A PRESENTATION ON “MARS ROVER”

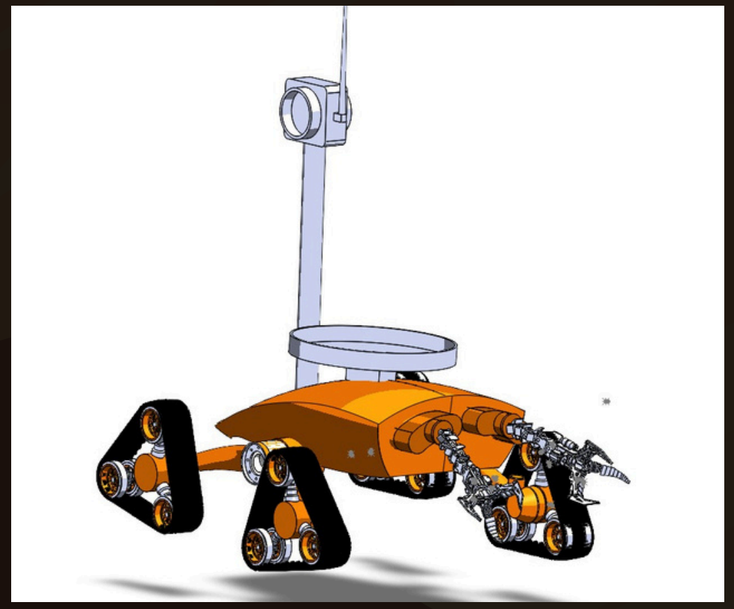
BY

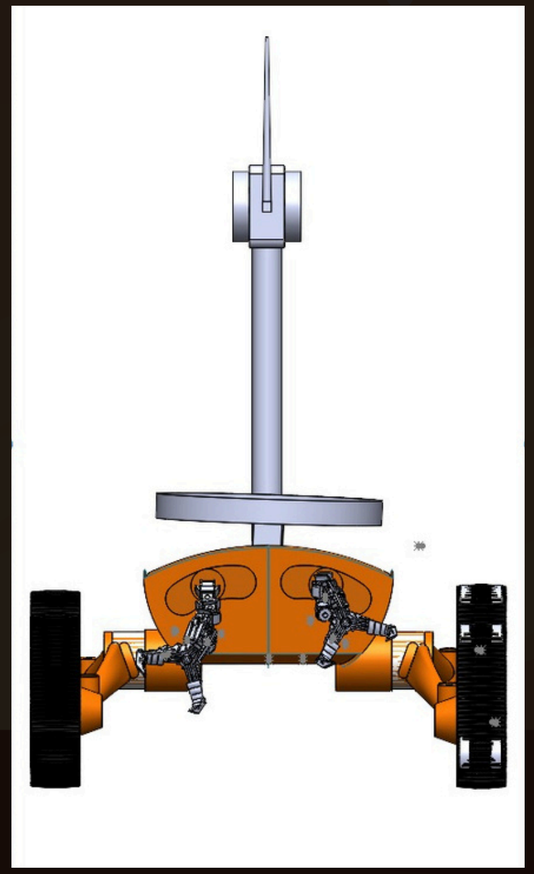
TEAM ENDURANCE

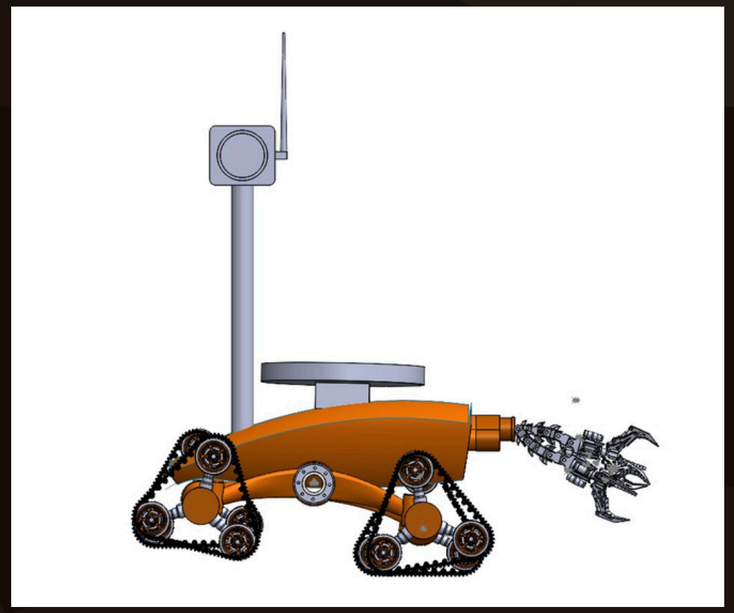
NSSC ‘25

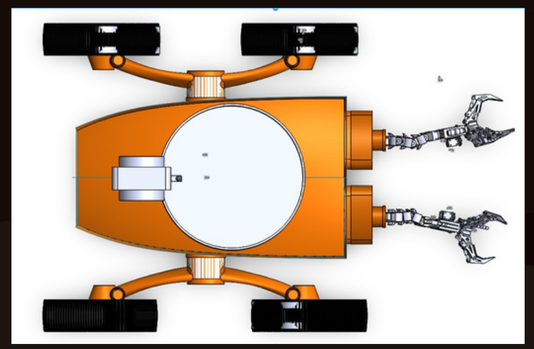
Table of contents

01
Rover Design







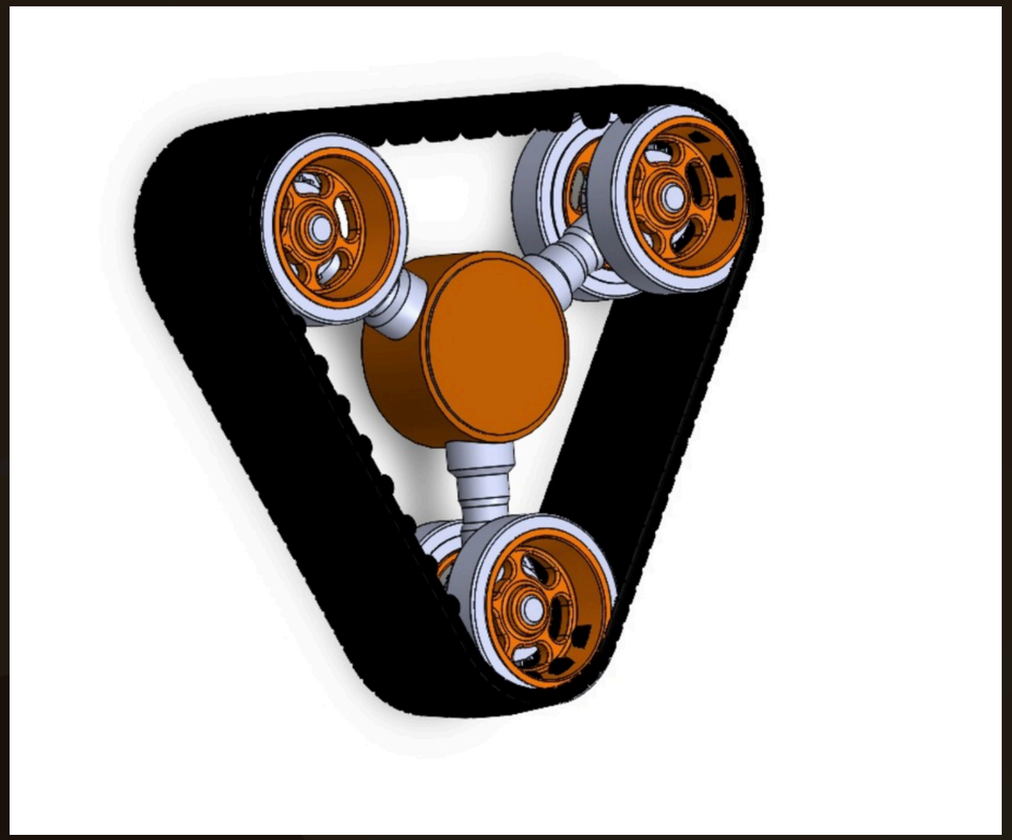


High-level Architecture

- Chassis: central body with mast (camera/antenna) and two front manipulators.
- Locomotion: 4 leg modules, each ending in tri-roller wheel assemblies.
- Manipulators: compact multi-DOF arms on the front.
- Payload: sensor suite, sample container, compute node.
- Power: battery pack + solar assist (optional).

Mechanical Design (In Solidworks):

- Leg/Module Geometry: curved boom arms with rotary joints at the body; tri-roller end gives a combination of wheel and track behaviour.
- Suspension: passive compliance in each leg (spring or elastomer) + limited pitch/yaw joints to keep contact over uneven surfaces.
- Manipulator: 4 DOF feet/arms with small brushless motors, harmonic or planetary gears, and a modular end-effector (gripper + drill adapter).
- Materials (planned): aluminium 6061/Ti for structural parts; carbon-fibre panels for enclosure; UHMW/PU for rollers.



Wheel design

Introduction

Tri-star wheel mechanism in a triangular configuration has been used. These three wheels are mounted on a central rotating hub, which itself can rotate to allow dynamic repositioning.

Working Principle

The tri-star wheel works in two primary modes:

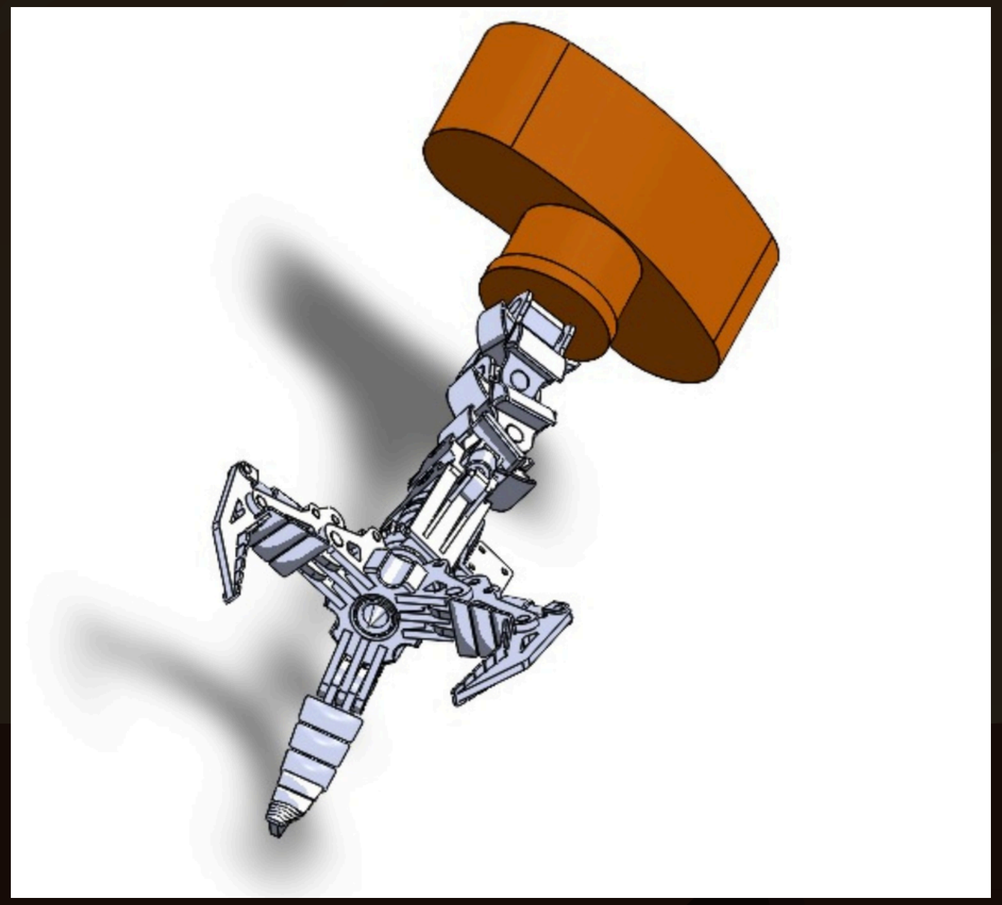
Normal Motion:

The three wheels rotate together, driving the rover smoothly along flat surfaces. It provides continuous contact with the ground, improving traction and weight distribution.

Obstacle-Climbing Mode:

When the assembly encounters a steepobstacle, the central hub rotates, bringing another wheel into contact with the ground allowomg the system to “walk” over the obstacle, maintaining traction without losing balance.

This dual mechanism allows the rover to handle terrain discontinuities that traditional wheels or tracks alone cannot manage.



Arm design

Introduction

A multi-jointed mechanical arm ending in a drilling or sampling mechanism, with an upper actuated rotary base actuator for azimuthal rotation of the arm about the rover’s main body has been used. The arm is typically multi-degree-of-freedom (DOF) for effective sample collection.

Working Principle

The base motor rotates the entire arm assembly horizontally to align with the target zone.

Arm segments providing movement in multiple axes of pitch, roll, yaw, extend or articulate to reach the surface or object.

The end effector (drill or gripper) equipped with a high-torque motor performs the required task of sample collection. Integrated sensors provide feedback to the rover’s control system for precise motion and error correction.

After operation, the arm retracts and locks in its stowed configuration for safe mobility.

02
Electronic Subsystems

Sensors

01 Proximity Sensor

Detects nearby objects without physical contact using signals.

02 Vision Sensor

Captures images to detect, recognize, and analyze objects.

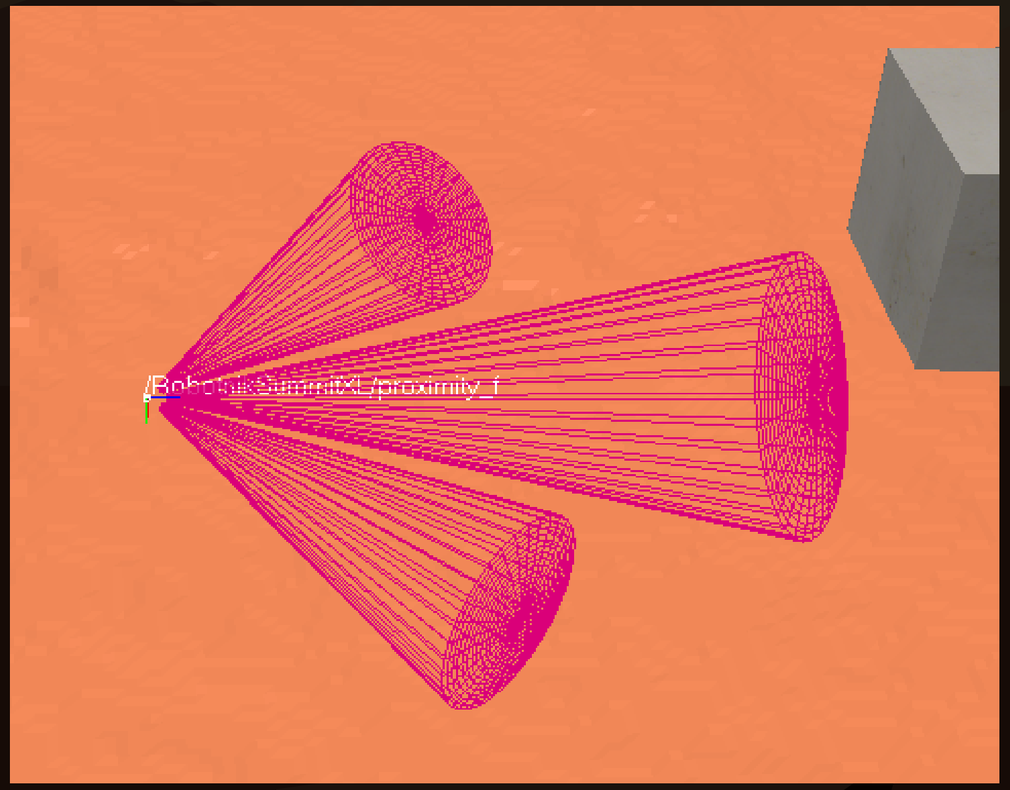
03 Wheel Encoders

Counts wheel rotations to calculate speed and distance

04 IMU

Measures acceleration, rotation, and orientation for motion tracking

Proximity Sensors:



Type:
Simulated cone-type proximity sensors (range/ultrasonic).

Working Principle:
Each sensor emits simulated rays (similar to ultrasonic/range sensors) forming a conical detection field. The returned signal strength or distance to the nearest object is continuously measured. When any sensor reading falls below a safety threshold (≈0.5 m), the rover automatically halts or slows, and executes a local avoidance maneuver (turn, then move forward).

Role in Mapping:
Detected obstacle points are tagged with the rover's current odometry position and added to the occupancy grid, updating both static and dynamic features for re-planning. This enhances local environmental awareness during D* re-planning cycles.

Reason for Not Using LiDAR + Potential Field/D* Lite:
LiDAR-based mapping performs well in structured settings but struggles with noisy or incomplete data in unstructured Martian terrain. Potential field and D* Lite methods also face issues like local minima traps and high tuning requirements. Hence, a simpler multi-sensor array was preferred for real-time, terrain-agnostic obstacle avoidance.

Vision System (Monocular Camera)

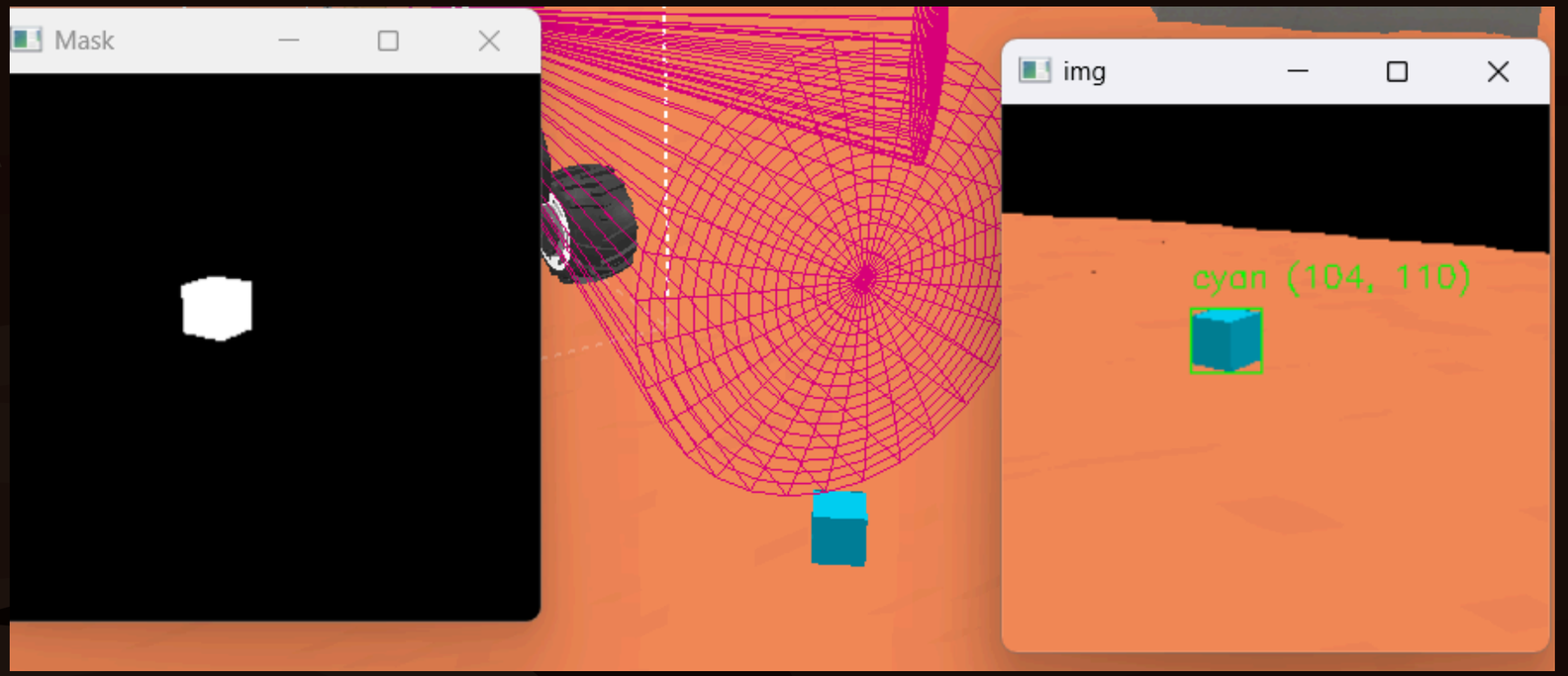
Type: Single vision sensor (simulated monocular camera) mounted on a pan-tilt mast.
Placement: Elevated on the mast with a 20° downward angle for optimal ground visibility.

Working Principle:

Captures image frames of the rover's surroundings in real time. These frames are analyzed in the software layer to identify target samples or landmarks.

Role in Navigation and Mapping:

The camera's field of view is combined with IMU and encoder data to link visual landmarks with positional coordinates, refining both 2D and 3D map accuracy.



03
Source Code

High Level Architecture

Handler Functions

set_wheel_velocities(...)

set_all_wheels(v)

set_wheel_speeds(l,a)

stop_wheels()

get_base_position()

angle_diff(*a*, *b*)

rotate_in_place(...)

Utility Functions

read_all_sensors_world_raw(...)

read_proximity_world(...)

log_current_position(...)

obstacle_avoidance_diff(...)

perform_stuck_recovery(...)

goto_point_dynamic(.....)

Main Script

Initialiize Log function and sensor handles

↓

Go to target coordinates with necessary arguments

↓

Once reached, go back to start coordinates

↓

Once successful, plot the path in 2D and 3D with matplotlib and save log as csv

```python
def goto_point_dynamic(target_xy,
                       forward_speed=2.0, rotate_speed=1.2,
                       dist_tolerance=0.20,
                       log_fn=None):
    """
    Dynamic point-to-point navigation with integrated obstacle avoidance and recovery.
    Combines proportional heading correction, obstacle avoidance, and stuck recovery.
    """

    tx, ty = target_xy
    last_pos = get_base_position()
    last_progress_time = time.time()
    it = 0
    reached_target = False

    while not reached_target:
        it += 1
        if log_fn: log_fn()                             # Log current position if logger provided

        # --- Compute navigation geometry ---
        x, y, z = get_base_position()
        dx, dy = tx - x, ty - y
        dist = math.hypot(dx, dy)                       # Distance to target
        target_yaw = math.atan2(dy, dx)                 # Desired heading
        yaw = get_base_yaw()                            # Current yaw
        heading_error = (target_yaw - yaw + math.pi) % (2*math.pi) - math.pi

        # --- Check for goal reached ---
        if dist <= dist_tolerance:
            stop_wheels()
            print("Reached target!")
            return True

        # --- Read proximity sensors ---
        _, sensor_data = read_all_sensors_world_raw()
        front = sensor_data.get('f', [False])[0]
        front_left = sensor_data.get('fl', [False])[0]
        front_right = sensor_data.get('fr', [False])[0]
        obstacle_detected = front or front_left or front_right

        # --- Obstacle avoidance ---
        if obstacle_detected:
            v, w = obstacle_avoidance_diff(sensor_data,
                                           max_forward=forward_speed,
                                           min_forward=0.10,
                                           front_slow_ratio=0.45,
                                           ang_gain=1.6,
                                           ang_cap=1.8,
                                           dist_max=2.0,
                                           smoothing_tau=0.12,
                                           accel_limit=3.5)
            set_wheel_speeds(v, w)
            time.sleep(0.2)
            continue

        # --- Orientation correction ---
        if abs(heading_error) > 0.38:
            stop_wheels()
            rot_dir = 1 if heading_error > 0 else -1
            rotate_in_place(direction=rot_dir, rot_speed=rotate_speed, duration=0.05)
        else:
            # Drive forward with proportional heading correction
            set_wheel_speeds(forward_speed, heading_error * 2.0)
            time.sleep(0.1)

        # --- Progress monitoring ---
        movement = math.hypot(x - last_pos[0], y - last_pos[1])
        if movement > 0.05:
            last_progress_time = time.time()
            last_pos = (x, y)
        elif (time.time() - last_progress_time) > 1.2:
            # Stuck detection triggered
            print("Rover seems stuck - performing recovery...")
            stop_wheels()

            # Reverse short distance to free from obstacle or terrain irregularity
            for _ in range(10):
                set_all_wheels(-1.5)
                time.sleep(0.1)
            stop_wheels()

            # Small rotation before retrying
            rot_dir = 1 if heading_error <= 0 else -1
            rotate_in_place(direction=rot_dir, rot_speed=rotate_speed, duration=0.1)
            stop_wheels()

            # Reset timers and resume
            last_progress_time = time.time()
            last_pos = get_base_position()
            continue

        print(f"[{it:03}] dist={dist:.2f}, head_err={heading_error:.2f}, pos=({x:.2f},{y:.2f},{z:.2f})")

    stop_wheels()
    return False
```

Navigation Logic:

Overall Strategy

The controller continuously drives the rover from the Start to the Target point (and back), while adapting its motion in real-time based on sensor feedback. Unlike global planners like A* or D*, which require pre-mapped environments, this system uses **local decision-making** to handle unknown terrain, closer to how real exploration rovers behave in uncharted planetary surfaces.

Path Planning Logic (goto_point_dynamic)

The go-to-point logic is built on a dynamic feedback loop:

1. The rover computes the vector between its current position (x, y) and the target (tx, ty) using Euclidean distance and heading error.
2. If the heading error exceeds a threshold (≈0.38 rad), the rover performs orientation correction via in-place rotation.
3. Once aligned, it proceeds forward using differential wheel velocities, adjusting angular velocity proportionally to heading error to maintain trajectory curvature.
4. Every cycle, position data is logged to produce both 2D and 3D path plots, color-coded by time, offering visual insight into motion dynamics.

This approach combines proportional heading control and distance-based termination for smooth convergence to target coordinates.

```python
# Persistent state dictionary for smoothing angular velocity commands
_OBS_STATE = {
    "prev_ang": 0.0,          # Previous smoothed angular command
    "prev_time": time.time(), # Timestamp of previous control loop
    "prev_ang_cmd": 0.0,      # Previous raw angular command (before smoothing)
}

def obstacle_avoidance_diff(sensor_data,
                            max_forward=0.8,
                            min_forward=0.10,
                            front_slow_ratio=0.55,
                            ang_gain=2.2,
                            ang_cap=2.4,
                            dist_max=1.6,
                            smoothing_tau=0.10,
                            accel_limit=4.0):

    """
    Reactive differential-drive obstacle avoidance controller.
    Adjusts forward (linear) and turning (angular) velocity based on proximity sensor readings.

    - Uses a weighted potential field approach: closer obstacles create stronger steering away.
    - Smooths angular changes to avoid jerky motion.
    - Implements adaptive slow-down when obstacles approach the front.
    """

    # Sensor mounting angles in radians (left, center, right)
    sensor_angles = {
        'fl': math.radians(45.0),
        'f': 0.0,
        'fr': math.radians(-45.0)
    }

    # Convert raw sensor distances into normalized weights [0, 1]
    weights = {}
    for k in sensor_angles:
        entry = sensor_data.get(k)
        if (not entry) or (entry[2] is None):
            weights[k] = 0.0
            continue

        d = float(entry[2])                     # Extract measured distance
        d = max(0.0, min(d, dist_max))          # Clamp distance to valid range
        w = (dist_max - d) / dist_max           # Closer = higher weight
        if k == 'f':                            # Give more importance to the front sensor
            w *= 1.6
        weights[k] = w

    total_w = sum(weights.values())

    # Compute steering direction: turn away from weighted obstacle directions
    steer = 0.0
    if total_w > 0.0:
        for k, w in weights.items():
            steer += -sensor_angles[k] * w
        steer /= total_w

    # Determine which sensors are actively triggered
    triggered = [k for k, w in weights.items() if w > 0.3]
    num_triggered = len(triggered)

    # Weighted front obstacle strength
    front_w = weights['f'] + 0.6 * (weights['fl'] + weights['fr'])
    # Check if only one side sensor is triggered
    side_triggered = (weights['fl'] > 0.3) ^ (weights['fr'] > 0.3)

    # ===== Behavior Cases =====
    if num_triggered >= 2:
        # Multiple obstacles → likely a corner or narrow passage
        linear_vel = -0.2   # Slight reverse
        # Decide turning bias based on which side is clearer
        if 'fl' not in triggered:
            steer = sensor_angles['fl']
        elif 'fr' not in triggered:
            steer = sensor_angles['fr']
        else:
            steer = math.pi * 0.5                   # Hard turn if both sides blocked
    elif side_triggered and weights['f'] < 0.2:
        # Single side obstacle: turn slightly away while maintaining speed
        linear_vel = max(min_forward, max_forward * (1.0 - 0.2 * front_w))
    else:
        # Normal forward movement with proportional slow-down
        linear_vel = max(min_forward, max_forward * (1.0 - front_slow_ratio * front_w))

    # Extra safety tuning for narrow corridors or sharp corners
    if weights['fl'] > 0.9 and weights['fr'] > 0.9 and weights['f'] < 0.3:
        linear_vel = min(linear_vel, 0.25)
        steer *= 0.3
    if (weights['f'] > 0.6) and (weights['fl'] > 0.6 or weights['fr'] > 0.6):
        steer *= 1.4
        linear_vel = min(linear_vel, 0.35)

    # Compute raw angular velocity
    raw_ang = ang_gain * steer
    raw_ang = max(-ang_cap, min(ang_cap, raw_ang))   # Clamp angular velocity

    # ===== Smoothing Section =====
    now = time.time()
    dt = max(1e-4, now - _OBS_STATE["prev_time"])
    max_dang = accel_limit * dt                      # Maximum allowed change in angular velocity
    prev_cmd = _OBS_STATE["prev_ang_cmd"]
    dang = raw_ang - prev_cmd
    if abs(dang) > max_dang:
        raw_ang = prev_cmd + math.copysign(max_dang, dang)

    # Exponential smoothing for angular response
    alpha = dt / (smoothing_tau + dt)
    ang_smoothed = (1.0 - alpha) * _OBS_STATE["prev_ang"] + alpha * raw_ang

    # Update previous state
    _OBS_STATE.update({
        "prev_ang": ang_smoothed,
        "prev_time": now,
        "prev_ang_cmd": raw_ang
    })

    # Safety floor on forward velocity
    linear_vel = max(linear_vel, min_forward)

    return linear_vel, ang_smoothed
```

Maneuver Logic:

Obstacle Avoidance (obstacle_avoidance_diff)

Obstacle detection relies on three proximity sensors (front-left, front, and front-right) whose readings are weighted by distance.

Each sensor has a corresponding field of view angle (+45°, 0°, -45°).

The avoidance algorithm applies the following logic:

Weighted Potential Field Response:

- The closer an obstacle, the higher its weight — this modifies the steering angle (steer) to turn away from the obstacle.

Front Sensitivity Bias:

- The front sensor is given higher importance (×1.6 weight factor) to prioritize head-on avoidance.

Adaptive Linear Velocity:

- The forward speed decreases as obstacles approach, ensuring smooth deceleration.

Bang–Bang Steering with Filtering:

- Angular velocity is capped (±2.4 rad/s) and filtered with an exponential smoothing filter (smoothing_tau) to prevent jerky turns.

Side and Corner Handling:

- Special cases detect when only one side sensor triggers, causing gentle steering away, while full front obstruction triggers a short reverse or pivot maneuver.

This model mimics the behavior-based navigation paradigm emphasizing reactive control over predictive mapping, ideal for unstructured environments like Mars.

```python
# ------------------- Setup -------------------
# Get object handles for the Start and Target dummy objects in CopelliaSim
start_obj = sim.getObject('/Start')
target_obj = sim.getObject('/Target')

# Get their positions (x, y, z) in world coordinates
start_xyz = sim.getObjectPosition(start_obj, -1)
target_xyz = sim.getObjectPosition(target_obj, -1)

# Extract only x, y for navigation (2D plane assumption)
start_xy = tuple(start_xyz[:2])
target_xy = tuple(target_xyz[:2])

# Initialize a log list to store rover positions over time
log = []
sim_start_time = None

# Function to log rover's current position and elapsed time during navigation
def log_current_position(log_list, start_time):
    pos = get_base_position()  # Fetch current rover position from CopelliaSim
    elapsed = time.time() - start_time
    log_list.append([round(pos[0], 3), round(pos[1], 3), round(pos[2], 3), round(elapsed,
2)])

print("\nStarting Env2...\n")
# Start the simulation inside CopelliaSim
sim.startSimulation()
time.sleep(0.3)
sim_start_time = time.time()

# Navigate from Start → Target using dynamic movement
success = goto_point_dynamic(
    target_xy,
    forward_speed=10.0,
    rotate_speed=10.0,
    dist_tolerance=1,  # stop when within 1m of target
    log_fn=lambda: log_current_position(log, sim_start_time),
    obstacle_threshold=1  # reactive obstacle avoidance threshold
)

time.sleep(0.3)

print("➡️ Navigating from Target → Start...")
# Now go back from Target → Start to verify bidirectional navigation
success_back = goto_point_dynamic(
    start_xy,
    forward_speed=10.0,
    rotate_speed=10.0,
    dist_tolerance=1,
    log_fn=lambda: log_current_position(log, sim_start_time),
    obstacle_threshold=1
)

# Check whether return journey was successful
if success_back:
    print("Returned to Start!")
else:
    print("Could not return to Start!")

# Stop the rover's wheels and simulation
stop_wheels()
sim.stopSimulation()

# Print the mission outcome
if success:
    print("Mission complete: Reached target!")
else:
    print("Mission ended: Could not reach target.")

# ------------------- Logging -------------------
# Save all logged rover positions to a CSV file
csv_path = "rover_path_env2.csv"
with open(csv_path, "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["x", "y", "z", "t"])
    writer.writerows(log)
print(f"Saved path log to {csv_path}")

# ------------------- Visualization -------------------
# If there's movement data, plot it in both 2D and 3D for analysis
if len(log) > 0:
    log_array = np.array(log)
    xs, ys, zs, ts = log_array[:,0], log_array[:,1], log_array[:,2], log_array[:,3]

    # --- 2D path visualization ---
    plt.figure()
    points = np.array([xs, ys]).T.reshape(-1, 1, 2)
    segments = np.concatenate([points[:-1], points[1:]], axis=1)
    from matplotlib.collections import LineCollection
    lc = LineCollection(segments, cmap='viridis', linewidths=2)
    lc.set_array(ts)
    plt.gca().add_collection(lc)
    plt.scatter([target_xy[0]], [target_xy[1]], c='r', label='target')
    plt.scatter([start_xy[0]], [start_xy[1]], c='g', label='start')
    plt.xlabel('X (m)'); plt.ylabel('Y (m)'); plt.title('2D path colored by time')
    plt.colorbar(lc, label='Time (s)')
    plt.legend()
    plt.axis('equal')
    plt.grid(True)
    plt.show()

    # --- 3D path visualization ---
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    p = ax.scatter(xs, ys, zs, c=ts, cmap='plasma', s=15)
    fig.colorbar(p, ax=ax, label='Time (s)')
    ax.set_xlabel('X'); ax.set_ylabel('Y'); ax.set_zlabel('Z')
    ax.set_title('3D path colored by time')
    plt.show()
```

Main Script Logic:

Initialization:

- The script fetches the positions of the Start and Target dummy objects placed inside CopelliaSim.

Dynamic Path Execution:

- It uses a custom motion control function goto_point_dynamic() that continuously moves the rover toward the target while checking for nearby obstacles (using threshold-based avoidance).

Bidirectional Test:

- Once the rover reaches the target, it returns back to the start — confirming two-way navigation performance.

Simulation & Logging:

- Throughout the simulation, the rover's (x, y, z) positions and timestamps are logged, saved to rover_path_env2.csv for later analysis.

Visualization:

- After simulation, the script uses Matplotlib to render:
- A 2D top-down view of the rover's path (colored by time elapsed)
- A 3D trajectory plot of rover movement through terrain.

Stuck Recovery Logic (perform_stuck_recovery)

If the rover's position does not change significantly over a time window, a stuck recovery routine is initiated:

1. **Reverse Oscillation:** Alternates left–right wheel speeds to "shake free" from small pits or obstructions.
2. **Reorientation:** Computes the new angle to the target and performs a deliberate rotation in that direction.
3. **Retry Navigation:** Once freed, the rover resumes its routine.

This mimics real rovers' "fault protection mode" — allowing autonomous recovery from minor terrain traps without external intervention.

```python
def perform_stuck_recovery(target_xy, stuck_check_fn):
    print("⚠️ Stuck detected — starting recovery sequence...")
    stop_wheels()
    time.sleep(0.1)

    #Phase 1: Reverse oscillation to unjam
    for _ in range(3):
        set_wheel_velocities(-3.5, -2.5, -3.5, -2.5)
        time.sleep(0.4)
        set_wheel_velocities(-2.5, -3.5, -2.5, -3.5)
        time.sleep(0.4)
        stop_wheels()
        time.sleep(0.1)
        if not stuck_check_fn():
            print("Freed during reverse oscillation.")
            return

    #Phase 2: Rotate toward target direction
    base_pos = get_base_position()
    base_yaw = get_base_yaw()
    dx, dy = target_xy[0] - base_pos[0], target_xy[1] - base_pos[1]
    target_angle = math.atan2(dy, dx)
    angle_to_target = angle_diff(target_angle, base_yaw)
    direction = 1 if angle_to_target > 0 else -1

    print(f"↪ Rotating {('left' if direction==1 else 'right')} toward target.")
    rotate_in_place(direction=direction, rot_speed=4.5, duration=0.8)
    stop_wheels()
    time.sleep(0.2)
    if not stuck_check_fn():
        print("Freed after reorientation.")
        return

    print("Recovery complete — resuming navigation.")
```

```python
# Function to detect a specific color object (here cyan) in an image frame
def filter_color(img, color):

    # Define HSV range for cyan color
    if color == 'cyan':
        lower_color = np.array([80, 70, 50])     # Lower HSV bound
        upper_color = np.array([100, 255, 255])  # Upper HSV bound

    # Convert BGR image (from camera) to HSV color space for better color filtering
    hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)

    # Create a binary mask where only the desired color range is white (255)
    mask = cv.inRange(hsv, lower_color, upper_color)
    cv.imshow('Mask', mask)   # Display the color mask for visualization

    # Find contours (edges/boundaries) in the mask
    contours, _ = cv.findContours(mask, cv.RETR_LIST, cv.CHAIN_APPROX_NONE)

    # Initialize variables for object detection
    cx, cy = 0, 0       # Object center (x, y)
    objsize = 0         # Area of detected object
    objDetected = False # Flag if object is found

    # Loop through all detected contours
    for contour in contours:
        if cv.contourArea(contour) > 100:   # Ignore very small/noisy detections
            objDetected = True
            objsize = cv.contourArea(contour)   # Calculate object size (area)
            x, y, w, h = cv.boundingRect(contour)  # Get bounding box coordinates
            cx, cy = x + w // 2, y + h // 2       # Calculate center of the object

            # Draw a rectangle and label around the detected object
            cv.rectangle(img, (x, y), (x + w, y + h), (0, 255, 0), 1)
            cv.putText(img, f'{color} ({cx}, {cy})', (x, y - 10),
                       cv.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)
            break   # Stop after detecting the first valid object

    # Return processed frame, detection flag, and object parameters
    return img, objDetected, cx, cy, objsize


# ---------------- Simulation Starts Here ----------------
sim.startSimulation()

# Get the handle of the vision sensor in CoppeliaSim
visionSensor = sim.getObject('/visionSensor')
print(f'visionSensor handle : {visionSensor}')

# Set the color to detect
color = ['cyan']

# Start by rotating left slightly to begin scanning
rotate_left(5)
i = 0

# Main loop: continuously read frames and search for the object
while True:
    img = readFrame()   # Capture frame from vision sensor
    img, detectObj, x, y, objsize = filter_color(img, color[i])  # Detect object
    cv.imshow('img', img)   # Display the camera feed with detection overlay

    if detectObj:
        # If object detected → calculate horizontal alignment error
        error = 128 - x
        p = 0.1 * error  # Simple proportional control to center the object

        # Move wheels with small speed difference to center object
        move(3 - p, 3 + p)
        print(objsize)

        # If object appears large enough (close to camera)
        if objsize > 1900:
            stop_wheels()
            print("The block is in front of me!")
            sim.stopSimulation()
            break

    else:
        # If object not found → keep rotating left slowly
        rotate_left(2.5)
        print("Rotleft else")

    # Manual exit option
    if cv.waitKey(1) & 0xFF == ord('e'):
        break
```

Recognize Samples (ENV 3):

Color detection:

- The function filter_color() converts each image frame into HSV color space and applies a color threshold to isolate a particular color — in this case, cyan.

Contour detection:

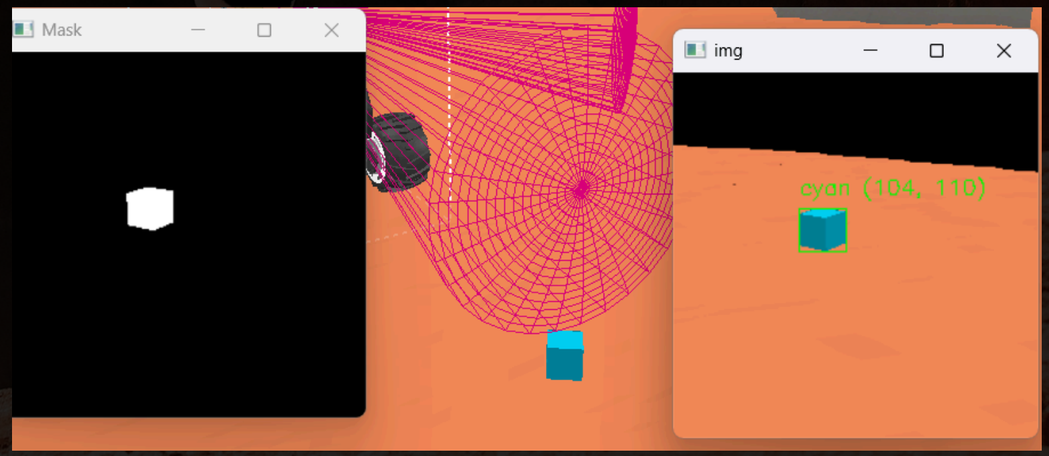
- Using OpenCV, the code detects the shape and position of the colored object and marks it with a rectangle and label on the frame.

Object tracking:

- If the object is detected, the rover adjusts its wheel speeds proportionally to align itself with the object's center (using basic proportional control).
- If the object isn't visible, it rotates left to keep scanning.

Approach and stop:

- When the detected object becomes large enough (area > 1900), meaning it's very close, the rover stops its wheels and ends the simulation.



A Thorough Comparison

Similarities

Our simulated rover emulates the core autonomy stack used in NASA's Perseverance and Curiosity rovers:

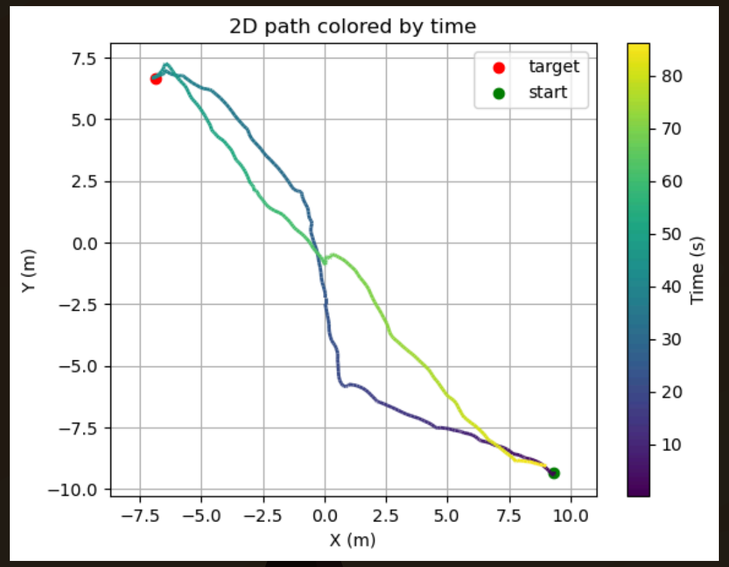
- **Local Hazard Avoidance:** Both rely on onboard sensors (e.g., stereo cameras or LIDAR) to detect obstacles and make path corrections.
- **Reactive Planning:** Real rovers also use near-field motion planning for unknown terrains, executing low-level control similar to your differential steering logic.
- **Autonomous Fault Recovery:** Real missions incorporate fault detection and recovery sequences to handle wheel slip, sand traps, or blocked paths.

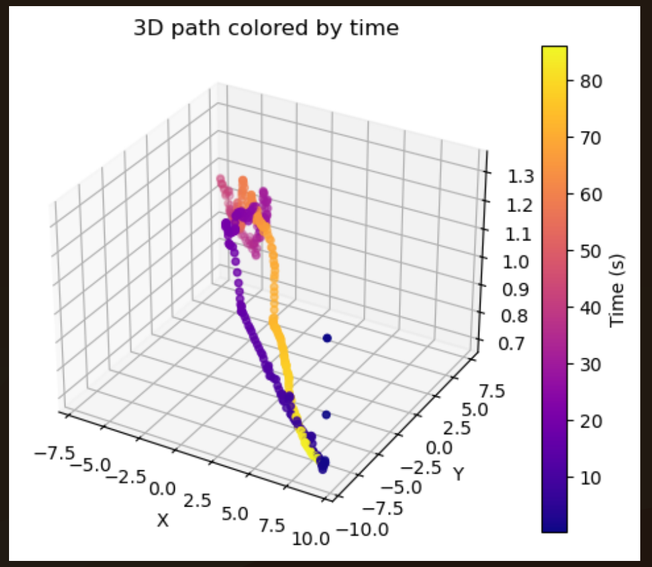
Differences

Feature	Principle	NASA Rover (Perseverance and Curiosity)	Our Rover
Autonomous Hazard Avoidance (AHA)	Real rovers employ stereo imaging and LIDAR to generate digital elevation maps, then apply path cost evaluation to select safe routes.	Perseverance's AEGIS system autonomously chooses waypoints avoiding rocks or craters.	Our proximity-based reactive control is a simplified version of this using local sensing rather than depth maps to steer around hazards.
Sample Detection and Collection	Robotic arm equipped with force sensors and camera-guided positioning performs drilling and sample caching.	Perseverance uses SHERLOC and WATSON instruments for identifying and collecting rock cores.	Our simulation's pick-and-place mechanic (though unimplemented due to import limitations) mirrors this robotic manipulation workflow.
Visual Odometry	Tracking movement using sequential camera frames to estimate displacement through feature matching (SIFT/SURF).	Used by Curiosity and Perseverance to correct wheel encoder drift and precisely localize their position on uneven terrain.	Our rover similarly uses encoder and IMU-based feedback (and in theory, visual cues from the Vision Sensor) for positional updates.

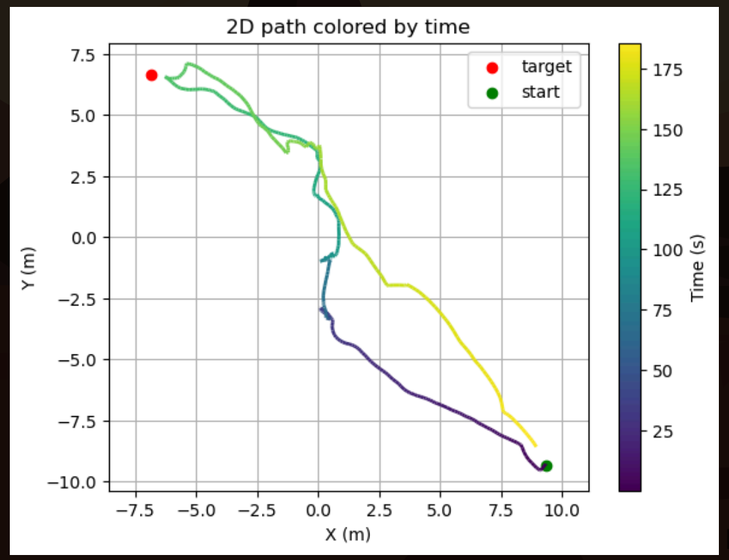
Our Achieved Paths

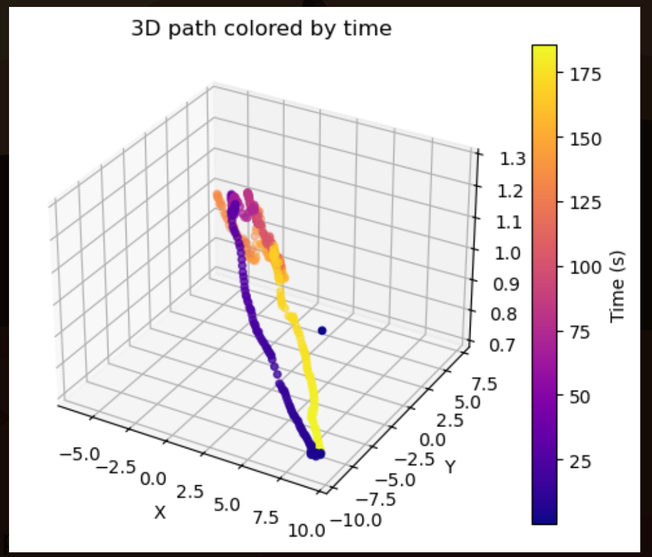
Env 1





Env 2





04
Challenges Faced

Challenges and Solutions

Importing issues of Solidworks URDF file to CopelliaSim

The issue of importing the SolidWorks-generated URDF file into CoppeliaSim could not be resolved due to the lack of sufficient support, tutorials, and resources available for this process (Not even in the resources provided by the ARES team). Consequently, the hardware team's SolidWorks rover model (provided in the hardware folder) could not be imported into the simulation environment.
This, in turn, prevented the software team from testing the rover in simulation and from implementing the pick-and-place mechanism.

CAD Model Creation

Achieved a unique design in SolidWorks that was robust yet simplified enough for efficient simulation. Prioritizing **primitive shapes** for the main chassis components and utilizing the Weldment feature to ensure a unified, rigid base structure.

Joint and Kinematic Assembly

Perfectly assembling over 20 parts (wheels, bogies, links) with accurate mates to ensure the rocker-bogie joints moved realistically without slippage or excessive friction. Rigorously defining **coincident, concentric, and parallel mates** in SolidWorks to ensure high-quality, non-redundant joints, and checking the full range of motion

Thank You

PRESENTED BY TEAM ENDURANCE

RISHOV SARKAR

SUPRATIM CHAKRABORTY

REVIEW MONDAL

SOUTRIK SAHA