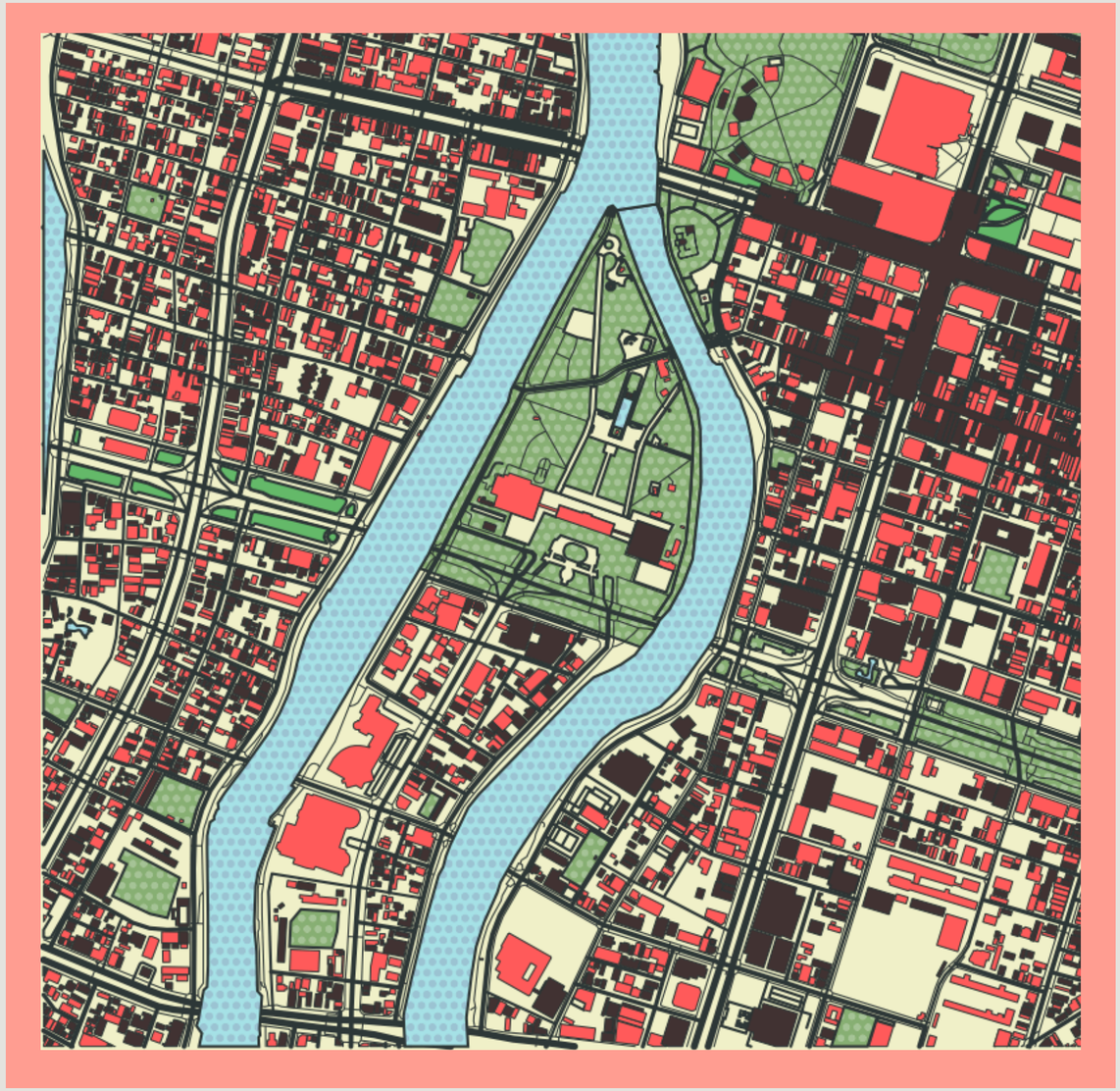
PYCON JAPAN 2025

DRESS CODE

HOW PYTHON WRITES THE FUTURE OF FASHION.

PRESENTED BY

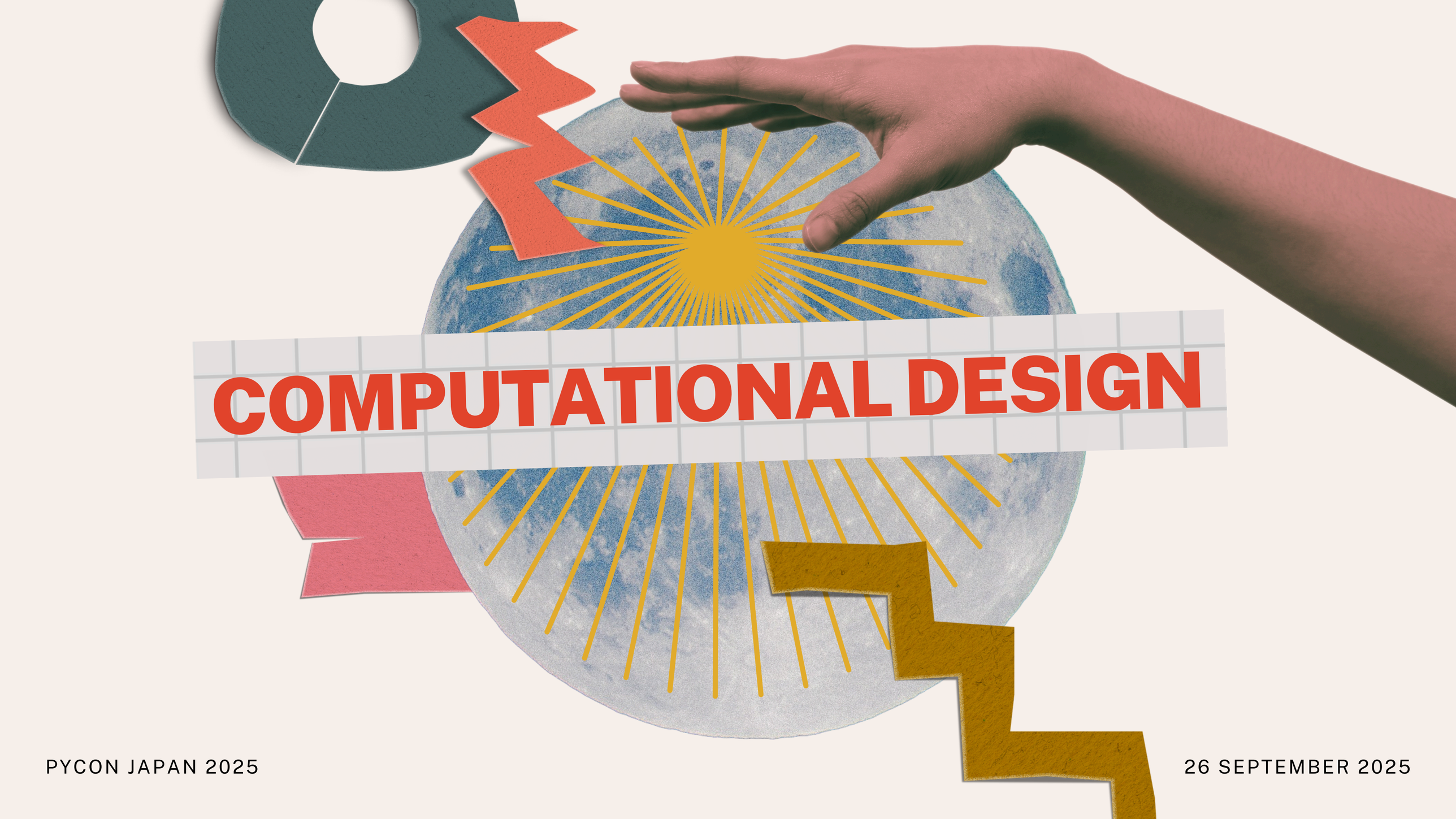
NEERAJ PANDEY
HITESH KHANDELWAL



https://prettymapp.streamlit.app/

From Data to Design

Turn anything into wearable art.

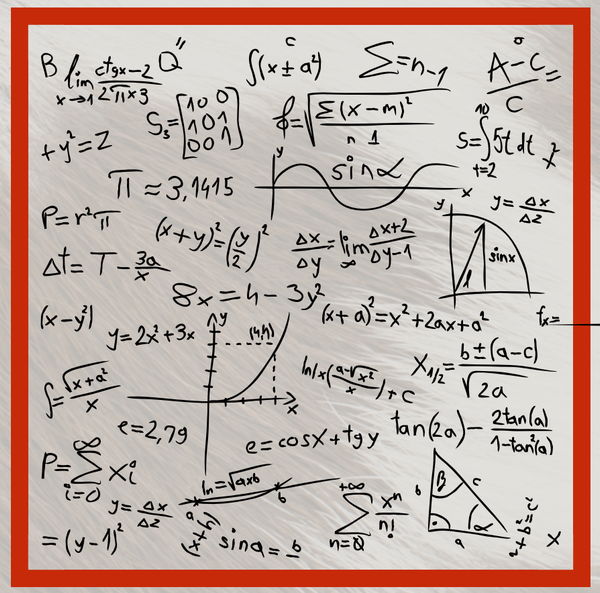


PYCON JAPAN 2025

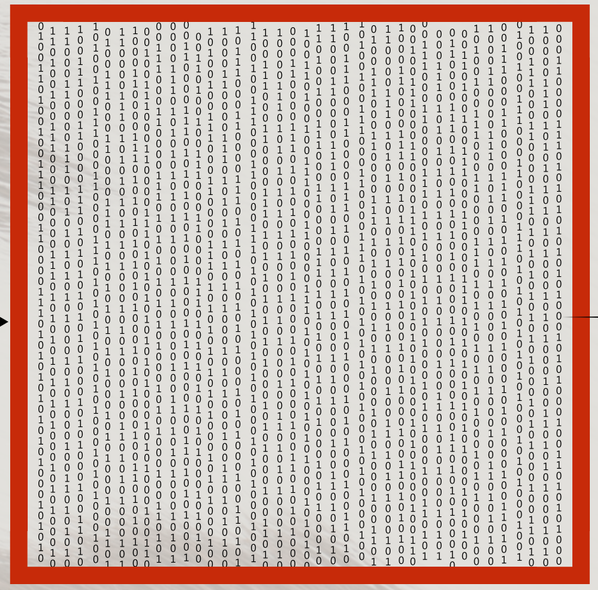
26 SEPTEMBER 2025

COMPUTATIONAL DESIGN

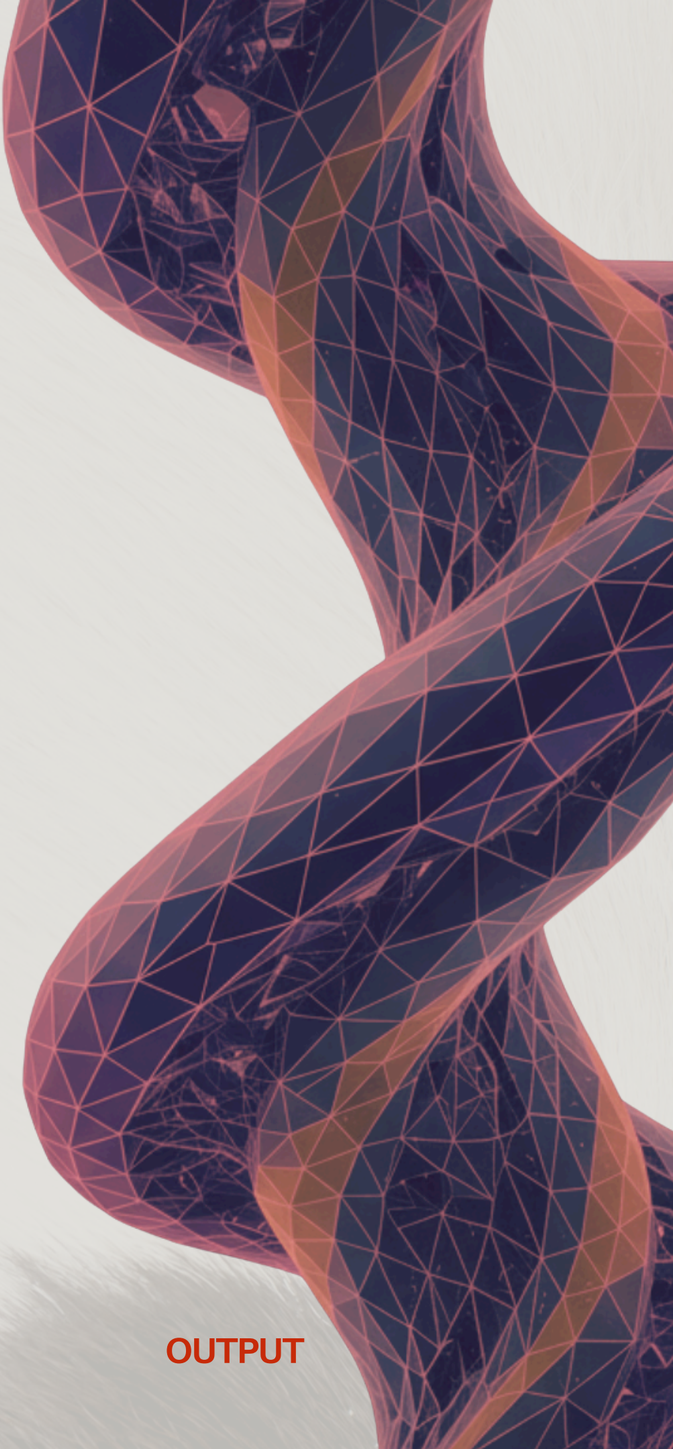
It uses **algorithms**, **math,** and **code** to generate, optimize, and iterate on forms —blending ***engineering, art, and technology.***



INPUT



PROCESS



OUTPUT

NANSHA SPORTS COMPLEX,
RENDER, ZAHA HADID ARCHITECTS

COMPUTATIONAL DESIGN TRANSFORMS ARCHITECTURE BY USING ***ALGORITHMS*** TO CREATE COMPLEX, EFFICIENT STRUCTURES—OPTIMIZING FOR FORM, FUNCTION, *SUSTAINABILITY*, AND AESTHETICS.

Spotlight on the architectures (e.g., Zaha Hadid's fluid buildings).

Architecture uses **generative algorithms** (e.g., genetic evolution or noise functions) for sustainable innovations, like **zero-waste facades** or adaptive urban planning.

IN FASHION

PARAMETRIC CUSTOMIZATION:

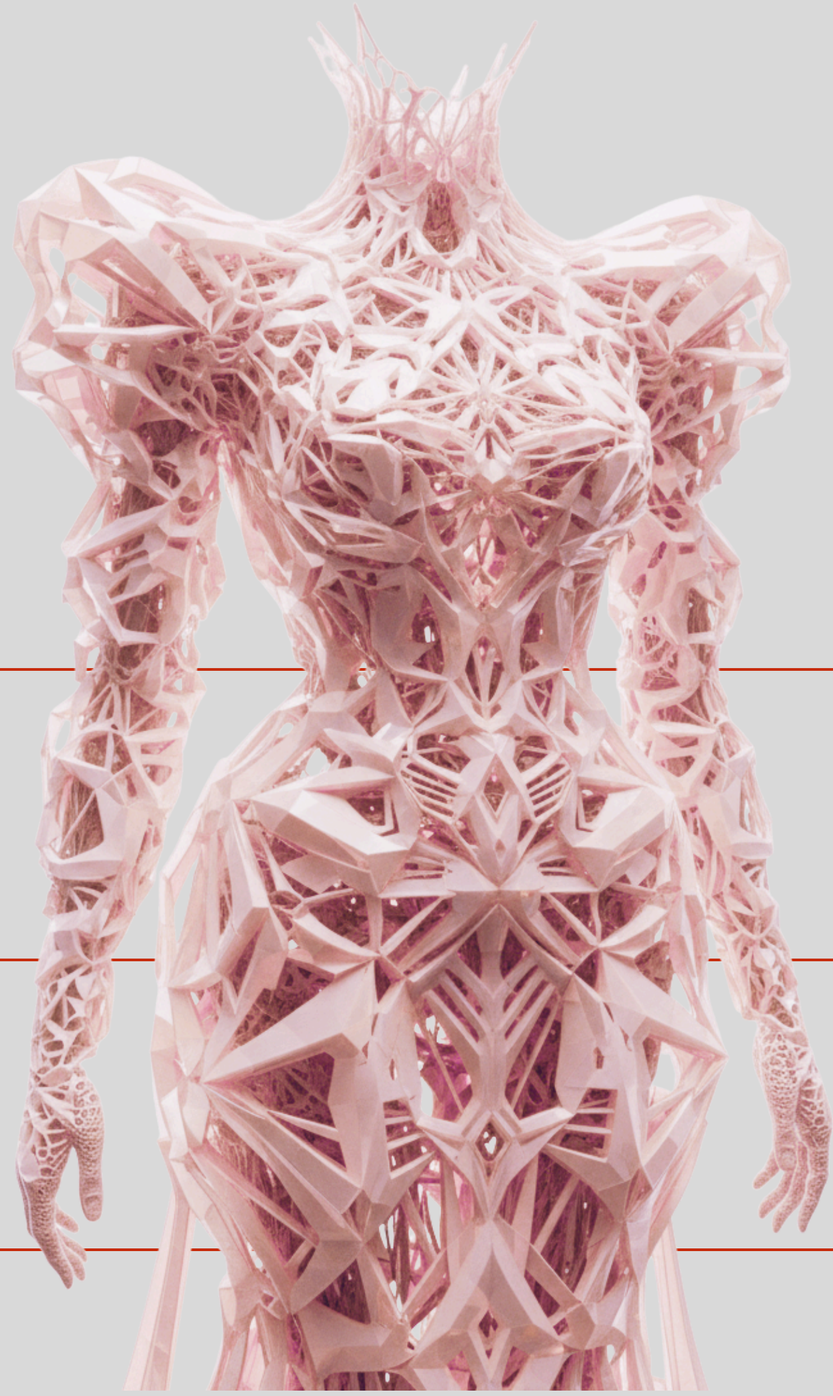
ALGORITHMS **OPTIMIZE SHAPES** FOR FIT AND STYLE, E.G., GENERATING INFINITE DRESS VARIATIONS FROM USER DATA LIKE BODY SCANS OR PREFERENCES - ***REDUCING WASTE IN PRODUCTION***.

GENERATIVE TEXTILES

CREATE INTRICATE PATTERNS FOR FABRICS, SUCH AS **LASER-CUT MOTIFS OR NOISE-BASED PRINTS**, INSPIRED BY NATURAL FORMS (E.G., VORONOI FOR ORGANIC JEWELRY).

SUSTINABLE OPTIMIZATIONS

USE NESTING ALGORITHMS TO MINIMIZE FABRIC SCRAPS IN BAG OR GLOVE DESIGNS; ALIGNS WITH 2025 TRENDS FOR ECO-FRIENDLY, MODULAR ACCESSORIES.



HISTORY & EVOLUTION

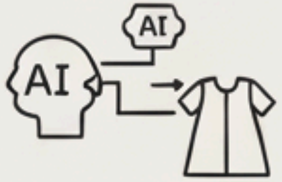
1960s

Ivan Sutlehleand's
– Interactive Design

1980s

Fractals & Chaos Theory
– Generative Art

2000s



Parametric Tools,
Early AI in Fashion
(GANs)

2025

AI Personalization,
Zero-Waste Algos

GENERATIVE ART & PARAMETRIC DESIGN IN COMPUTATIONAL DESIGN

ALGORITHMS TURN SIMPLE RULES INTO **DIVERSE, *UNPREDICTABLE* ARTWORK**S - DEMOCRATIZING DESIGN FOR FASHION INNOVATORS, ENABLING UNIQUE PIECES WITHOUT MANUAL REPETITION.

PYCON JAPAN 2025

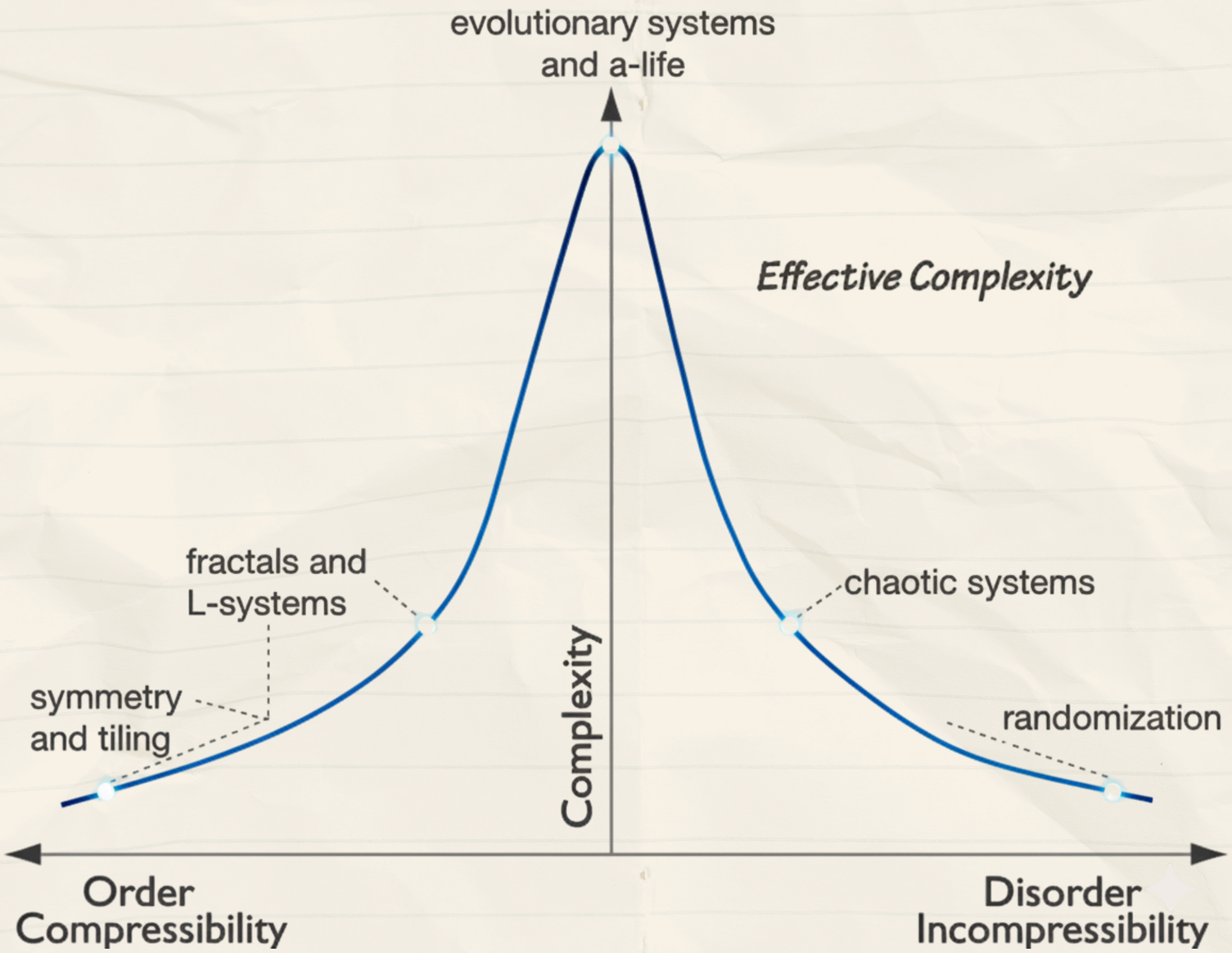
KEY PRINCIPLES

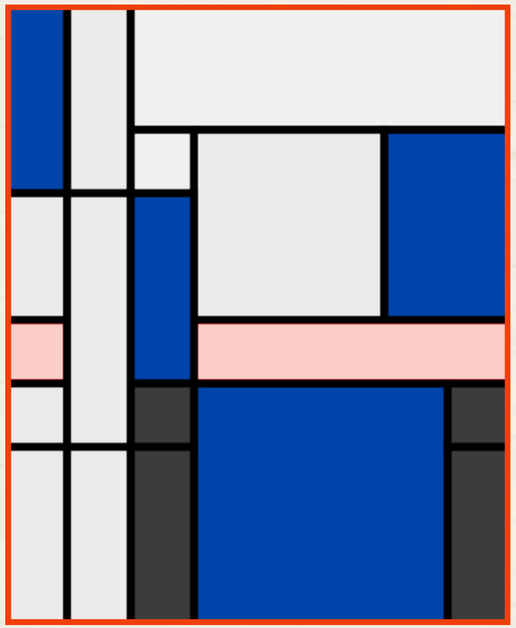
Builds on rules (e.g., recursion, randomness) to create complexity from simplicity - e.g., a single loop in Python evolves into intricate patterns.

Fractals for self-similar motifs (e.g., **Mandelbrot sets** in jewelry); Perlin Noise for organic, natural textures (e.g., **fabric simulations**); Cellular Automata (like Conway's Game of Life) for evolving, **grid-based designs**.

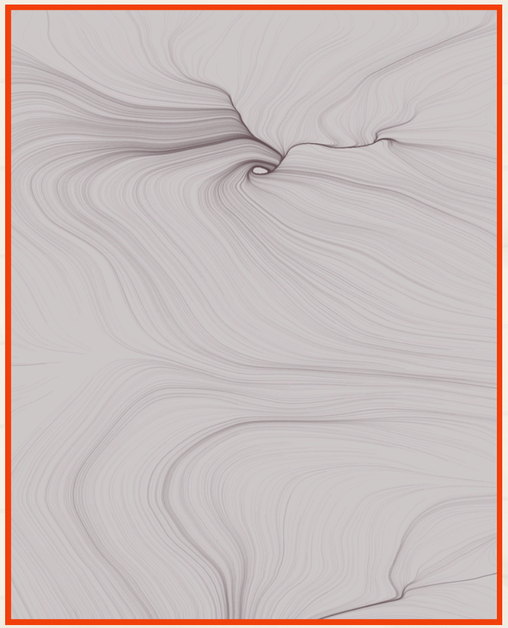
Balance between order and chaos—too simple is boring, too random is noise; aim for 'effective' outputs where algorithms yield meaningful, aesthetically pleasing results (e.g., structured randomness in reactive garments).

Libraries like Numpy for noise generation, Turtle for basic rules - scales to fashion via parametric variations for sustainability and personalization.


evolutionary systems
and a-life
Effective Complexity
fractals and
L-systems
symmetry
and tiling
chaotic systems
randomization
Complexity
Order
Compressibility
Disorder
Incompressibility



PIET EXPERIMENTS



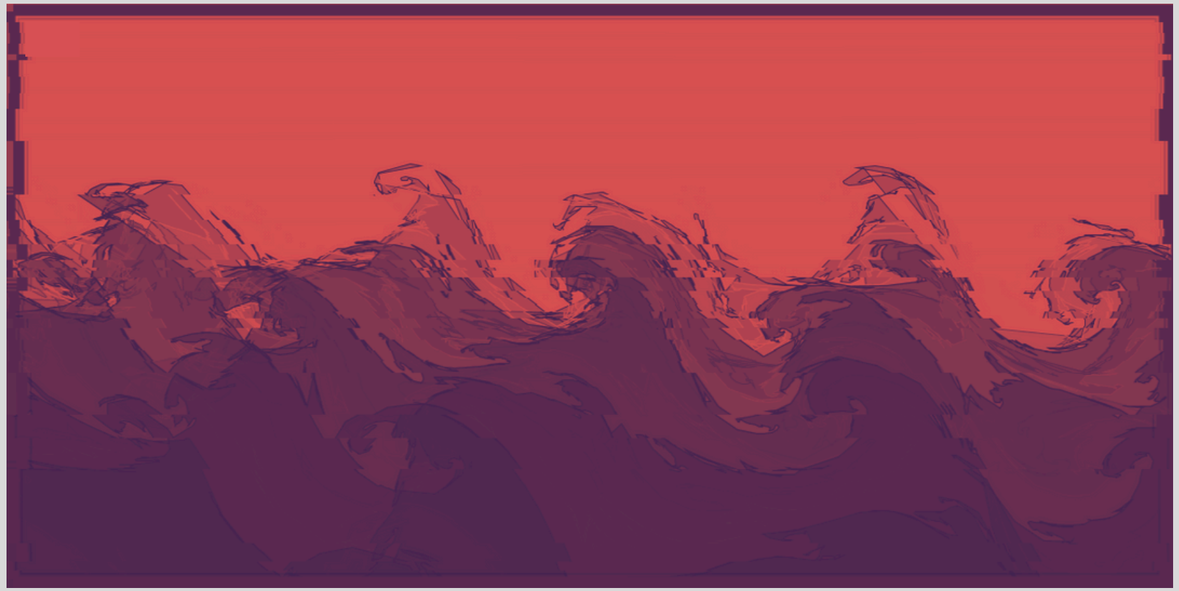
PERLIN NOISE

PIXEL SORTING

FLUIDS

FLUIDS

GENERATIVE ART IN FASHION.



GENERATIVE PATTERNS FOR UNIQUE PRINTS, E.G., *PERLIN NOISE* CREATING MARBLE-LIKE OR ORGANIC FABRICS FOR SCARVES - **REDUCES UNIFORMITY IN MASS PRODUCTION**.

GARMENTS: DATA-DRIVEN REACTIVE DESIGNS, E.G., **HEARTBEAT-MODULATED COLORS** OR CELLULAR AUTOMATA FOR EVOLVING PRINTS ON DRESSES THAT 'ADAPT' TO INPUTS.

3D-PRINTED JEWELRY VIA **FRACTALS** OR **VORONOI**, E.G., SELF-SIMILAR MOTIFS FOR ***EARRINGS/BRACELETS***, ADDING EMERGENT COMPLEXITY.

ALGORITHMS FOR EFFICIENT LAYOUTS, E.G., **NATURE-INSPIRED NESTING** TO **MINIMIZE WASTE** IN BAG/GLOVE PATTERNS - ALIGNS WITH 2025 ECO-TRENDS.

```python
def generate_japanese_wave(X, Y, wave_size, T, frame_counter):
    time_falloff = 1 - T / 5  # Decay over time
    wave_size = 5 / time_falloff * wave_intensity
    # Iterative wave deformation.
    while wave_size < W / 5 / time_falloff:
        A = ((X / wave_size + T + frame_counter * time_speed) % 1) * wave_size - wave_size / 2

        deform = 1 - ((A * A) / (wave_size * wave_size)) * 4

        japanese_modulation = py5.sin(T * 2 + layer_index * 0.5) * 0.1
        deform += japanese_modulation

        C = py5.cos(deform)
        S = py5.sin(deform)

        new_X = X + A * C + Y * S - A - 20
        new_Y = Y * C - A * S

        X, Y = new_X, new_Y

        wave_size /= 0.62

    return X, Y
```

Time-based wave parameter calculation.

Calculate wave position with time animation.

Deformation which creates wave curvature.

Add Japanese aesthetic modulation.

Update position using rotation matrix.

Scale wave for next iteration (fractal-like behavior).

PYCON JAPAN 2025

26 SEPTEMBER 2025

PYCON JAPAN 2025

ALGORITHMIC KNITWEAR ZERO WASTE MANUFACTURING

A GLIMPSE INTO THE VISUAL IDENTITY AND AESTHETIC DIRECTION

TRADITIONAL WASTE

ENVIRONMENTAL IMPACT

ECONOMIC LOSS

CONSUMER AWARENESS

TRADITIONAL FASHION WASTES FABRIC 15% OF THE MATERIAL THROWN AWAY.

ZERO-WASTE INNOVATION

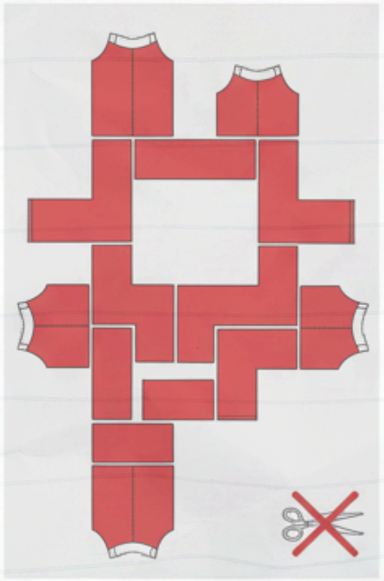
3D PRINTING

KNITTING

OPTIMIZATION

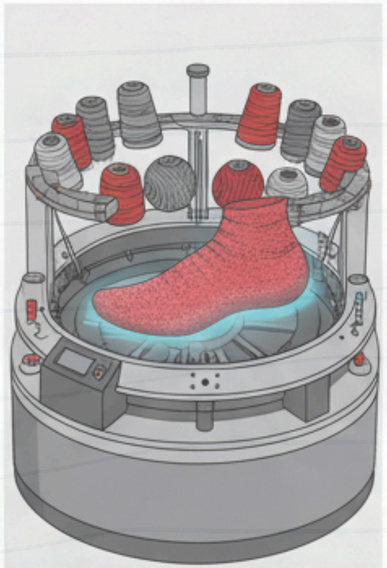
ZERO WASTE FASHION
DESIGN + TECHNOLOGY

THE SOLUTION: ZERO WASTE MANUFACTURING



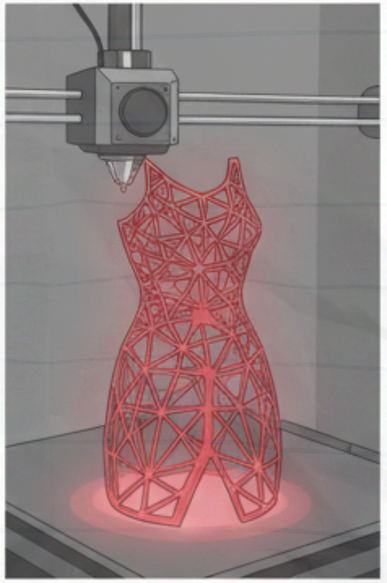
Zero Waste Pattern Cutting

Garment pieces interlock.
No fabric wasted.



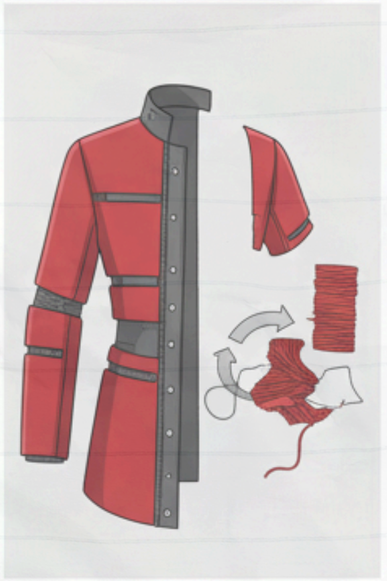
Knitting-to-Shape

Seamless 3D knitting.
Example: Nike Flylnkt



3D Printing

Layer-3-layer creation.
Example: Danit Peleg



Modular & Recycable Design

Components are reusable.
Example: ZEROBARRACENTO

```python
from knitout import Writer

# create knitout writer for a flat-bed machine
k = Writer("3 4")   # 2 beds, 4 needles each

k.comment("Cast on and knit a small swatch")
k.inhook("B")       # bring yarn carrier in
for n in range(1, 5):
    k.knit("+", "B" + str(n))   # knit forward
for n in range(4, 0, -1):
    k.knit("-", "B" + str(n))   # knit
k.releasehook("B")  # release yarn
k.out("swatch.k")

print("Generated knitout file: swatch.k")
```

This creates a new knitting program for a machine that has:

- 2 beds (front and back needle beds, like two rows of needles).
- 4 needles on each bed (numbered B1, B2, B3, B4, etc.).

knits forward and backward (like a zigzag), and saves as a .k knitout file.

PYCON JAPAN 2025

ALGORITHMIC TEXTILES & SMART FABRICS

A GLIMPSE INTO THE VISUAL IDENTITY AND AESTHETIC DIRECTION

THE PROBLEM

Old Ways / Static Fabrics

- Static, One-Size-Fits-All
- No Adapation to Heat or Movement
- Comfort Relies Solely on Design

Traditional fabrics are unmourtable and non-responsive

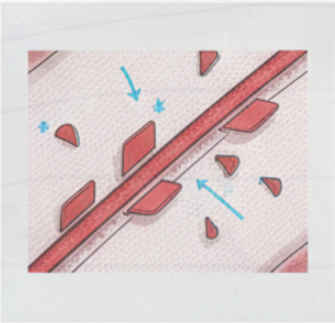
THE INVENTION

Smart Fabrics / Adaptive Technology

- AI + Sensors + Materials Science
- Fabrics that Adapt & Respond
- Optimal Comfort, Performance & Sustainabilty

Chromat x Intel

Adidas Futurecraft 4D

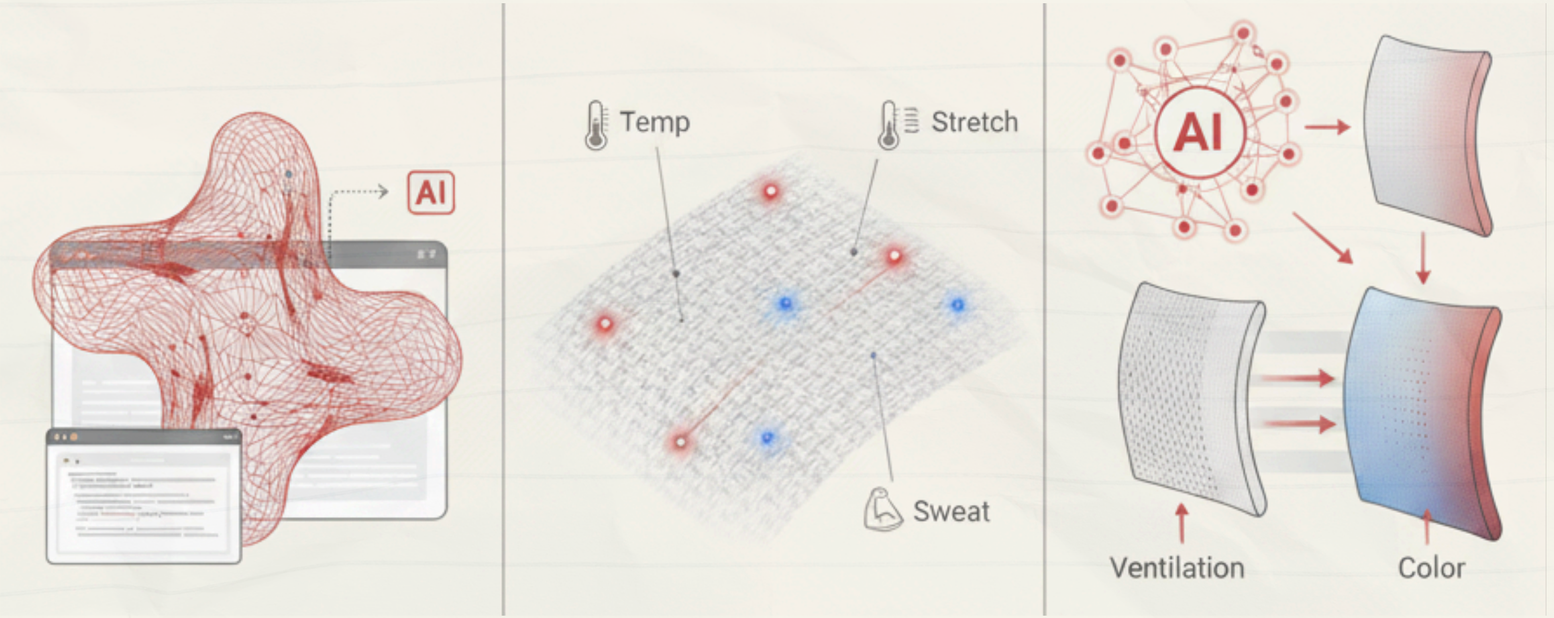


MIT biologic
No fabric wasted.

Modular & Recycable
Example: Danit Flyleg

DYNAMIC FITS
ADAPTION TO HEAT, PULSES ETC

HOW IT'S ACHIEVED



GENERATIVE DESIGN

EMBEDDED SENSORS

AI OPTIMISATIONS

```python
import numpy as np
import matplotlib.pyplot as plt

# simulate temperature range
temperature = np.linspace(20, 40, 100)  # °C

# fabric property: breathability decreases as temp rises, elasticity increases
breathability = np.exp(-(temperature - 20) / 10)
elasticity = 1 + 0.05 * (temperature - 20)

plt.plot(temperature, breathability, label="Breathability ↓")
plt.plot(temperature, elasticity, label="Elasticity ↑")
plt.xlabel("Temperature (°C)")
plt.ylabel("Fabric Property")
plt.title("Adaptive Fabric Response to Temperature")
plt.legend()
plt.show()
```

- At **low temperature** → fabric stays breathable, less elastic.
- At **high temperature** → fabric becomes less breathable but stretches more

Plotting and visualizing the response

PYCON JAPAN 2025

GENERATIVE AI + HERITAGE/ ARCHIVES

A GLIMPSE INTO THE VISUAL IDENTITY AND AESTHETIC DIRECTION

AI + HERITAGE: REMIXING HERITAGE

AI Input /
Training Data

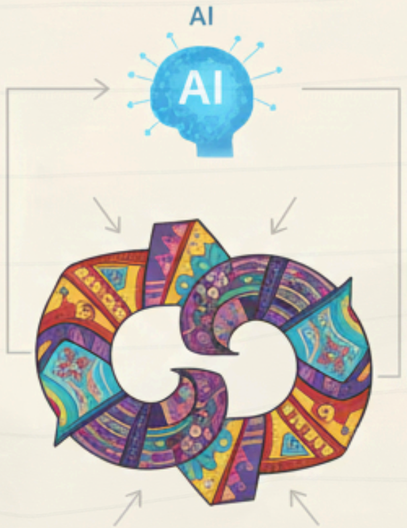
Generative AI

MAISON
VALENTINO

HOW IT'S ACHIEVED

1.
Training Datsets

2.
GANs / Diffusion Models



remix_motifs(traditional + modern)

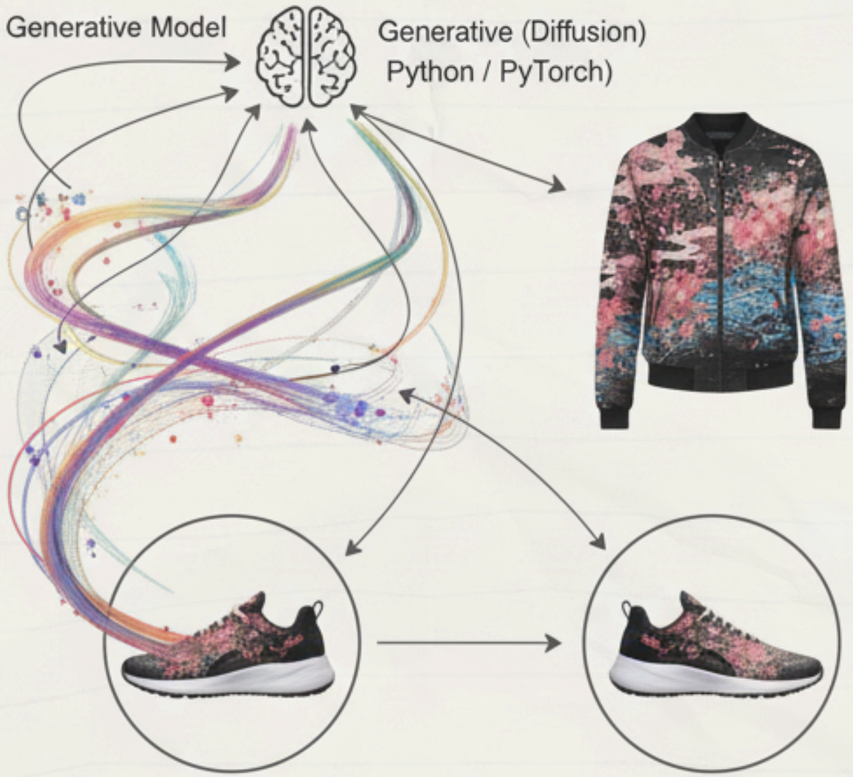
3.
Designer Refiement

TRAINING DATA SETS
DIFFUSION MODELS
DESIGNER REFINEMENT

Traditional Heritage

AI Input / Training Data

Generative AI + Modern Streetwear



Outcome: Blending Culture & Future

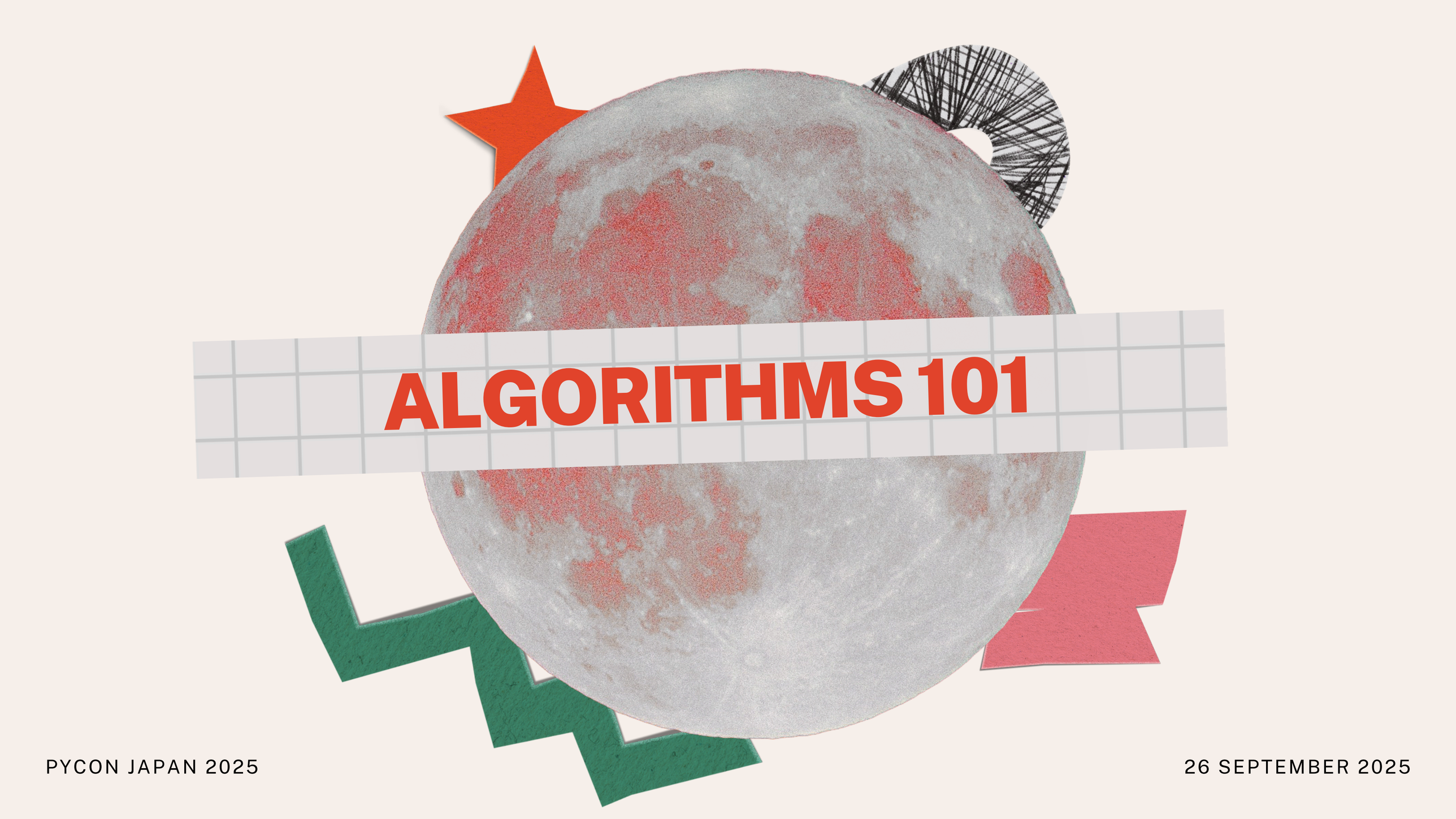
KIMONO MOTIFS WITH GENERATIVE AI

```python
from diffusers import StableDiffusionPipeline, DDPMScheduler
from transformers import AutoTokenizer
import torch

# load base diffusion model
model_id = "runwayml/stable-diffusion-v1-5"
pipe = StableDiffusionPipeline.from_pretrained(model_id, torch_dtype=torch.float16).to("cuda")
tokenizer = AutoTokenizer.from_pretrained(model_id)

# training loop (simplified)
for batch in train_dataset:
    images, captions = batch["image"], batch["caption"]
    latents = pipe.vae.encode(images).latent_dist.sample()
    noise = torch.randn_like(latents)
    timesteps = torch.randint(0, pipe.scheduler.config.num_train_timesteps, (latents.shape[0],))
    noisy_latents = pipe.scheduler.add_noise(latents, noise, timesteps)
    text_embeddings = pipe.text_encoder(tokenizer(captions, return_tensors="pt").input_ids.to("cuda"))
[0]
    noise_pred = pipe.unet(noisy_latents, timesteps, text_embeddings).sample
    loss = torch.nn.functional.mse_loss(noise_pred, noise)
    loss.backward()
    optimizer.step()

# ✅ use the fine-tuned model
prompt = "Modern streetwear jacket inspired by traditional Japanese kimono patterns"
image = pipe(prompt, guidance_scale=7.5).images[0]
image.save("kimono_streetwear.png")
```

Loading **pre-trained diffuser model for fine tuning**

Fine tuning diffuser model on kimonos dataset.

Final use of model



PYCON JAPAN 2025

26 SEPTEMBER 2025

THE CODE BEHIND CREATIVE FASHION

TRADITIONAL ALGOS

MATH-BASED FOR PATTERNS AND OPTIMIZATION- E.G., **WAVES**, **NOISE,** FRACTALS FOR TEXTILES.

AI BASED ALGORITHMS

MACHINE LEARNING FOR GENERATIVE AND ADAPTIVE DESIGNS - E.G., GANS, **DIFFUSION** FOR OUTFITS.

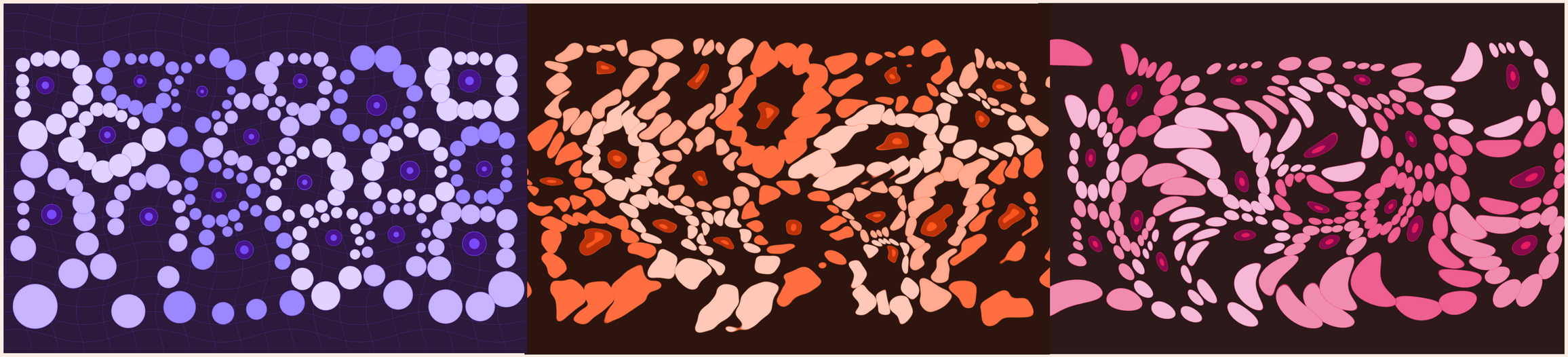
DRIVES 2025 TRENDS IN PERSONALIZATION, SUSTAINABILITY, AND INNOVATION FOR FABRICS/OUTFITS.

SINE WAVES

USES TRIGONOMETRIC FUNCTIONS TO CREATE REPEATING, **FLOWING PATTERNS** - IDEAL FOR WAVY TEXTILES OR GARMENT CONTOURS IN OUTFIT DESIGN.

GENERATE RIPPLE EFFECTS FOR DRESS HEMS OR FABRIC PRINTS; **PARAMETRIC** FOR VARIATIONS IN AMPLITUDE/FREQUENCY.

```
wave_value = py5.sin(x * 0.1)  # x is position, 0.1 controls frequency
y_offset = py5.sin(x * 0.05 + time) * 50 # Creates flowing fabric waves
```



```python
def draw_classic_petal():
    for radius in petal_range:

        edge_wave = sin(radius * frequency + theta * 3 + time) * amplitude * 8
        size_wave = sin(radius * 0.1 + time * 0.5) * 0.15 + 1.0


        x = center_x + (radius + edge_wave) * cos(theta) * size_wave
        y = center_y + (radius + edge_wave) * sin(theta) * size_wave


        petal_size += sin(radius * 0.25 + theta * 2 + time) * 3
```

Sine wave creates organic, natural petal edges.

Apply sine modulation to position.

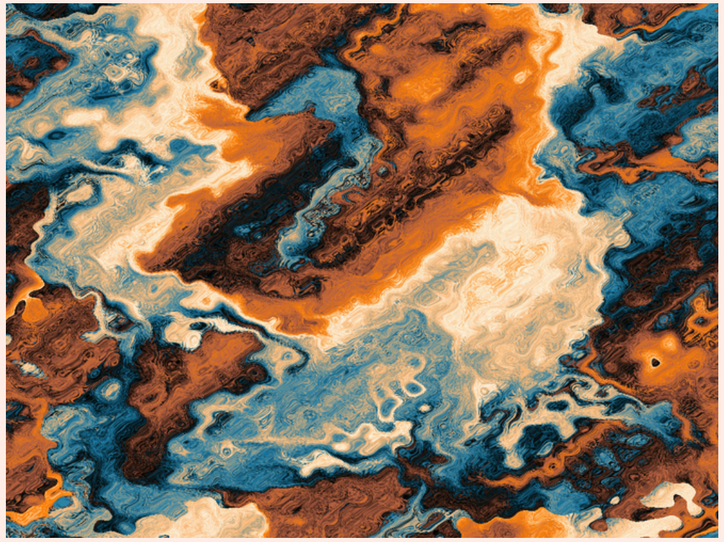
Sine-modulated petal size for organic texture.

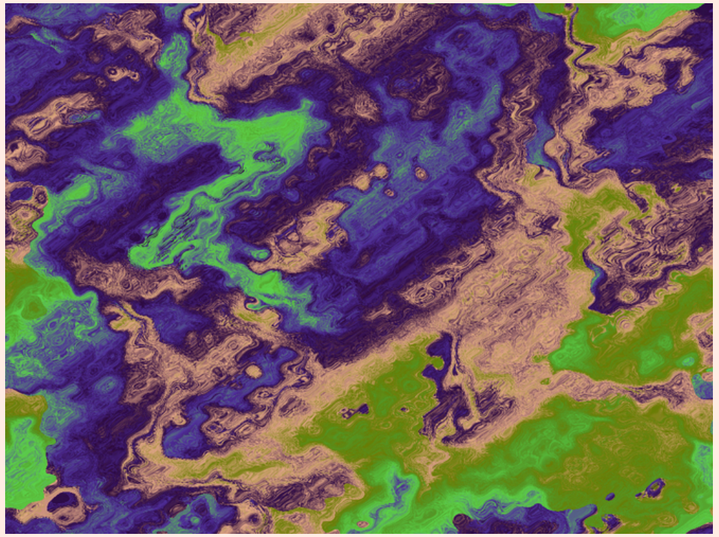
PERLIN NOISE

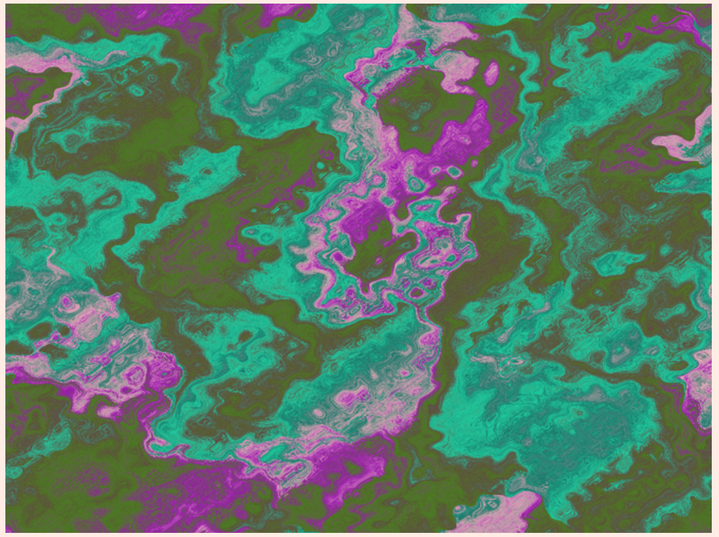
GRADIENT-BASED RANDOMNESS FOR SMOOTH, ORGANIC TEXTURES - ***MIMICS*** NATURAL VARIATIONS IN FABRIC DESIGN.

FASHION APPLICATION: CREATE MARBLE-LIKE OR **CLOUD PATTERNS** FOR SUSTAINABLE PRINTS; USED IN SCARF OR DRESS TEXTILES TO ADD DEPTH.

```
texture_value = py5.noise(x * 0.01, y * 0.01)  # Returns 0-1, smooth transitions
py5.noise_detail(4, 0.5); fabric_pattern = py5.noise(x * 0.02, y * 0.02, time * 0.1)
```







```python
def create_japanese_textile_pixel(x, y, time):
    base_texture = py5.noise(x * 0.5, y * 0.5, time + 50) * 60      # Large, smooth patterns
    medium_detail = py5.noise(x * 5, y * 5, time) * 60              # Medium frequency details
    fine_detail = py5.noise(x * 5, y * 5, time + 20) * 60           # Fine surface texture

    wave_pattern = 360 * (medium_detail / 1.5 - math.floor(medium_detail / 1.5))
    organic_flow = math.cos(math.radians(1.5 * wave_pattern)) * 20

    if mode == "flowing":
        hue_base = py5.remap(0.3 * fine_detail + 0.9 * base_texture, 0, 60, -60, 420)
    elif mode == "geometric":
        hue_base = py5.remap(medium_detail + fine_detail * 0.5, 0, 60, 0, 360)
    else:  # traditional
        hue_base = py5.remap(base_texture + medium_detail * 0.7, 0, 60, 0, 360)

    final_hue = (360 + 3 * hue_base + organic_flow) % 360

    surface_noise = py5.noise(x * 0.01, y * 0.01, time * 0.1)
    brightness_modifier = 0.8 + 0.4 * surface_noise

    return final_hue, brightness_modifier
```

Multi-octave noise sampling at different frequencies

Create organic wave motion (Japanese aesthetic)

Blend noise layers for different textile styles.

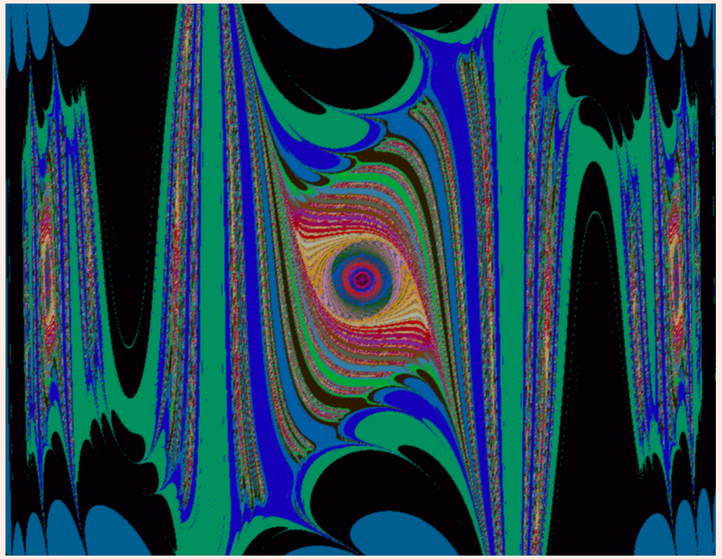
Add surface texture variation.

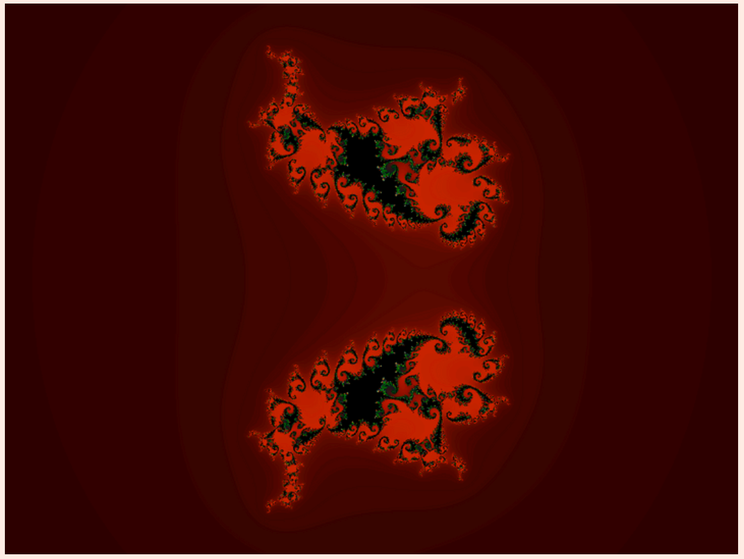
FRACTALS

SELF-SIMILAR PATTERNS VIA **RECURSION** - BUILDS COMPLEXITY FROM REPETITION FOR DETAILED TEXTILES.

FASHION APPLICATION: **RECURSIVE MOTIFS** IN GLOVE GRIPS OR BAG DESIGNS; ENHANCES UNIQUENESS IN SUSTAINABLE FASHION.

```
z = complex(x, y); iterations = 0
while abs(z) < 2 and iterations < 50: z = z*z + complex(x, y); iterations += 1
Map escape time to colors - creates the beautiful fractal patterns
color_value = py5.map(iterations, 0, 50, 0, 255); py5.fill(color_value, 100, 255)
```





```python
def julia_fractal_generator(x, y, a=1, b=0.2):
    """Main function showing Julia set fractal generation"""

    # Iterate through fractal calculation (max 100 iterations)
    for iter in range(100):
        # Apply complex mathematical transformations
        u = np.sin(x) * np.cosh(y)      # u_mul1(x) * u_mul2(y)
        v = np.cos(x) * np.sinh(y)      # v_mul1(x) * v_mul2(y)

        # Julia set transformation formula
        new_x = a * u - b * v * 4       # Real part transformation
        new_y = b * u + a * v / 1.4     # Imaginary part transformation

        x, y = new_x, new_y

        # Escape condition - if point diverges beyond threshold
        if abs(y) > 50:
            return iter  # Return iteration count for coloring

    return 0  # Point is in the Julia set (never escaped)
```

A geometrical figure where each part has the same statistical characters.

$$z = x + yi$$

$$z_{n+1} = z_n^2 + C$$

*z_{n+1} = f(z_n) where f(z) = a(sin(x)cosh(y) + i*cos(x)sinh(y)) - b(cos(x)sinh(y) + i*sin(x)cosh(y))*4*

FABRICATION PATHS

WHAT ARE FABRICATION PATHS?

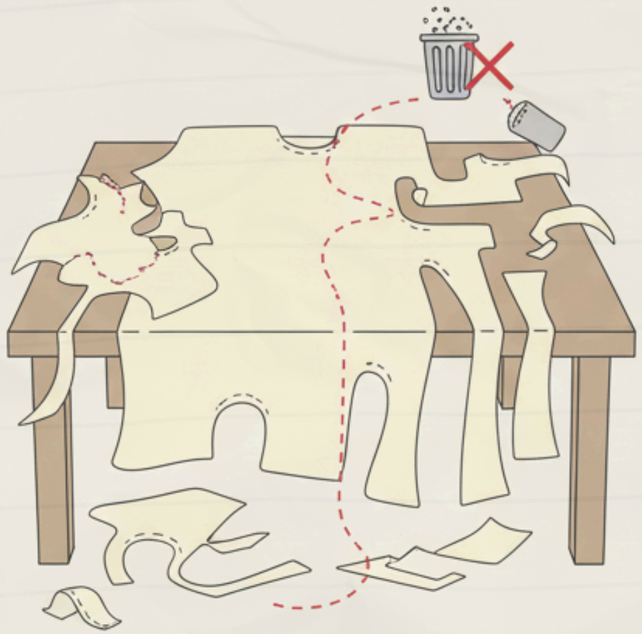
DIGITAL DESIGN BECOMES PHYSICAL OBJECT

DEFINE STEPS, METHODS AND TOOLS

2D SURFACES & 3D FORMATS

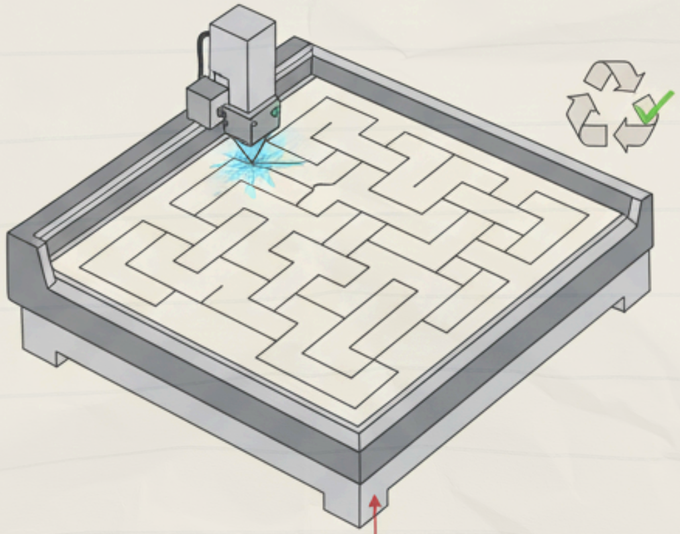
ENSURES DESIGNS ARE PRACTICAL

Traditional Method



- Ensures feasibility
- Optimizes material (less waste)
- Enables customization & complexity

Digital Fabiration

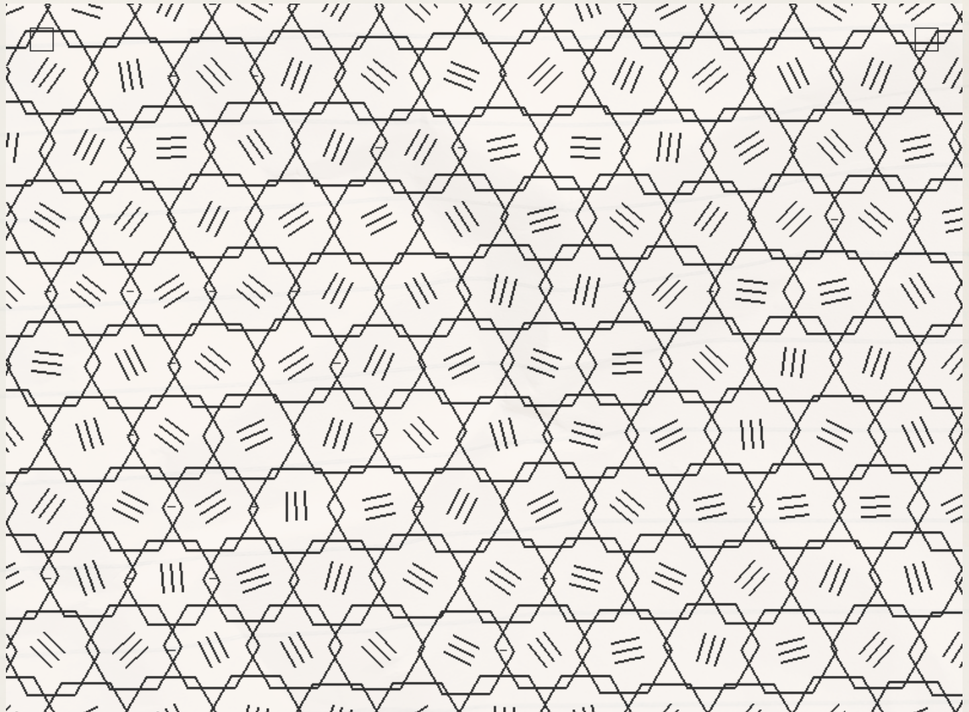


- Material Saved
- Time Saved
- Precision Gain

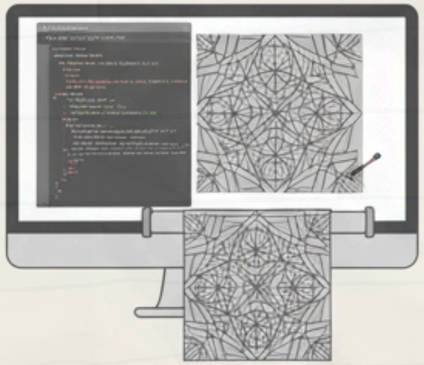
WHY THEY MATTER?

EFFICIENCY, INNOVATION, SUSTAINABILITY IN DESIGN & PRODUCTION

TECH & PYTHON IN FABRICATION PATHS



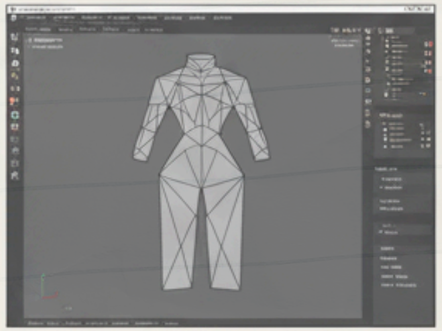
2D Fabiration Path



Pattern Generation → Embroidery Optimization

- Python enables algrriratinc control control
- 2D: pattern generation, motif placement
- 3D: parnmetic modeling, 3D printing
- Simulation, testing & iteration

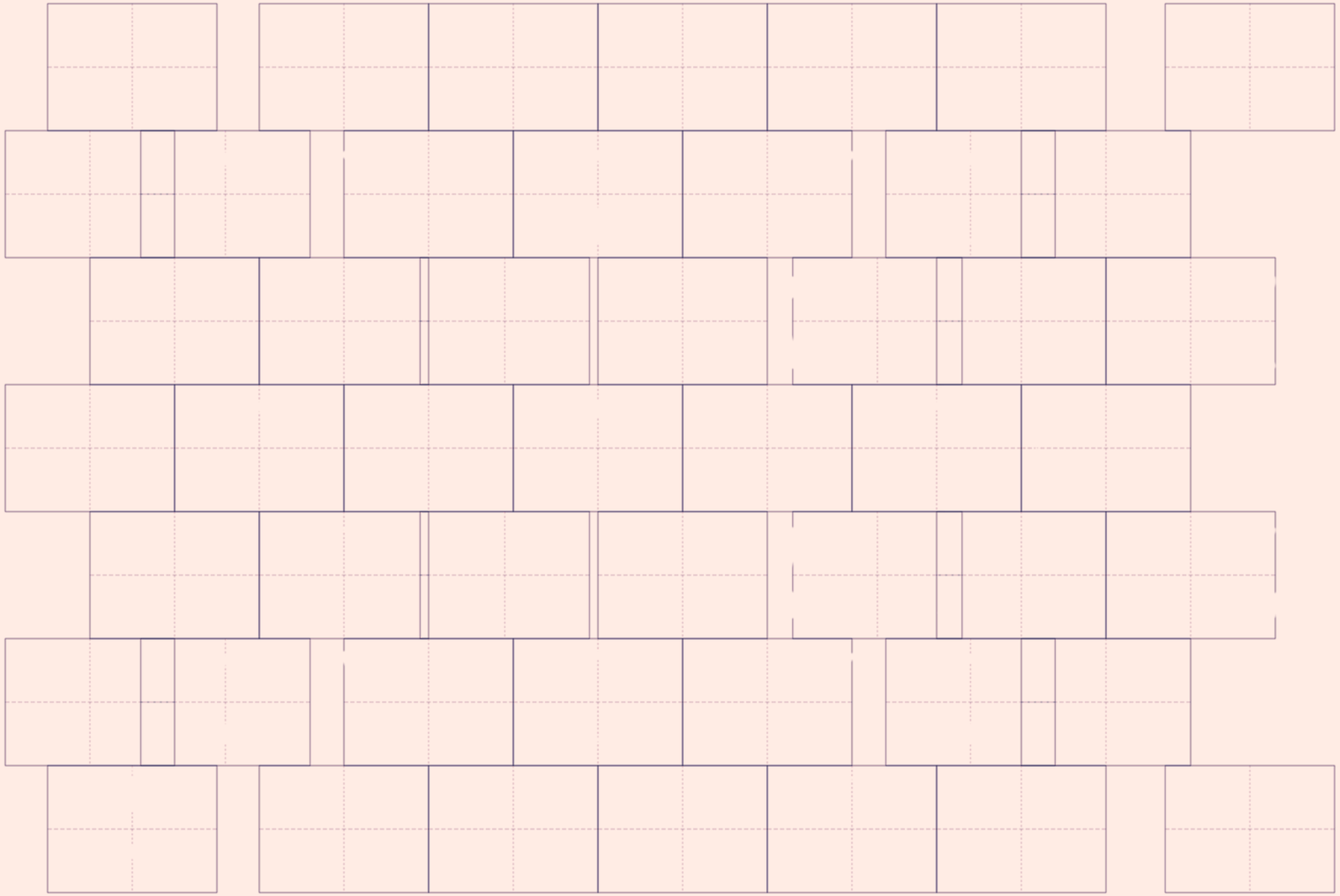
3D Fabiration Path



Parmeretic Modeling → 3D Knitting/Printing

Origami inspired Fabric Laser Cut (Miura fold tessellation).

PYCON JAPAN 2025



```python
def generate_miura_fold(fold_width=40, fold_height=30, complexity=3):
    """Simple Miura fold pattern generator"""
    pattern_data = []

    # Create grid of fold points
    for i in range(-complexity, complexity + 1):
        for j in range(-complexity, complexity + 1):
            # Calculate position with offset pattern
            x = i * fold_width + (j % 2) * fold_width/2
            y = j * fold_height

            # Generate three types of fold lines:

            # 1. Mountain folds (blue) - horizontal lines
            mountain_fold = {
                'type': 'mountain_fold',
                'coords': [(x - fold_width/2, y), (x + fold_width/2, y)]
            }
            pattern_data.append(mountain_fold)

            # 2. Valley folds (green) - vertical lines
            valley_fold = {
                'type': 'valley_fold',
                'coords': [(x, y - fold_height/2), (x, y + fold_height/2)]
            }
            pattern_data.append(valley_fold)

            # 3. Cut lines (red) - rectangle outline
            cut_line = {
                'type': 'cut_line',
                'coords': [(x - fold_width/2, y - fold_height/2),
                           (x + fold_width/2, y - fold_height/2),
                           (x + fold_width/2, y + fold_height/2),
                           (x - fold_width/2, y + fold_height/2)]
            }
            pattern_data.append(cut_line)

    return pattern_data
```

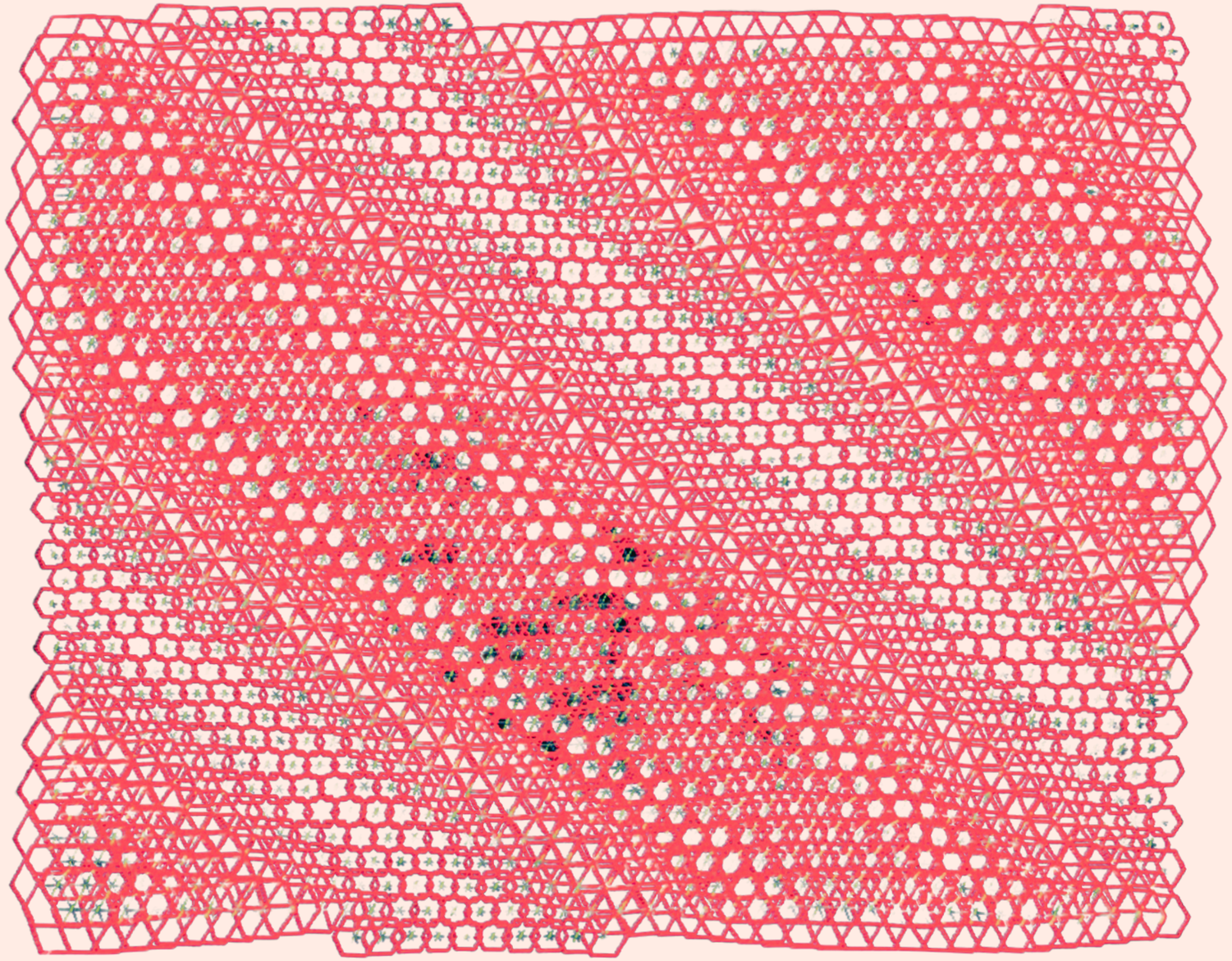
*Calculate position with offset pattern - creates characteristic **zigzag layout**.*

*Mountain fold: horizontal crease that **folds upward**.*

*Valley fold: vertical crease that **folds downward**.*

*Cut line: rectangular boundary for **laser cutting** (red in visualization).*

PYCON JAPAN 2025

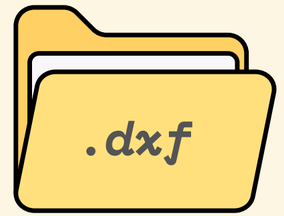


```python
def export_to_dxf(pattern_data, filename="origami.dxf"):
    """
    Export 2D origami pattern to DXF file format
    Core algorithm: Convert fold lines to DXF entities with different layers
    """

    # Step 1: Create DXF file header
    dxf_content = [
        "0", "SECTION", "2", "ENTITIES"  # Start DXF entities section
    ]

    # Step 2: Convert each fold line to DXF line entity
    for line in pattern_data:
        x1, y1, x2, y2, fold_type = line

        layer_name = fold_type.upper()  # "MOUNTAIN_FOLD", "VALLEY_FOLD", "CUT_LINE"

        # Create DXF line entity
        dxf_content.extend([
            "0", "LINE",           # Entity type: LINE
            "8", layer_name,       # Layer name (fold type)
            "10", str(x1),         # Start X coordinate
            "20", str(y1),         # Start Y coordinate
            "11", str(x2),         # End X coordinate
            "21", str(y2)          # End Y coordinate
        ])

    # Step 3: Close DXF file
    dxf_content.extend(["0", "ENDSEC", "0", "EOF"])

    # Step 4: Write to file
    with open(filename, 'w') as f:
        f.write('\n'.join(dxf_content))

    return f"DXF exported as {filename}"
```

pattern_data *: This is the list of 2D lines we generated earlier, where each line has its start (x1, y1), end (x2, y2) coordinates, and its fold_type (mountain, valley, or cut).*

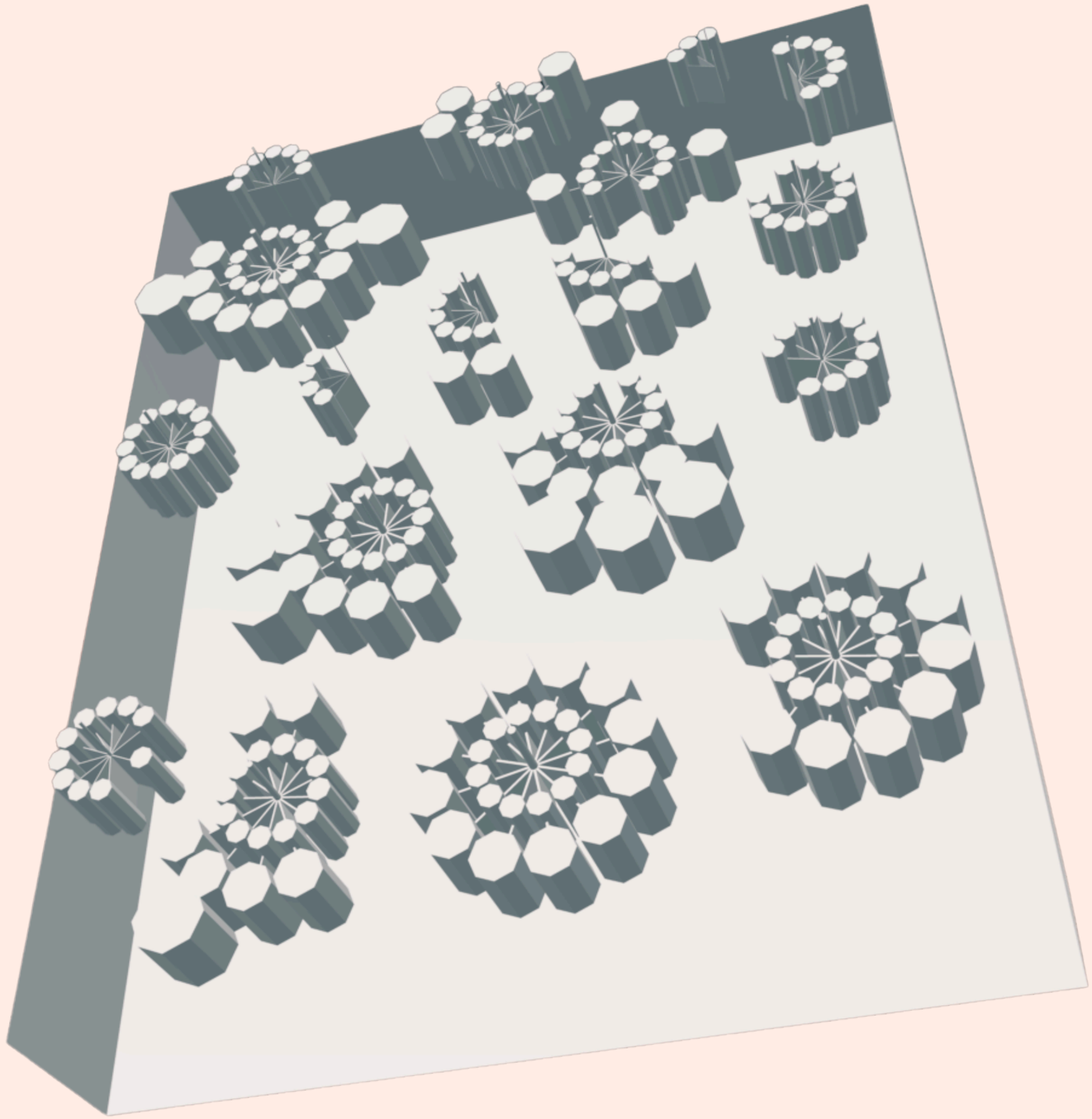
For each line , we unpack its components: the start X and Y, end X and Y, and the fold_type

- We append a series of group codes and values to our dxf_content list to define a single line in DXF.



Voronoi inspired 3D Earrings.

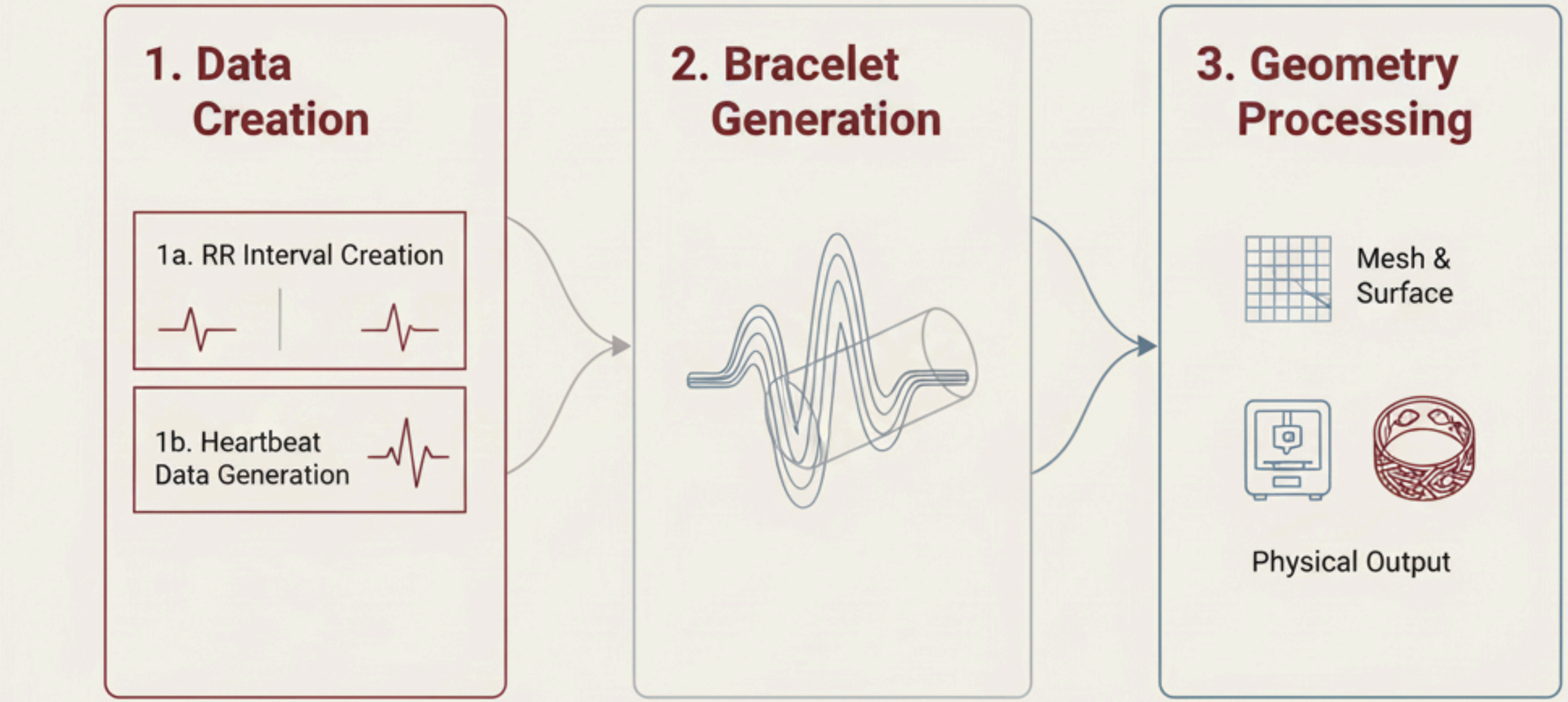
PYCON JAPAN 2025



PYCON JAPAN 2025

HEARTBEAT TO 3D BRACELET GENERATOR

HEARTBEAT TO BRACELET: HIGH LEVEL ALGORITHM



Phase-1 HeartBeat Data Generation

```python
def generate_rr_intervals(self, duration, heart_rate, stress_level, activity_level, emotion):
    """Generate R-R intervals based on physiological parameters"""
    # Base interval from heart rate
    base_rr = 60.0 / heart_rate

    # Emotion effects on heart rate variability
    emotion_effects = {
        'calm': {'hrv_factor': 1.2, 'irregularity': 0.05},
        'excited': {'hrv_factor': 0.8, 'irregularity': 0.15},
        'anxious': {'hrv_factor': 0.6, 'irregularity': 0.25},
        'relaxed': {'hrv_factor': 1.5, 'irregularity': 0.03}
    }

    effect = emotion_effects.get(emotion, emotion_effects['calm'])

    # Generate intervals with physiological variation
    num_beats = int(duration / base_rr)
    rr_intervals = []

    for i in range(num_beats):
        # Stress increases heart rate, reduces variability
        stress_factor = 1.0 - (stress_level * 0.3)
        activity_factor = 1.0 + (activity_level * 0.5)

        # Add natural variability
        variability = np.random.normal(0, effect['irregularity'])

        interval = base_rr * stress_factor * effect['hrv_factor'] * (1 + variability)
        rr_intervals.append(max(0.3, interval))  # Physiological limits

    return np.array(rr_intervals)
```

Function to generate RR intervals for ECG Grpah

We take the base heart rate (beats per second) and adjust it with emotional effects over the chosen duration. Using these inputs, we generate RR intervals in a loop that reflects all influencing factors.

This gives us a sample RR interval like : [0.424, 0.398, 0.456, 0.401, 0.445, 0.389, 0.467, 0.412, ...]. Based on the input parameters the RR interval varies

```python
def generate_heartbeat_signal(self, rr_intervals, sampling_rate=250):
    """Convert R-R intervals to ECG-like signal"""
    total_duration = np.sum(rr_intervals)
    t = np.linspace(0, total_duration, int(total_duration * sampling_rate))
    signal = np.zeros_like(t)

    current_time = 0
    for rr in rr_intervals:
        # Generate QRS complex (main heartbeat spike)
        qrs_center = current_time + rr * 0.3
        qrs_indices = np.where((t >= qrs_center - 0.05) & (t <= qrs_center + 0.05))[0]

        if len(qrs_indices) > 0:
            # Create realistic QRS shape
            qrs_t = t[qrs_indices] - qrs_center
            qrs_amplitude = np.exp(-50 * qrs_t**2) * (1 + 0.5 * np.sin(100 * np.pi * qrs_t))
            signal[qrs_indices] += qrs_amplitude

        current_time += rr

    return t, signal
```

This calculates the total duration from RR intervals and creates evenly spaced timestamps on the X-axis. It then initializes a zero-filled signal array of the same length for further processing.

This code finds the time indices where each heartbeat spike should occur and centers them around zero. At those positions, it generates a bell-curve-like QRS waveform with added amplitude variations to mimic a realistic heartbeat.

This gives returns 1D array of timestamps and signal

Phase-2 Braclet generation

```
t =      [0.0, 0.004, 0.008, ..., 1.24, 1.244, 1.248, ..., 3.0]  (TIME)
signal = [0.0,  0.0,  0.0, ..., 2.1,  2.8,   2.3, ..., 0.0]     (HEARTBEAT STRENGTH)
```

We have 1D data of **time stamps** and corresponding **heartbeats** signals.

```
resampled_signal = self.resample_heartbeat_data(heartbeat_data, num_points=100)
```

*We **downsample** from X original to 100 points, since the bracelet design uses 100 mapped data points. Because we want our bracelet to have 100 segments around the circle - **not too chunky, not too smooth.***

```
angles = np.linspace(0, 2*np.pi, len(resampled_signal), endpoint=False)
        vertices = []
        faces = []
```

This key step is bending the 1-D line into a circle using **100 evenly spaced angles.**

The function returns the bracelet's 3D mesh as vertices (3D points in space) and faces (triangles connecting those points) that together define its full geometry.

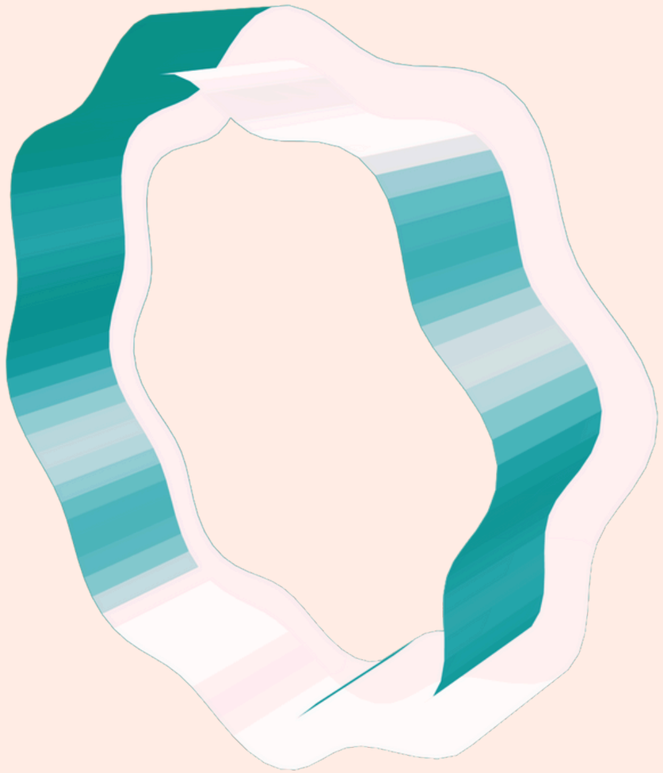
```
        for i, (angle, amplitude) in enumerate(zip(angles, resampled_signal)):
            # Convert heartbeat amplitude to radial variation
            radial_offset = amplitude * height_variation * pattern_intensity
```

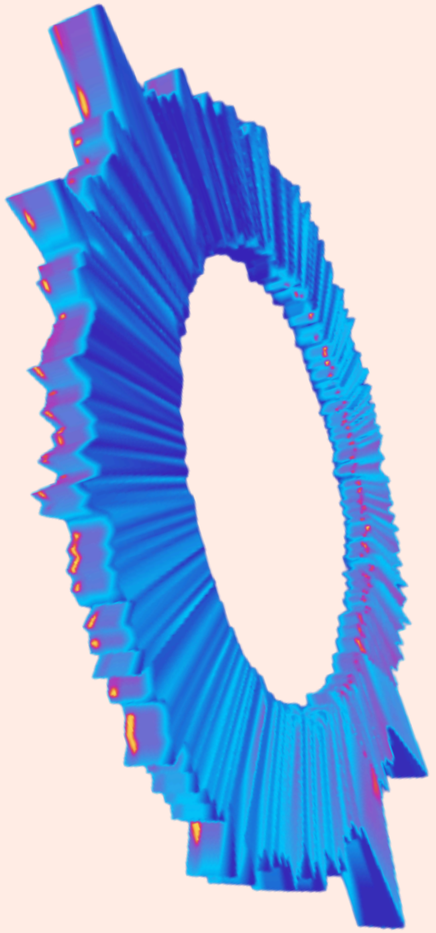
Once we have bent the 1-d line into **circle…** we use heartbeat data to determine **thickness.**

```
Point:  1    2    3    4    5   ...  98   99   100
Data:  [0.0, 0.1, 2.1, 2.8, 1.2, ..., 1.5, 0.8, 0.0]
Time:   0s  0.03s 0.06s 0.09s ...        ...   3s

Normal thickness:     ====  (base bracelet)
Strong heartbeat:   ========  (bulges out)
Weak heartbeat:       ==      (stays normal or thinner)
```

Heartbeat determines the points of thickness


Heart Rate (BPM)
70
Stress Level
0.5
Activity Level
0.3
Emotion
Anxious
Duration (seconds)
25





PYCON JAPAN 2025

3D PRINTABLE CAP/HAT GENERATOR

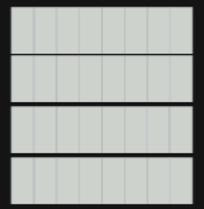
Input Parameters

```
cap_style:
"baseball"

pattern_type:
"mesh"

packing_type:
"medium_packed"

head_circumf:
58
```

→

Step 1

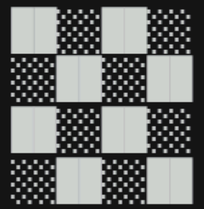
Solid Cap



→

Step 2

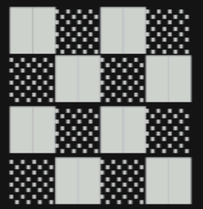
Patterned Cap



→

Step 3

Comfortable Cap



+ grooves
+ vents

→

Output

JSON Geometry

```
vertices:
[x,y,z...]

faces:
[0,1,2...]
```

```python
@app.route('/generate_cap', methods=['POST'])
def generate_cap():
    try:
        data = request.json
        cap_style = data.get('cap_style', 'baseball')        # Line 722: Get user input
        pattern_type = data.get('pattern_type', 'mesh')      # Line 723: Get user input
        packing_type = data.get('packing_type', 'medium_packed') # Line 724: Get user input
        head_circumference = float(data.get('head_circumference', 58)) # Line 725: Get user input
        material = data.get('material', 'PLA')               # Line 726: Get user input

        designer = ModernCapDesigner()                       # Line 728: Create designer instance

        # 🚀 HERE IT IS! LINE 730 - THE MAGIC CALL!
        geometry = designer.create_cap_geometry(cap_style, pattern_type, packing_type,
head_circumference)

        comfort_metrics = designer.calculate_comfort_metrics(geometry, material, packing_type) # Line
731
```

*Main function where we setup **core parameters** like cap style, pattern types, packing types etc and then using this we call the main **geometry function** to create 3D CAP*

```python
# Step 1: Create base cap geometry
vertices, faces = self.create_cap_base_geometry(cap_style, head_circumference)

# Step 2: Apply modern pattern
vertices, faces = self.apply_modern_pattern(vertices, faces, pattern_type, packing_type)

# Step 3: Add comfort features
vertices, faces = self.add_comfort_features(vertices, faces, cap_style)
```

*The create_cap_geometry function transforms user parameters into a complete 3D cap through three sequential steps: Step 1 generates the **solid cap** shape using trigonometric calculations to create 3D vertices and faces based on cap style and head circumference; Step 2 **applies patterns by cutting strategic holes and mesh designs** into the solid geometry for breathability and aesthetics; Step 3 **adds comfort features** like sweatband grooves and ventilation holes, then converts the final geometry to JSON format for frontend visualization and 3D printing*

```python
for i in range(theta_steps + 1):
    t = i / theta_steps  # Normalized height parameter [0,1]

    # 📊 MATHEMATICAL SHAPE CALCULATIONS
    if t < 0.7:  # Main crown mathematical profile
        height_factor = math.sin(t * math.pi * 0.7) * 0.8
        radius_factor = 1.0 - (t * 0.2)
    else:  # Brim transition mathematics
        height_factor = 0.1 + (1 - t) * 0.5
        radius_factor = 1.0

    # 🔄 APPLY SCALING TO GEOMETRY
    current_radius = head_radius * radius_factor
    current_height = crown_height * height_factor

    # 🌐 TRIGONOMETRIC 3D COORDINATE GENERATION
    for j in range(phi_steps):
        phi = (j / phi_steps) * 2 * math.pi  # Angle around circumference

        # 📍 FINAL 3D VERTEX CALCULATIONS
        x = current_radius * math.cos(phi)   # X-coordinate
        y = current_radius * math.sin(phi)   # Y-coordinate
        z = current_height                   # Z-coordinate (height)

        vertices.append([x, y, z])  # Store 3D point
```

```python
from flask import Flask, render_template, request, jsonify, send_file
import numpy as np
import math
import json
import trimesh
from scipy.spatial.distance import cdist
from scipy.optimize import minimize

app = Flask(__name__)
app.config['SECRET_KEY'] = 'modern-cap-designer-key'

class ModernCapDesigner:
    def __init__(self):
        # Material properties for 3D printing
        self.materials = {
            'PLA': {'density': 1.24, 'flexibility': 0.3, 'breathability': 0.4},
            'PETG': {'density': 1.27, 'flexibility': 0.6, 'breathability': 0.5},
            'TPU': {'density': 1.20, 'flexibility': 0.9, 'breathability': 0.8}
        }

        # Pattern packing configurations
        self.packing_types = {
            'close_packed': {'spacing': 0.8, 'hole_ratio': 0.6, 'density': 0.85},
            'medium_packed': {'spacing': 1.0, 'hole_ratio': 0.5, 'density': 0.65},
            'loose_packed': {'spacing': 1.4, 'hole_ratio': 0.4, 'density': 0.45}
        }
```

In the very first phase we set up parameters for the materials basis on which the hat will be constructed.

Density is the material's mass per unit volume (e.g., PLA: 1.24 g/cm³), flexibility shows how bendable it is (PLA: 0.3), and breathability indicates air flow (PLA: 0.4)

Packing parameters provide a scientific framework to ensure printability, comfort, strength, cost efficiency, and design variety. They guide decisions on hole size, breathability, material density, usage, and application-specific adaptations

Calculates the actual **weight** of the printed cap for comfort assessment

```
# In calculate_comfort_metrics() -
volume_estimate = surface_area * 0.2  # Assume 2mm thickness
density = 1.25 if material == 'PLA' else 1.27 if material == 'PETG' else 1.2  # g/cm³
estimated_weight = volume_estimate * density


# In calculate_comfort_metrics() -
flexibility = material_props['flexibility'] * 100

# In overall comfort calculation -
comfort_score = (breathability * 0.4 + flexibility * 0.3 + fit_quality * 0.3)

# In calculate_comfort_metrics() -
hole_area_ratio = packing_config['hole_ratio']
breathability = min(100, (hole_area_ratio * 100 + material_props['breathability'] * 50))
```

*- Determines how **comfortable** the cap feels when worn*
- Affects the overall comfort rating (30% weight in final score)
*- Higher **flexibility** = more comfortable fit*

*- Combines material **breathability** with pattern hole density*
*- Creates a **ventilation score** (40% weight in comfort calculation)*
- Higher breathability = better airflow and cooling

PYCON JAPAN 2025

THANK YOU

@N4JP4Y