Project dates: Jun 2025-Present

← Back to Home

Knitted Soft Sensors

Programmable textile sensing patches for soft robotic systems.

> **Note (pre-publication):** This project is part of ongoing research. To protect unpublished work, I'm not sharing the full knit pattern specification or the complete Knitout/JS codebase publicly. This page focuses on the fabrication workflow, materials stack, and validation methodology. Select details can be shared after publication or in a research interview.

What I built: A computational pipeline to knit custom resistive pressure sensors.

Why it matters: Soft robots need sensors that deform with them without breaking contact.

Proof: Integrated into a multi-zone robotic sleeve interacting with physical objects.

Problem / Goal

Soft robots need sensors that can bend, twist, and compress without breaking contact or adding rigid hardware. Textiles are naturally compliant and body-safe, but making them behave like reliable sensors requires treating knit structure as a **structure–property design problem**.

Project Specs

Role: Research Assistant

Stack: JS, Knitout, Python

Outcome: Functional sensing skin

Code withheld (IP)

My Contribution

I developed the entire fabrication pipeline, moving from manual pattern editing to a script-driven workflow. I designed the knitted structure integrating conductive yarn and silicone tubing for durability. I validated the sensors on benchtop rigs and integrated them into a full robotic sleeve demonstration.

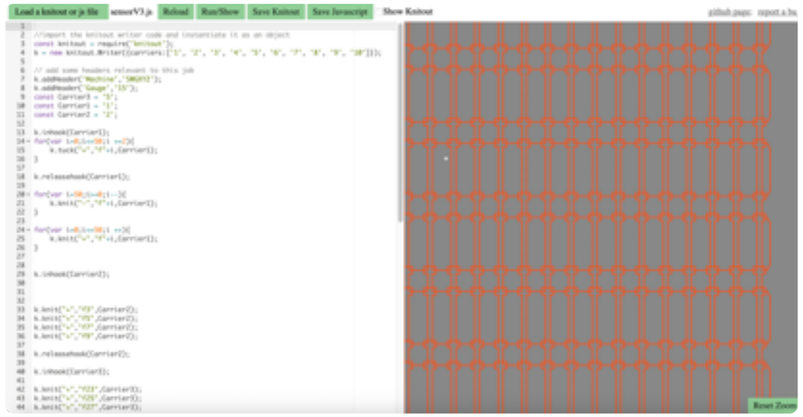
Technical Approach

Fabrication Platform & Materials

I monitored fabrication on a **Shima Seiki SWG091N2** WholeGarment machine. The material stack combines conductive yarn for sensing with silicone tubing integrated directly into the knit structure.

Computational Pipeline (JS → Knitout)

I built a JavaScript-based pipeline to generate **Knitout** machine instructions. This allows for repeatable pattern generation from high-level parameters.



Scripted generation workflow (JS → Knitout → Machine).

Validation / Results

Validation is ongoing with cyclic deformation tests (10k cycles) to measure sensitivity, hysteresis, and drift. Early results show reliable signal response.

Patch-level integration

Multi-zone sensing

Lessons + Next Steps

- Scale the sensor array to full-arm coverage.
- Explore alternative conductive materials (e.g., graphene-coated yarn).
- Automate calibration using machine-learning models.

Gallery

Shima Seiki SWG091N2 Fabrication

```
;!knitout-2
;;Carriers: 1 2 3 4 5 6 7 8 9 10
;;Machine: SWGXYZ
;;Gauge: 13
inhook 2;!source:-1
tuck + f0 2;!source:-1
tuck + f2 2;!source:-1
tuck + f4 2;!source:-1
tuck + f6 2;!source:-1
tuck + f8 2;!source:-1
tuck + f10 2;!source:-1
tuck + f12 2;!source:-1
tuck + f14 2;!source:-1
tuck + f16 2;!source:-1
tuck + f18 2;!source:-1
tuck + f20 2;!source:-1
tuck + f22 2;!source:-1
tuck + f24 2;!source:-1
tuck + f26 2;!source:-1
tuck + f28 2;!source:-1
tuck + f30 2;!source:-1
tuck + f32 2;!source:-1
tuck + f34 2;!source:-1
tuck + f36 2;!source:-1
tuck + f38 2;!source:-1
tuck + f40 2;!source:-1
releasehook 2;!source:-1
knit - f40 2;!source:-1
knit - f39 2;!source:-1
knit - f38 2;!source:-1
knit - f37 2;!source:-1
knit - f36 2;!source:-1
knit - f35 2;!source:-1
knit - f34 2;!source:-1
```

Knitout low-level instruction excerpt

Project dates: Sept 2025-Present

← Back to Home

Thermal Wearables

Knitted heating interfaces with closed-loop PID control for safe regulation.

Note (pre-publication): This project is part of ongoing research. To protect unpublished work, I'm not sharing the full knit pattern specification or the complete Knitout/JS codebase publicly. This page focuses on the fabrication workflow, materials stack, and validation methodology. Select details can be shared after publication or in a research interview.

What I built: Closed-loop thermal regulation system for wearable knit heaters.

Why it matters: Practical wearables need fast warm-up and stable safety controls despite body movement.

Proof: Validated PID implementation with real-time temperature tracking plots.

Problem / Goal

Wearable heating isn't just about turning power on. Practical systems need fast warm-up without overshoot and stable regulation under changing contact and airflow conditions. This project treats the **textile heater + fabrication constraints + control loop** as one coupled system.

My Contribution

I built the textile heating elements and integrated them into soft substrates. I developed the custom Arduino firmware for closed-loop PID control, ensuring safety and stability. I also built the Python-based testing workflow to log and visualize real-time thermal performance.

Technical Approach

TL;DR

Role: Research Assistant (Firmware, Fabrication, Testing)

Methods: PID Control, Wearable Systems, Embedded Design

Tools: Arduino, JavaScript, Python, C++, Shima Seiki

Outcome: Stable, safe wearable heating prototypes (mitten, cup sleeve, etc.).

Status: Pre-publication. Heater routing redacted.

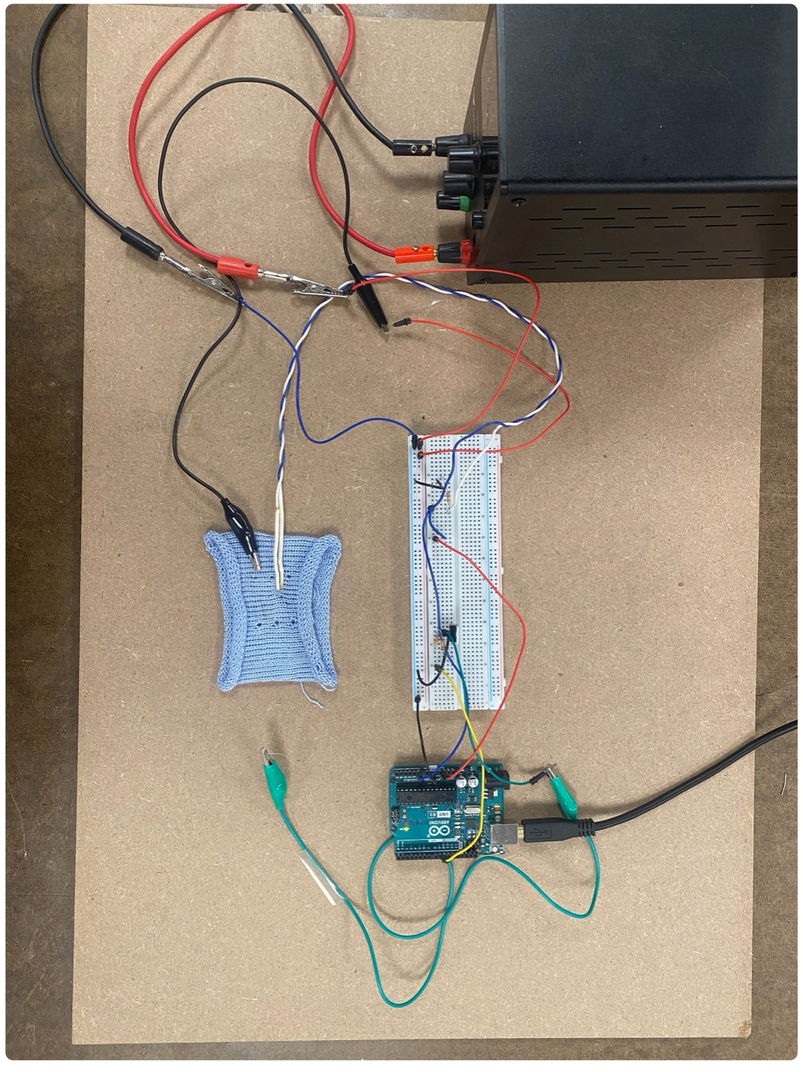
View GitHub Repo →

System Design & Fabrication

The heaters are knitted on a Shima Seiki SWG091N2 machine using conductive yarn routing (proprietary) integrated into a woolen substrate. The system includes a microcontroller-based driver and thermistor feedback.

Control & Software

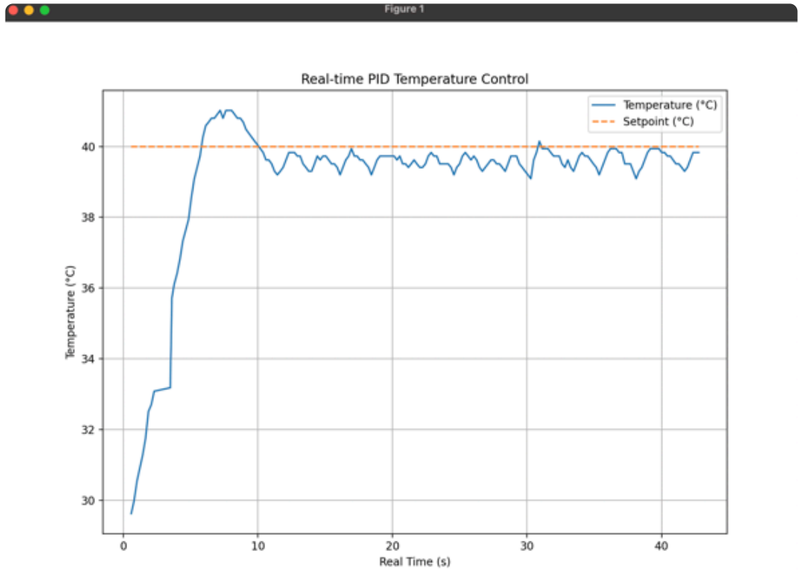
I implemented a real-time PID loop on Arduino to track temperature setpoints. The Python plotting suite reads serial streams to visualize step response and regulation error, enabling rapid tuning of Kp, Ki, and Kd gains.



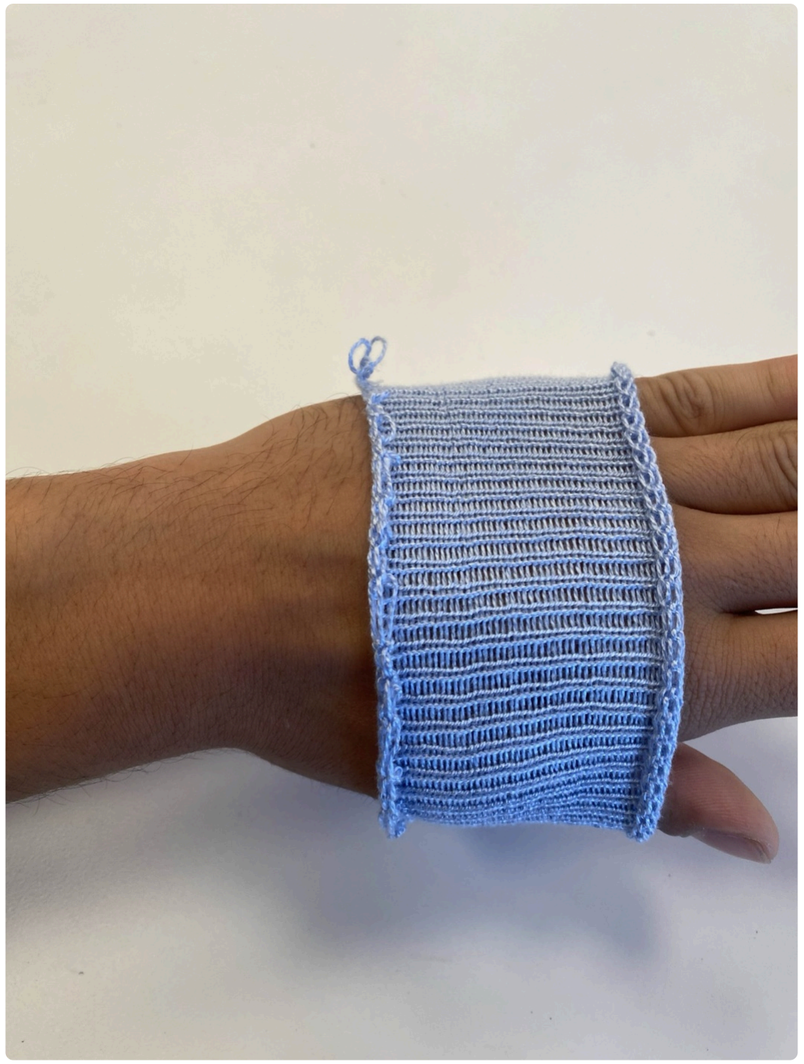
Benchtop setup for closed-loop control tuning.

Validation / Results

The system successfully tracks target temperatures with minimal overshoot after tuning. The plots below show the step response during a warm-up cycle.



Real-time PID temperature control trace: measured vs setpoint.



Wearable wrist sample.

Lessons + Next Steps

- improve response under changing contact conditions (airflow, pressure)
- move from breadboard prototype to a more wearable electronics package (robust wiring/strain relief)
- expand evaluation (repeatability across runs, drift, and perception thresholds)

Gallery

Knitout simulation

Fabrication on SWG091N2.

Project dates: Jul 2025-Oct 2025

← Back to Home

Mycelium Knit Modules (16 Cubits)

Modular knitted formworks for on-site mycelium architectural assembly.

What I built: Parametric knitted formworks to guide mycelium growth for a biophilic structure.

Why it matters: Demonstrates scalable biofabrication using breathable textile scaffolds.

Proof: Successfully deployed in the **16 Cubits** Sukkah Design Festival build.

Problem / Goal

This project was part of **16 Cubits**, an architectural competition to build a "biophilic sukkah" using local, bio-based materials. The challenge was to create a structural system that could be grown from mycelium (mushroom root network) but assembled rapidly on-site.

My goal was to design **modular textile formworks** that would contain the loose mycelium substrate, allowing it to bind together while providing the necessary aeration for growth.

TL;DR

Role: Research Assistant

Methods: Biofabrication, Knitting

Tools: Knitout, Shima Seiki, Dat

Outcome: Functional modular system

View GitHub Repo →

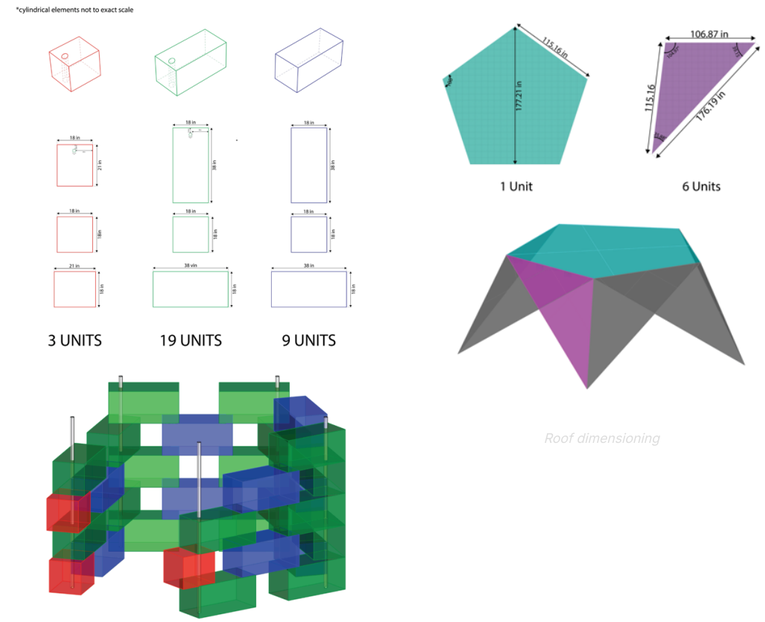
My Contribution

I led the **computational knitting pipeline**. I designed the parametric logic for the three module types (A, B, C), generated the machine code (Knitout → Dat), and managed the fabrication on Shima Seiki machines. I also coordinated the on-site assembly strategy.

Technical Approach

1. Module Logic & Standardization

We defined three module types to allow for structural variation: **Module A** (Standard), **Module B** (Corner), and **Module C** (Cap).

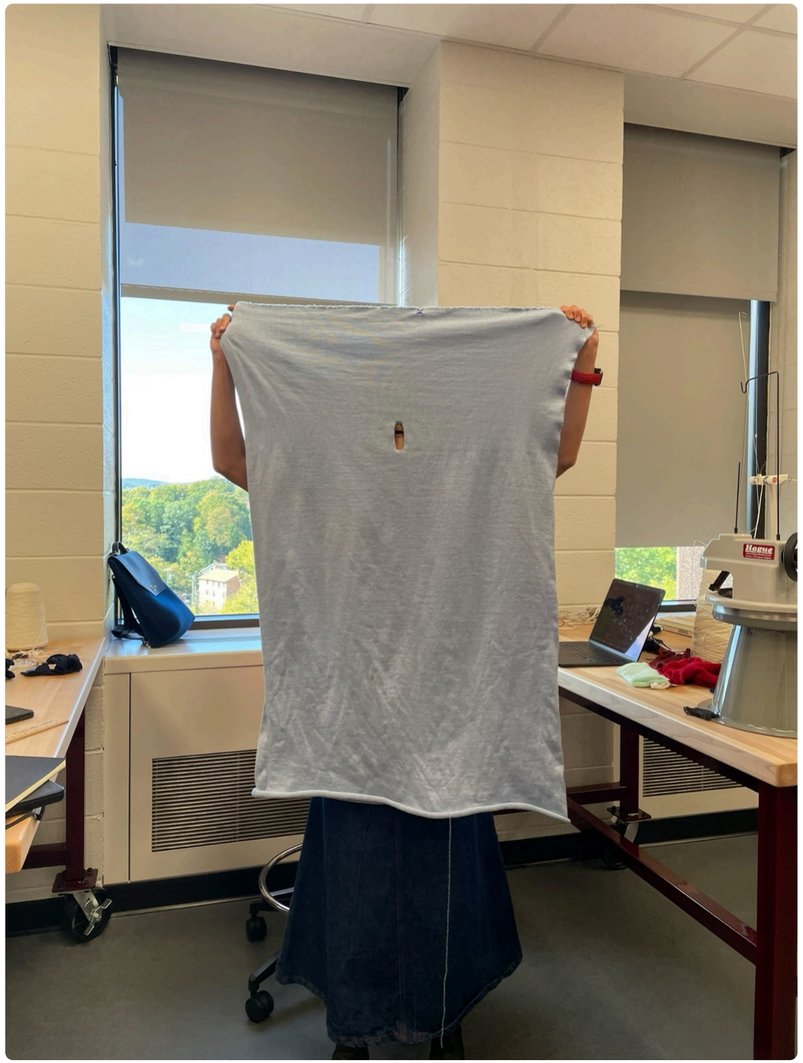


System breakdown: A, B, C.

Roof dimensioning

2. Computational Fabrication

The pipeline prioritized reproducibility. I used a parametric workflow: *Sketch → Knitout Generation → Machine Execution*. This allowed us to iterate on stiffness and scale rapidly without manually redesigning each pattern.

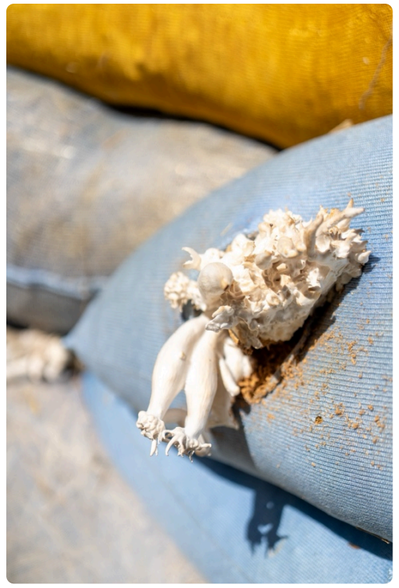


Toolchain: Parameter edits → Knitout → Dat → Knitpaint → Machine.

3. Biomaterial Integration

The knit structure had to be porous enough for hyphal penetration but tight enough to hold the substrate. We tested various gauges and tensions to find the optimal "breathable scaffold."

Textile interface allowing mycelium breathability.



Fruiting bodies emerging through the knit.

Validation / Results

The system was validated through a full-scale on-site build. The modules were successfully filled, stacked, and assembled into a self-supporting wall structure.

On-site wall assembly.

Lifting and placing modules.

Lessons + Next Steps

Material Uncertainty: Working with living materials meant that not all modules grew at the same rate. Future iterations would include better humidity control during the colonization phase.

Next Steps: developing "pre-colonized" yarns where the mycelium is embedded directly into the fiber before knitting, rather than filling a formwork post-fabrication.

Links

GitHub Repository 16 Cubits Festival

Project dates: Aug 2025

← Back to Home

Zelij Digital Jacquard

Algorithmic translation of traditional Moroccan geometric patterns into machine-knitted jacquard fabric.

What I built: A computational pipeline to convert binary patterns into double-bed jacquard knit structures.

Why it matters: Bridges heritage craft with digital fabrication, enabling rapid translation of complex geometries.

Proof: Successfully fabricated reversible jacquard textiles on Shima Seiki industrial machines.

Problem / Goal

Zelij (زليج) refers to the intricate geometric tilework that has defined Moroccan architecture for centuries. These patterns—often featuring tessellating stars and polygons—embody a sophisticated mathematical design language.

The goal of this project was to translate these traditional patterns into contemporary digital fabrication. The core challenge was maintaining pattern fidelity while translating high-resolution geometric designs into the discrete, physical constraints of industrial knitting (mapping pixels to stitches across two needle beds).

Project Specs

Role: Computational Designer

Methods: Jacquard, Pixel Mapping

Tools: JS, Knitout, Shima Seiki

Outcome: Algorithmic Knitting Pipeline

My Contribution

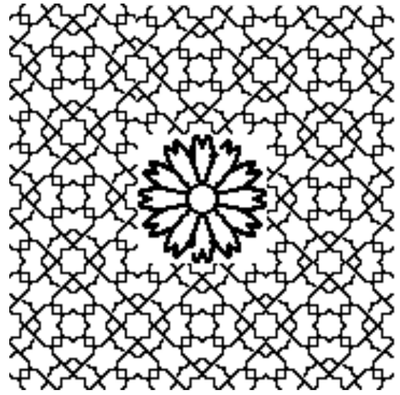
I developed the end-to-end computational pipeline (`chamomile.js`) that parses binary pattern images and generates low-level machine instructions (Knitout). I also managed the material fabrication on Shima Seiki machines, iterating on tension and carrier settings to achieve a stable double-jacquard structure.

Technical Approach

1. Pattern Processing

The workflow begins with selecting a tiling zelij motif (e.g., an eight-pointed star). The pattern is algorithmically multiplied to create a seamless repeat and formatted as a binary (2-color) PNG map, where pixel dimensions directly correspond to fabric stitches and courses.

Source zelij tile motif



Binarized pattern map

2. Pixel-to-Stitch Algorithm

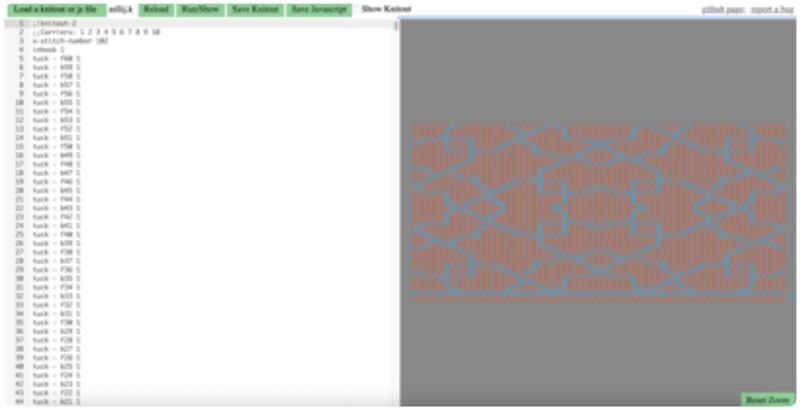
I wrote a custom JavaScript tool to process the image data. The algorithm performs a serpentine traversal (zigzag) of the image pixels and issues `knit` commands.

Crucially, it implements **Double Jacquard** logic: for every pixel row, the machine makes two passes (one for the backing color, one for the pattern color), ensuring a reversible fabric with no long floats.

```
    // Example of color mapping logic
    function do_carrier(carrier, character) {
        if (side[carrier] === '+') {
            // Knit Left
            for (let i = max; i >= min; i -= 1) {
                if (color_choice[i-min] === character) {
                    console.log(`knit - f${i} ${carrier}`);
                } else {
                    console.log(`knit - b${i} ${carrier}`);
                }
            }
            side[carrier] = '-';
        } else {
            // Knit Right (logic mirrored)
            // ...
        }
    }
```

3. Knitout & Visualization

The script outputs **Knitout** code, a machine-agnostic knitting language. I verified these instructions using a visualizer to check for dropped stitches or carrier collision issues before physical fabrication.



Knitout visualization showing front/back bed stitch assignments.

Validation / Results

The pipeline was validated by producing physical textile artifacts. The resulting fabric exhibits high pattern fidelity with crisp geometric lines. The double-bed structure provides stability and thickness suitable for upholstery or soft goods.

Shima Seiki machine fabricating the jacquard.

Final fabric showing reversible jacquard structure.

Lessons + Next Steps

Key Insight: Serpentine traversal (zigzag knitting) significantly reduced carriage travel time compared to unidirectional knitting, optimizing production speed.

Next Steps: The current system is limited to 2-color binary patterns. Future work will extend the algorithm to support 3+ color jacquard using multiple yarn carriers and dithering techniques for gradients.

Links

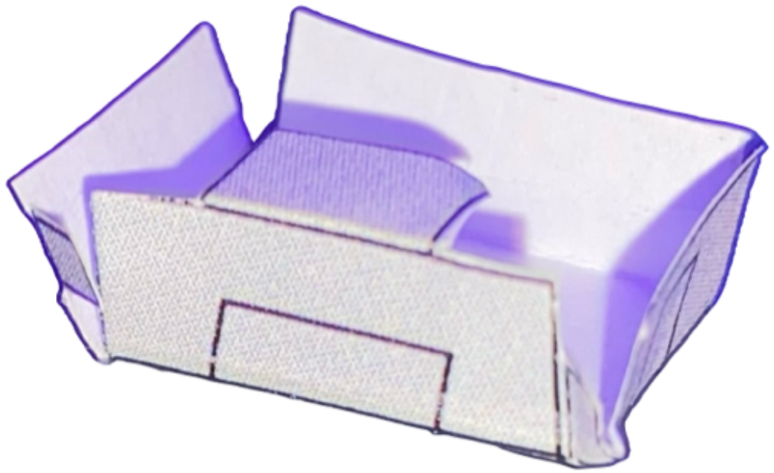
Code/Repository available upon request.

Project dates: Nov 2025

← Back to Home

Domain Specific Language for Sheet Metal

Computational Design, DSLs, Fabrication



What I built: An embedded Python DSL for designing folded sheet metal structures (e.g., enclosures).

Why it matters: Automates the tedious translation from 3D design intent to 2D fabrication patterns.

Proof: Automatically generates laser-ready SVGs from high-level Python code.

Problem / Goal

Designing complex folded sheet metal structures (like electronics enclosures or origami-inspired robots) manually in CAD is often tedious and prone to error, especially when calculating unfold patterns.

The goal was to create a **Domain Specific Language (DSL)** embedded in Python to automate this process, allowing users to define structures logically (as a hierarchy of tabs) rather than geometrically drafting every edge.

TL;DR

Role: Developer / Designer

Methods: DSL Design, Fabrication

Tools: Python, SVGWrite

Outcome: Design-to-Production Pipeline

My Contribution

I built the core logic and interpreter for the DSL:

- **Embedded DSL:** Implemented as a Python library where tabs are objects and relationships are defined via function calls.
- **Tree-Based Traversal:** Designed the interpreter to traverse the component tree recursively to compute global coordinates from local relative transformations.
- **Fabrication Export:** Automatically built the unfolded 2D geometry and exported it as an SVG for laser cutting, ensuring physical connectivity.

Technical Approach

1. The Tab Logic (Embedded DSL)

The `Tab` class is the fundamental primitive. The `generate_child_tab` function handles the complexity of attaching a new geometric piece to a parent edge, managing spatial offsets and bend angles.

```
@dataclass
class Tab:
    """ A structure that represents a tab and a bend with respect to the parent tab. """
    parent: Optional["Tab"]
    children: list["Tab"]
    position: np.ndarray        # Root position
    bend_angle: Optional[float] # Angle for 3D folding

def generate_child_tab(parent, side, offset, width, length, angle=0.0, bend_angle=np.pi/2):
    """
    Generate a child tab attached to a parent tab.
    side: 0=top, 1=right, 2=bottom, 3=left
    """
    tab = Tab(
        parent=parent,
        children=[],
        width=0.0,      # Computed from geometry
        side=side,
        offset=offset,
        child_width=width,
        child_length=length,
        bend_angle=bend_angle
    )
    parent.children.append(tab)
    return tab
```

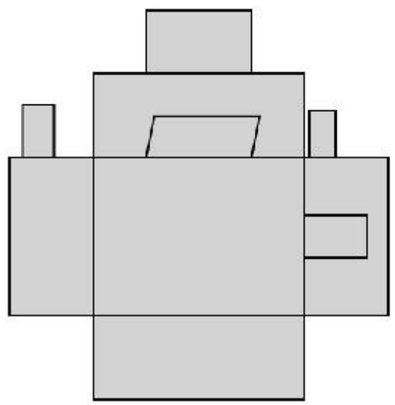
2. Usage Example: Defining a Structure

Users can define complex, nested geometries with just a few lines of code. This script defines a root base and attaches walls to create a box-like structure. The semantic clarity improves maintainability over raw coordinate lists.

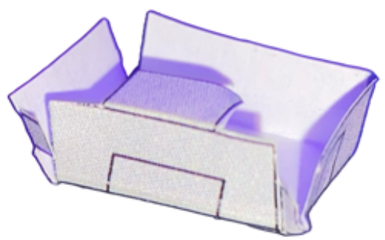
```
# Define the base of the structure
root = generate_root_tab(width=50, length=50)

# Attach four walls to the base
north_wall = generate_child_tab(root, side=0, offset=0, width=50, length=30, bend_angle=np.radians(90))
east_wall  = generate_child_tab(root, side=1, offset=0, width=50, length=30, bend_angle=np.radians(90))
south_wall = generate_child_tab(root, side=2, offset=0, width=50, length=30, bend_angle=np.radians(90))
west_wall  = generate_child_tab(root, side=3, offset=0, width=50, length=30, bend_angle=np.radians(90))

# Export the flat pattern for laser cutting
draw_svg(root, "output_pattern.svg")
```

Validation / Results

The system was validated by generating laser-cut patterns for physical prototypes. The folded structures perfectly matched the digital 3D visualization, confirming the accuracy of the unfolding algorithm.



Generated 2D flat pattern for laser cutting (SVG output).



3D visualization of the folded sheet metal structure.

Lessons + Next Steps

Key Insight: Recursion is a natural fit for physical assemblies that branch out from a core component. However, managing relative coordinate frames (local-to-global) requires careful matrix transformations.

Next Steps: Adding collision detection to prevent the user from designing physically impossible folds (self-intersection).

Links

Code/Repository available upon request.