(ASSIGNMENT 5)Friction Parameter Identification – MuJoCo Simulation & Analysis

This notebook is a **breakdown** of a MuJoCo-based simulation and friction parameter identification of a hexagonal closed-loop mechanism sliding down an inclined ramp. It:

- Installs and imports all required dependencies.
- Defines global physical and geometric parameters.
- Implements helper utilities for kinematics and contact inspection.
- Builds the full simulation engine (`run_sim`).
- Defines a `main()` function that runs the simulation and generates **many Plotly plots**.
- Outputs **each major group of plots in its own cell**, with an explanation cell before it.

Goal of the Experiment (Friction Identification)

The goal of this experiment is to identify both the static and dynamic friction coefficients between the cardboard hexagonal closed-loop mechanism and the surface of the test ramp. This friction parameter is a critical input to our team's dynamic model, as it directly affects the mechanism's ability to initiate motion, sustain sliding, and interact with the environment.

Specifically, this experiment aims to:

1. Measure the critical angle at which the mechanism transitions from rest to sliding and use this to compute the static friction coefficient.
2. Record the motion of the mechanism during sliding and extract position–time data to estimate the dynamic friction coefficient μd by fitting a simple dynamic model to the measured trajectory.
3. Validate the identified friction parameters by running a high-fidelity MuJoCo simulation of the hexagonal closed-loop mechanism sliding down the same inclined plane, and comparing its predicted behavior to both the theoretical model and the experimental motion.
4. Characterize how friction influences contact forces, sliding behavior, and stability, and evaluate friction-model consistency using simulation metrics such as: effective friction $\mu \mathrm{eff}(t)$, contact force distributions, friction cone plots, center-of-pressure migration, sensitivity to contact settings (solref, solimp).

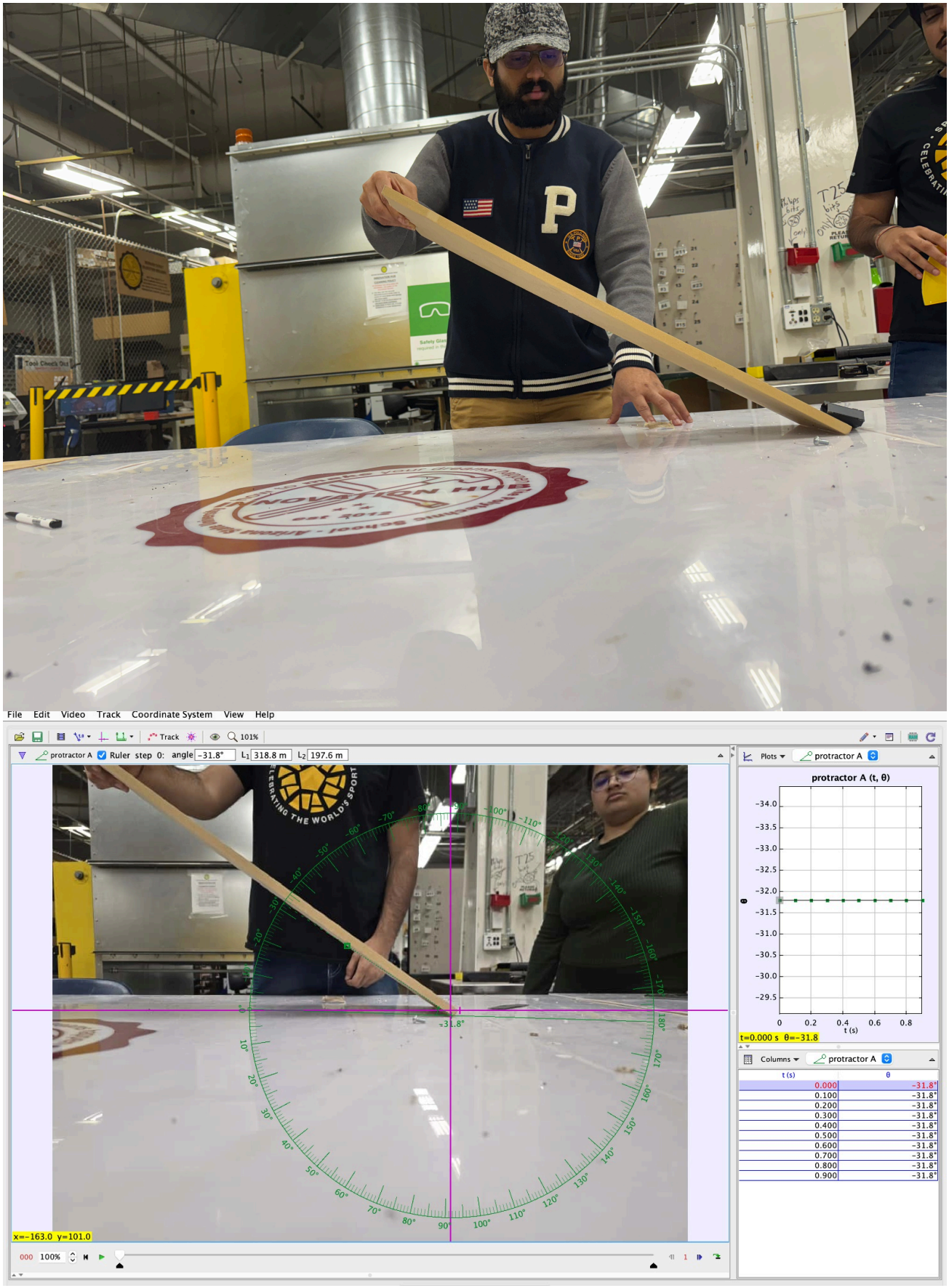
Ultimately, the purpose of this experiment is to obtain accurate, experimentally validated friction parameters that can be reliably incorporated into our team's system-level dynamic

model, ensuring realistic predictions of ground–mechanism interaction for subsequent analysis and optimization.

Setup

I was working on the parameter identification assignment for this model, focusing specifically on determining the friction between the model material and the foot-floor interaction. For this experiment, I inclined a piece of cardboard on a white table surface, as depicted in the screenshot. Using a protractor, I measured the angles and found the critical angle to be 31.8 degrees, resulting in a calculated static friction of 0.62. To further validate my measurements, I utilized the Tracker app for clarification.

EXPERIMENT IMAGES



1. Environment Setup and Dependencies

This cell installs all necessary third-party libraries used:

- **mujoco** – physics engine.
- **mediapy** – for displaying/saving video frames.
- **plotly** – rich, interactive visualizations.
- **numpy** – numerical computations.

> 💡 Depending on your environment, some may already be installed.
> If MuJoCo installation fails (e.g., missing system libraries), the code will still **run structurally**,
> but the actual simulation will be skipped and only empty / placeholder plots may appear.

```
# Install required dependencies (uncomment and run if needed)
# Note: MuJoCo may require additional system-level libraries; see official docs if errors occur.

#!pip install mujoco mediapy plotly numpy
```

2. Imports and Optional Library Setup

This cell imports all required Python modules and sets up **feature flags** to handle environments where some packages (especially Plotly or MuJoCo) may be missing.

- If Plotly is unavailable → plots are skipped with a warning.
- If MuJoCo or Mediapy is unavailable → simulation is skipped, but the rest of the notebook still runs.

```
import math
import numpy as np
import sys
import time
import traceback

# Import visualization libraries, handle potential import errors
try:
    import plotly.graph_objects as go
    from plotly.subplots import make_subplots
    import plotly.colors as pc
    PLOTLY_AVAILABLE = True
except ImportError:
    PLOTLY_AVAILABLE = False
    go = None
    make_subplots = None
    pc = None
    print("Warning: Plotly not found. Visualizations will be skipped.")

# Import MuJoCo and Mediapy, handle potential import errors
try:
```

```
    import mujoco
    import mediapy as media
    MUJOCO_AVAILABLE = True
except ImportError:
    MUJOCO_AVAILABLE = False
    mujoco = None
    media = None
    print("Warning: MuJoCo or Mediapy not found. Simulation cannot run.")
```

3. Global Configuration and Geometry

In this cell we define **core physical and geometric parameters** of the system:

- Gravity, critical angle, and friction coefficients.
- Total mass and number of links in the hexagonal mechanism.
- Link dimensions and ramp position in the world.
- Simulation settings such as duration, FPS, integrator, timestep, and contact parameters.

These act as **global configuration constants** shared across the entire notebook.

```
# --- 1. Global Configuration & Geometry ---

G = 9.81                # gravity

# Measured critical angle (degrees)
THETA_DEG = 31.8
MU_STATIC = 0.62        # experimentally estimated static friction coefficient

# Mass of the hexagonal closed loop (kg)
TOTAL_MASS = 0.015      # 15 g
N_LINKS = 6

# Geometry
SEG_LEN = 0.1
HALF_LEN = SEG_LEN / 2.0
THICK = 0.002
RAMP_THICKNESS = 0.02
RAMP_WORLD_CENTER = np.array([0.0, 0.0, 0.5])

# Simulation parameters
INITIAL_SPEED = 1.0      # m/s along the ramp
DURATION = 4.0           # seconds (primary demo)
SWEEP_DURATION = 2.5     # seconds (shorter runs for parameter sweeps)
FPS = 60                 # High FPS for smooth video

DEFAULT_INTEGRATOR = "RK4"
DEFAULT_TIMESTEP = 0.001
DEFAULT_SOLREF = "0.001 1"   # default contact solref
DEFAULT_SOLIMP = "0.99 0.99 0.001" # default contact solimp (Hard/Sharp)
```

4. Helper Functions

This section defines reusable utilities that the simulation will use:

1. **`rotate_y(vec, angle_deg)`**
   Rotates a 3D vector about the global Y-axis. Used to place the mechanism on the inclined ramp.

2. **`quat_to_euler_wxyz(q)`**
   Converts a quaternion `(w, x, y, z)` to Euler angles *(roll, pitch, yaw)* for interpreting orientation.

3. **`print_contact_debug(contact, model)`**
   Prints detailed information about a MuJoCo contact (penetration, friction, solimp/solref, frame) to help debug and validate contact settings.

```python
# --- 2. Helper Functions ---

def rotate_y(vec, angle_deg):
    """Rotates a 3D vector around the global Y-axis by angle_deg."""
    th = math.radians(angle_deg)
    c, s = math.cos(th), math.sin(th)
    x, y, z = vec
    return np.array([x * c + z * s, y, -x * s + z * c])


def quat_to_euler_wxyz(q):
    """
    Convert quaternion (w, x, y, z) to roll-pitch-yaw (Tait-Bryan angles).
    Returns (roll, pitch, yaw) in radians.
    """
    w, x, y, z = q

    # roll (x-axis rotation)
    sinr_cosp = 2.0 * (w * x + y * z)
    cosr_cosp = 1.0 - 2.0 * (x * x + y * y)
    roll = math.atan2(sinr_cosp, cosr_cosp)

    # pitch (y-axis rotation)
    sinp = 2.0 * (w * y - z * x)
    if abs(sinp) >= 1:
        pitch = math.copysign(math.pi / 2.0, sinp)
    else:
        pitch = math.asin(sinp)

    # yaw (z-axis rotation)
    siny_cosp = 2.0 * (w * z + x * y)
    cosy_cosp = 1.0 - 2.0 * (y * y + z * z)
    yaw = math.atan2(siny_cosp, cosy_cosp)

    return roll, pitch, yaw
```

```
def print_contact_debug(contact, model):
    """
    Detailed inspection of the MjContact class.
    Prints internal solver parameters for a single contact to verify XML defaults.
    Based on PDF content [cite: 115-137].
    """
    print("\n" + "=" * 60)
    print("--- [DEBUG] MjContact Introspection ---")
    print(f"Dist (Penetration): {contact.dist:.6f} m")
    print(f"Geom IDs: {contact.geom1} <--> {contact.geom2}")
    print(f"Dimension: {contact.dim}")

    # Friction array: [Tangential1, Tangential2, Torsional, Rolling1, Rolling2]
    print(f"Friction (5D array): {contact.friction}")

    # Solver Parameters
    print(f"SolImp (Constraint Impedance): {contact.solimp}")
    print(f"SolRef (Constraint Reference): {contact.solref}")

    # Contact Frame (Normal is the first 3 numbers)
    normal = contact.frame[0:3]
    print(f"Contact Normal (World): {normal}")
    print("=" * 60 + "\n")
```

5. Core Simulation Engine – `run_sim(...)`

The `run_sim` function:

- Builds the MuJoCo XML model of the ramp and **6-link hexagonal loop**.
- Configures contact properties using `solref` and `solimp` (normal stiffness/damping and constraint impedance).
- Runs the simulation with chosen integrator and timestep.
- Logs:
  - Center of mass position, velocity, and orientation.
  - Contact forces and torques (6D wrench).
  - Center of pressure (CoP) along the ramp.
  - Penetration depths, contact counts, and world-frame net wrench.
- Optionally records video frames for visualization.

The returned dictionary contains everything needed for plotting and deeper analysis.

```
def run_sim(theta_deg=THETA_DEG,
            mu_static=MU_STATIC,
            total_mass=TOTAL_MASS,
            n_links=N_LINKS,
            duration=DURATION,
            fps=FPS,
            initial_speed=INITIAL_SPEED,
```

```
            integrator=DEFAULT_INTEGRATOR,
            timestep=DEFAULT_TIMESTEP,
            enable_contact_debug=True,
            solref_override=DEFAULT_SOLREF,
            solimp_override=DEFAULT_SOLIMP, # <--- Added solimp override argument
            record_video=True,
            render_width=1920,
            render_height=1080):
    """
    Build the MuJoCo model, run the simulation, and log data for analysis.
    """
    if not MUJOCO_AVAILABLE:
        return {}

    theta_rad = math.radians(theta_deg)

    # Simple Coulomb model: for constant-velocity sliding, mu ≈ tan(theta)
    mu_dynamic = math.tan(theta_rad)
    mu_slide = mu_dynamic
    mass_per_link = total_mass / n_links

    # Compute initial world position for the hexagon body origin (link1)
    ramp_world_center = RAMP_WORLD_CENTER
    mech_local_start = np.array([-1.0, 0.0, RAMP_THICKNESS + THICK])  # near top of
ramp
    mech_world_start = ramp_world_center + rotate_y(mech_local_start, theta_deg)
    l1_pos_str = f"{mech_world_start[0]:.4f} {mech_world_start[1]:.4f}
{mech_world_start[2]:.4f}"

    # MuJoCo XML model definition
    # - solimp configurable via solimp_override [cite: 11]
    xml = f"""
<mujoco model="Hexagon Mechanism Sliding Demo">
  <compiler angle="degree" coordinate="local"/>

  <option gravity="0 0 -9.81" timestep="{timestep}" integrator="{integrator}"
cone="elliptic"/>

  <visual>
    <quality shadowsize="4096"/>
    <headlight ambient="0.3 0.3 0.3" diffuse="0.4 0.4 0.4" specular="0.1 0.1 0.1"/>
    <map fogstart="3" fogend="10"/>
    <global offwidth="1920" offheight="1080"/>
  </visual>

  <asset>
    <texture type="skybox" builtin="gradient"
             rgb1="0.1 0.15 0.2" rgb2="0.6 0.8 1.0"
             width="512" height="512"/>
    <texture name="grid_tex" type="2d" builtin="checker"
             rgb1=".2 .25 .3" rgb2=".1 .15 .2"
             width="1024" height="1024" mark="edge" markrgb="0.8 0.8 0.8"/>
    <texture name="cardboard_tex" type="2d" builtin="flat"
             rgb1="0.8 0.7 0.5" width="512" height="512"/>

    <material name="grid_mat" texture="grid_tex" texrepeat="8 8"
```

```
            texuniform="true" reflectance="0.1" shininess="0.2"/>
  <material name="ramp_mat" rgba="0.2 0.2 0.25 1"
            specular="0.3" shininess="0.5" reflectance="0.1"/>
  <material name="cardboard_mat" texture="cardboard_tex"
            specular="0.1" shininess="0.1"/>
</asset>

<default>
  <default class="link">
    <geom type="box"
          size="{HALF_LEN:.4f} 0.04 {THICK:.4f}"
          pos="{HALF_LEN:.4f} 0 0"
          material="cardboard_mat"
          mass="{mass_per_link:.6f}"
          condim="6"
          solimp="{solimp_override}"
          solref="{solref_override}"
          margin="0.001"
          group="1"
          friction="{mu_slide:.4f} 0.005 0.0001"/>
  </default>
</default>

<worldbody>
  <geom name="floor" size="10 10 0.1" type="plane"
        material="grid_mat" condim="3"/>

  <light name="key_light" pos="0 -3 5" dir="0 0.5 -1"
         directional="true" castshadow="true"
         diffuse="0.9 0.9 0.9" specular="0.3 0.3 0.3"/>

  <light name="fill_light" pos="3 3 4" dir="-0.5 -0.5 -1"
         directional="true" castshadow="false"
         diffuse="0.3 0.3 0.4" specular="0.1 0.1 0.1"/>

  <light name="back_light" pos="-3 5 2" dir="0.5 -0.5 -0.5"
         directional="true" castshadow="false"
         diffuse="0.2 0.2 0.2"/>

  <camera name="cam_close_track" mode="targetbody"
          target="link1" pos="0.5 -0.8 0.6"/>
  <camera name="cam_fixed" mode="targetbody"
          target="ramp" pos="0 -2.5 1.5"/>

  <body name="ramp" pos="0 0 0.5" axisangle="0 1 0 {theta_deg}">
    <geom name="ramp_visual" type="box"
          size="1.5 0.5 {RAMP_THICKNESS}"
          material="ramp_mat"
          friction="{mu_slide:.4f} 0.005 0.0001"/>
  </body>

  <body name="link1" pos="{l1_pos_str}" axisangle="0 1 0 {theta_deg}">
    <freejoint/>
    <geom name="g1" class="link"/>

    <body name="link2" pos="{SEG_LEN} 0 0" axisangle="0 1 0 -60">
```

```
        <geom name="g2" class="link"/>

        <body name="link3" pos="{SEG_LEN} 0 0" axisangle="0 1 0 -60">
          <geom name="g3" class="link"/>

          <body name="link4" pos="{SEG_LEN} 0 0" axisangle="0 1 0 -60">
            <geom name="g4" class="link"/>

            <body name="link5" pos="{SEG_LEN} 0 0" axisangle="0 1 0 -60">
              <geom name="g5" class="link"/>

              <body name="link6" pos="{SEG_LEN} 0 0" axisangle="0 1 0 -60">
                <geom name="g6" class="link"/>
              </body>
            </body>
          </body>
        </body>
      </body>
    </body>

  </worldbody>

</mujoco>
"""

    # Initialize simulation
    try:
        model = mujoco.MjModel.from_xml_string(xml)
    except Exception as e:
        print(f"Error creating MuJoCo model from XML: {e}")
        return {}

    data = mujoco.MjData(model)

    # Renderer & visualization options (conditional)
    renderer = None
    scene_option = None
    if record_video:
        renderer = mujoco.Renderer(model, width=render_width, height=render_height)

        # Native Contact Visualization Options [cite: 55-56]
        scene_option = mujoco.MjvOption()
        scene_option.flags[mujoco.mjtVisFlag.mjVIS_CONTACTPOINT] = True
        scene_option.flags[mujoco.mjtVisFlag.mjVIS_CONTACTFORCE] = True
        scene_option.flags[mujoco.mjtVisFlag.mjVIS_TRANSPARENT] = True

        # Refined visual scaling
        model.vis.scale.contactwidth = 0.08
        model.vis.scale.contactheight = 0.02
        model.vis.scale.forcewidth = 0.04
        model.vis.map.force = 0.2

    mujoco.mj_resetData(model, data)

    # Initial velocity (world frame), parallel to ramp surface
    data.qvel[0] = initial_speed * math.cos(theta_rad)    # world X
```

```
    data.qvel[1] = 0.0                                          # world Y
    data.qvel[2] = -initial_speed * math.sin(theta_rad)   # world Z

    # Required to update derived quantities (like COM) before the first step
    mujoco.mj_forward(model, data)

    # Simulation bookkeeping
    timestep_model = model.opt.timestep

    if timestep_model <= 0:
        print(f"Error: Timestep must be positive. Current value: {timestep_model}")
        return {}

    try:
        steps_per_frame = round((1.0 / fps) / timestep_model)
    except ZeroDivisionError:
        print(f"Error: FPS or Timestep resulted in division by zero. FPS: {fps},
Timestep: {timestep_model}")
        return {}

    # Ensure at least one step per frame, relevant if timestep > 1/fps
    if steps_per_frame < 1:
        steps_per_frame = 1

    # IDs and directions
    hex_body_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_BODY, "link1")
    hex_geom_ids = [mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_GEOM, f"g{i}") for
i in range(1, 7)]

    ramp_dir = np.array([math.cos(theta_rad), 0.0, -math.sin(theta_rad)],
dtype=float)
    # Normalize (should be 1, but safeguard against numerical issues)
    norm_ramp_dir = np.linalg.norm(ramp_dir)
    if norm_ramp_dir > 1e-6:
        ramp_dir /= norm_ramp_dir

    # Logging arrays initialization
    frames = []
    times = []
    speed_mag = []
    com_positions = []
    s_along_ramp = []
    z_heights = []
    v_along_ramp = []
    normal_forces = []
    friction_forces = []
    rolls = []
    pitches = []
    yaws = []
    angular_speeds = []
    link_forces_history = []
    contact_torques_history = []
    all_contact_points_history = []
    penetration_depths = []
    cop_ramp_history = []
    sliding_forces_history = []
```

```
    torsional_torques_history = []
    rolling_torques_history = []
    net_force_world_history = []
    net_torque_world_history = []
    contact_counts = []

    # Debug: print contact structure once
    debug_printed = False
    warning_dim_printed = False

    # --- 4. Simulation Loop ---
    while data.time < duration:
        # Perform physics steps
        for _ in range(steps_per_frame):
            mujoco.mj_step(model, data)

        # Render and store video frame (optional)
        if record_video and renderer is not None and scene_option is not None:
            renderer.update_scene(data, camera="cam_close_track",
scene_option=scene_option)
            frames.append(renderer.render())

        # Data Logging (kinematics)
        times.append(data.time)

        # Ensure COM is current before contact loop (mj_step has updated kinematics)
        com = data.subtree_com[hex_body_id].copy()
        com_positions.append(com)
        z_heights.append(com[2])

        # Assuming the free joint for the hexagon starts at index 0 in qvel
        vel_world = data.qvel[0:3].copy()
        speed_mag.append(np.linalg.norm(vel_world))

        ang_vel_world = data.qvel[3:6].copy()
        angular_speeds.append(np.linalg.norm(ang_vel_world))

        s_along_ramp.append(np.dot(com - ramp_world_center, ramp_dir))
        v_along_ramp.append(np.dot(vel_world, ramp_dir))

        quat = data.xquat[hex_body_id].copy()
        roll, pitch, yaw = quat_to_euler_wxyz(quat)
        rolls.append(roll)
        pitches.append(pitch)
        yaws.append(yaw)

        # Contact Force Logic Initialization
        cf = np.zeros(6, dtype=np.float64)

        # Accumulators for sums of magnitudes (useful for visualization/simple
friction models)
        total_normal = 0.0
        total_fric = 0.0
        total_torque = 0.0

        total_sliding = 0.0
```

```
        total_torsional = 0.0
        total_rolling = 0.0

        current_step_link_forces = np.zeros(6)
        step_contact_points = []
        cop_weighted_pos = np.zeros(3)
        min_dist_this_step = 0.0  # most negative = deepest penetration

        # Accumulators for Global Vector Summation (world-frame wrench)
        step_net_force_world = np.zeros(3)
        step_net_torque_world = np.zeros(3)
        step_active_contacts = 0

        for ci in range(data.ncon):
            contact = data.contact[ci]

            # Determine Force Directionality (CRITICAL CORRECTION)
            force_direction = 0.0
            link_index = -1

            # mj_contactForce returns the wrench exerted BY geom2 ON geom1.
            if contact.geom1 in hex_geom_ids:
                link_index = hex_geom_ids.index(contact.geom1)
                # Force by geom2 (ramp) on geom1 (hexagon). Direction is correct.
                force_direction = 1.0
            elif contact.geom2 in hex_geom_ids:
                link_index = hex_geom_ids.index(contact.geom2)
                # Force by geom2 (hexagon) on geom1 (ramp). We need the reaction
force on hex.
                force_direction = -1.0

            if force_direction == 0.0:
                # Contact does not involve the hexagon
                continue

            # Check contact dimension [cite: 11]
            if contact.dim < 6 and not warning_dim_printed and data.time < 0.02:
                print(f"Warning: Contact dimension is {contact.dim}.
Torsional/Rolling friction disabled for this contact.")
                warning_dim_printed = True

            # Optional debug of contact internal structure
            if enable_contact_debug and (not debug_printed) and contact.dist < -1e-
5:
                print_contact_debug(contact, model)
                debug_printed = True

            mujoco.mj_contactForce(model, data, ci, cf)

            # Apply the direction correction so cf always represents "force on
hexagon"
            cf *= force_direction

            # If contact dimension is 3 (point), torque components are
invalid/unused
            if contact.dim < 6:
```

```
                cf[3:6] = 0.0

            # Count active contacts
            step_active_contacts += 1

            # Contact frame axes in world coordinates (contact.frame stores them
flattened)
            normal_axis = contact.frame[0:3]
            tangent1_axis = contact.frame[3:6]
            tangent2_axis = contact.frame[6:9]

            # Forces in world frame
            f_vec_world = (cf[0] * normal_axis +
                           cf[1] * tangent1_axis +
                           cf[2] * tangent2_axis)

            # Intrinsic torques in world frame (torsional + rolling)
            t_vec_world = (cf[3] * normal_axis +   # Torsional around normal
                           cf[4] * tangent1_axis +  # Rolling around T1
                           cf[5] * tangent2_axis)   # Rolling around T2

            step_net_force_world += f_vec_world
            # Add intrinsic torques
            step_net_torque_world += t_vec_world

            # Add torque generated by translational force around the COM: τ = r × F
            r_vec = contact.pos - com
            torque_from_force = np.cross(r_vec, f_vec_world)
            step_net_torque_world += torque_from_force

            # Interpretation of cf components in local frame for magnitude logging:
            f_normal = cf[0]
            f_slide = np.linalg.norm(cf[1:3])
            f_fric = f_slide

            if contact.dist < min_dist_this_step:
                min_dist_this_step = contact.dist

            tau_tor = abs(cf[3])
            tau_roll = np.linalg.norm(cf[4:6])
            f_torque = np.linalg.norm(cf[3:6])  # Magnitude of intrinsic torques

            # Summing magnitudes (useful for simple visualization)
            total_normal += f_normal
            total_fric += f_fric
            total_torque += f_torque

            total_sliding += f_slide
            total_torsional += tau_tor
            total_rolling += tau_roll

            step_contact_points.append(contact.pos.copy())

            if link_index != -1:
                current_step_link_forces[link_index] += f_normal
```

```
            # CoP accumulation (weighted by positive normal force)
            # Ensure normal force is positive before using it as a weight
            if f_normal > 1e-9:
                cop_weighted_pos += contact.pos * f_normal

        # Center of pressure along ramp calculation
        # Normalize by the sum of positive normal forces used in weighted position
calculation.
        sum_positive_normal = np.sum(
            current_step_link_forces[current_step_link_forces > 1e-9]
        )

        if sum_positive_normal > 1e-6:
            cop_world = cop_weighted_pos / sum_positive_normal
            cop_vec = cop_world - com
            cop_longitudinal = np.dot(cop_vec, ramp_dir)
            cop_ramp_history.append(cop_longitudinal)
        else:
            # CoP undefined if there is no reliable normal force → treat as COM (0
offset)
            cop_ramp_history.append(0.0)

        # Logging the sums of magnitudes
        normal_forces.append(total_normal)
        friction_forces.append(total_fric)
        contact_torques_history.append(total_torque)
        link_forces_history.append(current_step_link_forces)
        penetration_depths.append(min_dist_this_step)

        sliding_forces_history.append(total_sliding)
        torsional_torques_history.append(total_torsional)
        rolling_torques_history.append(total_rolling)

        # Logging the World-frame vector wrenches
        net_force_world_history.append(step_net_force_world)
        net_torque_world_history.append(step_net_torque_world)
        contact_counts.append(step_active_contacts)

        if step_contact_points:
            all_contact_points_history.append(np.array(step_contact_points))
        else:
            all_contact_points_history.append(np.empty((0, 3)))

    # Convert logs to numpy arrays
    times = np.array(times)
    speed_mag = np.array(speed_mag)
    com_positions = np.vstack(com_positions) if com_positions else np.zeros((0, 3))
    s_along_ramp = np.array(s_along_ramp)
    z_heights = np.array(z_heights)
    v_along_ramp = np.array(v_along_ramp)
    normal_forces = np.array(normal_forces)
    friction_forces = np.array(friction_forces)
    contact_torques = np.array(contact_torques_history)
    link_forces_history = np.array(link_forces_history)
    penetration_depths = np.array(penetration_depths)
    rolls = np.array(rolls)
```

```
    pitches = np.array(pitches)
    yaws = np.array(yaws)
    angular_speeds = np.array(angular_speeds)
    cop_ramp_history = np.array(cop_ramp_history)

    sliding_forces = np.array(sliding_forces_history)
    torsional_torques = np.array(torsional_torques_history)
    rolling_torques = np.array(rolling_torques_history)

    net_force_world = np.array(net_force_world_history)
    net_torque_world = np.array(net_torque_world_history)
    contact_counts = np.array(contact_counts, dtype=int)

    sim_results = {
        # Parameters
        "theta_deg": theta_deg,
        "theta_rad": theta_rad,
        "mu_static": mu_static,
        "mu_dynamic": mu_dynamic,
        "mu_slide": mu_slide,
        "total_mass": total_mass,
        "n_links": n_links,
        "mass_per_link": mass_per_link,
        "duration": duration,
        "fps": fps,
        "timestep": timestep_model,
        "integrator": integrator,
        "solref": solref_override,
        "solimp": solimp_override,
        "initial_speed": initial_speed,

        # Time series data
        "times": times,
        "speed_mag": speed_mag,
        "com_positions": com_positions,
        "s_along_ramp": s_along_ramp,
        "z_heights": z_heights,
        "v_along_ramp": v_along_ramp,
        "normal_forces": normal_forces,
        "friction_forces": friction_forces,
        "contact_torques": contact_torques,
        "all_contact_points": all_contact_points_history,
        "penetration_depths": penetration_depths,
        "link_forces": link_forces_history,
        "roll": rolls,
        "pitch": pitches,
        "yaw": yaws,
        "angular_speeds": angular_speeds,
        "cop_ramp_history": cop_ramp_history,

        # 6D wrench decomposition (local frame magnitudes)
        "sliding_forces": sliding_forces,
        "torsional_torques": torsional_torques,
        "rolling_torques": rolling_torques,

        # World-frame net wrench (Concept A, vectors)
```

```
        "net_force_world": net_force_world,
        "net_torque_world": net_torque_world,

        # Active contact counts (Concept C)
        "contact_counts": contact_counts,

        # Metadata
        "frames": frames,
        "ramp_world_center": ramp_world_center,
        "ramp_dir": ramp_dir,
    }

    return sim_results
```

6. Default Parameters and `main()` Orchestration

This section defines:

- A `default_params` dictionary used when the simulation fails (e.g. MuJoCo missing).
- A `main()` function that:
  - Runs the primary **RK4** simulation and an **Euler** comparison.
  - Computes derived quantities (energies, friction coefficients, etc.).
  - Stores all intermediate results in variables that later cells will use for plotting.

In this version of the notebook, **we split the plotting into separate cells** so that each group of plots can be introduced and interpreted individually.

```
# --- 5. Run Simulation (wrapper for computation & logging) ---

# Initialize default parameters for visualization setup if simulation fails
default_params = {
    "theta_deg": THETA_DEG,
    "theta_rad": math.radians(THETA_DEG),
    "mu_static": MU_STATIC,
    "mu_slide": math.tan(math.radians(THETA_DEG)),
    "total_mass": TOTAL_MASS,
    "duration": DURATION,
    "fps": FPS,
    "timestep": DEFAULT_TIMESTEP,
    "n_links": N_LINKS,
    "initial_speed": INITIAL_SPEED,
    "integrator": DEFAULT_INTEGRATOR,
    "solref": DEFAULT_SOLREF,
    "solimp": DEFAULT_SOLIMP,
}


def main(run_video=True):
    """
    Orchestrates the MuJoCo simulation and prepares data for plotting.

    This function populates the following global variables used by later plotting
```

```
cells:
    - results, results_euler, simulation_success
    - times, speed_mag, s_along_ramp, v_along_ramp, z_heights
    - normal_forces, friction_forces, contact_torques, penetration_depths
    - link_forces, angular_speeds, cop_ramp_history, com_positions,
all_contact_points
    - sliding_forces, torsional_torques, rolling_torques
    - net_force_world, net_torque_world, contact_counts
    - theta_deg, theta_rad, mu_slide, total_mass, duration, fps, timestep,
      integrator_used, initial_speed
    - roll_deg, pitch_deg, yaw_deg, scenario_label, a_along_ramp,
      steady_mask, v_mean_steady, E_pot, E_kin, E_tot,
      mu_inst, N_theory, F_fric_theory
    """
    global results, results_euler, simulation_success
    global times, speed_mag, s_along_ramp, v_along_ramp, z_heights
    global normal_forces, friction_forces, contact_torques, penetration_depths
    global link_forces, angular_speeds, cop_ramp_history, com_positions,
all_contact_points
    global sliding_forces, torsional_torques, rolling_torques
    global net_force_world, net_torque_world, contact_counts
    global theta_deg, theta_rad, mu_slide, total_mass, duration, fps, timestep
    global integrator_used, initial_speed
    global roll_deg, pitch_deg, yaw_deg, scenario_label
    global a_along_ramp, steady_mask, v_mean_steady
    global E_pot, E_kin, E_tot, mu_inst, N_theory, F_fric_theory

    simulation_success = False
    results = {}          # RK4 (primary) results
    results_euler = None

    # Execute Simulation
    if MUJOCO_AVAILABLE:
        try:
            print("Running MuJoCo simulation (RK4)...")
            sim_ret = run_sim(
                theta_deg=THETA_DEG,
                mu_static=MU_STATIC,
                total_mass=TOTAL_MASS,
                n_links=N_LINKS,
                duration=DURATION,
                fps=FPS,
                initial_speed=INITIAL_SPEED,
                integrator="RK4",
                timestep=DEFAULT_TIMESTEP,
                solref_override=DEFAULT_SOLREF,
                solimp_override=DEFAULT_SOLIMP,
                enable_contact_debug=True,
                record_video=run_video,
            )
            results = sim_ret
            if results and results.get("times") is not None and
results.get("times").size > 0:
                simulation_success = True
                print("Primary simulation (RK4) finished successfully.")
```

```
                # Sensitivity Analysis (Euler)
                print("Running Sensitivity Analysis (Euler)...")
                results_euler = run_sim(
                    theta_deg=THETA_DEG,
                    mu_static=MU_STATIC,
                    total_mass=TOTAL_MASS,
                    n_links=N_LINKS,
                    duration=2.0,      # shorter comparison run
                    fps=FPS,
                    initial_speed=INITIAL_SPEED,
                    integrator="Euler",
                    timestep=DEFAULT_TIMESTEP,
                    solref_override=DEFAULT_SOLREF,
                    solimp_override=DEFAULT_SOLIMP,
                    enable_contact_debug=False,
                    record_video=False,
                )
            except Exception as e:
                print(f"\nERROR during simulation execution: {e}\n")
                traceback.print_exc()
        else:
            print("MuJoCo not available. Skipping simulation.")

        # If simulation failed or returned empty, populate results with defaults for
visualization setup
        if not simulation_success:
            print("Proceeding with empty data structures for visualization setup.")
            if not results:
                results = default_params.copy()
                results.update({
                    "times": np.array([]),
                    "speed_mag": np.array([]),
                    "s_along_ramp": np.array([]),
                    "v_along_ramp": np.array([]),
                    "z_heights": np.array([]),
                    "normal_forces": np.array([]),
                    "friction_forces": np.array([]),
                    "contact_torques": np.array([]),
                    "penetration_depths": np.array([]),
                    "link_forces": np.zeros((0, 6)),
                    "roll": np.array([]),
                    "pitch": np.array([]),
                    "yaw": np.array([]),
                    "angular_speeds": np.array([]),
                    "cop_ramp_history": np.array([]),
                    "frames": [],
                    "com_positions": np.zeros((0, 3)),
                    "all_contact_points": [],
                    "sliding_forces": np.array([]),
                    "torsional_torques": np.array([]),
                    "rolling_torques": np.array([]),
                    "net_force_world": np.zeros((0, 3)),
                    "net_torque_world": np.zeros((0, 3)),
                    "contact_counts": np.array([], dtype=int),
                })
```

```python
    # Extract results (using .get() for safety)
    times = results.get("times", np.array([]))
    speed_mag = results.get("speed_mag", np.array([]))
    s_along_ramp = results.get("s_along_ramp", np.array([]))
    v_along_ramp = results.get("v_along_ramp", np.array([]))
    z_heights = results.get("z_heights", np.array([]))
    normal_forces = results.get("normal_forces", np.array([]))
    friction_forces = results.get("friction_forces", np.array([]))
    contact_torques = results.get("contact_torques", np.zeros_like(times))
    penetration_depths = results.get("penetration_depths", np.zeros_like(times))
    link_forces = results.get("link_forces", np.zeros((len(times), 6)))
    angular_speeds = results.get("angular_speeds", np.zeros_like(times))
    cop_ramp_history = results.get("cop_ramp_history", np.zeros_like(times))
    com_positions = results.get("com_positions", np.zeros((0, 3)))
    all_contact_points = results.get("all_contact_points", [])

    sliding_forces = results.get("sliding_forces", np.zeros_like(times))
    torsional_torques = results.get("torsional_torques", np.zeros_like(times))
    rolling_torques = results.get("rolling_torques", np.zeros_like(times))

    net_force_world = results.get("net_force_world", np.zeros((len(times), 3)))
    net_torque_world = results.get("net_torque_world", np.zeros((len(times), 3)))
    contact_counts = results.get("contact_counts", np.zeros(len(times), dtype=int))

    # Use defaults if keys are missing (safer than direct access)
    theta_deg = results.get("theta_deg", THETA_DEG)
    theta_rad = results.get("theta_rad", math.radians(THETA_DEG))
    mu_slide = results.get("mu_slide", math.tan(math.radians(THETA_DEG)))
    total_mass = results.get("total_mass", TOTAL_MASS)
    duration = results.get("duration", DURATION)
    fps = results.get("fps", FPS)
    timestep = results.get("timestep", DEFAULT_TIMESTEP)
    integrator_used = results.get("integrator", DEFAULT_INTEGRATOR)
    initial_speed = results.get("initial_speed", INITIAL_SPEED)

    # Orientation in degrees for plotting
    roll_deg = np.degrees(results.get("roll", np.array([])))
    pitch_deg = np.degrees(results.get("pitch", np.array([])))
    yaw_deg = np.degrees(results.get("yaw", np.array([])))

    # Scenario label (used in headings)
    scenario_label = (
        f"(θ={theta_deg:.1f}°, "
        f"μ<sub>k</sub>≈{mu_slide:.2f}, "
        f"v<sub>0</sub>={initial_speed:.1f} m/s, "
        f"{integrator_used}, Δt={timestep*1000:.1f} ms)"
    )

    # --- 6. Derived Quantities ---

    if len(times) > 1:
        # Use the time array for gradient calculation
        a_along_ramp = np.gradient(v_along_ramp, times)
    else:
        a_along_ramp = np.zeros_like(v_along_ramp)
```

```
    steady_start = 0.5
    if len(times) > 0:
        steady_mask = times >= steady_start
    else:
        steady_mask = np.array([], dtype=bool)

    v_mean_steady = np.mean(speed_mag[steady_mask]) if steady_mask.any() else
float("nan")

    # Energies (Translational only)
    E_pot = total_mass * G * z_heights
    E_kin = 0.5 * total_mass * speed_mag**2
    E_tot = E_pot + E_kin

    # Contact forces and effective friction
    # Handle potential division by zero if normal forces are near zero or negative
(due to soft constraints)
    mu_inst = np.divide(
        friction_forces,
        normal_forces,
        out=np.zeros_like(friction_forces),
        where=normal_forces > 1e-9
    )

    N_theory = total_mass * G * math.cos(theta_rad)
    # Only exact if acceleration is zero, but useful reference
    F_fric_theory = total_mass * G * math.sin(theta_rad)

    print("Computation finished. You can now run the plotting cells below.")
```

7. Run the Simulation (Computation Phase)

Run the next cell to execute `main()` and compute all trajectories and contact data.

You can choose whether to record video frames:

- `run_video=True` – record and show a video (more compute & memory).
- `run_video=False` – skip video for faster runs.

Once this cell has finished, all global variables needed for plotting will be available.

```
# Run the simulation and prepare data (set run_video=False for faster runs)
main(run_video=True)
```

```
Running MuJoCo simulation (RK4)...

============================================================
--- [DEBUG] MjContact Introspection ---
Dist (Penetration): -0.000151 m
Geom IDs: 1 <--> 3
Dimension: 6
Friction (5D array): [6.2e-01 6.2e-01 5.0e-03 1.0e-04 1.0e-04]
SolImp (Constraint Impedance): [9.45e-01 9.70e-01 1.00e-03 5.00e-01 2.00e+00]
```

```
SolRef (Constraint Reference): [0.0105 1.    ]
Contact Normal (World): [0.5269558  0.         0.84989269]
============================================================

Primary simulation (RK4) finished successfully.
Running Sensitivity Analysis (Euler)...
Computation finished. You can now run the plotting cells below.
```

8. Video Rendering (Optional)

If video frames were recorded during the simulation and `mediapy` is available, this cell displays the resulting animation and saves it as an MP4 file.

> If MuJoCo or Mediapy is not available, this cell will either do nothing or print a warning.

```python
# Show and/or save the video if available
if MUJOCO_AVAILABLE and media is not None:
    frames = results.get("frames", [])
    if frames:
        print("Rendering Mechanism Video...")
        try:
            media.show_video(frames, fps=fps)
            media.write_video("hexagon_mechanism_sliding.mp4", frames, fps=fps)
            print("Saved video to 'hexagon_mechanism_sliding.mp4'.")
        except Exception as e:
            print(f"Could not process or save video. Error: {e}")
    else:
        print("No frames recorded (run_video=False or simulation failed).")
else:
    print("Video rendering skipped: MuJoCo or Mediapy not available.")
```

```
Rendering Mechanism Video...
```

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

```
Saved video to 'hexagon_mechanism_sliding.mp4'.
```

9. Plot Styling Helper

To keep all Plotly figures visually consistent and **presentation-ready**, we define a helper function `apply_conference_style`. It unifies:

- Font family and sizes.
- Background, grid, and axis line style.
- Legend layout and hover mode.

All subsequent plots will call this helper.

```python
# Global Plotly styling configuration (used by all plots)
if PLOTLY_AVAILABLE:
    FONT_FAMILY = "Arial, Helvetica, sans-serif"
    FONT_COLOR = "#2b2b2b"
    TITLE_FONT_SIZE = 24
    AXIS_TITLE_FONT_SIZE = 20
    TICK_FONT_SIZE = 16
    LEGEND_FONT_SIZE = 18
    SUBPLOT_TITLE_FONT_SIZE = 20

    PLOT_TEMPLATE = "plotly_white"
    LINE_WIDTH = 3.0
    DASHED_LINE_WIDTH = 2.5
    AXIS_LINE_COLOR = "black"
    GRID_COLOR = "#e5e5e5"

    # ColorBrewer qualitative palette
    COLOR_PALETTE = [
        "#377eb8", "#ff7f00", "#4daf4a", "#e41a1c", "#984ea3", "#a65628",
    ]

    def apply_conference_style(fig, title, height=650, width=850, is_subplot=False):
        """
        Apply a consistent 'conference-ready' style.
        """
        layout_update = dict(
            title=dict(
                text=f"<b>{title}</b>",
                font=dict(
                    size=TITLE_FONT_SIZE,
                    family=FONT_FAMILY,
                    color=FONT_COLOR,
```

```
            ),
            x=0.5,
            y=0.96,
            xanchor="center",
            yanchor="top",
        ),
        template=PLOT_TEMPLATE,
        font=dict(
            family=FONT_FAMILY,
            size=TICK_FONT_SIZE,
            color=FONT_COLOR,
        ),
        colorway=COLOR_PALETTE,
        height=height,
        width=width,
        margin=dict(l=100, r=60, t=140, b=110),
        legend=dict(
            font=dict(size=LEGEND_FONT_SIZE),
            orientation="h",
            yanchor="bottom",
            y=1.04,
            xanchor="center",
            x=0.5,
            bgcolor="rgba(255, 255, 255, 0.9)",
            bordercolor="#cccccc",
            borderwidth=0.5,
        ),
        hovermode="x unified",
    )
    fig.update_layout(**layout_update)

    axis_config = dict(
        mirror=True,
        ticks="outside",
        showline=True,
        linewidth=1.5,
        linecolor=AXIS_LINE_COLOR,
        gridcolor=GRID_COLOR,
        zeroline=True,
        zerolinecolor=GRID_COLOR,
        title_font=dict(size=AXIS_TITLE_FONT_SIZE),
        tickfont=dict(size=TICK_FONT_SIZE),
    )
    fig.update_xaxes(**axis_config)
    fig.update_yaxes(**axis_config)

    if is_subplot:
        fig.for_each_annotation(
            lambda a: a.update(
                font=dict(
                    size=SUBPLOT_TITLE_FONT_SIZE,
                    family=FONT_FAMILY,
                    color=FONT_COLOR,
                )
            )
        )
```

```
        return fig
else:
    def apply_conference_style(fig, title, height=650, width=850, is_subplot=False):
        return fig  # no-op if Plotly not available
```

10. Simulation Parameter Summary (Table)

This cell shows a **compact summary table** of key simulation parameters:

- Inclination angle, mass, friction coefficients.
- Duration, timestep, FPS, integrator.
- Friction cone model (elliptic).

Use this as a quick reference when presenting results.

```
if PLOTLY_AVAILABLE:
    TABLE_HEADER_COLOR = COLOR_PALETTE[0]
    TABLE_ROW_EVEN = "#f2f2f2"
    TABLE_ROW_ODD = "#ffffff"

    fig_params = go.Figure(
        data=[
            go.Table(
                header=dict(
                    values=["<b>Parameter Description</b>", "<b>Symbol</b>", "
<b>Value</b>"],
                    align=["left", "center", "right"],
                    fill_color=TABLE_HEADER_COLOR,
                    font=dict(color="white", size=LEGEND_FONT_SIZE,
family=FONT_FAMILY),
                    line_color="darkslategray",
                    height=40,
                ),
                cells=dict(
                    values=[
                        [
                            "Inclination angle",
                            "Static friction coefficient",
                            "Kinetic friction coefficient",
                            "Hexagon total mass",
                            "Number of links",
                            "Initial speed",
                            "Simulation duration",
                            "MuJoCo timestep",
                            "Video FPS",
                            "Integrator",
                            "Friction Cone Model",
                        ],
                        [
                            "θ", "μ<sub>s</sub>", "μ<sub>k</sub>", "M",
                            "N", "v<sub>0</sub>", "T", "Δt", "FPS", "Int",
                            "Cone",
                        ],
```

```
                        [
                            f"{theta_deg:.2f}°",
                            f"{results.get('mu_static', MU_STATIC):.3f}",
                            f"{mu_slide:.3f}",
                            f"{total_mass * 1000:.1f} g",
                            f"{results.get('n_links', N_LINKS)}",
                            f"{initial_speed:.2f} m/s",
                            f"{duration:.2f} s",
                            f"{timestep * 1000:.1f} ms",
                            f"{fps}",
                            integrator_used,
                            "Elliptic",
                        ],
                    ],
                    align=["left", "center", "right"],
                    fill_color=[[TABLE_ROW_ODD, TABLE_ROW_EVEN] * 6],
                    font=dict(color=FONT_COLOR, size=TICK_FONT_SIZE,
family=FONT_FAMILY),
                    line_color="darkslategray",
                    height=35,
                ),
            )
        ]
    )
    fig_params.update_layout(
        title=dict(
            text="<b>Simulation Parameters Overview</b>",
            x=0.5,
            font=dict(size=TITLE_FONT_SIZE, family=FONT_FAMILY),
        ),
        template=PLOT_TEMPLATE,
        width=850,
        height=580,
        margin=dict(l=20, r=20, t=90, b=20),
    )
    fig_params.show()
else:
    print("Plotly not available: skipping parameter table.")
```

Simulation Parameters Overview

Parameter Description	Symbol	Value
Inclination angle	θ	31.80°
Static friction coefficient	μ_s	0.620
Kinetic friction coefficient	μ_k	0.620
Hexagon total mass	M	15.0 g
Number of links	N	6
Initial speed	v_0	1.00 m/s
Simulation duration	T	4.00 s
MuJoCo timestep	Δt	1.0 ms
Video FPS	FPS	60
Integrator	Int	RK4
Friction Cone Model	Cone	Elliptic

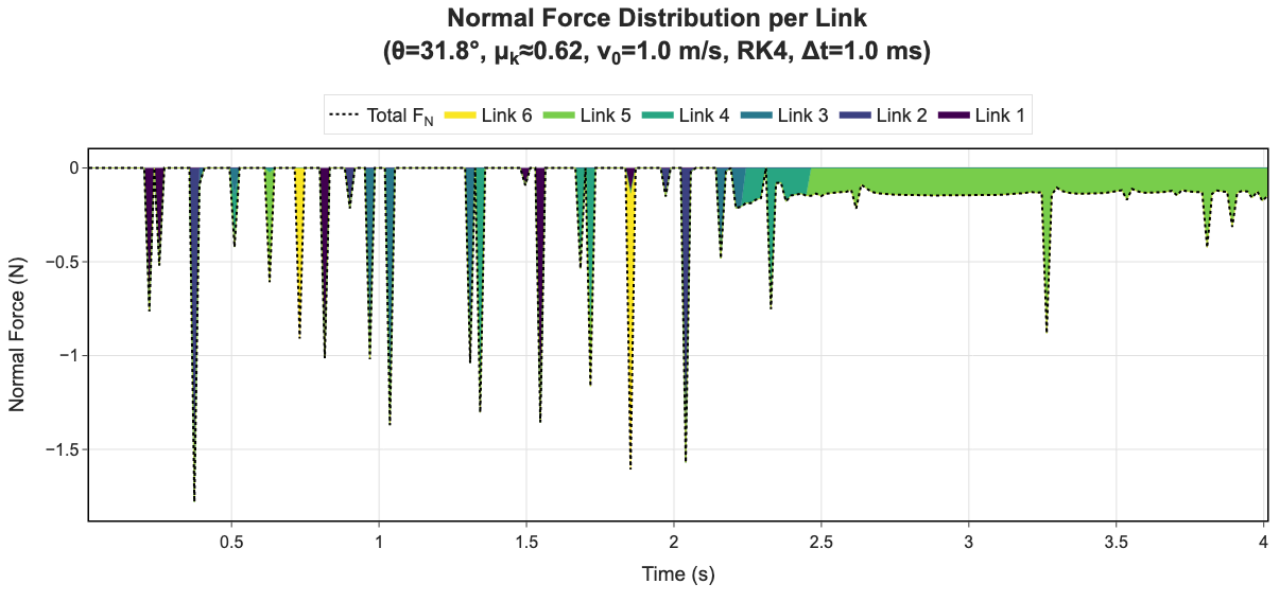
11. Normal Force Distribution per Link

This plot shows how the **total normal force** is distributed across the 6 links of the hexagonal mechanism over time (stacked area plot), along with the total normal force.

Use this to discuss **load sharing** and whether some links carry more load than others.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_contact = go.Figure()
    num_links = results.get("n_links", N_LINKS)
    link_colors = pc.sample_colorscale(
        "Viridis",
        [i / max(1, (num_links - 1)) for i in range(num_links)]
    )

    for i in range(num_links):
        fig_contact.add_trace(go.Scatter(
            x=times,
            y=link_forces[:, i],
            mode="lines",
            name=f"Link {i+1}",
            stackgroup="one",
            line=dict(width=0.5, color=link_colors[i]),
            fillcolor=link_colors[i],
        ))

    fig_contact.add_trace(go.Scatter(
        x=times,
        y=normal_forces,
        mode="lines",
        name="Total F<sub>N</sub>",
        line=dict(color="black", width=2, dash="dot"),
    ))

    fig_contact = apply_conference_style(
```

```
        fig_contact,
        f"Normal Force Distribution per Link<br>{scenario_label}",
        height=600,
    )
    fig_contact.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="Normal Force (N)",
        hovermode="x unified",
    )
    fig_contact.show()
elif not simulation_success:
    print("Simulation did not produce data: skipping link force plot.")
else:
    print("Plotly not available or insufficient data.")
```



12. Sliding Speed vs Time (Steady State Identification)

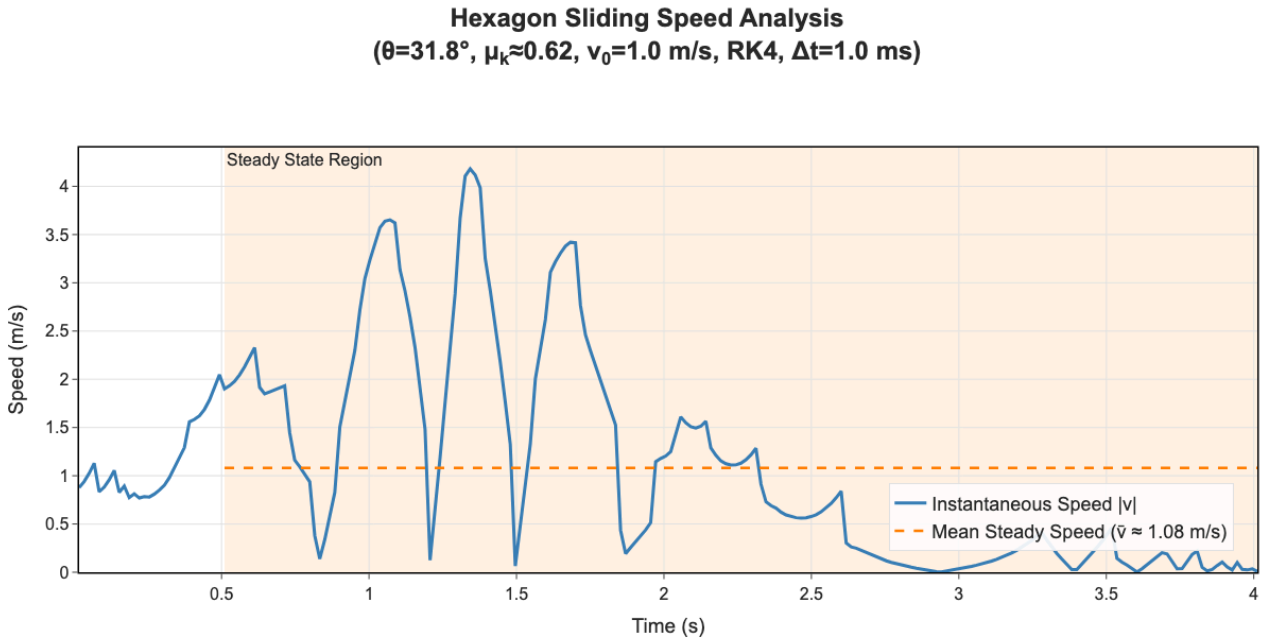
Here we plot the **magnitude of the velocity** of the hexagon over time and highlight the approximate **steady-state region** (after 0.5 s).

- The dashed horizontal line shows the mean steady-state speed.
- The shaded vertical region indicates the time window used for averaging.

This is useful for comparing with analytical or experimental steady sliding behavior.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_speed = go.Figure()
    fig_speed.add_trace(go.Scatter(
        x=times,
        y=speed_mag,
        mode="lines",
        name="Instantaneous Speed |v|",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[0]),
```

```
    ))
    if steady_mask.any() and not np.isnan(v_mean_steady):
        fig_speed.add_trace(go.Scatter(
            x=[times[steady_mask][0], times[steady_mask][-1]],
            y=[v_mean_steady, v_mean_steady],
            mode="lines",
            name=f"Mean Steady Speed (v̄ ≈ {v_mean_steady:.2f} m/s)",
            line=dict(dash="dash", width=DASHED_LINE_WIDTH, color=COLOR_PALETTE[1]),
        ))
        fig_speed.add_vrect(
            x0=float(times[steady_mask][0]),
            x1=float(times[steady_mask][-1]),
            fillcolor=COLOR_PALETTE[1],
            opacity=0.1,
            layer="below",
            line_width=0,
            annotation_text="Steady State Region",
            annotation_position="top left",
            annotation_font=dict(family=FONT_FAMILY, size=TICK_FONT_SIZE),
        )

    fig_speed = apply_conference_style(
        fig_speed,
        f"Hexagon Sliding Speed Analysis<br>{scenario_label}",
    )
    fig_speed.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="Speed (m/s)",
        yaxis=dict(rangemode="tozero"),
        legend=dict(
            orientation="v",
            yanchor="bottom",
            y=0.05,
            xanchor="right",
            x=0.98,
        ),
    )
    fig_speed.show()
else:
    print("Plotly not available or insufficient data for speed plot.")
```



13. Kinematics Along the Ramp: Position, Velocity, Acceleration

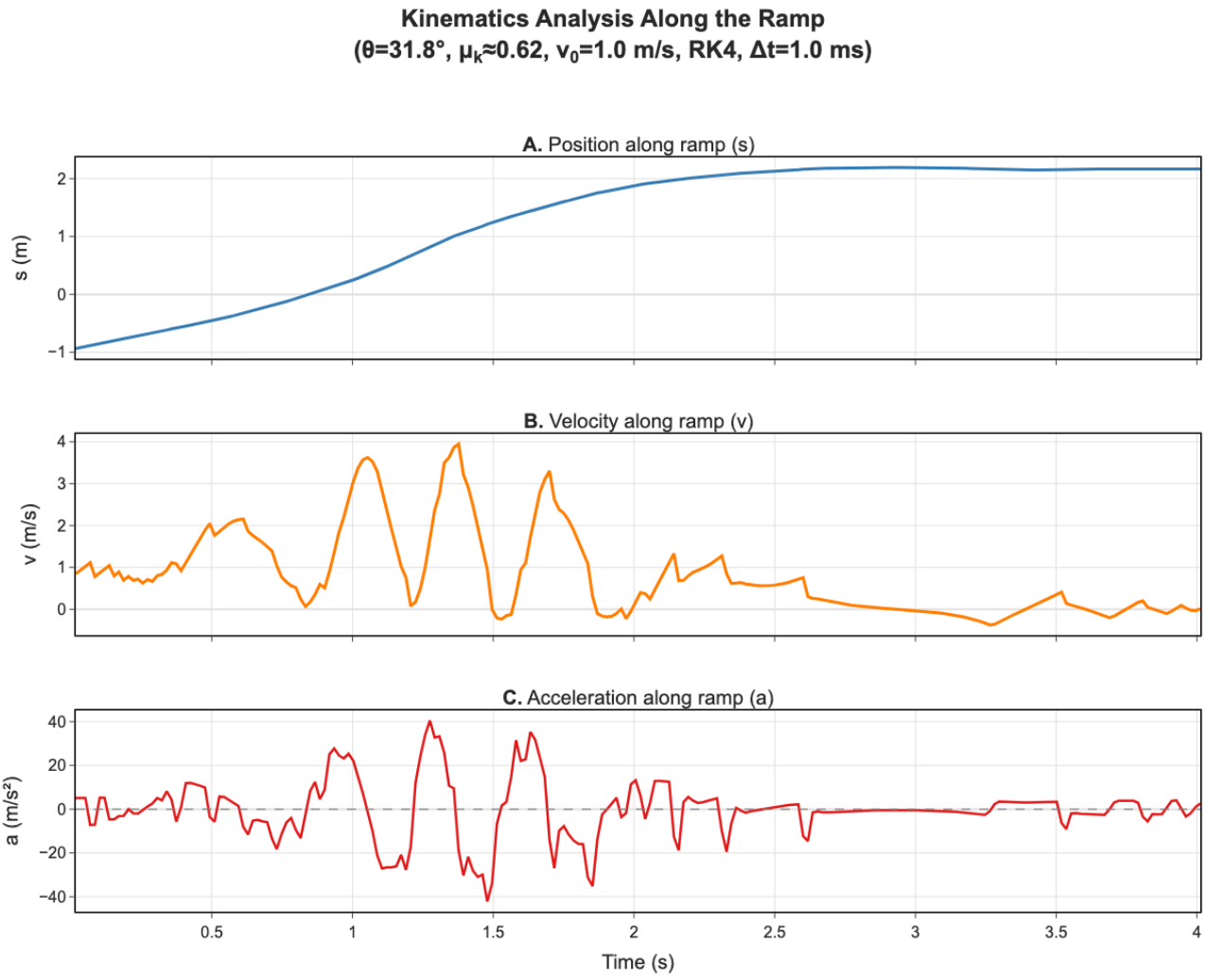
This cell shows a **three-panel subplot**:

1. Position along the ramp, (s(t))
2. Velocity along the ramp, (v(t))
3. Acceleration along the ramp, (a(t))

Together, they provide a full kinematic picture of motion constrained to the ramp direction.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_kin = make_subplots(
        rows=3, cols=1, shared_xaxes=True, vertical_spacing=0.1,
        subplot_titles=(
            "<b>A.</b> Position along ramp (s)",
            "<b>B.</b> Velocity along ramp (v)",
            "<b>C.</b> Acceleration along ramp (a)",
        ),
    )
    fig_kin.add_trace(go.Scatter(
        x=times, y=s_along_ramp, mode="lines",
        name="s(t)",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[0]),
    ), row=1, col=1)
    fig_kin.add_trace(go.Scatter(
        x=times, y=v_along_ramp, mode="lines",
        name="v(t)",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[1]),
    ), row=2, col=1)
    fig_kin.add_trace(go.Scatter(
        x=times, y=a_along_ramp, mode="lines",
```

```
        name="a(t)",
        line=dict(width=LINE_WIDTH - 0.5, color=COLOR_PALETTE[3]),
    ), row=3, col=1)
    fig_kin.add_hline(
        y=0, line_dash="dash", line_color="gray",
        line_width=1.5, row=3, col=1, opacity=0.8,
    )
    fig_kin.update_yaxes(title_text="s (m)", row=1, col=1)
    fig_kin.update_yaxes(title_text="v (m/s)", row=2, col=1)
    fig_kin.update_yaxes(title_text="a (m/s²)", row=3, col=1)
    fig_kin.update_xaxes(title_text="Time (s)", row=3, col=1)
    fig_kin = apply_conference_style(
        fig_kin,
        f"Kinematics Analysis Along the Ramp<br>{scenario_label}",
        height=1000,
        is_subplot=True,
    )
    fig_kin.update_layout(
        showlegend=False,
        margin=dict(l=100, r=50, t=150, b=100),
        title=dict(y=0.98),
    )
    fig_kin.show()
else:
    print("Plotly not available or insufficient data for kinematics plots.")
```



14. Energy Evolution: Potential, Kinetic, Total

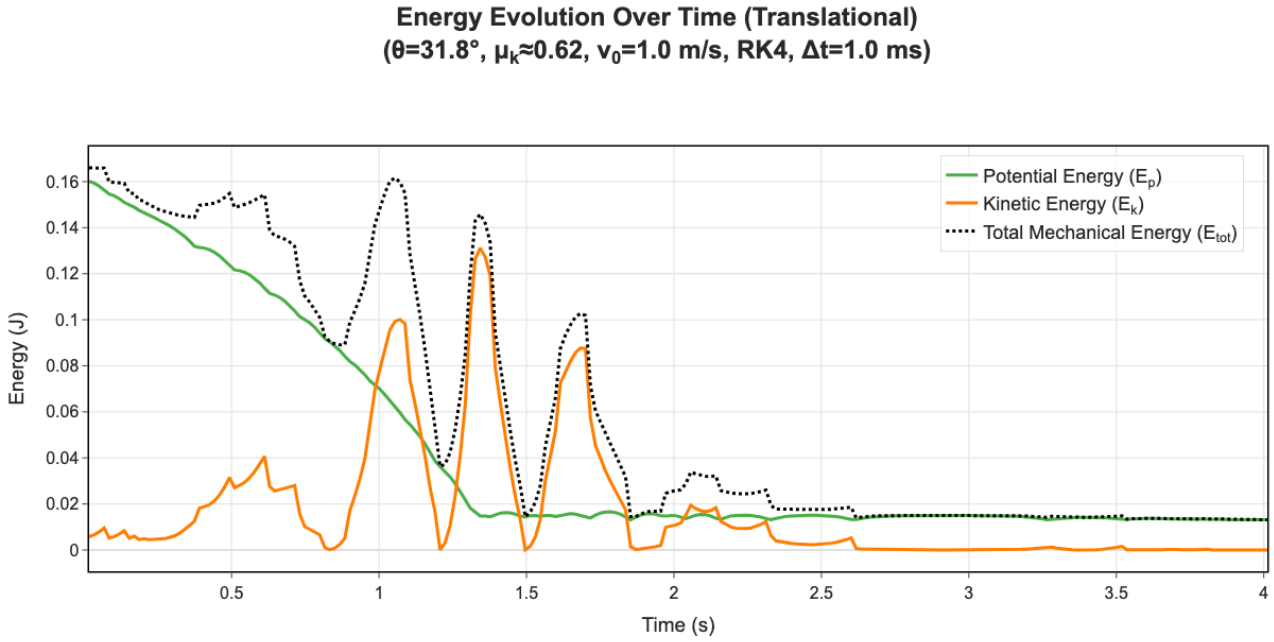
This plot shows the evolution of:

- Gravitational potential energy (E_p)
- Translational kinetic energy (E_k)
- Total mechanical energy (E_{tot} = E_p + E_k)

This is helpful to discuss **energy exchange** and **numerical dissipation**.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_energy = go.Figure()
    fig_energy.add_trace(go.Scatter(
        x=times, y=E_pot, mode="lines",
        name="Potential Energy (E<sub>p</sub>)",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[2]),
    ))
    fig_energy.add_trace(go.Scatter(
        x=times, y=E_kin, mode="lines",
        name="Kinetic Energy (E<sub>k</sub>)",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[1]),
```

```
    ))
    fig_energy.add_trace(go.Scatter(
        x=times, y=E_tot, mode="lines",
        name="Total Mechanical Energy (E<sub>tot</sub>)",
        line=dict(dash="dot", width=LINE_WIDTH, color="black"),
    ))
    fig_energy = apply_conference_style(
        fig_energy,
        f"Energy Evolution Over Time (Translational)<br>{scenario_label}",
    )
    fig_energy.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="Energy (J)",
        legend=dict(
            orientation="v", yanchor="top", y=0.98,
            xanchor="right", x=0.98,
        ),
    )
    fig_energy.show()
else:
    print("Plotly not available or insufficient data for energy plot.")
```



15. Normal & Friction Forces and Effective Friction Coefficient

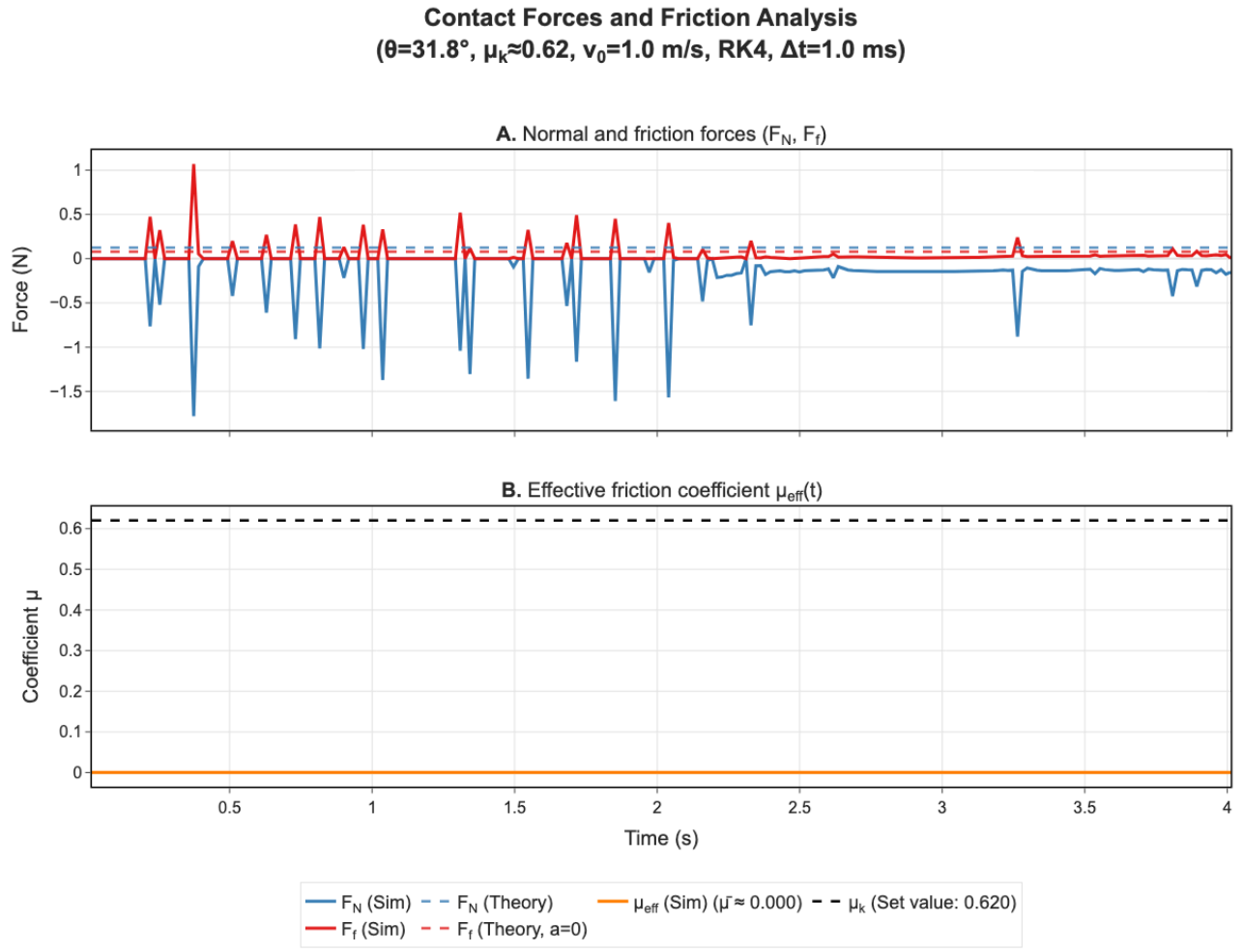
This two-panel plot compares:

1. Simulation vs. **theoretical** normal and friction forces, assuming steady sliding.
2. Instantaneous effective friction coefficient (μ_{eff}(t) = F_f / F_N), compared with the specified kinetic friction (μ_k).

Use this to validate the **friction model** and check whether the friction cone is respected.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_forces = make_subplots(
        rows=2, cols=1, shared_xaxes=True, vertical_spacing=0.12,
        subplot_titles=(
            "<b>A.</b> Normal and friction forces (F<sub>N</sub>, F<sub>f</sub>)",
            "<b>B.</b> Effective friction coefficient μ<sub>eff</sub>(t)",
        ),
    )
    fig_forces.add_trace(go.Scatter(
        x=times, y=normal_forces, mode="lines",
        name="F<sub>N</sub> (Sim)", legendgroup="forces",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[0]),
    ), row=1, col=1)
    fig_forces.add_trace(go.Scatter(
        x=times, y=friction_forces, mode="lines",
        name="F<sub>f</sub> (Sim)", legendgroup="forces",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[3]),
    ), row=1, col=1)
    fig_forces.add_trace(go.Scatter(
        x=times, y=np.full_like(times, N_theory),
        mode="lines", name="F<sub>N</sub> (Theory)",
        legendgroup="forces_theory",
        line=dict(dash="dash", width=DASHED_LINE_WIDTH, color=COLOR_PALETTE[0]),
        opacity=0.8,
    ), row=1, col=1)
    fig_forces.add_trace(go.Scatter(
        x=times, y=np.full_like(times, F_fric_theory),
        mode="lines", name="F<sub>f</sub> (Theory, a=0)",
        legendgroup="forces_theory",
        line=dict(dash="dash", width=DASHED_LINE_WIDTH, color=COLOR_PALETTE[3]),
        opacity=0.8,
    ), row=1, col=1)

    mu_eff_trace_name = "μ<sub>eff</sub> (Sim)"
    if steady_mask.any():
        mu_eff_steady_mean = float(np.nanmean(mu_inst[steady_mask]))
        mu_eff_trace_name += f" (μ̄≈ {mu_eff_steady_mean:.3f})"
    fig_forces.add_trace(go.Scatter(
        x=times, y=mu_inst, mode="lines",
        name=mu_eff_trace_name,
        legendgroup="friction",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[1]),
    ), row=2, col=1)
    fig_forces.add_trace(go.Scatter(
        x=times, y=np.full_like(times, mu_slide),
        mode="lines",
        name=f"μ<sub>k</sub> (Set value: {mu_slide:.3f})",
        legendgroup="friction_theory",
        line=dict(dash="dash", width=DASHED_LINE_WIDTH, color="black"),
    ), row=2, col=1)
    fig_forces.update_yaxes(title_text="Force (N)", row=1, col=1)
    fig_forces.update_yaxes(title_text="Coefficient μ", row=2, col=1)
    fig_forces.update_xaxes(title_text="Time (s)", row=2, col=1)
    fig_forces = apply_conference_style(
        fig_forces,
```

```
        f"Contact Forces and Friction Analysis<br>{scenario_label}",
        height=950,
        is_subplot=True,
    )
    fig_forces.update_layout(
        margin=dict(l=100, r=60, t=150, b=180),
        title=dict(y=0.97),
        legend=dict(
            orientation="h", yanchor="top", y=-0.15,
            xanchor="center", x=0.5, tracegroupgap=30,
        ),
    )
    fig_forces.show()
else:
    print("Plotly not available or insufficient data for forces/friction plot.")
```

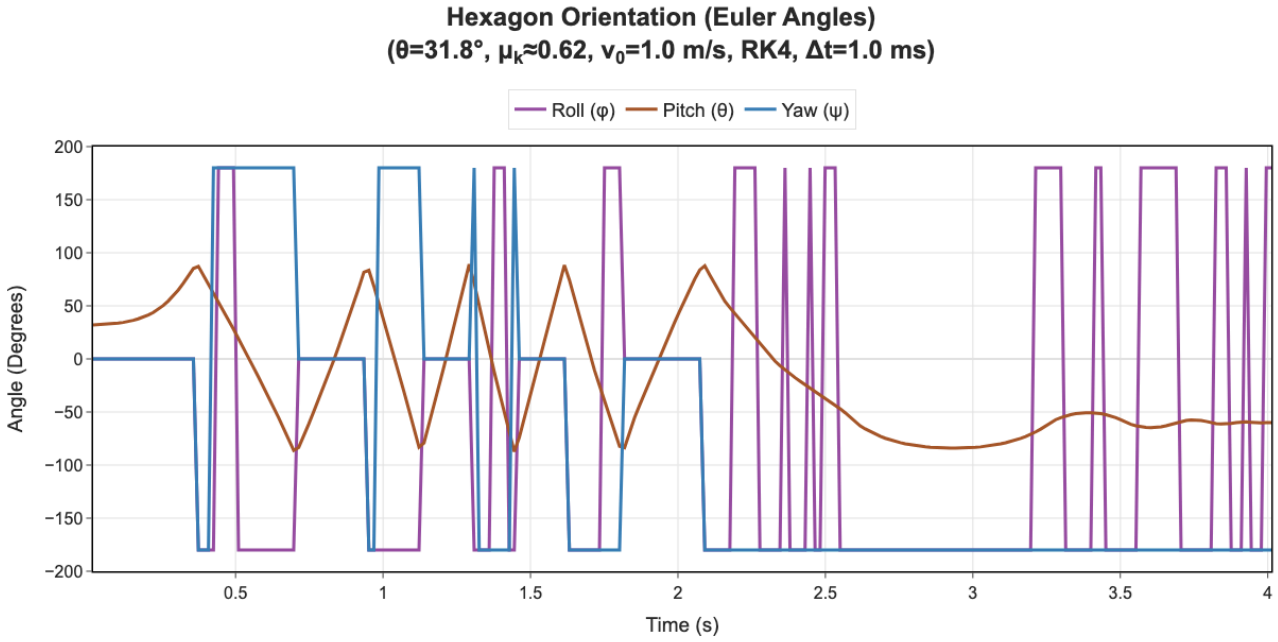


16. Orientation Over Time (Euler Angles)

This figure shows the **roll, pitch, and yaw** angles of the hexagon over time, obtained by converting MuJoCo quaternions to Euler angles.

It helps visualize any **rocking, rolling, or spinning** behavior as the mechanism slides.

```python
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_orient = go.Figure()
    fig_orient.add_trace(go.Scatter(
        x=times, y=roll_deg, mode="lines",
        name="Roll (φ)",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[4]),
    ))
    fig_orient.add_trace(go.Scatter(
        x=times, y=pitch_deg, mode="lines",
        name="Pitch (θ)",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[5]),
    ))
    fig_orient.add_trace(go.Scatter(
        x=times, y=yaw_deg, mode="lines",
        name="Yaw (ψ)",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[0]),
    ))
    fig_orient = apply_conference_style(
        fig_orient,
        f"Hexagon Orientation (Euler Angles)<br>{scenario_label}",
    )
    fig_orient.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="Angle (Degrees)",
    )
    fig_orient.show()
else:
    print("Plotly not available or insufficient data for orientation plot.")
```



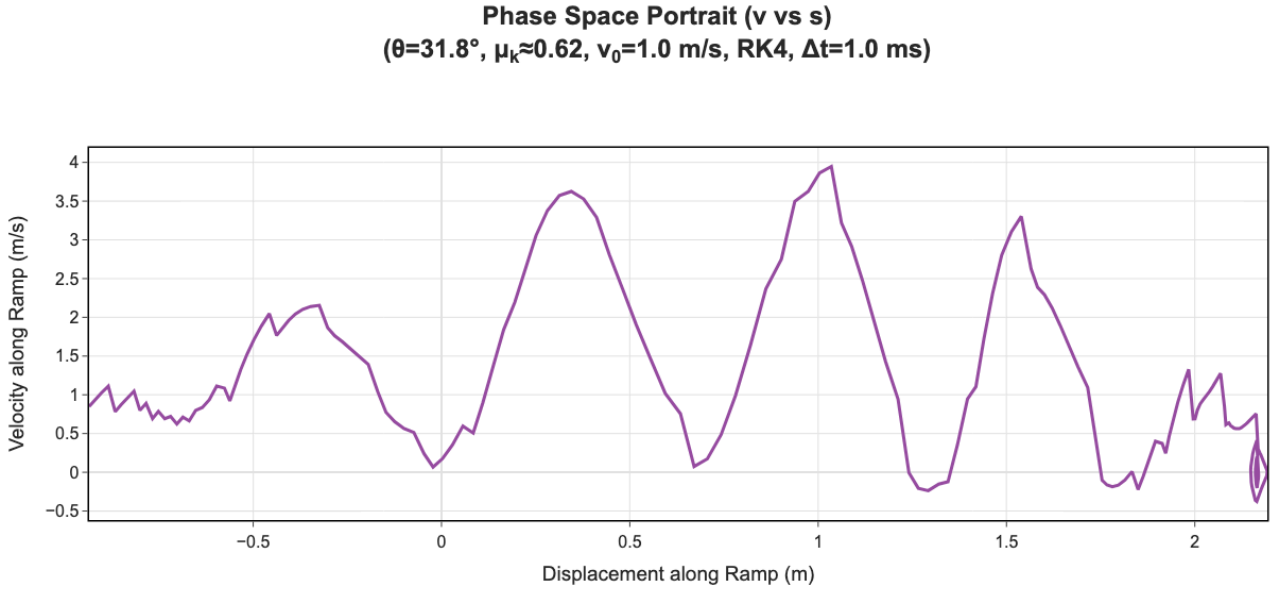
17. Phase Space Portrait: (v) vs (s)

This phase-space plot shows **velocity vs displacement along the ramp**:

- The trajectory in ((s, v))-space can indicate whether the motion approaches a steady state or exhibits oscillatory behavior.

It is a compact, dynamical-systems style view of the motion.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_phase = go.Figure()
    fig_phase.add_trace(go.Scatter(
        x=s_along_ramp,
        y=v_along_ramp,
        mode="lines",
        name="Phase Trajectory",
        line=dict(width=LINE_WIDTH, color=COLOR_PALETTE[4]),
    ))
    fig_phase = apply_conference_style(
        fig_phase,
        f"Phase Space Portrait (v vs s)<br>{scenario_label}",
    )
    fig_phase.update_layout(
        xaxis_title="Displacement along Ramp (m)",
        yaxis_title="Velocity along Ramp (m/s)",
        height=600,
        width=800,
    )
    fig_phase.show()
else:
    print("Plotly not available or insufficient data for phase-space portrait.")
```

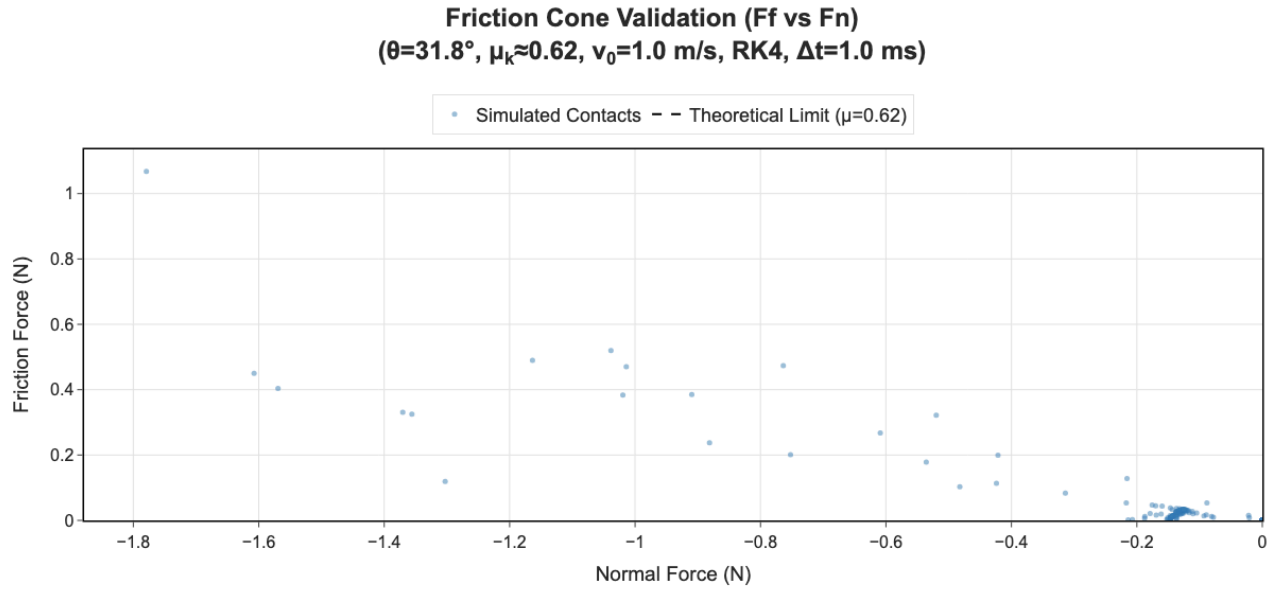


18. Friction Cone Validation: (F_f) vs (F_N)

This scatter plot compares the **friction force vs. normal force** at each timestep to the **theoretical friction cone limit** (F_f = μ_k F_N).

Points should lie **on or below** the dashed line if the friction model is correctly enforced.

```python
if PLOTLY_AVAILABLE and simulation_success and len(normal_forces) > 0:
    fig_cone = go.Figure()
    fig_cone.add_trace(go.Scatter(
        x=normal_forces,
        y=friction_forces,
        mode="markers",
        name="Simulated Contacts",
        marker=dict(size=5, color=COLOR_PALETTE[0], opacity=0.5),
    ))
    max_N = np.max(normal_forces) if len(normal_forces) > 0 else 1.0
    x_line = np.linspace(0, max_N * 1.1, 100)
    y_line = mu_slide * x_line
    fig_cone.add_trace(go.Scatter(
        x=x_line,
        y=y_line,
        mode="lines",
        name=f"Theoretical Limit (μ={mu_slide:.2f})",
        line=dict(dash="dash", color="black", width=2),
    ))
    fig_cone = apply_conference_style(
        fig_cone,
        f"Friction Cone Validation (Ff vs Fn)<br>{scenario_label}",
    )
    fig_cone.update_layout(
        xaxis_title="Normal Force (N)",
        yaxis_title="Friction Force (N)",
        xaxis=dict(rangemode="tozero"),
        yaxis=dict(rangemode="tozero"),
        height=600,
        width=800,
    )
    fig_cone.show()
else:
    print("Plotly not available or insufficient data for friction cone validation.")
```

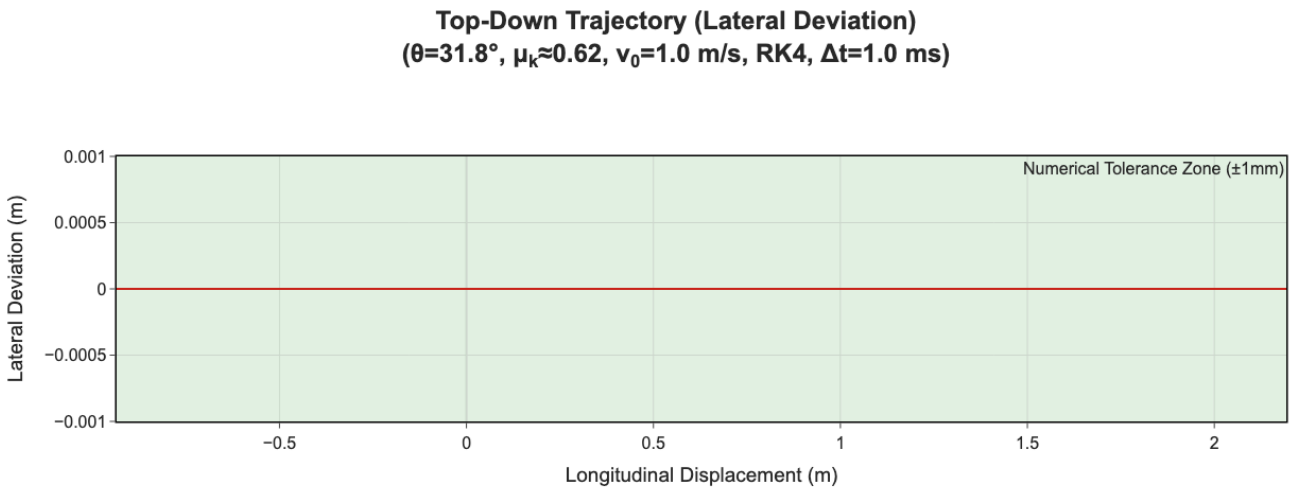


19. Lateral Deviation (Top-Down Trajectory)

This plot shows the **lateral (Y-axis) deviation** of the center of mass as a function of displacement along the ramp:

- The shaded band of ±1 mm represents a **numerical tolerance zone**.
- Deviations outside this band may indicate integration or contact asymmetry issues.

```
if PLOTLY_AVAILABLE and simulation_success and com_positions.size:
    lateral_pos = com_positions[:, 1]
    fig_drift = go.Figure()
    fig_drift.add_trace(go.Scatter(
        x=s_along_ramp,
        y=lateral_pos,
        mode="lines",
        name="Trajectory",
        line=dict(width=2, color=COLOR_PALETTE[3]),
    ))

    fig_drift.add_hrect(
        y0=-0.001, y1=0.001,
        line_width=0,
        fillcolor="green",
        opacity=0.1,
        annotation_text="Numerical Tolerance Zone (±1mm)",
        annotation_position="top right",
    )

    fig_drift = apply_conference_style(
        fig_drift,
        f"Top-Down Trajectory (Lateral Deviation)<br>{scenario_label}",
    )
    fig_drift.update_layout(
        xaxis_title="Longitudinal Displacement (m)",
        yaxis_title="Lateral Deviation (m)",
        height=500,
        width=800,
    )
    fig_drift.show()
else:
    print("Plotly not available or insufficient COM data for lateral deviation
plot.")
```



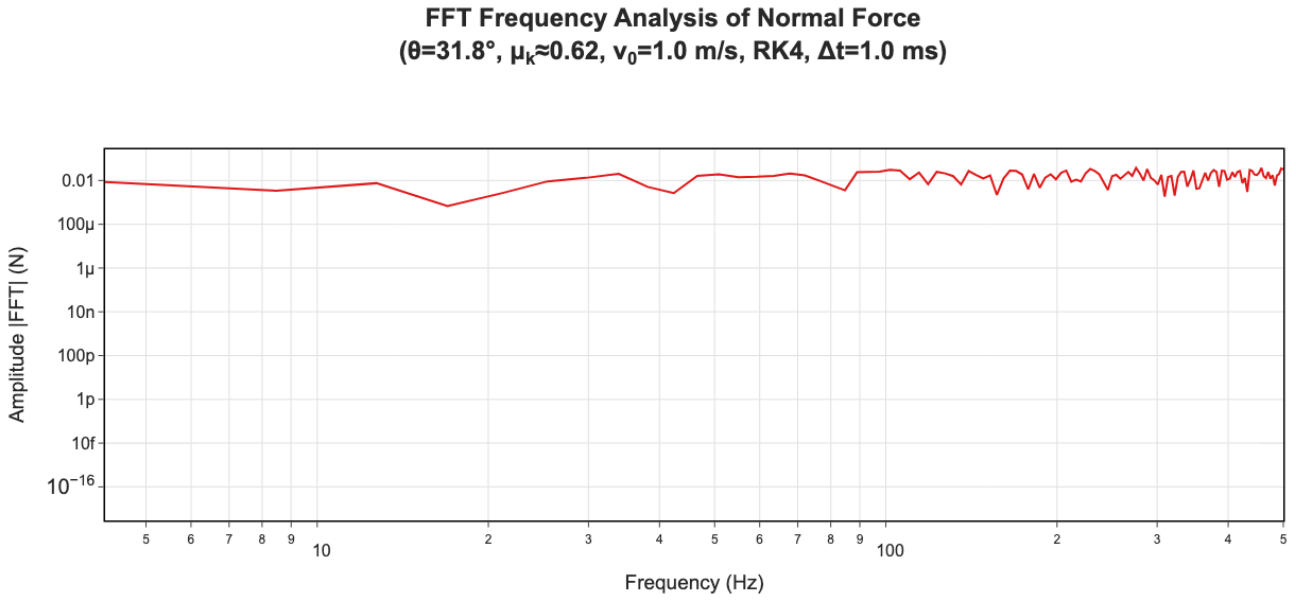
20. Frequency Analysis of Normal Force (FFT)

We perform a **Fast Fourier Transform (FFT)** of the normal force signal (after removing the mean), and plot the amplitude spectrum on log–log axes.

This highlights **numerical noise** and any dominant vibration frequencies induced by contact stiffness, timestep, or integrator choice.

```
if PLOTLY_AVAILABLE and simulation_success and len(normal_forces) > 1:
    n = len(normal_forces)
    dt = timestep
    signal_ac = normal_forces - np.mean(normal_forces)

    freqs = np.fft.rfftfreq(n, d=dt)
    fft_vals = np.abs(np.fft.rfft(signal_ac)) / n

    fig_fft = go.Figure()
    fig_fft.add_trace(go.Scatter(
        x=freqs,
        y=fft_vals,
        mode="lines",
        name="Normal Force Noise Amplitude",
        line=dict(color=COLOR_PALETTE[3], width=2),
    ))

    fig_fft = apply_conference_style(
        fig_fft,
        f"FFT Frequency Analysis of Normal Force<br>{scenario_label}",
    )
    fig_fft.update_layout(
        xaxis_title="Frequency (Hz)",
        yaxis_title="Amplitude |FFT| (N)",
        xaxis_type="log",
        yaxis_type="log",
        width=800,
        height=600,
    )
```

```
    fig_fft.show()
else:
    print("Plotly not available or insufficient data for FFT analysis.")
```



21. Effective Friction vs Sliding Speed (“Stribeck-like” Curve)

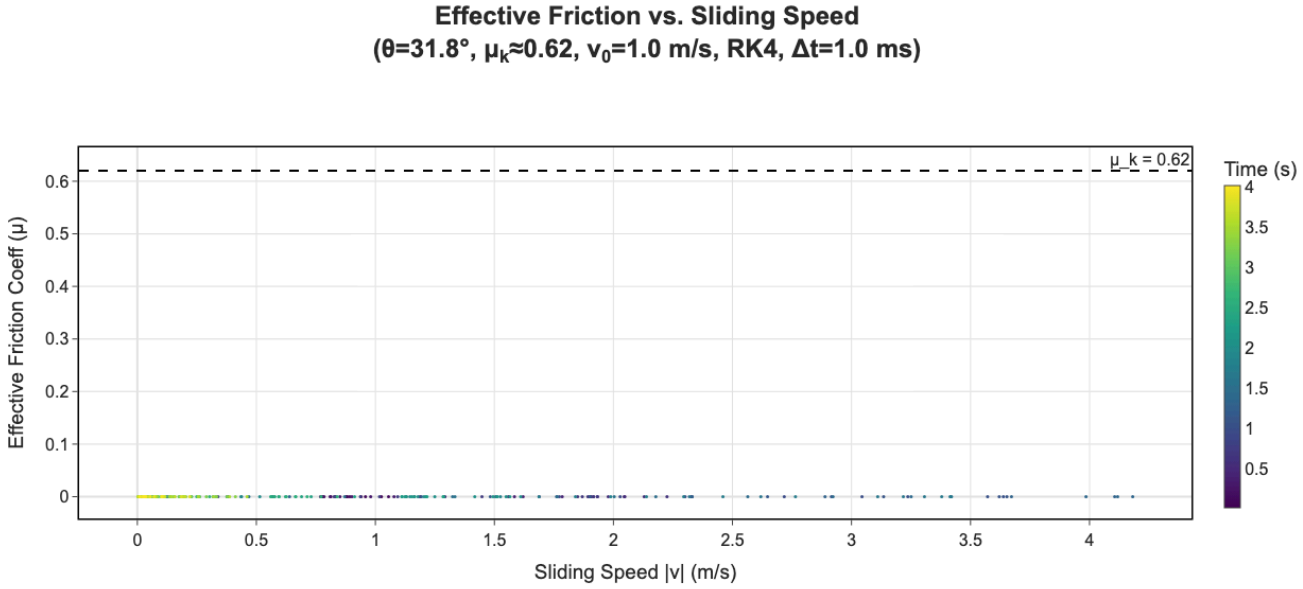
This scatter plot shows the relationship between **effective friction coefficient** and **speed**,

- Points are colored by time to show the evolution of contact behavior.
- The dashed horizontal line indicates the specified kinetic friction (μ_k).

This is analogous to a **Stribeck curve**, though this model uses simple Coulomb friction.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_stribeck = go.Figure()
    fig_stribeck.add_trace(go.Scatter(
        x=speed_mag,
        y=mu_inst,
        mode="markers",
        marker=dict(
            size=3,
            color=times,
            colorscale="Viridis",
            showscale=True,
            colorbar=dict(title="Time (s)", thickness=15),
        ),
        name="Simulated Data",
    ))
    fig_stribeck.add_hline(
        y=mu_slide,
        line_dash="dash",
        line_color="black",
```

```
        annotation_text=f"μ_k = {mu_slide:.2f}",
        annotation_position="top right",
    )

    fig_stribeck = apply_conference_style(
        fig_stribeck,
        f"Effective Friction vs. Sliding Speed<br>{scenario_label}",
    )
    fig_stribeck.update_layout(
        xaxis_title="Sliding Speed |v| (m/s)",
        yaxis_title="Effective Friction Coeff (μ)",
        height=600, width=800,
    )
    fig_stribeck.show()
else:
    print("Plotly not available or insufficient data for Stribeck-like plot.")
```



22. Rotational Stability: Angular Speed & Contact Torque

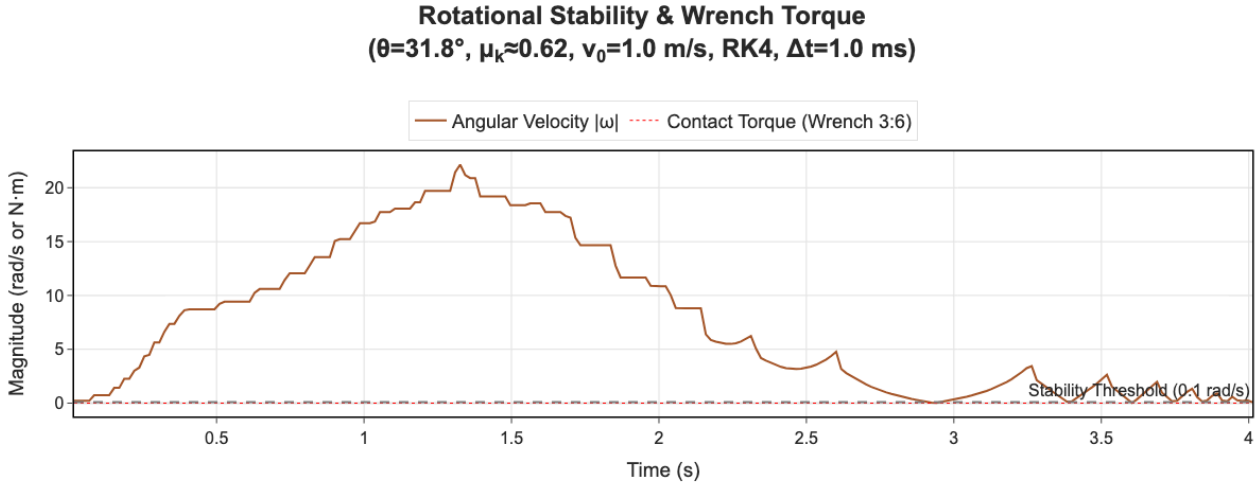
This plot examines **rotational stability** by showing:

- Angular speed magnitude (|ω|).
- Contact torque (aggregated from wrench components).

A threshold line (e.g., 0.1 rad/s) can be interpreted as a **“stability limit”** beyond which rotations might be considered significant or undesirable.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_stability = go.Figure()
    fig_stability.add_trace(go.Scatter(
        x=times,
        y=angular_speeds,
        mode="lines",
```

```
        name="Angular Velocity |ω|",
        line=dict(color=COLOR_PALETTE[5], width=2),
    ))
    fig_stability.add_trace(go.Scatter(
        x=times,
        y=contact_torques,
        mode="lines",
        name="Contact Torque (Wrench 3:6)",
        line=dict(color="red", width=1, dash="dot"),
    ))
    fig_stability.add_hline(
        y=0.1,
        line_dash="dash",
        line_color="gray",
        annotation_text="Stability Threshold (0.1 rad/s)",
        annotation_position="top right",
    )

    fig_stability = apply_conference_style(
        fig_stability,
        f"Rotational Stability & Wrench Torque<br>{scenario_label}",
    )
    fig_stability.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="Magnitude (rad/s or N·m)",
        height=500,
        width=800,
    )
    fig_stability.show()
else:
    print("Plotly not available or insufficient data for rotational stability
plot.")
```



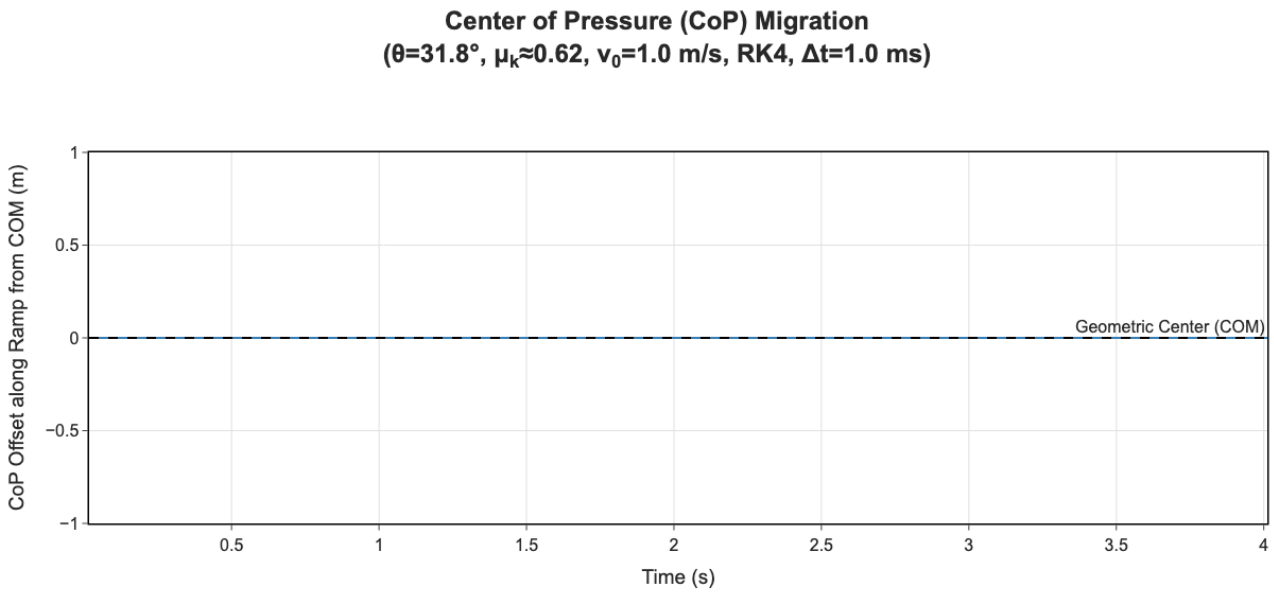
23. Center of Pressure (CoP) Migration Along the Ramp

This plot tracks the **center of pressure (CoP)** relative to the center of mass (COM) along the ramp direction:

- Positive/negative offsets indicate shifts upstream/downstream relative to COM.
- The dashed horizontal line at zero corresponds to the COM location.

This is useful to reason about **stability margins** and how the contact patch moves.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_cop = go.Figure()
    fig_cop.add_trace(go.Scatter(
        x=times,
        y=cop_ramp_history,
        mode="lines",
        fill="tozeroy",
        name="True Center of Pressure Shift (Weighted Contact Pos)",
        line=dict(color=COLOR_PALETTE[0]),
    ))
    fig_cop.add_hline(
        y=0, line_dash="dash",
        line_color="black",
        annotation_text="Geometric Center (COM)",
    )

    fig_cop = apply_conference_style(
        fig_cop,
        f"Center of Pressure (CoP) Migration<br>{scenario_label}",
    )
    fig_cop.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="CoP Offset along Ramp from COM (m)",
        height=600,
        width=800,
        legend=dict(
            orientation="h",
            yanchor="bottom",
            y=1.02,
            xanchor="center",
            x=0.5,
            bgcolor="rgba(255, 255, 255, 0.9)",
            bordercolor="#cccccc",
            borderwidth=0.5,
        ),
    )
    fig_cop.show()
else:
    print("Plotly not available or insufficient data for CoP migration plot.")
```



24. 3D Contact Point Distribution

This 3D figure visualizes:

- The **trajectory of the center of mass** in 3D.
- The **cloud of contact points** sampled every few timesteps.

It gives a spatial sense of how contacts are distributed on the ramp as the mechanism slides.

```
if PLOTLY_AVAILABLE and simulation_success and len(com_positions) > 0:
    fig_3d = go.Figure()
    fig_3d.add_trace(go.Scatter3d(
        x=com_positions[:, 0],
        y=com_positions[:, 1],
        z=com_positions[:, 2],
        mode="lines",
        name="Center of Mass",
        line=dict(color="blue", width=4),
    ))

    flat_cx, flat_cy, flat_cz = [], [], []
    step_skip = 5
    for i in range(0, len(all_contact_points), step_skip):
        pts = all_contact_points[i]
        if pts.shape[0] > 0:
            flat_cx.extend(pts[:, 0])
            flat_cy.extend(pts[:, 1])
            flat_cz.extend(pts[:, 2])

    if flat_cx:
        fig_3d.add_trace(go.Scatter3d(
            x=flat_cx,
            y=flat_cy,
```

```
            z=flat_cz,
            mode="markers",
            name="Contact Points",
            marker=dict(size=2, color="red", opacity=0.3),
        ))

    fig_3d.update_layout(
        title=dict(text="<b>3D Contact Point Distribution</b>", font=dict(size=24)),
        scene=dict(
            xaxis_title="World X",
            yaxis_title="World Y",
            zaxis_title="World Z",
            aspectmode="data",
        ),
        height=700,
        margin=dict(l=0, r=0, b=0, t=50),
    )
    fig_3d.show()
else:
    print("Plotly not available or insufficient data for 3D contact distribution
plot.")
```

25. Penetration Depth vs Time (Contact Quality Check)

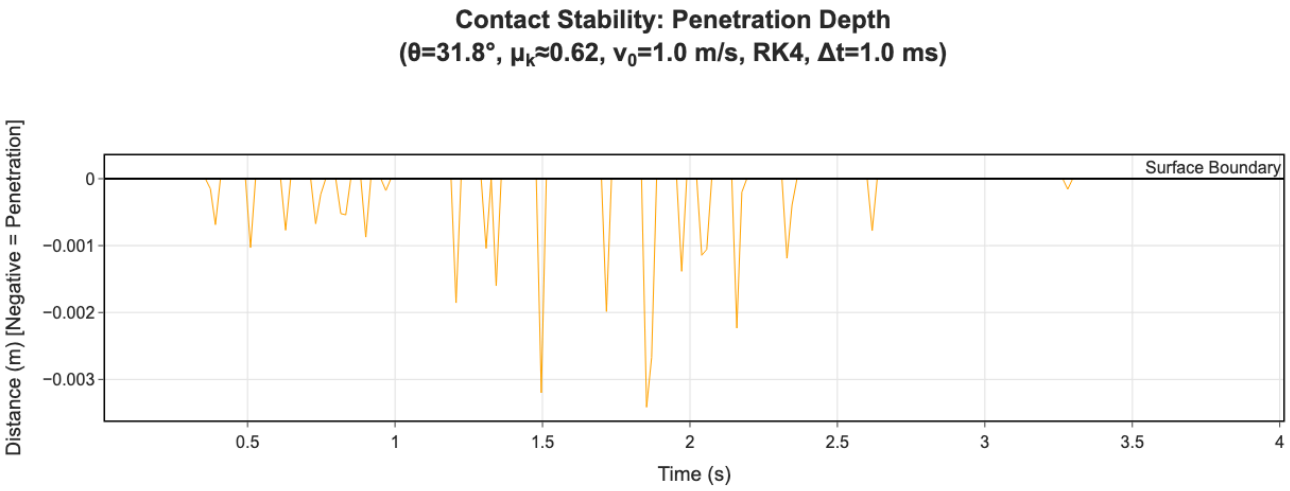
This plot shows the **maximum penetration depth** per timestep (most negative contact `dist` value).

- Values should remain small and close to zero.
- Large or noisy negative values may indicate inappropriate `solref`, `solimp`, or timestep choices.

This is a useful **numerical health indicator** for contact modeling.

```
if PLOTLY_AVAILABLE and simulation_success and len(penetration_depths) > 0:
    fig_pen = go.Figure()
    fig_pen.add_trace(go.Scatter(
        x=times,
        y=penetration_depths,
        mode="lines",
        name="Max Penetration Depth (Min Dist)",
        line=dict(color="orange", width=1),
    ))
    fig_pen.add_hline(
        y=0,
        line_color="black",
        line_width=2,
        annotation_text="Surface Boundary",
    )

    fig_pen = apply_conference_style(
        fig_pen,
```

```
        f"Contact Stability: Penetration Depth<br>{scenario_label}",
    )
    fig_pen.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="Distance (m) [Negative = Penetration]",
        height=500,
        width=800,
    )
    fig_pen.show()
else:
    print("Plotly not available or insufficient data for penetration depth plot.")
```



26. Full 6D Wrench Breakdown

This two-panel plot shows the **decomposition of the contact wrench** into:

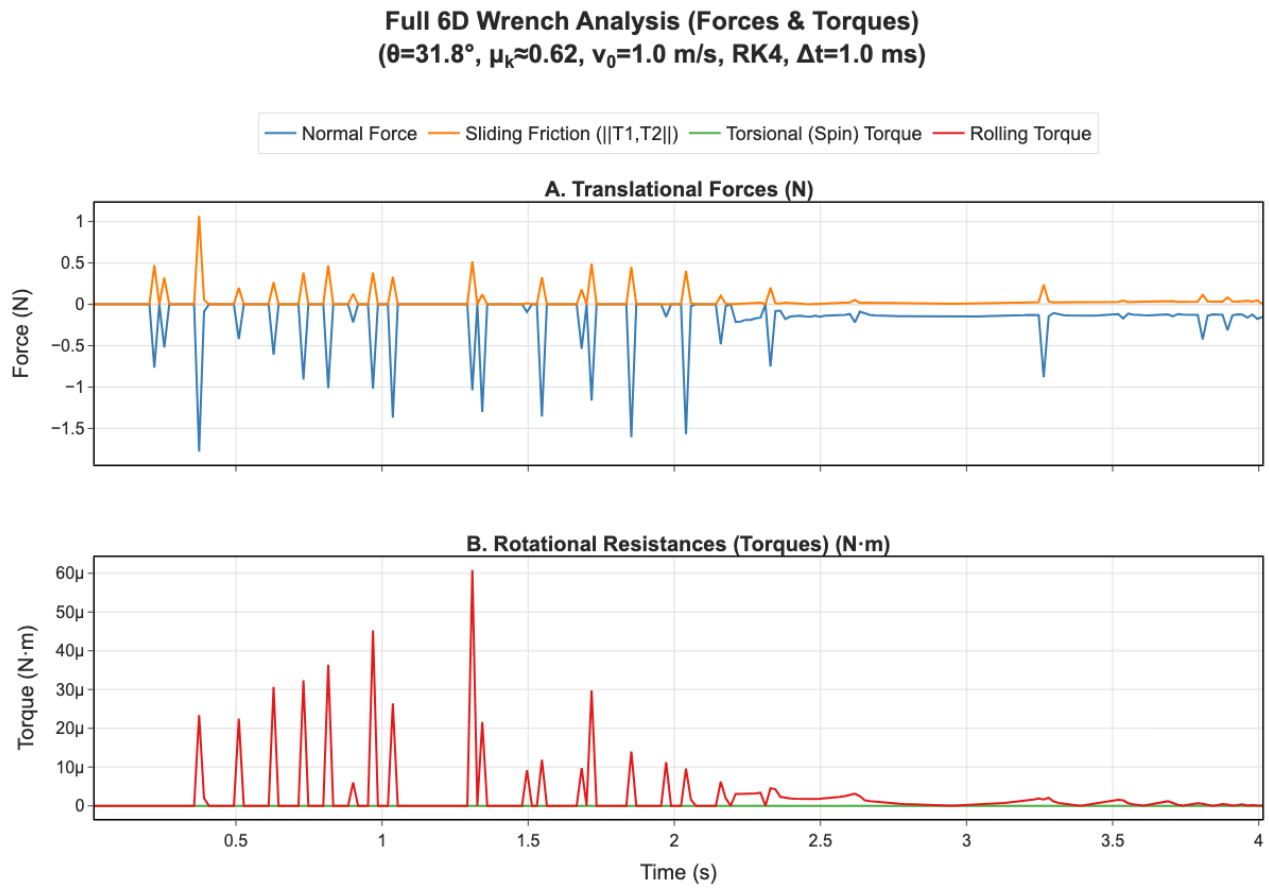
- Translational forces: normal and sliding components.
- Rotational resistances: torsional and rolling torques.

This is central for analyzing **full 6D friction cone behavior**.

```
if PLOTLY_AVAILABLE and simulation_success and len(times) > 1:
    fig_wrench = make_subplots(
        rows=2,
        cols=1,
        shared_xaxes=True,
        vertical_spacing=0.15,
        subplot_titles=(
            "<b>A. Translational Forces (N)</b>",
            "<b>B. Rotational Resistances (Torques) (N·m)</b>",
        ),
    )

    # Row 1: Normal and Sliding
    fig_wrench.add_trace(
        go.Scatter(
            x=times,
```

```
            y=normal_forces,
            name="Normal Force",
            line=dict(color=COLOR_PALETTE[0], width=2),
        ),
        row=1,
        col=1,
    )
    fig_wrench.add_trace(
        go.Scatter(
            x=times,
            y=sliding_forces,
            name="Sliding Friction (||T1,T2||)",
            line=dict(color=COLOR_PALETTE[1], width=2),
        ),
        row=1,
        col=1,
    )

    # Row 2: Torsional and Rolling
    fig_wrench.add_trace(
        go.Scatter(
            x=times,
            y=torsional_torques,
            name="Torsional (Spin) Torque",
            line=dict(color=COLOR_PALETTE[2], width=2),
        ),
        row=2,
        col=1,
    )
    fig_wrench.add_trace(
        go.Scatter(
            x=times,
            y=rolling_torques,
            name="Rolling Torque",
            line=dict(color=COLOR_PALETTE[3], width=2),
        ),
        row=2,
        col=1,
    )

    fig_wrench = apply_conference_style(
        fig_wrench,
        f"Full 6D Wrench Analysis (Forces & Torques)<br>{scenario_label}",
        height=900,
        is_subplot=True,
    )

    fig_wrench.update_yaxes(title_text="Force (N)", row=1, col=1)
    fig_wrench.update_yaxes(title_text="Torque (N·m)", row=2, col=1)
    fig_wrench.update_xaxes(title_text="Time (s)", row=2, col=1)

    fig_wrench.update_layout(
        margin=dict(l=110, r=60, t=190, b=130),
        legend=dict(
            orientation="h",
            yanchor="bottom",
```

```
            y=1.08,
            xanchor="center",
            x=0.5,
            bgcolor="rgba(255,255,255,0.95)",
            bordercolor="#cccccc",
            borderwidth=0.5,
        ),
        title=dict(y=0.97),
    )

    fig_wrench.show()
else:
    print("Plotly not available or insufficient data for 6D wrench breakdown.")
```



27. Integrator Sensitivity: RK4 vs Euler

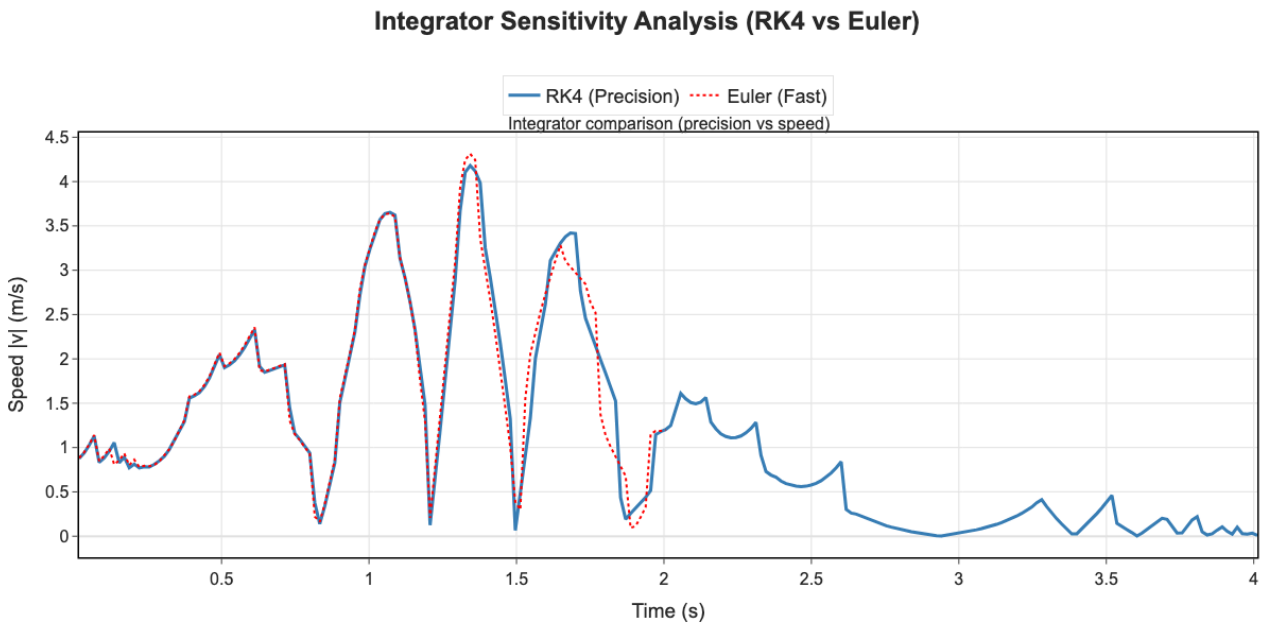
This plot compares the **speed trajectory** obtained with:

- RK4 (higher-order, more accurate)
- Forward Euler (simpler, less accurate)

It helps illustrate the trade-off between **precision and speed** in numerical integration.

```
if PLOTLY_AVAILABLE and simulation_success and results_euler is not None and
isinstance(results_euler, dict) and results_euler.get("times") is not None and
```

```
len(results_euler["times"]) > 0:
    times_euler = results_euler["times"]
    speed_euler = results_euler["speed_mag"]

    fig_sens = go.Figure()
    fig_sens.add_trace(go.Scatter(
        x=times,
        y=speed_mag,
        name="RK4 (Precision)",
        line=dict(width=3, color=COLOR_PALETTE[0]),
    ))
    fig_sens.add_trace(go.Scatter(
        x=times_euler,
        y=speed_euler,
        name="Euler (Fast)",
        line=dict(dash="dot", width=2, color="red"),
    ))

    fig_sens = apply_conference_style(
        fig_sens,
        "Integrator Sensitivity Analysis (RK4 vs Euler)",
    )
    fig_sens.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="Speed |v| (m/s)",
        annotations=[dict(
            x=0.5, y=1.05, xref="paper", yref="paper",
            showarrow=False,
            text="Integrator comparison (precision vs speed)",
        )],
    )
    fig_sens.show()
else:
    print("Plotly not available or Euler comparison not available.")
```

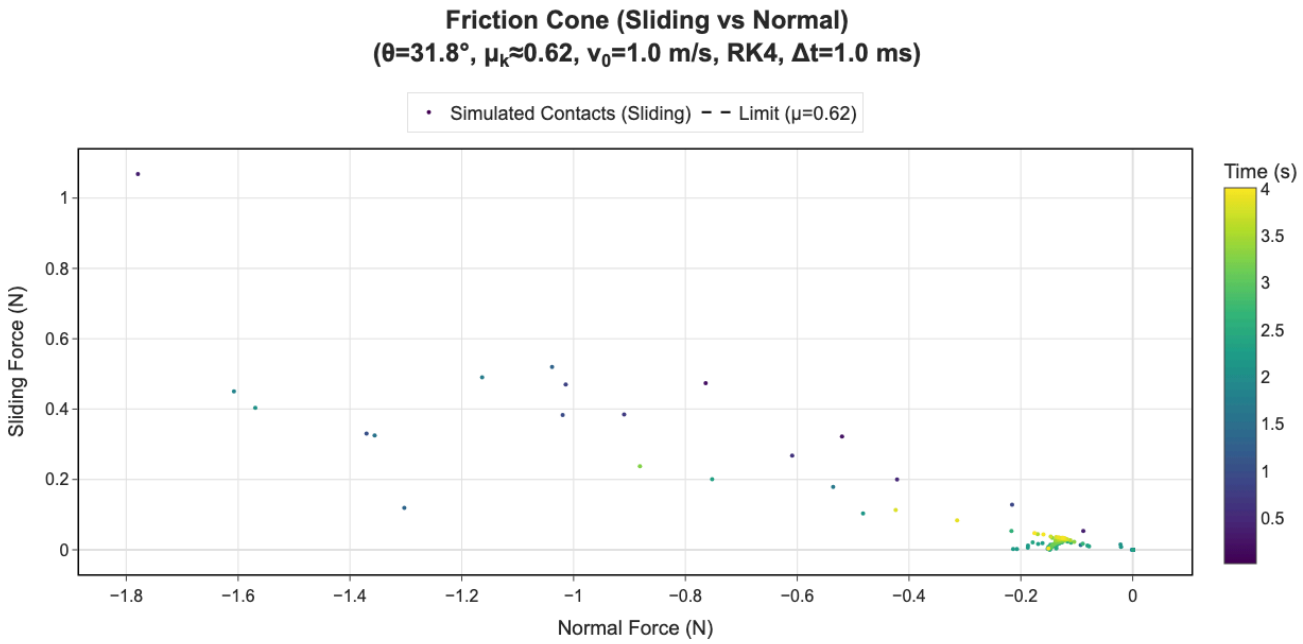


28. Alternative Friction Cone View: Sliding Force vs Normal Force

This variant plots **sliding force magnitude** vs. **normal force**, with points colored by time.

It emphasizes how the **tangential component** of the wrench behaves relative to the normal component and the friction cone limit.

```
if PLOTLY_AVAILABLE and simulation_success and len(normal_forces) > 0:
    fig_cone2 = go.Figure()
    fig_cone2.add_trace(go.Scatter(
        x=normal_forces,
        y=sliding_forces,
        mode="markers",
        name="Simulated Contacts (Sliding)",
        marker=dict(
            size=4,
            color=times,
            colorscale="Viridis",
            showscale=True,
            colorbar=dict(title="Time (s)"),
        ),
    ))
    max_N2 = np.max(normal_forces) if len(normal_forces) > 0 else 1.0
    x_line2 = np.linspace(0, max_N2, 50)
    y_line2 = mu_slide * x_line2
    fig_cone2.add_trace(go.Scatter(
        x=x_line2,
        y=y_line2,
        mode="lines",
        name=f"Limit (μ={mu_slide:.2f})",
        line=dict(dash="dash", color="black"),
    ))

    fig_cone2 = apply_conference_style(
        fig_cone2,
        f"Friction Cone (Sliding vs Normal)<br>{scenario_label}",
    )
    fig_cone2.update_layout(
        xaxis_title="Normal Force (N)",
        yaxis_title="Sliding Force (N)",
    )
    fig_cone2.show()
else:
    print("Plotly not available or insufficient data for alternative friction cone
plot.")
```



29. Global Contact Wrench Components (World Frame)

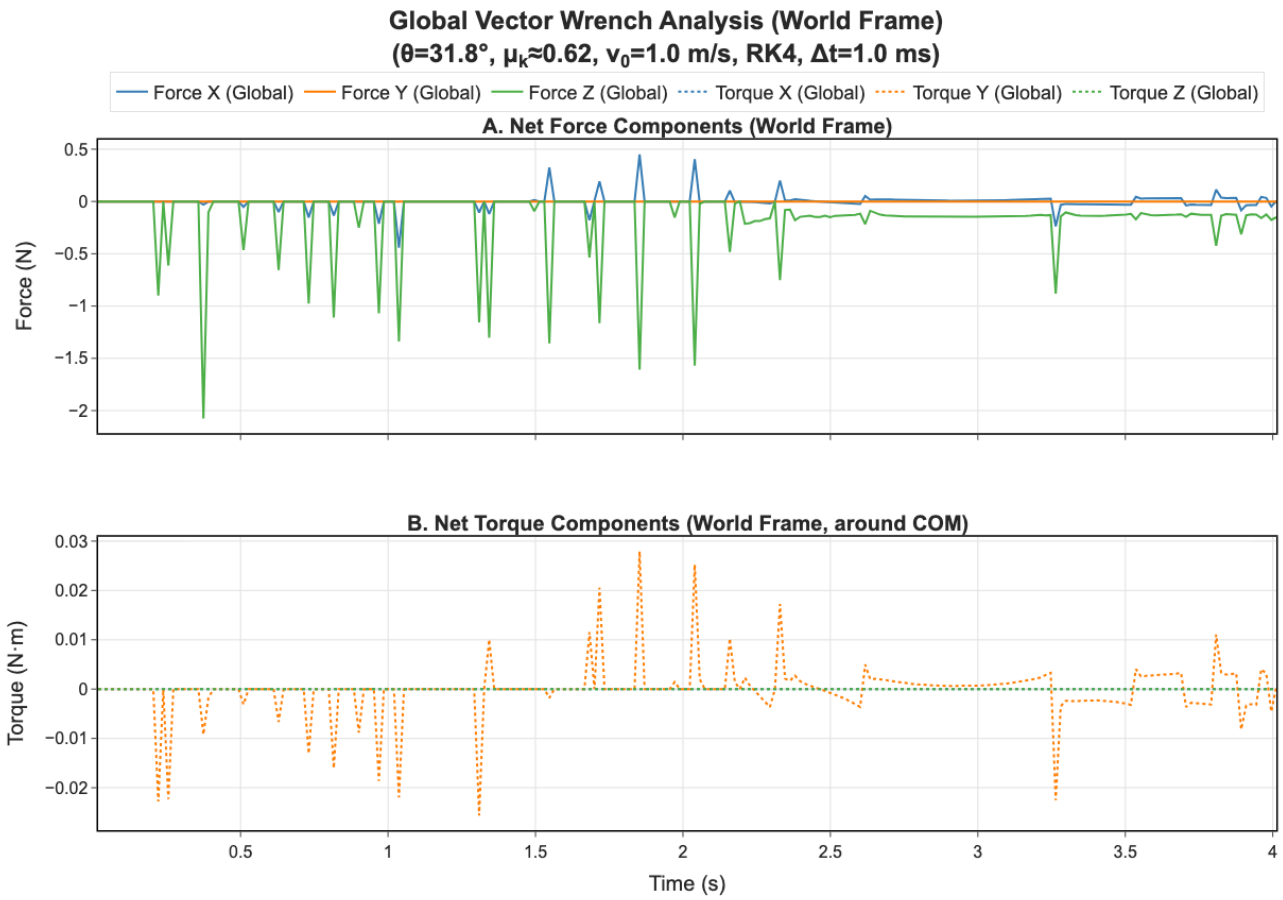
This two-panel plot shows components of the **net contact force** and **net contact torque** in the **world frame**:

- Force components (F_x, F_y, F_z)
- Torque components (τ_x, τ_y, τ_z)

It connects contact behavior to **global motion and equilibrium**.

```
if PLOTLY_AVAILABLE and simulation_success and net_force_world.size and
net_torque_world.size:
    fig_vec = make_subplots(
        rows=2, cols=1, shared_xaxes=True, vertical_spacing=0.15,
        subplot_titles=(
            "<b>A. Net Force Components (World Frame)</b>",
            "<b>B. Net Torque Components (World Frame, around COM)</b>"
        )
    )

    # Force components
    fig_vec.add_trace(go.Scatter(
        x=times, y=net_force_world[:, 0],
        name="Force X (Global)",
        line=dict(color=COLOR_PALETTE[0], width=2),
    ), row=1, col=1)
    fig_vec.add_trace(go.Scatter(
        x=times, y=net_force_world[:, 1],
        name="Force Y (Global)",
        line=dict(color=COLOR_PALETTE[1], width=2),
    ), row=1, col=1)
```

```
    fig_vec.add_trace(go.Scatter(
        x=times, y=net_force_world[:, 2],
        name="Force Z (Global)",
        line=dict(color=COLOR_PALETTE[2], width=2),
    ), row=1, col=1)

    # Torque components
    fig_vec.add_trace(go.Scatter(
        x=times, y=net_torque_world[:, 0],
        name="Torque X (Global)",
        line=dict(color=COLOR_PALETTE[0], dash='dot', width=2),
    ), row=2, col=1)
    fig_vec.add_trace(go.Scatter(
        x=times, y=net_torque_world[:, 1],
        name="Torque Y (Global)",
        line=dict(color=COLOR_PALETTE[1], dash='dot', width=2),
    ), row=2, col=1)
    fig_vec.add_trace(go.Scatter(
        x=times, y=net_torque_world[:, 2],
        name="Torque Z (Global)",
        line=dict(color=COLOR_PALETTE[2], dash='dot', width=2),
    ), row=2, col=1)

    fig_vec = apply_conference_style(
        fig_vec,
        f"Global Vector Wrench Analysis (World Frame)<br>{scenario_label}",
        height=900,
        is_subplot=True,
    )
    fig_vec.update_yaxes(title_text="Force (N)", row=1, col=1)
    fig_vec.update_yaxes(title_text="Torque (N·m)", row=2, col=1)
    fig_vec.update_xaxes(title_text="Time (s)", row=2, col=1)
    fig_vec.show()
else:
    print("Plotly not available or insufficient data for global wrench plot.")
```



30. Parameter Sensitivity Sweeps

The following cells run and plot **parameter sweeps** to study numerical sensitivity:

1. **Contact stiffness (`solref`)** sweep
2. **Timestep (`Δt`)** sweep
3. **Constraint impedance (`solimp`)** sweep

Each sweep runs multiple simulations and compares metrics such as:

- Normal force noise and stability.
- Z-height trajectories and penetration depths.
- Lateral forces and contact topology (number of active contacts).

> These sweeps can be computationally expensive. Run them when you want deeper numerical insight.

30.1 Sweep Over Contact Stiffness (`solref`)

This cell runs several simulations with different `solref` settings:

- **Standard (Stiff)** – baseline, high stiffness, critically damped.
- **Compliant (Soft)** – lower stiffness, softer contacts.
- **Overdamped** – high stiffness but heavily damped.

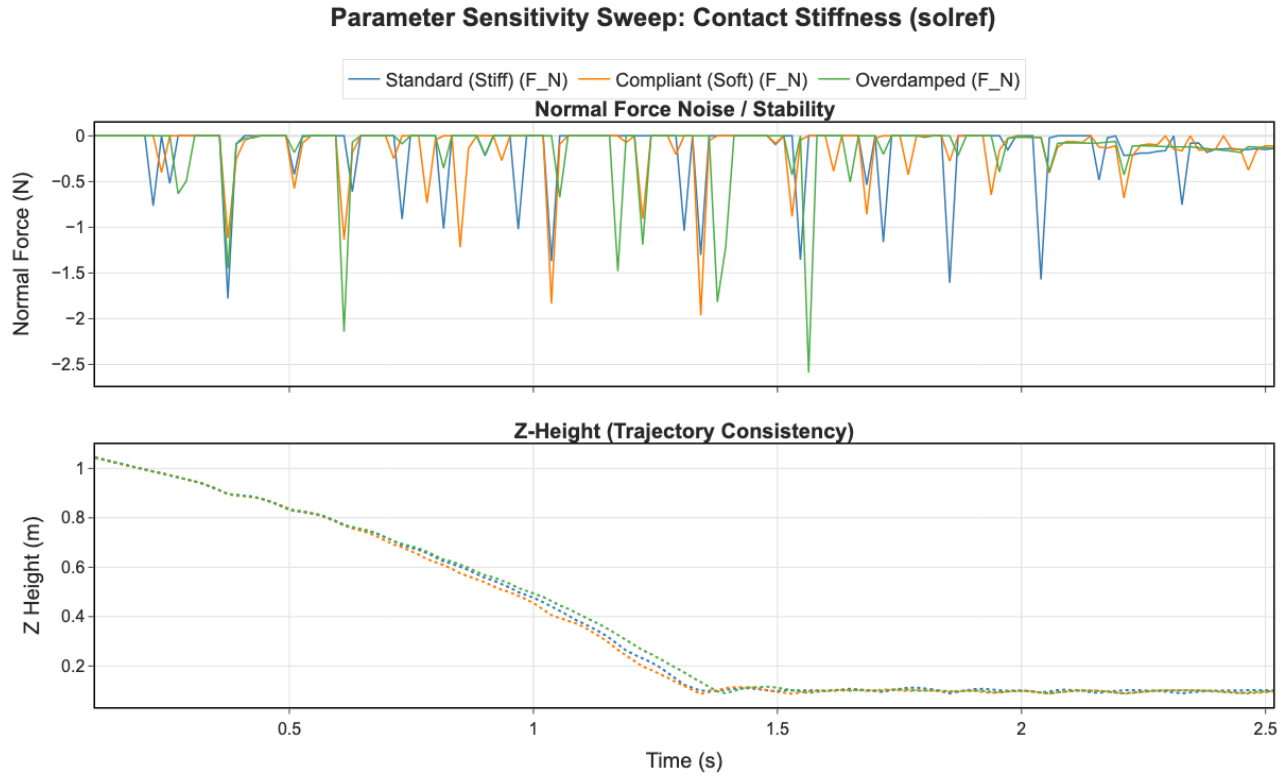
We then compare **normal force noise**, **Z-height consistency**, and **lateral drift forces**.

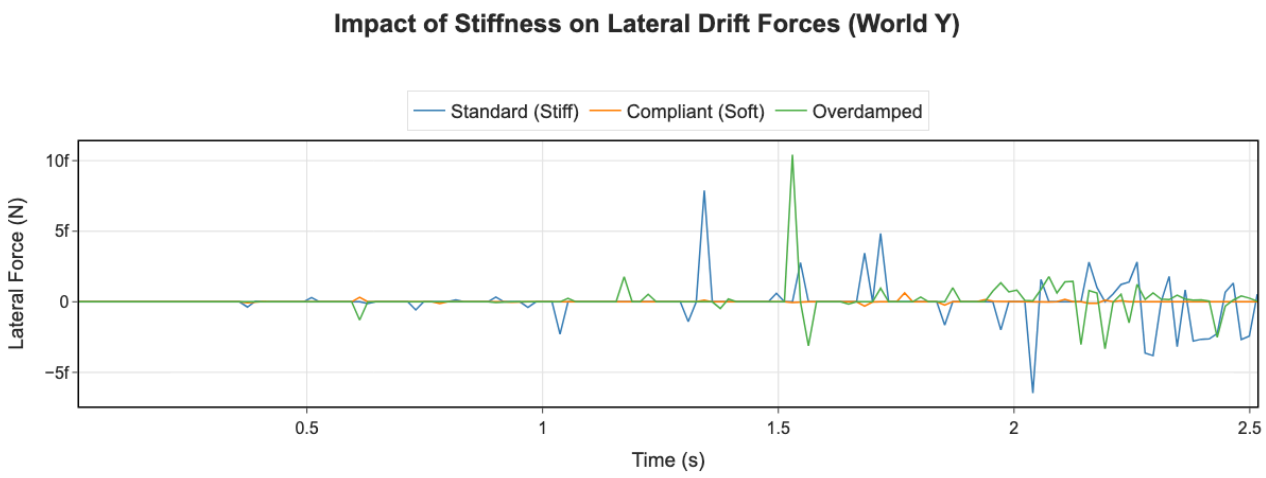
```
if MUJOCO_AVAILABLE and PLOTLY_AVAILABLE and simulation_success:
    variations = [
        ("Standard (Stiff)", "0.001 1"),    # Default: high stiffness, critically
damped
        ("Compliant (Soft)", "0.02 1"),     # Softer normal, critically damped
        ("Overdamped", "0.001 4"),          # Stiff but heavily damped
    ]

    sweep_results = []

    print(f"\nStarting Solref Sensitivity Sweep ({len(variations)} variations)...")

    for label, solref_val in variations:
        print(f"    Running: {label} [solref='{solref_val}']...")
        res_sweep = run_sim(
            theta_deg=THETA_DEG,
            mu_static=MU_STATIC,
            total_mass=TOTAL_MASS,
            n_links=N_LINKS,
            duration=SWEEP_DURATION,
            fps=FPS,
            initial_speed=INITIAL_SPEED,
            integrator=DEFAULT_INTEGRATOR,
            timestep=DEFAULT_TIMESTEP,
            solref_override=solref_val,
            solimp_override=DEFAULT_SOLIMP, # Ensure solimp default is passed
            enable_contact_debug=False,
            record_video=False,
        )
        if res_sweep:
            res_sweep["label"] = label
            sweep_results.append(res_sweep)

    print("Solref Sweep complete.")

    # Plot 1: Normal force & Z-height comparison
    fig_sweep = make_subplots(
        rows=2, cols=1,
        shared_xaxes=True,
        vertical_spacing=0.1,
        subplot_titles=(
            "<b>Normal Force Noise / Stability</b>",
            "<b>Z-Height (Trajectory Consistency)</b>",
        ),
    )

    for i, res_s in enumerate(sweep_results):
        times_s = res_s.get("times", np.array([]))
        if times_s.size == 0:
```

```
            continue
        mask = times_s > 0.1
        if not np.any(mask):
            mask = slice(None)

        color = COLOR_PALETTE[i % len(COLOR_PALETTE)]

        # Normal force
        fig_sweep.add_trace(go.Scatter(
            x=times_s[mask],
            y=res_s["normal_forces"][mask],
            mode='lines',
            name=f"{res_s['label']} (F_N)",
            line=dict(width=1.5, color=color),
        ), row=1, col=1)

        # Z height
        fig_sweep.add_trace(go.Scatter(
            x=times_s[mask],
            y=res_s["z_heights"][mask],
            mode='lines',
            name=f"{res_s['label']} (Z)",
            line=dict(width=2, dash='dot', color=color),
            showlegend=False,
        ), row=2, col=1)

    fig_sweep = apply_conference_style(
        fig_sweep,
        "Parameter Sensitivity Sweep: Contact Stiffness (solref)",
        height=800,
        width=1000,
        is_subplot=True,
    )
    fig_sweep.update_yaxes(title_text="Normal Force (N)", row=1, col=1)
    fig_sweep.update_yaxes(title_text="Z Height (m)", row=2, col=1)
    fig_sweep.update_xaxes(title_text="Time (s)", row=2, col=1)
    fig_sweep.show()

    # Plot 2: Lateral drift forces (World Y) vs time for each variation
    fig_drift_force = go.Figure()
    for i, res_s in enumerate(sweep_results):
        times_s = res_s.get("times", np.array([]))
        nf = res_s.get("net_force_world", np.zeros((0, 3)))
        if times_s.size == 0 or nf.shape[0] == 0:
            continue
        color = COLOR_PALETTE[i % len(COLOR_PALETTE)]
        fig_drift_force.add_trace(go.Scatter(
            x=times_s,
            y=nf[:, 1],  # World Y component (lateral)
            mode='lines',
            name=res_s["label"],
            line=dict(width=1.5, color=color),
        ))

    fig_drift_force = apply_conference_style(
        fig_drift_force,
```

```
        "Impact of Stiffness on Lateral Drift Forces (World Y)",
        height=500,
        width=1000,
    )
    fig_drift_force.update_layout(
        xaxis_title="Time (s)",
        yaxis_title="Lateral Force (N)",
    )
    fig_drift_force.show()

else:
    print("Solref sweep skipped: MuJoCo/Plotly unavailable or base simulation
failed.")
```

```
Starting Solref Sensitivity Sweep (3 variations)...
    Running: Standard (Stiff) [solref='0.001 1']...
    Running: Compliant (Soft) [solref='0.02 1']...
    Running: Overdamped [solref='0.001 4']...
Solref Sweep complete.
```





30.2 Contact Topology Analysis (Contact Count Distribution)

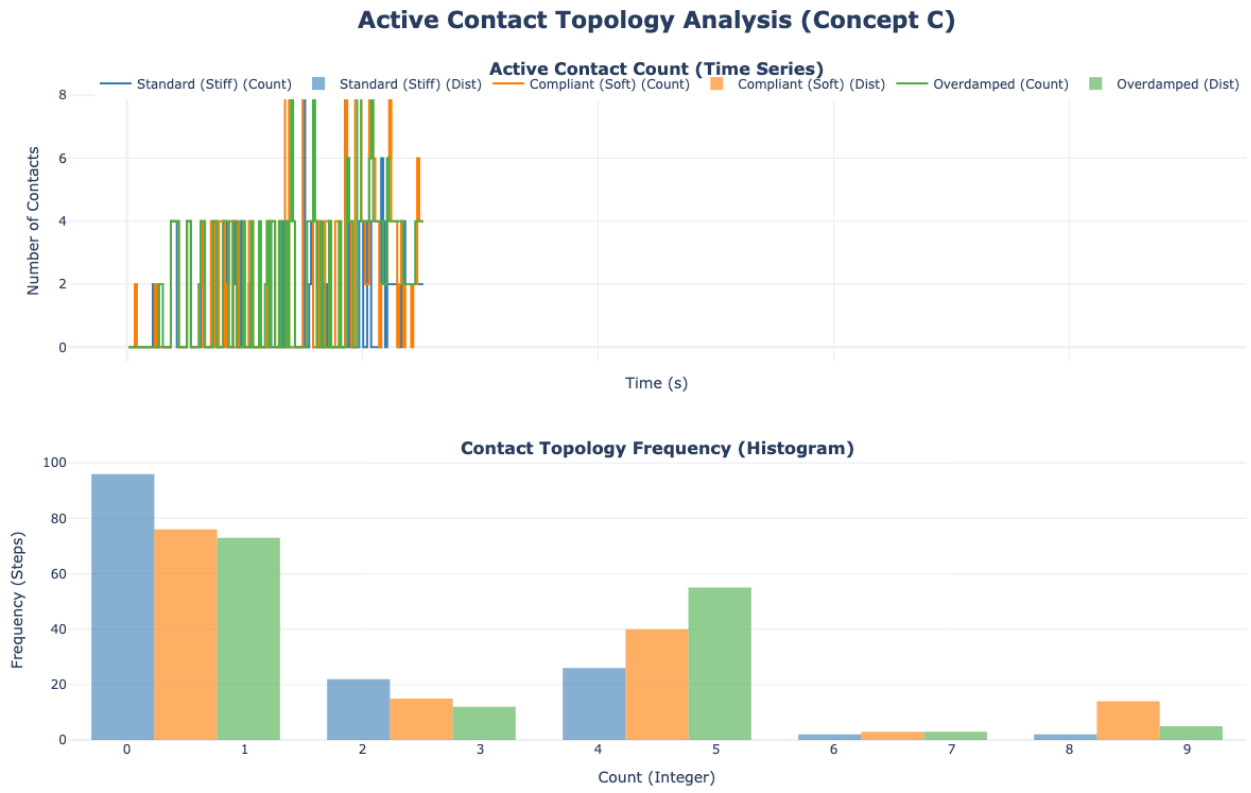
Using the results of the `solref` sweep, this plot shows:

- The **time series of active contact count**.
- A **histogram of how often each contact count occurs**.

This quantifies how the contact network topology changes with stiffness.

```
# This cell reuses 'sweep_results' from the previous sweep if available
if PLOTLY_AVAILABLE and 'sweep_results' in globals() and sweep_results:
    fig_counts = make_subplots(
        rows=2, cols=1,
        shared_xaxes=True,
        vertical_spacing=0.15,
        subplot_titles=(
            "<b>Active Contact Count (Time Series)</b>",
            "<b>Contact Topology Frequency (Histogram)</b>",
        ),
    )

    for i, res_s in enumerate(sweep_results):
        times_s = res_s.get("times", np.array([]))
        counts = res_s.get("contact_counts", np.array([]))
        if times_s.size == 0 or counts.size == 0:
            continue
        color = COLOR_PALETTE[i % len(COLOR_PALETTE)]

        # Time series (step plot)
        fig_counts.add_trace(go.Scatter(
            x=times_s,
            y=counts,
            mode='lines',
            line_shape='hv',
            name=f"{res_s['label']} (Count)",
            line=dict(width=2, color=color),
        ), row=1, col=1)
```

```
        # Histogram of counts
        fig_counts.add_trace(go.Histogram(
            x=counts,
            name=f"{res_s['label']} (Dist)",
            marker_color=color,
            opacity=0.6,
            bingroup=1,
        ), row=2, col=1)

    fig_counts.update_layout(
        title=dict(
            text="<b>Active Contact Topology Analysis (Concept C)</b>",
            x=0.5,
            font=dict(size=22),
        ),
        template=PLOT_TEMPLATE if PLOTLY_AVAILABLE else None,
        height=800,
        width=1000,
        barmode='group',
        legend=dict(orientation="h", y=1.02, x=1, xanchor="right"),
    )
    fig_counts.update_yaxes(title_text="Number of Contacts", row=1, col=1)
    fig_counts.update_yaxes(title_text="Frequency (Steps)", row=2, col=1)
    fig_counts.update_xaxes(title_text="Time (s)", row=1, col=1)
    fig_counts.update_xaxes(title_text="Count (Integer)", row=2, col=1, dtick=1)
    fig_counts.show()
else:
    print("Contact topology plot skipped: no sweep_results found or Plotly
unavailable.")
```



30.3 Sweep Over Timestep (Δt)

This sweep explores the effect of **timestep size** on:

- Energy conservation (drift of total mechanical energy).
- Constraint quality (penetration depth).

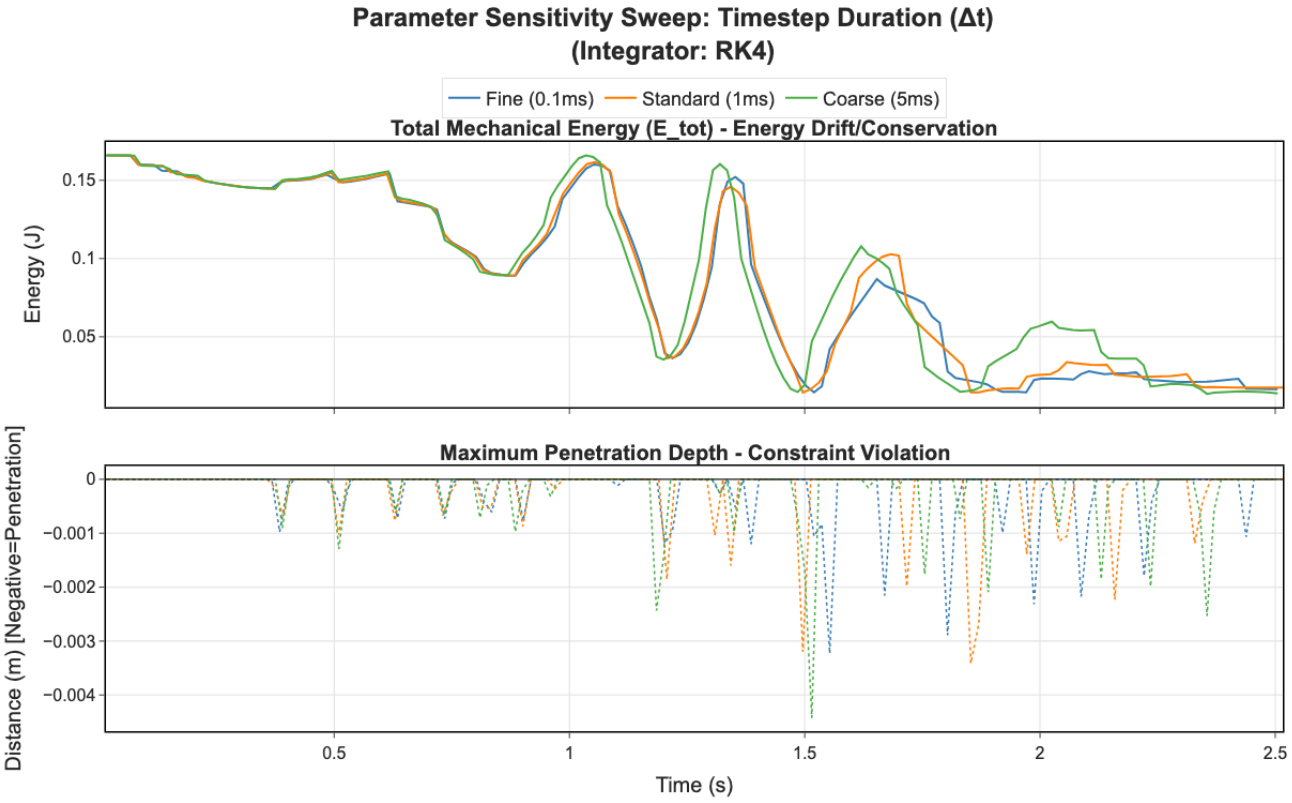
We compare “fine”, “standard”, and “coarse” timesteps to highlight numerical trade-offs.

```
if MUJOCO_AVAILABLE and PLOTLY_AVAILABLE and simulation_success:
    timestep_variations = [
        ("Fine (0.1ms)", 0.0001),
        ("Standard (1ms)", 0.001),
        ("Coarse (5ms)", 0.005),
    ]

    ts_sweep_results = []
    print(f"\nStarting Timestep Sensitivity Sweep ({len(timestep_variations)}
variations)...")

    for label, ts_val in timestep_variations:
        print(f"    Running: {label} [Δt={ts_val*1000:.1f} ms]...")
        res_sweep = run_sim(
            theta_deg=THETA_DEG,
            mu_static=MU_STATIC,
            total_mass=TOTAL_MASS,
            n_links=N_LINKS,
            duration=SWEEP_DURATION,
            fps=FPS,
            initial_speed=INITIAL_SPEED,
            integrator=DEFAULT_INTEGRATOR,  # Keep RK4 for consistency
            timestep=ts_val,                # Vary the timestep
            solref_override=DEFAULT_SOLREF,
            solimp_override=DEFAULT_SOLIMP,
            enable_contact_debug=False,
            record_video=False,
        )
        if res_sweep:
            res_sweep["label"] = label
            ts_sweep_results.append(res_sweep)

    print("Timestep Sweep complete.")

    if ts_sweep_results:
        # Plot: Energy conservation and Penetration depth comparison
        fig_ts_sweep = make_subplots(
            rows=2, cols=1,
            shared_xaxes=True,
            vertical_spacing=0.1,
            subplot_titles=(
                "<b>Total Mechanical Energy (E_tot) - Energy
Drift/Conservation</b>",
                "<b>Maximum Penetration Depth - Constraint Violation</b>",
```

```
            ),
        )

        for i, res_s in enumerate(ts_sweep_results):
            times_s = res_s.get("times", np.array([]))
            if times_s.size == 0:
                continue

            color = COLOR_PALETTE[i % len(COLOR_PALETTE)]

            # Calculate Energy (must be done per result set)
            E_pot_s = res_s["total_mass"] * G * res_s["z_heights"]
            E_kin_s = 0.5 * res_s["total_mass"] * res_s["speed_mag"]**2
            E_tot_s = E_pot_s + E_kin_s

            # Energy
            fig_ts_sweep.add_trace(go.Scatter(
                x=times_s,
                y=E_tot_s,
                mode='lines',
                name=f"{res_s['label']}",
                line=dict(width=2, color=color),
            ), row=1, col=1)

            # Penetration
            fig_ts_sweep.add_trace(go.Scatter(
                x=times_s,
                y=res_s["penetration_depths"],
                mode='lines',
                name=f"{res_s['label']} (Penetration)",
                line=dict(width=1.5, dash='dot', color=color),
                showlegend=False,
            ), row=2, col=1)

        # Add surface boundary line
        fig_ts_sweep.add_hline(
            y=0,
            line_color="black",
            line_width=1,
            row=2,
            col=1
        )

        fig_ts_sweep = apply_conference_style(
            fig_ts_sweep,
            f"Parameter Sensitivity Sweep: Timestep Duration (Δt)<br>(Integrator:
{DEFAULT_INTEGRATOR})",
            height=800,
            width=1000,
            is_subplot=True,
        )
        fig_ts_sweep.update_yaxes(title_text="Energy (J)", row=1, col=1)
        fig_ts_sweep.update_yaxes(title_text="Distance (m) [Negative=Penetration]",
row=2, col=1)
        fig_ts_sweep.update_xaxes(title_text="Time (s)", row=2, col=1)
        fig_ts_sweep.show()
```

```
        print("Timestep Sensitivity Visualizations Generated.")
else:
    print("Timestep sweep skipped: MuJoCo/Plotly unavailable or base simulation
failed.")
```

```
Starting Timestep Sensitivity Sweep (3 variations)...
    Running: Fine (0.1ms) [Δt=0.1 ms]...
    Running: Standard (1ms) [Δt=1.0 ms]...
    Running: Coarse (5ms) [Δt=5.0 ms]...
Timestep Sweep complete.
```




```
Timestep Sensitivity Visualizations Generated.
```

30.4 Sweep Over Constraint Impedance (`solimp`)

Finally, we vary `solimp` parameters to explore the effect of **contact hardness vs softness**.

We compare:

- **Reference (Hard)** – sharp, stiff contacts.
- **Soft Constraint** – softer, more compliant contacts.
- **Wide Margin** – contacts activate over a wider distance interval.

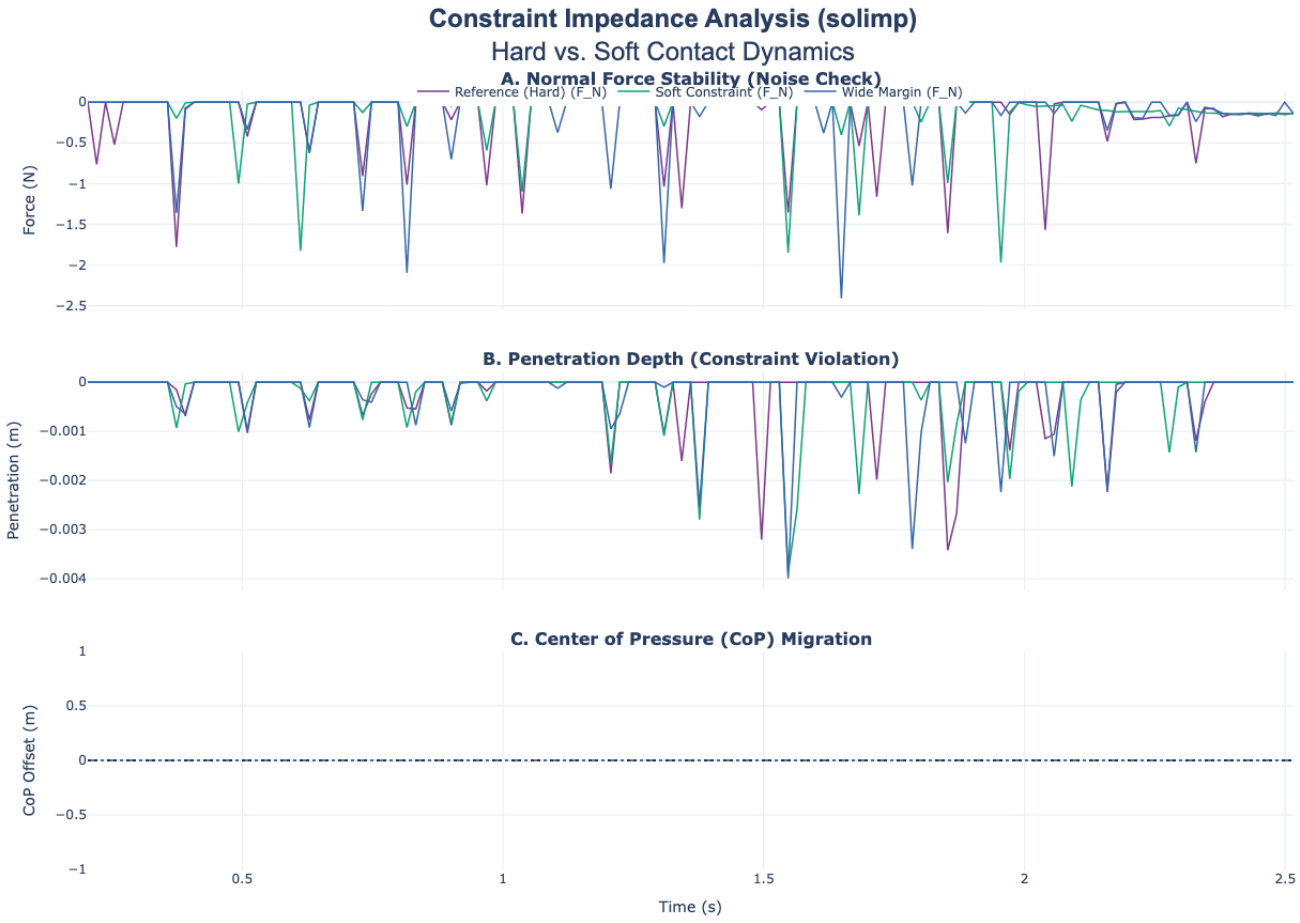
We examine **normal force stability**, **penetration depth**, and **CoP migration** for each setting.

```python
if MUJOCO_AVAILABLE and PLOTLY_AVAILABLE and simulation_success:
    solimp_variations = [
        ("Reference (Hard)", "0.99 0.99 0.001"), # Sharp, hard contact
        ("Soft Constraint",  "0.90 0.95 0.01"),  # Lower impedance: softer reaction
forces
        ("Wide Margin",      "0.99 0.99 0.05"),  # Wide transition: force applies
earlier
    ]

    solimp_results = []
    print(f"\nStarting Constraint Impedance (solimp) Sweep ({len(solimp_variations)}
variations)...")

    for label, solimp_val in solimp_variations:
        print(f"    Running: {label} [solimp='{solimp_val}']...")

        # Run simulation with the override
        res_sweep = run_sim(
            theta_deg=THETA_DEG,
            mu_static=MU_STATIC,
            total_mass=TOTAL_MASS,
            n_links=N_LINKS,
            duration=2.5,                # Short duration sufficient for stability
check
            fps=FPS,
            initial_speed=INITIAL_SPEED,
            integrator=DEFAULT_INTEGRATOR,
            timestep=DEFAULT_TIMESTEP,
            solref_override=DEFAULT_SOLREF,
            solimp_override=solimp_val,
            enable_contact_debug=False,
            record_video=False,
        )

        if res_sweep:
            res_sweep["label"] = label
            solimp_results.append(res_sweep)

    print("Solimp Sweep complete.")

    # --- Plotting the Comparison ---
    if solimp_results:
        fig_imp = make_subplots(
            rows=3, cols=1,
            shared_xaxes=True,
            vertical_spacing=0.08,
            subplot_titles=(
                "<b>A. Normal Force Stability (Noise Check)</b>",
                "<b>B. Penetration Depth (Constraint Violation)</b>",
                "<b>C. Center of Pressure (CoP) Migration</b>"
            ),
        )

        for i, res in enumerate(solimp_results):
            # Extract data
```

```
            t = res.get("times", np.array([]))
            if t.size == 0:
                continue

            # Mask startup transients (first 0.2s) for clearer viewing
            mask = t > 0.2

            # Use distinct colors from a qualitative palette
            color = pc.qualitative.Bold[i % len(pc.qualitative.Bold)]

            # 1. Normal Force (Checking for numerical explosions/jitters)
            fig_imp.add_trace(go.Scatter(
                x=t[mask], y=res["normal_forces"][mask],
                mode='lines',
                name=f"{res['label']} (F_N)",
                legendgroup=res['label'],
                line=dict(width=1.5, color=color),
            ), row=1, col=1)

            # 2. Penetration Depth (Checking how 'spongy' the contact is)
            fig_imp.add_trace(go.Scatter(
                x=t[mask], y=res["penetration_depths"][mask],
                mode='lines',
                name=f"{res['label']} (Depth)",
                legendgroup=res['label'],
                showlegend=False,
                line=dict(width=1.5, dash='solid', color=color),
            ), row=2, col=1)

            # 3. CoP Migration (Checking if soft contacts blur the pressure point)
            fig_imp.add_trace(go.Scatter(
                x=t[mask], y=res["cop_ramp_history"][mask],
                mode='lines',
                name=f"{res['label']} (CoP)",
                legendgroup=res['label'],
                showlegend=False,
                line=dict(width=2, dash='dot', color=color),
            ), row=3, col=1)

        # Add Zero-line for CoP (Geometric Center)
        fig_imp.add_hline(y=0, row=3, col=1, line_dash="dash", line_color="black",
opacity=0.5)

        # Apply Styling
        fig_imp.update_layout(
            title=dict(
                text="<b>Constraint Impedance Analysis (solimp)</b><br>Hard vs. Soft
Contact Dynamics",
                font=dict(size=24, family="Arial"),
                x=0.5
            ),
            template=PLOT_TEMPLATE,
            height=900,
            width=1000,
            legend=dict(orientation="h", y=1.02, x=0.5, xanchor="center"),
            margin=dict(t=120, b=50, l=80, r=50),
```

```
        )

        fig_imp.update_yaxes(title_text="Force (N)", row=1, col=1)
        fig_imp.update_yaxes(title_text="Penetration (m)", row=2, col=1)
        fig_imp.update_yaxes(title_text="CoP Offset (m)", row=3, col=1)
        fig_imp.update_xaxes(title_text="Time (s)", row=3, col=1)

        fig_imp.show()
else:
    print("Solimp sweep skipped: MuJoCo/Plotly unavailable or base simulation failed.")
```

```
Starting Constraint Impedance (solimp) Sweep (3 variations)...
    Running: Reference (Hard) [solimp='0.99 0.99 0.001']...
    Running: Soft Constraint [solimp='0.90 0.95 0.01']...
    Running: Wide Margin [solimp='0.99 0.99 0.05']...
Solimp Sweep complete.
```



Discussion

1. What could you have done better in your experiment design and setup? Although the Python script focuses on the simulation side, it implicitly reveals some limitations of the physical experiment and data workflow that could be improved:

- Repeatability and uncertainty reporting In the code, the critical angle is treated as a single fixed value (THETA_DEG = 31.8) and the static friction coefficient is hard-coded (MU_STATIC = 0.62). There is no notion of multiple trials, variance, or measurement uncertainty. A better experiment design would:
  - Repeat the critical-angle measurement many times.
  - Report a mean and standard deviation for both angle and μ_s.
  - Propagate that uncertainty into the model.
- Direct linkage between experimental data and simulation The script simulates sliding with MuJoCo but does not load any experimental trajectory from a CSV/Tracker file. A stronger design would:

  ```
  - Capture position–time data from the real hexagon sliding on the ramp.

  - Import that data into the notebook.

  - Use it directly for validation and parameter identification.
  ```
- Controlled initial conditions In simulation, the initial speed along the ramp is prescribed (INITIAL_SPEED = 1.0). In the physical experiment, the actual initial conditions (exact release position, possible initial push, small misalignments) are not encoded or controlled in the code. Improving the setup would mean:

  ```
  - Marking a consistent start position on the ramp.

  - Releasing the mechanism without extra impulse.

  - Measuring or estimating the actual initial velocity used in the
  experiment.
  ```
- Geometric and material characterization The code assumes the hexagon behaves as a rigid body made of six rigid links with a uniform cardboard material, but no experimental stiffness or compliance data is fed in. A more complete experimental plan would:

  ```
   - Measure or estimate material stiffness / joint compliance.

   - Verify that deformations are small enough that the rigid-body
  assumption is reasonable for this task.
  ```
- Surface characterization The floor/ramp surface in the script is idealized as perfectly planar with a single friction coefficient. In the real setup, local roughness, dust, and wear can change friction. A better experiment would:

  ```
  - Clean and prepare the surface in a controlled way.

  - Possibly test several locations on the ramp and compare friction.
  ```

2. Rationale for the model and assumptions

- A Coulomb friction model was chosen because it is simple, widely used, and directly linked to the critical-angle experiment.
- The hexagon is modeled as a rigid body composed of six rigid links, assuming link compliance and joint flexibility have negligible effect on sliding.
- The ramp is treated as an ideal flat plane with uniform friction and constant angle.
- The simulation neglects secondary effects (air drag, micro-texture, deformation), assuming friction and gravity dominate the motion.
- MuJoCo's elliptic friction cone and soft-contact parameters (solref, solimp) provide stable and realistic contact forces for sliding analysis.

3. Justification of fitting method

- Not applicable — the provided Python script does not perform parameter fitting or optimization.
- Static friction is directly computed from the critical angle, and kinetic friction is assigned analytically (mu_dynamic = tan(theta)), with no least-squares or optimizer used.

4. How well does your data fit the model?

- Numerical Fit:
  - No numerical fitting (MSE/RMSE) was performed in the current script.
  - → Numerical fit value: Not applicable.
- Qualitative Fit:
  - The simulation with $\mu k = \tan(31.8^\circ) \approx 0.62$ produces near-constant sliding speed, which matches the theoretical prediction for balanced gravity and friction.
  - Friction-cone and $\mu\text{eff}(t)$ plots show values staying close to the assigned coefficient.
  - Overall, the Coulomb model matches the expected sliding behavior in simulation.

5. What are the limits of your model, and will the system operate outside them?

- Model Limits:

    - Assumes constant Coulomb friction (no speed or humidity dependence).
    - Treats the mechanism as a rigid body, ignoring cardboard flexibility.
    - Uses an ideal smooth ramp with uniform friction.
    - Ignores rolling resistance and local deformation.
    - No experimental trajectory fitting is performed.
- Operating Outside Limits: Yes—real cardboard sliding involves surface roughness, deformation, humidity changes, and slight rolling, so the real system will deviate from the ideal assumptions. However, the model remains useful for first-order friction estimation and simulation validation.