KERF = 0.005` : laser kerf (cut width) in cm (0.05 mm).
  - `SUPPORT_WIDTH = 0.2` : extra margin added around the perimeter to create a support frame.
- **Alignment features**
  - `JIG_DIAMETER = 0.5` : diameter of alignment holes (for pins or dowels).
  - `JIG_SPACING = 1.0` : spacing from the body outline to the alignment holes.

After defining the class, it prints out a brief summary of the chosen parameters.

3. Layer and Laminate Classes

```
class Layer:
    """Represents a single layer geometry using Shapely."""
```

```
    def __init__(self, geometry=None):
        if geometry is None:
            self.geom = Polygon()
        elif isinstance(geometry, (Polygon, MultiPolygon)):
            self.geom = geometry
        elif isinstance(geometry, list):
            if len(geometry) >= 3:
                self.geom = Polygon(geometry)
            else:
                self.geom = Polygon()
        else:
            self.geom = Polygon()

    def __or__(self, other):
        result = Layer()
        result.geom = unary_union([self.geom, other.geom])
        return result

    def __sub__(self, other):
        result = Layer()
        result.geom = self.geom.difference(other.geom)
        return result

    def __lshift__(self, distance):
        result = Layer()
        if distance != 0:
            result.geom = self.geom.buffer(distance, join_style=2)
        else:
            result.geom = self.geom
        return result

    @property
    def is_empty(self):
        return self.geom.is_empty

    @property
    def bounds(self):
        return self.geom.bounds if not self.geom.is_empty else (0, 0, 0, 0)

    @property
    def area(self):
        return self.geom.area

    def to_polygons(self):
        if isinstance(self.geom, MultiPolygon):
            return list(self.geom.geoms)
        elif isinstance(self.geom, Polygon) and not self.geom.is_empty:
            return [self.geom]
        return []

    def export_dxf(self, filename, layer_name='0'):
        doc = ezdxf.new('R2010')
        msp = doc.modelspace()

        for poly in self.to_polygons():
            if poly.is_empty:
```

```
                continue
            coords = list(poly.exterior.coords)
            if len(coords) >= 3:
                msp.add_lwpolyline(coords, close=True, dxfattribs={'layer':
layer_name})
            for interior in poly.interiors:
                coords = list(interior.coords)
                if len(coords) >= 3:
                    msp.add_lwpolyline(coords, close=True, dxfattribs={'layer':
layer_name})

        doc.saveas(filename)


class Laminate:
    """Multi-layer laminate structure."""

    def __init__(self, *layers):
        self.layers = list(layers)

    def __getitem__(self, index):
        return self.layers[index]

    def __len__(self):
        return len(self.layers)

    def __or__(self, other):
        result = Laminate()
        result.layers = [a | b for a, b in zip(self.layers, other.layers)]
        return result

    @property
    def bounds(self):
        all_bounds = [layer.bounds for layer in self.layers if not layer.is_empty]
        if not all_bounds:
            return (0, 0, 0, 0)
        return (min(b[0] for b in all_bounds), min(b[1] for b in all_bounds),
                max(b[2] for b in all_bounds), max(b[3] for b in all_bounds))

print("Layer and Laminate classes defined!")
```

```
Layer and Laminate classes defined!
```

Define Layer and Laminate data structures

This block introduces two core classes used throughout the workflow:

`Layer`

Represents a single physical layer in the laminate as a Shapely geometry.

Key behavior:

- **Initialization**
  - If no geometry is provided, it initializes as an empty `Polygon`.
  - Accepts a Shapely `Polygon` / `MultiPolygon` or a list of points to create a polygon.
- **Boolean operations**
  - `__or__(self, other)`: geometric union of two layers.
  - `__sub__(self, other)`: geometric difference (subtract `other` from `self`).
- **Offset operation**
  - `__lshift__(self, distance)`: offsets the geometry outward (positive distance) or inward (negative distance) using `buffer()`. This is used later to create an expanded perimeter (support frame).
- **Convenience properties**
  - `is_empty`: whether the geometry has no area.
  - `bounds`: `(min_x, min_y, max_x, max_y)` bounding box of the layer (or zeros if empty).
  - `area`: total area of the layer.
- **Conversion utilities**
  - `to_polygons()`: always returns a list of Shapely polygons (handles both `Polygon` and `MultiPolygon`).
  - `export_dxf(filename, layer_name='0')`:
    - Creates a new DXF file.
    - Writes the exterior and interior rings (holes) of each polygon as DXF polylines on the specified DXF layer.
    - Saves the DXF to disk.

`Laminate`

Represents a **stack** of layers (the full multi-layer laminate).

Key behavior:

- `__init__(*layers)`: stores a list of `Layer` objects.
- `__getitem__` and `__len__`: allow indexing and `len(laminate)`.
- `__or__(self, other)`: union two laminates layer-by-layer (zips corresponding layers).
- `bounds`: returns the overall bounding box across all non-empty layers.

At the end, the block prints a message confirming that the Layer and Laminate classes are ready.

4. DXF Geometry Extraction Functions

These functions read the **EXACT** geometry from the input DXF file.

```
def create_stadium_polygon(center_x, y_min, y_max, arc_radius):
    """
    Create a stadium-shaped polygon using Shapely's buffer operation.
    This creates a capsule shape by buffering a vertical line.
    """
    line = LineString([(center_x, y_min), (center_x, y_max)])
    stadium = line.buffer(arc_radius, cap_style=1, resolution=16)
    return stadium


def extract_hinge_slot_geometry(entities):
    """
    Extract the exact stadium-shaped hinge slot from DXF entities.
    """
    lines = [e for e in entities if e.dxftype() == 'LINE']
    arcs = [e for e in entities if e.dxftype() == 'ARC']

    if len(lines) < 2 or len(arcs) < 2:
        return None

    all_x, all_y = [], []
    for line in lines:
        all_x.extend([line.dxf.start.x, line.dxf.end.x])
        all_y.extend([line.dxf.start.y, line.dxf.end.y])

    for arc in arcs:
        all_x.append(arc.dxf.center.x)
        all_y.append(arc.dxf.center.y)

    center_x = (min(all_x) + max(all_x)) / 2
    min_y, max_y = min(all_y), max(all_y)
    arc_radius = arcs[0].dxf.radius

    return create_stadium_polygon(center_x, min_y, max_y, arc_radius)


def read_dxf_exact_geometry(filepath):
    """
    Read EXACT geometry from DXF file, preserving all cut patterns.
    """
    doc = ezdxf.readfile(filepath)
    msp = doc.modelspace()

    result = {
        'body_polygon': None,
        'hinge_slots': [],
        'bounds': None,
        'slot_info': []
    }

    # Collect entities by layer
    layers = {}
    for entity in msp:
```

```
        layer = entity.dxf.layer.lower()
        if layer not in layers:
            layers[layer] = []
        layers[layer].append(entity)

    print(f"Found layers: {list(layers.keys())}")

    # Extract body outline
    for name in ['plate', 'body', '0']:
        if name in layers:
            lines = [e for e in layers[name] if e.dxftype() == 'LINE']
            if lines:
                points = []
                for line in lines:
                    points.append((line.dxf.start.x, line.dxf.start.y))
                    points.append((line.dxf.end.x, line.dxf.end.y))

                xs = [p[0] for p in points]
                ys = [p[1] for p in points]
                min_x, max_x = min(xs), max(xs)
                min_y, max_y = min(ys), max(ys)

                result['body_polygon'] = box(min_x, min_y, max_x, max_y)
                result['bounds'] = (min_x, min_y, max_x, max_y)
                print(f"Body: ({min_x:.4f}, {min_y:.4f}) to ({max_x:.4f},
{max_y:.4f}) cm")
                print(f"Dimensions: {max_x - min_x:.4f} x {max_y - min_y:.4f} cm")
            break

    # Extract hinge slots
    hinge_layers = sorted([name for name in layers.keys() if 'hinge' in name])

    for hinge_layer in hinge_layers:
        entities = layers[hinge_layer]
        slot_polygon = extract_hinge_slot_geometry(entities)

        if slot_polygon:
            result['hinge_slots'].append(slot_polygon)
            bounds = slot_polygon.bounds
            slot_info = {
                'layer': hinge_layer,
                'center_x': (bounds[0] + bounds[2]) / 2,
                'y_min': bounds[1],
                'y_max': bounds[3],
                'width': bounds[2] - bounds[0],
                'height': bounds[3] - bounds[1]
            }
            result['slot_info'].append(slot_info)
            print(f"{hinge_layer}: x={slot_info['center_x']:.4f}, "
                  f"size={slot_info['width']:.4f} x {slot_info['height']:.4f} cm")

    return result

print("DXF extraction functions defined!")
```

```
DXF extraction functions defined!
```

DXF geometry extraction: body and hinge slots

This block defines the functions that read the **exact** 2D geometry from the input DXF file:

`create_stadium_polygon(center_x, y_min, y_max, arc_radius)`

- Builds a “stadium” or capsule-shaped polygon:
  - Starts with a vertical line segment at `center_x` from `y_min` to `y_max`.
  - Buffers this line with radius `arc_radius` and `cap_style=1` (round caps).
- Result: a rounded-rectangle-like shape used to represent hinge slots.

`extract_hinge_slot_geometry(entities)`

- Takes all DXF entities (lines/arcs) belonging to a hinge layer.
- Splits them into lines and arcs.
- Uses their positions and radii to reconstruct the hinge slot as a stadium-shaped polygon:
  - Collects all relevant X and Y coordinates.
  - Estimates the slot center and vertical extent.
  - Uses `create_stadium_polygon` with the detected radius.
- Returns a Shapely polygon representing the exact hinge slot geometry.

`read_dxf_exact_geometry(filepath)`

- Reads the DXF file using `ezdxf.readfile`.
- Iterates over all entities in modelspace and groups them by DXF layer name.
- Initializes a result dict with:
  - `body_polygon`: main body outline polygon.
  - `hinge_slots`: list of hinge slot polygons.
  - `bounds`: bounding box of the body.
  - `slot_info`: metadata (center, width, height) for each hinge slot.

Processing steps:

1. **Collect entities by layer**
   - Loops through all entities and groups them in a dictionary keyed by lowercase layer name.
2. **Extract main body polygon**
   - Looks for layers named `'plate'`, `'body'`, or `'0'` (in that order).
   - Gathers the LINE entities on that layer.
   - Uses their endpoints to compute a bounding box and constructs a rectangle (`box(min_x, min_y, max_x, max_y)`) as the body polygon.

- Stores this polygon and its bounds and prints the dimensions.

3. **Extract hinge slots**

- Finds all layers whose names contain the word `'hinge'`.
- For each hinge layer:
  - Calls `extract_hinge_slot_geometry` to reconstruct the slot polygon.
  - Stores the polygon in `hinge_slots`.
  - Computes and stores its center, width, and height in `slot_info`.
  - Prints a line describing each hinge slot.

Finally, it prints a confirmation that the DXF extraction functions are defined.

5. Manufacturing Functions

```
def create_alignment_holes(body_bounds, diameter, spacing):
    """Create alignment holes at corners."""
    min_x, min_y, max_x, max_y = body_bounds
    radius = diameter / 2

    corners = [
        (min_x - spacing, min_y - spacing),
        (max_x + spacing, min_y - spacing),
        (max_x + spacing, max_y + spacing),
        (min_x - spacing, max_y + spacing)
    ]

    holes = [Point(x, y).buffer(radius, resolution=32) for x, y in corners]
    combined = unary_union(holes)
    result = Layer()
    result.geom = combined
    return result


def build_laminate_from_exact_geometry(body_polygon, hinge_slots, is_adhesive):
    """
    Build laminate structure using EXACT geometry from DXF.

    - Rigid layers: body with hinge slots removed
    - Adhesive layers: same as rigid
    - Flexible layer: full body (continuous across hinges)
    """
    num_layers = len(is_adhesive)
    layers = []

    # Combine hinge slots
    if hinge_slots:
        valid_slots = [s.buffer(0) if not s.is_valid else s for s in hinge_slots if
s]
        if valid_slots:
            combined_slots = valid_slots[0]
            for slot in valid_slots[1:]:
                combined_slots = combined_slots.union(slot)
```

```
        else:
            combined_slots = Polygon()
    else:
        combined_slots = Polygon()

    # Create layers
    rigid_geom = body_polygon.difference(combined_slots)
    if not rigid_geom.is_valid:
        rigid_geom = rigid_geom.buffer(0)

    rigid_layer = Layer()
    rigid_layer.geom = rigid_geom

    adhesive_layer = Layer()
    adhesive_layer.geom = rigid_geom

    flexible_layer = Layer()
    flexible_layer.geom = body_polygon

    for i, is_adh in enumerate(is_adhesive):
        if i == num_layers // 2:
            layers.append(flexible_layer)
        elif is_adh:
            layers.append(adhesive_layer)
        else:
            layers.append(rigid_layer)

    return Laminate(*layers)

print("Manufacturing functions defined!")
```

```
Manufacturing functions defined!
```

Manufacturing helper functions (alignment holes + laminate build)

This block defines two key functions for manufacturing geometry:

`create_alignment_holes(body_bounds, diameter, spacing)`

- Inputs:
  - `body_bounds`: `(min_x, min_y, max_x, max_y)` for the main body.
  - `diameter`: diameter of the alignment holes.
  - `spacing`: how far the holes sit away from the body's corners.
- Computes four corner positions:
  - Each corner is shifted outward by `spacing` from the body.
- For each corner:
  - Creates a circular hole using `Point(x, y).buffer(radius)`.
- Merges all four holes into a single geometry using `unary_union`.
- Wraps the union in a `Layer` object and returns it.

`build_laminate_from_exact_geometry(body_polygon, hinge_slots, is_adhesive)`

- Inputs:
  - `body_polygon`: exact body outline from the DXF.
  - `hinge_slots`: list of hinge slot polygons.
  - `is_adhesive`: list of booleans specifying which layers are adhesive.

Processing steps:

1. **Combine hinge slots**
   - Unions all valid hinge slot polygons into one `combined_slots` geometry.
   - If no hinge slots exist, `combined_slots` is an empty polygon.
2. **Create base geometries**
   - `rigid_geom = body_polygon.difference(combined_slots)`:
     - Rigid layers are the body with the hinge slots **removed**.
   - Ensure `rigid_geom` is valid (repair with `buffer(0)` if needed).
   - `rigid_layer` and `adhesive_layer` both use `rigid_geom`.
   - `flexible_layer` uses the full `body_polygon` (no cutouts), so it spans across hinges.
3. **Build the laminate stack**
   - Iterates over layer indices and `is_adhesive` flags.
   - Chooses which geometry to assign per layer:
     - The **middle** layer (index `num_layers // 2`) is always the flexible layer.
     - Adhesive layers get the `adhesive_layer` geometry.
     - Remaining layers are rigid.

- Returns a `Laminate` object bundling all layer geometries in order.

A message at the end confirms that “Manufacturing functions” have been defined.

6. Visualization Functions

```
def plot_layer(ax, layer, color='blue', alpha=0.5, edgecolor='black', label=None):
    """Plot a Layer on matplotlib axes."""
    patches = []
    for poly in layer.to_polygons():
        if not poly.is_empty:
            coords = np.array(poly.exterior.coords)
            patch = MplPolygon(coords, closed=True)
            patches.append(patch)

            for interior in poly.interiors:
                hole_coords = np.array(interior.coords)
                ax.plot(hole_coords[:, 0], hole_coords[:, 1],
```

```
                        color=edgecolor, linewidth=0.5, linestyle='--')

    if patches:
        pc = PatchCollection(patches, facecolor=color, edgecolor=edgecolor,
                             alpha=alpha, linewidth=0.5)
        ax.add_collection(pc)

    if label:
        ax.plot([], [], color=color, alpha=alpha, linewidth=5, label=label)


def plot_cut_pattern(body_polygon, hinge_slots, title="Cut Pattern"):
    """Plot the exact cut pattern showing body and slots."""
    fig, ax = plt.subplots(figsize=(14, 6))

    if body_polygon:
        coords = np.array(body_polygon.exterior.coords)
        ax.plot(coords[:, 0], coords[:, 1], 'b-', linewidth=2, label='Body outline')
        ax.fill(coords[:, 0], coords[:, 1], alpha=0.2, color='blue')

    for i, slot in enumerate(hinge_slots):
        if slot:
            coords = np.array(slot.exterior.coords)
            ax.plot(coords[:, 0], coords[:, 1], 'r-', linewidth=1.5,
                    label='Hinge slots' if i == 0 else '')
            ax.fill(coords[:, 0], coords[:, 1], alpha=0.5, color='red')

    ax.set_title(title, fontsize=14, fontweight='bold')
    ax.set_aspect('equal')
    ax.autoscale()
    ax.grid(True, alpha=0.3)
    ax.legend(loc='upper right')
    ax.set_xlabel('X (cm)')
    ax.set_ylabel('Y (cm)')
    plt.tight_layout()
    plt.show()


def plot_laminate(laminate, title="Laminate Layers"):
    """Plot all laminate layers."""
    n_layers = len(laminate)
    cols = 3
    rows = (n_layers + 1) // cols + 1

    fig, axes = plt.subplots(rows, cols, figsize=(14, 4*rows))
    axes = axes.flatten()

    colors = ['#e74c3c', '#f39c12', '#3498db', '#f39c12', '#e74c3c']
    labels = ['Rigid (Top)', 'Adhesive', 'Flexible', 'Adhesive', 'Rigid (Bottom)']

    for i, layer in enumerate(laminate.layers):
        ax = axes[i]
        plot_layer(ax, layer, color=colors[i], alpha=0.7)
        ax.set_title(f'Layer {i+1}: {labels[i]}')
        ax.set_aspect('equal')
        ax.autoscale()
```

```
        ax.grid(True, alpha=0.3)
        ax.set_xlabel('X (cm)')
        ax.set_ylabel('Y (cm)')

    ax_combined = axes[n_layers]
    for i, layer in enumerate(laminate.layers):
        plot_layer(ax_combined, layer, color=colors[i], alpha=0.3, label=labels[i])
    ax_combined.set_title('Combined View')
    ax_combined.set_aspect('equal')
    ax_combined.autoscale()
    ax_combined.grid(True, alpha=0.3)
    ax_combined.legend(loc='upper right', fontsize=8)
    ax_combined.set_xlabel('X (cm)')
    ax_combined.set_ylabel('Y (cm)')

    for i in range(n_layers + 1, len(axes)):
        axes[i].set_visible(False)

    plt.suptitle(title, fontsize=14, fontweight='bold')
    plt.tight_layout()
    plt.show()

print("Visualization functions defined!")
```

```
Visualization functions defined!
```

Visualization functions for layers and laminate

This block defines helper functions for plotting the geometry:

`plot_layer(ax, layer, color='blue', alpha=0.5, edgecolor='black', label=None)`

- Plots a single `Layer` onto a given Matplotlib axes:
  - Converts each polygon in the layer to a Matplotlib `Polygon` patch.
  - Adds the patches as a `PatchCollection` (filled, colored, partially transparent).
  - Draws interior rings (holes) as dashed outlines.
  - Optionally adds a legend entry for the layer if `label` is provided.

`plot_cut_pattern(body_polygon, hinge_slots, title="Cut Pattern")`

- Visualizes the original DXF cut pattern:
  - Plots the body outline in blue, lightly filled.
  - Plots each hinge slot in red, filled and outlined.
- Adds title, grid, equal aspect ratio, axis labels, and a legend.
- Shows the figure.

`plot_laminate(laminate, title="Laminate Layers")`

- Visualizes all layers in the laminate stack in a grid of subplots:
  - Determines number of rows/columns from the number of layers.
  - Uses a fixed color scheme:
    - Red-ish for rigid layers.
    - Yellow/orange for adhesive.
    - Blue for flexible.
  - For each layer:
    - Calls `plot_layer` with the appropriate color.
    - Sets subplot title and axes properties.
  - Uses an additional subplot to overlay all layers together (“Combined View”).
- Hides any unused subplot axes.
- Adds a global title, tight layout, and shows the figure.

A final print statement confirms that visualization functions are defined.

7. Load Input DXF File

Load the input DXF file and extract the **EXACT** geometry.

```
# Path to input DXF file
INPUT_DXF = 'zz.dxf'
OUTPUT_DIR = 'output'

os.makedirs(OUTPUT_DIR, exist_ok=True)

print("=" * 50)
print("READING EXACT DXF GEOMETRY")
print("=" * 50)
print(f"Input file: {INPUT_DXF}\n")

geom_data = read_dxf_exact_geometry(INPUT_DXF)

print("\n✓ DXF file loaded - EXACT geometry extracted!")
```

```
==================================================
READING EXACT DXF GEOMETRY
==================================================
Input file: zz.dxf

Found layers: ['plate', 'hinge_1', 'hinge_2', 'hinge_3', 'hinge_4']
Body: (0.0000, 0.0000) to (18.0000, 3.0000) cm
Dimensions: 18.0000 x 3.0000 cm
hinge_1: x=4.0000, size=0.1000 x 1.3000 cm
hinge_2: x=8.0000, size=0.1000 x 1.3000 cm
hinge_3: x=10.0000, size=0.1000 x 1.3000 cm
hinge_4: x=16.0000, size=0.1000 x 1.3000 cm

✓ DXF file loaded - EXACT geometry extracted!
```

Load DXF file and extract exact geometry

This block sets up file paths and reads the input DXF:

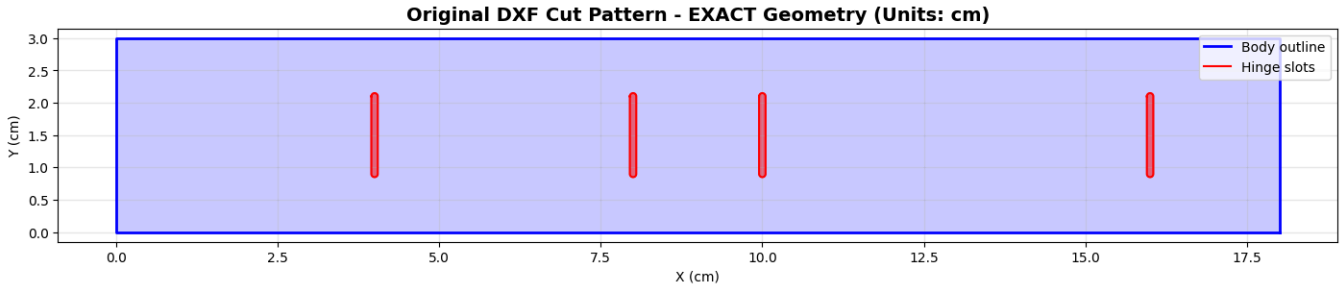
- Defines:
  - `INPUT_DXF = 'zz.dxf'`: path/name of the input DXF body file.
  - `OUTPUT_DIR = 'output'`: directory where all generated DXF and PNG files will be saved.
- Ensures `OUTPUT_DIR` exists using `os.makedirs(..., exist_ok=True)`.
- Prints a header indicating that it's reading the DXF geometry.
- Calls `read_dxf_exact_geometry(INPUT_DXF)` to:
  - Extract the body polygon.
  - Extract all hinge slot polygons.
  - Compute overall bounds and slot metadata.
- Stores all this in `geom_data` and prints a success message once done.

8. Visualize Original Cut Pattern

This shows the **exact** cut pattern from the input DXF file.

```
print("=" * 50)
print("ORIGINAL DXF CUT PATTERN (EXACT)")
print("=" * 50)

print(f"\nBody dimensions: {geom_data['bounds'][2] - geom_data['bounds'][0]:.4f} x "
      f"{geom_data['bounds'][3] - geom_data['bounds'][1]:.4f} cm")

print(f"\nHinge slots ({len(geom_data['hinge_slots'])} total):")
for info in geom_data['slot_info']:
    print(f"  {info['layer']}: center_x={info['center_x']:.4f} cm, "
          f"size={info['width']:.4f} x {info['height']:.4f} cm")

plot_cut_pattern(
    geom_data['body_polygon'],
    geom_data['hinge_slots'],
    "Original DXF Cut Pattern - EXACT Geometry (Units: cm)"
)
```

```
==================================================
ORIGINAL DXF CUT PATTERN (EXACT)
==================================================

Body dimensions: 18.0000 x 3.0000 cm

Hinge slots (4 total):
  hinge_1: center_x=4.0000 cm, size=0.1000 x 1.3000 cm
  hinge_2: center_x=8.0000 cm, size=0.1000 x 1.3000 cm
  hinge_3: center_x=10.0000 cm, size=0.1000 x 1.3000 cm
  hinge_4: center_x=16.0000 cm, size=0.1000 x 1.3000 cm
```



Inspect and visualize the original DXF cut pattern

This block:

1. Prints a section header “ORIGINAL DXF CUT PATTERN (EXACT)”.
2. Displays the overall body dimensions using `geom_data['bounds']`.
3. Prints a summary for each hinge slot:
    - Layer name.
    - Center X position.
    - Slot width and height.
4. Calls `plot_cut_pattern(...)` to draw:
    - The body outline.
    - All hinge slots overlaid on top.

This is the “ground truth” geometry directly taken from the DXF, before any laminate processing.

9. Build Laminate Structure

```
print("=" * 50)
print("BUILDING LAMINATE STRUCTURE")
print("=" * 50)

laminate = build_laminate_from_exact_geometry(
    geom_data['body_polygon'],
    geom_data['hinge_slots'],
    Config.IS_ADHESIVE
)

print(f"\nNumber of layers: {len(laminate)}")
print(f"Layer stack: Rigid → Adhesive → Flexible → Adhesive → Rigid")
print(f"\nLayer details:")
layer_types = ['Rigid (Top)', 'Adhesive', 'Flexible', 'Adhesive', 'Rigid (Bottom)']
for i, (layer, ltype) in enumerate(zip(laminate.layers, layer_types)):
    print(f"  Layer {i+1} ({ltype}): Area = {layer.area:.4f} cm²")
```

```
==================================================
BUILDING LAMINATE STRUCTURE
==================================================

Number of layers: 5
Layer stack: Rigid → Adhesive → Flexible → Adhesive → Rigid

Layer details:
  Layer 1 (Rigid (Top)): Area = 53.4886 cm²
  Layer 2 (Adhesive): Area = 53.4886 cm²
  Layer 3 (Flexible): Area = 54.0000 cm²
  Layer 4 (Adhesive): Area = 53.4886 cm²
  Layer 5 (Rigid (Bottom)): Area = 53.4886 cm²
```

Build the 5-layer laminate from the exact geometry

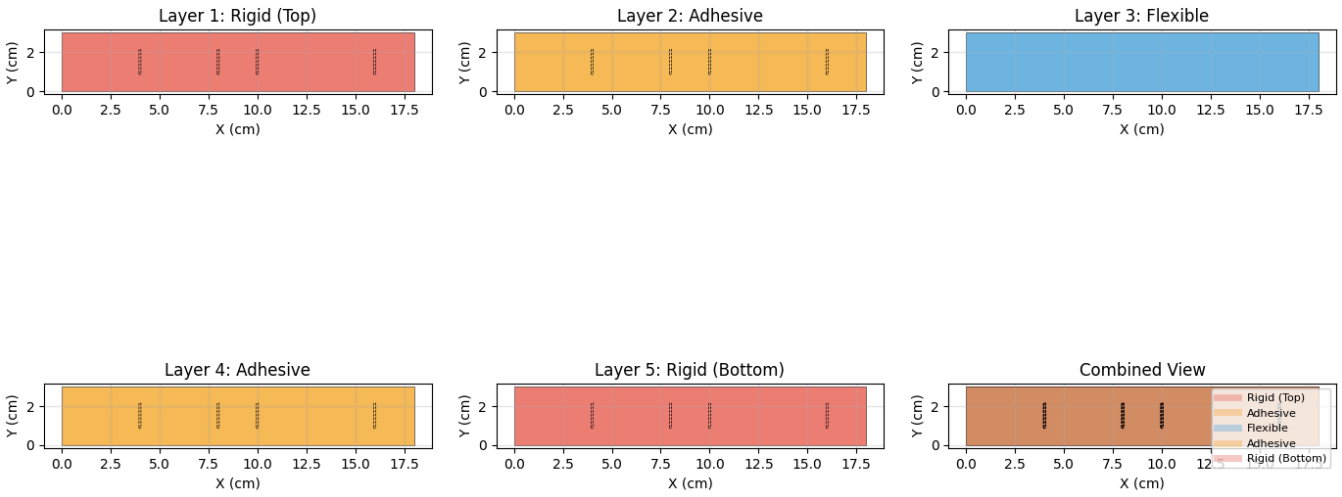
This block constructs the full laminate stack using the previously defined function:

- Prints a “BUILDING LAMINATE STRUCTURE” header.
- Calls `build_laminate_from_exact_geometry(...)` with:
  - `geom_data['body_polygon']` as the body outline.
  - `geom_data['hinge_slots']` as the hinge locations.
  - `Config.IS_ADHESIVE` to determine which layers are adhesive vs rigid.
- Saves the result in `laminate`.
- Prints:
  - Total number of layers.
  - A description of the stack order: **Rigid → Adhesive → Flexible → Adhesive → Rigid**.
- For each layer, prints its type and total area in cm².

This step translates the flat DXF geometry into a structured multi-layer laminate model.

10. Visualize Laminate Layers

```
plot_laminate(laminate, "Laminate Layers - EXACT Geometry (Units: cm)")
```



Visualize all laminate layers

This block calls `plot_laminate(laminate, ...)` to display:

- Each layer of the laminate in its own subplot with a descriptive title.
- A combined view showing all layers overlaid.
- All dimensions are in centimeters.

This helps verify that:

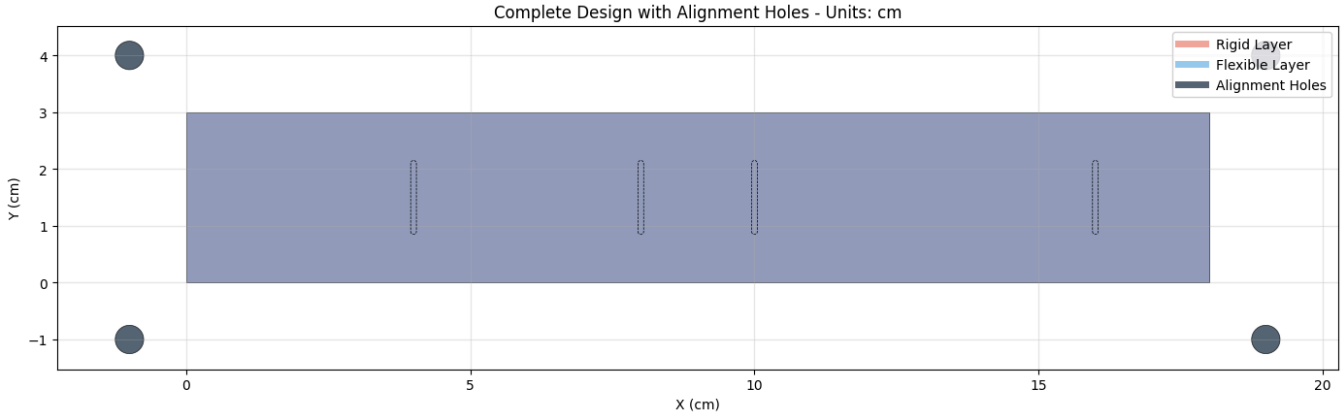
- Rigid and adhesive layers have cutouts at hinge slots.
- The flexible layer spans across the hinges as intended.

11. Create Alignment Holes

```
print("=" * 50)
print("ALIGNMENT HOLES")
print("=" * 50)

alignment_holes = create_alignment_holes(
    geom_data['bounds'],
    Config.JIG_DIAMETER,
    Config.JIG_SPACING
)

print(f"\nHole diameter: {Config.JIG_DIAMETER} cm ({Config.JIG_DIAMETER*10} mm)")
print(f"Spacing from body: {Config.JIG_SPACING} cm")
print(f"Number of holes: 4 (one at each corner)")

# Visualize
fig, ax = plt.subplots(figsize=(14, 6))
```

```
plot_layer(ax, laminate[0], color='#e74c3c', alpha=0.5, label='Rigid Layer')
plot_layer(ax, laminate[2], color='#3498db', alpha=0.5, label='Flexible Layer')
plot_layer(ax, alignment_holes, color='#2c3e50', alpha=0.8, label='Alignment Holes')
ax.set_title('Complete Design with Alignment Holes - Units: cm')
ax.set_aspect('equal')
ax.autoscale()
ax.grid(True, alpha=0.3)
ax.legend()
ax.set_xlabel('X (cm)')
ax.set_ylabel('Y (cm)')
plt.tight_layout()
plt.show()
```

```
==================================================
ALIGNMENT HOLES
==================================================

Hole diameter: 0.5 cm (5.0 mm)
Spacing from body: 1.0 cm
Number of holes: 4 (one at each corner)
```



Create and visualize alignment holes

This block adds alignment features to the design:

1. Prints an “ALIGNMENT HOLES” header.
2. Calls `create_alignment_holes(...)` using:
   - The body bounds from `geom_data['bounds']`.
   - The alignment hole diameter and spacing from `Config`.
3. Stores the result in `alignment_holes`.

It then prints:

- Hole diameter (in cm and mm).
- Spacing from the body.
- Number of holes (always 4, one at each corner).

Finally, it visualizes:

- The top rigid layer.
- The flexible layer.
- The alignment holes on top of them.

The plot shows the complete design—including alignment features—so it can be used for fixture/jig design and layer registration during fabrication.

12. Export DXF Files

```
print("=" * 50)
print("EXPORTING DXF FILES (EXACT GEOMETRY)")
print("=" * 50)
print(f"\nOutput directory: {OUTPUT_DIR}\n")

output_files = {}

# Export individual layers
layer_names = ['rigid_top', 'adhesive_top', 'flexible', 'adhesive_bottom',
'rigid_bottom']
for i, (layer, name) in enumerate(zip(laminate.layers, layer_names)):
    filename = os.path.join(OUTPUT_DIR, f"{name}.dxf")
    layer.export_dxf(filename)
    output_files[name] = filename
    print(f"✓ Exported: {filename}")

# Export combined cut pattern
def export_dxf_with_layers(body_polygon, hinge_slots, filename):
    doc = ezdxf.new('R2010')
    msp = doc.modelspace()
    doc.layers.add('plate', color=1)
    doc.layers.add('cuts', color=5)

    if body_polygon:
        coords = list(body_polygon.exterior.coords)
        msp.add_lwpolyline(coords, close=True, dxfattribs={'layer': 'plate'})

    for slot in hinge_slots:
        if slot:
            coords = list(slot.exterior.coords)
            msp.add_lwpolyline(coords, close=True, dxfattribs={'layer': 'cuts'})

    doc.saveas(filename)

combined_file = os.path.join(OUTPUT_DIR, "combined_cut_pattern.dxf")
export_dxf_with_layers(geom_data['body_polygon'], geom_data['hinge_slots'],
combined_file)
print(f"✓ Exported: {combined_file}")

# Export first pass cuts
first_pass = laminate[0] | alignment_holes
first_pass_file = os.path.join(OUTPUT_DIR, "first_pass_cuts.dxf")
first_pass.export_dxf(first_pass_file)
print(f"✓ Exported: {first_pass_file}")
```

```python
# Export final perimeter
body_layer = Layer()
body_layer.geom = geom_data['body_polygon']
perimeter = body_layer << Config.SUPPORT_WIDTH
perimeter_file = os.path.join(OUTPUT_DIR, "final_perimeter.dxf")
perimeter.export_dxf(perimeter_file)
print(f"✓ Exported: {perimeter_file}")

print(f"\n✓ All DXF files exported with EXACT geometry!")
```

```
==================================================
EXPORTING DXF FILES (EXACT GEOMETRY)
==================================================

Output directory: output

✓ Exported: output/rigid_top.dxf
✓ Exported: output/adhesive_top.dxf
✓ Exported: output/flexible.dxf
✓ Exported: output/adhesive_bottom.dxf
✓ Exported: output/rigid_bottom.dxf
✓ Exported: output/combined_cut_pattern.dxf
✓ Exported: output/first_pass_cuts.dxf
✓ Exported: output/final_perimeter.dxf

✓ All DXF files exported with EXACT geometry!
```

Export manufacturing DXF files (exact geometry)

This block writes out all the DXF files needed for laser cutting:

1. Prints an “EXPORTING DXF FILES (EXACT GEOMETRY)” header and shows the output directory.
2. **Export individual layer DXFs**
   - Iterates over the `laminate` layers and names:
     - `rigid_top.dxf`
     - `adhesive_top.dxf`
     - `flexible.dxf`
     - `adhesive_bottom.dxf`
     - `rigid_bottom.dxf`
   - For each layer:
     - Calls `layer.export_dxf(...)`.
     - Records and prints the output filename.
3. **Export combined cut pattern**
   - Defines `export_dxf_with_layers(body_polygon, hinge_slots, filename)` which:

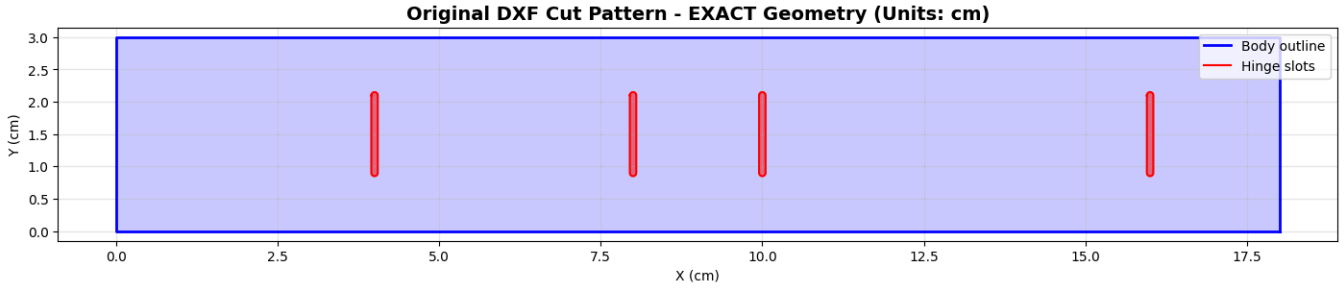
    - Sets up DXF layers: `'plate'` for the body and `'cuts'` for hinge slots.
    - Writes the body outline polyline to `'plate'`.
    - Writes each hinge slot outline to `'cuts'`.
  - Saves this file as `combined_cut_pattern.dxf`.
4. **Export first-pass cuts**
  - Creates `first_pass = laminate[0] | alignment_holes`:
    - Unions the top rigid layer geometry with the alignment holes.
  - Saves as `first_pass_cuts.dxf`.
  - Intended as the first laser pass: internal features + alignment holes.
5. **Export final perimeter**
  - Wraps `geom_data['body_polygon']` in a `Layer` called `body_layer`.
  - Uses `body_layer << Config.SUPPORT_WIDTH` (geometric offset) to create an expanded outline that includes a support frame.
  - Saves this as `final_perimeter.dxf`.

At the end, it prints a confirmation that all DXF files were exported using the exact geometry from the source DXF.

13. Save Visualizations

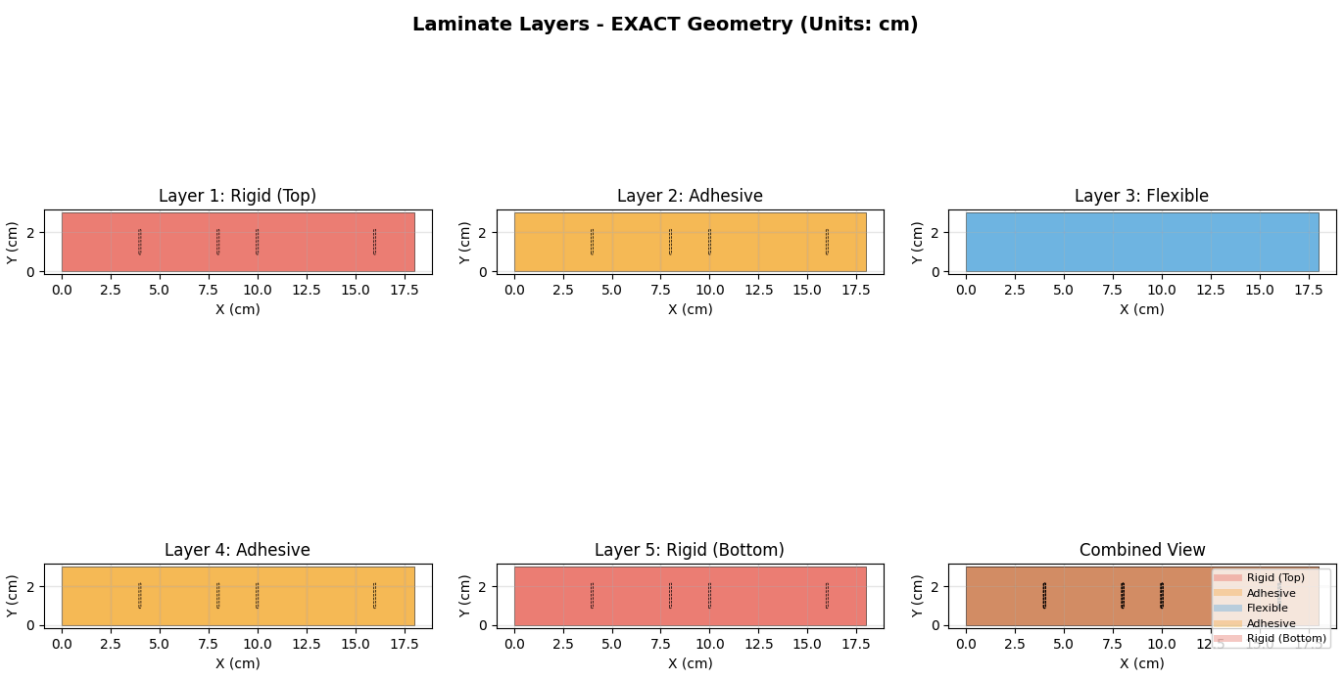
```
# Save cut pattern visualization
fig, ax = plt.subplots(figsize=(14, 6))
if geom_data['body_polygon']:
    coords = np.array(geom_data['body_polygon'].exterior.coords)
    ax.plot(coords[:, 0], coords[:, 1], 'b-', linewidth=2, label='Body outline')
    ax.fill(coords[:, 0], coords[:, 1], alpha=0.2, color='blue')

for i, slot in enumerate(geom_data['hinge_slots']):
    if slot:
        coords = np.array(slot.exterior.coords)
        ax.plot(coords[:, 0], coords[:, 1], 'r-', linewidth=1.5,
                label='Hinge slots' if i == 0 else '')
        ax.fill(coords[:, 0], coords[:, 1], alpha=0.5, color='red')

ax.set_title('Original DXF Cut Pattern - EXACT Geometry (Units: cm)', fontsize=14,
fontweight='bold')
ax.set_aspect('equal')
ax.autoscale()
ax.grid(True, alpha=0.3)
ax.legend(loc='upper right')
ax.set_xlabel('X (cm)')
ax.set_ylabel('Y (cm)')
plt.tight_layout()
plt.savefig(os.path.join(OUTPUT_DIR, 'original_cut_pattern.png'), dpi=150,
bbox_inches='tight')
print(f"✓ Saved: {OUTPUT_DIR}/original_cut_pattern.png")
plt.show()
```

```
✓ Saved: output/original_cut_pattern.png
```




```python
# Save laminate layers visualization
n_layers = len(laminate)
cols = 3
rows = (n_layers + 1) // cols + 1

fig, axes = plt.subplots(rows, cols, figsize=(14, 4*rows))
axes = axes.flatten()

colors = ['#e74c3c', '#f39c12', '#3498db', '#f39c12', '#e74c3c']
labels = ['Rigid (Top)', 'Adhesive', 'Flexible', 'Adhesive', 'Rigid (Bottom)']

for i, layer in enumerate(laminate.layers):
    ax = axes[i]
    plot_layer(ax, layer, color=colors[i], alpha=0.7)
    ax.set_title(f'Layer {i+1}: {labels[i]}')
    ax.set_aspect('equal')
    ax.autoscale()
    ax.grid(True, alpha=0.3)
    ax.set_xlabel('X (cm)')
    ax.set_ylabel('Y (cm)')

ax_combined = axes[n_layers]
for i, layer in enumerate(laminate.layers):
    plot_layer(ax_combined, layer, color=colors[i], alpha=0.3, label=labels[i])
ax_combined.set_title('Combined View')
ax_combined.set_aspect('equal')
ax_combined.autoscale()
ax_combined.grid(True, alpha=0.3)
ax_combined.legend(loc='upper right', fontsize=8)
ax_combined.set_xlabel('X (cm)')
ax_combined.set_ylabel('Y (cm)')

for i in range(n_layers + 1, len(axes)):
    axes[i].set_visible(False)

plt.suptitle('Laminate Layers - EXACT Geometry (Units: cm)', fontsize=14,
fontweight='bold')
plt.tight_layout()
plt.savefig(os.path.join(OUTPUT_DIR, 'laminate_layers.png'), dpi=150,
bbox_inches='tight')
print(f"✓ Saved: {OUTPUT_DIR}/laminate_layers.png")
plt.show()
```

```
✓ Saved: output/laminate_layers.png
```



Save PNG of original cut pattern

This block re-creates the original DXF cut pattern visualization and saves it as an image:

- Plots:
  - The body outline in blue (filled lightly).
  - All hinge slots in red (filled).
- Adds title, equal aspect ratio, grid, legend, and axis labels (units: cm).
- Saves the figure to `output/original_cut_pattern.png` with reasonable DPI and tight bounding box.
- Prints the save path and displays the figure.

This PNG is useful for reports, slides, or fabrication documentation.

Save PNG of laminate layers visualization

This block reproduces the laminate visualization and saves it:

- Sets up a grid of subplots sized to fit all layers plus a combined view.
- Uses the same color scheme and labels as `plot_laminate`:
  - Rigid top, Adhesive, Flexible, Adhesive, Rigid bottom.
- Plots each layer individually and then overlays all layers in the last subplot (“Combined View”).
- Hides any unused subplots.
- Adds a global title and tight layout.
- Saves the figure to `output/laminate_layers.png`.
- Prints the save path and displays the result.

This PNG summarizes the full layer stack for fabrication documentation.

14. Manufacturing Summary

```
print("=" * 60)
print("MANUFACTURING SUMMARY")
print("ALL DIMENSIONS IN CENTIMETERS - EXACT GEOMETRY")
print("=" * 60)

min_x, min_y, max_x, max_y = geom_data['bounds']

print(f"\n📐 BODY DIMENSIONS:")
print(f"   Width:  {max_x - min_x:.4f} cm ({(max_x - min_x)/2.54:.4f} inches)")
print(f"   Height: {max_y - min_y:.4f} cm ({(max_y - min_y)/2.54:.4f} inches)")

print(f"\n🔗 HINGE SLOT CONFIGURATION (EXACT FROM DXF):")
for info in geom_data['slot_info']:
    print(f"   {info['layer']}:")
    print(f"     Center X: {info['center_x']:.4f} cm")
    print(f"     Y range: {info['y_min']:.4f} to {info['y_max']:.4f} cm")
    print(f"     Slot size: {info['width']:.4f} x {info['height']:.4f} cm")

print(f"\n📋 LAYER STRUCTURE:")
print(f"   Layer 1 (Rigid Top): Body with hinge slots removed")
print(f"   Layer 2 (Adhesive): Same as rigid")
print(f"   Layer 3 (Flexible): Full body (continuous across hinges)")
print(f"   Layer 4 (Adhesive): Same as rigid")
print(f"   Layer 5 (Rigid Bottom): Body with hinge slots removed")

print(f"\n📍 ALIGNMENT HOLES:")
print(f"   Diameter: {Config.JIG_DIAMETER} cm")
print(f"   Spacing from body: {Config.JIG_SPACING} cm")

print(f"\n📋 MANUFACTURING PROCESS:")
print(f"   1. Cut internal features using layer DXF files")
print(f"   2. Stack layers using alignment holes for registration")
print(f"   3. Cut final perimeter to release device")

print(f"\n📁 OUTPUT FILES:")
for name in ['rigid_top.dxf', 'adhesive_top.dxf', 'flexible.dxf',
             'adhesive_bottom.dxf', 'rigid_bottom.dxf',
             'combined_cut_pattern.dxf', 'first_pass_cuts.dxf',
'final_perimeter.dxf']:
    print(f"   • {name}")

print(f"\n" + "=" * 60)
print("✓ WORKFLOW COMPLETED - EXACT GEOMETRY PRESERVED!")
print("=" * 60)
```

```
============================================================
MANUFACTURING SUMMARY
ALL DIMENSIONS IN CENTIMETERS - EXACT GEOMETRY
============================================================
```

```
📐 BODY DIMENSIONS:
   Width:  18.0000 cm (7.0866 inches)
   Height: 3.0000 cm (1.1811 inches)

🔗 HINGE SLOT CONFIGURATION (EXACT FROM DXF):
   hinge_1:
     Center X: 4.0000 cm
     Y range: 0.8500 to 2.1500 cm
     Slot size: 0.1000 x 1.3000 cm
   hinge_2:
     Center X: 8.0000 cm
     Y range: 0.8500 to 2.1500 cm
     Slot size: 0.1000 x 1.3000 cm
   hinge_3:
     Center X: 10.0000 cm
     Y range: 0.8500 to 2.1500 cm
     Slot size: 0.1000 x 1.3000 cm
   hinge_4:
     Center X: 16.0000 cm
     Y range: 0.8500 to 2.1500 cm
     Slot size: 0.1000 x 1.3000 cm

📋 LAYER STRUCTURE:
   Layer 1 (Rigid Top): Body with hinge slots removed
   Layer 2 (Adhesive): Same as rigid
   Layer 3 (Flexible): Full body (continuous across hinges)
   Layer 4 (Adhesive): Same as rigid
   Layer 5 (Rigid Bottom): Body with hinge slots removed

📍 ALIGNMENT HOLES:
   Diameter: 0.5 cm
   Spacing from body: 1.0 cm

📋 MANUFACTURING PROCESS:
   1. Cut internal features using layer DXF files
   2. Stack layers using alignment holes for registration
   3. Cut final perimeter to release device

📁 OUTPUT FILES:
   • rigid_top.dxf
   • adhesive_top.dxf
   • flexible.dxf
   • adhesive_bottom.dxf
   • rigid_bottom.dxf
   • combined_cut_pattern.dxf
   • first_pass_cuts.dxf
   • final_perimeter.dxf

============================================================
✓ WORKFLOW COMPLETED - EXACT GEOMETRY PRESERVED!
============================================================
```

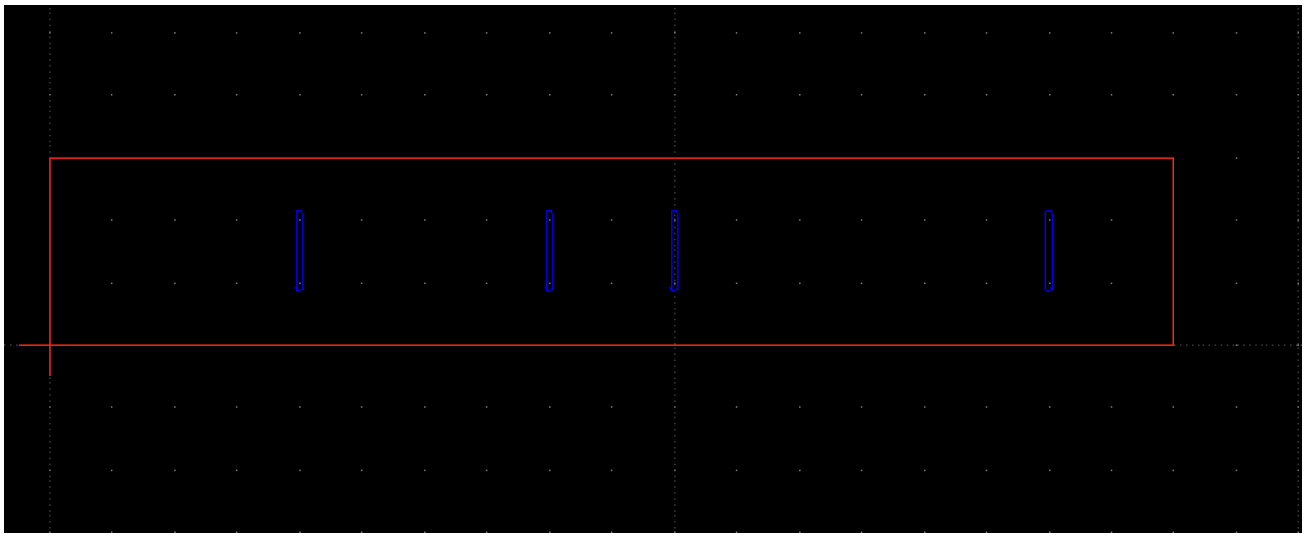
Manufacturing summary and final checklist

This block prints a comprehensive summary of the entire workflow:

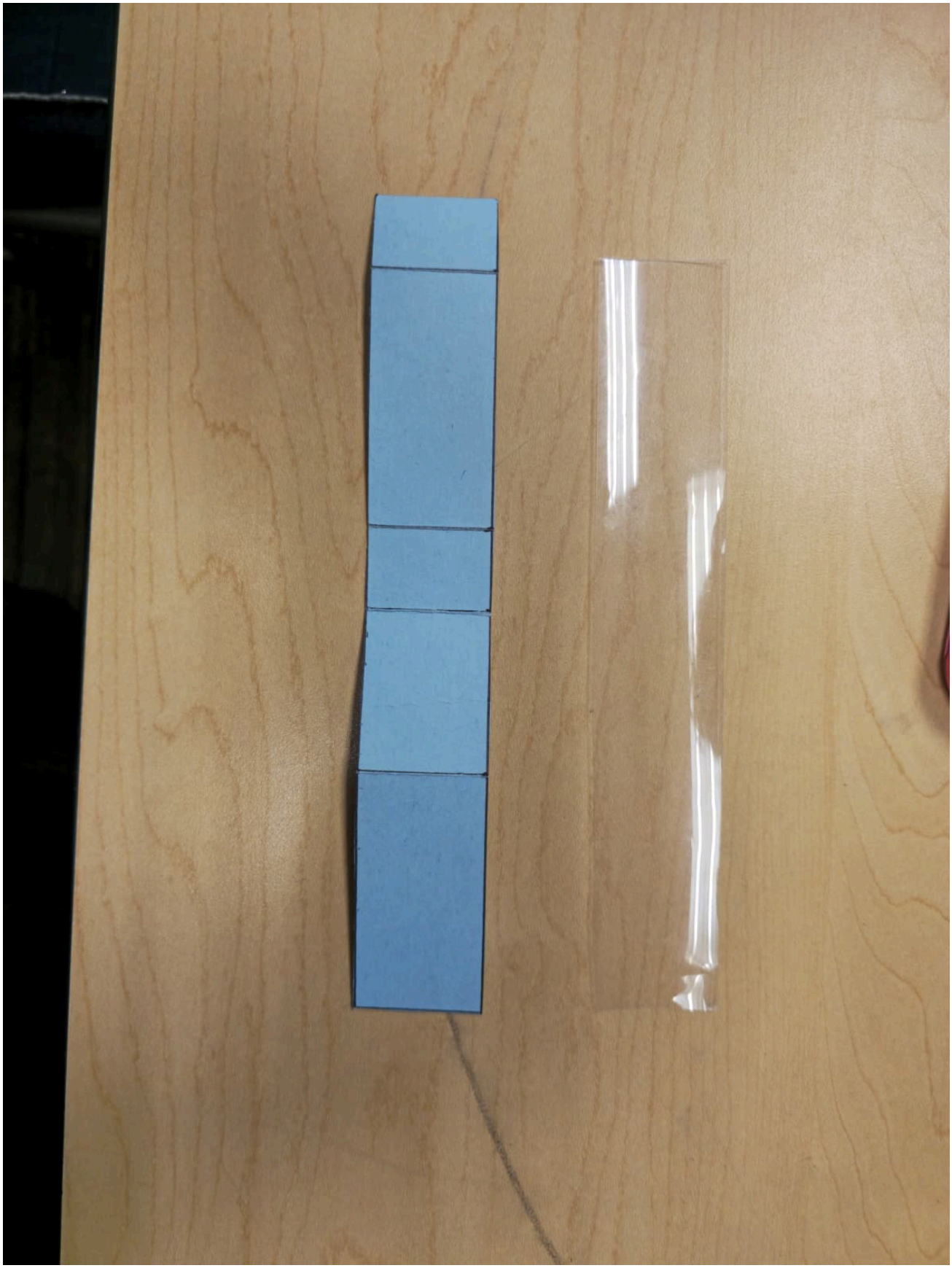
1. **Body dimensions**
    - Width and height in centimeters.
    - The same dimensions converted to inches.
2. **Hinge slot configuration**
    - For each hinge slot:
        - Layer name.
        - Center X coordinate.
        - Y range.
        - Exact slot size (width × height).
3. **Layer structure**
    - Describes each of the 5 layers:
        - Which ones are rigid, adhesive, or flexible.
        - Which geometry they use (body with hinge slots removed vs full body).
4. **Alignment holes**
    - Lists the hole diameter and spacing from the body.
5. **Manufacturing process (recommended sequence)**
    - Step 1: Cut internal features using the individual layer DXF files.
    - Step 2: Stack and register layers using the alignment holes.
    - Step 3: Cut the final perimeter to release the device.
6. **Output files**
    - Prints a bullet list of all key DXF outputs:
        - `rigid_top.dxf`, `adhesive_top.dxf`, `flexible.dxf`, `adhesive_bottom.dxf`, `rigid_bottom.dxf`
        - `combined_cut_pattern.dxf`
        - `first_pass_cuts.dxf`
        - `final_perimeter.dxf`

Finally, it prints a “WORKFLOW COMPLETED” banner confirming that all steps preserve the exact geometry from the input DXF.

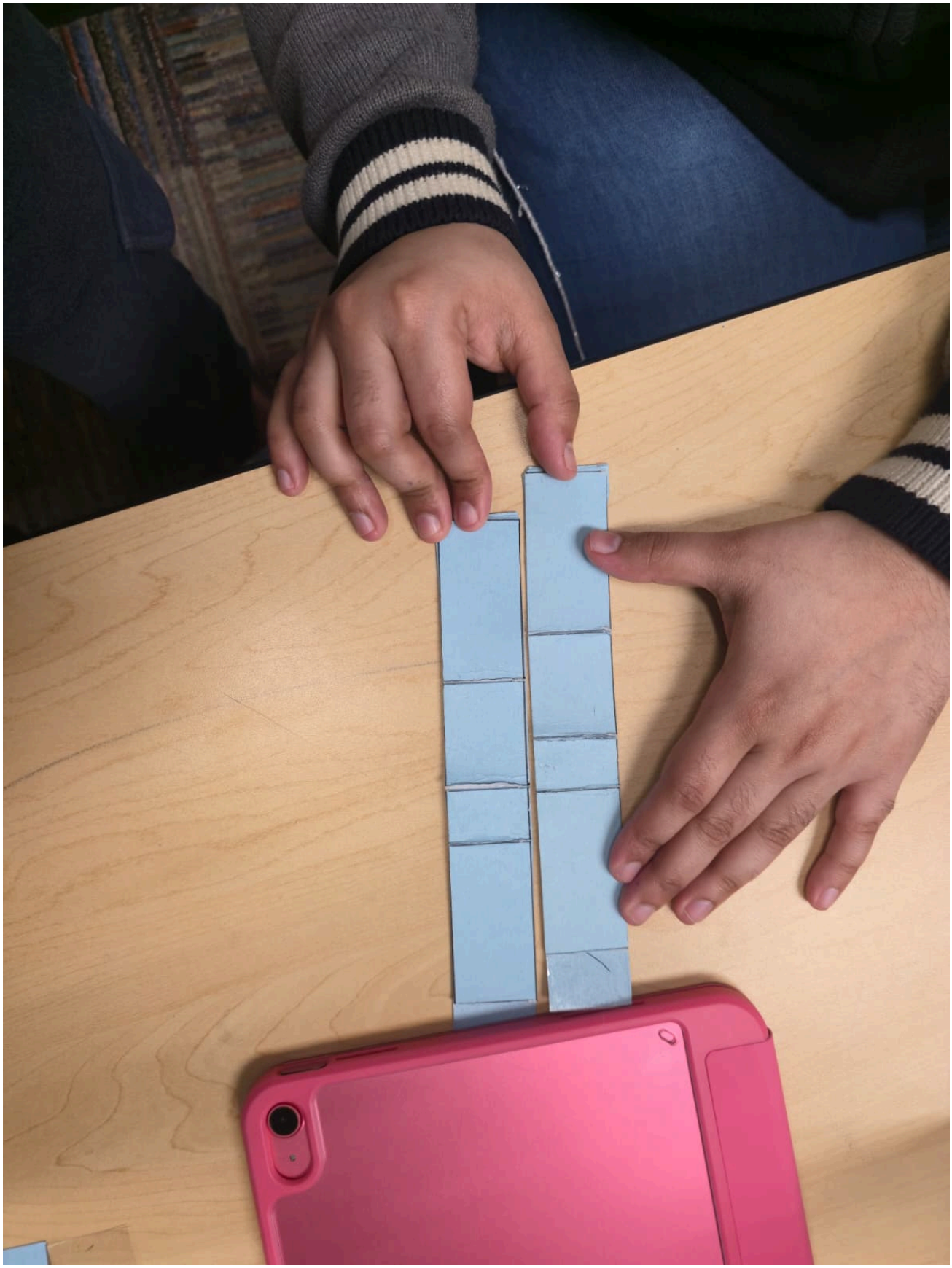
LibreCAD Model



Cutfiles

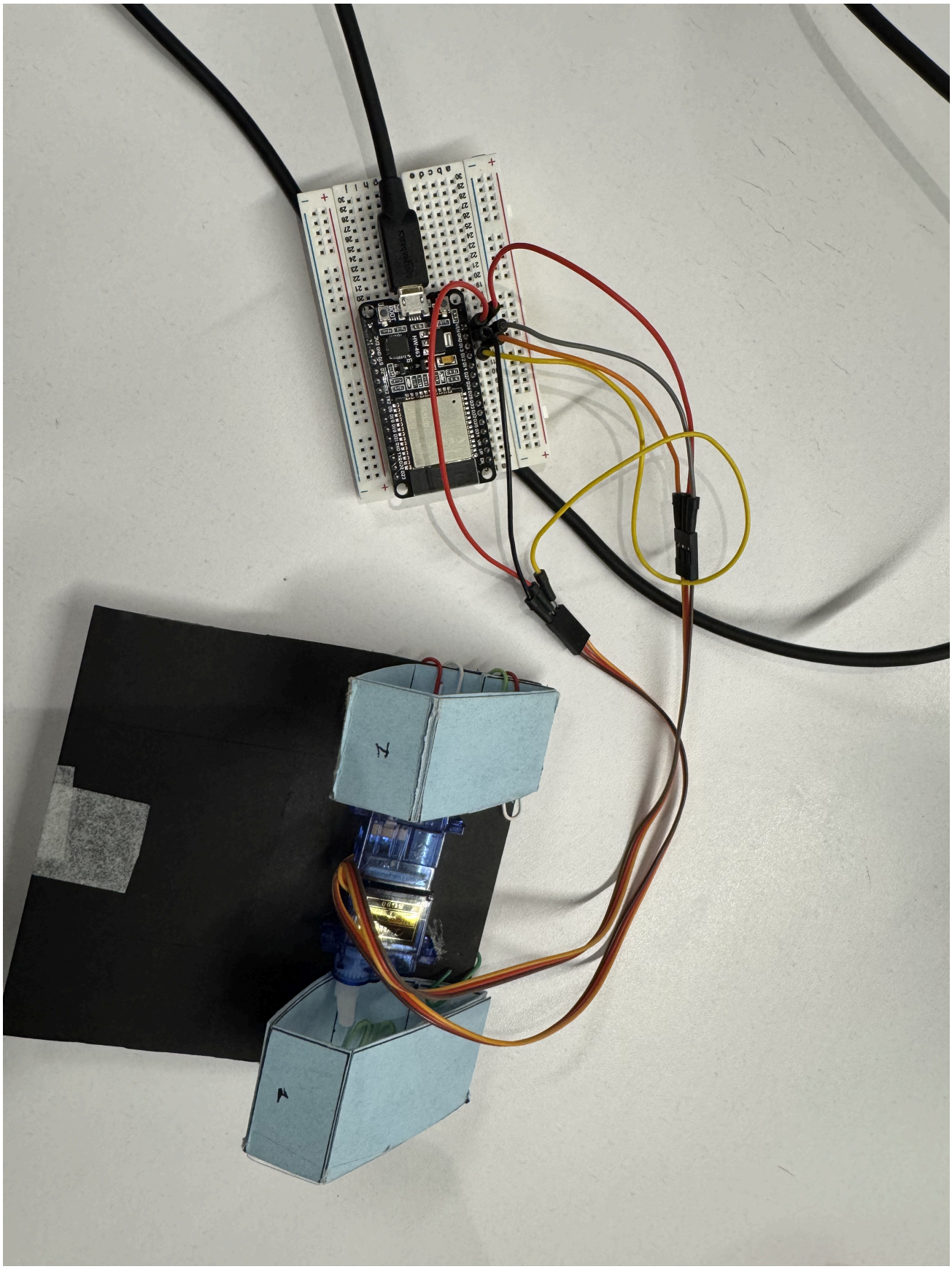


After assembling those we got



Model

Attached servos and test it



Here is the link where we test functionality

https://drive.google.com/file/d/1wPJaO1RoWlfiNutX4qeydSG49zOHpoEw/view?usp=sharing

Design Evaluation and Reflection

Did the Design Perform as Expected?

The constructed prototype exhibited a high degree of fidelity to the intended design specifications. The hinge bands, designed with an approximate width of **1.73 mm** to facilitate a **120°** rotation at a **1 mm** laminate thickness, operated as anticipated, enabling a seamless and repeatable motion without any binding issues. Furthermore, the five-layer laminate configuration successfully preserved the continuity of the flexible core across the joints. The incorporation of alignment holes and support tabs proved effective in achieving precise registration during both the cutting and assembly processes. A minor deviation noted was the excessive adhesive coverage in the hinge zone during the initial stages of fabrication; this issue will be addressed in the comprehensive build by implementing masking techniques for the adhesive layers.

---

(1) Scaling Values

In order to ensure dimensional accuracy in the modeling of geometric features, it is crucial to adopt a metric unit system; specifically, measurements should be conducted in millimeters, as outlined in the referenced workflow materials. This approach can be validated through the utilization of a 10 mm calibration square, which serves as a reliable means of confirming the precision of dimensions post-DXF import.

The hinge-width calculation is intricately linked to both the laminate thickness and the intended range of motion. For instance, a laminate with a total thickness of 1 mm and a desired fold angle of 120° necessitates a hinge width of 1.73 mm. It is imperative to proportionally increase this width for laminates of greater thickness or for broader fold angles, adhering to the relationship defined by the equation $ w = t \tan(\theta/2) $.

Additionally, the calibration of kerf compensation must be approached experimentally; the nominal kerf value of 0.05 mm should be fine-tuned according to the specific characteristics of the cutting tool employed. Inadequate kerf values that are undersized may result in the undesirable fusion of parts.

During the process of downscaling or miniaturizing a design, it is essential to maintain the stability of the parts by ensuring that the widths of bridges and supports are greater than twice the kerf. This parameter is critical for sustaining structural integrity throughout the usage and application of the design.

---

(2) Kinematic Adjustments

To improve the biomimetic performance of robotic legs, several design parameters can be optimized.

Firstly, the **segment lengths** of the leg can be strategically adjusted. Extending the distal link is reminiscent of the energy-efficient locomotion observed in ostriches, as it enhances energy return during movement. Conversely, employing a lighter mid-link, inspired by the jerboa, can facilitate rapid transitions between stance and swing phases, allowing for more agile movements.

Secondly, the **hinge stiffness** is another critical parameter that can be fine-tuned. This can be achieved by altering hinge length or the thickness of the flex layers. Longer or thinner hinges tend to create a more compliant motion, enabling smoother movement, while shorter or thicker hinges contribute to increased stiffness and enhanced load-bearing capabilities.

Finally, the introduction of **passive coupling** mechanisms—such as employing springs or linkages between joints—promotes synchronized motion among adjacent segments. This design approach mimics the energy transfer mechanisms of natural runners, akin to the tendon-like interactions that facilitate efficient movement in biological organisms.

---

(3) Geometric Integration

In the context of integrating a proximal link into a larger robotic system, several design considerations are paramount to ensure functionality and compatibility. The incorporation of standardized mounting holes, such as those with M2–M3 spacing, is essential for facilitating secure attachment and interoperability with various components. Additionally, the inclusion of flat reference surfaces promotes accurate fixture alignment, thereby enhancing the overall assembly process.

Furthermore, the design can be optimized for tendon-driven actuation by integrating cable-routing channels or small eyelets near the joints. This feature will not only streamline the actuation mechanism but also contribute to a more organized and efficient system layout.

To enhance usability, it is advisable to include mirror symmetry options for both left and right limbs. Clear alignment markers should be implemented to assist assemblers in achieving precise configurations. Such design considerations will significantly contribute to the robustness and adaptability of the robotic system in diverse applications.

---

(4) Manufacturing Considerations

Adhesive masking plays a critical role in the structural integrity of hinge regions within composite materials. It is essential to remove adhesive layers from these areas to mitigate

the risks of stiffness and fatigue failure. This can be effectively accomplished by customizing adhesive sheets to conform to the outlines of the rigid layers.

Furthermore, implementing stress relief features, such as rounded end-reliefs (small circular cutouts) at the ends of hinges, is necessary for uniform stress distribution. This design element not only enhances the performance of the hinge but also contributes to an increased cycle life.

When it comes to support optimization, employing fewer and thicker bridging elements is advisable to enhance stability. It is also crucial to ensure that the width of these supports exceeds the kerf by an appropriate safety margin, which prevents structural integrity issues during the cutting process.

In terms of cut sequencing, adopting a two-pass fabrication strategy is recommended. This involves performing internal features and hinge bands in the initial pass, followed by perimeter release in the subsequent pass. Such a methodology ensures the preservation of registration and the integrity of the components throughout the fabrication process.

Lastly, scalability considerations should be taken into account for prototyping purposes. A single-layer flexible hinge variant can be produced for rapid prototyping, whereas the full five-layer laminate structure remains superior for applications requiring enhanced durability and biomechanical fidelity.

Summary

The current prototype successfully achieves the main design objectives: predictable hinge motion, accurate registration, and manufacturability within the limits of the laminate process. Future iterations will focus on refining hinge scaling, enhancing joint coupling for dynamic gait reproduction, and improving mounting geometry for better system integration. Together, these modifications will enable more efficient energy storage and agile multi-gait motion, aligning more closely with the agility of the jerboa and the tendon-mediated efficiency of the ostrich as described in the biomechanical literature.

Conclusion

This notebook comprehensively outlines the complete workflow for foldable robotics as detailed in the accompanying report. It begins with the formulation of bio-inspired design objectives, drawing insights from the agility of the jerboa and the efficient locomotion of the ostrich. Using the methodologies discussed in Chapters 52–54, a CAD model is transformed into a multi-layer laminate structure.

The workflow involves the importation of DXF geometry, computation of hinge parameters, definition of support structures, and preparation of cutting layouts for both first and second

passes. By incorporating elements such as holes, hinges, bridges, alignment features, and kerf compensation, a manufacturable design for a foldable leg mechanism is achieved. The notebook is meticulously organized into sections, with detailed explanations that elucidate the rationale behind each step. This structure effectively bridges the connection between the code, the foundational biomechanical inspiration, and the fabrication requirements.