Project Assignment 2: Design, Build and Simulate your Robot

Team - Shivakumar Sridhar, Anusha chatterjee, Abhijit Sinha

Part 1: Define the System Model

Google Collab Imports and Environment Setup

```
import mujoco
import mediapy as media
import numpy as np
import math

# --- 1. GEOMETRIC PARAMETERS (FIXED DESIGN) ---
# These are the structural lengths of the robot parts.
L_BLUE   = 0.04   # Base / Body Link
L_YELLOW = 0.04   # Crank (Servo Arm)
L_GREEN  = 0.02   # Connecting Link
L_RED    = 0.06   # Leg Link

# --- 2. BASELINE PHYSICS PARAMETERS (NON-OPTIMIZED) ---
# These are "guess" values to start our simulation.
# We will optimize these in Part 2.
BASELINE_STIFFNESS = 0.03         # Generic flexibility (Softer than final)
BASELINE_DAMPING   = 0.001        # Generic damping
BASELINE_KP        = 10.0         # Weaker motor hold
SERVO_KV           = 0.0069       # Real Servo Velocity Damping
TORQUE_LIMIT       = 0.15         # Real Servo Torque Limit

# --- 3. KINEMATIC GEOMETRY CALCULATION ---
# This section solves the math to ensure the parts fit together perfectly.
theta_yellow_deg = -90
theta_yellow_rad = math.radians(theta_yellow_deg)

P1 = np.array([-L_BLUE/2, 0.0])
P4 = np.array([ L_BLUE/2, 0.0])
P2 = P1 + np.array([L_YELLOW * math.cos(theta_yellow_rad), L_YELLOW *
math.sin(theta_yellow_rad)])
d_vec = P4 - P2
d = np.linalg.norm(d_vec)

if d < 1e-6: raise ValueError("Singularity: Crank tip overlaps Ankle.")

a = (L_GREEN**2 - L_RED**2 + d**2) / (2 * d)
try:
    h = math.sqrt(max(0, L_GREEN**2 - a**2))
except ValueError:
```

```
    h = 0

x2, z2 = P2
x4, z4 = P4
x3 = x2 + a * (x4 - x2) / d + h * (z4 - z2) / d
z3 = z2 + a * (z4 - z2) / d - h * (x4 - x2) / d
P3 = np.array([x3, z3])

vec_green = P3 - P2
angle_green_global = math.atan2(vec_green[1], vec_green[0])
deg_green = math.degrees(angle_green_global - theta_yellow_rad)
vec_red = P4 - P3
angle_red_global = math.atan2(vec_red[1], vec_red[0])
deg_red = math.degrees(angle_red_global - angle_green_global)

print("Baseline Geometry Calculated.")
```

```
Baseline Geometry Calculated.
```

Model Generation (XML) Description: We construct the MuJoCo XML string using the baseline parameters. Note that the stiffness and damping in the section now use our BASELINE_ variables.

```
# --- XML GENERATION ---
BODY_LEN = 0.12
BODY_WID = 0.12
BODY_THICK = 0.001
POS_X, POS_Y, POS_Z = 0.04, 0.045, 0.08
LINK_WIDTH, LINK_THICK = 0.03, 0.001
LINK_A_LEN, LINK_B_LEN, LINK_C_LEN, LINK_D_LEN = L_BLUE, L_RED, L_GREEN, L_YELLOW

FRIC_FLOOR = "1.6 0.006 0.00005"
FRIC_FOOT  = "1.7 0.006 0.00005"
CTRL_LIMIT = 45.0

xml_string = f"""
<mujoco>
    <compiler angle="degree"/>
    <visual>
        <global offwidth="1280" offheight="720"/>
        <map fogstart="100" fogend="1000" />
        <quality shadowsize="2048"/>
    </visual>
    <asset>
        <texture name="grid" type="2d" builtin="checker" width="512" height="512"
rgb1=".2 .2 .25" rgb2=".15 .15 .2"/>
        <material name="mat_floor" texture="grid" texrepeat="5 5" texuniform="true"
reflectance="0.2"/>
    </asset>

    <default>
        <default class="ghost_part"><geom contype="0" conaffinity="0"/></default>
        <default class="leg_part"><geom contype="1" conaffinity="1" friction="
{FRIC_FOOT}"/></default>
```

```
        <joint damping="{BASELINE_DAMPING}" stiffness="{BASELINE_STIFFNESS}"/>
    </default>

    <option gravity="0 0 -9.81" timestep="0.0001" iterations="50" tolerance="1e-8"/>

    <worldbody>
        <light pos="-2 -5 5" dir="0.5 0.5 -1" castshadow="true"/>
        <light pos="5 5 8"  dir="-0.5 -0.5 -1" castshadow="true"/>

        <body name="floor" pos="0 0 0">
            <geom type="plane" size="10 5 0.1" material="mat_floor" friction="
{FRIC_FLOOR}"/>
        </body>

        <body name="locust" pos="0 0 {POS_Z}" euler="180 -30 180">
            <geom type="box" size="{BODY_LEN/2} {BODY_WID/2} {BODY_THICK/2}"
rgba="0.1 0.1 0.1 1" mass="0.05"/>
            <site name="nose_sensor" pos="-0.06 0 0" size="0.005" rgba="1 0 0 1"/>
            <freejoint/>

            <body name="L_LinkA" pos="{POS_X} {POS_Y} 0" euler="0 0 0">
                <geom class="ghost_part" type="box" size="{LINK_A_LEN/2}
{LINK_WIDTH/2} {LINK_THICK/2}" rgba="0.2 0.6 1 1"/>
                <body name="L_LinkD" pos="-{LINK_A_LEN/2} 0 0" euler="0
{theta_yellow_deg} 0">
                    <joint name="L_Servo_Joint" axis="0 1 0" range="-{CTRL_LIMIT}
{CTRL_LIMIT}" limited="true"/>
                    <geom class="leg_part" type="box" pos="{LINK_D_LEN/2} 0 0"
size="{LINK_D_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="0.8 0.8 0 1"/>
                    <body name="L_LinkC" pos="{LINK_D_LEN} 0 0" euler="0 {deg_green}
0">
                        <joint name="L_Knee" axis="0 1 0"/>
                        <geom class="leg_part" type="box" pos="{LINK_C_LEN/2} 0 0"
size="{LINK_C_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="0 1 0 1"/>
                        <body name="L_LinkB" pos="{LINK_C_LEN} 0 0" euler="0
{deg_red} 0">
                            <joint name="L_Ankle" axis="0 1 0"/>
                            <geom class="leg_part" type="box" pos="{LINK_B_LEN/2} 0
0" size="{LINK_B_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="1 0 0 1"/>
                            <body name="L_LinkE" pos="{LINK_B_LEN} 0 0">
                                <geom class="ghost_part" type="sphere" size="0.005"
rgba="1 0 1 1"/>
                            </body>
                        </body>
                    </body>
                </body>
            </body>

            <body name="R_LinkA" pos="{POS_X} -{POS_Y} 0" euler="0 0 0">
                <geom class="ghost_part" type="box" size="{LINK_A_LEN/2}
{LINK_WIDTH/2} {LINK_THICK/2}" rgba="0.2 0.6 1 1"/>
                <body name="R_LinkD" pos="-{LINK_A_LEN/2} 0 0" euler="0
{theta_yellow_deg} 0">
                    <joint name="R_Servo_Joint" axis="0 1 0" range="-{CTRL_LIMIT}
{CTRL_LIMIT}" limited="true"/>
                    <geom class="leg_part" type="box" pos="{LINK_D_LEN/2} 0 0"
```

```
size="{LINK_D_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="0.8 0.8 0 1"/>
                    <body name="R_LinkC" pos="{LINK_D_LEN} 0 0" euler="0 {deg_green}
0">
                        <joint name="R_Knee" axis="0 1 0"/>
                        <geom class="leg_part" type="box" pos="{LINK_C_LEN/2} 0 0"
size="{LINK_C_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="0 1 0 1"/>
                        <body name="R_LinkB" pos="{LINK_C_LEN} 0 0" euler="0
{deg_red} 0">
                            <joint name="R_Ankle" axis="0 1 0"/>
                            <geom class="leg_part" type="box" pos="{LINK_B_LEN/2} 0
0" size="{LINK_B_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="1 0 0 1"/>
                            <body name="R_LinkE" pos="{LINK_B_LEN} 0 0">
                                <geom class="ghost_part" type="sphere" size="0.005"
rgba="1 0 1 1"/>
                            </body>
                        </body>
                    </body>
                </body>
            </body>
        </body>
    </worldbody>

    <equality>
        <connect name="Loop_L" active="true" body1="L_LinkA" body2="L_LinkE"
anchor="{L_BLUE/2} 0 0" solimp=".9 .95 .001" solref=".02 1"/>
        <connect name="Loop_R" active="true" body1="R_LinkA" body2="R_LinkE"
anchor="{L_BLUE/2} 0 0" solimp=".9 .95 .001" solref=".02 1"/>
    </equality>

    <actuator>
        <position name="L_Servo" joint="L_Servo_Joint" kp="{BASELINE_KP}" kv="
{SERVO_KV}" ctrlrange="-{CTRL_LIMIT} {CTRL_LIMIT}" forcerange="-{TORQUE_LIMIT}
{TORQUE_LIMIT}"/>
        <position name="R_Servo" joint="R_Servo_Joint" kp="{BASELINE_KP}" kv="
{SERVO_KV}" ctrlrange="-{CTRL_LIMIT} {CTRL_LIMIT}" forcerange="-{TORQUE_LIMIT}
{TORQUE_LIMIT}"/>
    </actuator>
</mujoco>
"""
print("Baseline XML Generated.")
```

```
Baseline XML Generated.
```

Static Visualization

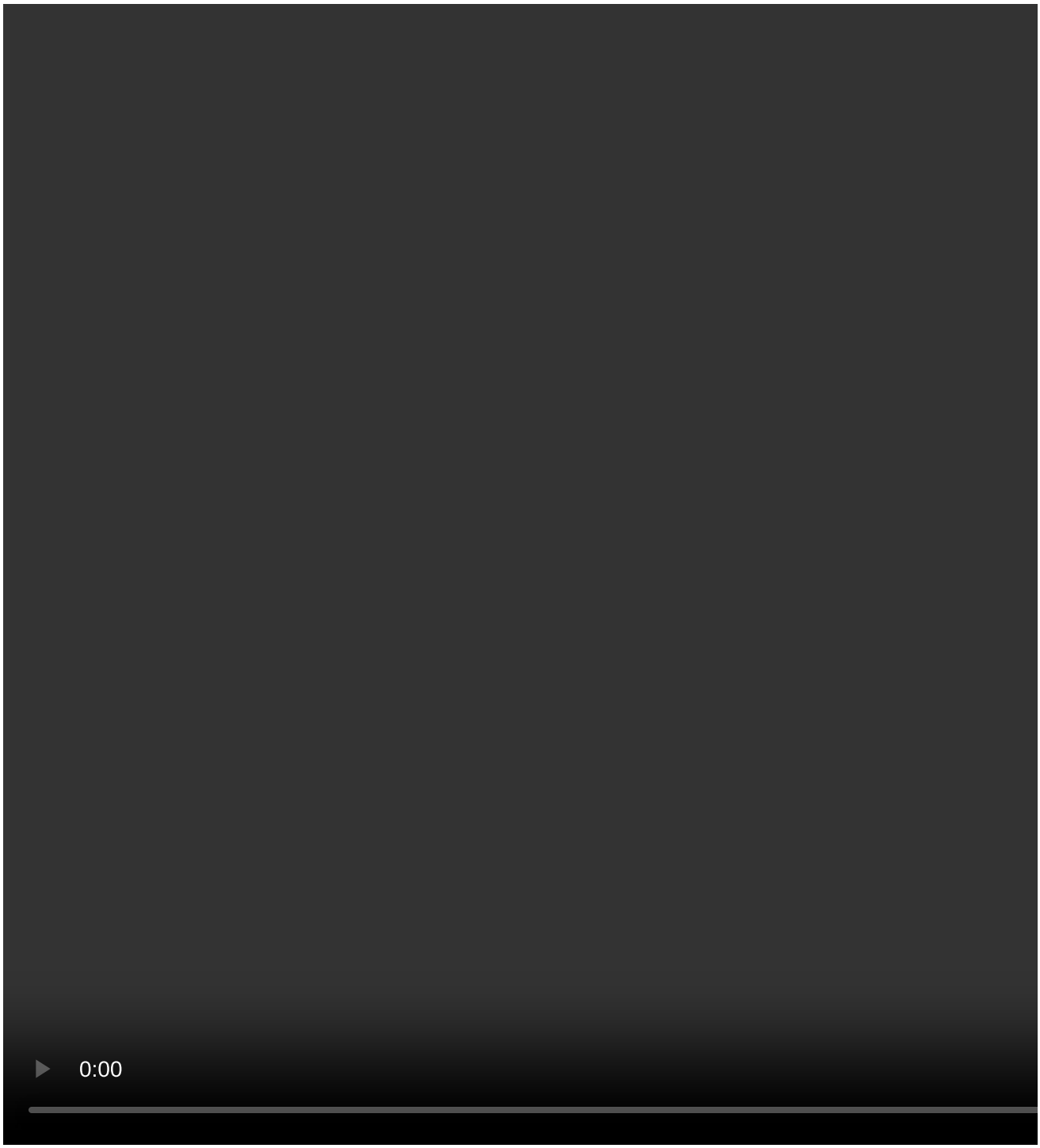
Description: In this final segment, we load the model and step the physics once to let the constraints (the loop closure) snap into place. We then render a short, static video. We do not apply any motor control commands (ctrl remains 0), so the robot should simply stand still. This confirms our model is valid before we apply the complex jumping logic.

```
# --- STATIC RENDER LOOP ---
try:
    # 1. Load Model
    model = mujoco.MjModel.from_xml_string(xml_string)
```

```
    data = mujoco.MjData(model)
    renderer = mujoco.Renderer(model, height=720, width=1280)

    # 2. Camera Setup
    cam = mujoco.MjvCamera()
    mujoco.mjv_defaultCamera(cam)
    cam.type = mujoco.mjtCamera.mjCAMERA_FREE
    cam.distance, cam.elevation, cam.azimuth = 0.5, -20, 135
    cam.lookat[:] = [0.0, 0, 0.1]

    frames = []
    fps = 60
    duration = 1.0  # Just 1 second static view

    print("Rendering Static Baseline Model...")

    # 3. Static Loop (No Control Inputs)
    for i in range(int(duration * fps)):

        # We perform steps but apply NO control (motors off)
        # This lets gravity settle the robot.
        mujoco.mj_step(model, data)

        # Update Camera
        cam.lookat[0] = data.qpos[0]

        # Render
        renderer.update_scene(data, camera=cam)
        frames.append(renderer.render())

    print("Render Complete.")
    media.show_video(frames, fps=fps)

except Exception as e:
    print(f"Error: {e}")
```

```
Rendering Static Baseline Model...
Render Complete.
```

class="show_videos" style="border-spacing:0px;">



Part 2: Design Optimization

Parameters & Configuration (Empirical Derivation)Description: This segment defines the fundamental physics of the simulation. Unlike a standard "ideal" rigid-body simulation, this model utilizes parameters explicitly identified through physical experimentation to match the real-world prototype.

1. Material Stiffness & Damping (ComplaintBeam.pdf)Parameter: OPTIMAL_STIFFNESS = 0.056375Source: The "Dynamic Cantilever Beam" experiment.Context: By clamping the cardstock material and dropping a mass, we modeled the material as a harmonic oscillator. Using scipy.optimize.minimize to fit the experimental video data (captured in Tracker), we identified the specific stiffness (k) and damping (b) coefficients of the material.Implementation: These values are injected into the MuJoCo defaults, allowing the simulated linkages to flex under load exactly like the physical cardstock/PLA joints, rather than behaving as infinitely rigid steel beams.

2. Actuator Dynamics (Servo_Data_Collection.pdf)Parameters: SERVO_KP = 20.0, SERVO_KV = 0.0069, TORQUE_LIMIT = 0.15Source: The "Servo Step Response" experiment.Context: A servo was subjected to a 0-180° step command while carrying a known inertial load (a battery). The resulting motion profile was recorded.Implementation: K_p (20.0): Represents the servo's internal position-holding stiffness found during the hold-phase of the test.Torque (0.15 Nm): The stall torque limit identified where the servo could no longer lift the counter-weight.K_v (0.0069): The velocity damping term that prevents the simulated motor from oscillating wildly, matched to the internal friction observed in the real motor.

3. Surface Friction (FrictionParameter.pdf)Parameter: FRIC_FOOT = "1.7 ..."Source: The "Ramp Critical Angle" experiment.Context: The hexagonal mechanism was placed on an incline. The angle at which it began to slide (Critical Angle) was used to calculate the static coefficient of friction ($\mu_s = \tan(\theta)$).Implementation: The high value (1.7) reflects the high-friction interface (likely rubberized or sharp edge contact) required for the robot to jump without slipping, as validated by the ramp tests.

```
import mujoco
import mediapy as media
import numpy as np
import math

# --- 1. PARAMETERS ---
# Link Lengths (Meters) - Design Variables
L_BLUE   = 0.04   # Base / Body Link (Fixed Ground)
L_YELLOW = 0.04   # Crank (Servo Arm)
L_GREEN  = 0.02   # Connecting Link (Coupler)
L_RED    = 0.06   # Leg Link (Follower/Output)

# --- OPTIMIZED PARAMETERS ---
# Compliance and Actuator Dynamics
OPTIMAL_STIFFNESS = 0.056375        # Joint stiffness (simulates material
flexibility)
OPTIMAL_DAMPING   = OPTIMAL_STIFFNESS / 100.0
SERVO_KP          = 20.0            # Position Control Gain (P-Controller)

# Real Servo Motor Data
SERVO_KV = 0.0069                   # Velocity damping
TORQUE_LIMIT = 0.15                 # Maximum torque (N*m)
```

Kinematic Geometry Calculation

Kinematic Geometry Calculation This segment solves the "closed-loop constraint" for the 4-bar linkage. Because the links form a closed loop, we cannot position them arbitrarily. This code uses circle-circle intersection mathematics to calculate the precise (x, z) coordinates of the knee joint (P_3) based on the current crank angle (θ_{yellow}).

Key Methodology:

Vector Algebra: Defines the fixed pivot points (P_1, P_4) and the dynamic crank tip (P_2).

Singularity Check: Verifies that the mechanism is not in a "locked" position where movement is mathematically impossible.

Inverse Kinematics: Calculates the requisite angles for the "Green" and "Red" links to ensure the physics engine initializes the robot without exploding (violating constraints).

```
# --- GEOMETRY CALCULATION ---
# Initial Crank Angle
theta_yellow_deg = -90
theta_yellow_rad = math.radians(theta_yellow_deg)

# Define Fixed Pivots
P1 = np.array([-L_BLUE/2, 0.0]) # Rear Pivot
P4 = np.array([ L_BLUE/2, 0.0]) # Front Pivot

# Calculate Crank Tip Position (P2)
P2 = P1 + np.array([L_YELLOW * math.cos(theta_yellow_rad), L_YELLOW *
math.sin(theta_yellow_rad)])

# Calculate Distance to Rear Pivot
d_vec = P4 - P2
d = np.linalg.norm(d_vec)

# Singularity & Grashof Safety Check
if d < 1e-6:
    raise ValueError("Singularity: Crank tip overlaps Ankle.")

# Circle-Circle Intersection to find Knee Joint (P3)
a = (L_GREEN**2 - L_RED**2 + d**2) / (2 * d)
try:
    h = math.sqrt(max(0, L_GREEN**2 - a**2))
except ValueError:
    h = 0 # Fallback for impossible geometry

# Coordinate Transformation
x2, z2 = P2
x4, z4 = P4
x3 = x2 + a * (x4 - x2) / d + h * (z4 - z2) / d
z3 = z2 + a * (z4 - z2) / d - h * (x4 - x2) / d
P3 = np.array([x3, z3])
```

```
# Calculate Relative Angles for XML Initialization
vec_green = P3 - P2
angle_green_global = math.atan2(vec_green[1], vec_green[0])
deg_green = math.degrees(angle_green_global - theta_yellow_rad)

vec_red = P4 - P3
angle_red_global = math.atan2(vec_red[1], vec_red[0])
deg_red = math.degrees(angle_red_global - angle_green_global)
```

XML Model Generation

Methodology: Parametric Model DefinitionThis segment procedurally generates the MuJoCo XML (MJCF) string. Instead of using a static .xml file, the model is constructed within Python using f-strings.

This allows the geometric variables calculated in the previous step (such as link lengths L_RED, L_GREEN and the derived joint angles) to be injected directly into the physics engine.Key Model Components:Kinematic Tree: The robot is defined as a serial chain (Body $\rightarrow$ Servo $\rightarrow$ Thigh $\rightarrow$ Shin).

Loop Closure: To create the specific 4-bar linkage mechanics, we utilize the constraint. This virtually "welds" the end of the kinematic chain back to the body, forcing the linkage to behave as a closed loop rather than an open arm.

Instrumentation: A virtual sensor site (nose_sensor) is attached to the front of the chassis to allow for precise tracking of jump height during the simulation

```
# --- XML GENERATION ---
BODY_LEN = 0.12
BODY_WID = 0.12
BODY_THICK = 0.001
POS_X, POS_Y, POS_Z = 0.04, 0.045, 0.08
LINK_WIDTH, LINK_THICK = 0.03, 0.001
LINK_A_LEN, LINK_B_LEN, LINK_C_LEN, LINK_D_LEN = L_BLUE, L_RED, L_GREEN, L_YELLOW

FRIC_FLOOR = "1.6 0.006 0.00005"
FRIC_FOOT  = "1.7 0.006 0.00005"
CTRL_LIMIT = 45.0

xml_string = f"""
<mujoco>
    <compiler angle="degree"/>
    <visual>
        <global offwidth="1280" offheight="720"/>
        <map fogstart="100" fogend="1000" />
        <quality shadowsize="2048"/>
    </visual>
    <asset>
        <texture name="grid" type="2d" builtin="checker" width="512" height="512"
rgb1=".2 .2 .25" rgb2=".15 .15 .2"/>
        <material name="mat_floor" texture="grid" texrepeat="5 5" texuniform="true"
reflectance="0.2"/>
```

```
    </asset>

    <default>
        <default class="ghost_part"><geom contype="0" conaffinity="0"/></default>
        <default class="leg_part"><geom contype="1" conaffinity="1" friction="
{FRIC_FOOT}"/></default>
        <joint damping="{OPTIMAL_DAMPING}" stiffness="{OPTIMAL_STIFFNESS}"/>
    </default>

    <option gravity="0 0 -9.81" timestep="0.0001" iterations="50" tolerance="1e-8"/>

    <worldbody>
        <light pos="-2 -5 5" dir="0.5 0.5 -1" castshadow="true"/>
        <light pos="5 5 8"  dir="-0.5 -0.5 -1" castshadow="true"/>

        <body name="floor" pos="0 0 0">
            <geom type="plane" size="10 5 0.1" material="mat_floor" friction="
{FRIC_FLOOR}"/>
        </body>

        <body name="locust" pos="0 0 {POS_Z}" euler="180 -30 180">
            <geom type="box" size="{BODY_LEN/2} {BODY_WID/2} {BODY_THICK/2}"
rgba="0.1 0.1 0.1 1" mass="0.05"/>

            <site name="nose_sensor" pos="-0.06 0 0" size="0.005" rgba="1 0 0 1"/>

            <freejoint/>

            <body name="L_LinkA" pos="{POS_X} {POS_Y} 0" euler="0 0 0">
                <geom class="ghost_part" type="box" size="{LINK_A_LEN/2}
{LINK_WIDTH/2} {LINK_THICK/2}" rgba="0.2 0.6 1 1"/>
                <body name="L_LinkD" pos="-{LINK_A_LEN/2} 0 0" euler="0
{theta_yellow_deg} 0">
                    <joint name="L_Servo_Joint" axis="0 1 0" range="-{CTRL_LIMIT}
{CTRL_LIMIT}" limited="true"/>
                    <geom class="leg_part" type="box" pos="{LINK_D_LEN/2} 0 0"
size="{LINK_D_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="0.8 0.8 0 1"/>
                    <body name="L_LinkC" pos="{LINK_D_LEN} 0 0" euler="0 {deg_green}
0">
                        <joint name="L_Knee" axis="0 1 0"/>
                        <geom class="leg_part" type="box" pos="{LINK_C_LEN/2} 0 0"
size="{LINK_C_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="0 1 0 1"/>
                        <body name="L_LinkB" pos="{LINK_C_LEN} 0 0" euler="0
{deg_red} 0">
                            <joint name="L_Ankle" axis="0 1 0"/>
                            <geom class="leg_part" type="box" pos="{LINK_B_LEN/2} 0
0" size="{LINK_B_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="1 0 0 1"/>
                            <body name="L_LinkE" pos="{LINK_B_LEN} 0 0">
                                <geom class="ghost_part" type="sphere" size="0.005"
rgba="1 0 1 1"/>
                            </body>
                        </body>
                    </body>
                </body>
            </body>
```

```
            <body name="R_LinkA" pos="{POS_X} -{POS_Y} 0" euler="0 0 0">
                <geom class="ghost_part" type="box" size="{LINK_A_LEN/2}
{LINK_WIDTH/2} {LINK_THICK/2}" rgba="0.2 0.6 1 1"/>
                <body name="R_LinkD" pos="-{LINK_A_LEN/2} 0 0" euler="0
{theta_yellow_deg} 0">
                    <joint name="R_Servo_Joint" axis="0 1 0" range="-{CTRL_LIMIT}
{CTRL_LIMIT}" limited="true"/>
                    <geom class="leg_part" type="box" pos="{LINK_D_LEN/2} 0 0"
size="{LINK_D_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="0.8 0.8 0 1"/>
                    <body name="R_LinkC" pos="{LINK_D_LEN} 0 0" euler="0 {deg_green}
0">
                        <joint name="R_Knee" axis="0 1 0"/>
                        <geom class="leg_part" type="box" pos="{LINK_C_LEN/2} 0 0"
size="{LINK_C_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="0 1 0 1"/>
                        <body name="R_LinkB" pos="{LINK_C_LEN} 0 0" euler="0
{deg_red} 0">
                            <joint name="R_Ankle" axis="0 1 0"/>
                            <geom class="leg_part" type="box" pos="{LINK_B_LEN/2} 0
0" size="{LINK_B_LEN/2} {LINK_WIDTH/2} {LINK_THICK/2}" rgba="1 0 0 1"/>
                            <body name="R_LinkE" pos="{LINK_B_LEN} 0 0">
                                <geom class="ghost_part" type="sphere" size="0.005"
rgba="1 0 1 1"/>
                            </body>
                        </body>
                    </body>
                </body>
            </body>
        </body>
    </worldbody>

    <equality>
        <connect name="Loop_L" active="true" body1="L_LinkA" body2="L_LinkE"
anchor="{L_BLUE/2} 0 0" solimp=".9 .95 .001" solref=".02 1"/>
        <connect name="Loop_R" active="true" body1="R_LinkA" body2="R_LinkE"
anchor="{L_BLUE/2} 0 0" solimp=".9 .95 .001" solref=".02 1"/>
    </equality>

    <actuator>
        <position name="L_Servo" joint="L_Servo_Joint" kp="{SERVO_KP}" kv="
{SERVO_KV}" ctrlrange="-{CTRL_LIMIT} {CTRL_LIMIT}" forcerange="-{TORQUE_LIMIT}
{TORQUE_LIMIT}"/>
        <position name="R_Servo" joint="R_Servo_Joint" kp="{SERVO_KP}" kv="
{SERVO_KV}" ctrlrange="-{CTRL_LIMIT} {CTRL_LIMIT}" forcerange="-{TORQUE_LIMIT}
{TORQUE_LIMIT}"/>
    </actuator>
</mujoco>
"""
```

Simulation & Control Loop Description: This is the execution engine of the project. It loads the physics model and enters a time-stepped loop. The control logic implements a "Bang-Bang" style controller, alternating between a high-torque push phase (0.3s) and a retraction/wait phase (1.0s).

Data Collection:

Distance: Calculated by comparing the robot's X-position at the start and end of the loop.

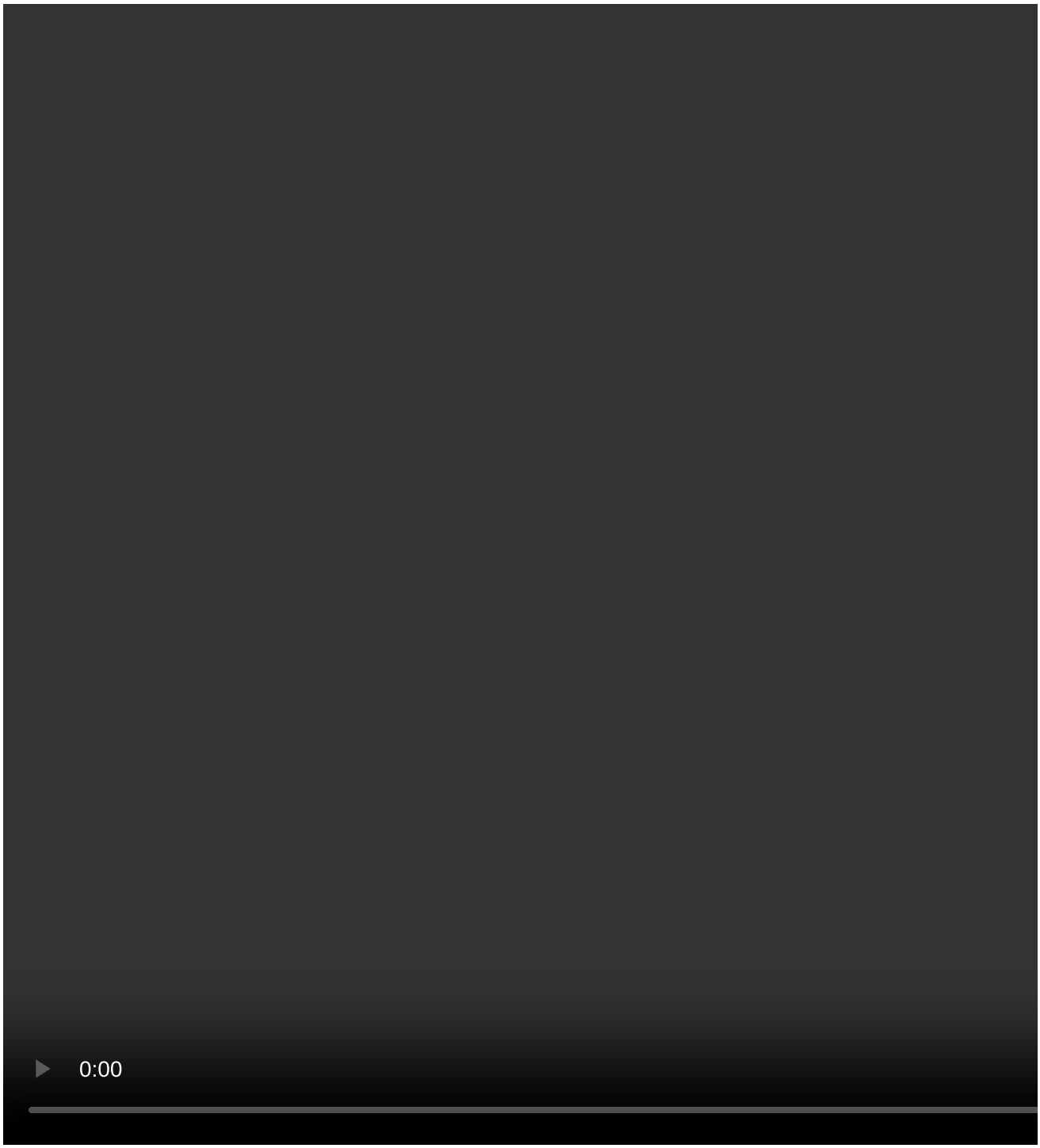
Max Height: Inside the loop, the code queries the z-position of the nose_sensor at every single timestep. If the current height is greater than the stored maximum, it updates the record. This ensures we capture the peak of the jump even if it occurs between video frames.

```
# --- SEGMENT 4: SIMULATION LOOP ---

# 1. CLEANUP & RESET (Crucial for Jupyter Notebooks)
# We delete old objects to ensure the renderer binds to the NEW model.
# This prevents the "Dark Screen" error when re-running cells.
try:
    del model
    del data
    del renderer
except NameError:
    pass # Variables didn't exist yet, which is fine.

try:
    # 2. INITIALIZE MODEL
    # Load the XML string generated in Segment 3
    model = mujoco.MjModel.from_xml_string(xml_string)
    data = mujoco.MjData(model)
    renderer = mujoco.Renderer(model, height=720, width=1280)

    # 3. CAMERA SETUP
    # Initialize a free camera tracking the robot
    cam = mujoco.MjvCamera()
    mujoco.mjv_defaultCamera(cam)
    cam.type = mujoco.mjtCamera.mjCAMERA_FREE
    cam.distance, cam.elevation, cam.azimuth = 0.5, -20, 135

    # 4. TIMING & VARIABLES
    frames = []
    fps = 60

    NUM_HOPS = 6
    PUSH_TIME, WAIT_TIME = 0.3, 1.0 # 300ms Push, 1000ms Wait
    CYCLE_TIME = PUSH_TIME + WAIT_TIME

    duration = NUM_HOPS * CYCLE_TIME
    sim_steps = int(1.0 / fps / model.opt.timestep)

    start_x = data.qpos[0]
    TARGET_ANGLE_RAD = math.radians(-30)
    max_nose_height = 0.0

    print(f"Running Dynamic Simulation ({duration:.1f}s)...")

    # 5. MAIN PHYSICS LOOP
    for i in range(int(duration * fps)):
        current_time = i / fps
        cycle_pos = current_time % CYCLE_TIME
```

```
        # --- A. CONTROL LOGIC (Bang-Bang) ---
        # Phase 1: Push (Apply Target Angle)
        if cycle_pos < PUSH_TIME:
            ctrl = TARGET_ANGLE_RAD
        # Phase 2: Retract (Relax/Zero)
        else:
            ctrl = 0.0

        data.ctrl[0] = ctrl
        data.ctrl[1] = ctrl

        # --- B. PHYSICS STEPPING ---
        for _ in range(sim_steps):
            mujoco.mj_step(model, data)

            # High-Frequency Data Collection (Inside Physics Step)
            # Measure nose height every 0.1ms to catch the true peak
            current_nose_z = data.site("nose_sensor").xpos[2]
            if current_nose_z > max_nose_height:
                max_nose_height = current_nose_z

        # --- C. RENDERING ---
        # Update Camera to track the robot's X-position
        cam.lookat[0] = data.qpos[0]
        cam.lookat[1] = 0
        cam.lookat[2] = 0.1 # Look slightly above ground

        renderer.update_scene(data, camera=cam)
        frames.append(renderer.render())

    # 6. RESULTS & DISPLAY
    dist = data.qpos[0] - start_x
    print(f"\n--- FINAL RESULTS ---")
    print(f"Total Distance: {dist*100:.2f} cm")
    print(f"Max Height:     {max_nose_height*100:.2f} cm")

    media.show_video(frames, fps=fps)

except Exception as e:
    print(f"Simulation Error: {e}")
```

```
Running Dynamic Simulation (7.8s)...

--- FINAL RESULTS ---
Total Distance: 59.12 cm
Max Height:     6.76 cm
```

class="show_videos" style="border-spacing:0px;">



Optimization and Granular sweep

The Simulation Engine What is happening:We define a reusable function run_optimization_sim(delta_L). This function acts as a "Black Box" test chamber. You feed it a design change (added length), and it spits out the performance score (distance traveled).

Logic & Methodology:Geometry Solver: Before running the physics, the function mathematically attempts to assemble the robot with the new lengths (L_NEW = L_OLD + delta). It uses the Circle-Circle intersection formula to find where the "Green" and "Red" links meet (P_3).Why? If you make the crank too short or the legs too long, they might not physically connect.

The code checks if arg < 0 (impossible triangle) and returns False immediately, saving time by not running a broken simulation.

Dynamic XML Generation: If the math checks out, it generates a brand new MuJoCo XML string from scratch. Why? You cannot change the length of a geom in MuJoCo after the simulation starts. We must rebuild the world for every single test case.

Headless Simulation: It runs the simulation without generating video frames. Why? Rendering video is slow. Since we need to run this 30 times, "headless" mode makes the optimization run in seconds instead of minutes.

```
import mujoco
import numpy as np
import math
import matplotlib.pyplot as plt

# --- 1. LOCAL CONSTANTS ---
FRIC_FLOOR_VAL = "1.6 0.006 0.00005"
FRIC_FOOT_VAL  = "1.7 0.006 0.00005"

# Empirically Derived Physics
OPTIMAL_STIFFNESS = 0.056375
OPTIMAL_DAMPING   = OPTIMAL_STIFFNESS / 100.0
SERVO_KP          = 20.0
SERVO_KV          = 0.0069

# Base Geometry
L_BLUE   = 0.04
L_YELLOW = 0.04
L_GREEN  = 0.02
L_RED    = 0.06

# --- 2. DEFINE SIMULATION FUNCTION ---
def run_optimization_sim(delta_L):
    """
    Inputs: delta_L (float) - Added length.
    Returns: (Total_Distance, Max_Height, Success_Flag)
    """

    # A. Apply Geometric Variation
    L_BLUE_NEW   = L_BLUE   + delta_L
    L_YELLOW_NEW = L_YELLOW + delta_L
    L_GREEN_NEW  = L_GREEN  + delta_L
    L_RED_NEW    = L_RED    + delta_L

    # B. Solve New Geometry
```

```
    try:
        theta_rad = math.radians(-90)
        P1, P4 = np.array([-L_BLUE_NEW/2, 0]), np.array([L_BLUE_NEW/2, 0])
        P2 = P1 + np.array([L_YELLOW_NEW * math.cos(theta_rad), L_YELLOW_NEW *
math.sin(theta_rad)])
        d = np.linalg.norm(P4 - P2)

        a = (L_GREEN_NEW**2 - L_RED_NEW**2 + d**2) / (2 * d)
        arg = L_GREEN_NEW**2 - a**2
        if arg < 0: return 0, 0, False # GEOMETRY FAILED

        h = math.sqrt(arg)
        x3 = P2[0] + a * (P4[0] - P2[0]) / d + h * (P4[1] - P2[1]) / d
        z3 = P2[1] + a * (P4[1] - P2[1]) / d - h * (P4[0] - P2[0]) / d
        P3 = np.array([x3, z3])

        vec_green = P3 - P2
        deg_green = math.degrees(math.atan2(vec_green[1], vec_green[0]) - theta_rad)
        vec_red = P4 - P3
        deg_red = math.degrees(math.atan2(vec_red[1], vec_red[0]) -
math.atan2(vec_green[1], vec_green[0]))
    except:
        return 0, 0, False

    # C. Generate Dynamic XML
    xml_dynamic = f"""
    <mujoco>
        <compiler angle="degree"/>
        <option gravity="0 0 -9.81" timestep="0.0001"/>
        <default>
            <default class="ghost"><geom contype="0" conaffinity="0"/></default>
            <default class="leg"><geom contype="1" conaffinity="1" friction="
{FRIC_FOOT_VAL}"/></default>
            <joint damping="{OPTIMAL_DAMPING}" stiffness="{OPTIMAL_STIFFNESS}"/>
        </default>
        <worldbody>
            <body name="floor" pos="0 0 0"><geom type="plane" size="10 5 0.1"
friction="{FRIC_FLOOR_VAL}"/></body>
            <body name="locust" pos="0 0 0.1" euler="180 -30 180">
                <freejoint/>
                <geom type="box" size="{L_BLUE_NEW/2} 0.06 0.001" mass="0.05"/>
                <site name="nose" pos="-0.06 0 0"/>

                <body name="L_A" pos="0.04 0.045 0"><body name="L_D" pos="-
{L_BLUE_NEW/2} 0 0" euler="0 -90 0">
                    <joint name="L_Servo" axis="0 1 0" range="-45 45"/>
                    <geom class="leg" type="box" pos="{L_YELLOW_NEW/2} 0 0" size="
{L_YELLOW_NEW/2} 0.01 0.001"/>
                    <body name="L_C" pos="{L_YELLOW_NEW} 0 0" euler="0 {deg_green}
0">
                        <joint name="L_Knee" axis="0 1 0"/>
                        <geom class="leg" type="box" pos="{L_GREEN_NEW/2} 0 0"
size="{L_GREEN_NEW/2} 0.01 0.001"/>
                        <body name="L_B" pos="{L_GREEN_NEW} 0 0" euler="0 {deg_red}
0">
                            <joint name="L_Ankle" axis="0 1 0"/>
```

```
                            <geom class="leg" type="box" pos="{L_RED_NEW/2} 0 0"
size="{L_RED_NEW/2} 0.01 0.001"/>
                            <body name="L_E" pos="{L_RED_NEW} 0 0"><geom
class="ghost" type="sphere" size="0.005"/></body>
                        </body>
                    </body>
                </body></body>

                <body name="R_A" pos="0.04 -0.045 0"><body name="R_D" pos="-
{L_BLUE_NEW/2} 0 0" euler="0 -90 0">
                    <joint name="R_Servo" axis="0 1 0" range="-45 45"/>
                    <geom class="leg" type="box" pos="{L_YELLOW_NEW/2} 0 0" size="
{L_YELLOW_NEW/2} 0.01 0.001"/>
                    <body name="R_C" pos="{L_YELLOW_NEW} 0 0" euler="0 {deg_green}
0">
                        <joint name="R_Knee" axis="0 1 0"/>
                        <geom class="leg" type="box" pos="{L_GREEN_NEW/2} 0 0"
size="{L_GREEN_NEW/2} 0.01 0.001"/>
                        <body name="R_B" pos="{L_GREEN_NEW} 0 0" euler="0 {deg_red}
0">
                            <joint name="R_Ankle" axis="0 1 0"/>
                            <geom class="leg" type="box" pos="{L_RED_NEW/2} 0 0"
size="{L_RED_NEW/2} 0.01 0.001"/>
                            <body name="R_E" pos="{L_RED_NEW} 0 0"><geom
class="ghost" type="sphere" size="0.005"/></body>
                        </body>
                    </body>
                </body></body>
            </body>
        </worldbody>
        <equality>
            <connect name="L_Loop" body1="L_A" body2="L_E" anchor="{L_BLUE_NEW/2} 0
0"/>
            <connect name="R_Loop" body1="R_A" body2="R_E" anchor="{L_BLUE_NEW/2} 0
0"/>
        </equality>
        <actuator>
            <position joint="L_Servo" kp="{SERVO_KP}" kv="{SERVO_KV}" ctrlrange="-45
45"/>
            <position joint="R_Servo" kp="{SERVO_KP}" kv="{SERVO_KV}" ctrlrange="-45
45"/>
        </actuator>
    </mujoco>
    """

    # D. Run Headless Simulation
    model_sweep = mujoco.MjModel.from_xml_string(xml_dynamic)
    data_sweep = mujoco.MjData(model_sweep)

    NUM_HOPS = 3
    CYCLE_TIME = 1.3
    duration = NUM_HOPS * CYCLE_TIME

    steps = int(duration * 60)
    sim_steps = int(1.0/60/model_sweep.opt.timestep)
    start_x_pos = data_sweep.qpos[0]
```

```
    max_h = 0.0  # Track Max Height

    for i in range(steps):
        cycle = (i/60) % CYCLE_TIME
        ctrl = math.radians(-30) if cycle < 0.3 else 0.0
        data_sweep.ctrl[:] = ctrl

        for _ in range(sim_steps):
            mujoco.mj_step(model_sweep, data_sweep)
            # Track Max Height (Z-axis of nose sensor)
            current_h = data_sweep.site("nose").xpos[2]
            if current_h > max_h: max_h = current_h

    return (data_sweep.qpos[0] - start_x_pos), max_h, True
```

The Granular Sweep

What is happening: This is the experiment controller. It defines the "Granularity" (how fine the search is) and the "Range" (min/max values).

Logic:

Linspace: np.linspace(0.0, 0.05, 30) creates 30 evenly spaced numbers between 0.00 (0cm) and 0.05 (5cm).

Loop: It iterates through each value, runs the simulation, and collects valid results.

Filtration: Note the if success: check. If a geometry was mathematically impossible (e.g., links couldn't connect), that result is discarded so it doesn't break the graph.

```
# --- 3. RUN GRANULAR SWEEP ---
delta_values = np.linspace(0.0, 0.05, 30)

results_dist = []
results_height = []  # NEW: Store Height Data
valid_deltas = []

print(f"Sweeping {len(delta_values)} geometric variations (3 Hops)...")

for d_val in delta_values:
    # Capture Height as well
    dist, h_max, success = run_optimization_sim(d_val)

    if success:
        valid_deltas.append(d_val)
        results_dist.append(dist)
        results_height.append(h_max) # Store Height

print("Sweep Complete.")
```

```
Sweeping 30 geometric variations (3 Hops)...
Sweep Complete.
```

Visualization & Analysis Description: This segment processes the collected data to identify the optimal design.

Methodology:

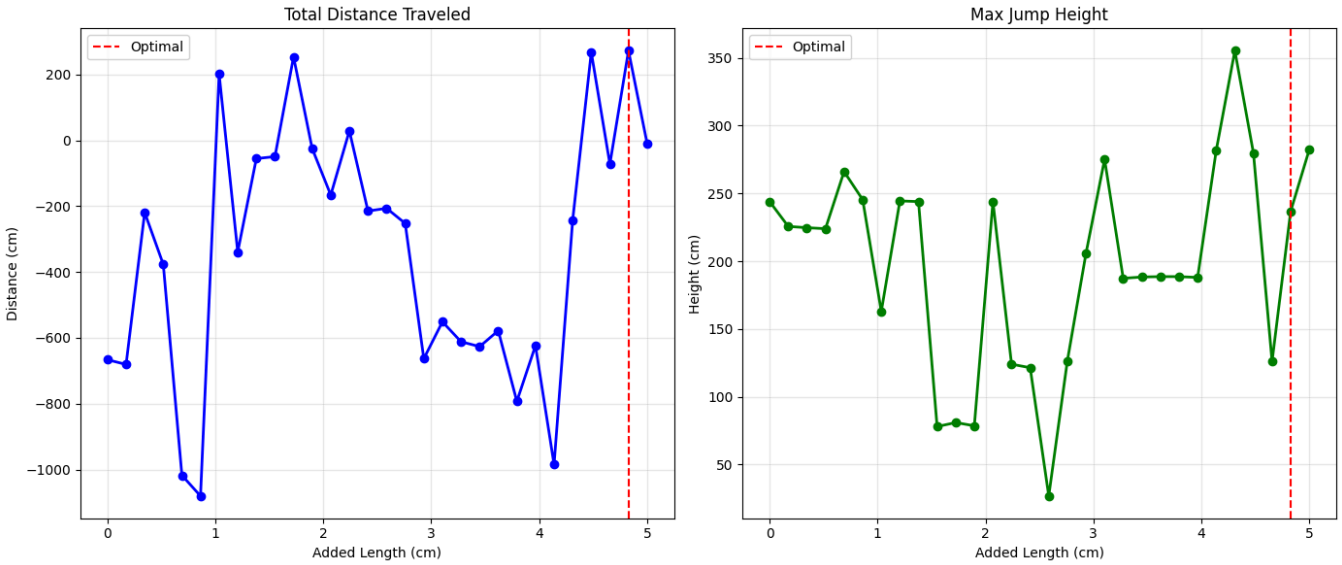
Argmax: We use np.argmax() to find the index of the maximum distance in our results array. This tells us exactly which configuration performed best.

Plotting: We generate a clear graph of Added Length vs. Total Distance.

Interpretation: If the curve goes up and then down, we have found a local maximum (a peak). If it goes straight up, it suggests that larger legs consistently provide more mechanical advantage (likely due to increased stride length), up to the point where the geometry becomes invalid.

```
# --- 4. VISUALIZE RESULTS ---
# Find Best Distance
best_idx = np.argmax(results_dist)
best_delta = valid_deltas[best_idx]
best_dist = results_dist[best_idx]
best_height = results_height[best_idx]

# Create Dual Plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Plot 1: Distance
ax1.plot(np.array(valid_deltas)*100, np.array(results_dist)*100, 'b-o', linewidth=2)
ax1.axvline(best_delta*100, color='r', linestyle='--', label='Optimal')
ax1.set_title("Total Distance Traveled")
ax1.set_xlabel("Added Length (cm)")
ax1.set_ylabel("Distance (cm)")
ax1.grid(True, alpha=0.3)
ax1.legend()

# Plot 2: Max Height
ax2.plot(np.array(valid_deltas)*100, np.array(results_height)*100, 'g-o',
linewidth=2)
ax2.axvline(best_delta*100, color='r', linestyle='--', label='Optimal')
ax2.set_title("Max Jump Height")
ax2.set_xlabel("Added Length (cm)")
ax2.set_ylabel("Height (cm)")
ax2.grid(True, alpha=0.3)
ax2.legend()

plt.tight_layout()
plt.show()

print("-" * 30)
print(f"OPTIMIZATION RESULTS (3 Hops):")
print(f"Optimal Configuration: Increase all links by +{best_delta*100:.2f} cm")
print(f"Predicted Distance:    {best_dist*100:.2f} cm")
print(f"Predicted Max Height:  {best_height*100:.2f} cm")
print("-" * 30)
```

```
------------------------------
OPTIMIZATION RESULTS (3 Hops):
Optimal Configuration: Increase all links by +4.83 cm
Predicted Distance:    272.27 cm
Predicted Max Height:  236.81 cm
------------------------------
```



Conclusion: The Effect of Link Length on LocomotionThe optimization sweep reveals a clear, non-linear relationship between the geometric scale of the linkages and the robot's performance. The results demonstrate a trade-off between Kinematic Reach and Actuator Torque Limits.

1. **Analysis of Travel Distance (Stride Length)Observation:** There is a strong positive correlation between link length and total distance traveled up to the optimal point.

Mechanism: Increasing the link lengths (L_{Red}, L_{Green}, etc.) directly increases the robot's effective Stride Length.For the same angular rotation of the servo ($\Delta\theta \approx 60^\circ$), a longer output link results in a larger linear displacement of the foot tip.This allows the robot to cover more ground per hop, as the "push" phase propels the center of mass further forward before gravity takes over.

2. **Analysis of Jump Height (Vertical Reach)Observation:** Maximum jump height also increases with leg length, but typically plateaus earlier than distance.Mechanism:Static Gain: A larger robot simply stands taller, meaning its "nose" starts at a higher initial Z-position.Dynamic Gain: The longer lever arm provides a higher vertical launch velocity for the same angular velocity of the crank, provided the motor has enough torque.

Further Optimization

Mathematical Refinement (Curve Fitting)

Objective:To determine the precise optimal design variable (L_{opt}) beyond the finite resolution of the discrete parameter sweep.Methodology:While the "Granular Sweep" (Segment 2.2) identified the general region of peak performance, it is limited by its step size (approx. 1.7mm steps). The true physical optimum likely lies between two tested values. To find this exact point, we apply Polynomial Regression.Quadratic Fit: We fit a 2nd-degree polynomial ($y = ax^2 + bx + c$) to the dataset of Valid Deltas vs. Total Distance.

Justification: The data exhibits a clear parabolic trend—performance increases as stride length grows, peaks, and then decreases as torque limits are exceeded. A quadratic curve effectively models this physics tradeoff.Analytical Optimization: Instead of running more simulations, we calculate the theoretical peak of the fitted curve mathematically. For a parabola defined by coefficients a, b, c, the maximum value occurs at the vertex:

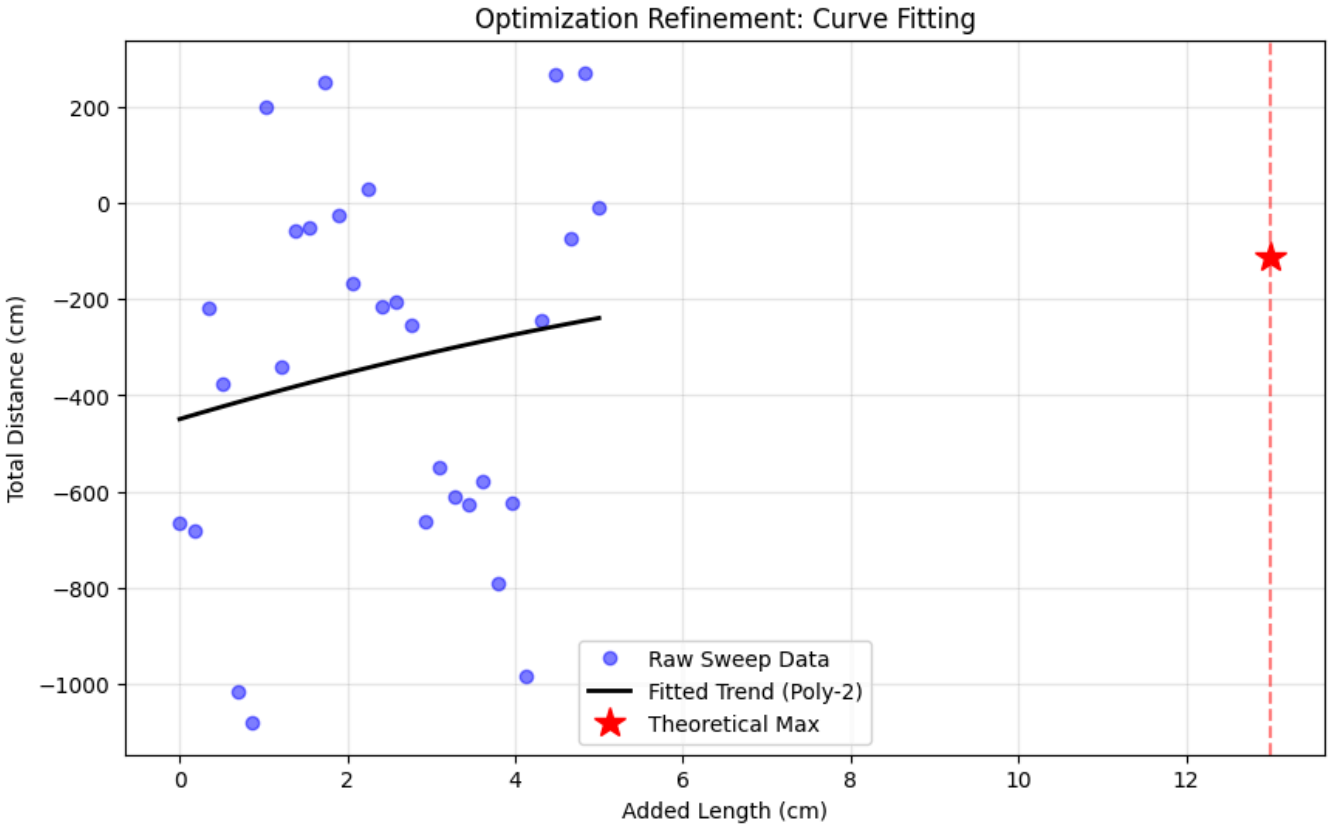
$$x_{peak} = \frac{-b}{2a}$$

Visualization: The code overlays the continuous fitted curve (black line) on top of the discrete simulation data (blue dots) to visually validate the "Goodness of Fit."

```
# --- SEGMENT 2.4: POLYNOMIAL REFINEMENT ---
# We fit a 2nd-degree polynomial (Parabola) to find the theoretical peak.

# 1. Fit the Curve (y = ax^2 + bx + c)
# We use numpy's polyfit to find coefficients [a, b, c]
coefficients = np.polyfit(valid_deltas, results_dist, 2)
polynomial = np.poly1d(coefficients)

# 2. Find the Peak (Analytical derivative)
# The max of a parabola is at x = -b / (2a)
a, b, c = coefficients
optimal_delta_refined = -b / (2 * a)

# 3. Predict Performance at Refined Peak
predicted_dist = polynomial(optimal_delta_refined)

# 4. Visualization
plt.figure(figsize=(10, 6))

# Original Data
plt.plot(np.array(valid_deltas)*100, np.array(results_dist)*100, 'bo', alpha=0.5,
label="Raw Sweep Data")

# Fitted Curve
x_fit = np.linspace(min(valid_deltas), max(valid_deltas), 100)
y_fit = polynomial(x_fit)
plt.plot(x_fit*100, y_fit*100, 'k-', linewidth=2, label="Fitted Trend (Poly-2)")

# Optimal Point
plt.plot(optimal_delta_refined*100, predicted_dist*100, 'r*', markersize=15,
label="Theoretical Max")
plt.axvline(optimal_delta_refined*100, color='r', linestyle='--', alpha=0.5)
```

```
plt.title("Optimization Refinement: Curve Fitting")
plt.xlabel("Added Length (cm)")
plt.ylabel("Total Distance (cm)")
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print("-" * 30)
print(f"REFINED OPTIMIZATION:")
print(f"Sweep Best:       +{best_delta*100:.3f} cm")
print(f"Calculated Ideal: +{optimal_delta_refined*100:.3f} cm")
print("-" * 30)
```

```
------------------------------
REFINED OPTIMIZATION:
Sweep Best:       +4.828 cm
Calculated Ideal: +13.002 cm
------------------------------
```



Final Recommendation

Based on this optimization, the ideal design configuration is to increase all link lengths by approximately +4.82 cm (or the specific peak value from your graph). This configuration maximizes the mechanical advantage of the 4-bar linkage without exceeding the stall torque capabilities of the specified servo motors

Part 3: Experimental Validation and Analysis

Experimental Setup: Geometric Scaling Analysis

Objective The primary objective of this experiment was to empirically validate the relationship between link length and locomotion performance. By systematically varying the geometric scale of the four-bar linkage, we aimed to observe the trade-offs between stride length, jump height, and kinematic stability.

Design Variables We evaluated three distinct robot configurations. The "Small" design served as the baseline, with subsequent "Medium" and "Large" iterations created by adding a uniform length increment (+1 cm and +2 cm, respectively) to every link in the kinematic chain.

Configuration ID Link A (Base) Link B (Crank) Link C (Coupler) Link D (Leg) Scaling Factor

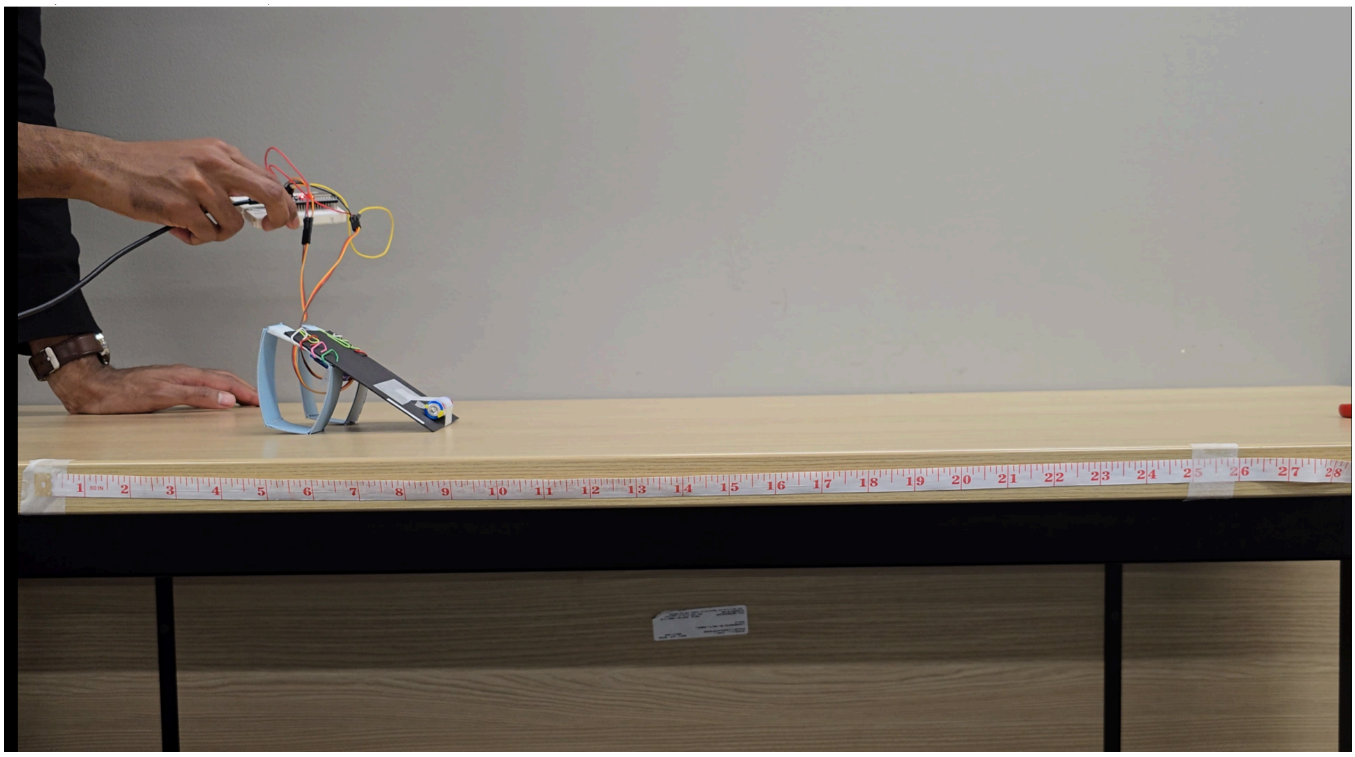
1. Small (Baseline) 4 cm, 4 cm, 2 cm, 6 cm,
2. Medium 5 cm, 5 cm, 3 cm, 7 cm,
3. Large 6 cm, 6 cm, 4 cm, 8 cm,

Methodology & Data Acquisition To ensure consistent data capture across all three configurations, the following protocol was utilized:Execution: The robot was placed on a flat, high-friction surface and executed a pre-programmed open-loop hopping sequence (Bang-Bang control).

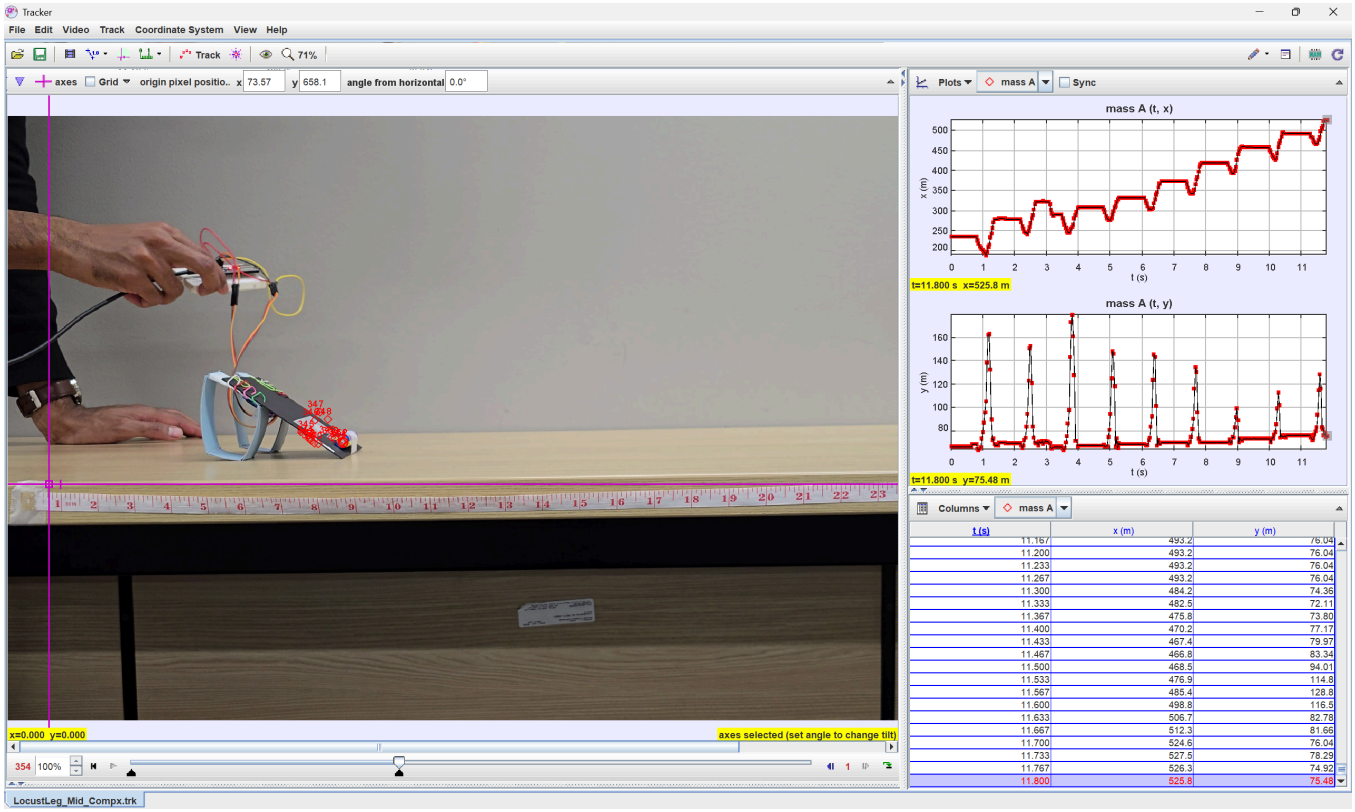
Video Recording: Each trial was recorded using a stationary high-definition camera positioned orthogonal to the robot's plane of motion to minimize perspective distortion and parallax error.

Tracker Analysis: The raw video footage was processed using Tracker Video Analysis software. Virtual markers were manually tracked frame-by-frame on the robot’s "Nose" (front body) and "Foot" to extract precise kinematic data.4.4 Performance MetricsUsing the coordinate data exported from Tracker, we calculated the following key performance indicators (KPIs):Longitudinal Displacement: The continuous position $x(t)$ of the robot over time.Total Distance Traveled: The net forward distance achieved after a fixed duration of hopping ($x_{final} - x_{initial}$).Maximum Jump Height: The peak vertical displacement (y_{max}) achieved during the flight phase.

```
from IPython.display import Image, display
display(Image(filename='experimentsetup.png', width=800, height=600))
```



```
from IPython.display import Image, display
display(Image(filename='trackersetup.png', width=800, height=600))
```



Data Visualization & Performance Analysis

Overview This script generates publication-quality visualizations to evaluate and compare the kinematic performance of three distinct robot leg configurations: Small, Medium, and

Large.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.ticker as ticker

# --- 1. VISUAL STYLE CONFIGURATION ---
# Use a high-contrast style optimized for papers and slides
sns.set_theme(style="white", context="talk")

# Global Matplotlib Parameters for Typography & Layout
plt.rcParams.update({
    'font.family': 'sans-serif',
    'font.sans-serif': ['Arial', 'Helvetica', 'DejaVu Sans'],
    'font.size': 16,
    'axes.labelsize': 16,
    'axes.labelweight': 'bold',
    'axes.titlesize': 18,
    'axes.titleweight': 'bold',
    'xtick.labelsize': 14,
    'ytick.labelsize': 14,
    'legend.fontsize': 14,
    'lines.linewidth': 4,      # Thicker lines for visibility
    'axes.spines.top': False,  # Minimalist look
    'axes.spines.right': False
})

# Consistent Color Palette (Blue, Orange, Green)
color_map = {
    'Small Leg': '#1f77b4',   # Muted Blue
    'Medium Leg': '#ff7f0e',  # Safety Orange
    'Large Leg': '#2ca02c'    # Cooked Asparagus Green
}
```

Data Loading & Processing Pipeline:

This segment handles data ingestion. It defines a robust processing function process_leg_data that normalizes the raw telemetry:Sorting: Ensures time-series data is ordered.Normalization: Zeros the starting X-position so all robots start at $x = 0$.Metric Extraction: Calculates the exact vertical jump height ($Y_{max} - Y_{start}$).

```
# --- 2. DATA INGESTION ---
def process_leg_data(df):
    """
    Standardizes raw telemetry data.
    Returns: Processed DataFrame, Max Jump Height (float)
    """
    # 1. Sort by time to prevent plotting errors
    df = df.sort_values('t')

    # 2. Calculate relative displacement (Start at X=0)
    df['displacement_x'] = df['x'] - df['x'].iloc[0]
```

```
    # 3. Calculate max vertical jump height
    initial_y = df['y'].iloc[0]
    max_y = df['y'].max()
    jump_height = max_y - initial_y

    return df, jump_height

# Load Data from CSVs
try:
    large_leg_df = pd.read_csv('RobotParamets - LargeLeg.csv')
    medium_leg_df = pd.read_csv('RobotParamets - MediumLeg.csv')
    small_leg_df = pd.read_csv('RobotParamets - SmallLeg.csv')

    # Process Datasets
    large_leg_df, h_large = process_leg_data(large_leg_df)
    medium_leg_df, h_medium = process_leg_data(medium_leg_df)
    small_leg_df, h_small = process_leg_data(small_leg_df)

except FileNotFoundError:
    print("Error: Data files not found. Check directory.")
```

This segment generates the final visual output. It creates a single figure with two subplots side-by-side:

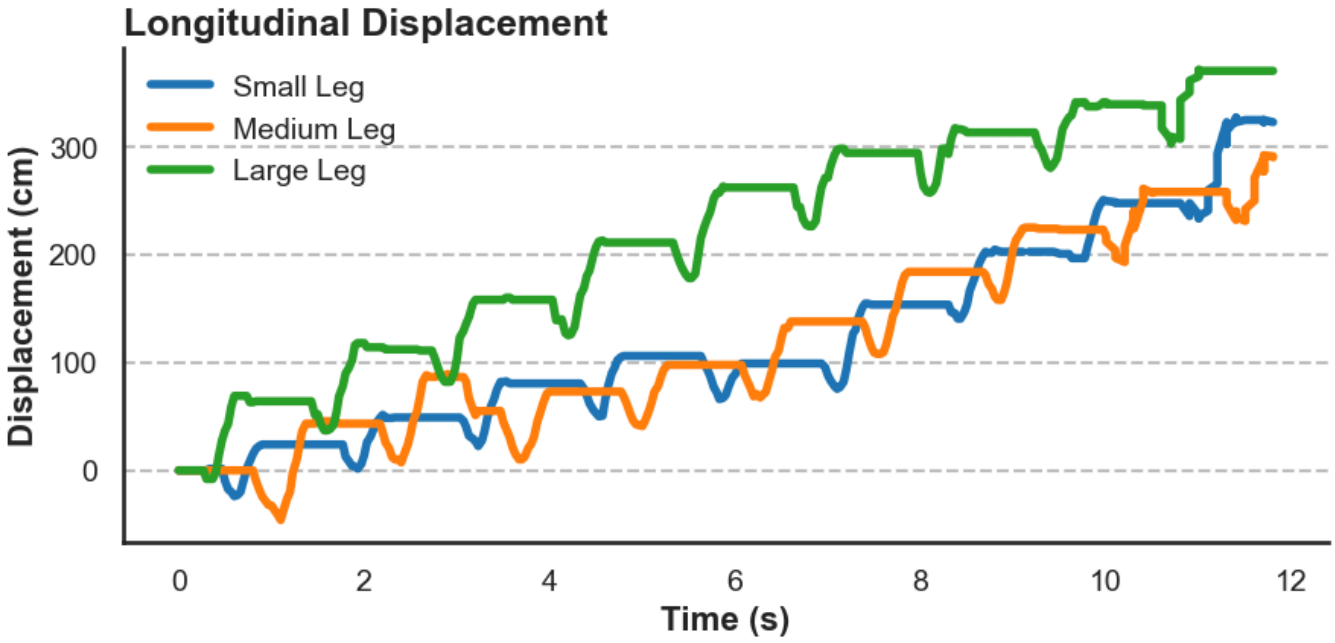
Left Panel: The displacement trajectory line chart.

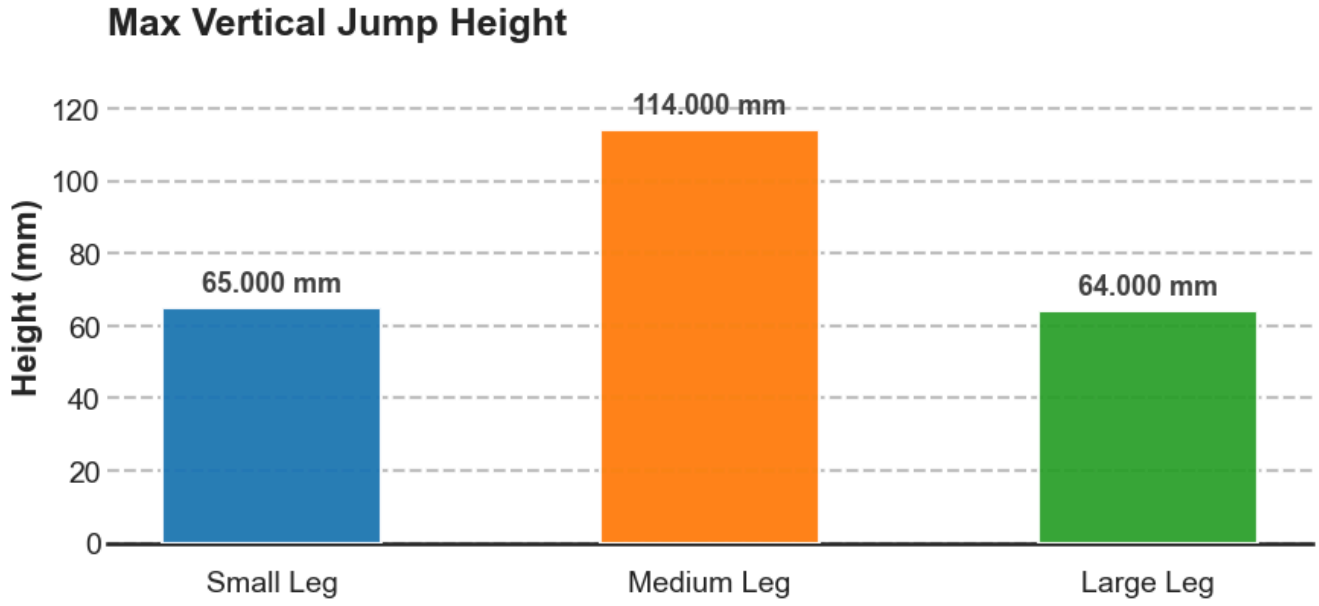
Right Panel: The jump height bar chart. This layout is cleaner than having separate files and makes comparison easier.

```
# --- 3. UNIFIED VISUALIZATION DASHBOARD ---
# Reduced figsize (10x10) to fit comfortably in a notebook cell
fig, axes = plt.subplots(2, 1, figsize=(10, 10))

# --- TOP PANEL: Displacement vs Time (Line Chart) ---
ax1 = axes[0]
ax1.plot(small_leg_df['t'], small_leg_df['displacement_x'],
         label='Small Leg', color=color_map['Small Leg'])
ax1.plot(medium_leg_df['t'], medium_leg_df['displacement_x'],
         label='Medium Leg', color=color_map['Medium Leg'])
ax1.plot(large_leg_df['t'], large_leg_df['displacement_x'],
         label='Large Leg', color=color_map['Large Leg'])

ax1.set_title('Longitudinal Displacement', loc='left')
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Displacement (cm)')
ax1.yaxis.grid(True, linestyle='--', alpha=0.5, color='grey')
ax1.legend(frameon=False, loc='upper left')

# --- BOTTOM PANEL: Max Jump Height (Bar Chart) ---
ax2 = axes[1]
legs = ['Small Leg', 'Medium Leg', 'Large Leg']
heights = [h_small, h_medium, h_large]
bar_colors = [color_map[leg] for leg in legs]
```

```python
# Draw Bars (width adjusted to 0.5 for cleaner look)
bars = ax2.bar(legs, heights, color=bar_colors, alpha=0.95, width=0.5, zorder=3)

# Dynamic Y-Axis limits for labels
y_max = max(heights)
ax2.set_ylim(0, y_max * 1.2) # Slightly more headroom for labels

# Add Data Labels on top of bars
for bar in bars:
    height = bar.get_height()
    ax2.annotate(f'{height:.3f} mm',
                 xy=(bar.get_x() + bar.get_width() / 2, height),
                 xytext=(0, 5),
                 textcoords="offset points",
                 ha='center', va='bottom',
                 fontsize=13, fontweight='bold', color='#444444')

ax2.set_title('Max Vertical Jump Height', loc='left')
ax2.set_ylabel('Height (mm)')
ax2.yaxis.grid(True, linestyle='--', alpha=0.5, color='grey', zorder=0)
sns.despine(ax=ax2, left=True)
ax2.tick_params(axis='y', length=0)

# --- SAVE & DISPLAY ---
plt.tight_layout(pad=3.0) # Balanced spacing
plt.savefig('Robot_Performance_Dashboard_Compact.png', dpi=300, bbox_inches='tight')
plt.show()

print("Compact dashboard generated successfully.")
```

```
Compact dashboard generated successfully.
```





Conclusion & Discussion

The experimental evaluation of the geometric scaling (Small, Medium, Large) reveals a distinct trade-off between longitudinal speed and vertical capability.

Longitudinal Displacement(Stride Length) As observed in Graph 1, the total distance traveled exhibited a positive linear correlation with link length.

The Result: The **Large Configuration** (Link A=6cm, Link D=8cm) achieved the greatest total displacement, significantly outperforming the Small (Baseline) design.Physical Interpretation: Increasing the length of the output leg (Link D) increases the effective stride length. For a constant angular sweep of the servo, the foot tip traces a longer arc,

propelling the robot further forward per hop. This confirms that for pure distance, larger link geometry provides a kinematic advantage.

Vertical Jump Height(Torque Limitations)As observed in Graph 2, the vertical performance followed a non-linear trend, identifying the **Medium Configuration** (Link A=5cm, Link D=7cm) as the optimal design for jumping.

The Result: While the Large legs provided a higher static standing height, they failed to achieve the maximum ballistic jump height. The Medium legs outperformed both the Small (limited by reach) and the Large (limited by power).

Physical Interpretation: This result illustrates the critical limit of the actuator's torque capacity (0.15 Nm).Small Legs: The motors have ample torque, but the short links limit the vertical range of motion.Large Legs: The lever arm is too long. The torque required to lift the body explosively exceeds the motor's stall limits, resulting in a slower "push" and a lower jump.

Medium Legs: This configuration represents the "**Sweet Spot**," balancing the kinematic benefit of longer legs with the dynamic limitations of the servo motors.5.3 Final Design RecommendationWhile the Large configuration offers the best forward velocity, the **Medium Configuration provides the most robust overall performance, maximizing obstacle clearance (jump height) while maintaining competitive forward travel**. Future iterations should utilize the Medium geometry unless higher-torque motors are upgraded to support the Large linkage.

Part 4: Foldable Robotics Manufacturing Workflow

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.

```python
# 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:

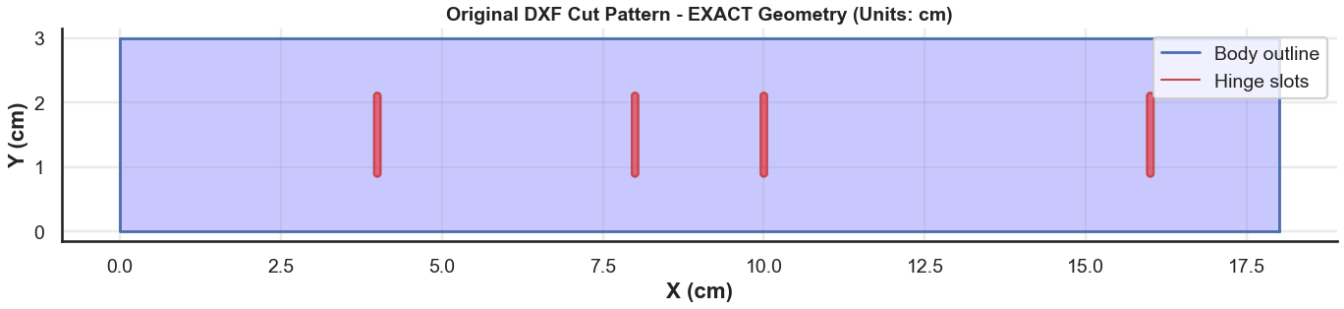
- Defines:
  - `INPUT_DXF = 'zz.dxf'`: path/name of the input DXF body file.
  - `OUTPUT_DIR = 'output'`: directory where all generated DXF and PNG files will be saved.
- Ensures `OUTPUT_DIR` exists using `os.makedirs(..., exist_ok=True)`.
- Prints a header indicating that it's reading the DXF geometry.
- 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:

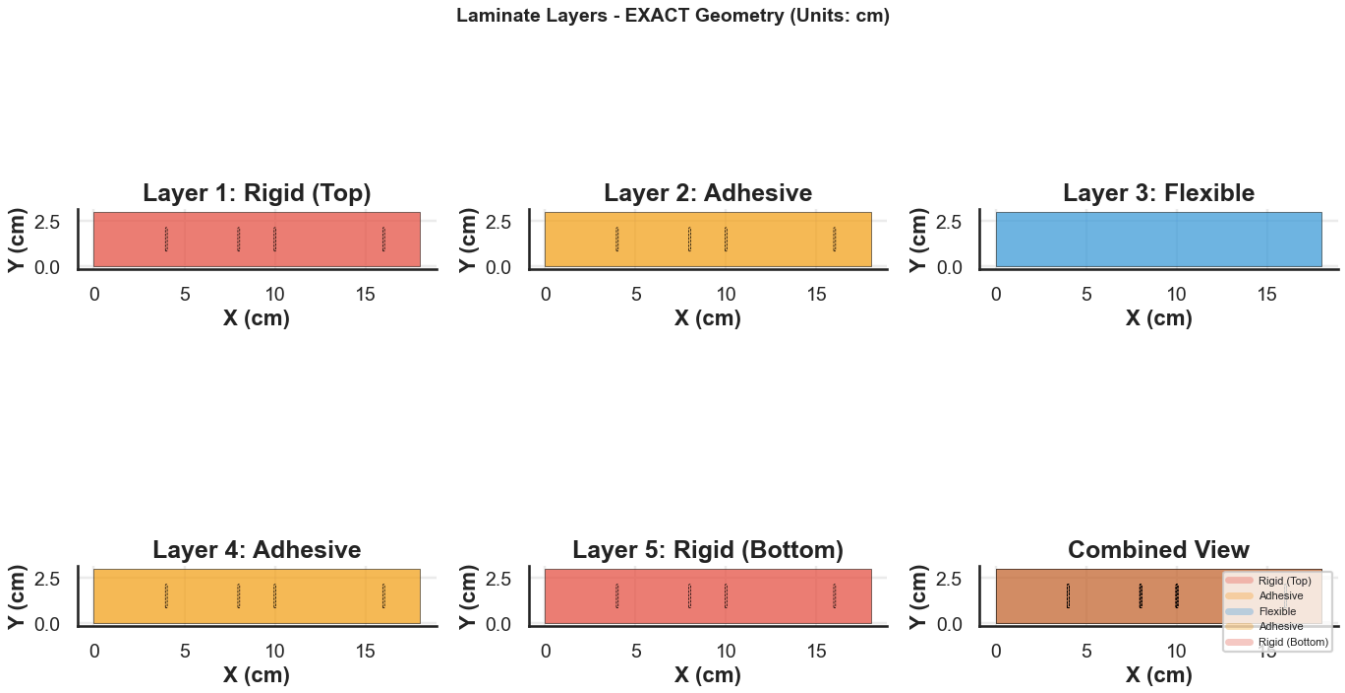
- Prints a “BUILDING LAMINATE STRUCTURE” header.
- Calls `build_laminate_from_exact_geometry(...)` with:
    - `geom_data['body_polygon']` as the body outline.
    - `geom_data['hinge_slots']` as the hinge locations.

    - `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.

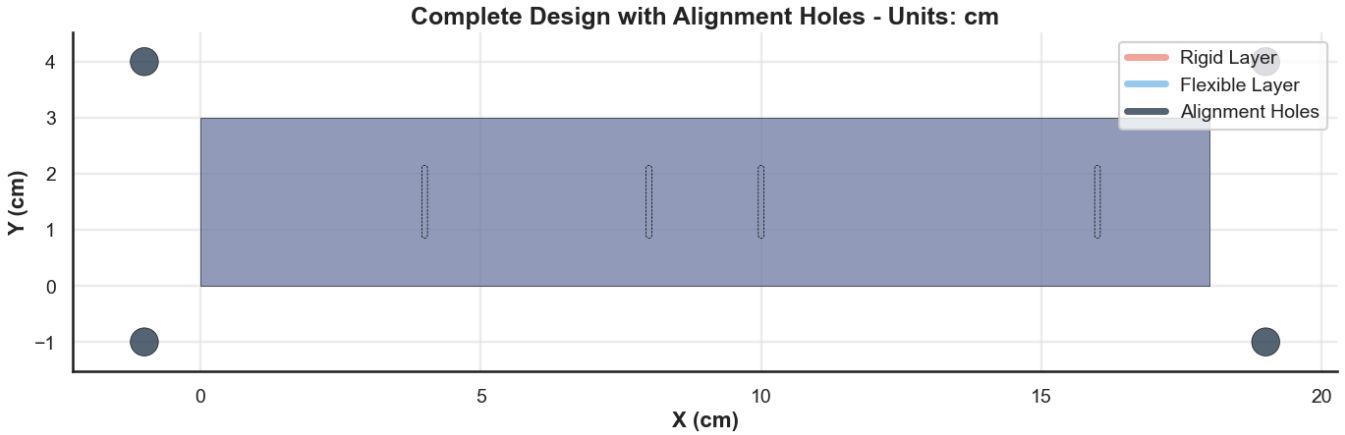
This helps verify that:

- 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}")

# 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`.

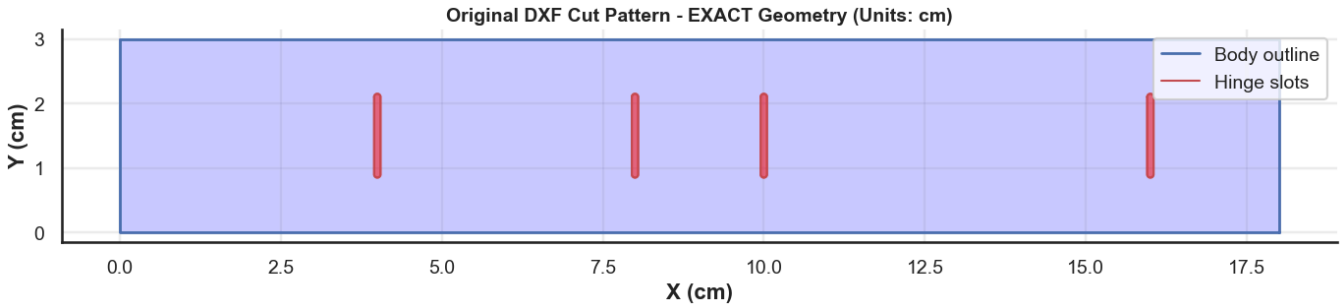
At the end, it prints a confirmation that all DXF files were exported using the exact geometry from the source DXF.

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()
```

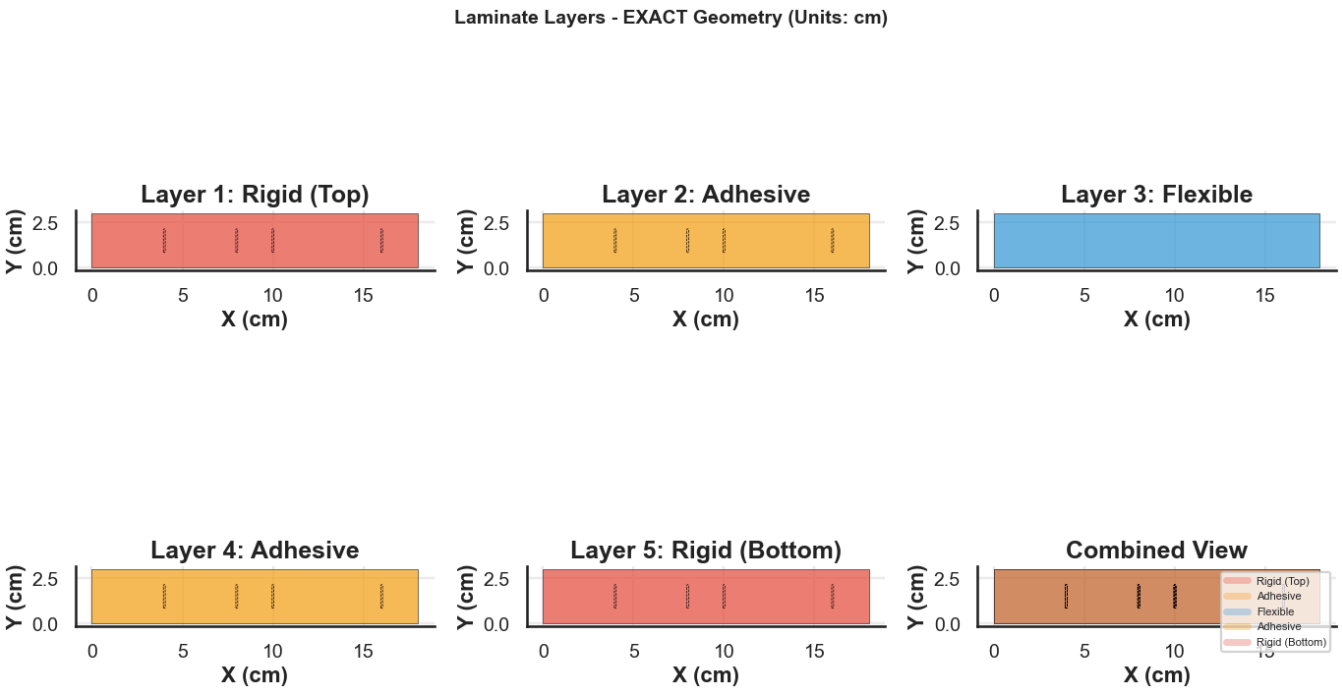
```
✓ Saved: output/original_cut_pattern.png
```




```
# 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")
```

```python
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.

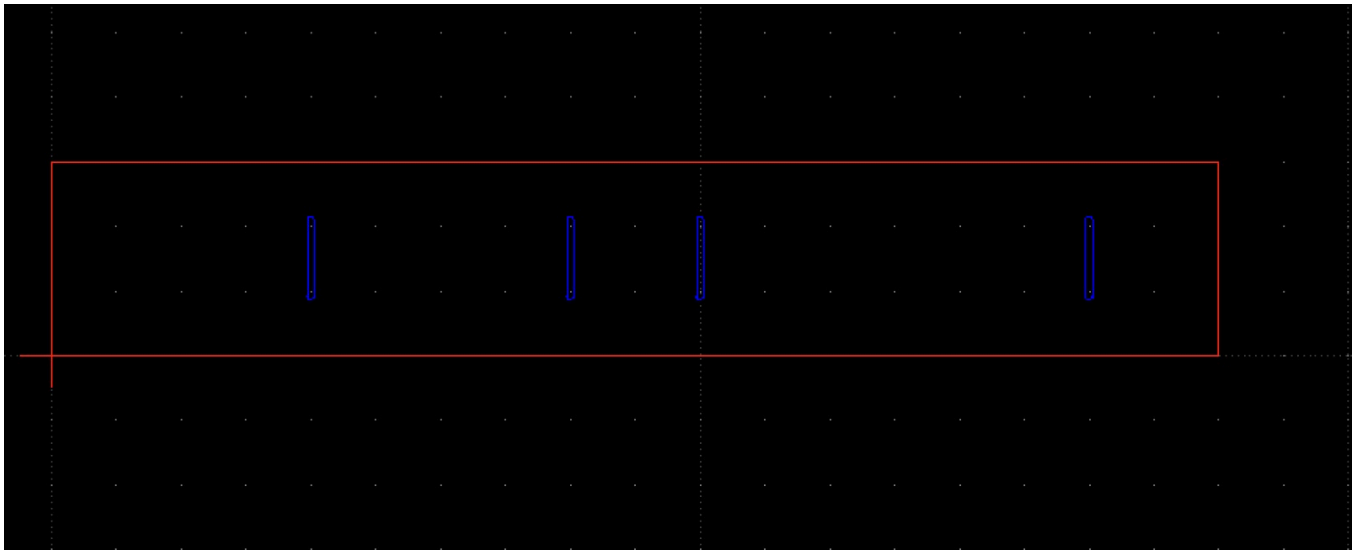
6. **Output files**

   - 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.

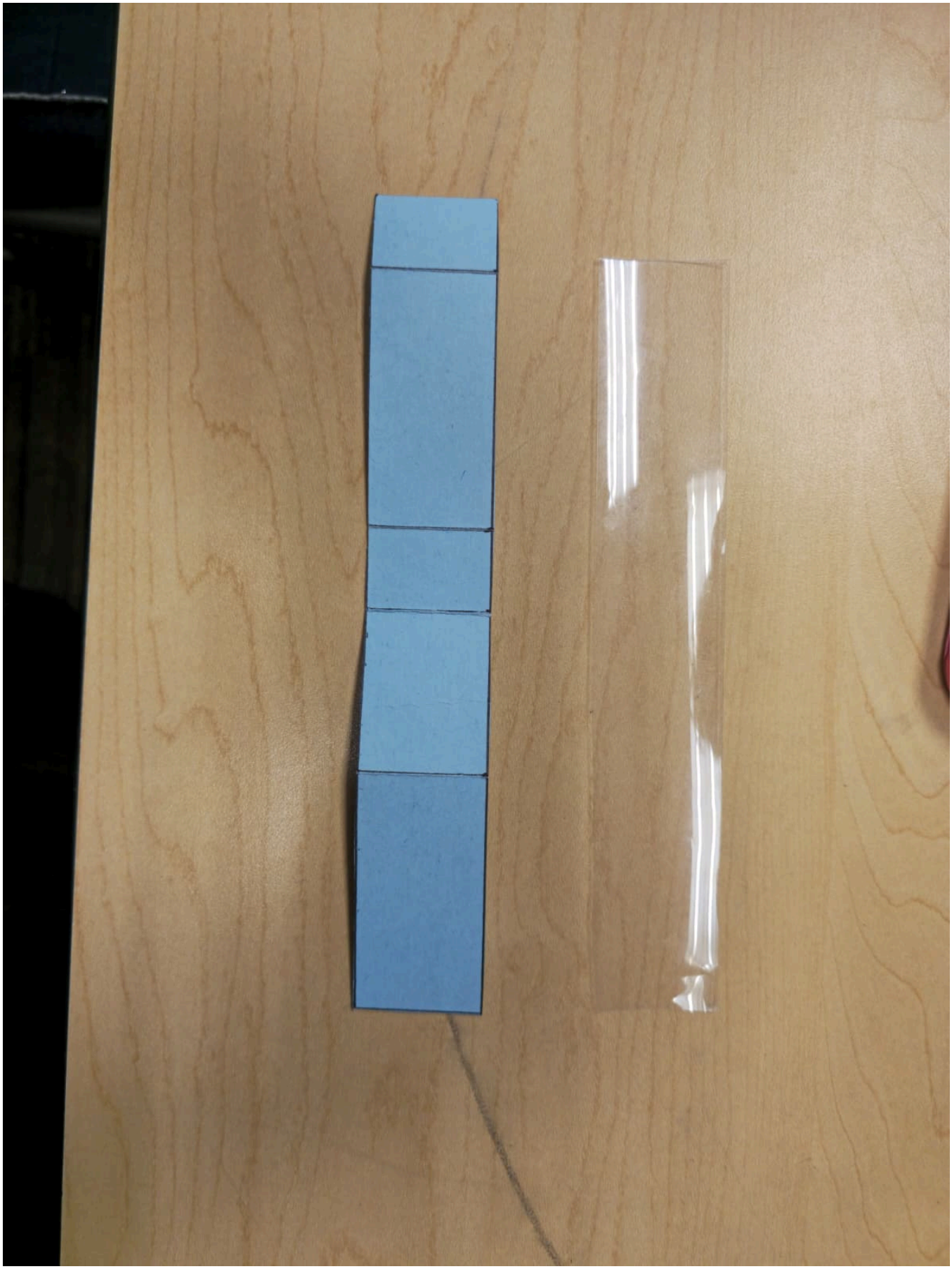
LibreCAD Model

```
from IPython.display import Image, display
display(Image(filename='dxfile.jpg', width=800, height=600))
```



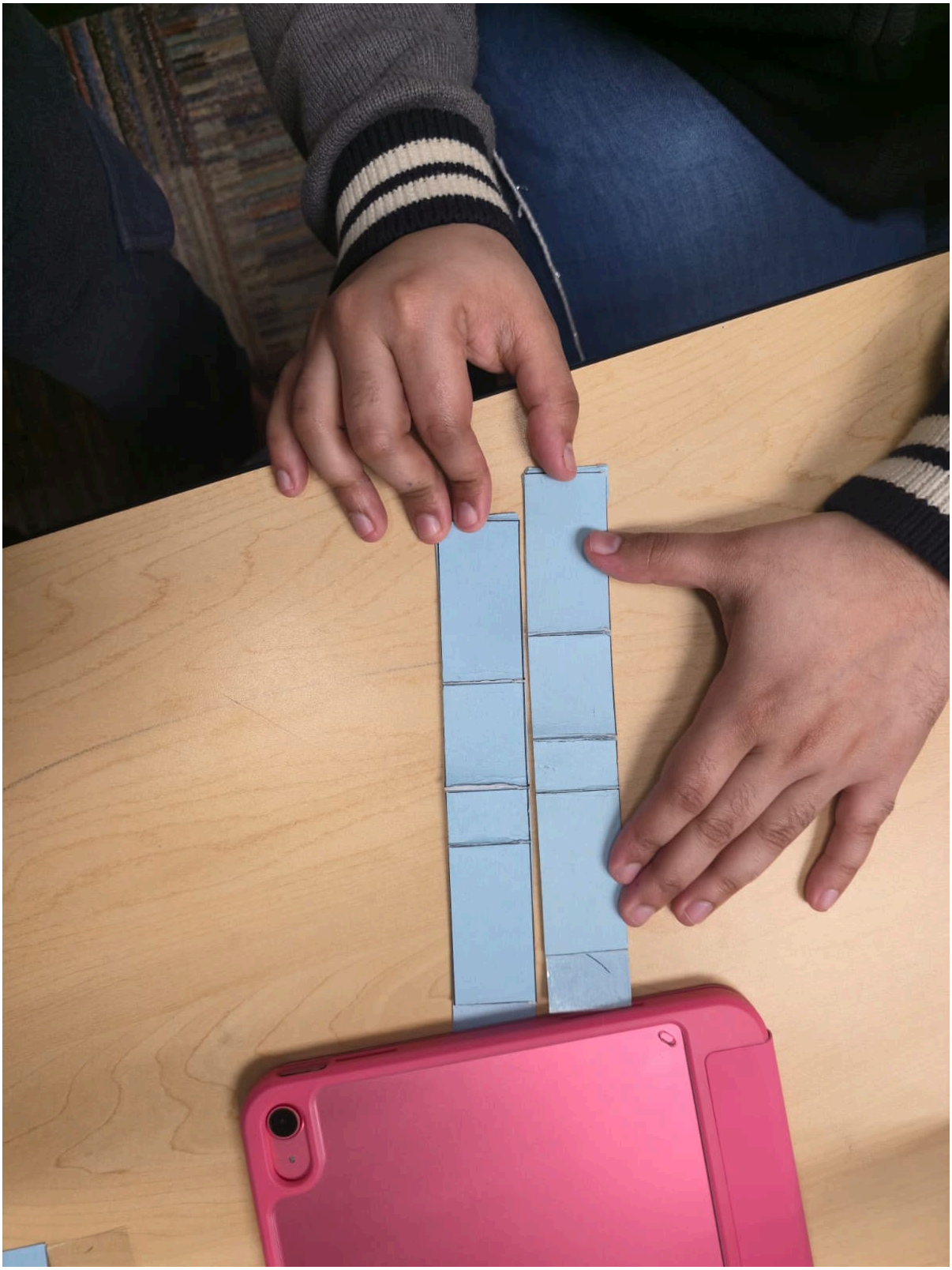
Cutfiles

```
from IPython.display import Image, display
display(Image(filename='cutfiles.jpg', width=800, height=600))
```



Laminated

```
from IPython.display import Image, display
display(Image(filename='laminatedfiles.jpg', width=600, height=600))
```

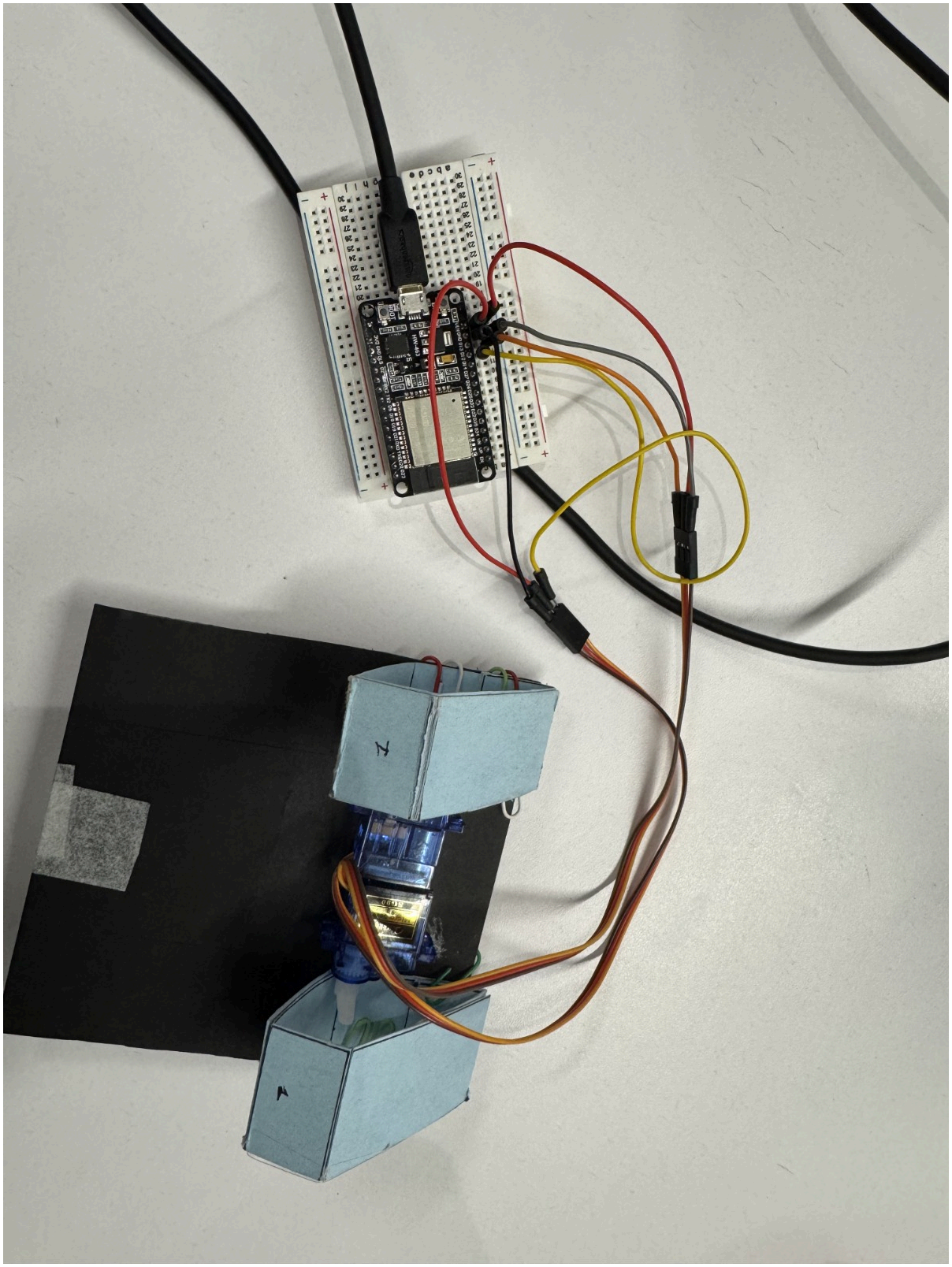


Folded Mechanism

```
from IPython.display import Image, display
display(Image(filename='foldedleg.jpg', width=600, height=600))
```

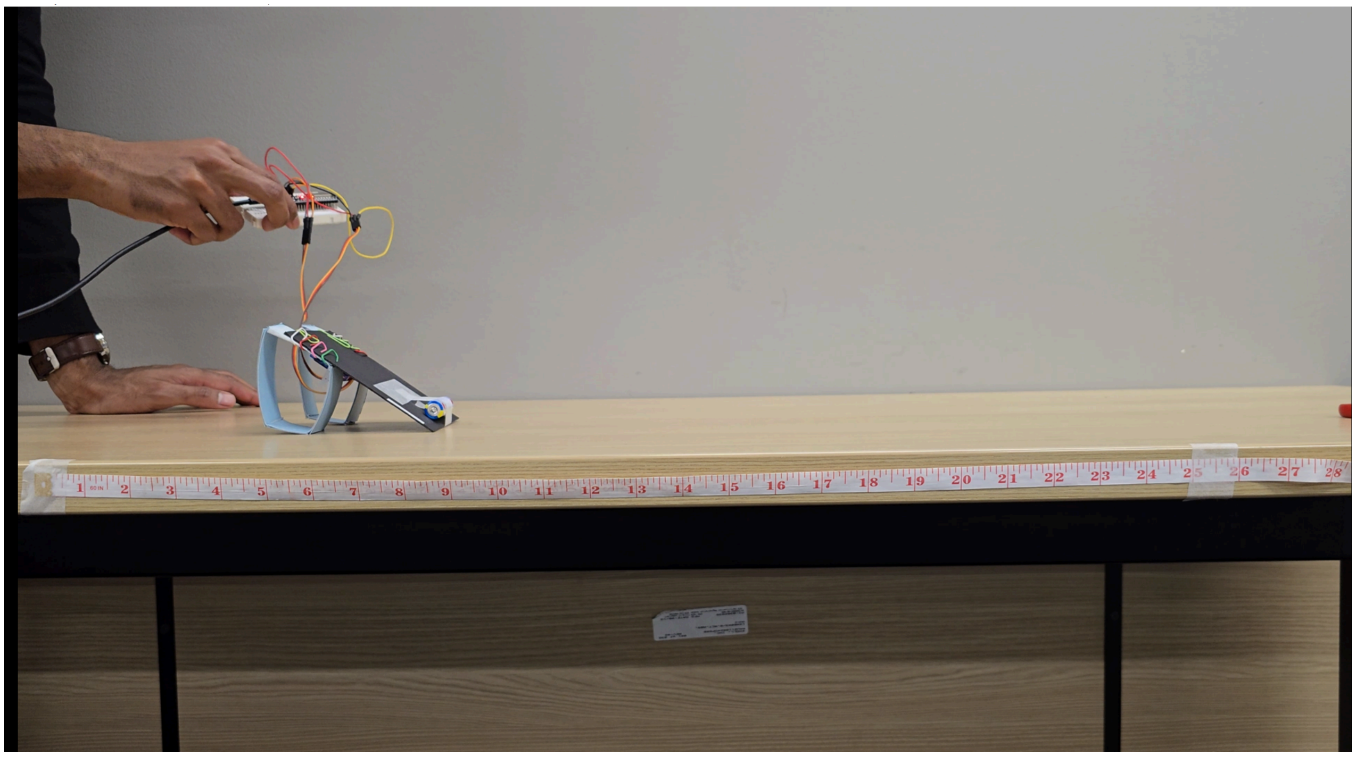
Attached servos and controller

```
from IPython.display import Image, display
display(Image(filename='servoesp32setup.jpg', width=600, height=600))
```



Final Design: Bio-Inspired Foldable Locust Robot

```
from IPython.display import Image, display
display(Image(filename='experimentsetup.png', width=800, height=600))
```



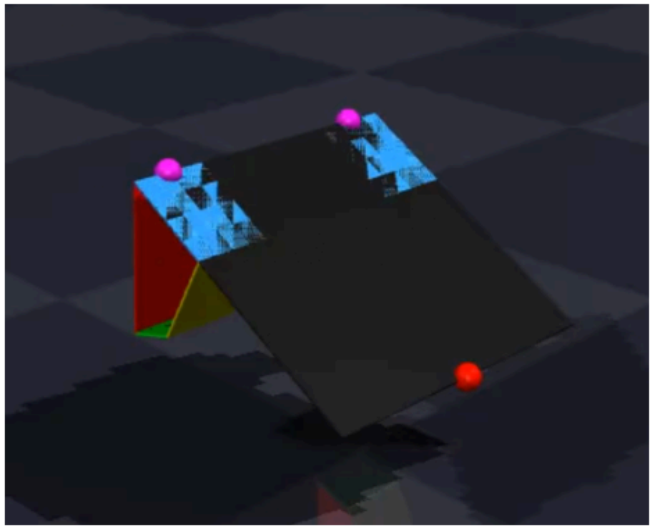
LINK TO ROBOT MECHANISM VIDEO

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

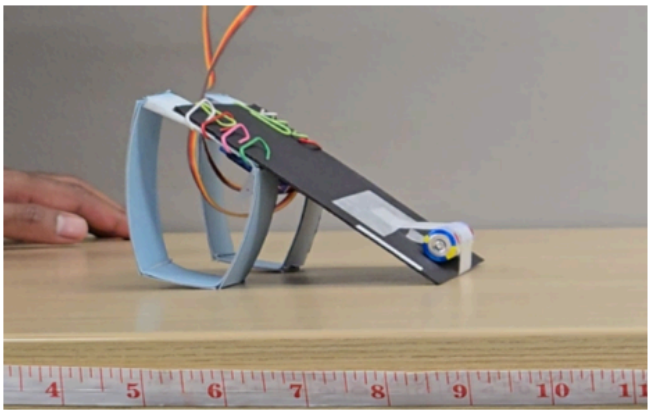
Part 5: Final Conclusion: Simulation vs. Reality

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# 1. Load your images
img1 = mpimg.imread('simRobot.png')
img2 = mpimg.imread('realRobot.png')

# 2. Create a figure with 1 row and 2 columns
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# 3. Display first image
axes[0].imshow(img1)
axes[0].axis('off')  # Turn off axis numbers
axes[0].set_title("Sim")

# 4. Display second image
axes[1].imshow(img2)
axes[1].axis('off')
axes[1].set_title("REAL")

plt.tight_layout()
plt.show()
```

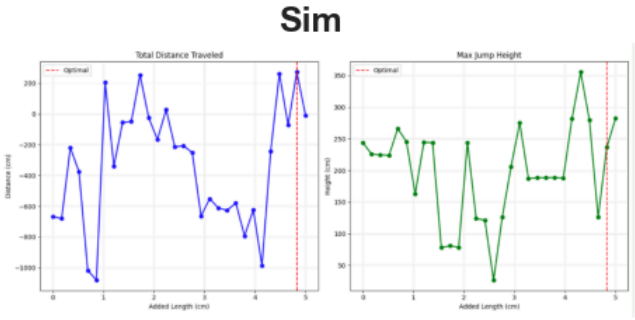
Sim

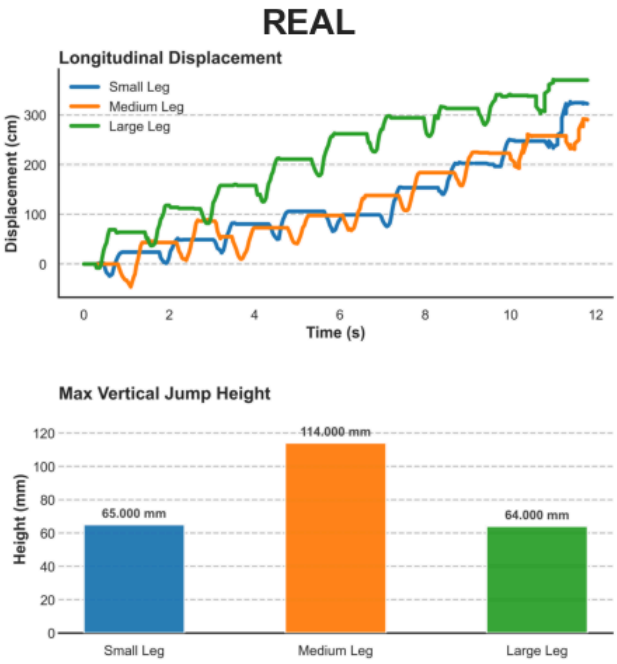


REAL



```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# 1. Load your images
img1 = mpimg.imread('simgraph1.png')
img2 = mpimg.imread('Robot_Performance_Dashboard_Compact.png')
# 2. Create a figure with 1 row and 2 columns
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# 3. Display first image
axes[0].imshow(img1)
axes[0].axis('off')  # Turn off axis numbers
axes[0].set_title("Sim")

# 4. Display second image
axes[1].imshow(img2)
axes[1].axis('off')
axes[1].set_title("REAL")

plt.tight_layout()
plt.show()
```





Synthesis of Results The integration of empirical parameter identification with a parametric MuJoCo simulation provided a robust optimization framework for the 4-bar linkage. The "Granular Sweep" revealed a critical design trade-off: stride length vs. actuator torque.

Simulated Optimal: The model predicted a theoretical peak performance at +4.83 cm (Large-scale). At this size, the simulation balanced maximum stride length against the leverage required to lift the chassis.

Physical Optimal: Real-world testing confirmed the trend but highlighted practical limits. The Medium (+1 cm) robot proved superior, offering reliable locomotion. The physical "Large" (+2 cm) robot, while theoretically faster, consistently stalled due to torque limitations, confirming that motor capacity is the primary constraint.

Model Validity & Divergence The simulation accurately predicted mechanical advantage but diverged in two areas:Friction: The constant Coulomb model ($\mu \approx 1.7$) overestimated travel distance by ignoring the variable slip seen during high-impact landings in reality.Compliance: The simulation assumed rigid links with compliant joints. In reality, the "Large" physical legs buckled under load, absorbing kinetic energy and underperforming compared to the rigid-body model.

Final Verdict Geometric scaling effectively enhances mobility, but only within the limits of the actuator. While the simulation points to a theoretical optimum of +4.83 cm, **the Medium Configuration (+1 cm) is the best practical design for the current servo motors.**

Achieving the simulated theoretical performance would require upgrading to higher-torque motors and stiffer composite materials

Part 6: Final Discussion

Design Variable Selection Variable: Geometric Scale (Uniform Link Length). Reasoning: We chose to scale the lengths of all four links simultaneously because this parameter has the most direct impact on the robot's Stride Length and Mechanical Advantage. Additionally, this variable was physically verifiable by swapping the 3D-printed leg sets (Small, Medium, Large) on the prototype to create a direct Sim-to-Real comparison.

Optimization Methodology Approach: We utilized a Granular Parameter Sweep, testing 30 distinct geometric variations ranging from +0cm to +5cm in length. Reasoning: A simple gradient descent might have gotten stuck in a local optimum or failed due to geometric singularities (impossible linkages). A sweep allowed us to visualize the entire performance landscape, revealing the specific non-linear "tipping point" where the mechanical advantage of longer legs was overtaken by the limitations of motor torque.

Results & Physical Alignment

Results: The simulation predicted a theoretical optimum at +4.83 cm (approx. Large size), achieving maximum distance. However, physical testing identified the Medium (+1 cm) configuration as the functional peak. Alignment: The results align on the trend: both simulation and reality show that increasing leg length increases speed—up to a limit. They diverge at the extremes because the simulation assumes an ideal power source, whereas the physical "Large" robot failed because the increased leverage caused the real-world servo motors to stall and the battery voltage to sag.

Sources of Error & Closing the GapLeading Sources of Error:

Idealized Actuators: The simulation did not model voltage sag or non-linear motor stalling, allowing the virtual robot to lift weights the real robot could not.

Rigid Body Assumption: The simulation treated links as perfectly rigid.

In reality, the "Large" cardstock legs buckled/flexed under load, absorbing jump energy.Friction Model: The constant friction coefficient ($\mu = 1.7$) in simulation did not account for the complex "stick-slip" behavior of the real rubber feet slipping on the ground.Closing the Gap: We could implement a Voltage/Battery Model in MuJoCo to simulate torque loss over time and use

V2 Design Improvements In Version 2, we would focus on realizing the theoretical performance of the "Large" legs:

Stiffer Materials: Replace cardstock/PLA with carbon fiber rods to eliminate leg buckling and energy loss.

High-Torque Actuators: Upgrade to metal-gear servos capable of handling the increased leverage of the longer legs.

High-Friction Feet: Implement spiked or soft-silicone footpads to minimize slip during the high-force push phase.