```
!pip install gym numpy stable-baselines3 matplotlib shapely requests shimmy
```

```
Requirement already satisfied: gym in /usr/local/lib/python3.11/dist-packages (0.25.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (2.0.2)
Collecting stable-baselines3
  Downloading stable_baselines3-2.6.0-py3-none-any.whl.metadata (4.8 kB)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages (3.10.0)
Requirement already satisfied: shapely in /usr/local/lib/python3.11/dist-packages (2.1.0)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (2.32.3)
Collecting shimmy
  Downloading Shimmy-2.0.0-py3-none-any.whl.metadata (3.5 kB)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from gym) (3.1.1)
Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.11/dist-packages (from gym) (0.0.8)
Requirement already satisfied: gymnasium<1.2.0,>=0.29.1 in /usr/local/lib/python3.11/dist-packages (from stable-baselines3) (1.1.
Requirement already satisfied: torch<3.0,>=2.3 in /usr/local/lib/python3.11/dist-packages (from stable-baselines3) (2.6.0+cu124)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from stable-baselines3) (2.2.2)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (1.3.2)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (4.57.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (1.4.8)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (24.2)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests) (2.4.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests) (2025.4.26)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.11/dist-packages (from gymnasium<1.2.0,>=0.29.1
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/python3.11/dist-packages (from gymnasium<1.2.0,>=0.2
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib) (1.17.
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from torch<3.0,>=2.3->stable-baselines3) (3.1
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch<3.0,>=2.3->stable-baselines3) (3.4
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch<3.0,>=2.3->stable-baselines3) (3.1.6
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from torch<3.0,>=2.3->stable-baselines3) (2025.
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/python3.11/dist-packages (from torch<3.0,>=2.3->st
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch<3.0,>=2.3->stable-
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch<3.0,>=2.3->stabl
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch<3.0,>=2.3->stable-baselines3)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packages (from torch<3.0,>=2.3->stable-baselines3)
```

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from shapely.geometry import Polygon, Point
import random
import time
import math
from collections import deque

class UAVCoverageEnv(gym.Env):
    def __init__(self, coverage_area, no_fly_zones, wind_data, payload_weight, battery_capacity, current_battery):
        super(UAVCoverageEnv, self).__init__()
        print("Initializing UAV Environment...")

        # Convert coverage area and no-fly zones into polygons
        self.coverage_area = Polygon([(p["lon"], p["lat"]) for p in coverage_area])
        self.no_fly_zones = [Polygon([(p["lon"], p["lat"]) for p in zone]) for zone in no_fly_zones]

        # Assign wind data
        self.wind_data = wind_data

        # Calculate area dimensions
        minx, miny, maxx, maxy = self.coverage_area.bounds
        width = maxx - minx
        height = maxy - miny
```

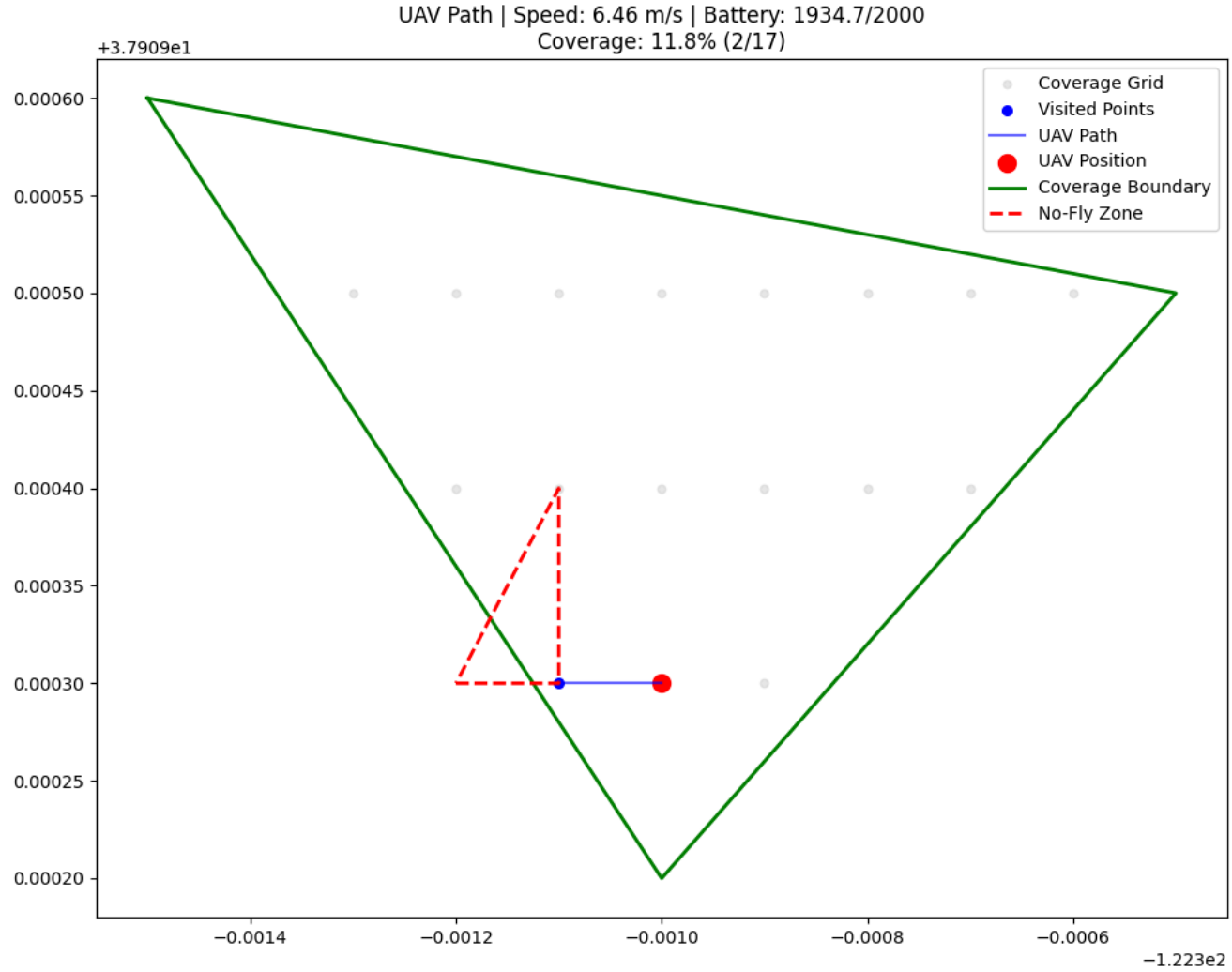
```
        # UAV Parameters (User Input)
        self.payload_weight = payload_weight
        self.battery_capacity = battery_capacity
        self.current_battery = current_battery

        # Initialize grid_size temporarily before speed computation
        self.grid_size = 1

        # Determine grid size based on area dimensions
        # Aim for approximately 100 points in the grid
        target_points = 100
        calculated_grid_size = min(width, height) / math.sqrt(target_points)

        # Compute UAV Speed
        self.uav_speed = self._compute_dynamic_speed()

        # Update grid_size - ensure minimum size for visibility
        self.grid_size = max(0.0001, min(calculated_grid_size, min(width, height) / math.sqrt(target_points)))

        print(f"Coverage area: width={width}, height={height}")
        print(f"Initial grid size set to: {self.grid_size}")

        # Generate grid
        self.grid = self._generate_grid()

        if len(self.grid) == 0:
            # As a last resort, add the center point of the coverage area
            minx, miny, maxx, maxy = self.coverage_area.bounds
            center_x = (minx + maxx) / 2
            center_y = (miny + maxy) / 2
            print(f"WARNING: No valid grid points generated. Adding center point at ({center_x}, {center_y}).")
            self.grid.append((center_x, center_y))

        self.grid_indices = {point: i for i, point in enumerate(self.grid)}
        self.position = self.grid[0]
        self.visited = set([self.position])  # Mark initial position as visited
        self.visited_indices = {0}  # Indices of visited points

        # RL Action & Observation Spaces
        self.action_space = gym.spaces.Discrete(len(self.grid))  # Action is which grid point to visit
        self.observation_space = gym.spaces.Box(low=-1, high=1, shape=(5,), dtype=np.float32)

        # For visualization
        self.trajectory = [self.position]

        # Debug counters
        self.stuck_count = 0
        self.last_position = None

        print(f"Environment initialized with {len(self.grid)} grid points. Grid size: {self.grid_size}")

    def _compute_dynamic_speed(self):
        """ Adjust speed based on battery level, payload, and wind resistance. """
        base_speed = 10
        wind_effect = self._compute_wind_effect()

        normal_power = 100 + (self.payload_weight * 0.5)
        battery_factor = self.current_battery / self.battery_capacity
        if battery_factor < 0.3:
            base_speed *= 0.7

        wind_factor = max(0.5, 1 - wind_effect)
        speed = base_speed * wind_factor

        return max(2, min(15, speed))

    def _compute_wind_effect(self):
        """ Calculates wind effect on energy consumption. """
        wind_direction = np.deg2rad(self.wind_data["direction"])
        move_angle = np.arctan2(1, 1)  # Use static angle
        wind_effect = np.cos(wind_direction - move_angle) * self.wind_data["speed"] / 10
        return wind_effect

    def _generate_grid(self):
        """ Generate a grid within the coverage area, avoiding no-fly zones. """
        minx, miny, maxx, maxy = self.coverage_area.bounds
        grid = []
        x, y = minx, miny

        # Debug information
        print(f"Generating grid with size {self.grid_size}")
```

```
        print(f"Coverage area bounds: {self.coverage_area.bounds}")

        # Calculate approximate dimensions of coverage area
        width = maxx - minx
        height = maxy - miny
        print(f"Coverage area dimensions: width={width}, height={height}")

        # Estimate number of grid points
        est_points = int((width / self.grid_size) * (height / self.grid_size))
        print(f"Estimated number of points (if all in bounds): {est_points}")

        while y <= maxy:
            while x <= maxx:
                point = Point(x, y)
                if self.coverage_area.contains(point) and not any(zone.contains(point) for zone in self.no_fly_zones):
                    grid.append((x, y))
                x += self.grid_size
            x = minx
            y += self.grid_size

        print(f"Generated {len(grid)} valid grid points")
        return grid

    def _find_nearest_unvisited(self):
        """Find the nearest unvisited grid point."""
        current_idx = self.grid_indices[self.position]
        unvisited_indices = set(range(len(self.grid))) - self.visited_indices

        if not unvisited_indices:
            return None  # All points visited

        nearest_idx = None
        min_distance = float('inf')
        min_energy = float('inf')

        for idx in unvisited_indices:
            next_pos = self.grid[idx]
            distance = math.sqrt((self.position[0] - next_pos[0])**2 + (self.position[1] - next_pos[1])**2)
            energy_needed = self._compute_energy_usage(self.position, next_pos)

            # Check if we can reach this point with our energy
            if energy_needed <= self.current_battery:
                if distance < min_distance:
                    min_distance = distance
                    min_energy = energy_needed
                    nearest_idx = idx

        return nearest_idx

    def step(self, action=None):
        """Move UAV to the next point in coverage pattern."""
        # Find nearest unvisited point if no action is provided
        if action is None:
            next_idx = self._find_nearest_unvisited()
            if next_idx is None:
                # If all points are visited or unreachable, stay in place
                print("All reachable points have been visited.")
                return self._get_observation(), 0, True, {}
        else:
            next_idx = action

        # Check if the action is valid
        if next_idx >= len(self.grid):
            print(f"Invalid action: {next_idx}. Staying in place.")
            return self._get_observation(), -1, False, {}

        next_position = self.grid[next_idx]

        # Calculate energy for this move
        energy_used = self._compute_energy_usage(self.position, next_position)

        # Debug: track if we're stuck at the same position
        if self.position == self.last_position:
            self.stuck_count += 1
            if self.stuck_count > 5:
                print(f"WARNING: UAV appears stuck at {self.position} for {self.stuck_count} steps")
                # Try to find any unvisited point we can reach
                unvisited = set(range(len(self.grid))) - self.visited_indices
                for idx in unvisited:
                    alt_pos = self.grid[idx]
                    alt_energy = self._compute_energy_usage(self.position, alt_pos)
                    if alt_energy <= self.current_battery:
```

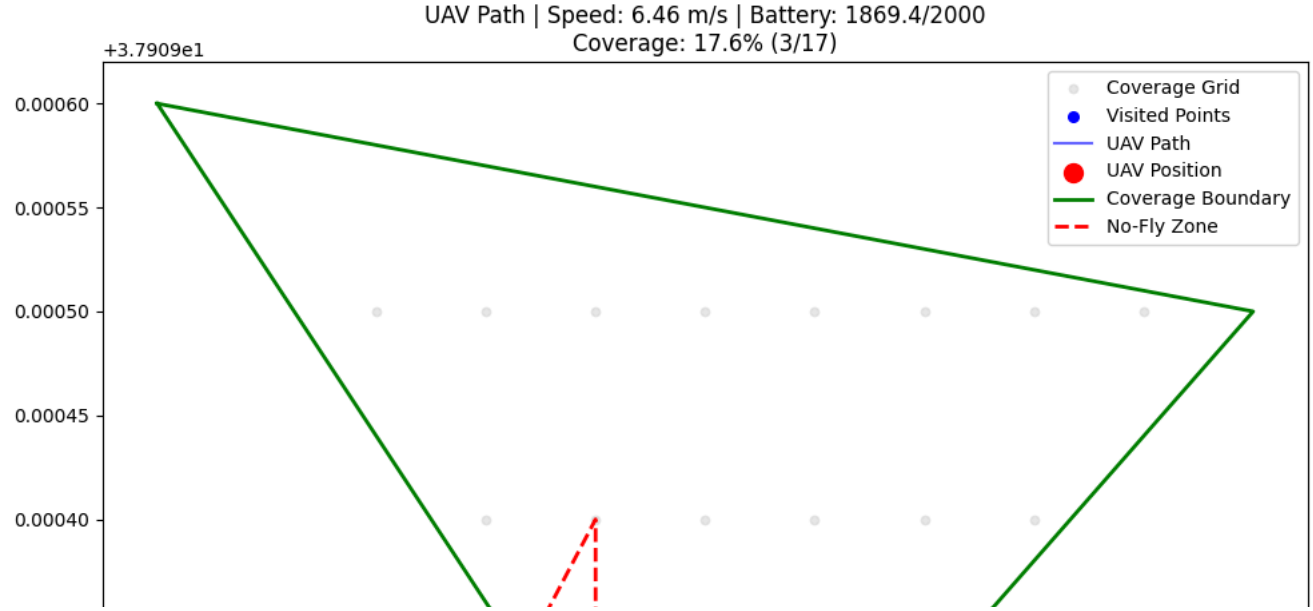
```
                    next_idx = idx
                    next_position = alt_pos
                    energy_used = alt_energy
                    print(f"Breaking stuck pattern by moving to {next_position}")
                    break
        else:
            self.stuck_count = 0

        self.last_position = self.position

        # If we have enough energy, move to the next position
        if energy_used <= self.current_battery:
            prev_position = self.position
            self.position = next_position
            self.trajectory.append(self.position)

            # Mark as visited
            self.visited.add(self.position)
            self.visited_indices.add(next_idx)

            # Update battery
            self.current_battery -= energy_used

            # Calculate reward based on whether this is a new point
            reward = 1 if next_position not in self.visited else -0.5

            print(f"Step: Moved from {prev_position} to {next_position}")
            print(f"Energy used: {energy_used:.2f}, remaining: {self.current_battery:.2f}")
            print(f"Visited {len(self.visited)}/{len(self.grid)} points")
        else:
            # Not enough energy
            print(f"Not enough energy to move to {next_position}.")
            print(f"Required: {energy_used:.2f}, Available: {self.current_battery:.2f}")
            reward = -1

        # Check if done (all points visited or out of energy)
        done = (len(self.visited) == len(self.grid)) or (self.current_battery <= 0)

        return self._get_observation(), reward, done, {}

    def _get_observation(self):
        """Get current observation state."""
        coverage_percentage = len(self.visited) / len(self.grid)
        print(f"Coverage: {coverage_percentage*100:.1f}% ({len(self.visited)}/{len(self.grid)})")

        return np.array([
            self.position[0],
            self.position[1],
            self.current_battery / self.battery_capacity,
            self.uav_speed / 15,
            coverage_percentage
        ], dtype=np.float32)

    def _compute_energy_usage(self, from_pos, to_pos):
        """ Calculate power consumption for movement between two positions. """
        distance = math.sqrt((from_pos[0] - to_pos[0])**2 + (from_pos[1] - to_pos[1])**2)
        if distance == 0:
            return 0

        # Direction of movement
        dx, dy = to_pos[0] - from_pos[0], to_pos[1] - from_pos[1]
        move_angle = math.atan2(dy, dx) if (dx != 0 or dy != 0) else 0

        # Calculate wind effect based on movement direction
        wind_direction = np.deg2rad(self.wind_data["direction"])
        wind_effect = math.cos(wind_direction - move_angle) * self.wind_data["speed"] / 10

        power_usage = 100 + (self.payload_weight * 0.5)
        wind_resistance = 1 + abs(wind_effect) * 0.3

        # Scale energy usage by distance relative to grid size
        return power_usage * wind_resistance * (self.uav_speed / 10) * (distance / self.grid_size)

    def reset(self):
        """ Reset UAV position and battery for a new simulation. """
        self.position = self.grid[0]
        self.visited = set([self.position])  # Mark initial position as visited
        self.visited_indices = {0}  # Reset visited indices
        self.trajectory = [self.position]  # Reset trajectory
        self.current_battery = self.battery_capacity
        self.uav_speed = self._compute_dynamic_speed()
        self.stuck_count = 0
```

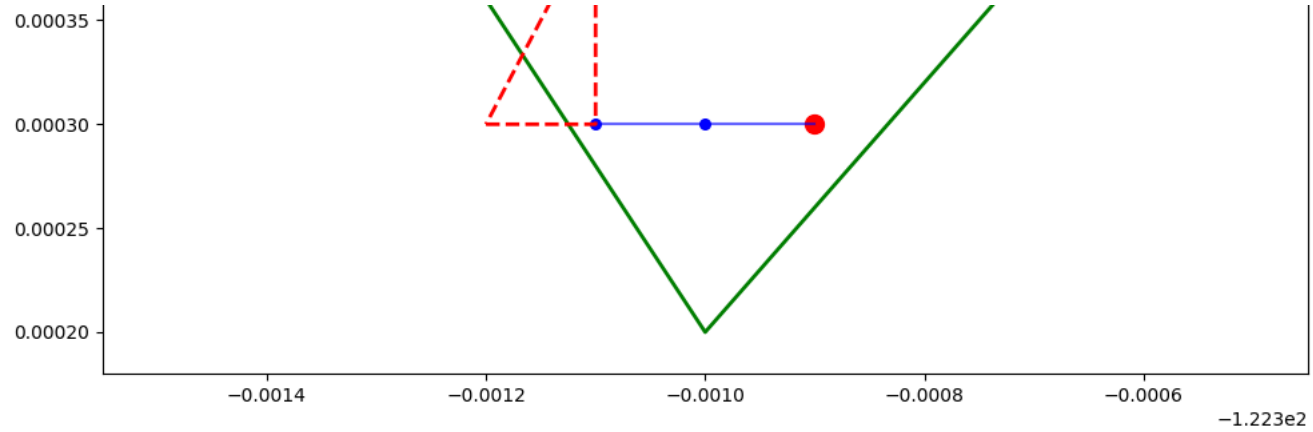
```
            self.last_position = None

            return self._get_observation()

        def render(self):
            """ Visualize UAV path and speed dynamically. """
            plt.figure(figsize=(10, 8))

            # Plot the grid points
            x, y = zip(*self.grid) if self.grid else ([], [])
            plt.scatter(x, y, c="lightgray", alpha=0.5, s=20, label="Coverage Grid")

            # Plot the visited points
            visited_x, visited_y = zip(*self.visited) if self.visited else ([], [])
            if visited_x:  # Only plot if there are visited points
                plt.scatter(visited_x, visited_y, c="blue", s=30, label="Visited Points")

            # Plot the trajectory path
            if len(self.trajectory) > 1:
                trajectory_x, trajectory_y = zip(*self.trajectory)
                plt.plot(trajectory_x, trajectory_y, 'b-', linewidth=1.5, alpha=0.6, label="UAV Path")

            # Plot current position
            plt.scatter(*self.position, c="red", s=100, label="UAV Position")

            # Plot coverage area boundary
            x, y = self.coverage_area.exterior.xy
            plt.plot(x, y, c="green", linewidth=2, label="Coverage Boundary")

            # Plot no-fly zones
            for i, zone in enumerate(self.no_fly_zones):
                x, y = zone.exterior.xy
                label = "No-Fly Zone" if i == 0 else ""
                plt.plot(x, y, c="red", linestyle='--', linewidth=2, label=label)

            plt.title(f"UAV Path | Speed: {self.uav_speed:.2f} m/s | Battery: {self.current_battery:.1f}/{self.battery_capacity}\nCoverage:

            # Only show one legend entry for each type
            handles, labels = plt.gca().get_legend_handles_labels()
            by_label = dict(zip(labels, handles))
            plt.legend(by_label.values(), by_label.keys())

            plt.tight_layout()
            plt.show()
            # Keep figure open briefly to show movement
            plt.pause(0.5)
            plt.close()

    # Example Usage
    coverage_area = [{"lon": -122.3015, "lat": 37.9096},
        {"lon": -122.3010, "lat": 37.9092},
        {"lon": -122.3005, "lat": 37.9095},
        {"lon": -122.3015, "lat": 37.9096}]
    no_fly_zones = [[{"lon": -122.3012, "lat": 37.9093}, {"lon": -122.3011, "lat": 37.9093}, {"lon": -122.3011, "lat": 37.9094}]]
    wind_data = {"speed": 5, "direction": 90}

    # Initialize Environment
    env = UAVCoverageEnv(coverage_area, no_fly_zones, wind_data, payload_weight=2, battery_capacity=2000, current_battery=2000)

    # Run Simulation
    obs = env.reset()
    done = False
    max_steps = 200  # Increased max steps

    try:
        step_count = 0
        while not done and step_count < max_steps:
            # Move to nearest unvisited point
            obs, reward, done, _ = env.step(None)
            env.render()
            step_count += 1
            time.sleep(0.2)  # Slow down visualization

        if step_count >= max_steps:
            print(f"Simulation stopped after {max_steps} steps")
        print(f"Final coverage: {len(env.visited)/len(env.grid)*100:.1f}% ({len(env.visited)}/{len(env.grid)})")
        print(f"Remaining battery: {env.current_battery:.1f}/{env.battery_capacity}")
    except Exception as e:
        print(f"Error during simulation: {e}")
```

```
Initializing UAV Environment...
Coverage area: width=0.0010000000000047748, height=0.00039999999999906777
Initial grid size set to: 0.0001
Generating grid with size 0.0001
Coverage area bounds: (-122.3015, 37.9092, -122.3005, 37.9096)
Coverage area dimensions: width=0.0010000000000047748, height=0.00039999999999906777
Estimated number of points (if all in bounds): 40
Generated 17 valid grid points
Environment initialized with 17 grid points. Grid size: 0.0001
Coverage: 5.9% (1/17)
Step: Moved from (-122.30109999999999, 37.9093) to (-122.30099999999999, 37.9093)
Energy used: 65.29, remaining: 1934.71
Visited 2/17 points
Coverage: 11.8% (2/17)
```

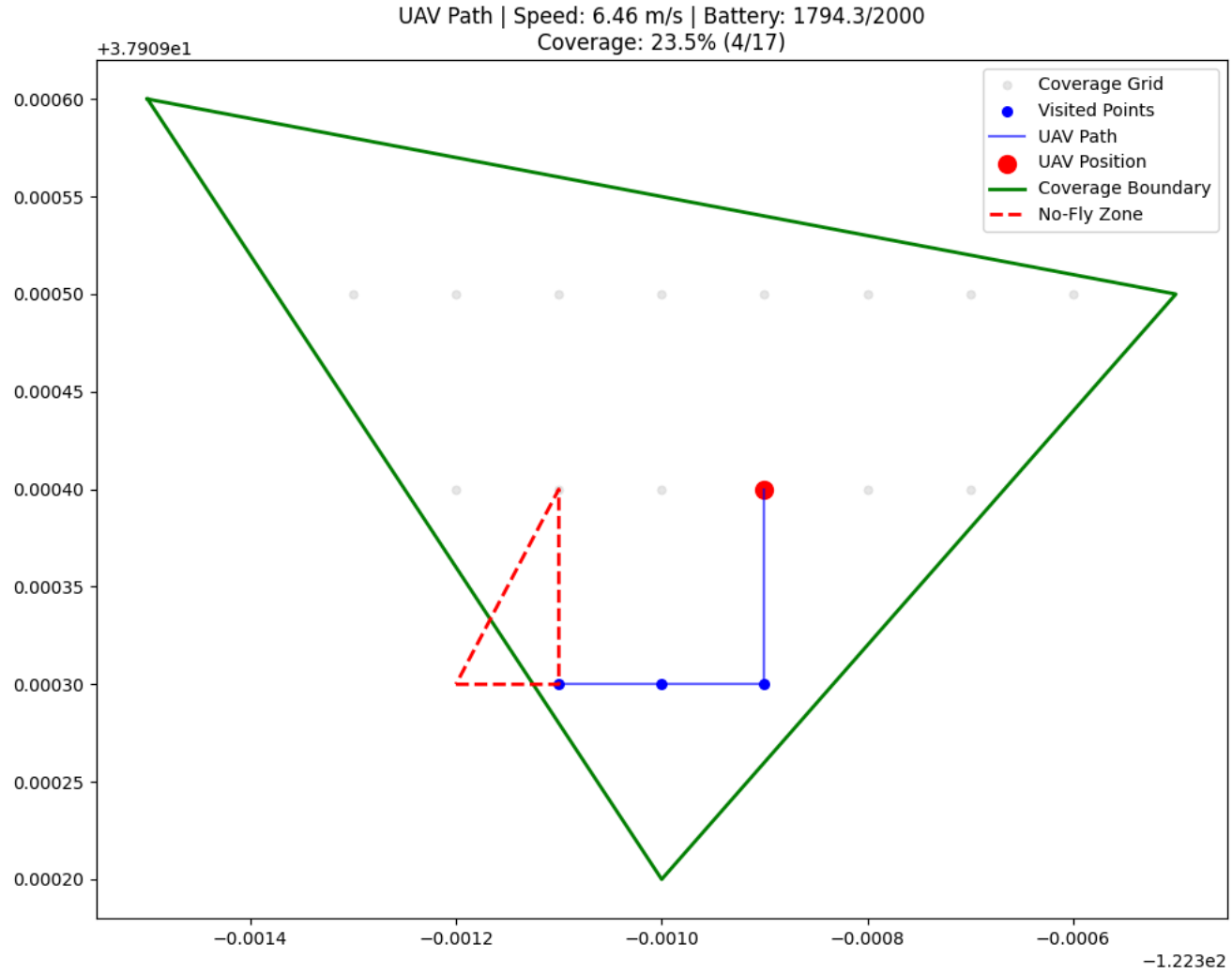



```
Step: Moved from (-122.30099999999999, 37.9093) to (-122.30089999999998, 37.9093)
Energy used: 65.29, remaining: 1869.42
Visited 3/17 points
Coverage: 17.6% (3/17)
```

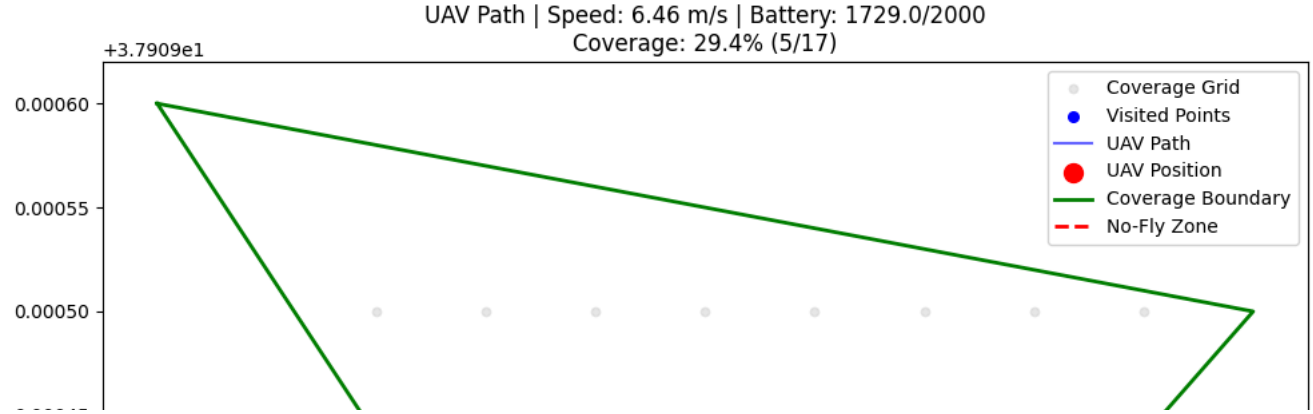


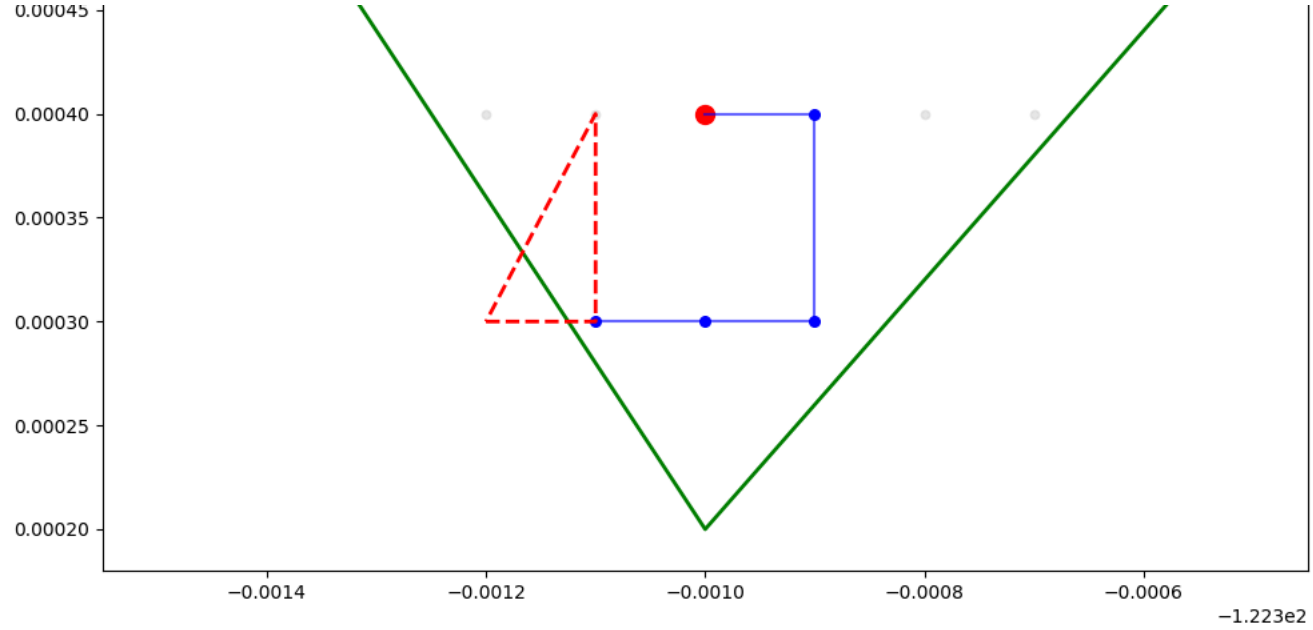



```
Step: Moved from (-122.30089999999998, 37.9093) to (-122.30089999999998, 37.909400000000005)
Energy used: 75.08, remaining: 1794.33
Visited 4/17 points
Coverage: 23.5% (4/17)
```

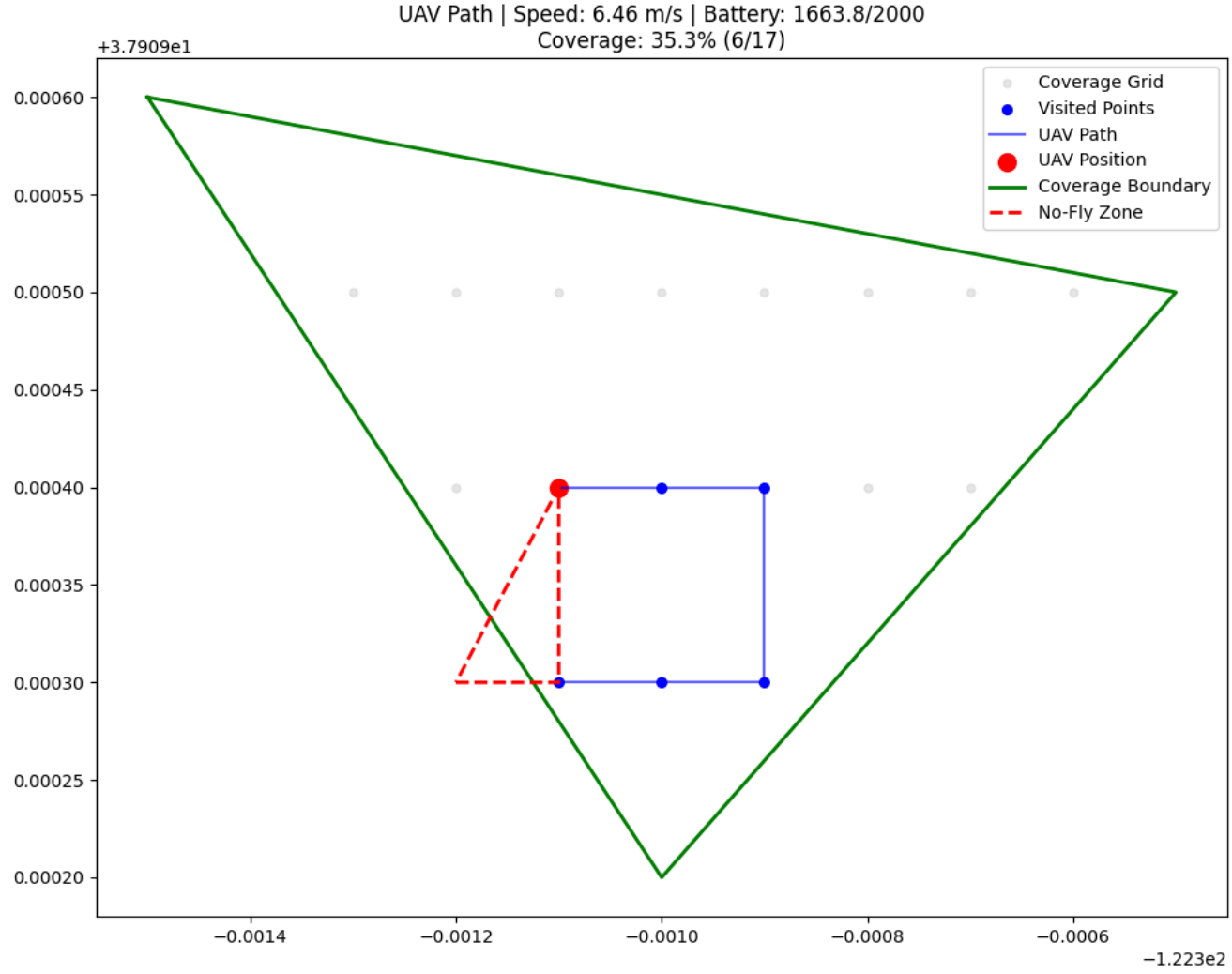



```
Step: Moved from (-122.30089999999998, 37.909400000000005) to (-122.30099999999999, 37.909400000000005)
Energy used: 65.29, remaining: 1729.04
Visited 5/17 points
Coverage: 29.4% (5/17)
```

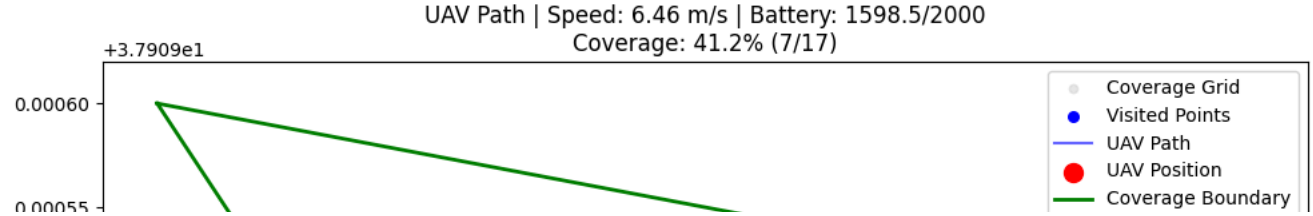


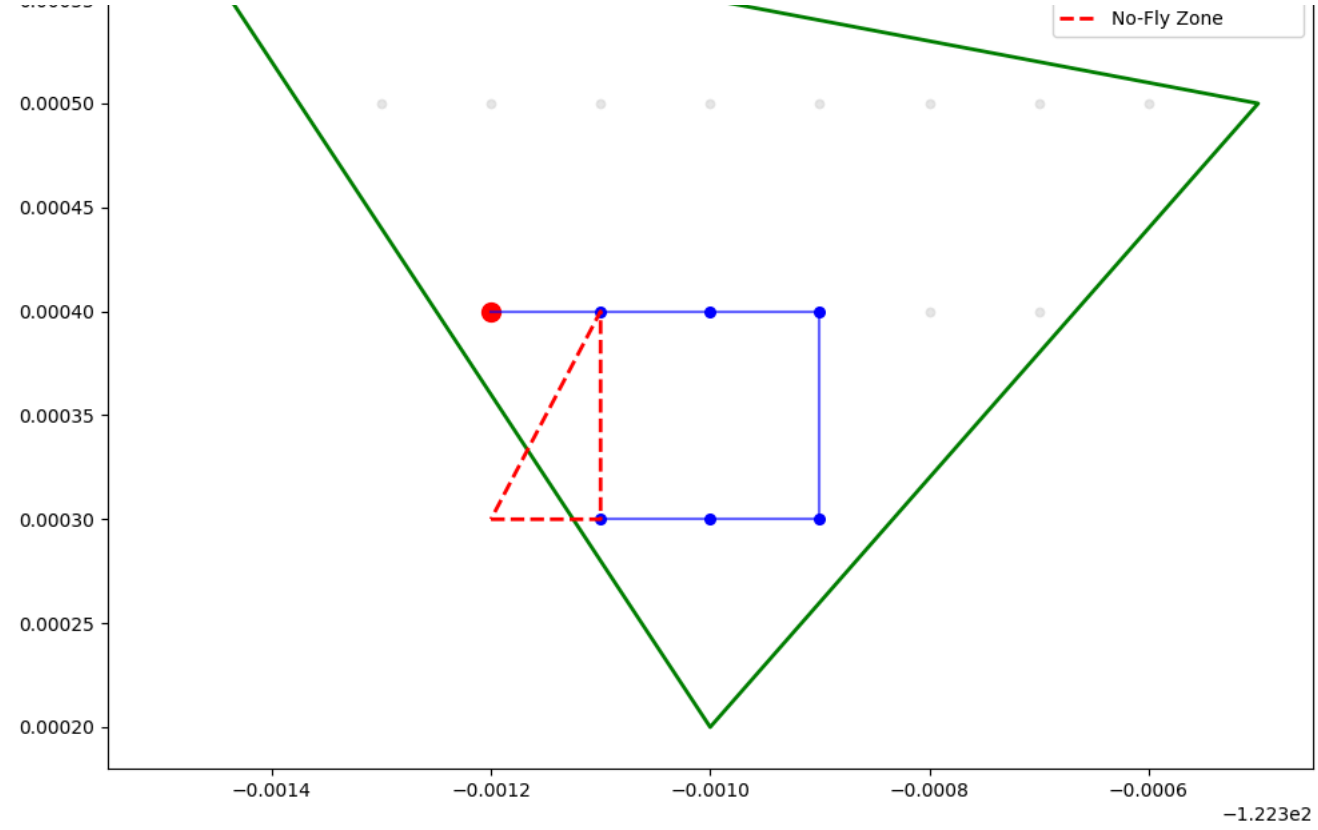



```
Step: Moved from (-122.30099999999999, 37.909400000000005) to (-122.30109999999999, 37.909400000000005)
Energy used: 65.29, remaining: 1663.75
Visited 6/17 points
Coverage: 35.3% (6/17)
```

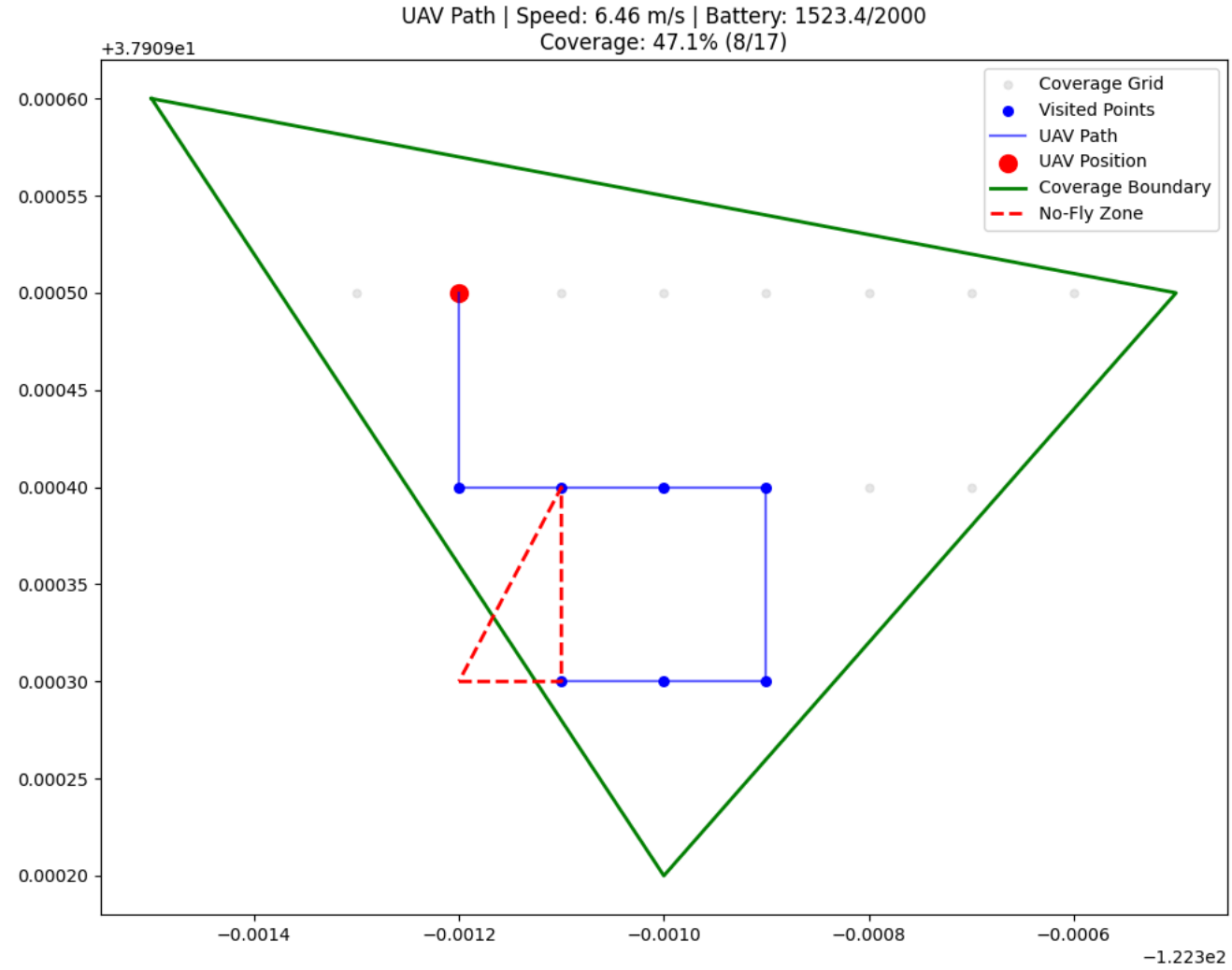



```
Step: Moved from (-122.30109999999999, 37.909400000000005) to (-122.3012, 37.909400000000005)
Energy used: 65.29, remaining: 1598.46
Visited 7/17 points
Coverage: 41.2% (7/17)
```

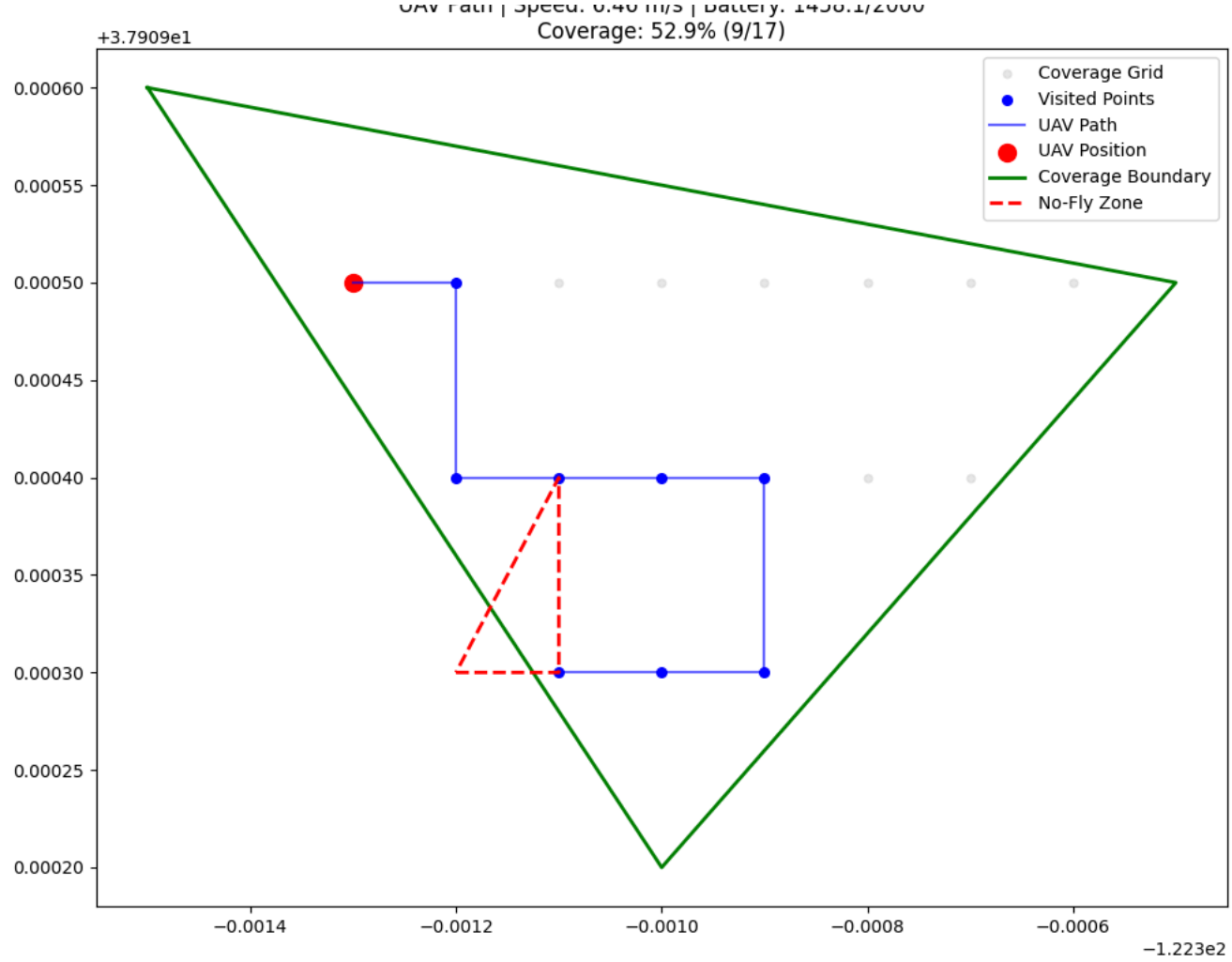




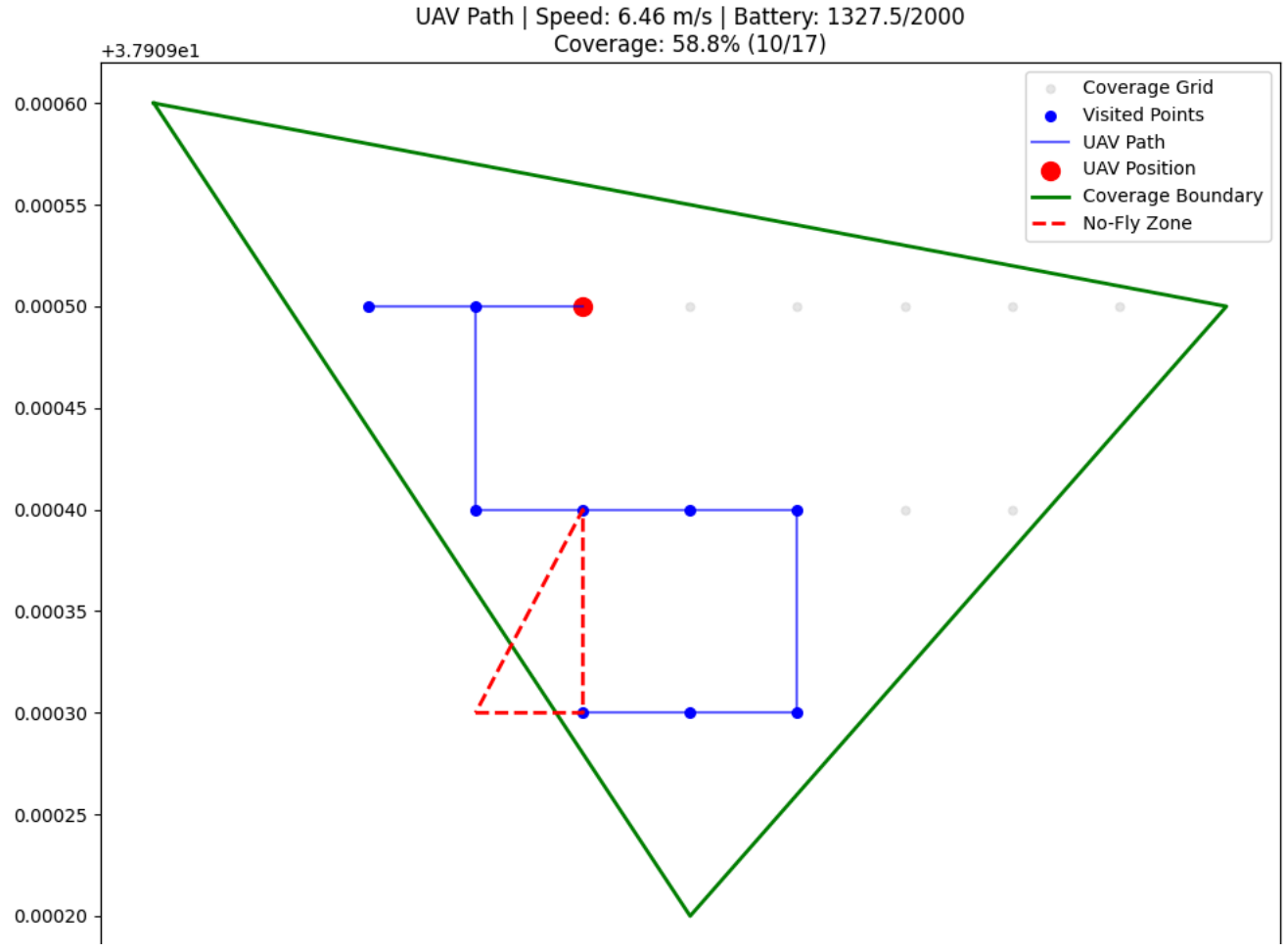

```
Step: Moved from (-122.3012, 37.909400000000005) to (-122.3012, 37.90950000000001)
Energy used: 75.08, remaining: 1523.37
Visited 8/17 points
Coverage: 47.1% (8/17)
```

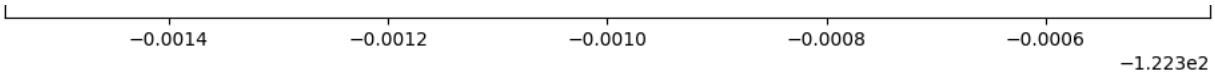



```
Step: Moved from (-122.3012, 37.90950000000001) to (-122.3013, 37.90950000000001)
Energy used: 65.29, remaining: 1458.08
Visited 9/17 points
Coverage: 52.9% (9/17)
```

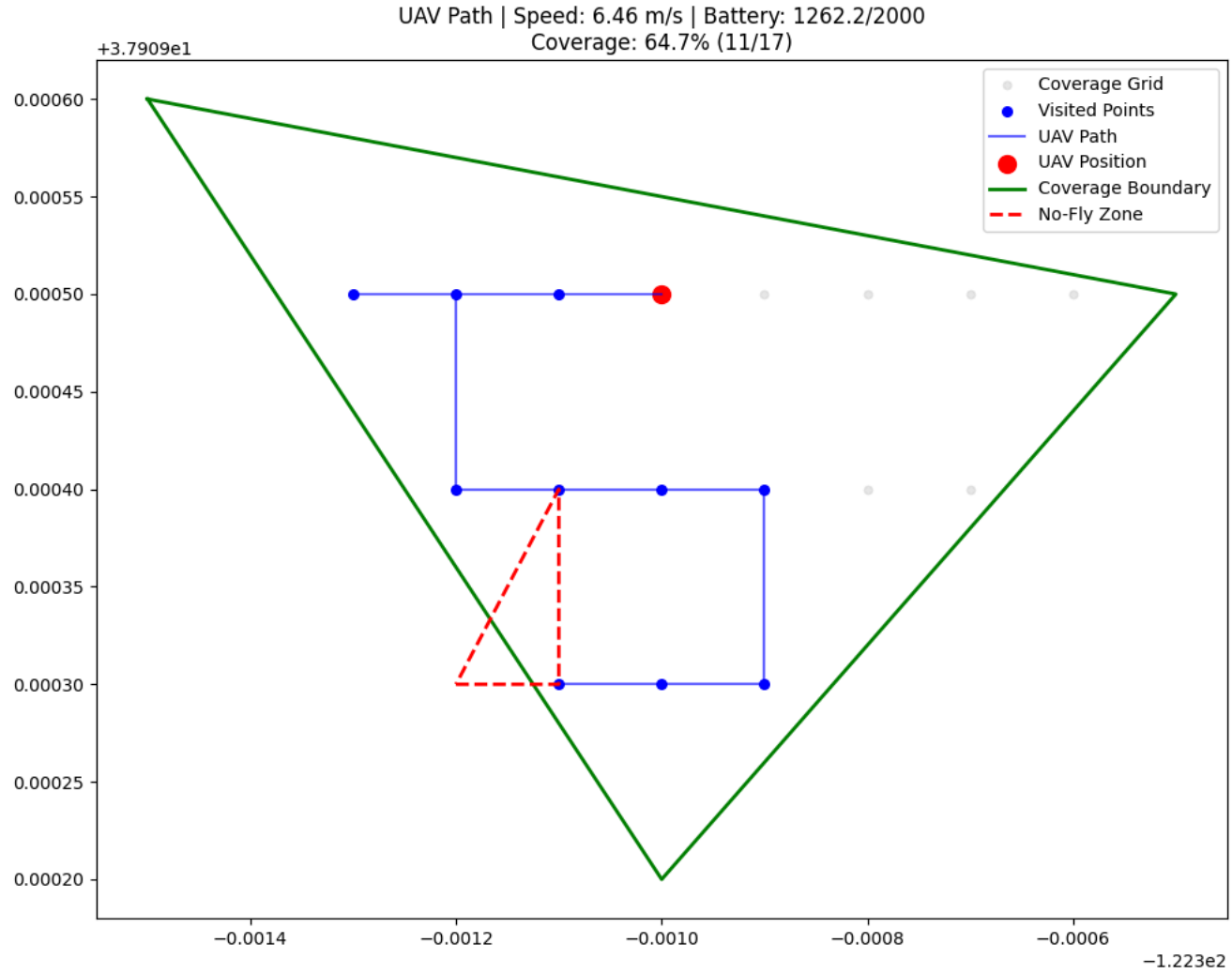



```
Step: Moved from (-122.3013, 37.90950000000001) to (-122.30109999999999, 37.90950000000001)
Energy used: 130.58, remaining: 1327.50
Visited 10/17 points
Coverage: 58.8% (10/17)
```

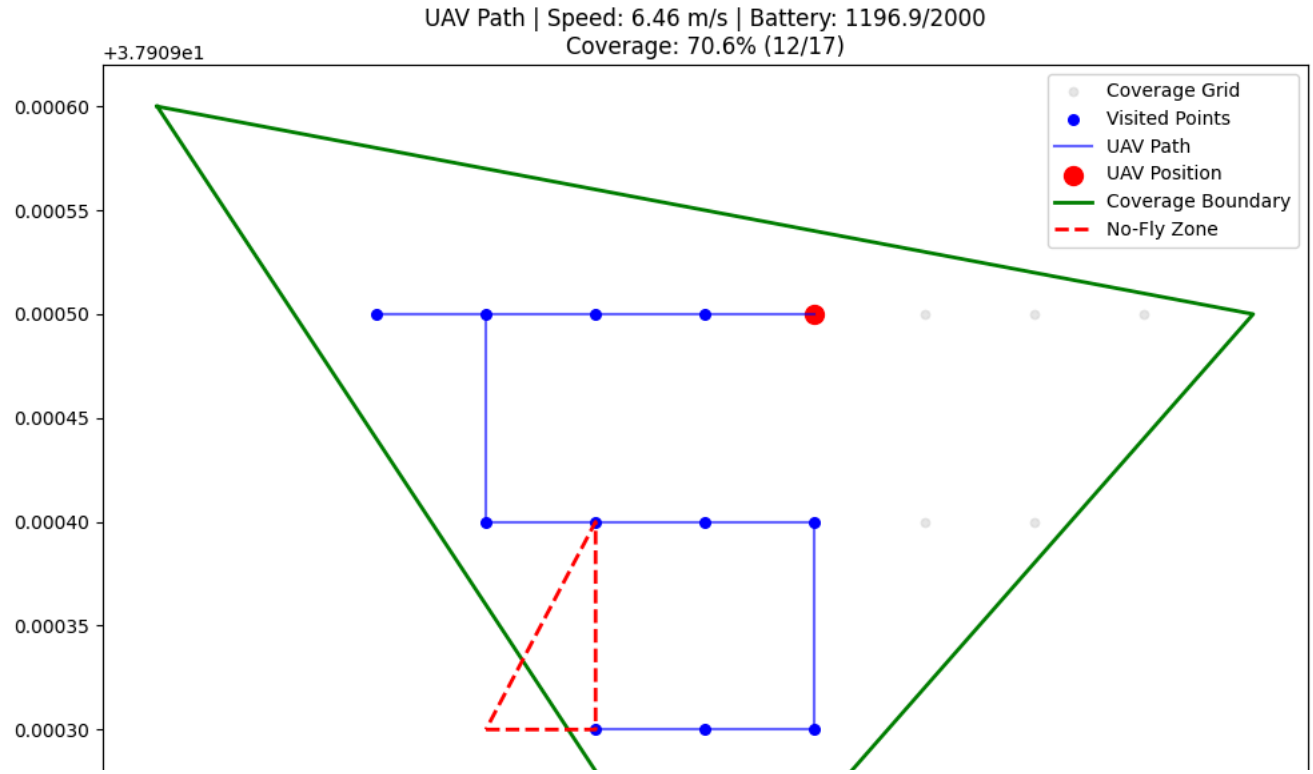


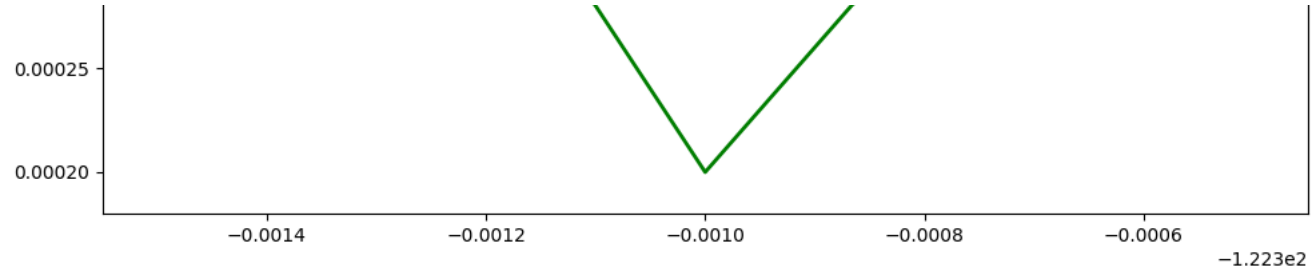



```
Step: Moved from (-122.30109999999999, 37.90950000000001) to (-122.30099999999999, 37.90950000000001)
Energy used: 65.29, remaining: 1262.21
Visited 11/17 points
Coverage: 64.7% (11/17)
```

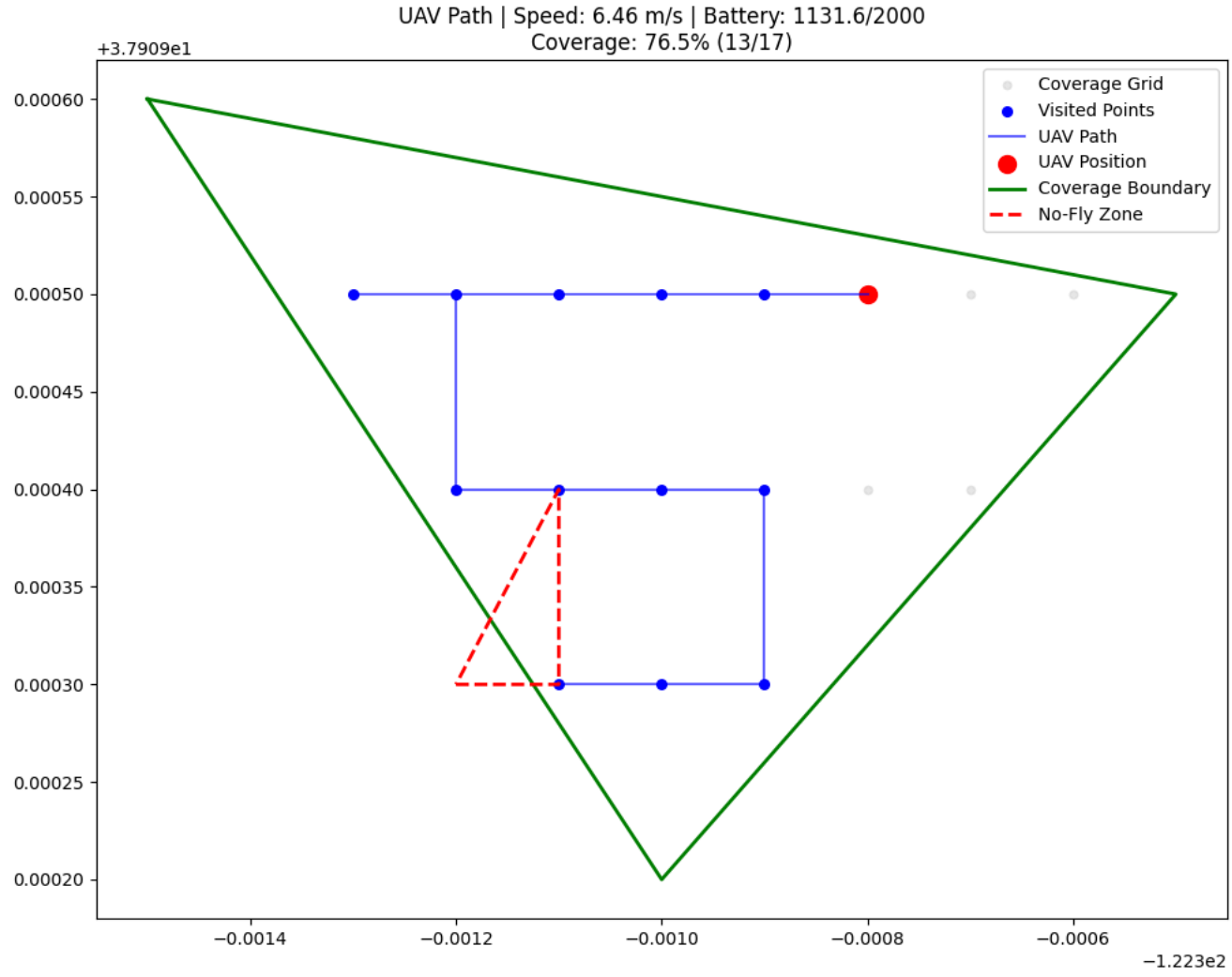



```
Step: Moved from (-122.30099999999999, 37.90950000000001) to (-122.30089999999998, 37.90950000000001)
Energy used: 65.29, remaining: 1196.92
Visited 12/17 points
Coverage: 70.6% (12/17)
```

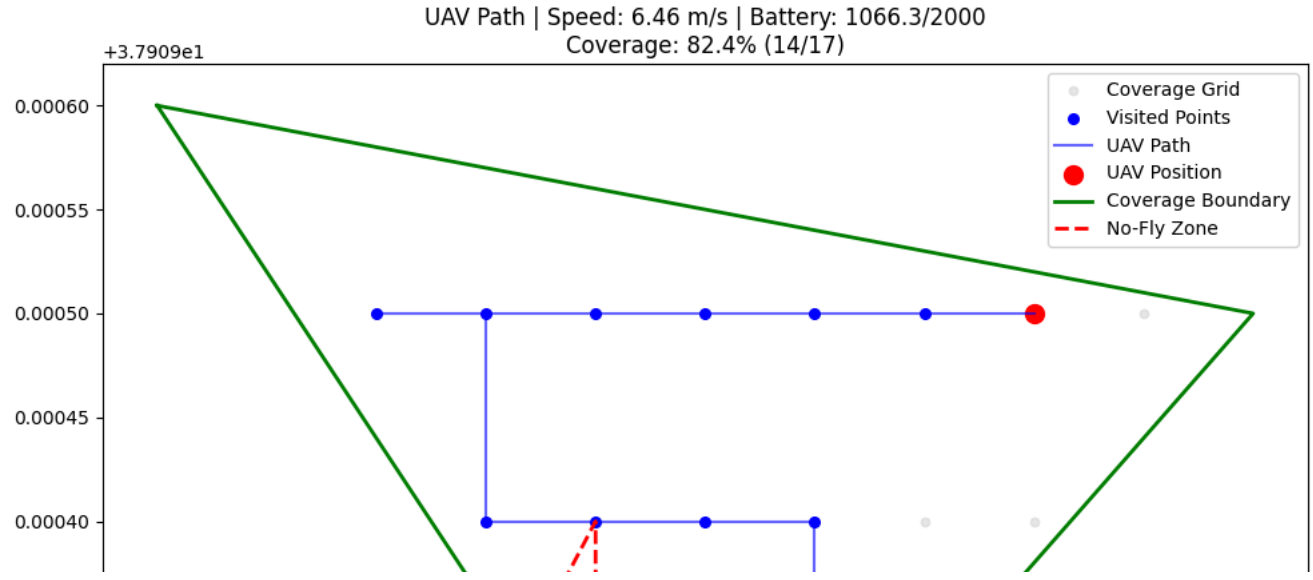


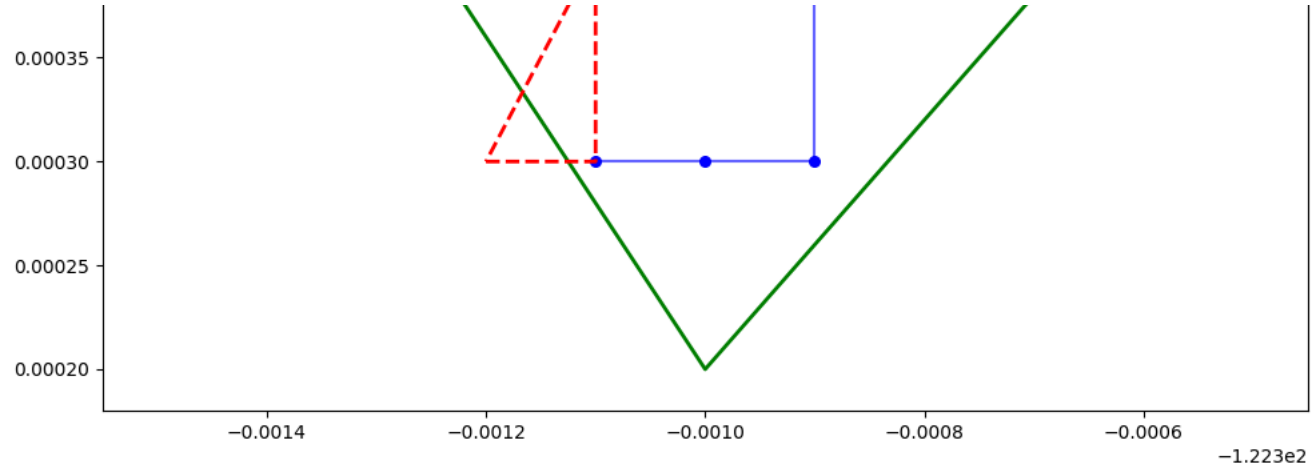



```
Step: Moved from (-122.30089999999998, 37.90950000000001) to (-122.30079999999998, 37.90950000000001)
Energy used: 65.29, remaining: 1131.63
Visited 13/17 points
Coverage: 76.5% (13/17)
```

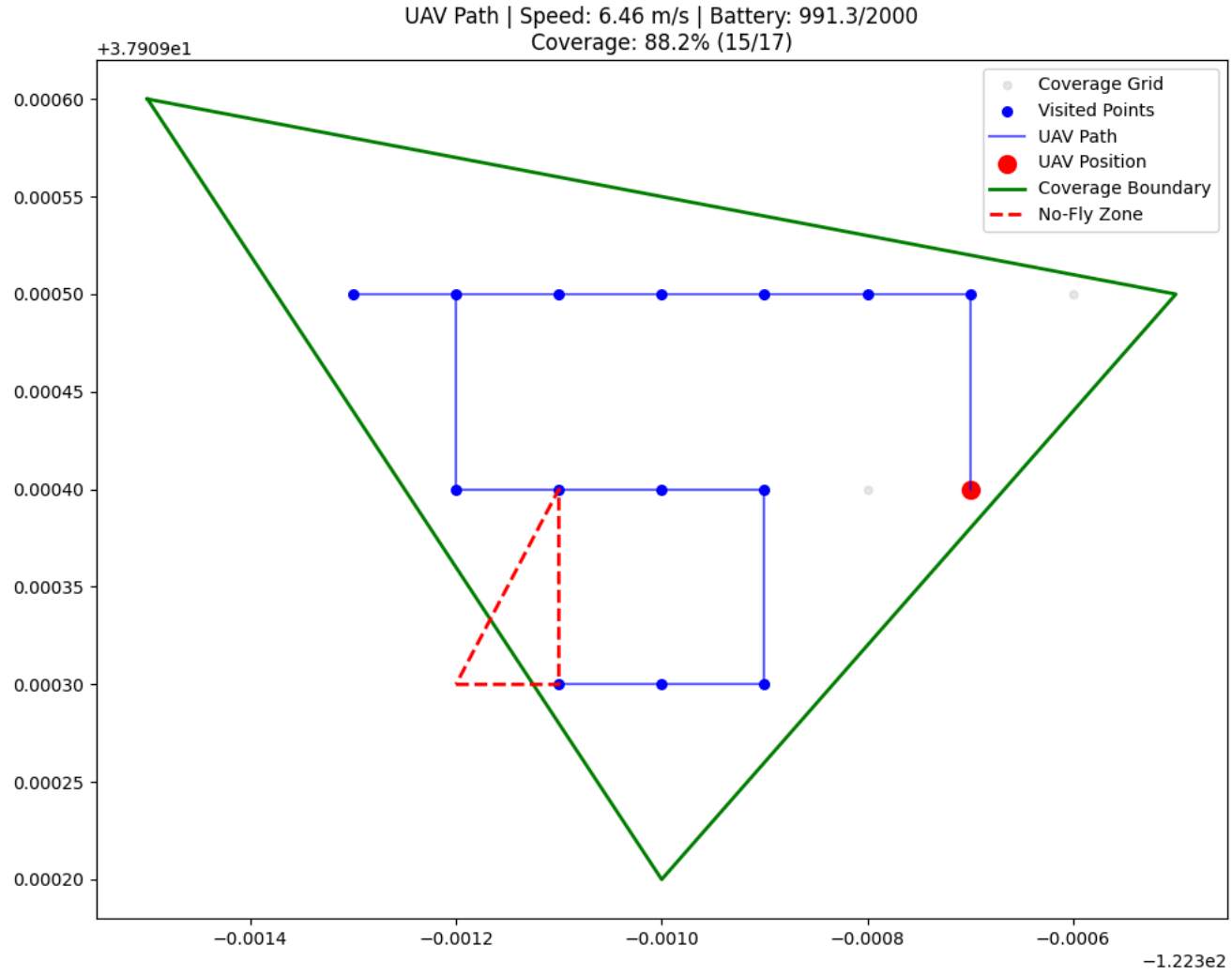



```
Step: Moved from (-122.30079999999998, 37.90950000000001) to (-122.30069999999998, 37.90950000000001)
Energy used: 65.29, remaining: 1066.34
Visited 14/17 points
Coverage: 82.4% (14/17)
```

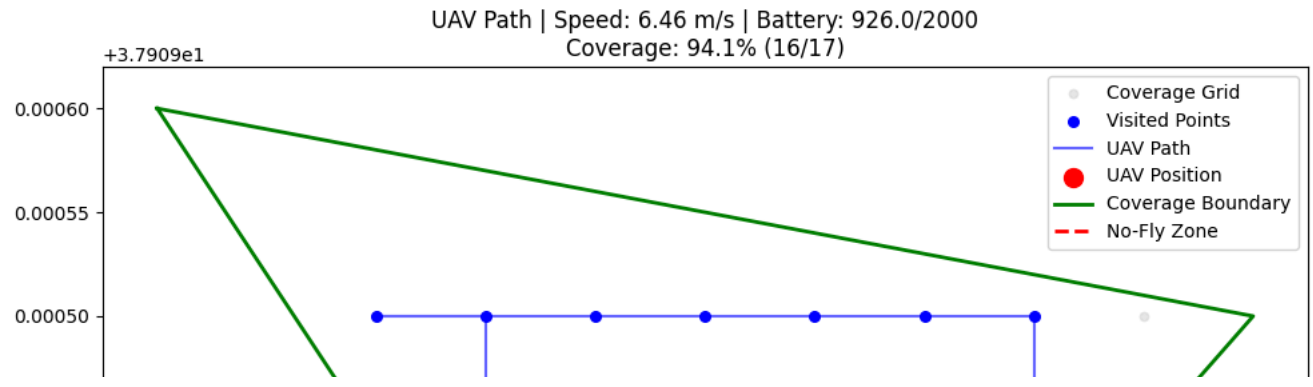


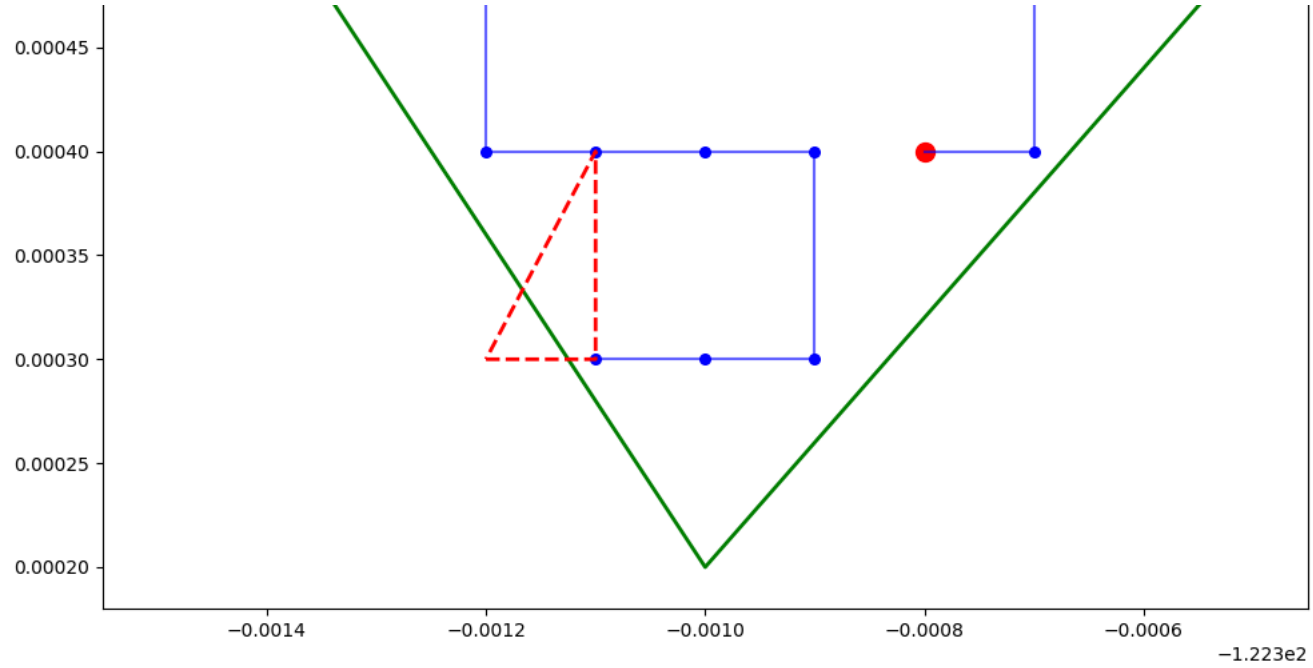



```
Step: Moved from (-122.30069999999998, 37.90950000000001) to (-122.30069999999998, 37.909400000000005)
Energy used: 75.08, remaining: 991.25
Visited 15/17 points
Coverage: 88.2% (15/17)
```

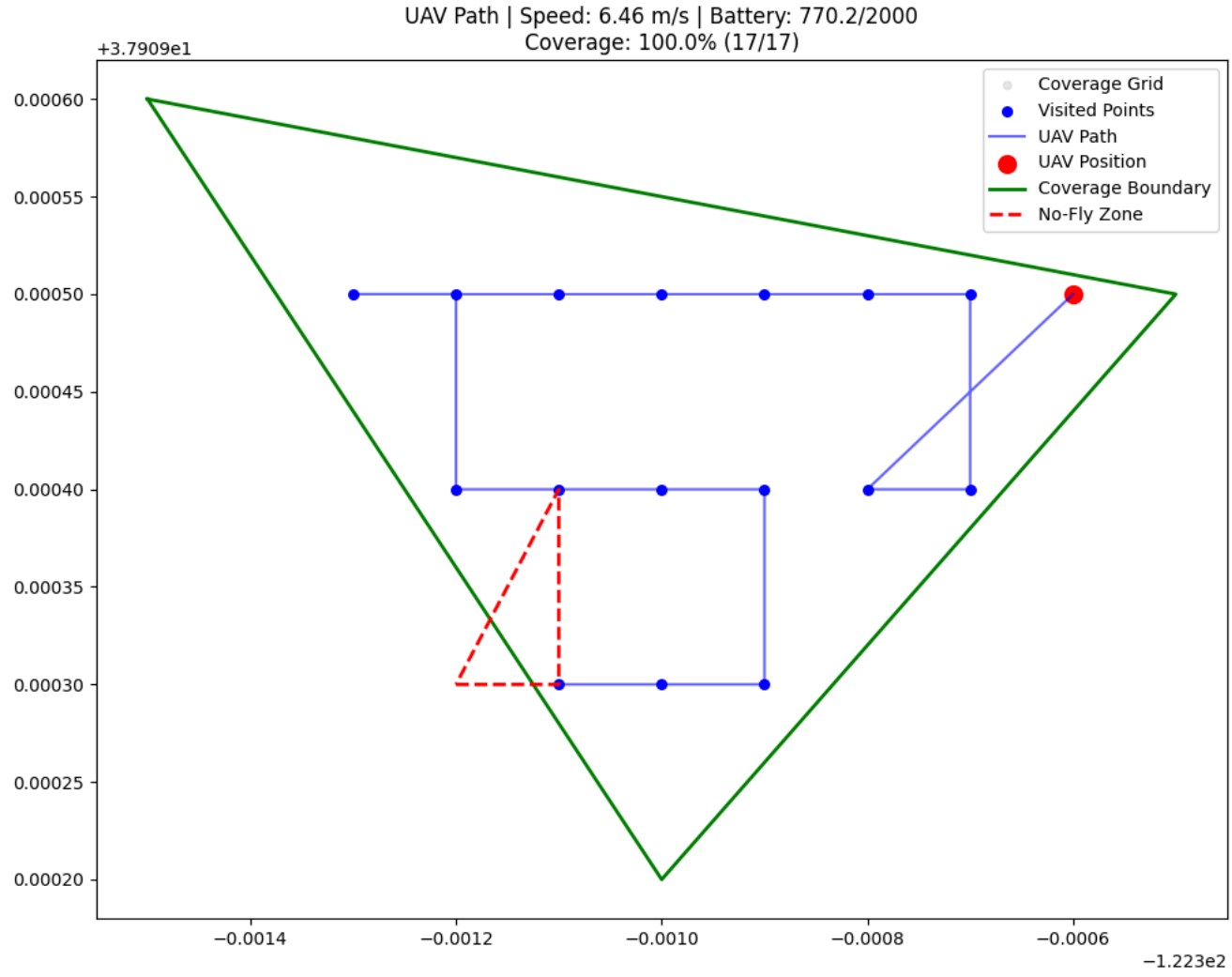



```
Step: Moved from (-122.30069999999998, 37.909400000000005) to (-122.30079999999998, 37.909400000000005)
Energy used: 65.29, remaining: 925.96
Visited 16/17 points
Coverage: 94.1% (16/17)
```






```
Step: Moved from (-122.30079999999998, 37.909400000000005) to (-122.30059999999997, 37.90950000000001)
Energy used: 155.79, remaining: 770.17
Visited 17/17 points
Coverage: 100.0% (17/17)
```




```
Final coverage: 100.0% (17/17)
Remaining battery: 770.2/2000
```

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from shapely.geometry import Polygon, Point, LineString
import random
import time
import math
from collections import deque
import heapq

class AdaptiveUAVCoverageEnv(gym.Env):
    def __init__(self, coverage_area, no_fly_zones, wind_data, payload_weight, battery_capacity, current_battery):
        super(AdaptiveUAVCoverageEnv, self).__init__()
        print("Initializing Adaptive UAV Environment...")

        # Convert coverage area and no-fly zones into polygons
        self.coverage_area = Polygon([(p["lon"], p["lat"]) for p in coverage_area])
        self.no_fly_zones = [Polygon([(p["lon"], p["lat"]) for p in zone]) for zone in no_fly_zones]

        # Wind data
        self.wind_data = wind_data

        self.energy_estimates = []

        # Calculate area dimensions and area
        minx, miny, maxx, maxy = self.coverage_area.bounds
        self.width = maxx - minx
        self.height = maxy - miny
        self.area = self.coverage_area.area

        # UAV Parameters
        self.payload_weight = payload_weight
        self.battery_capacity = battery_capacity
        self.current_battery = current_battery
        self.initial_battery = current_battery

        # Dynamic parameters
        self.energy_per_area_unit = None  # Will calculate after first movements
        self.coverage_radius = None       # Dynamic coverage radius

        # Initialize base grid size
        self.base_grid_size = min(self.width, self.height) / 10
        self.min_grid_size = self.base_grid_size / 5
        self.max_grid_size = self.base_grid_size * 2

        # Initialize UAV speed
        self.uav_speed = self._compute_dynamic_speed()

        # Generate initial grid with base size
        self.grid_size = self.base_grid_size
        self.grid = self._generate_grid(self.grid_size)

        # Make sure we have at least one point
        if len(self.grid) == 0:
            center_x = (minx + maxx) / 2
            center_y = (miny + maxy) / 2
            print(f"WARNING: No valid grid points generated. Adding center point at ({center_x}, {center_y}).")
            self.grid.append((center_x, center_y))

        # Initialize coverage tracking
        self.position = self.grid[0]
        self.visited = set([self.position])
        self.coverage_map = {}  # Maps points to their coverage radius
        self.coverage_map[self.position] = self._get_current_coverage_radius()

        # Visualization data
        self.trajectory = [self.position]
        self.energy_usage_log = []
        self.speed_log = []
        self.coverage_radius_log = []

        # Debug counters
        self.stuck_count = 0
        self.last_position = None
        self.last_coverage_percentage = 0
        self.energy_estimates = []

        # Set up observation and action spaces
        self.action_space = gym.spaces.Discrete(8)  # 8 directions
        self.observation_space = gym.spaces.Box(low=-1, high=1, shape=(6,), dtype=np.float32)

        # Algorithm settings
```

```
        self.adaptive_step_size = True
        self.dynamic_speed = True
        self.energy_reserve_percentage = 0.1  # Keep 10% battery reserve for emergencies

        print(f"Environment initialized with {len(self.grid)} grid points.")
        print(f"Coverage area: {self.area:.6f} square units")
        print(f"Initial grid size: {self.grid_size:.6f}")

    def _compute_dynamic_speed(self):
        """Adjust speed based on battery level, payload, and wind resistance."""
        base_speed = 10

        # Battery factor (higher speed when more battery available)
        battery_factor = 0.7 + (0.3 * self.current_battery / self.battery_capacity)

        # Payload factor (lower speed with heavier payload)
        payload_factor = max(0.5, 1 - (self.payload_weight / 20))

        # Wind effect
        wind_effect = self._compute_wind_effect()
        wind_factor = max(0.5, 1 - wind_effect)

        # Calculate speed
        speed = base_speed * battery_factor * payload_factor * wind_factor

        return max(2, min(15, speed))

    def _compute_wind_effect(self):
        """Calculate wind effect based on current conditions."""
        wind_direction = np.deg2rad(self.wind_data["direction"])
        move_angle = np.arctan2(1, 1)  # Default angle
        wind_effect = np.cos(wind_direction - move_angle) * self.wind_data["speed"] / 10
        return abs(wind_effect)

    def _generate_grid(self, grid_size):
        """Generate a grid within the coverage area, avoiding no-fly zones."""
        minx, miny, maxx, maxy = self.coverage_area.bounds
        grid = []
        x, y = minx, miny

        while y <= maxy:
            while x <= maxx:
                point = Point(x, y)
                if self.coverage_area.contains(point) and not any(zone.contains(point) for zone in self.no_fly_zones):
                    grid.append((x, y))
                x += grid_size
            x = minx
            y += grid_size

        print(f"Generated {len(grid)} grid points with grid size {grid_size}")
        return grid

    def _get_current_coverage_radius(self):
        """Calculate the coverage radius based on current energy and remaining area."""
        # Base coverage radius
        base_radius = self.grid_size / 2

        # Adjust radius based on battery percentage
        battery_percentage = self.current_battery / self.battery_capacity

        # If we have energy estimates, use them to predict needed radius
        if self.energy_estimates:
            avg_energy_per_unit = sum(self.energy_estimates) / len(self.energy_estimates)
            remaining_area = self.area * (1 - self._calculate_coverage_percentage())
            remaining_energy = self.current_battery - (self.battery_capacity * self.energy_reserve_percentage)

            # If we have plenty of energy, use larger coverage radius
            if remaining_energy > avg_energy_per_unit * remaining_area:
                adaptive_radius = base_radius * (1 + battery_percentage)
            else:
                # Calculate minimum radius needed to cover remaining area with available energy
                adaptive_radius = max(
                    self.min_grid_size / 2,
                    base_radius * min(1.0, remaining_energy / (avg_energy_per_unit * remaining_area))
                )
        else:
            # No energy estimates yet, use battery percentage
            adaptive_radius = base_radius * max(0.5, min(2.0, battery_percentage * 2))

        return max(self.min_grid_size / 2, min(adaptive_radius, self.max_grid_size))

    def _calculate_coverage_percentage(self):
        """Calculate the current coverage percentage."""
```

```
        if not self.coverage_map:
            return 0

        # Create a buffer around each visited point based on its coverage radius
        covered_area = None
        for point, radius in self.coverage_map.items():
            point_obj = Point(point)
            buffer = point_obj.buffer(radius)

            if covered_area is None:
                covered_area = buffer
            else:
                covered_area = covered_area.union(buffer)

        # Calculate intersection with coverage area to get actual coverage
        if covered_area:
            coverage_intersection = self.coverage_area.intersection(covered_area)
            return min(1.0, coverage_intersection.area / self.area)
        return 0

    def _is_point_covered(self, point):
        """Check if a point is already covered by existing coverage."""
        point_obj = Point(point)

        for visited_point, radius in self.coverage_map.items():
            visited_point_obj = Point(visited_point)
            if point_obj.distance(visited_point_obj) <= radius:
                return True

        return False

    def _find_best_next_point(self):
        """Find the best next point to visit for optimal coverage."""
        coverage_percentage = self._calculate_coverage_percentage()

        # If we're already at 100% coverage, return None
        if coverage_percentage >= 0.99:
            return None

        # Get current position
        current_x, current_y = self.position

        # Calculate current coverage radius
        current_radius = self._get_current_coverage_radius()

        # Generate potential next positions based on current grid size
        directions = [
            (1, 0), (-1, 0), (0, 1), (0, -1),
            (1, 1), (-1, -1), (1, -1), (-1, 1)
        ]

        # Adaptive step size based on current energy and coverage
        if self.adaptive_step_size:
            # Smaller steps when battery is low or coverage is high
            step_multiplier = max(0.5, min(2.0, self.current_battery / self.battery_capacity))

            # Adjust step size based on coverage percentage
            if coverage_percentage > 0.8:
                step_multiplier *= 0.5  # Use smaller steps for fine-tuning at high coverage
        else:
            step_multiplier = 1.0

        # Step size is determined by coverage radius and step multiplier
        step_size = current_radius * 1.5 * step_multiplier

        # Calculate potential next positions
        candidates = []
        for dx, dy in directions:
            next_x = current_x + dx * step_size
            next_y = current_y + dy * step_size
            next_point = (next_x, next_y)

            # Check if the point is valid (inside coverage area, outside no-fly zones)
            point_obj = Point(next_point)
            if not self.coverage_area.contains(point_obj) or any(zone.contains(point_obj) for zone in self.no_fly_zones):
                continue

            # Calculate energy required to move to this point
            energy_required = self._compute_energy_usage(self.position, next_point)

            # Skip if we don't have enough energy
            if energy_required >= self.current_battery:
                continue
```

```
                continue

            # Calculate how much new area this point would cover
            coverage_gain = self._estimate_coverage_gain(next_point, current_radius)

            # Calculate a score based on energy efficiency and coverage gain
            if coverage_gain > 0:
                score = coverage_gain / energy_required
            else:
                score = 0

            candidates.append((next_point, score, energy_required, coverage_gain))

        # If we couldn't find any valid next points with the current step size,
        # try with a smaller step size (half the current one)
        if not candidates and step_size > self.min_grid_size:
            reduced_step_size = max(self.min_grid_size, step_size / 2)
            for dx, dy in directions:
                next_x = current_x + dx * reduced_step_size
                next_y = current_y + dy * reduced_step_size
                next_point = (next_x, next_y)

                # Check if the point is valid
                point_obj = Point(next_point)
                if not self.coverage_area.contains(point_obj) or any(zone.contains(point_obj) for zone in self.no_fly_zones):
                    continue

                energy_required = self._compute_energy_usage(self.position, next_point)

                if energy_required >= self.current_battery:
                    continue

                coverage_gain = self._estimate_coverage_gain(next_point, current_radius)

                if coverage_gain > 0:
                    score = coverage_gain / energy_required
                else:
                    score = 0

                candidates.append((next_point, score, energy_required, coverage_gain))

        # If we still can't find any valid next points, try desperate measures
        if not candidates:
            print("WARNING: Couldn't find any valid next points. Trying last resort measures.")

            # Find uncovered areas and try to reach them
            min_distance = float('inf')
            best_uncovered_point = None

            # Sample points throughout the coverage area
            sample_size = max(100, int(self.area * 10000))  # Adjust sampling density
            minx, miny, maxx, maxy = self.coverage_area.bounds

            for _ in range(sample_size):
                # Generate random point within bounds
                rand_x = random.uniform(minx, maxx)
                rand_y = random.uniform(miny, maxy)
                sample_point = (rand_x, rand_y)

                # Check if point is valid and not already covered
                point_obj = Point(sample_point)
                if not self.coverage_area.contains(point_obj) or any(zone.contains(point_obj) for zone in self.no_fly_zones):
                    continue

                if self._is_point_covered(sample_point):
                    continue

                # Calculate distance to this point
                distance = math.sqrt((self.position[0] - sample_point[0])**2 + (self.position[1] - sample_point[1])**2)

                # Calculate energy required
                energy_required = self._compute_energy_usage(self.position, sample_point)

                # If we can reach it with our energy, consider it
                if energy_required < self.current_battery and distance < min_distance:
                    min_distance = distance
                    best_uncovered_point = sample_point

            # If we found an uncovered point we can reach, use it
            if best_uncovered_point:
                energy_required = self._compute_energy_usage(self.position, best_uncovered_point)
                coverage_gain = self._estimate_coverage_gain(best_uncovered_point, current_radius)
```

```
                if coverage_gain > 0:
                    score = coverage_gain / energy_required
                else:
                    score = 0

                candidates.append((best_uncovered_point, score, energy_required, coverage_gain))

        # Sort candidates by score (highest first)
        candidates.sort(key=lambda x: x[1], reverse=True)

        # Return the best candidate, or None if there are no valid candidates
        return candidates[0][0] if candidates else None

    def _estimate_coverage_gain(self, point, radius):
        """Estimate how much new area would be covered by visiting this point."""
        point_obj = Point(point)
        new_buffer = point_obj.buffer(radius)

        # Calculate current covered area
        current_covered_area = None
        for visited_point, visited_radius in self.coverage_map.items():
            visited_point_obj = Point(visited_point)
            buffer = visited_point_obj.buffer(visited_radius)

            if current_covered_area is None:
                current_covered_area = buffer
            else:
                current_covered_area = current_covered_area.union(buffer)

        if current_covered_area is None:
            # Nothing covered yet, so this is all new coverage
            intersection_with_coverage_area = self.coverage_area.intersection(new_buffer)
            return intersection_with_coverage_area.area
        else:
            # Calculate the new area this point would add
            combined_area = current_covered_area.union(new_buffer)
            new_area = combined_area.area - current_covered_area.area

            # Ensure we only count area within the coverage boundary
            intersection_with_coverage_area = self.coverage_area.intersection(combined_area)
            current_intersection = self.coverage_area.intersection(current_covered_area)

            return intersection_with_coverage_area.area - current_intersection.area

    def _compute_energy_usage(self, from_pos, to_pos):
        """Calculate energy usage for movement between two positions."""
        # Calculate distance
        distance = math.sqrt((from_pos[0] - to_pos[0])**2 + (from_pos[1] - to_pos[1])**2)
        if distance == 0:
            return 0

        # Direction of movement
        dx, dy = to_pos[0] - from_pos[0], to_pos[1] - from_pos[1]
        move_angle = math.atan2(dy, dx) if (dx != 0 or dy != 0) else 0

        # Wind effect based on movement direction
        wind_direction = np.deg2rad(self.wind_data["direction"])
        wind_effect = math.cos(wind_direction - move_angle) * self.wind_data["speed"] / 10

        # Base power usage
        power_usage = 100 + (self.payload_weight * 0.5)

        # Wind resistance factor
        wind_resistance = 1 + abs(wind_effect) * 0.3

        # Speed factor (higher speed = more energy usage)
        speed_factor = self.uav_speed / 10

        # Distance factor
        distance_factor = distance / self.grid_size

        # Calculate final energy usage
        energy_usage = power_usage * wind_resistance * speed_factor * distance_factor

        # Store energy usage per unit distance for future estimates
        if distance > 0:
            energy_per_unit = energy_usage / distance
            self.energy_estimates.append(energy_per_unit)
            # Keep only the last 20 estimates
            if len(self.energy_estimates) > 20:
                self.energy_estimates = self.energy_estimates[-20:]

        return energy_usage
```

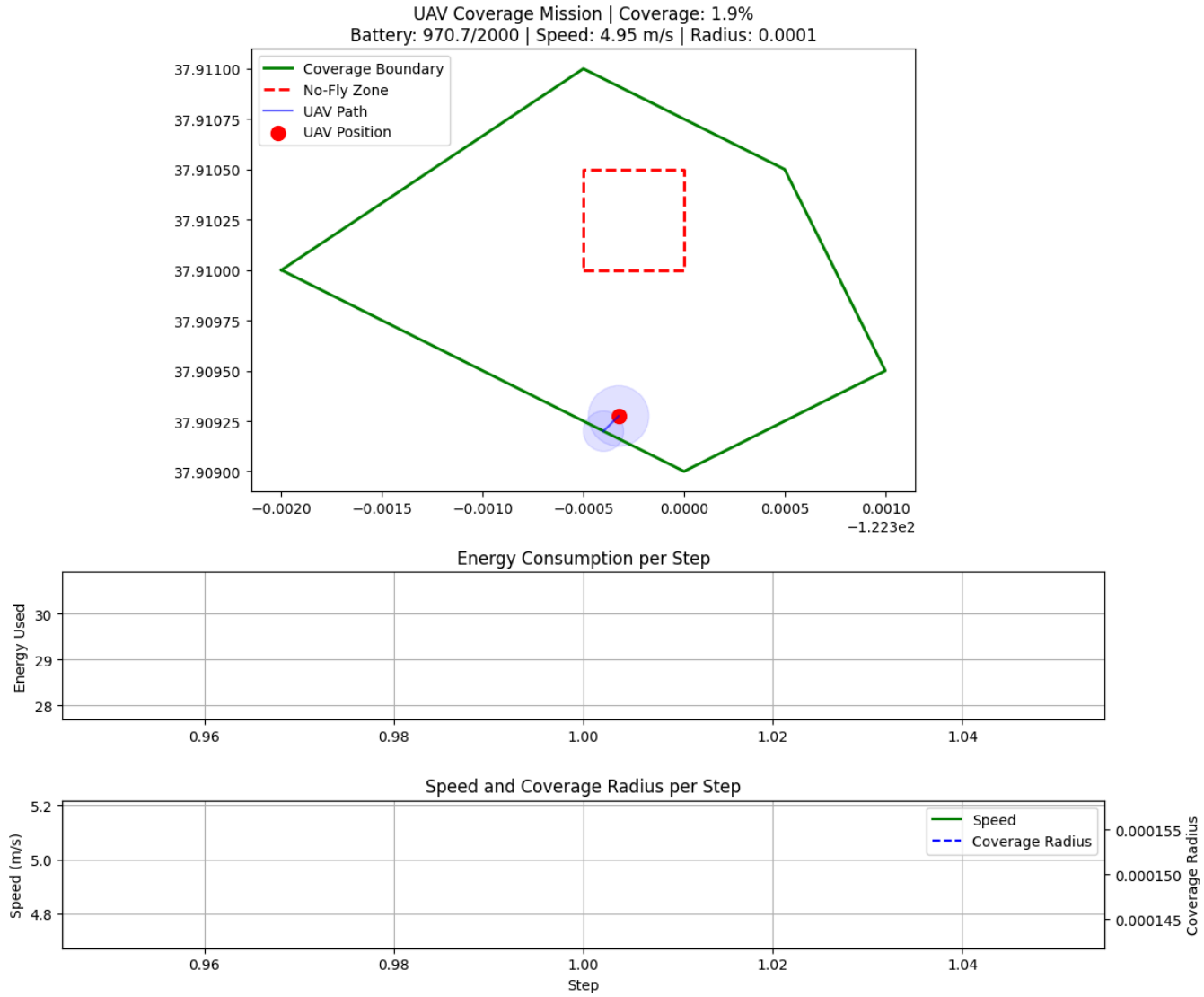
```
    def step(self, action=None):
        """Move UAV to next position for optimal coverage."""
        # Update UAV speed if dynamic speed is enabled
        if self.dynamic_speed:
            self.uav_speed = self._compute_dynamic_speed()

        # Find the best next point to visit
        next_position = self._find_best_next_point()

        # If no valid next point found, we're either done or out of energy
        if next_position is None:
            coverage_percentage = self._calculate_coverage_percentage()
            print(f"No valid next position found. Coverage: {coverage_percentage*100:.1f}%")

            # Check if we have satisfactory coverage
            if coverage_percentage >= 0.99:
                print("Achieved full coverage! Mission complete.")
                return self._get_observation(), 10, True, {"success": True}
            else:
                # Check if we're out of energy
                if self.current_battery < self.battery_capacity * 0.1:
                    print("Low battery! Unable to complete coverage.")
                    return self._get_observation(), -5, True, {"success": False, "reason": "low_battery"}
                else:
                    print("Unable to find path to uncovered areas.")
                    return self._get_observation(), -5, True, {"success": False, "reason": "no_path"}

        # Calculate energy for this move
        energy_used = self._compute_energy_usage(self.position, next_position)

        # Check if we have enough energy
        if energy_used <= self.current_battery:
            prev_position = self.position
            self.position = next_position
            self.trajectory.append(self.position)

            # Update coverage map with this position and its coverage radius
            current_radius = self._get_current_coverage_radius()
            self.coverage_map[self.position] = current_radius

            # Update tracking data
            self.visited.add(self.position)
            self.current_battery -= energy_used

            # Log data for visualization
            self.energy_usage_log.append(energy_used)
            self.speed_log.append(self.uav_speed)
            self.coverage_radius_log.append(current_radius)

            # Calculate reward
            coverage_percentage = self._calculate_coverage_percentage()
            coverage_gain = coverage_percentage - self.last_coverage_percentage
            self.last_coverage_percentage = coverage_percentage

            reward = coverage_gain * 100  # Scale reward based on coverage gain

            print(f"Step: Moved from {prev_position} to {next_position}")
            print(f"Energy used: {energy_used:.2f}, remaining: {self.current_battery:.2f}")
            print(f"Coverage: {coverage_percentage*100:.1f}%")
            print(f"Coverage radius: {current_radius:.4f}")
        else:
            # Not enough energy
            print(f"Not enough energy to move to {next_position}.")
            print(f"Required: {energy_used:.2f}, Available: {self.current_battery:.2f}")
            reward = -1

        # Check if done (full coverage or out of energy)
        done = (self._calculate_coverage_percentage() >= 0.99) or (self.current_battery <= 0)

        return self._get_observation(), reward, done, {}

    def _get_observation(self):
        """Get current observation state."""
        coverage_percentage = self._calculate_coverage_percentage()

        return np.array([
            self.position[0],
            self.position[1],
            self.current_battery / self.battery_capacity,
            self.uav_speed / 15,
            coverage_percentage,
            self._get_current_coverage_radius() / self.max_grid_size
```

```
            self._get_current_coverage_radius() / self.max_grid_size
        ], dtype=np.float32)

    def reset(self):
        """Reset UAV for a new mission."""
        # Reset UAV state
        minx, miny, maxx, maxy = self.coverage_area.bounds
        self.position = self.grid[0]
        self.visited = set([self.position])
        self.trajectory = [self.position]
        self.current_battery = self.initial_battery
        self.uav_speed = self._compute_dynamic_speed()

        # Reset coverage tracking
        current_radius = self._get_current_coverage_radius()
        self.coverage_map = {self.position: current_radius}
        self.last_coverage_percentage = 0

        # Reset logs
        self.energy_usage_log = []
        self.speed_log = []
        self.coverage_radius_log = []
        self.energy_estimates = []

        return self._get_observation()

    def render(self, mode='human'):
        """Visualize UAV coverage and path."""
        plt.figure(figsize=(12, 10))

        # Create subplots: main plot and resource usage
        gs = plt.GridSpec(3, 1, height_ratios=[3, 1, 1])
        ax1 = plt.subplot(gs[0])
        ax2 = plt.subplot(gs[1])
        ax3 = plt.subplot(gs[2])

        # Main coverage plot
        # Plot coverage area boundary
        x, y = self.coverage_area.exterior.xy
        ax1.plot(x, y, c="green", linewidth=2, label="Coverage Boundary")

        # Plot no-fly zones
        for i, zone in enumerate(self.no_fly_zones):
            x, y = zone.exterior.xy
            label = "No-Fly Zone" if i == 0 else ""
            ax1.plot(x, y, c="red", linestyle='--', linewidth=2, label=label)

        # Plot the coverage circles for each visited point
        for point, radius in self.coverage_map.items():
            circle = plt.Circle(point, radius, color='blue', fill=True, alpha=0.1)
            ax1.add_patch(circle)

        # Plot the trajectory path
        if len(self.trajectory) > 1:
            trajectory_x, trajectory_y = zip(*self.trajectory)
            ax1.plot(trajectory_x, trajectory_y, 'b-', linewidth=1.5, alpha=0.6, label="UAV Path")

        # Plot current position
        ax1.scatter(*self.position, c="red", s=100, label="UAV Position")

        # Set title with info
        coverage_percentage = self._calculate_coverage_percentage()
        ax1.set_title(f"UAV Coverage Mission | Coverage: {coverage_percentage*100:.1f}%\n"
                      f"Battery: {self.current_battery:.1f}/{self.battery_capacity} | "
                      f"Speed: {self.uav_speed:.2f} m/s | "
                      f"Radius: {self._get_current_coverage_radius():.4f}")

        ax1.set_aspect('equal')

        # Energy usage plot
        steps = range(1, len(self.energy_usage_log) + 1)
        if self.energy_usage_log:
            ax2.plot(steps, self.energy_usage_log, 'r-', label="Energy Usage")
            ax2.set_ylabel("Energy Used")
            ax2.set_title("Energy Consumption per Step")
            ax2.grid(True)

        # Speed and coverage radius plot
        if self.speed_log and self.coverage_radius_log:
            ax3.plot(steps, self.speed_log, 'g-', label="Speed")
            ax3.set_xlabel("Step")
            ax3.set_ylabel("Speed (m/s)")
            ax3.grid(True)
```

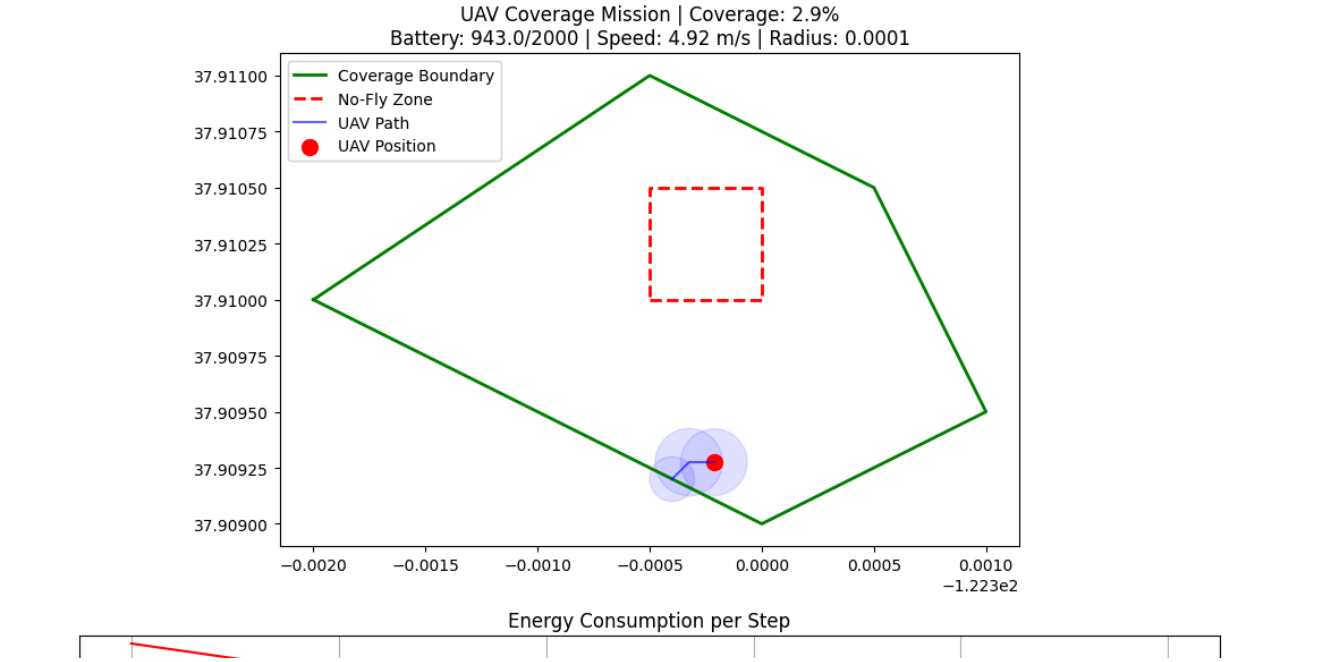
```
            # Twin axis for coverage radius
            ax3b = ax3.twinx()
            ax3b.plot(steps, self.coverage_radius_log, 'b--', label="Coverage Radius")
            ax3b.set_ylabel("Coverage Radius")

            # Combine legends
            lines1, labels1 = ax3.get_legend_handles_labels()
            lines2, labels2 = ax3b.get_legend_handles_labels()
            ax3.legend(lines1 + lines2, labels1 + labels2, loc='upper right')

            ax3.set_title("Speed and Coverage Radius per Step")

        # Handle legend for main plot
        handles, labels = ax1.get_legend_handles_labels()
        by_label = dict(zip(labels, handles))
        ax1.legend(by_label.values(), by_label.keys())

        plt.tight_layout()
        plt.show()
        plt.pause(0.2)
        plt.close()

# Example Usage with different coverage areas
def run_simulation(coverage_area, no_fly_zones, battery_capacity):
    wind_data = {"speed": 5, "direction": 90}

    # Initialize Environment
    env = AdaptiveUAVCoverageEnv(
        # coverage_area,
        # no_fly_zones,
        # wind_data,
        # payload_weight=2,
        # battery_capacity=battery_capacity,
        # current_battery=battery_capacity
        coverage_area=test_scenarios[0]["coverage_area"],
        no_fly_zones=test_scenarios[0]["no_fly_zones"],
        wind_data={"speed": 5, "direction": 90},  # Provide wind_data directly
        payload_weight=2,
        battery_capacity=2000,
        current_battery=1000
    )

    # Run Simulation
    obs = env.reset()
    done = False
    max_steps = 300  # Allow more steps for complex areas

    try:
        step_count = 0
        while not done and step_count < max_steps:
            # Use the algorithm's built-in decision making
            obs, reward, done, info = env.step(None)
            env.render()
            step_count += 1
            time.sleep(0.1)  # Slow down visualization

        if step_count >= max_steps:
            print(f"Simulation stopped after {max_steps} steps")

        coverage_percentage = env._calculate_coverage_percentage()
        print(f"Final coverage: {coverage_percentage*100:.1f}%")
        print(f"Remaining battery: {env.current_battery:.1f}/{env.battery_capacity}")

        # Return success status
        success = coverage_percentage >= 0.99
        return success, coverage_percentage, env.current_battery

    except Exception as e:
        print(f"Error during simulation: {e}")
        return False, 0, 0

# Define test scenarios
test_scenarios = [

    {
        "name": "Complex Polygon",
        "coverage_area": [
            {"lon": -122.3020, "lat": 37.9100},
            {"lon": -122.3010, "lat": 37.9095},
            {"lon": -122.3000, "lat": 37.9090},
            {"lon": -122.2990, "lat": 37.9095},
            {"lon": -122.2995, "lat": 37.9105},
```

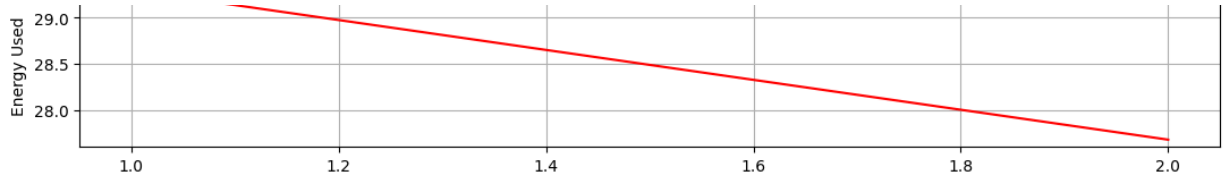
```
                {"lon": -122.3005, "lat": 37.9110},
                {"lon": -122.3020, "lat": 37.9100}
            ],
            "no_fly_zones": [
                [{"lon": -122.3005, "lat": 37.9100},
                 {"lon": -122.3000, "lat": 37.9100},
                 {"lon": -122.3000, "lat": 37.9105},
                 {"lon": -122.3005, "lat": 37.9105}]
            ],
            "battery_capacity": 200000
        }
    ]

    # Run the main test scenario
    run_simulation(
        coverage_area=test_scenarios[0]["coverage_area"],
        no_fly_zones=test_scenarios[0]["no_fly_zones"],
        battery_capacity=test_scenarios[0]["battery_capacity"]
    )
```

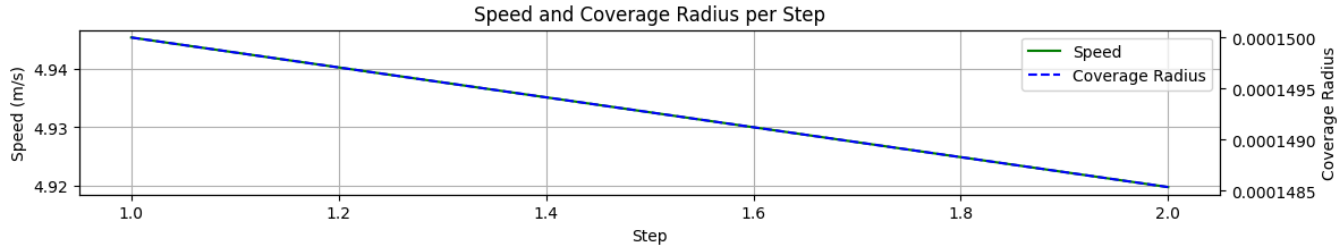
```
Initializing Adaptive UAV Environment...
Generated 78 grid points with grid size 0.00020000000000024442
Environment initialized with 78 grid points.
Coverage area: 0.000003 square units
Initial grid size: 0.000200
Step: Moved from (-122.30039999999995, 37.9092) to (-122.30032499999996, 37.909275)
Energy used: 29.30, remaining: 970.70
Coverage: 1.9%
Coverage radius: 0.0002
```



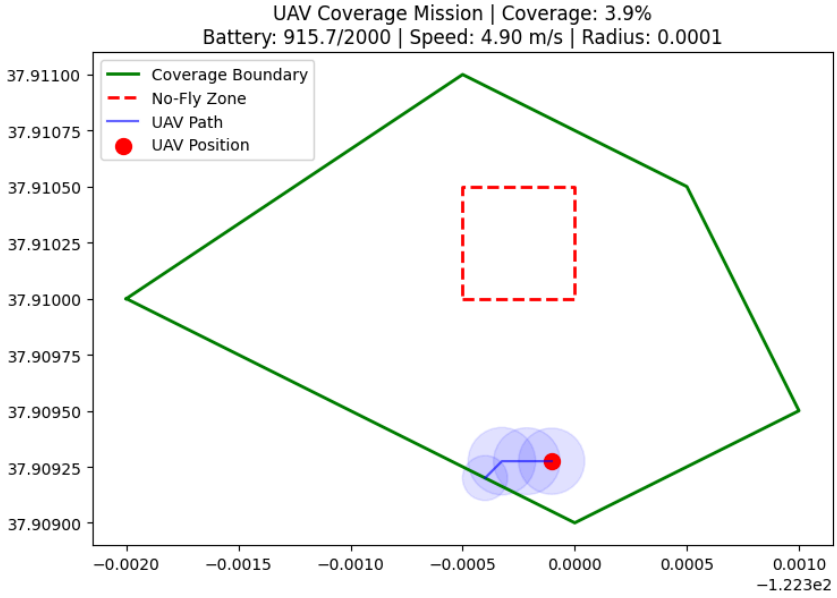

```
Step: Moved from (-122.30032499999996, 37.909275) to (np.float64(-122.30021359868712), np.float64(37.909275))
Energy used: 27.68, remaining: 943.02
Coverage: 2.9%
Coverage radius: 0.0001
```

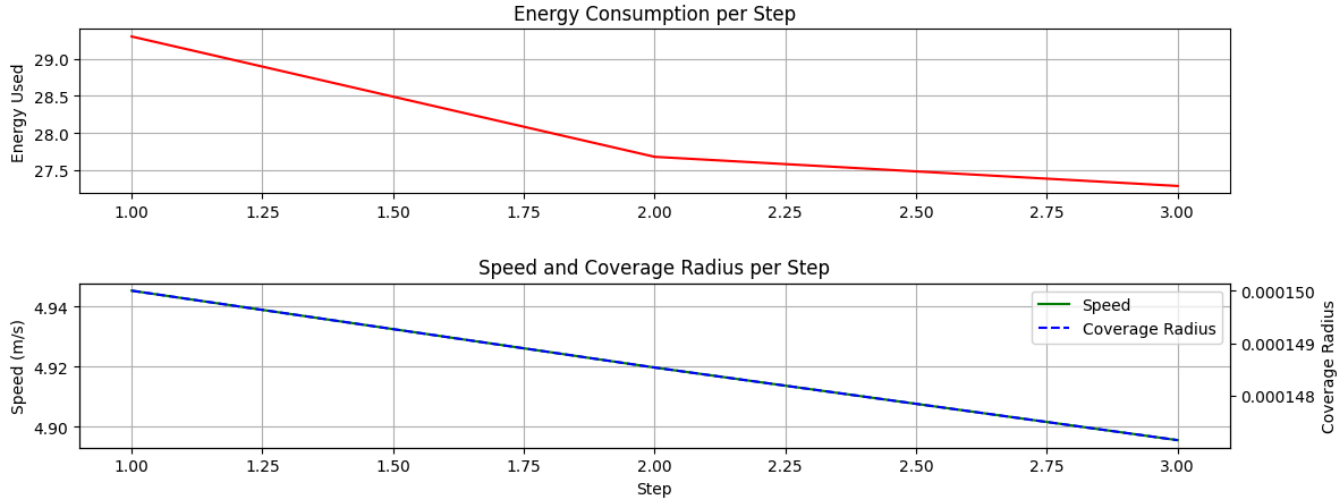




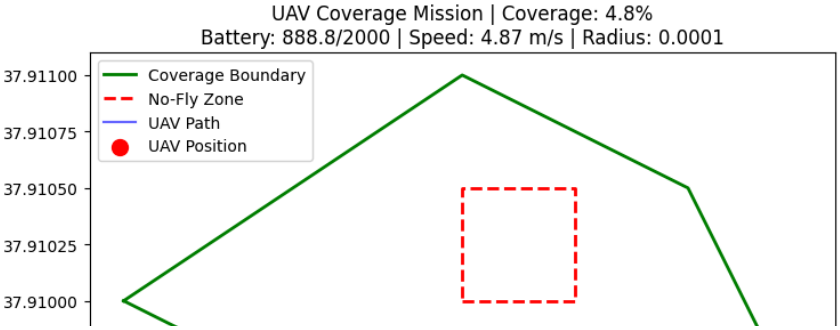


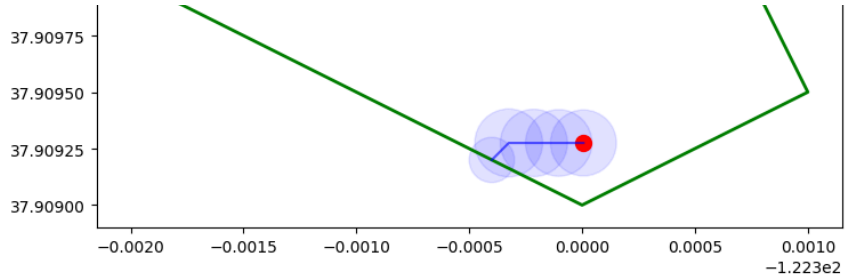

```
Step: Moved from (np.float64(-122.30021359868712), np.float64(37.909275)) to (np.float64(-122.30010323527497), np.float64(37.909275
Energy used: 27.28, remaining: 915.74
Coverage: 3.9%
Coverage radius: 0.0001
```

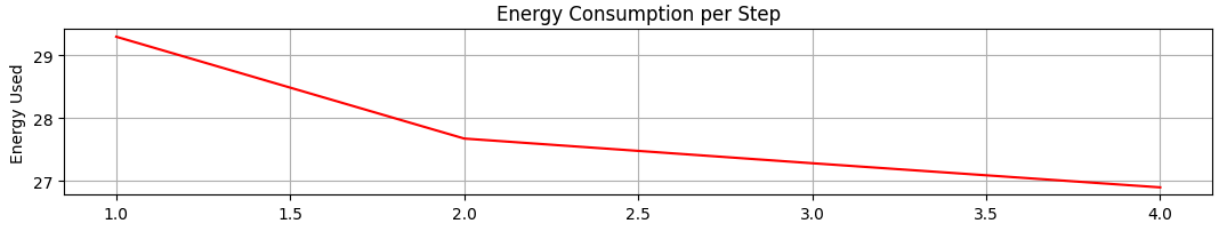


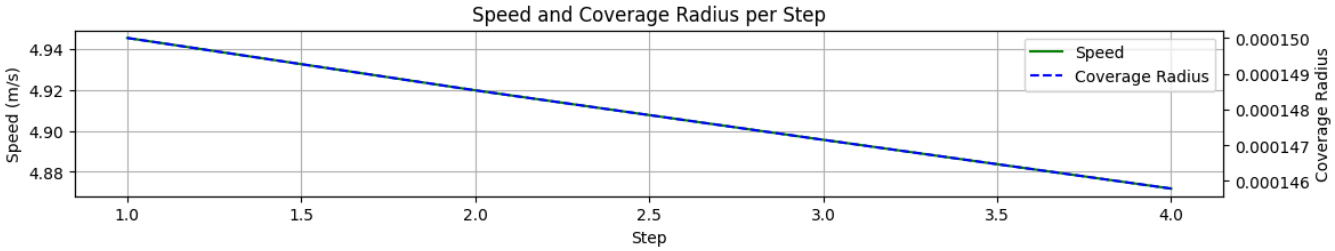



```
Step: Moved from (np.float64(-122.30010323527497), np.float64(37.909275)) to (np.float64(-122.2999938950454), np.float64(37.909275)
Energy used: 26.90, remaining: 888.84
Coverage: 4.8%
Coverage radius: 0.0001
```

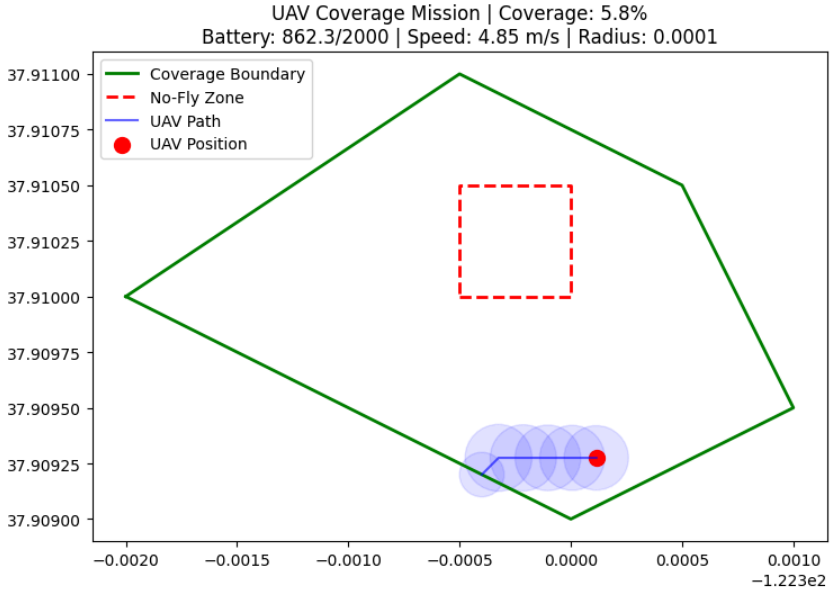


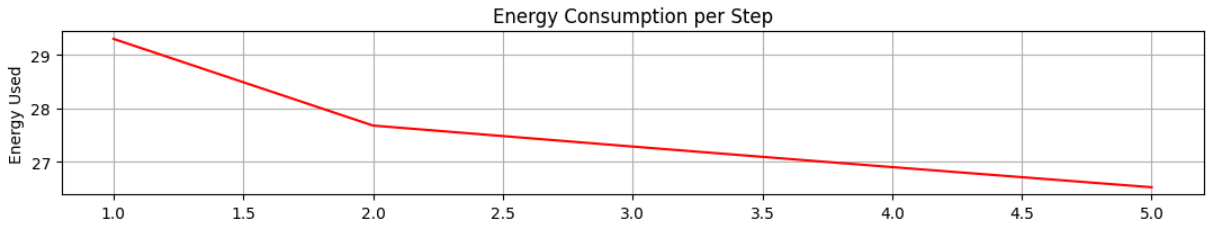


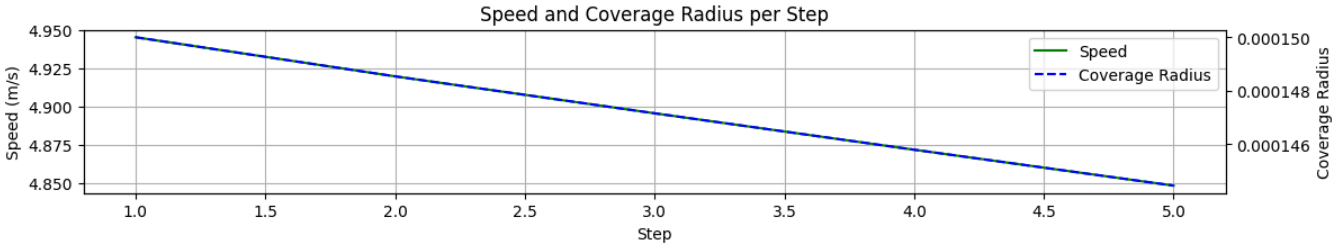




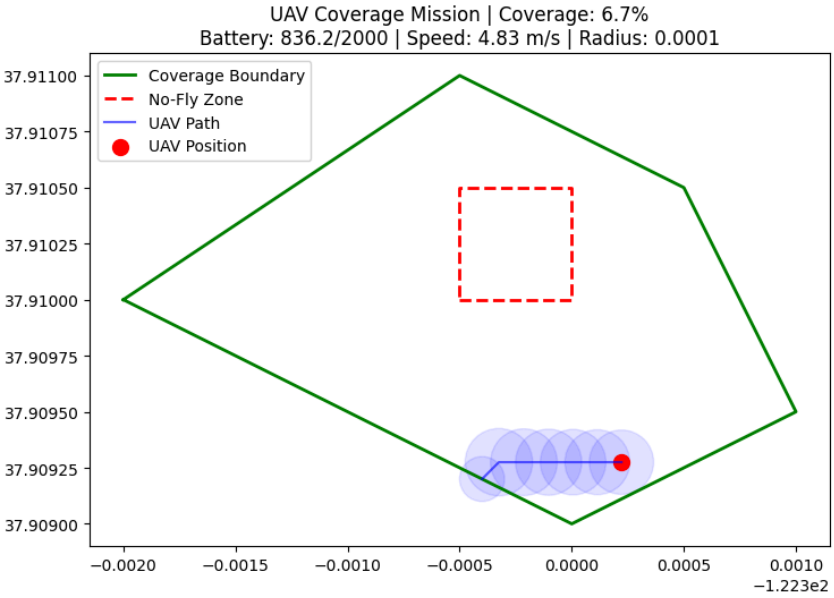

```
Step: Moved from (np.float64(-122.2999938950454), np.float64(37.909275)) to (np.float64(-122.29988556358195), np.float64(37.909275)
Energy used: 26.52, remaining: 862.32
Coverage: 5.8%
Coverage radius: 0.0001
```

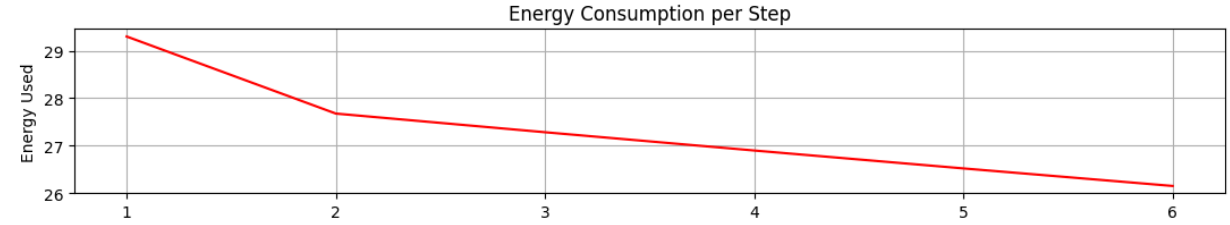


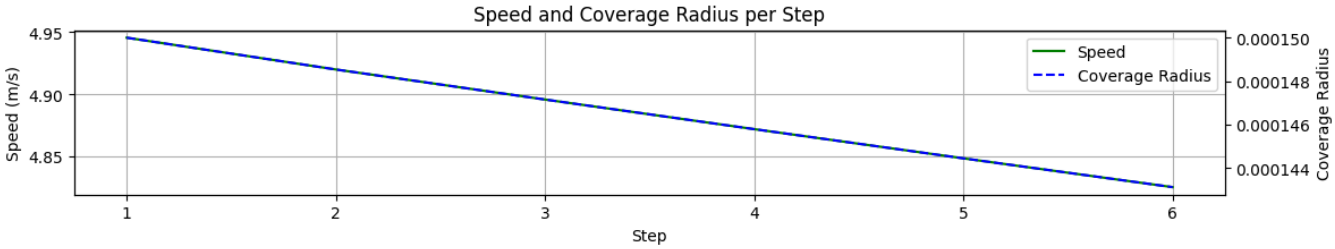




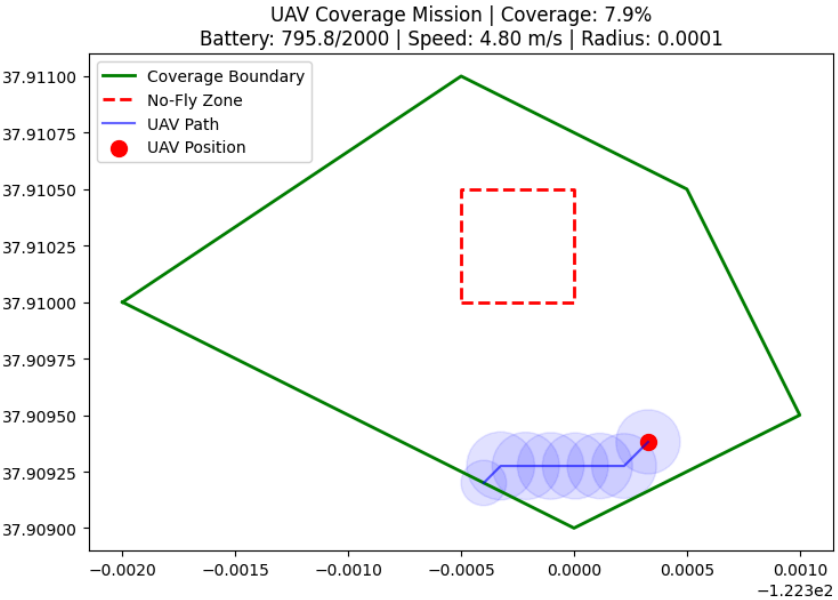

```
Step: Moved from (np.float64(-122.29988556358195), np.float64(37.909275)) to (np.float64(-122.29977822676162), np.float64(37.909275
Energy used: 26.15, remaining: 836.16
Coverage: 6.7%
Coverage radius: 0.0001
```

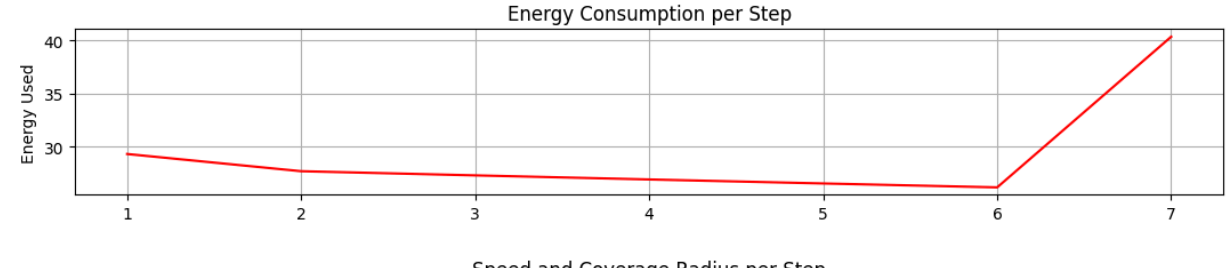


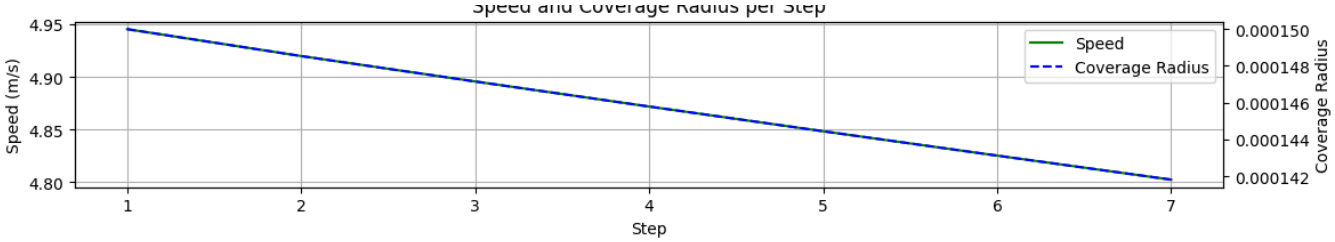




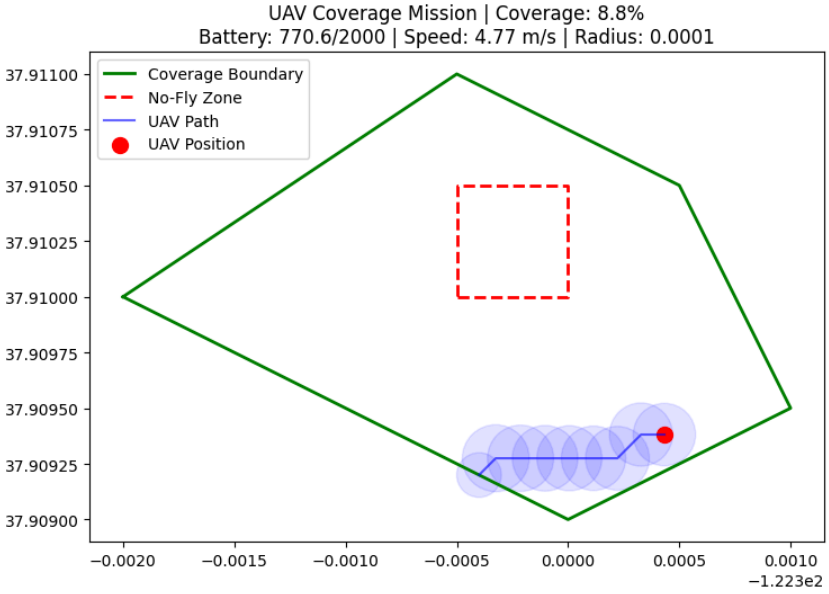

```
Step: Moved from (np.float64(-122.29977822676162), np.float64(37.909275)) to (np.float64(-122.29967187074695), np.float64(37.909381
Energy used: 40.35, remaining: 795.81
Coverage: 7.9%
Coverage radius: 0.0001
```

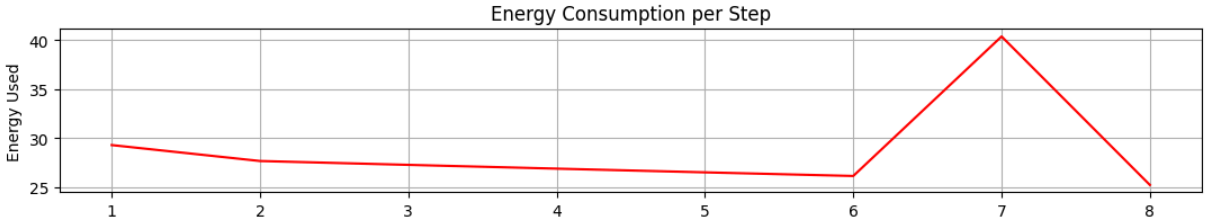


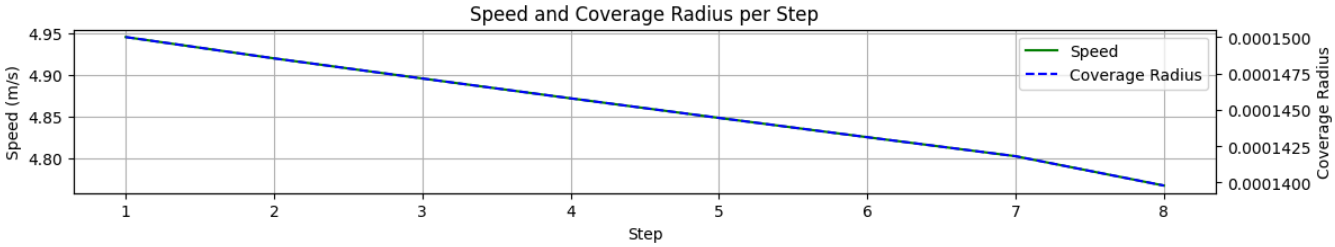




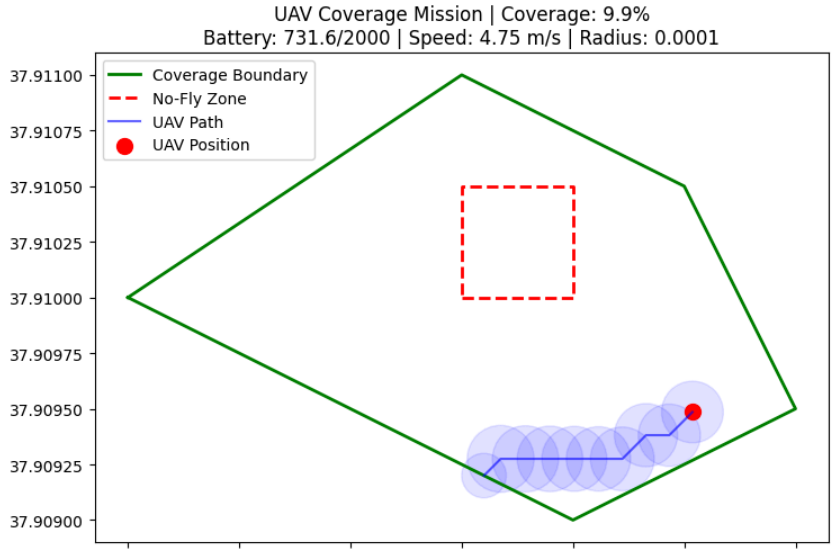

```
Step: Moved from (np.float64(-122.29967187074695), np.float64(37.90938135601467)) to (np.float64(-122.29956702771177), np.float64(37
Energy used: 25.24, remaining: 770.57
Coverage: 8.8%
Coverage radius: 0.0001
```

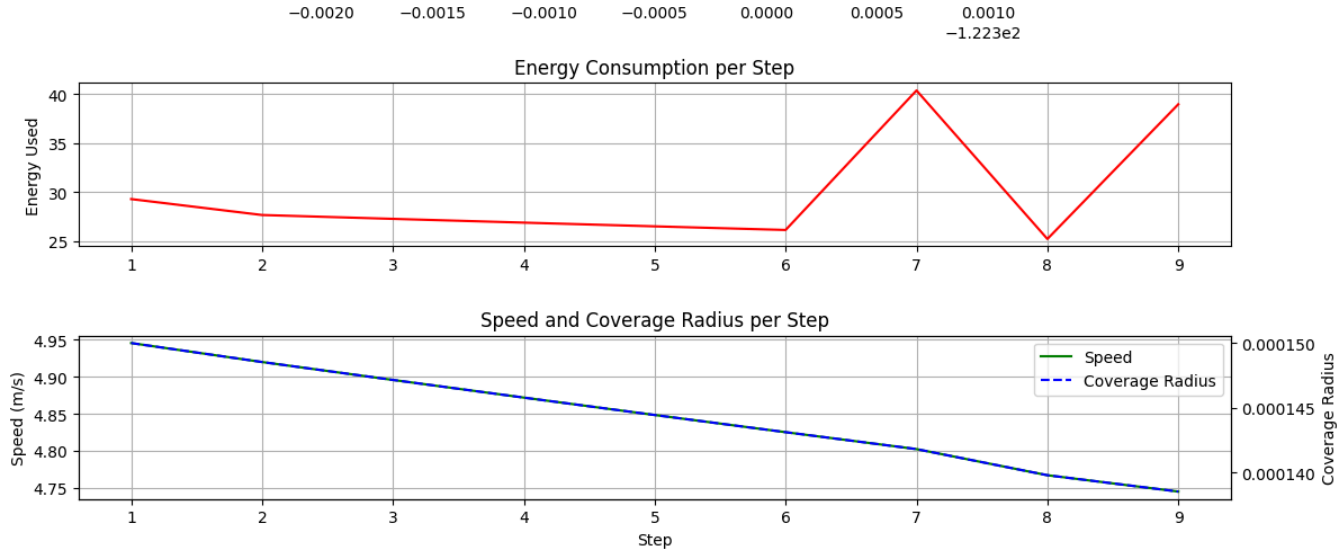




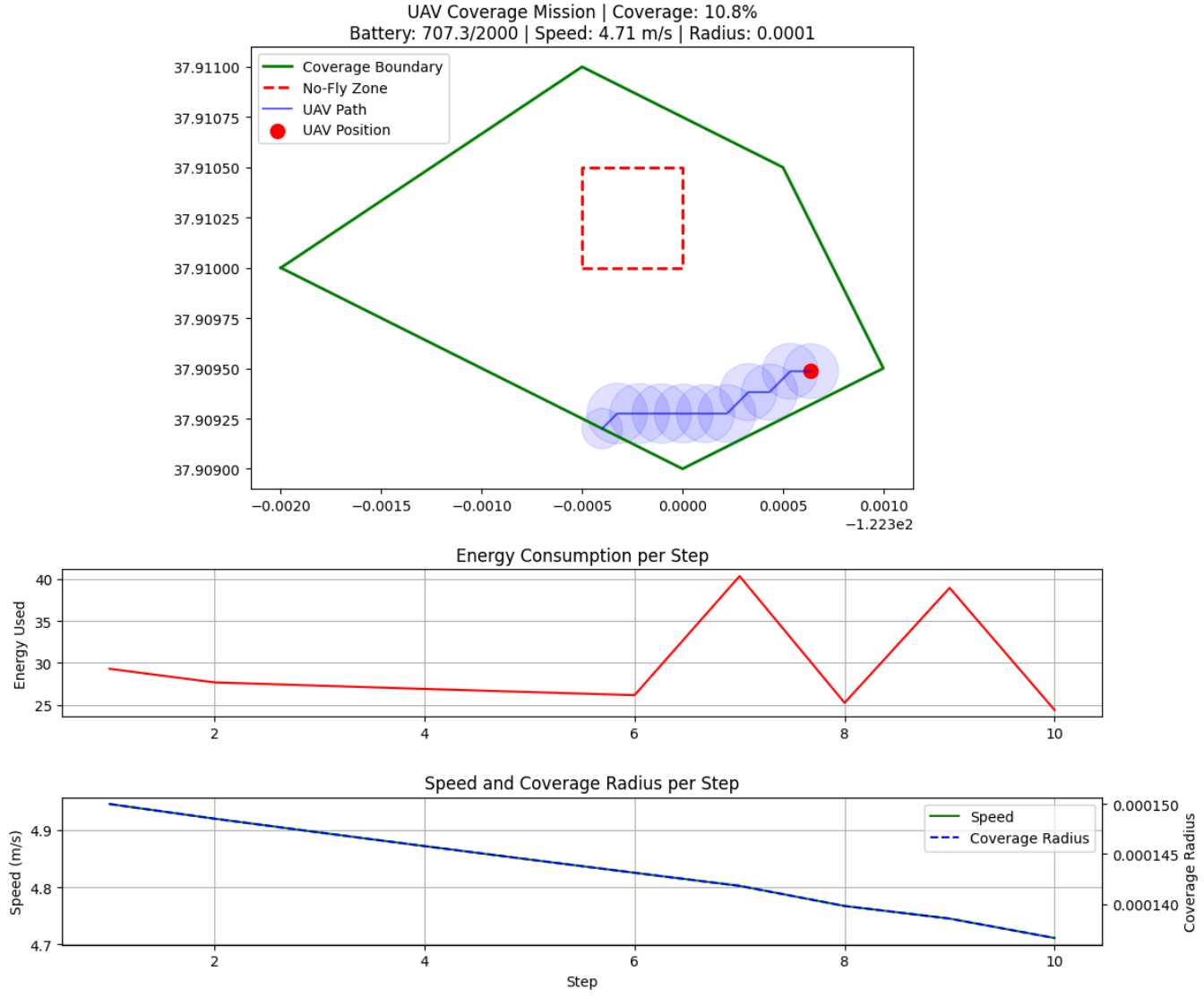



```
Step: Moved from (np.float64(-122.29956702771177), np.float64(37.90938135601467)) to (np.float64(-122.2994631311722), np.float64(37
Energy used: 38.94, remaining: 731.63
Coverage: 9.9%
Coverage radius: 0.0001
```

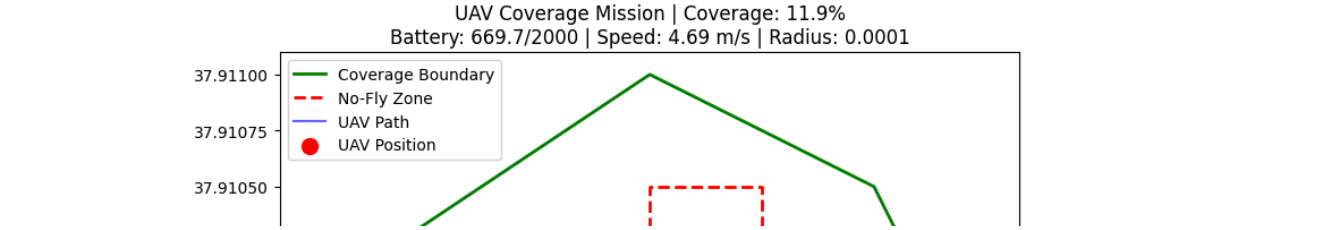


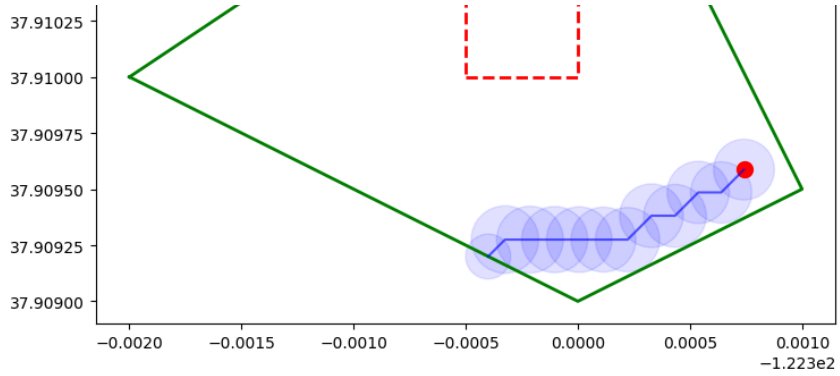


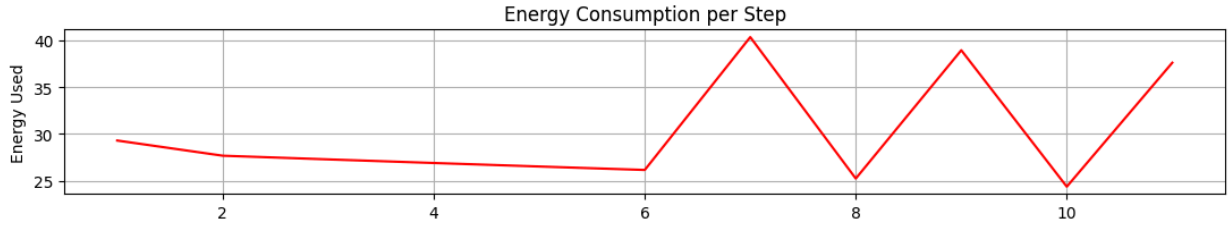

```
Step: Moved from (np.float64(-122.2994631311722), np.float64(37.90948525255425)) to (np.float64(-122.29936069500899), np.float64(37
Energy used: 24.37, remaining: 707.26
Coverage: 10.8%
Coverage radius: 0.0001
```

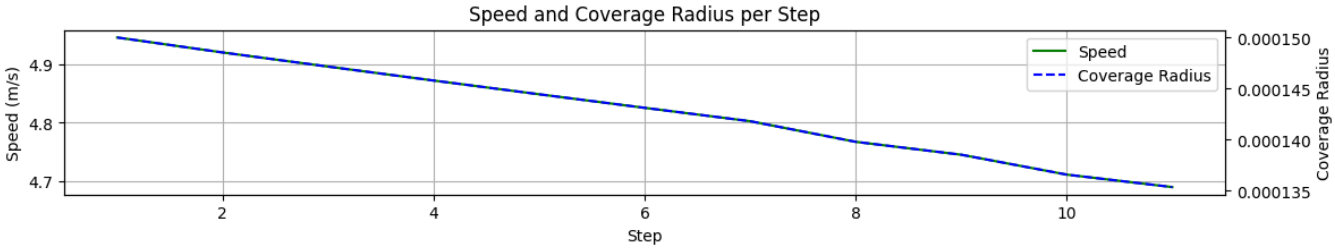



```
Step: Moved from (np.float64(-122.29936069500899), np.float64(37.90948525255425)) to (np.float64(-122.29925917274689), np.float64(37
Energy used: 37.61, remaining: 669.65
Coverage: 11.9%
Coverage radius: 0.0001
```

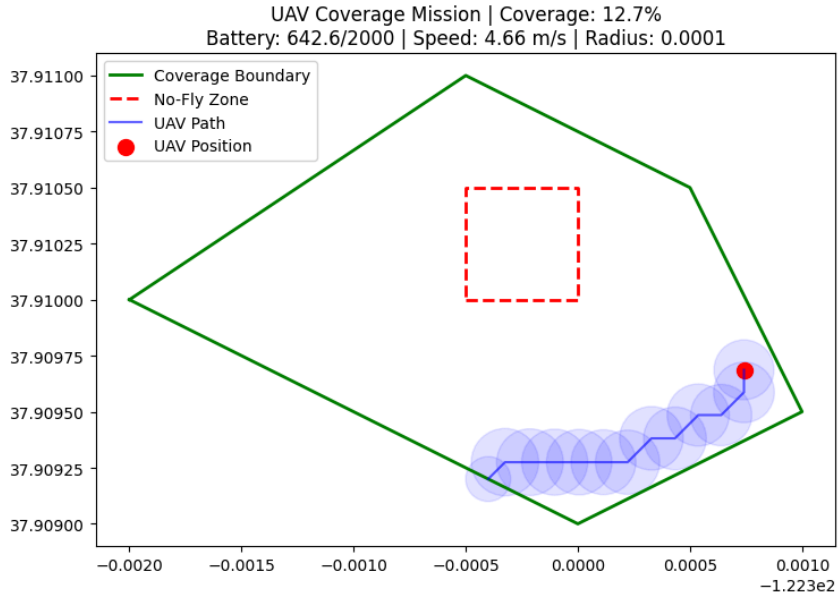


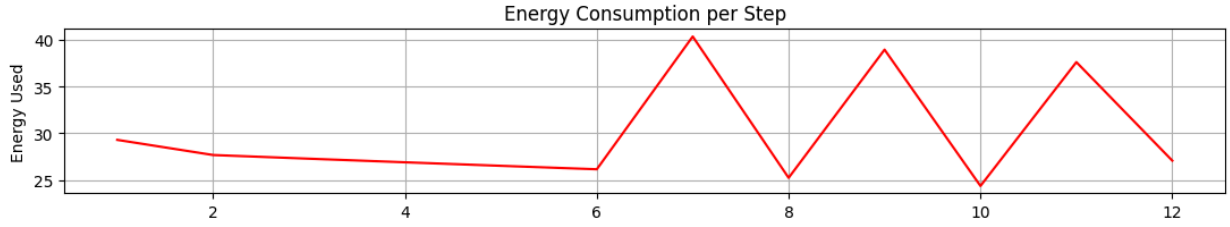


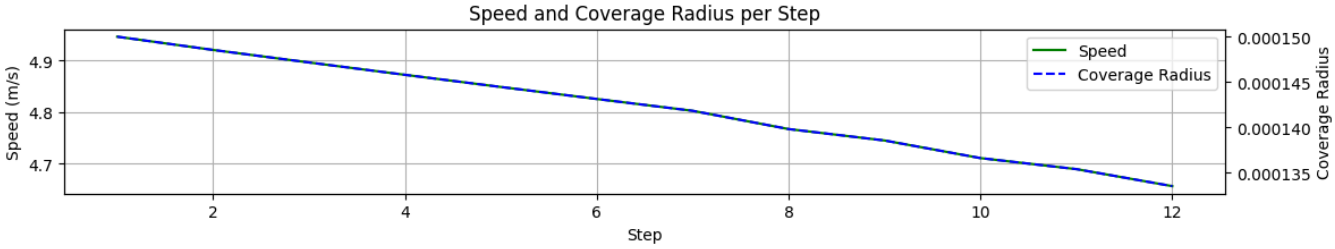




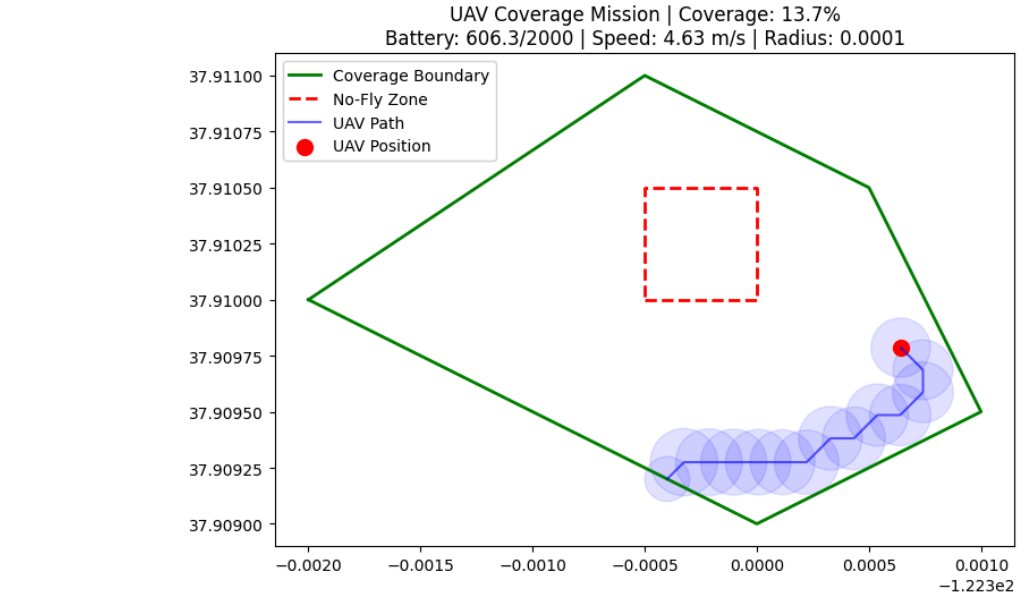

```
Step: Moved from (np.float64(-122.29925917274689), np.float64(37.90958677481635)) to (np.float64(-122.29925917274689), np.float64(37
Energy used: 27.08, remaining: 642.57
Coverage: 12.7%
Coverage radius: 0.0001
```

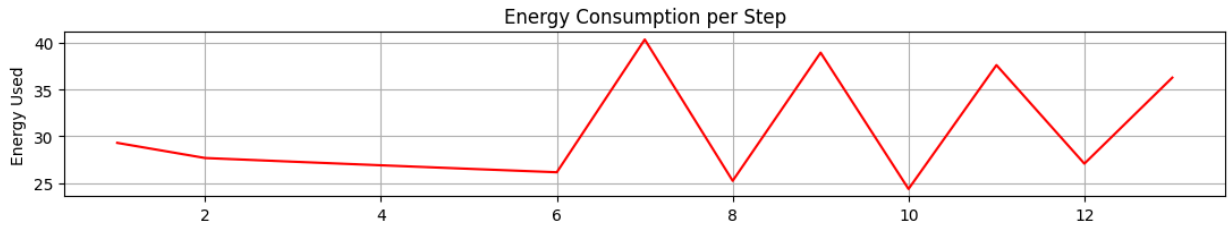


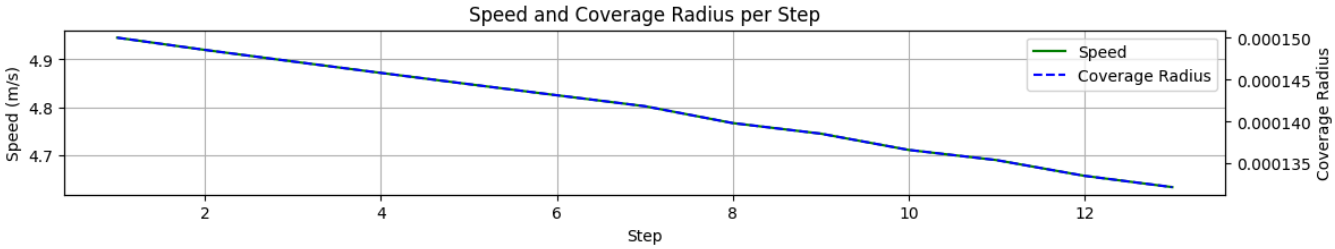




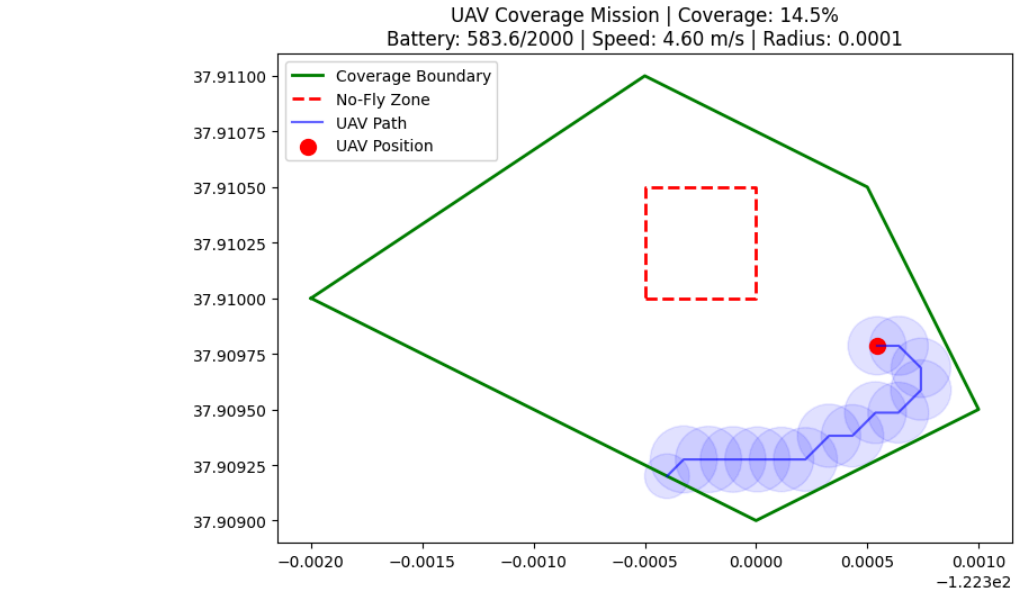
```
Step: Moved from (np.float64(-122.29925917274689), np.float64(37.90968688669185)) to (np.float64(-122.29935826927628), np.float64(37
Energy used: 36.27, remaining: 606.30
Coverage: 13.7%
Coverage radius: 0.0001
```

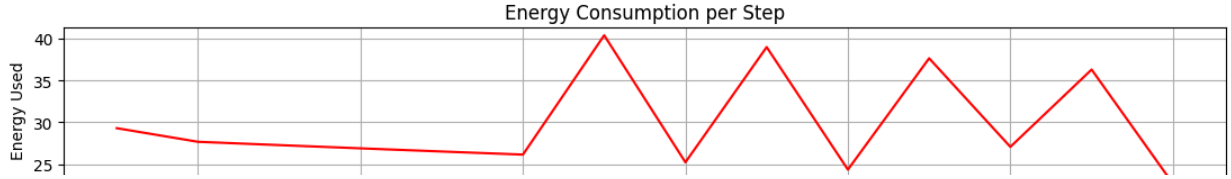


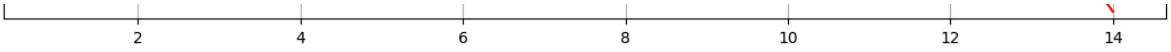


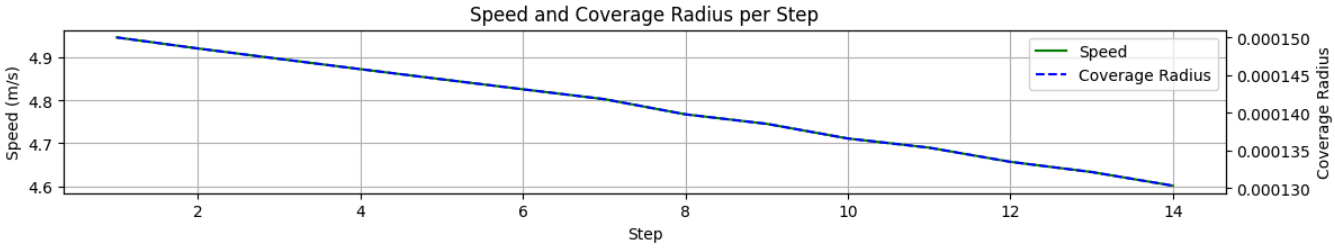



```
Step: Moved from (np.float64(-122.29935826927628), np.float64(37.90978598322125)) to (np.float64(-122.29945600568955), np.float64(37
Energy used: 22.71, remaining: 583.59
Coverage: 14.5%
Coverage radius: 0.0001
```

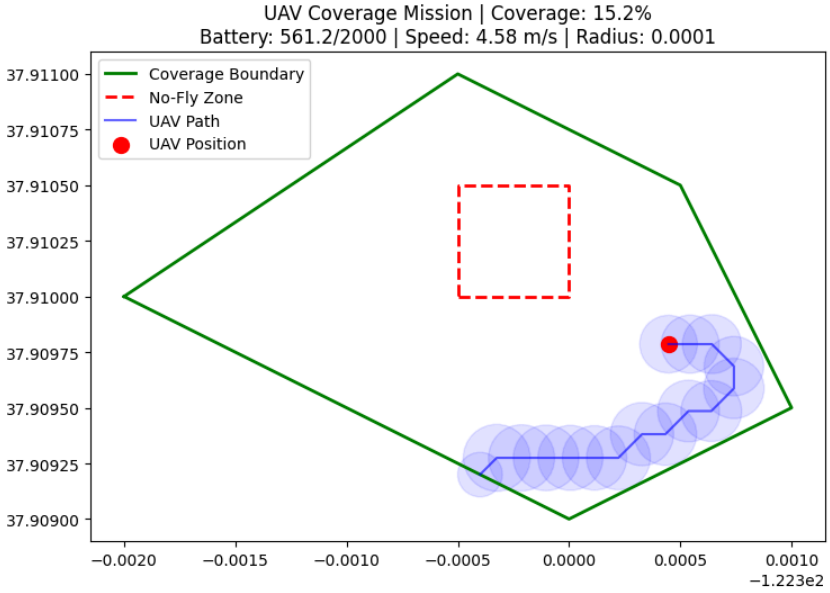


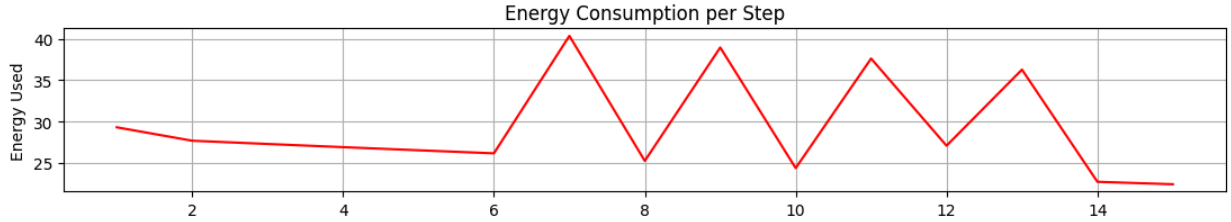


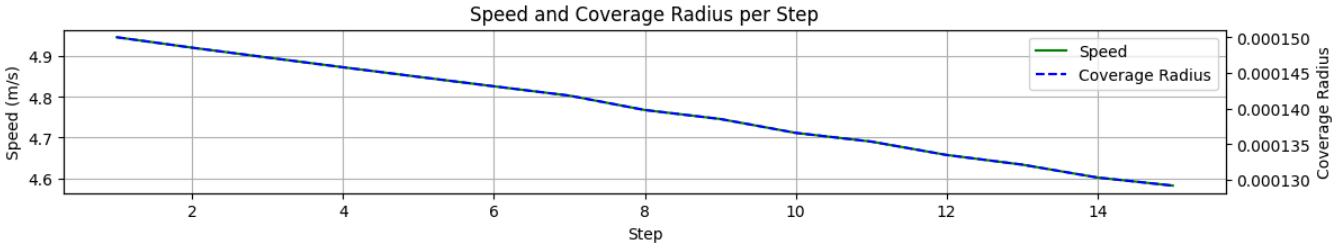




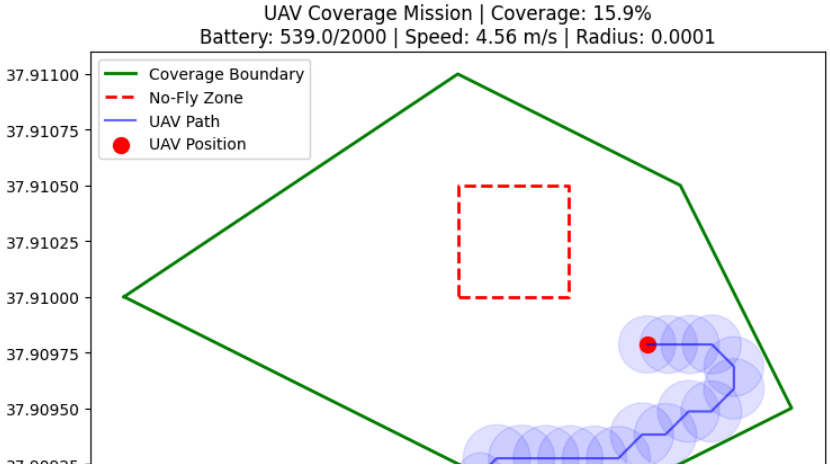

```
Step: Moved from (np.float64(-122.29945600568955), np.float64(37.90978598322125)) to (np.float64(-122.29955289037495), np.float64(37
Energy used: 22.42, remaining: 561.17
Coverage: 15.2%
Coverage radius: 0.0001
```

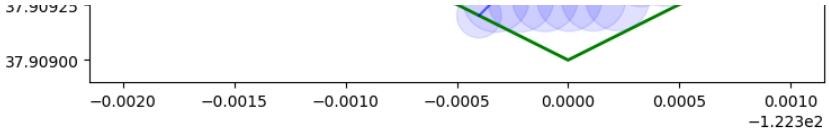


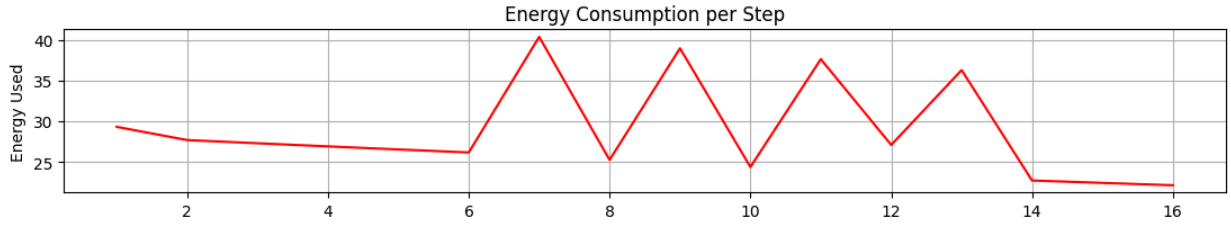


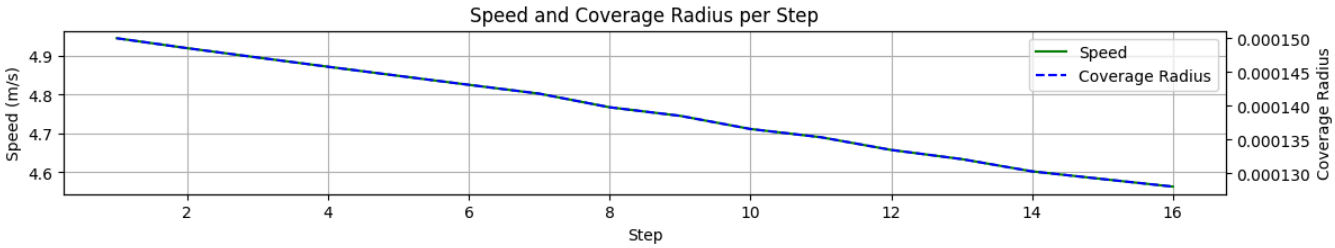



```
Step: Moved from (np.float64(-122.29955289037495), np.float64(37.90978598322125)) to (np.float64(-122.29964893439166), np.float64(37
Energy used: 22.13, remaining: 539.05
Coverage: 15.9%
Coverage radius: 0.0001
```

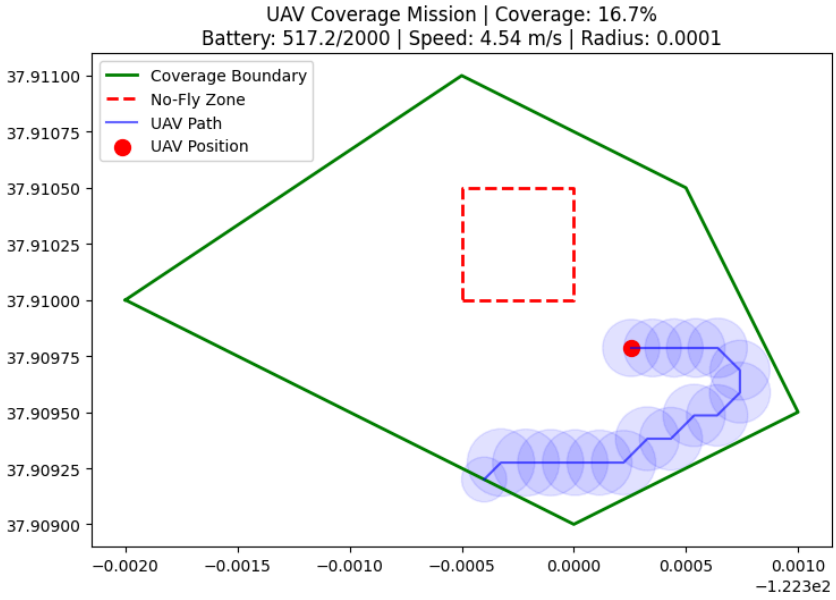


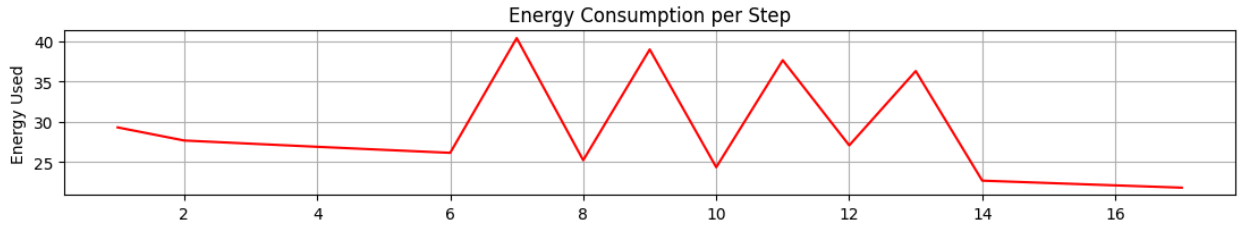


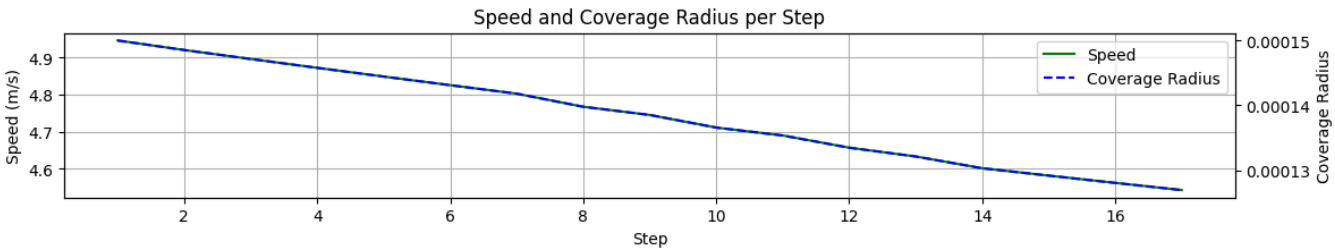




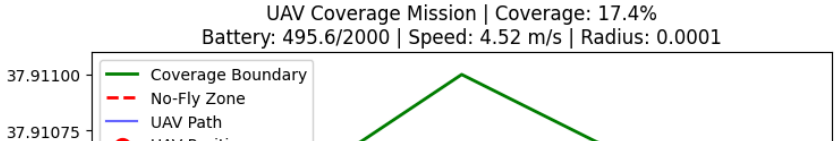

```
Step: Moved from (np.float64(-122.29964893439166), np.float64(37.90978598322125)) to (np.float64(-122.29974414859252), np.float64(37
Energy used: 21.84, remaining: 517.20
Coverage: 16.7%
Coverage radius: 0.0001
```

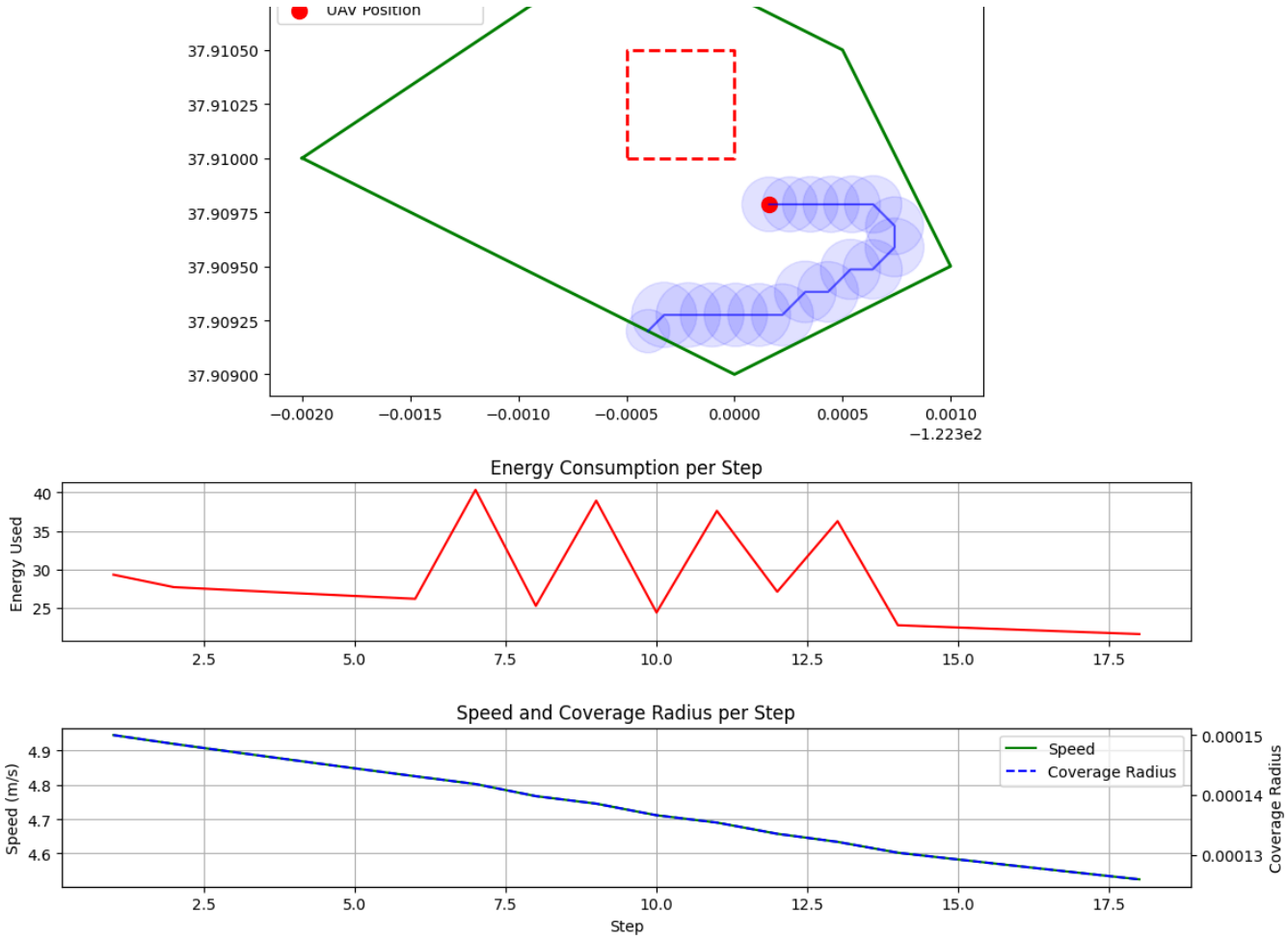




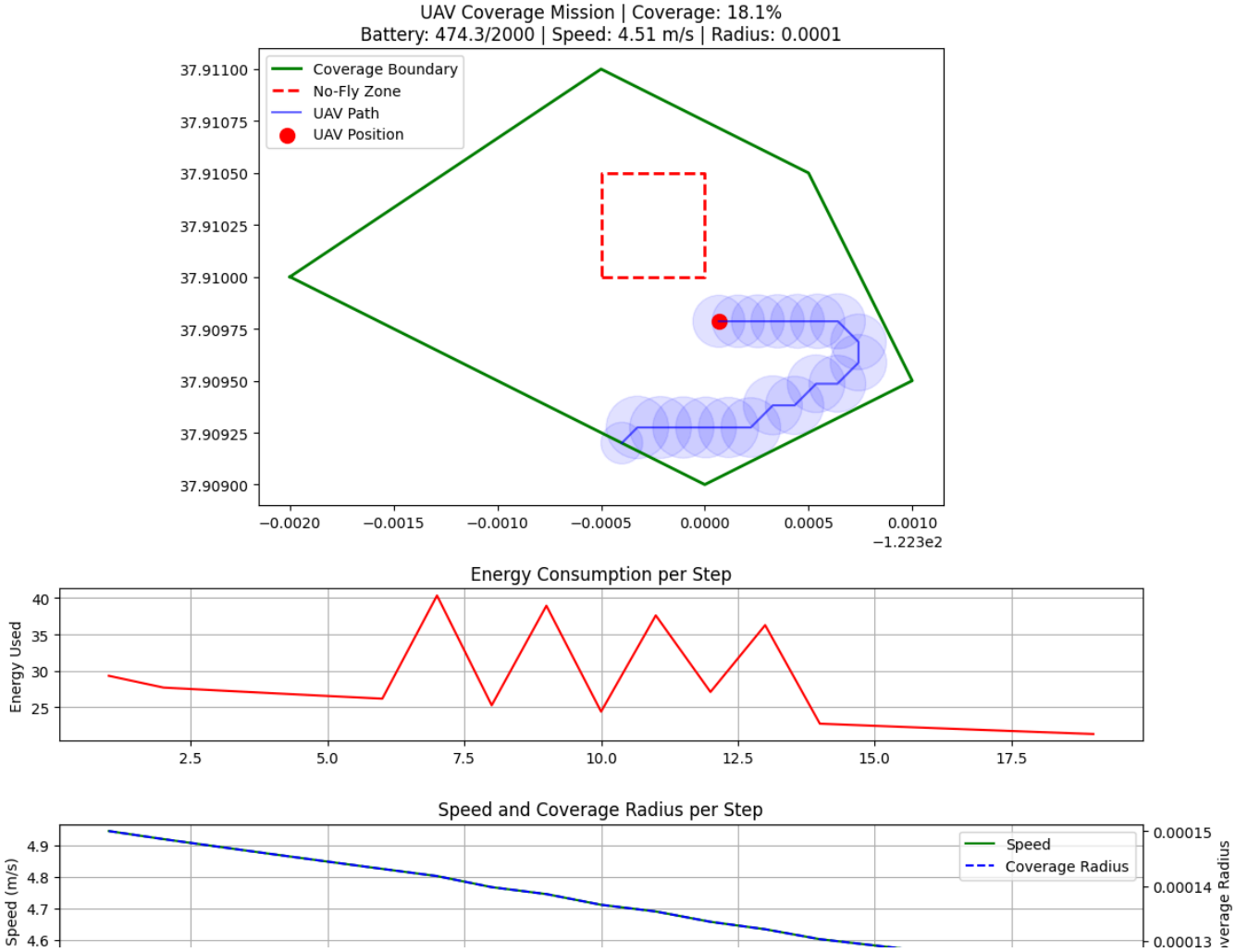


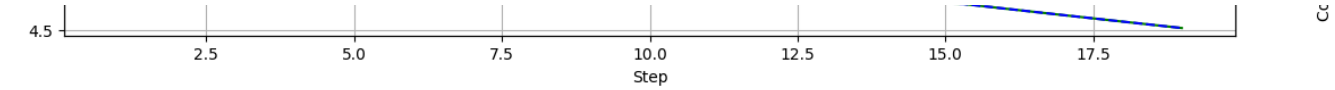

```
Step: Moved from (np.float64(-122.29974414859252), np.float64(37.90978598322125)) to (np.float64(-122.29983854362924), np.float64(37
Energy used: 21.57, remaining: 495.64
Coverage: 17.4%
Coverage radius: 0.0001
```



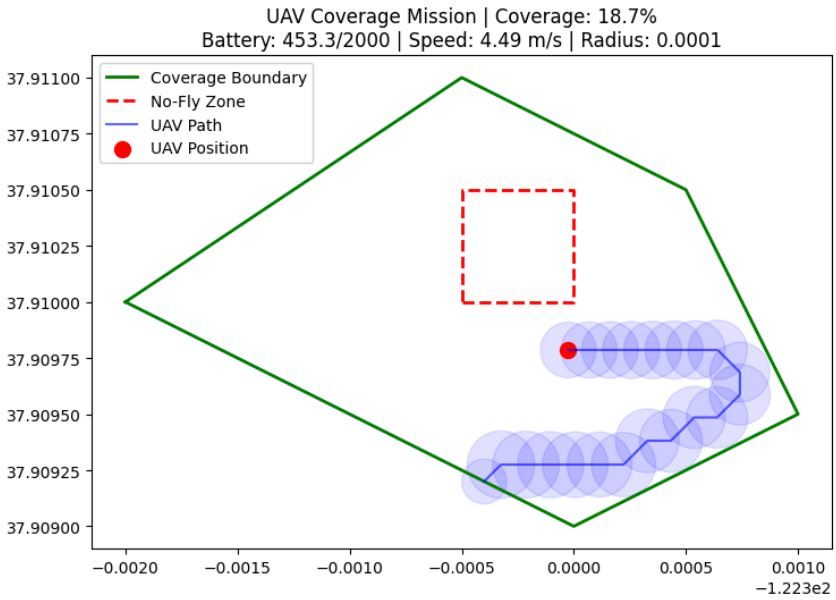


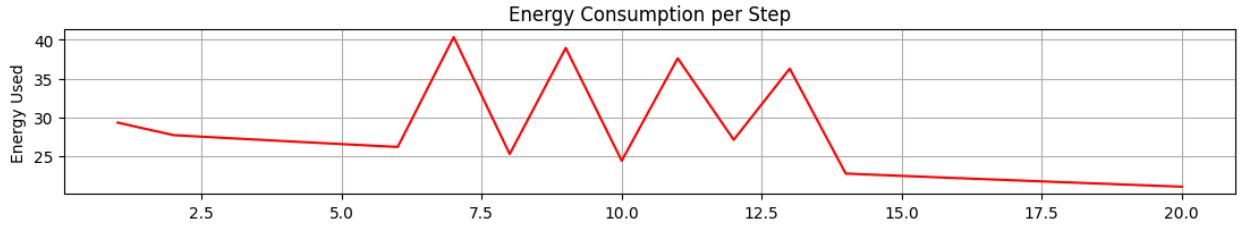

```
Step: Moved from (np.float64(-122.29983854362924), np.float64(37.90978598322125)) to (np.float64(-122.2999321299572), np.float64(37
Energy used: 21.29, remaining: 474.34
Coverage: 18.1%
Coverage radius: 0.0001
```

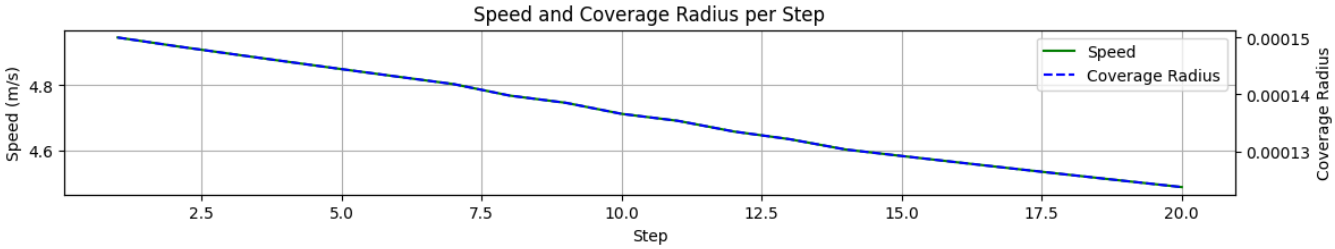




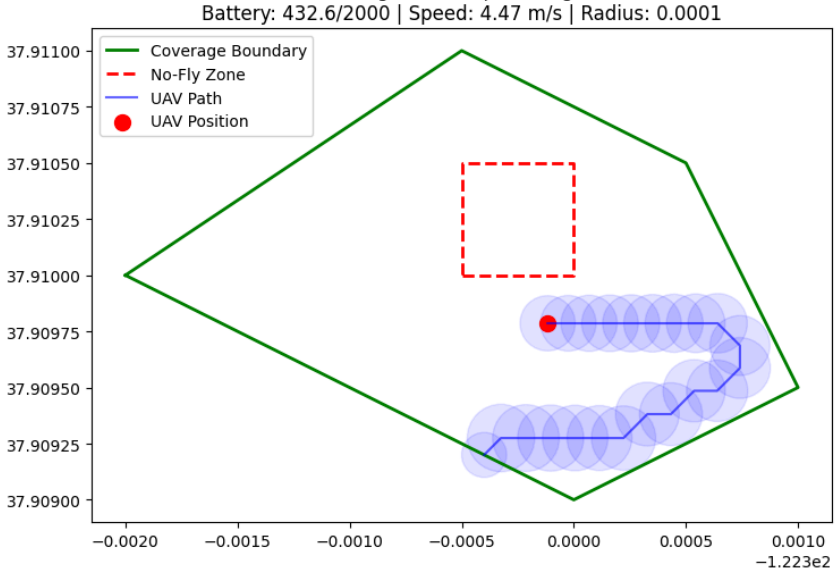

```
Step: Moved from (np.float64(-122.2999321299572), np.float64(37.90978598322125)) to (np.float64(-122.30002491784036), np.float64(37
Energy used: 21.02, remaining: 453.32
Coverage: 18.7%
Coverage radius: 0.0001
```

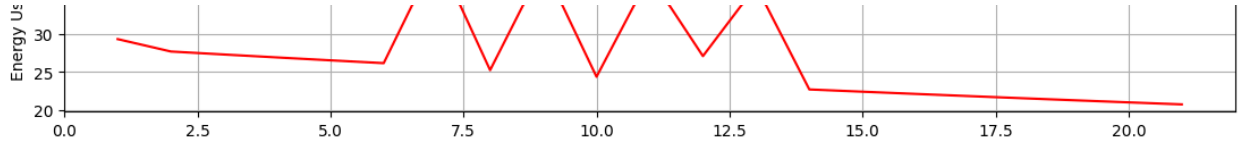


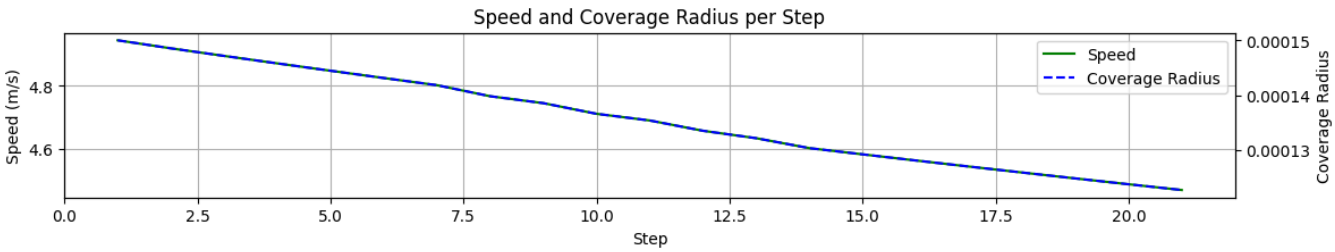




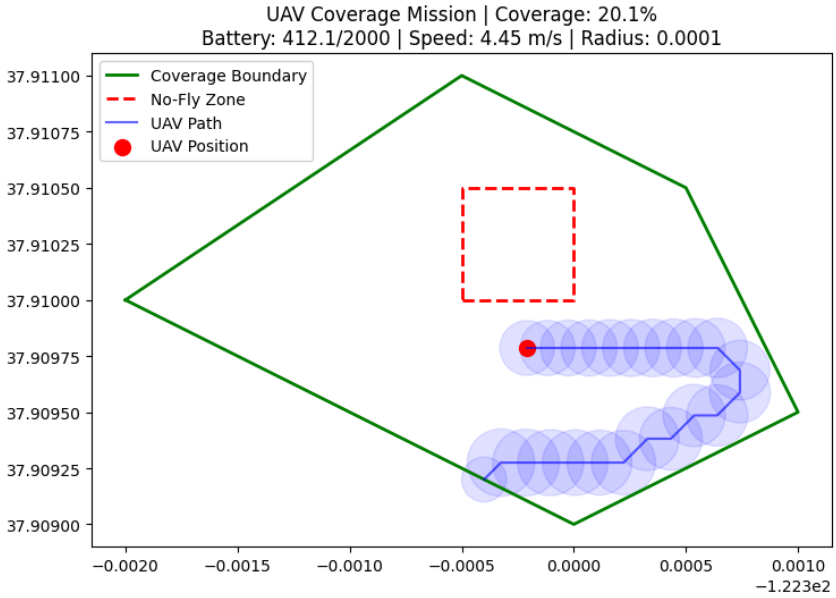

```
Step: Moved from (np.float64(-122.30002491784036), np.float64(37.90978598322125)) to (np.float64(-122.30011691735585), np.float64(37
Energy used: 20.76, remaining: 432.56
Coverage: 19.4%
Coverage radius: 0.0001
```

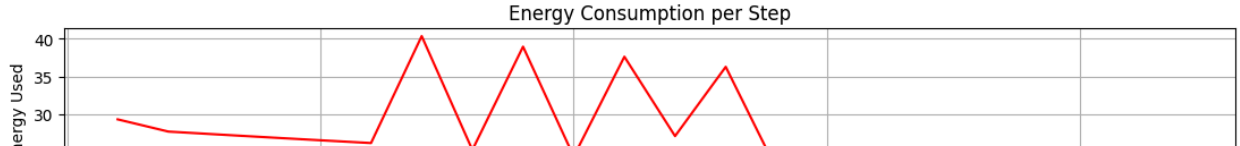


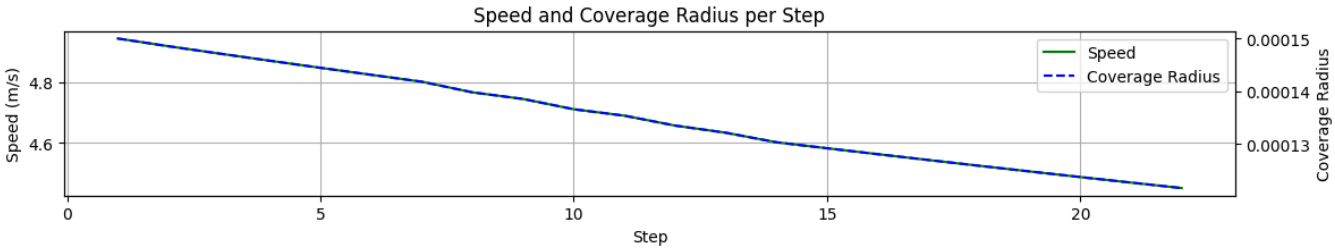




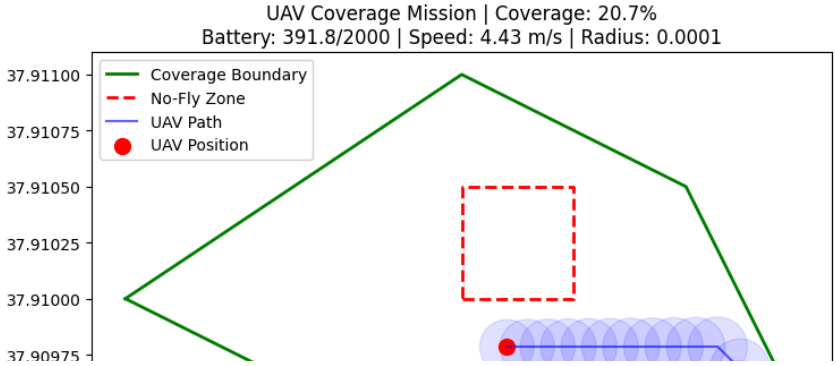

```
Step: Moved from (np.float64(-122.30011691735585), np.float64(37.90978598322125)) to (np.float64(-122.30020813839846), np.float64(37
Energy used: 20.50, remaining: 412.06
Coverage: 20.1%
Coverage radius: 0.0001
```

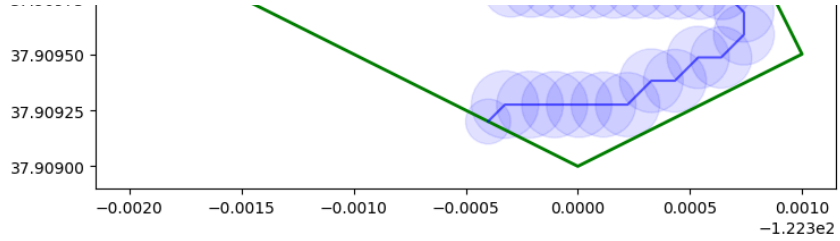


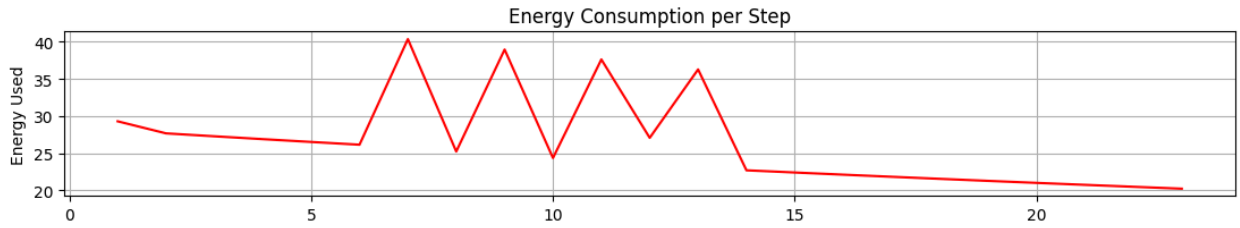


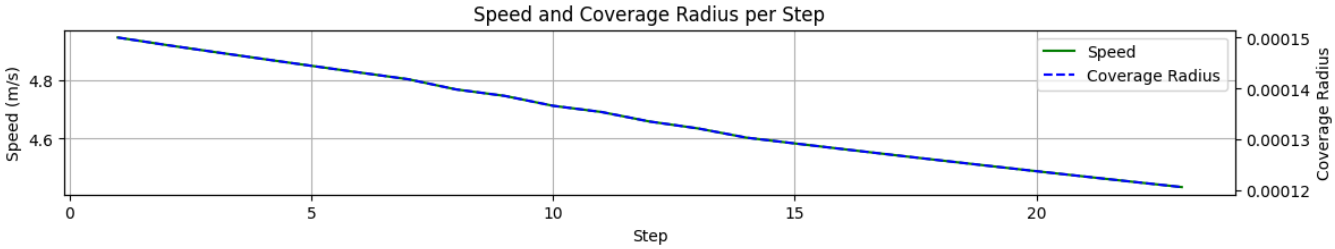



```
Step: Moved from (np.float64(-122.30020813839846), np.float64(37.90978598322125)) to (np.float64(-122.30029859068507), np.float64(37
Energy used: 20.25, remaining: 391.82
Coverage: 20.7%
Coverage radius: 0.0001
```

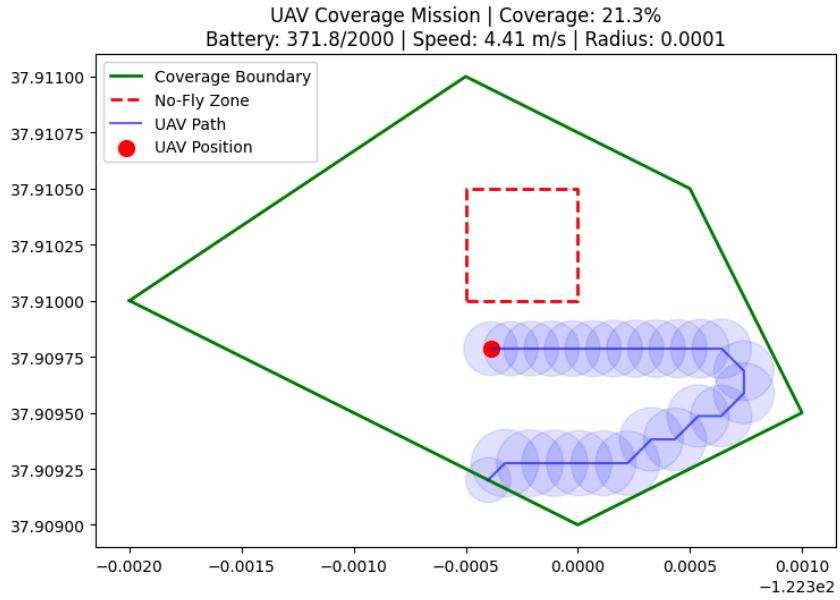


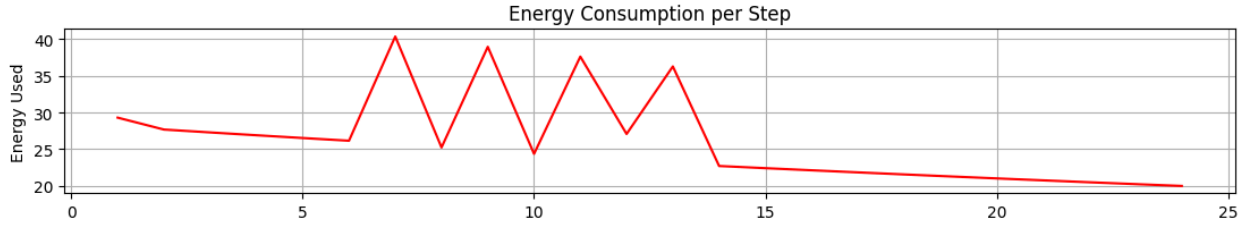


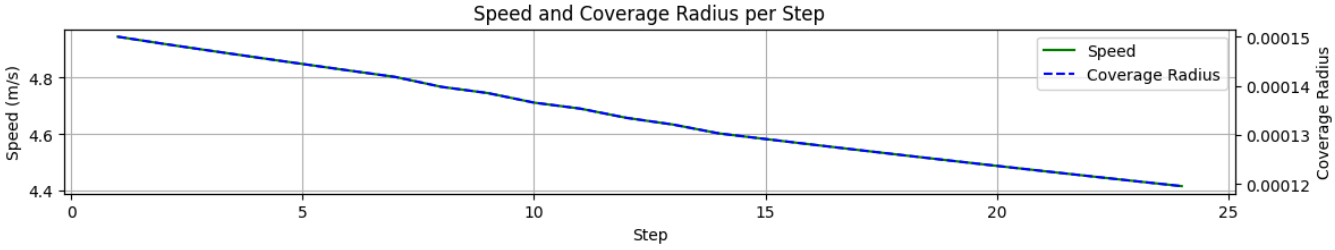





```
Step: Moved from (np.float64(-122.30029859068507), np.float64(37.90978598322125)) to (np.float64(-122.30038828375883), np.float64(37
Energy used: 20.00, remaining: 371.82
Coverage: 21.3%
Coverage radius: 0.0001
```

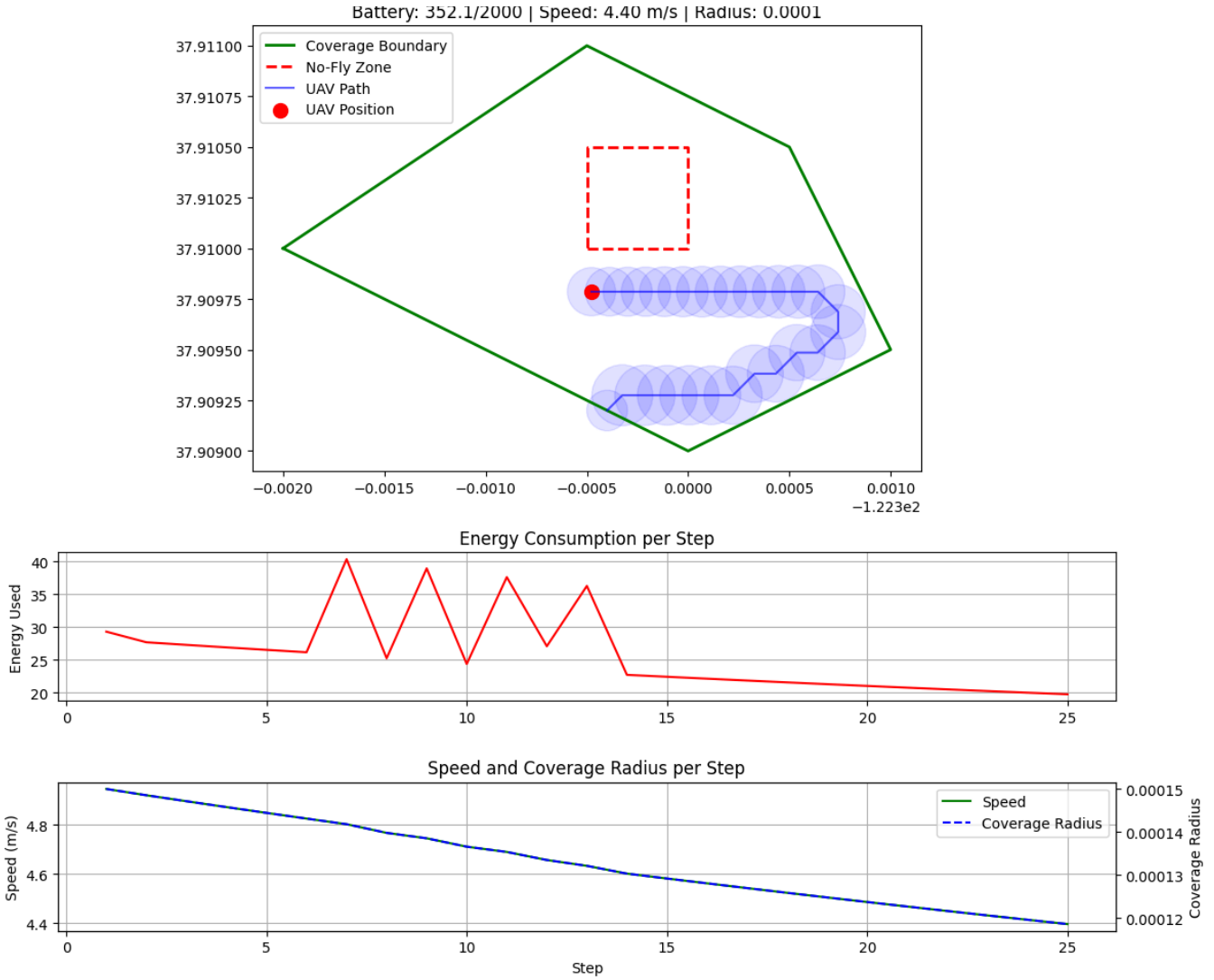




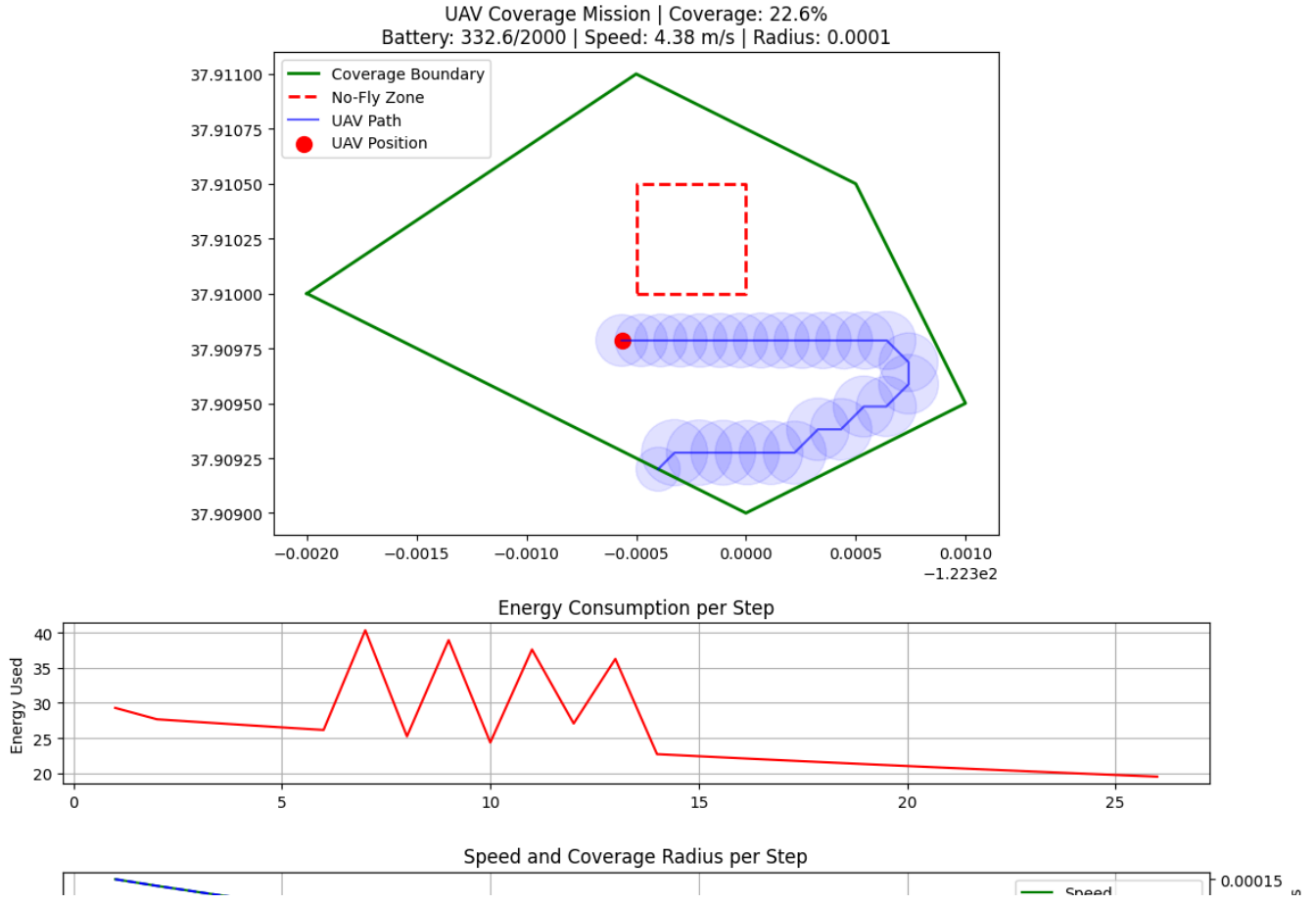


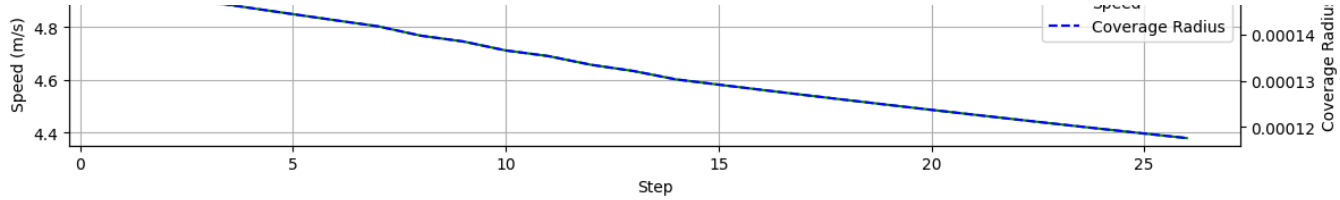

```
Step: Moved from (np.float64(-122.30038828375883), np.float64(37.90978598322125)) to (np.float64(-122.30047722699331), np.float64(37
Energy used: 19.75, remaining: 352.07
Coverage: 22.0%
Coverage radius: 0.0001
```

UAV Coverage Mission | Coverage: 22.0%

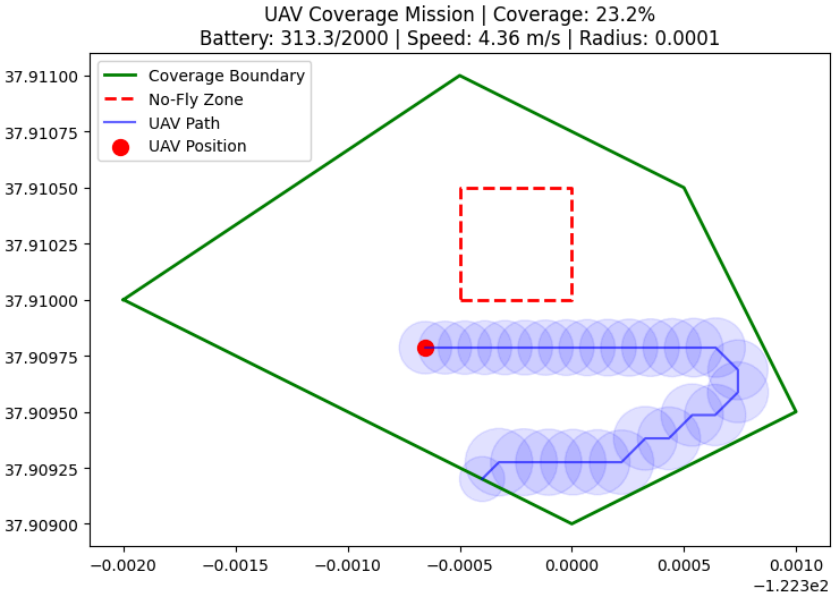


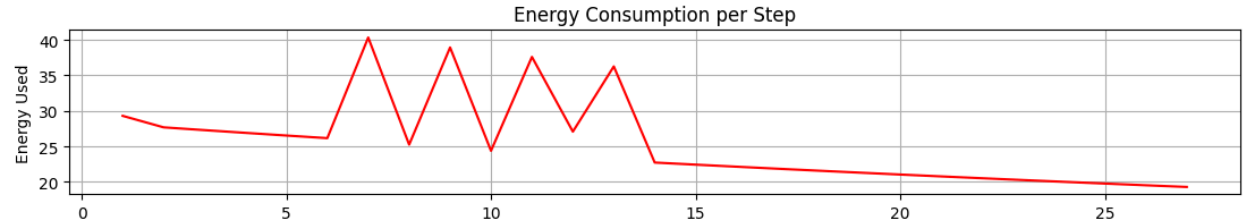

```
Step: Moved from (np.float64(-122.30047722699331), np.float64(37.90978598322125)) to (np.float64(-122.30056542959646), np.float64(37
Energy used: 19.51, remaining: 332.56
Coverage: 22.6%
Coverage radius: 0.0001
```

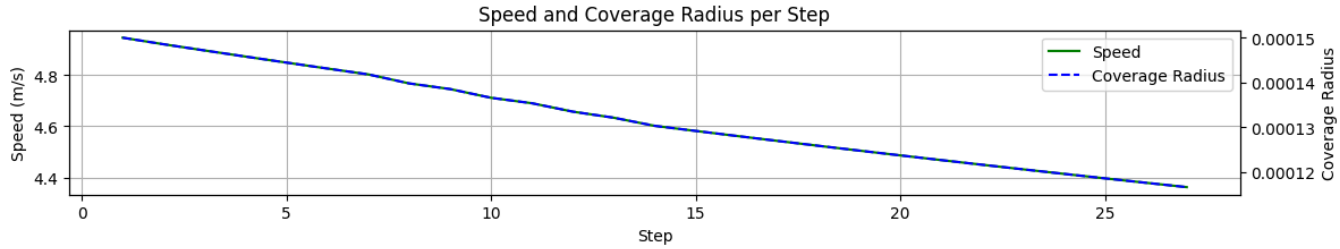




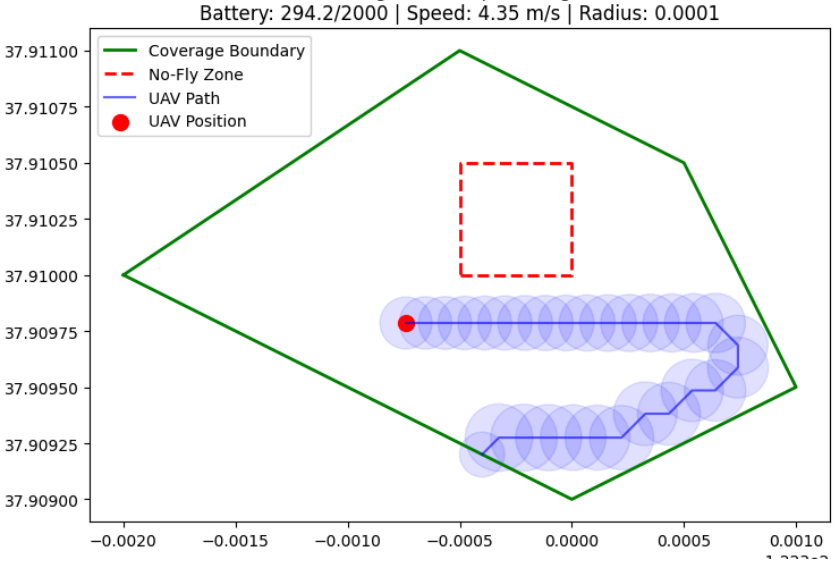

```
Step: Moved from (np.float64(-122.30056542959646), np.float64(37.90978598322125)) to (np.float64(-122.30065290061454), np.float64(37
Energy used: 19.27, remaining: 313.29
Coverage: 23.2%
Coverage radius: 0.0001
```



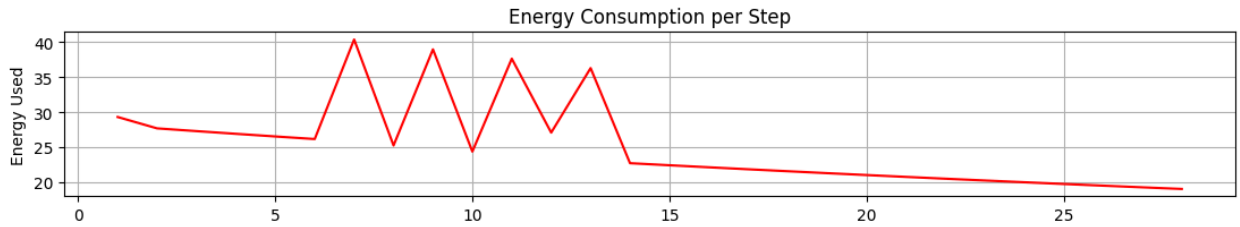


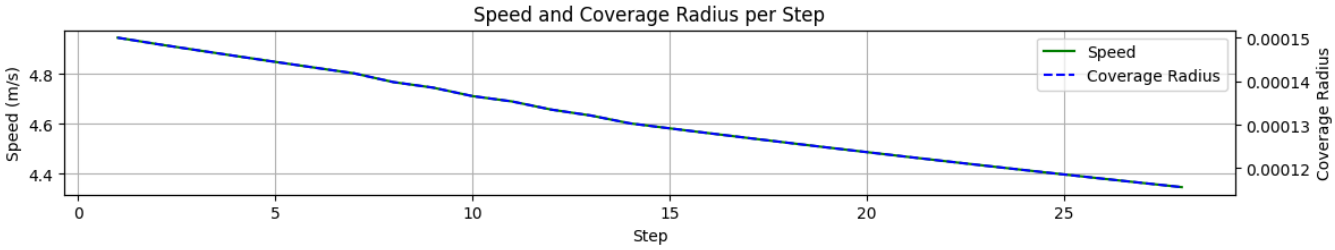



```
Step: Moved from (np.float64(-122.30065290061454), np.float64(37.90978598322125)) to (np.float64(-122.30073964893582), np.float64(37
Energy used: 19.04, remaining: 294.25
Coverage: 23.8%
Coverage radius: 0.0001
```

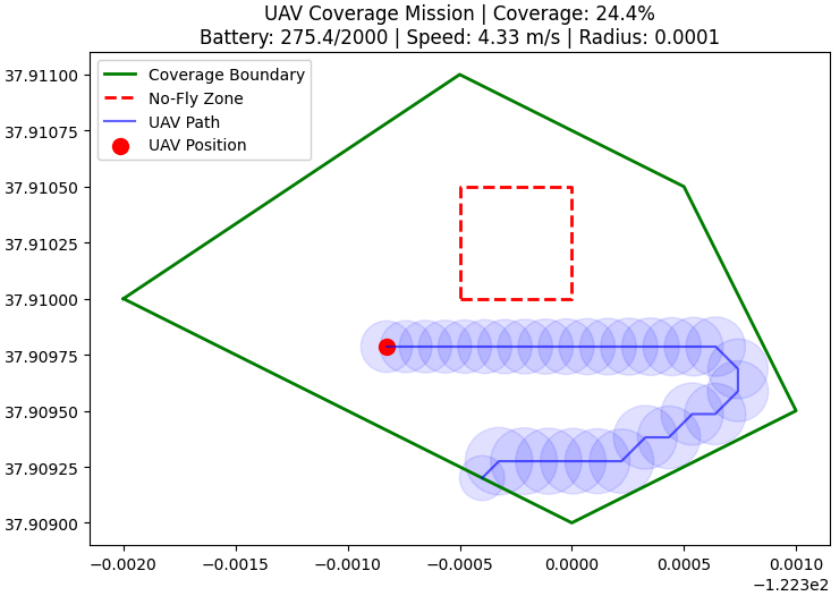


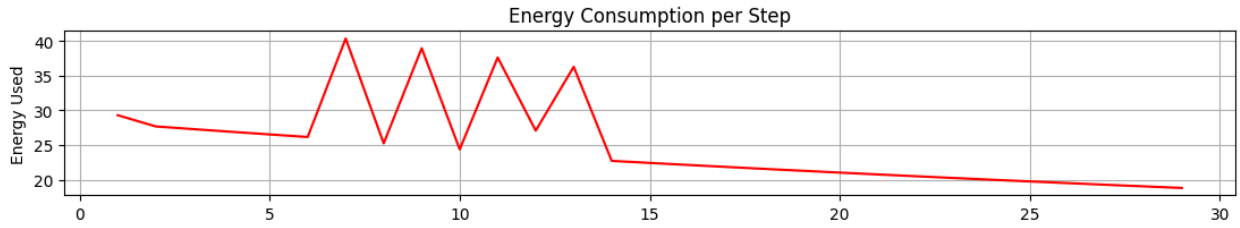
−1.223e2

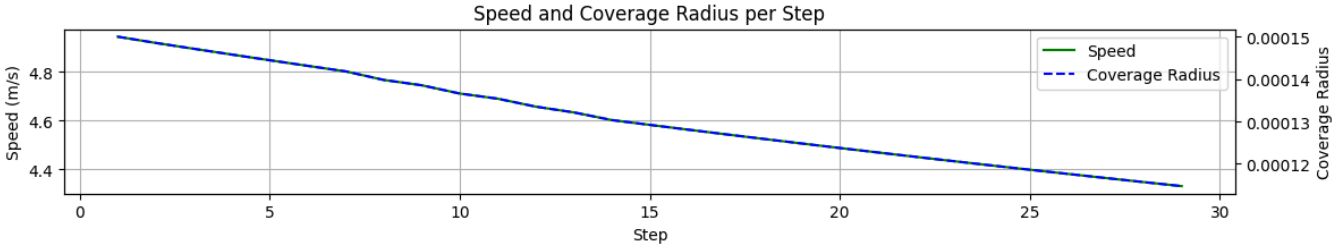




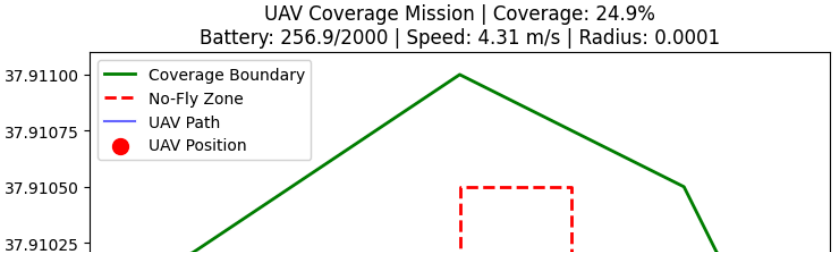

```
Step: Moved from (np.float64(-122.30073964893582), np.float64(37.90978598322125)) to (np.float64(-122.30082568329429), np.float64(37
Energy used: 18.81, remaining: 275.44
Coverage: 24.4%
Coverage radius: 0.0001
```

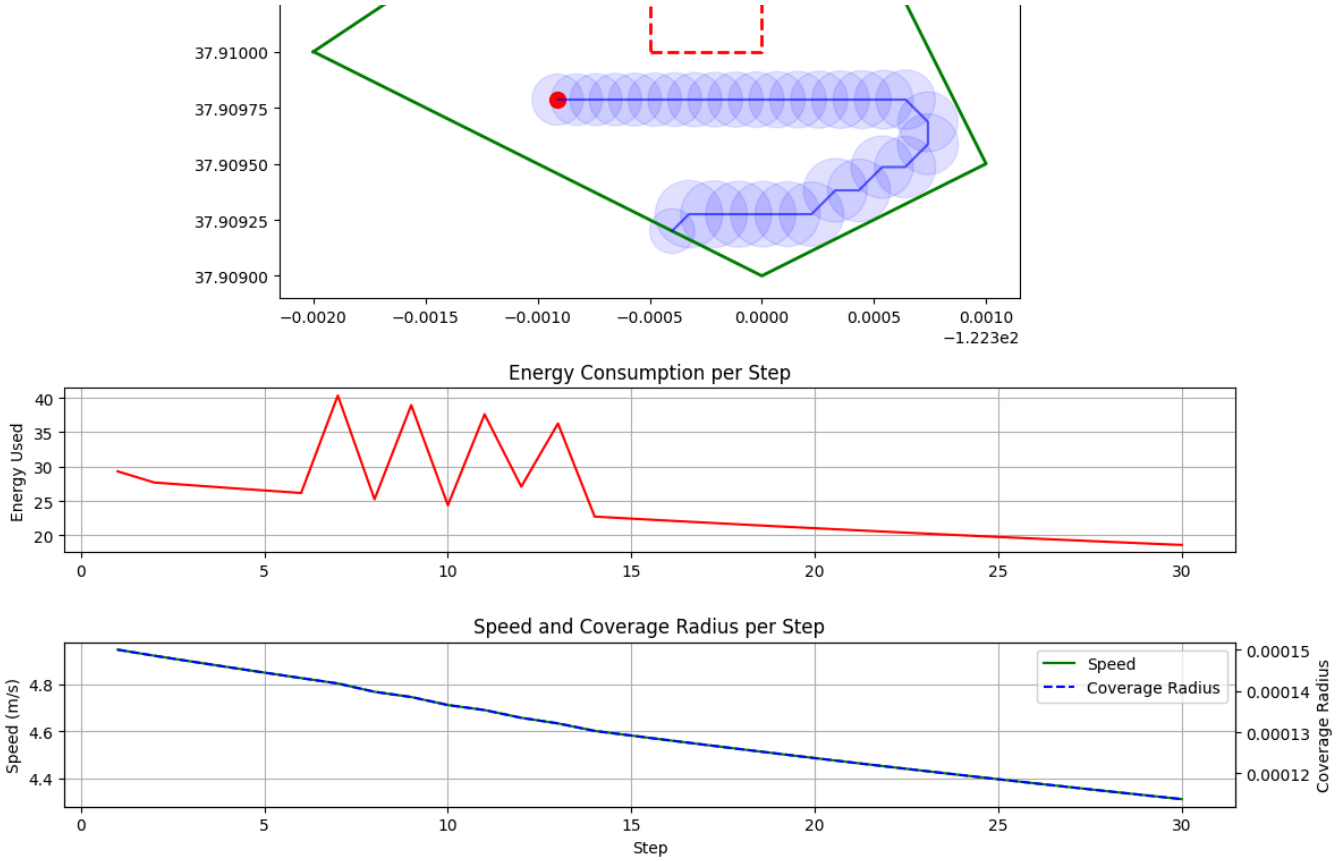




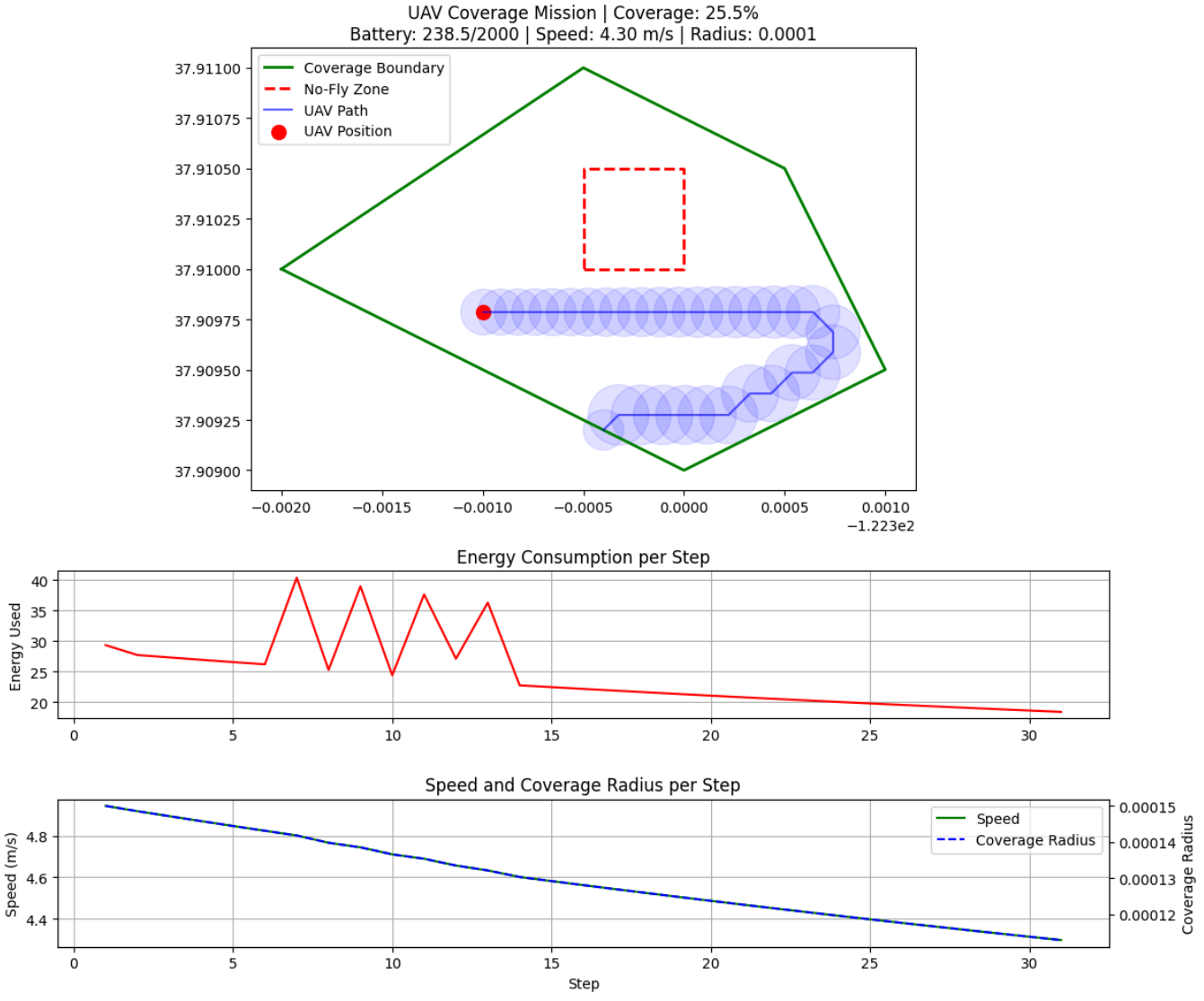



```
Step: Moved from (np.float64(-122.30082568329429), np.float64(37.90978598322125)) to (np.float64(-122.30091101227316), np.float64(37
Energy used: 18.59, remaining: 256.85
Coverage: 24.9%
Coverage radius: 0.0001
```



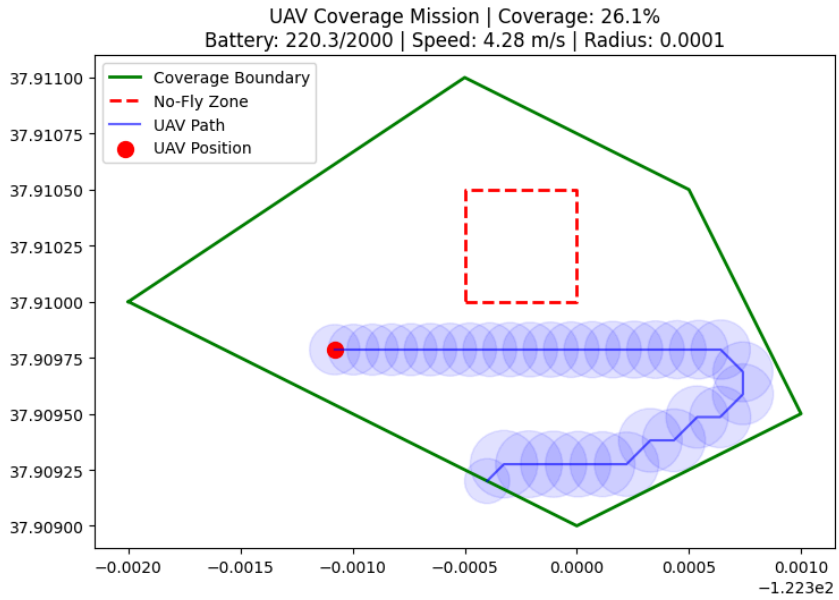


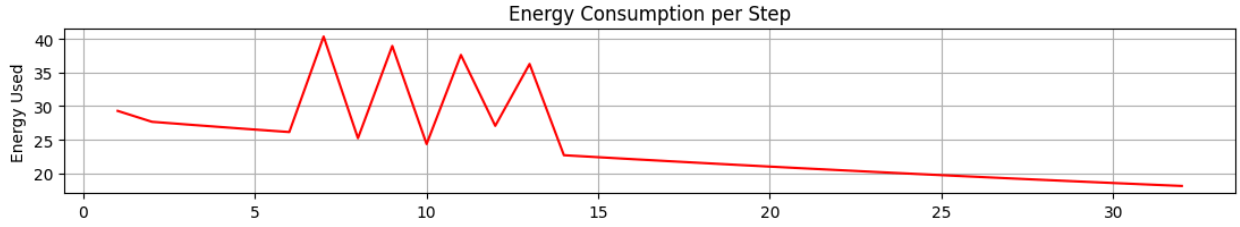

```
Step: Moved from (np.float64(-122.30091101227316), np.float64(37.90978598322125)) to (np.float64(-122.30099564430834), np.float64(37
Energy used: 18.36, remaining: 238.49
Coverage: 25.5%
Coverage radius: 0.0001
```

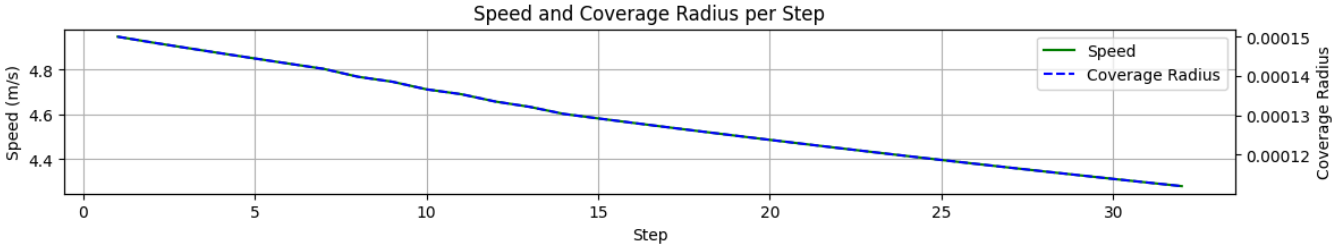



```
Step: Moved from (np.float64(-122.30099564430834), np.float64(37.90978598322125)) to (np.float64(-122.30107958769179), np.float64(37
```

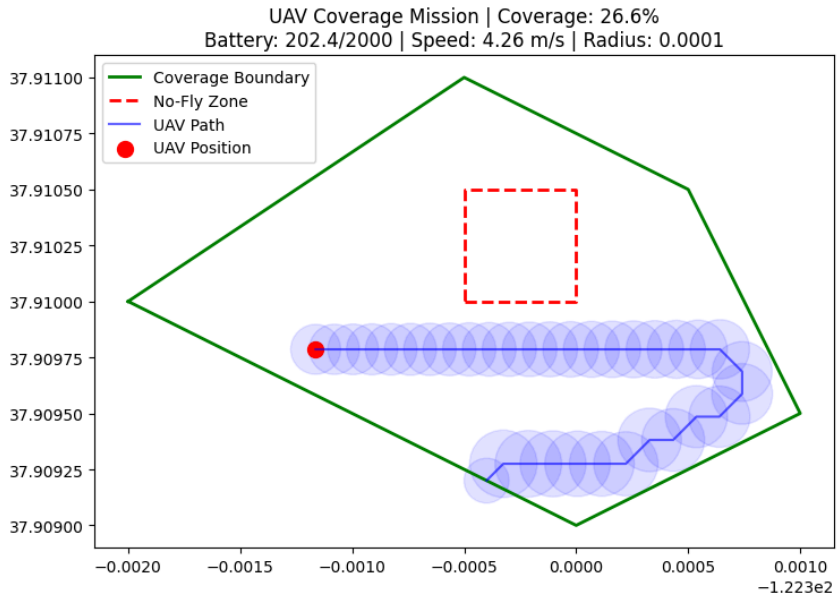
```
Energy used: 18.15, remaining: 220.34
Coverage: 26.1%
Coverage radius: 0.0001
```

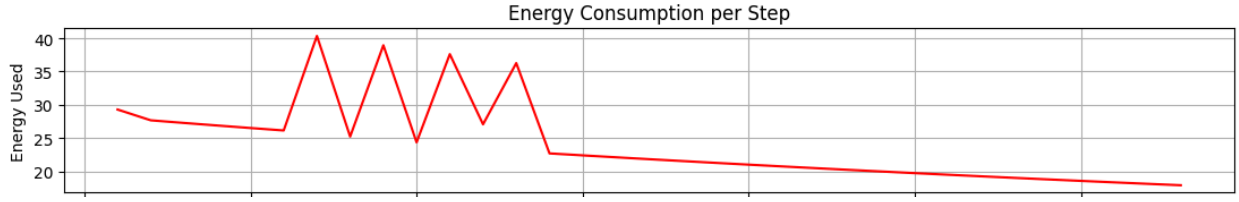


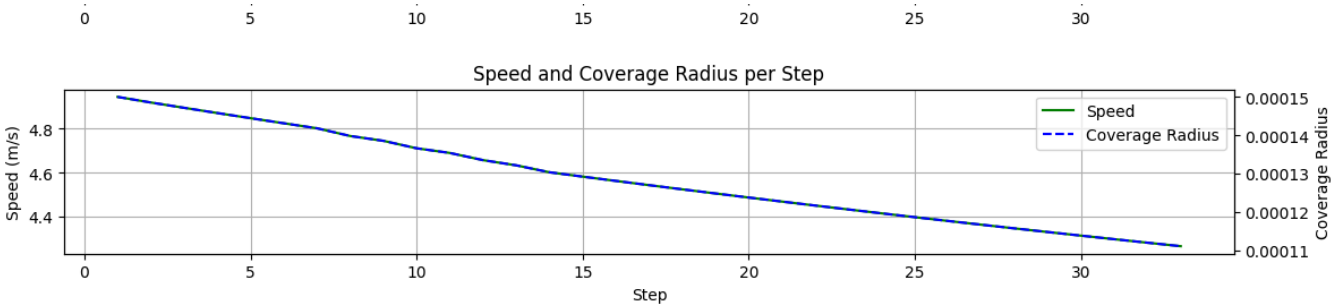




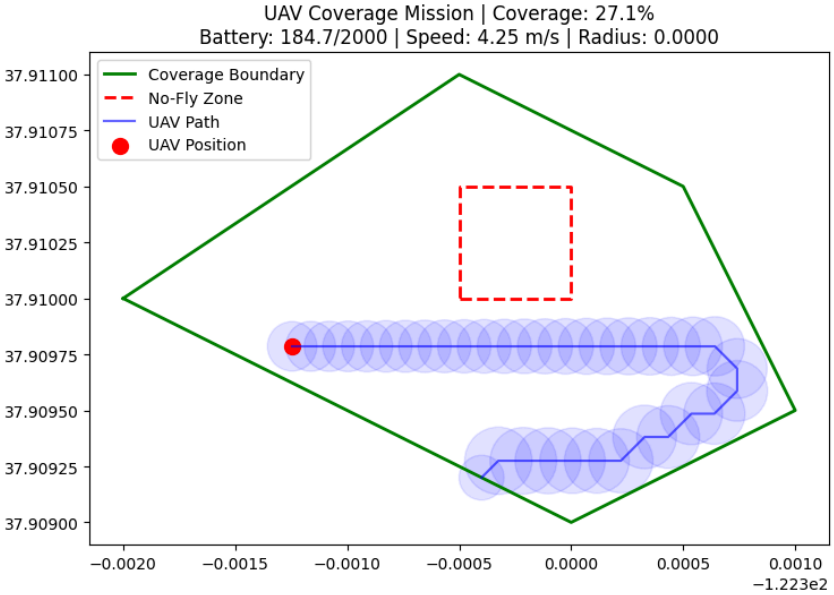

```
Step: Moved from (np.float64(-122.30107958769179), np.float64(37.90978598322125)) to (np.float64(-122.30116285057473), np.float64(37
Energy used: 17.93, remaining: 202.41
Coverage: 26.6%
Coverage radius: 0.0001
```

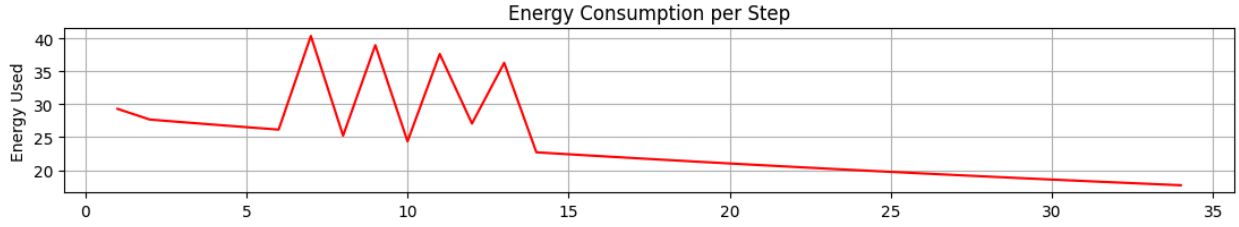


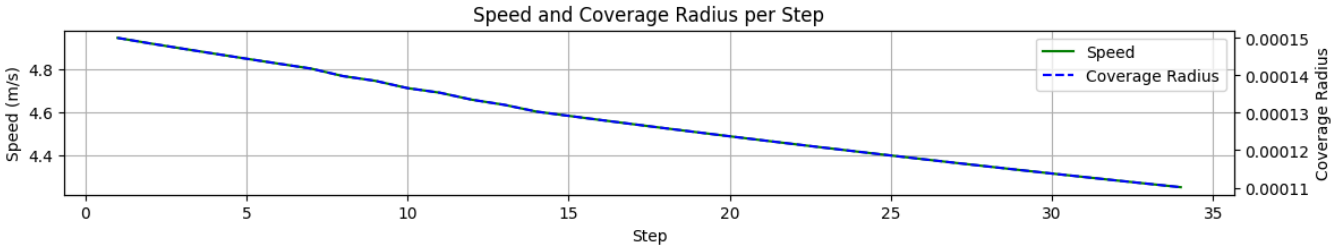




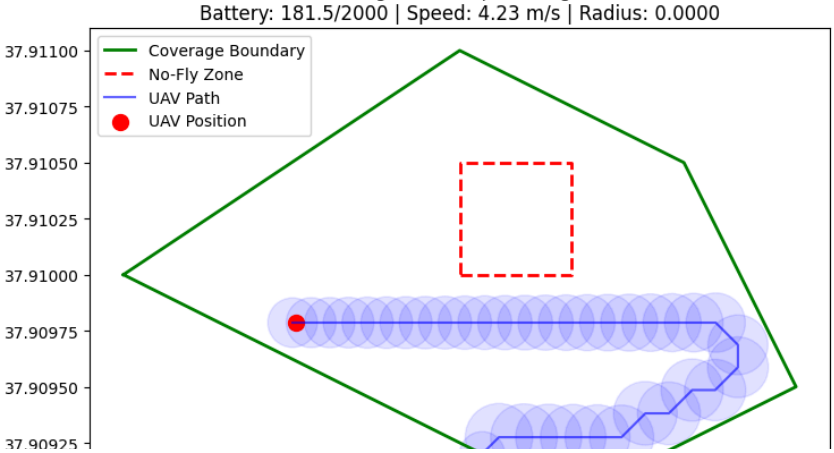

```
Step: Moved from (np.float64(-122.30116285057473), np.float64(37.90978598322125)) to (np.float64(-122.30124544097086), np.float64(37
Energy used: 17.72, remaining: 184.69
Coverage: 27.1%
Coverage radius: 0.0001
```

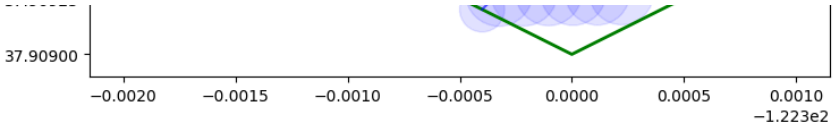


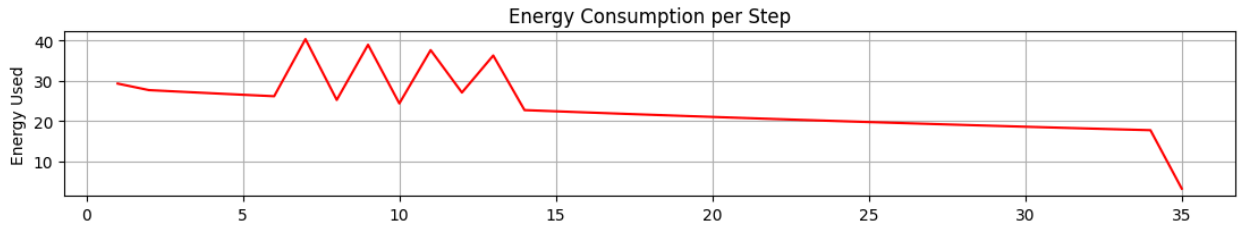


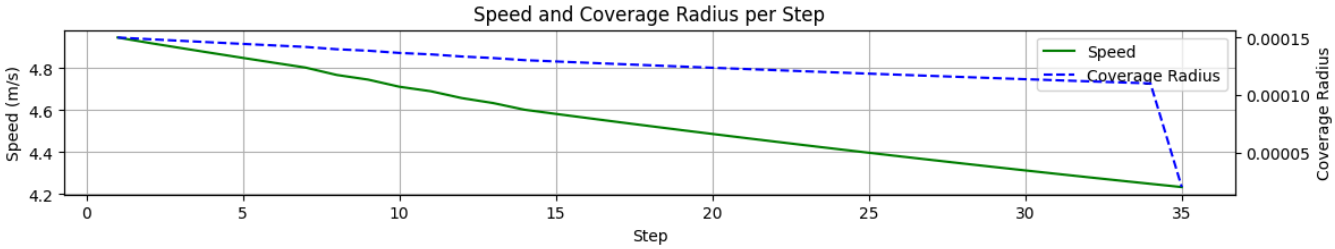



```
Step: Moved from (np.float64(-122.30124544097086), np.float64(37.90978598322125)) to (np.float64(-122.30123044097085), np.float64(37
Energy used: 3.21, remaining: 181.48
Coverage: 27.1%
Coverage radius: 0.0000
```

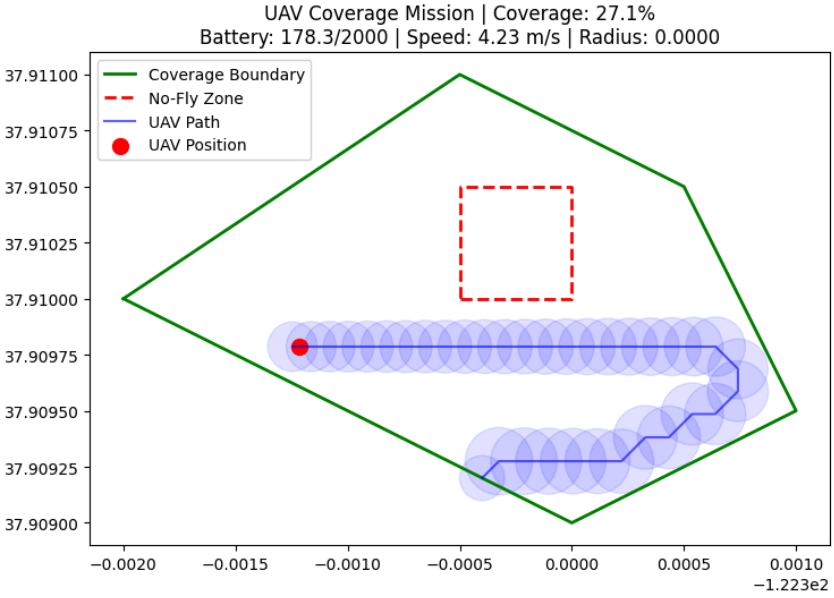


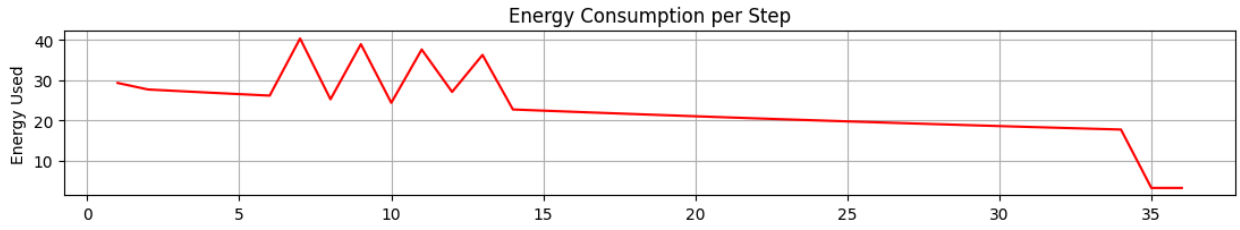


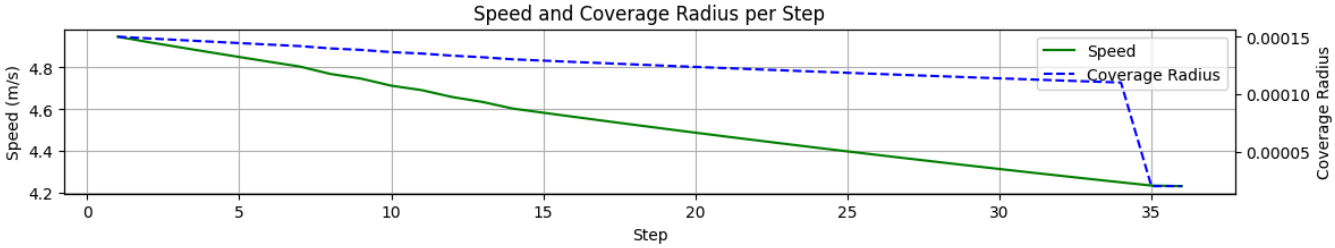




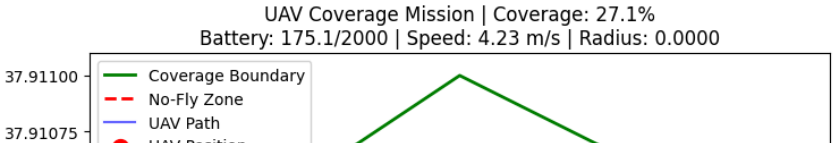

```
Step: Moved from (np.float64(-122.30123044097085), np.float64(37.90978598322125)) to (np.float64(-122.30121544097085), np.float64(37
Energy used: 3.20, remaining: 178.28
Coverage: 27.1%
Coverage radius: 0.0000
```

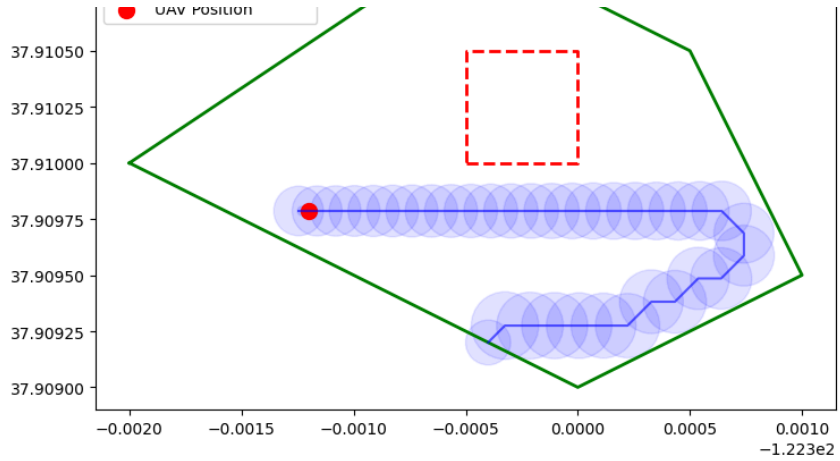


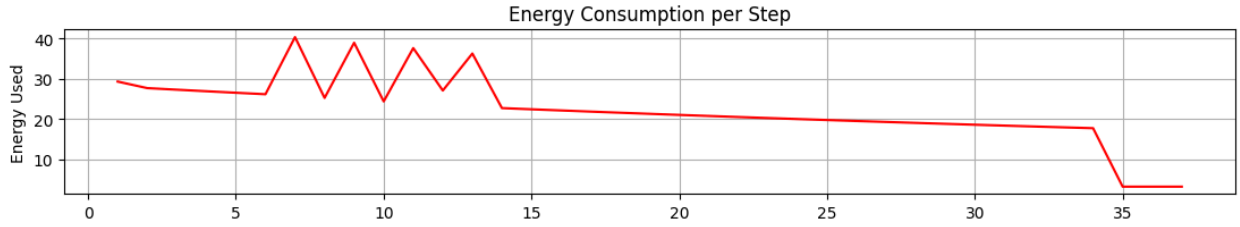


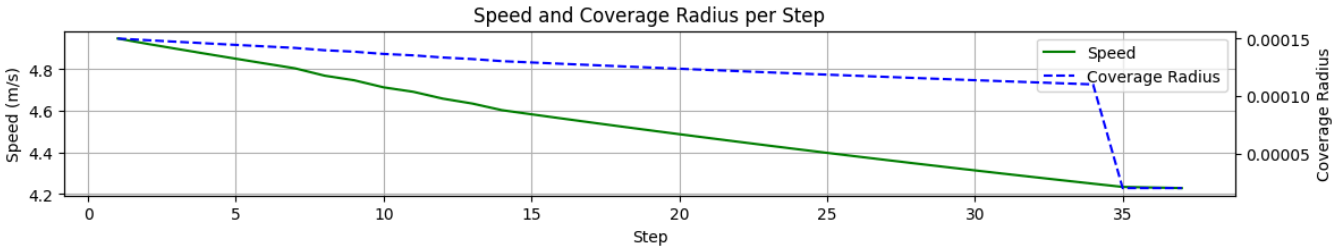



```
Step: Moved from (np.float64(-122.30121544097085), np.float64(37.90978598322125)) to (np.float64(-122.30120044097085), np.float64(37
Energy used: 3.20, remaining: 175.07
Coverage: 27.1%
Coverage radius: 0.0000
```

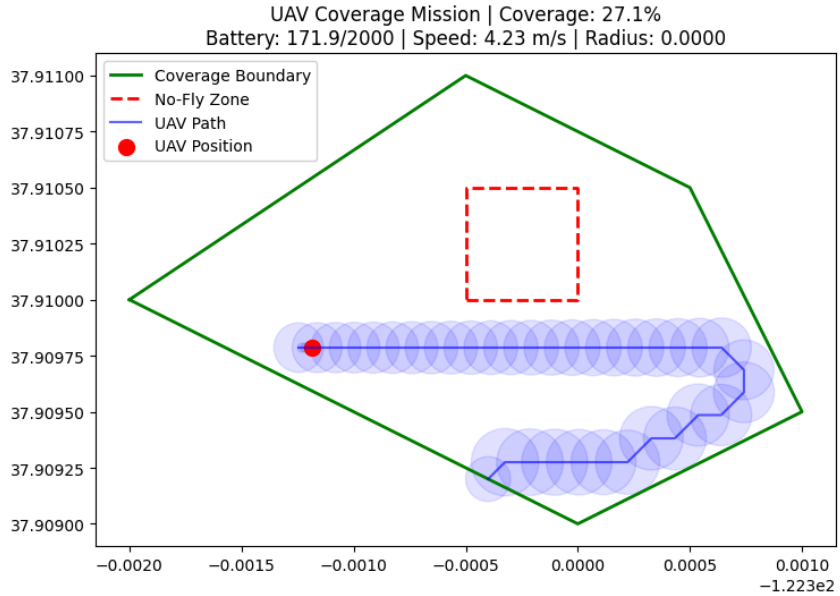


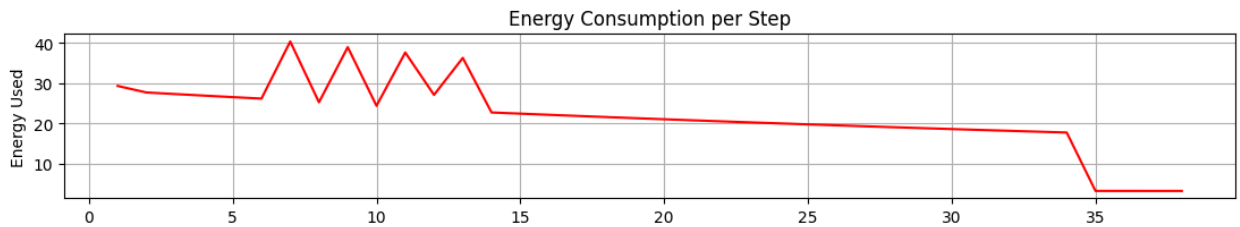


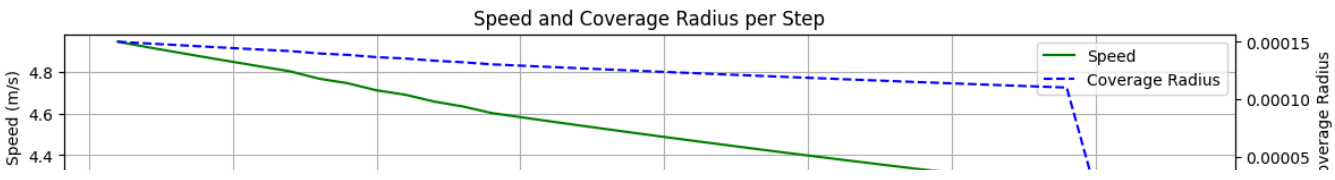




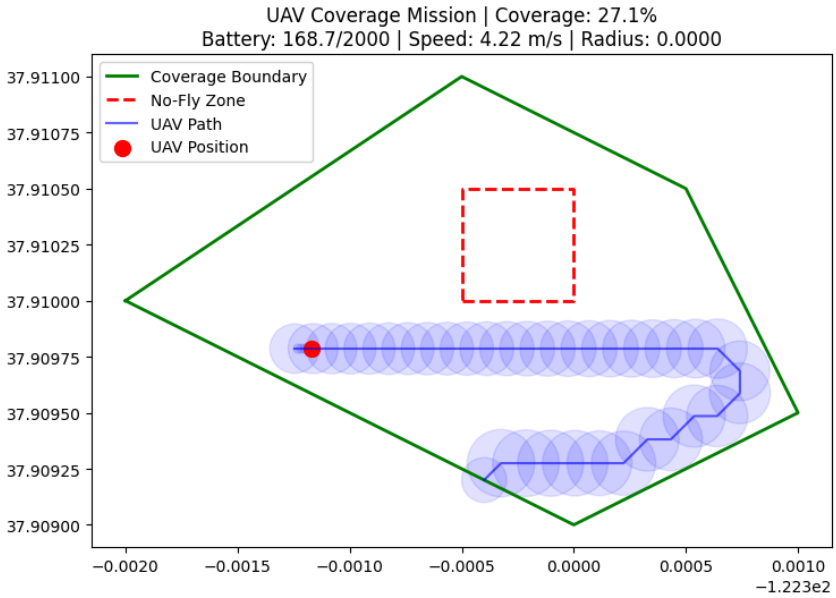

```
Step: Moved from (np.float64(-122.30120044097085), np.float64(37.90978598322125)) to (np.float64(-122.30118544097084), np.float64(37
Energy used: 3.20, remaining: 171.87
Coverage: 27.1%
Coverage radius: 0.0000
```

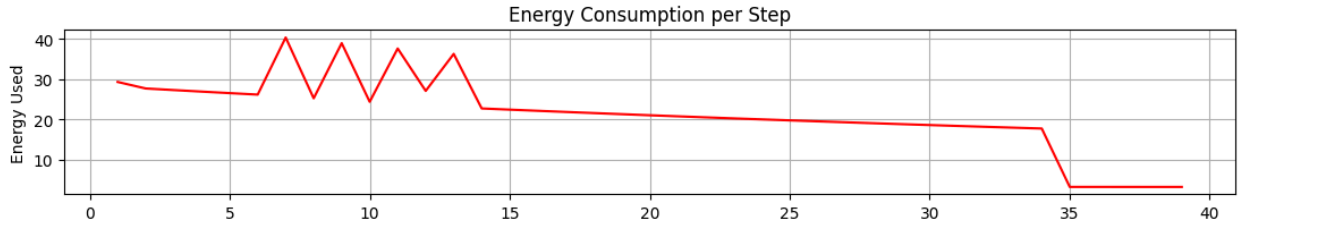


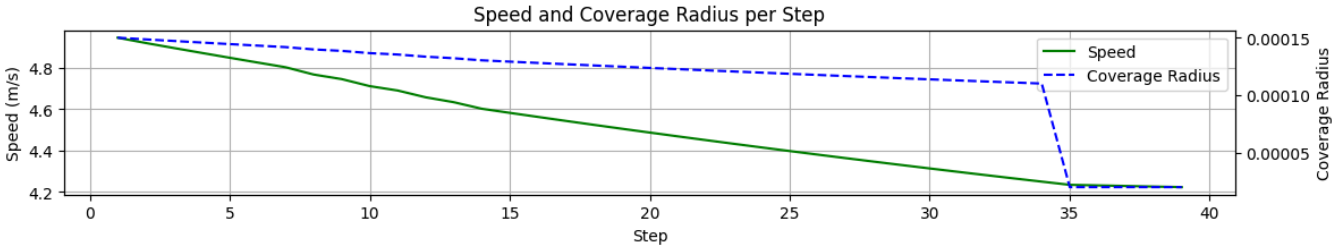




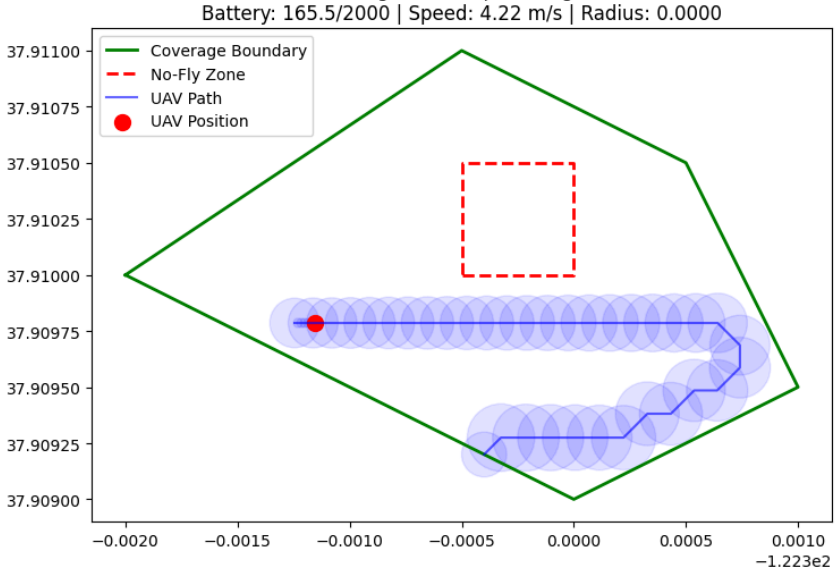

```
Step: Moved from (np.float64(-122.30118544097084), np.float64(37.90978598322125)) to (np.float64(-122.30117044097084), np.float64(37
Energy used: 3.20, remaining: 168.67
Coverage: 27.1%
Coverage radius: 0.0000
```

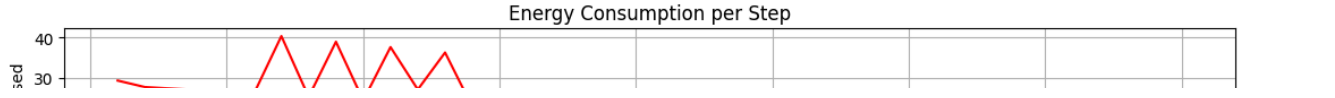


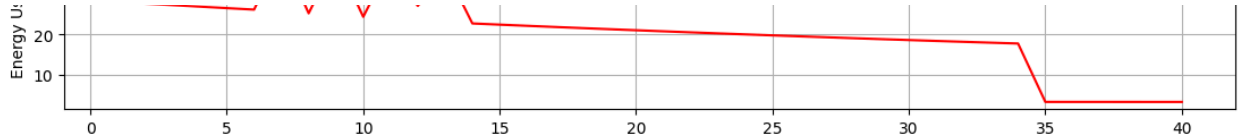


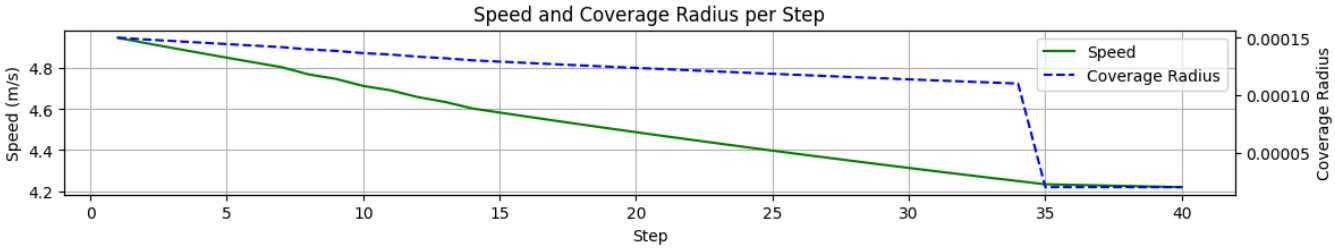



```
Step: Moved from (np.float64(-122.30117044097084), np.float64(37.90978598322125)) to (np.float64(-122.30115544097083), np.float64(37
Energy used: 3.20, remaining: 165.48
Coverage: 27.1%
Coverage radius: 0.0000
```

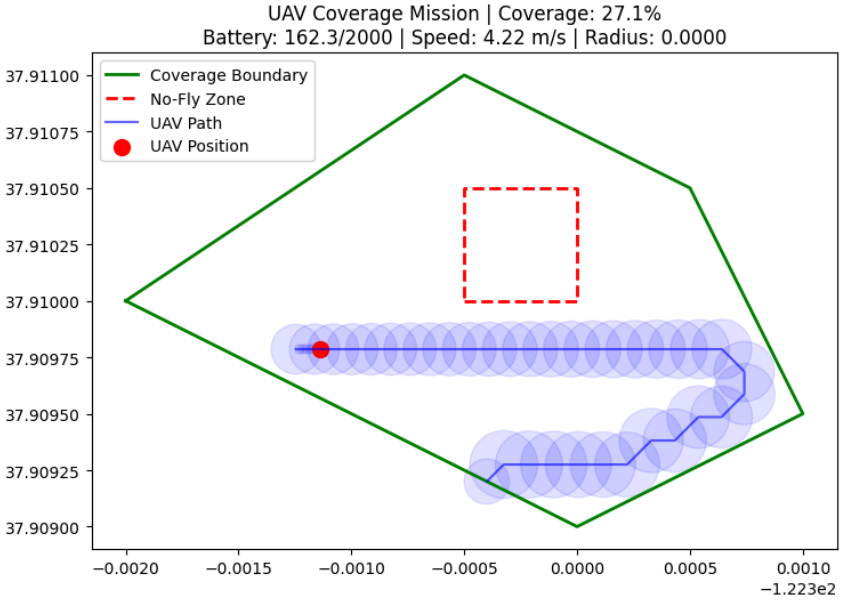


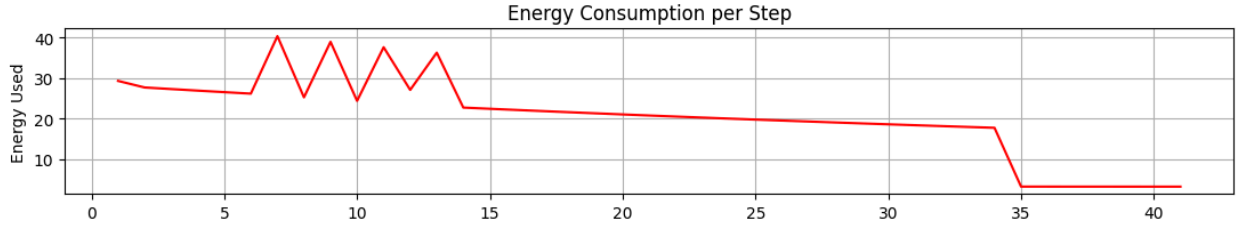


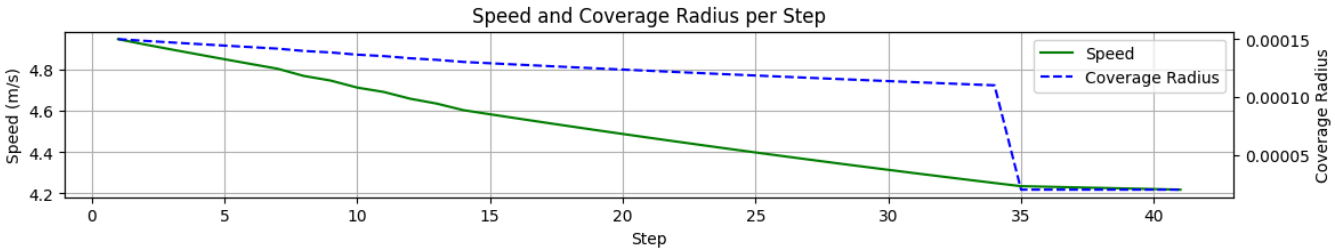




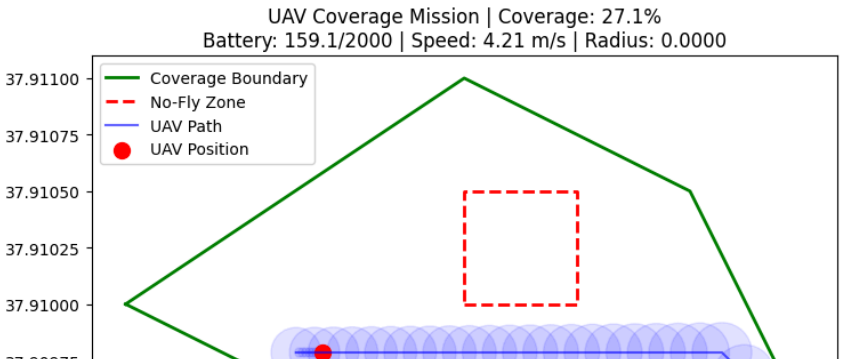

```
Step: Moved from (np.float64(-122.30115544097083), np.float64(37.90978598322125)) to (np.float64(-122.30114044097083), np.float64(37
Energy used: 3.19, remaining: 162.28
Coverage: 27.1%
Coverage radius: 0.0000
```

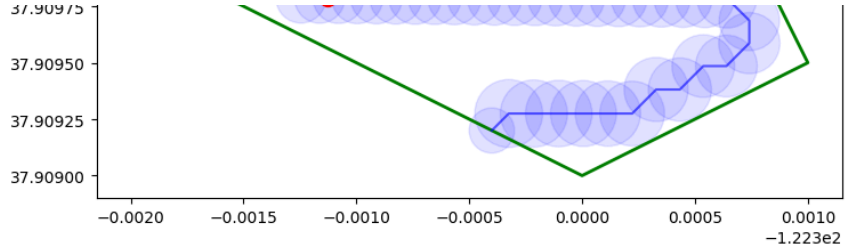


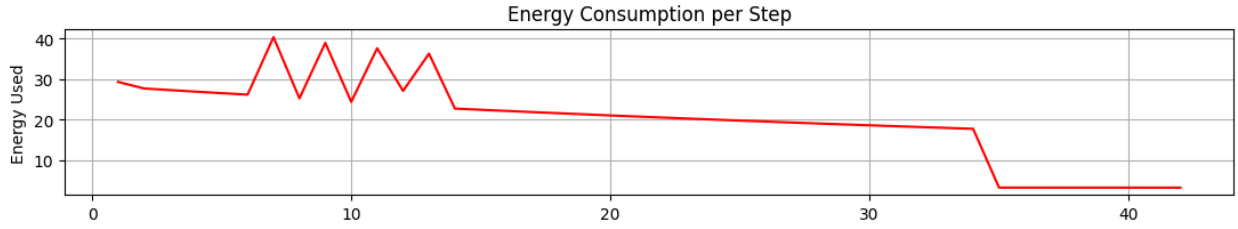


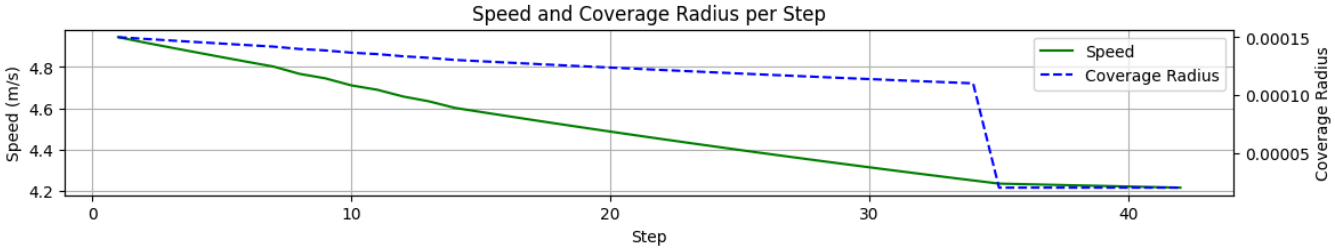



```
Step: Moved from (np.float64(-122.30114044097083), np.float64(37.90978598322125)) to (np.float64(-122.30112544097082), np.float64(37
Energy used: 3.19, remaining: 159.09
Coverage: 27.1%
Coverage radius: 0.0000
```

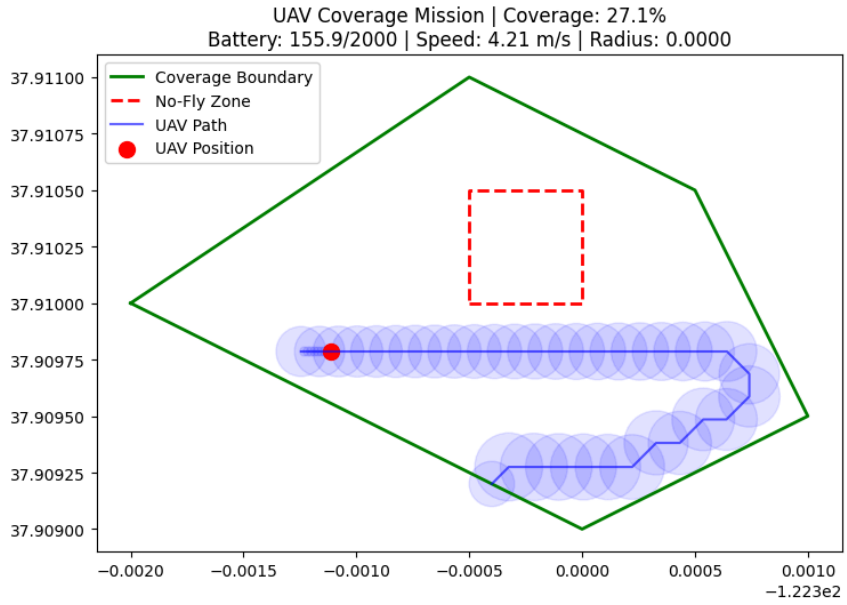


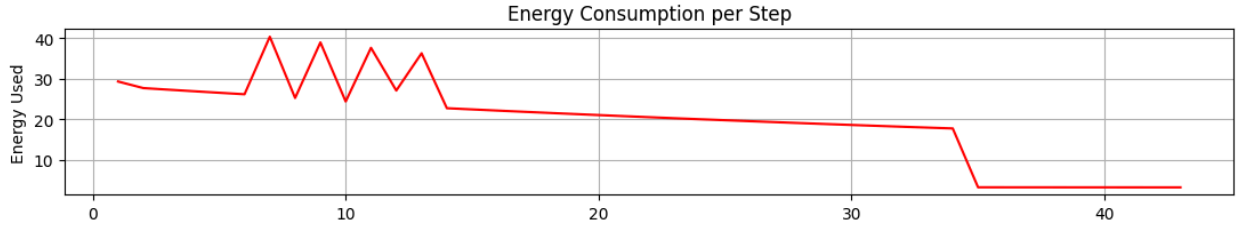


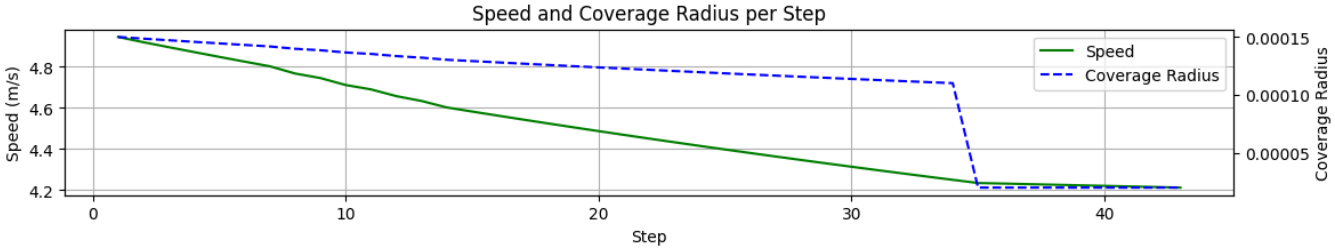




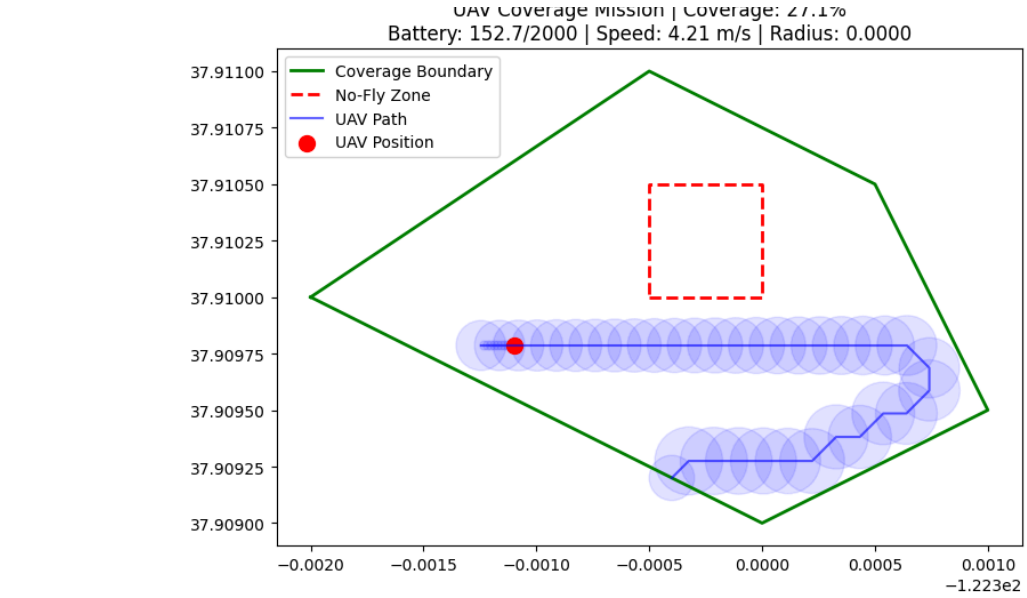

```
Step: Moved from (np.float64(-122.30112544097082), np.float64(37.90978598322125)) to (np.float64(-122.30111044097082), np.float64(37
Energy used: 3.19, remaining: 155.90
Coverage: 27.1%
Coverage radius: 0.0000
```

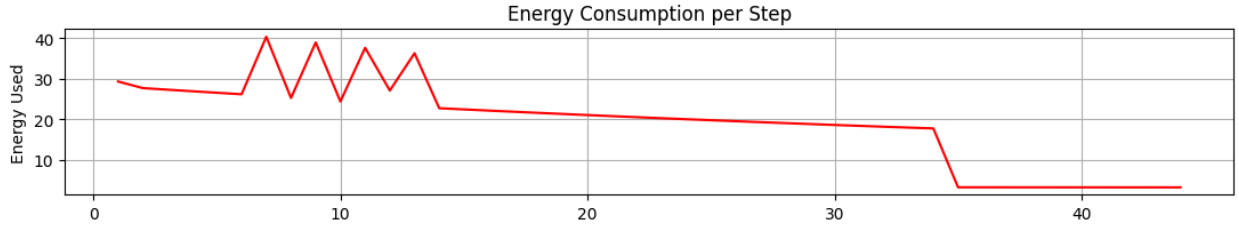


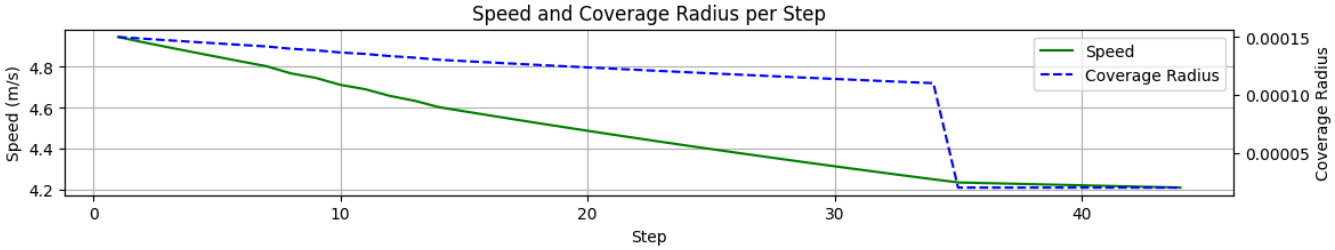




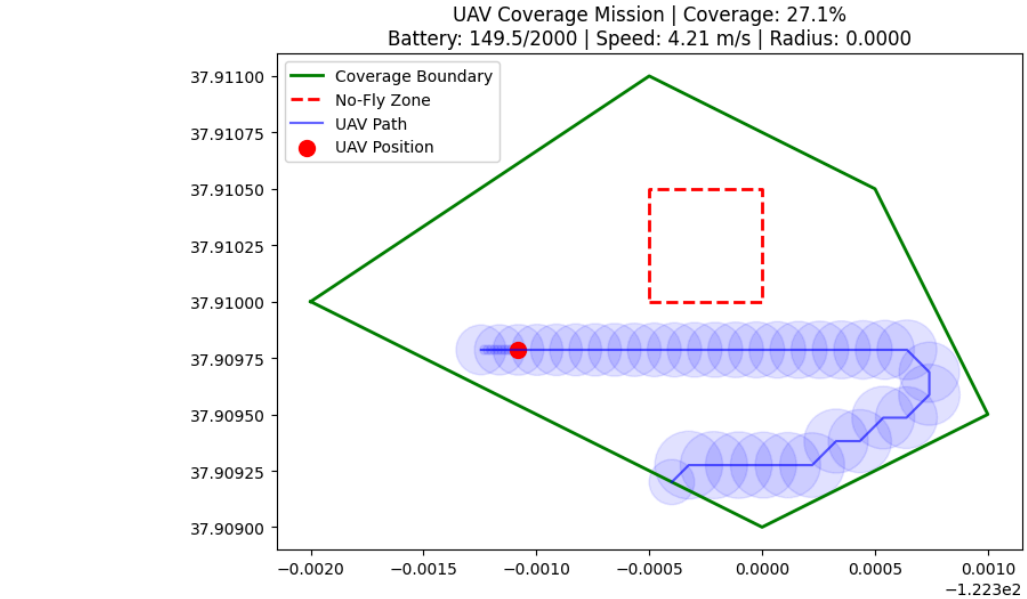

```
Step: Moved from (np.float64(-122.30111044097082), np.float64(37.90978598322125)) to (np.float64(-122.30109544097081), np.float64(37
Energy used: 3.19, remaining: 152.71
Coverage: 27.1%
Coverage radius: 0.0000
```

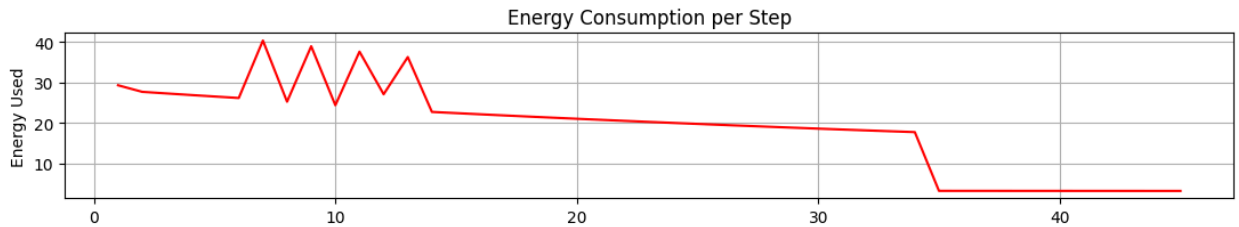




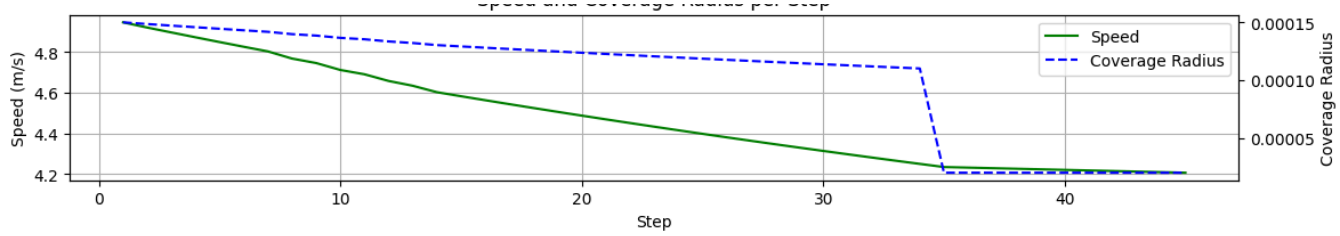



```
Step: Moved from (np.float64(-122.30109544097081), np.float64(37.90978598322125)) to (np.float64(-122.30108044097081), np.float64(37
Energy used: 3.19, remaining: 149.53
Coverage: 27.1%
Coverage radius: 0.0000
```

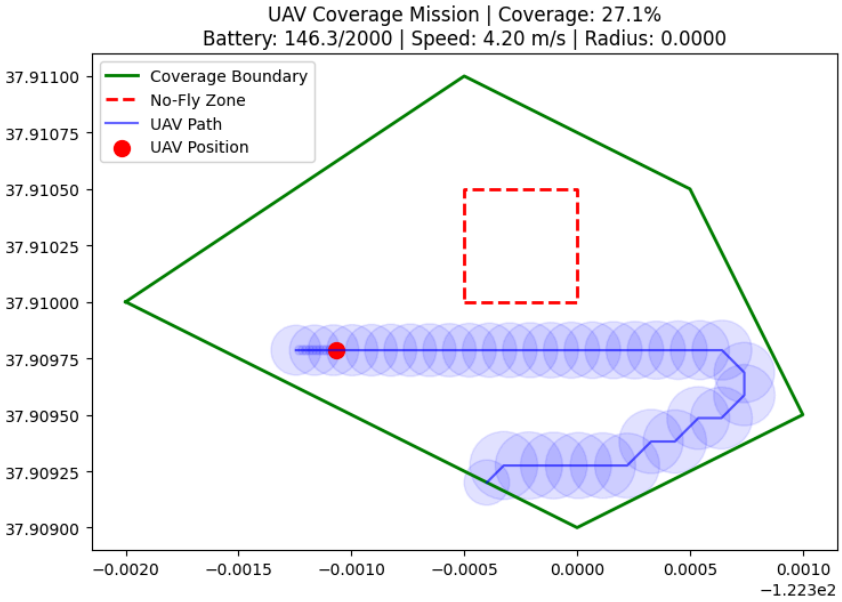


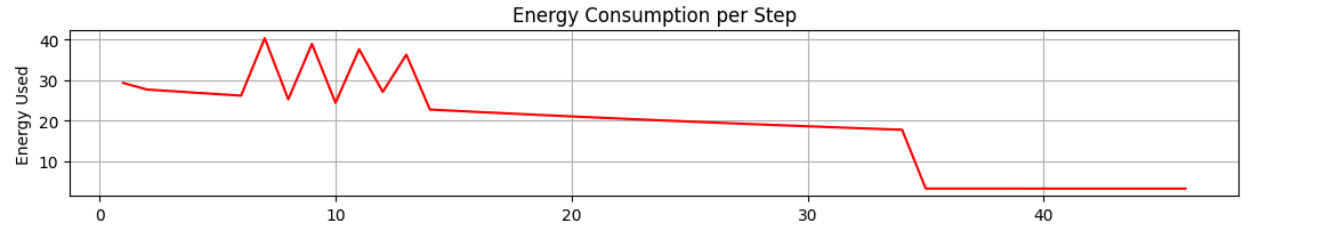


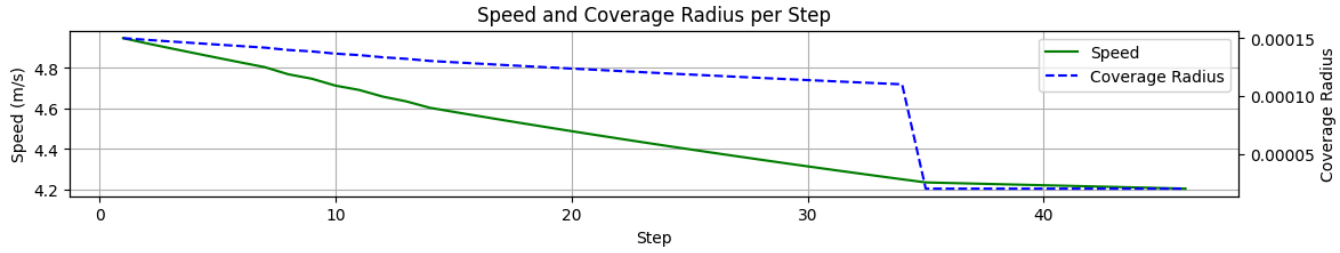
Speed and Coverage Radius per Step



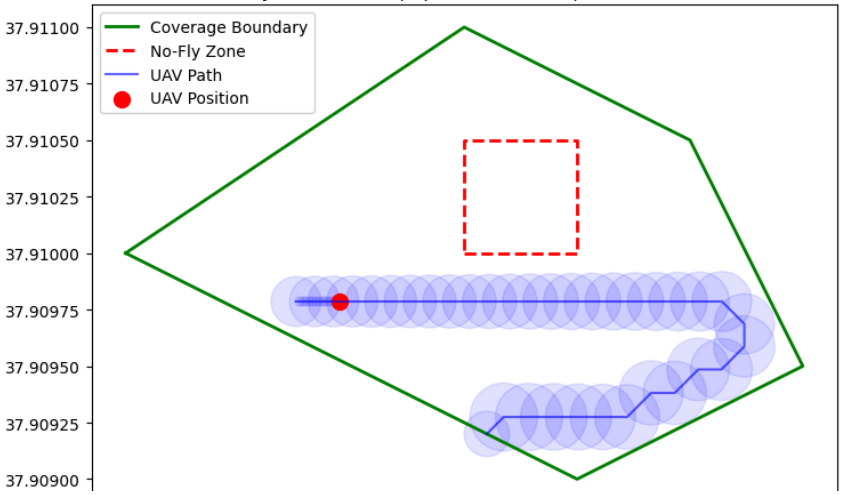

```
Step: Moved from (np.float64(-122.30108044097081), np.float64(37.90978598322125)) to (np.float64(-122.3010654409708), np.float64(37
Energy used: 3.18, remaining: 146.34
Coverage: 27.1%
Coverage radius: 0.0000
```

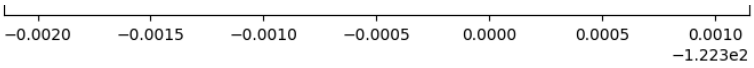


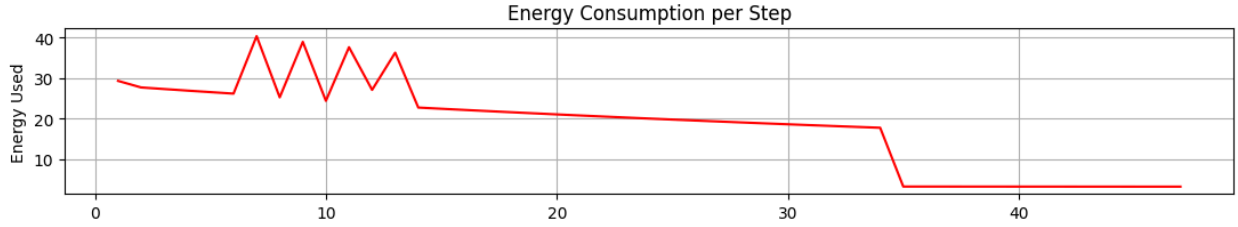


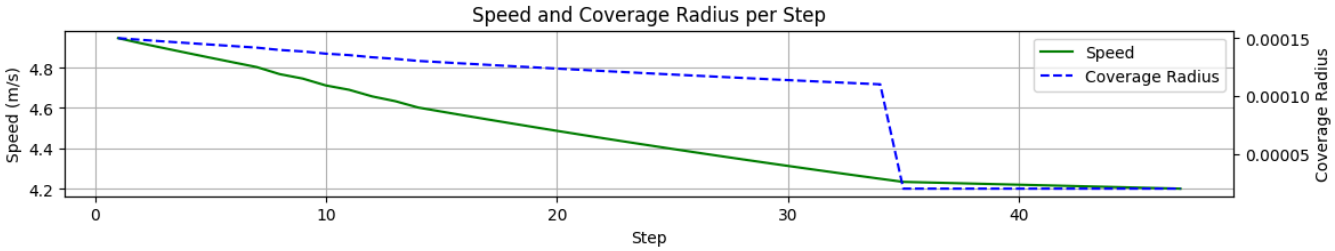



```
Step: Moved from (np.float64(-122.3010654409708), np.float64(37.90978598322125)) to (np.float64(-122.3010504409708), np.float64(37.9
Energy used: 3.18, remaining: 143.16
Coverage: 27.1%
Coverage radius: 0.0000
```

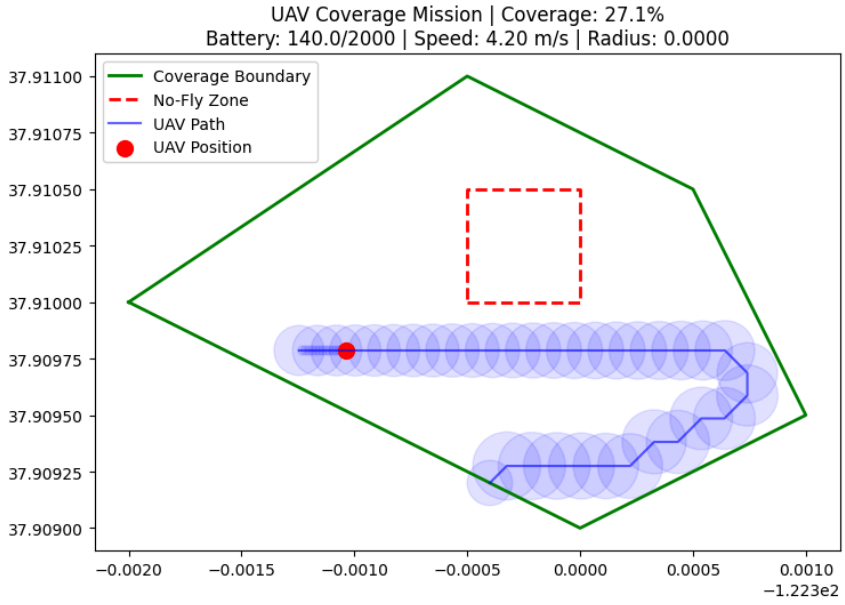


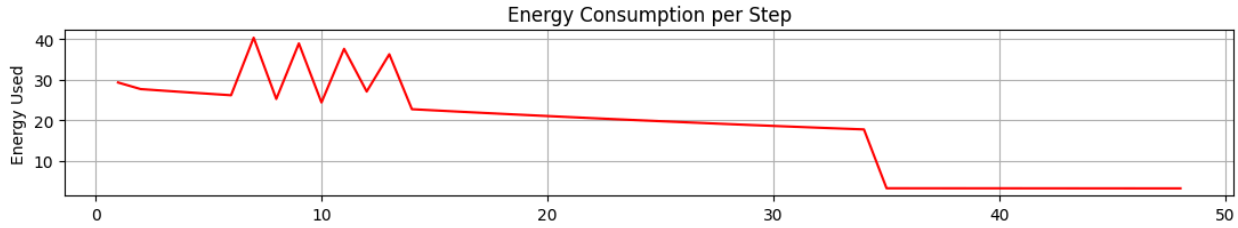


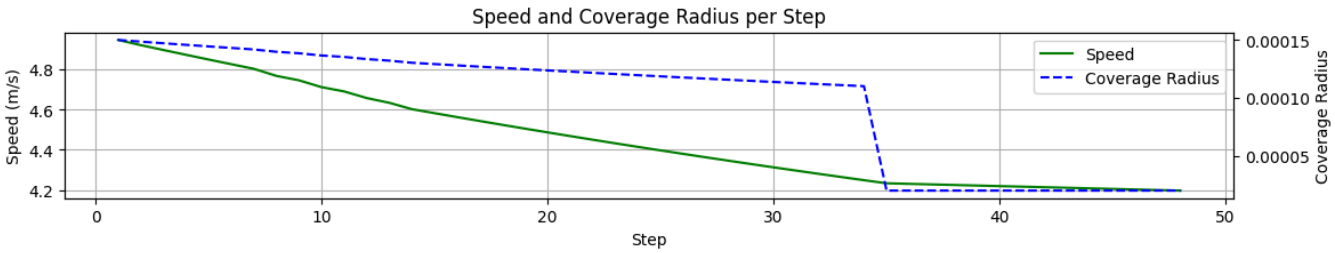




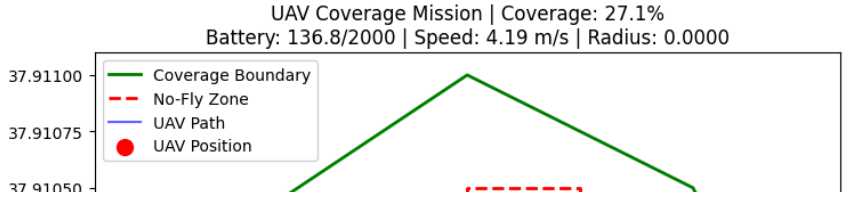

```
Step: Moved from (np.float64(-122.3010504409708), np.float64(37.90978598322125)) to (np.float64(-122.3010354409708), np.float64(37.9
Energy used: 3.18, remaining: 139.98
Coverage: 27.1%
Coverage radius: 0.0000
```

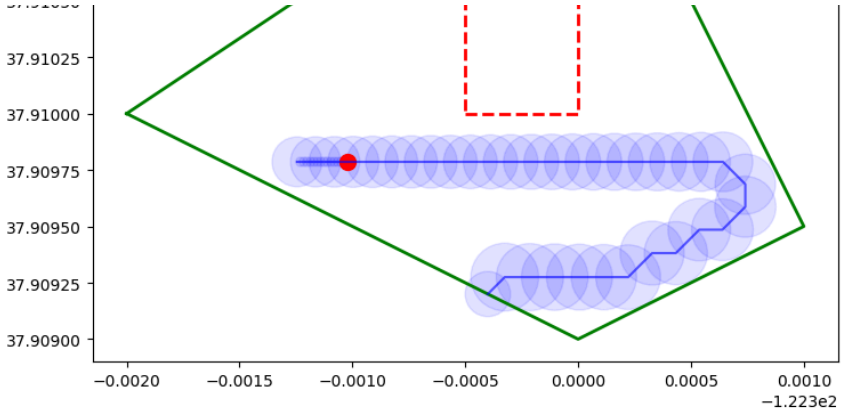


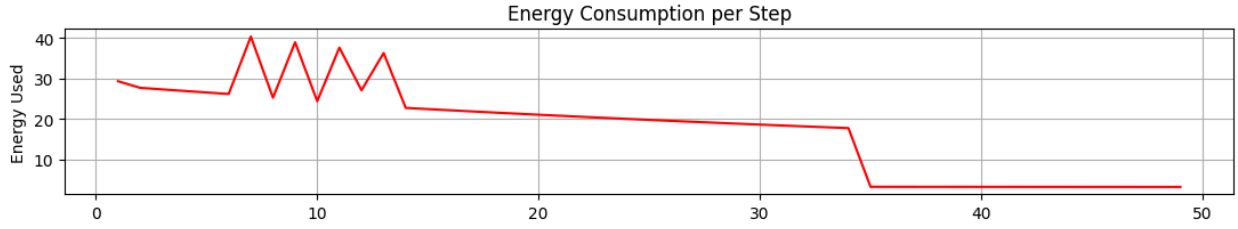


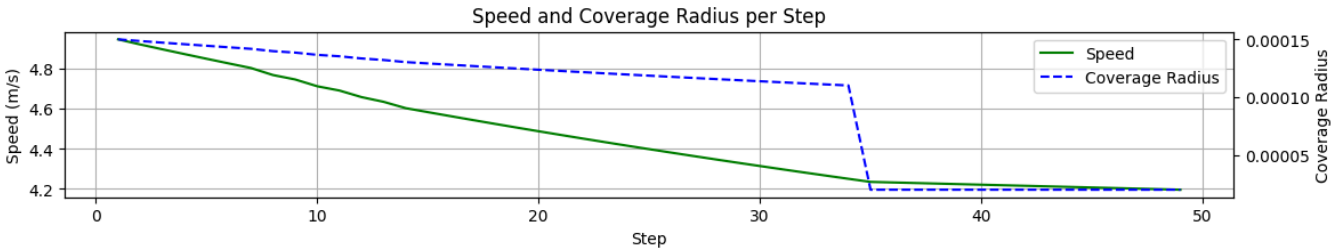



```
Step: Moved from (np.float64(-122.3010354409708), np.float64(37.90978598322125)) to (np.float64(-122.30102044097079), np.float64(37
Energy used: 3.18, remaining: 136.80
Coverage: 27.1%
Coverage radius: 0.0000
```

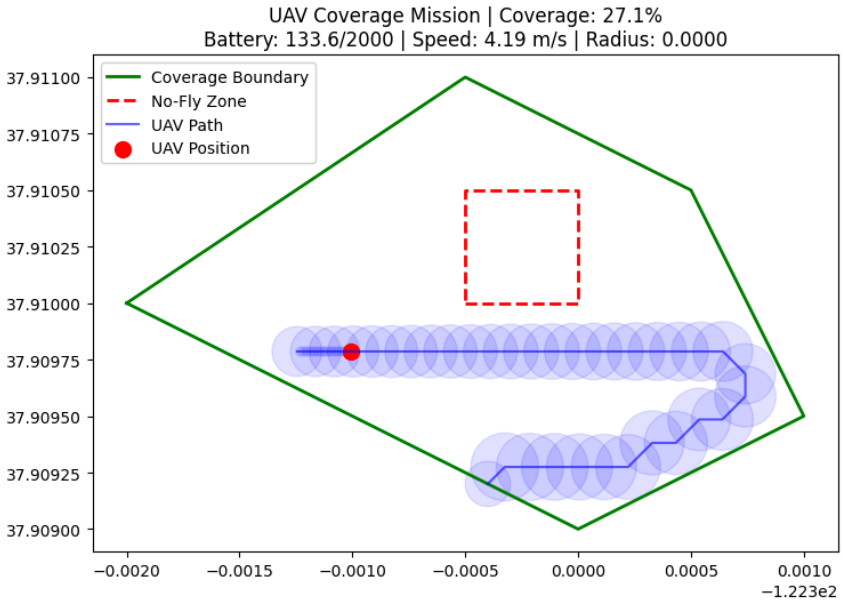


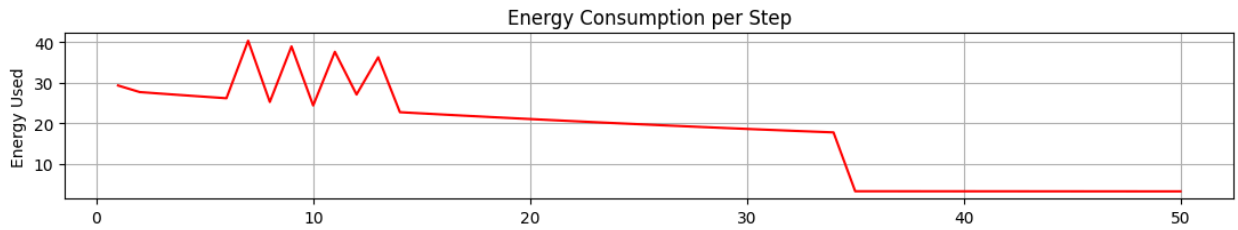


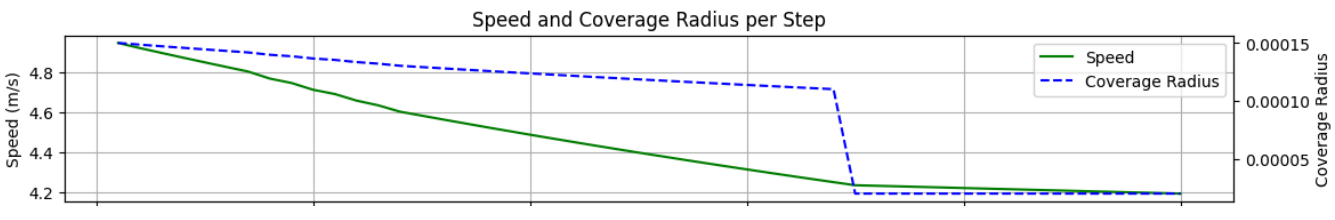




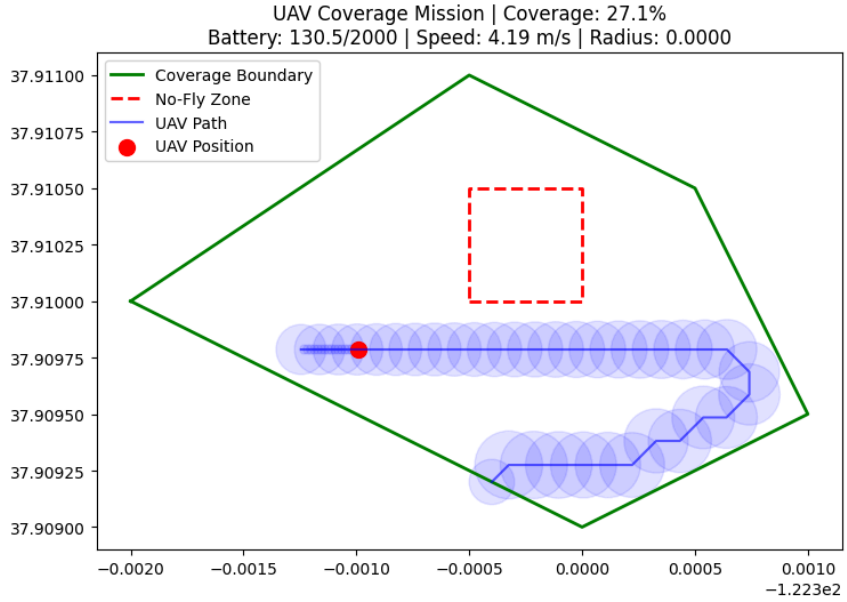

```
Step: Moved from (np.float64(-122.30102044097079), np.float64(37.90978598322125)) to (np.float64(-122.30100544097078), np.float64(37
Energy used: 3.18, remaining: 133.63
Coverage: 27.1%
Coverage radius: 0.0000
```

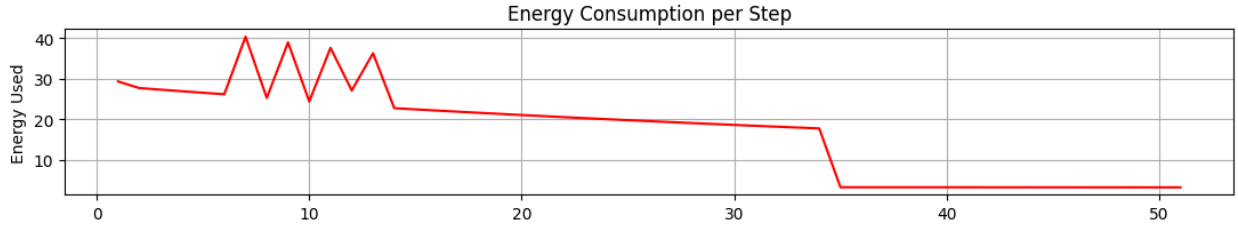


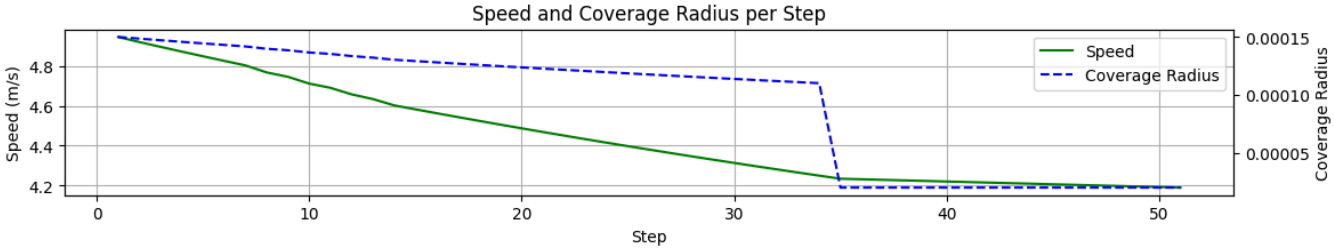




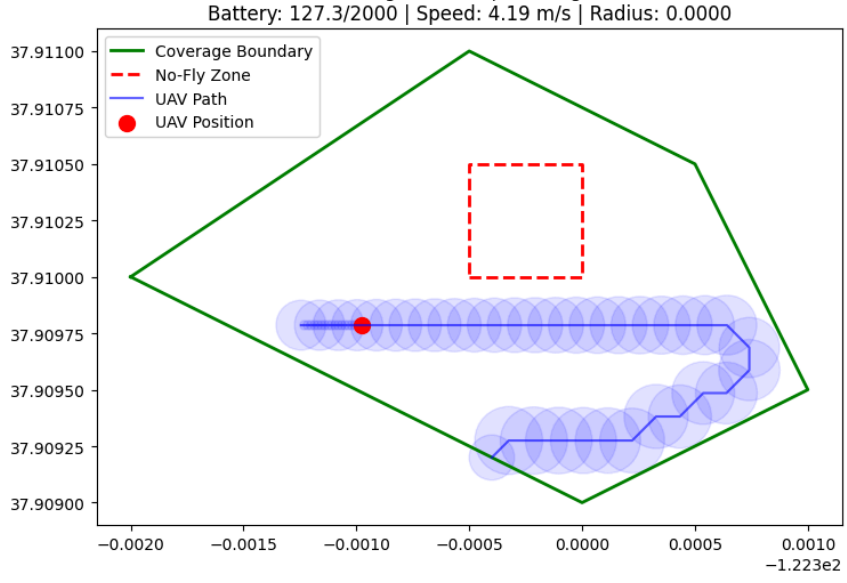

```
Step: Moved from (np.float64(-122.30100544097078), np.float64(37.90978598322125)) to (np.float64(-122.30099044097078), np.float64(37
Energy used: 3.17, remaining: 130.45
Coverage: 27.1%
Coverage radius: 0.0000
```

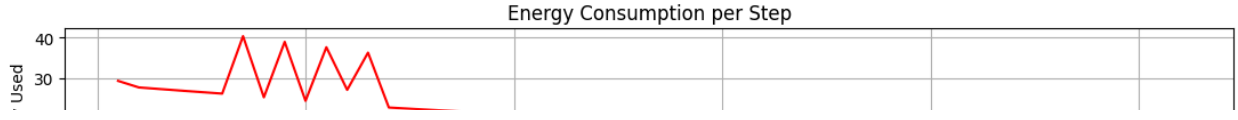


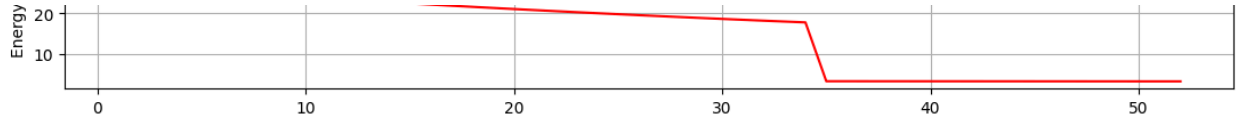


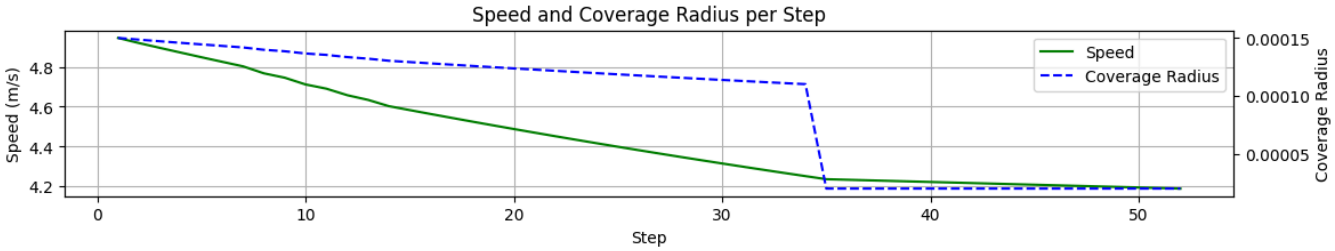



```
Step: Moved from (np.float64(-122.30099044097078), np.float64(37.90978598322125)) to (np.float64(-122.30097544097077), np.float64(37
Energy used: 3.17, remaining: 127.28
Coverage: 27.1%
Coverage radius: 0.0000
```

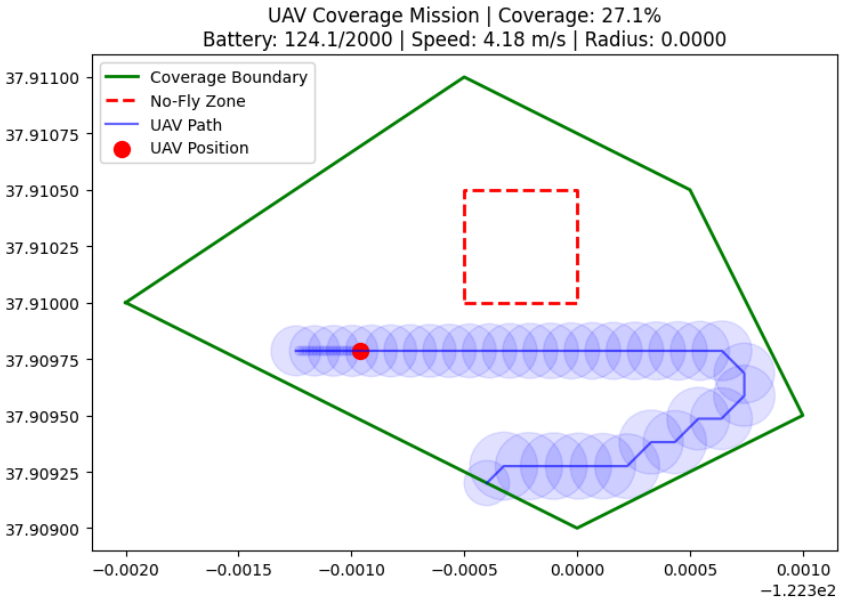


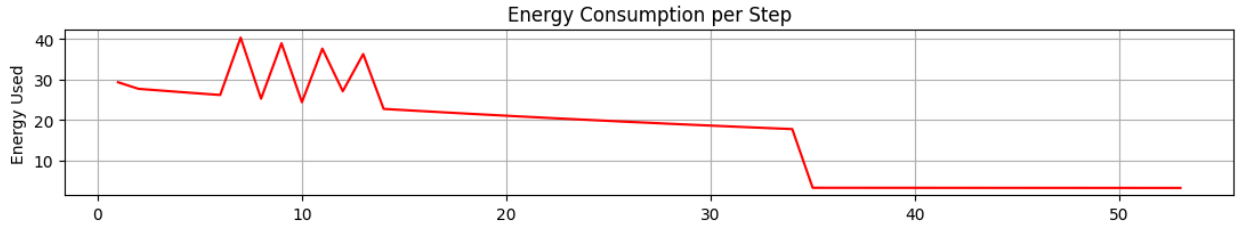


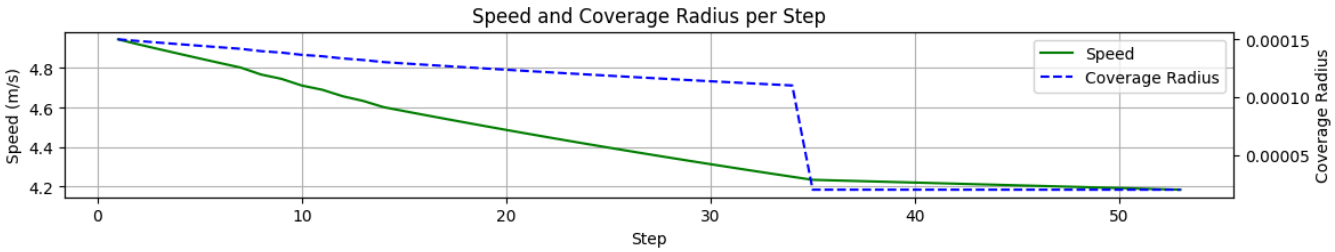




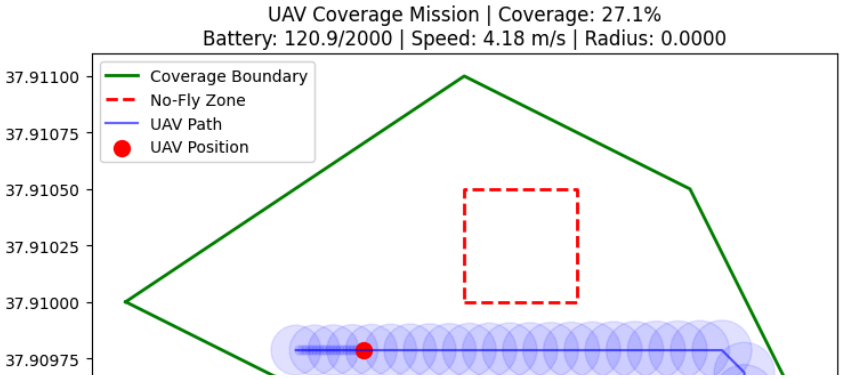

```
Step: Moved from (np.float64(-122.30097544097077), np.float64(37.90978598322125)) to (np.float64(-122.30096044097077), np.float64(37
Energy used: 3.17, remaining: 124.11
Coverage: 27.1%
Coverage radius: 0.0000
```

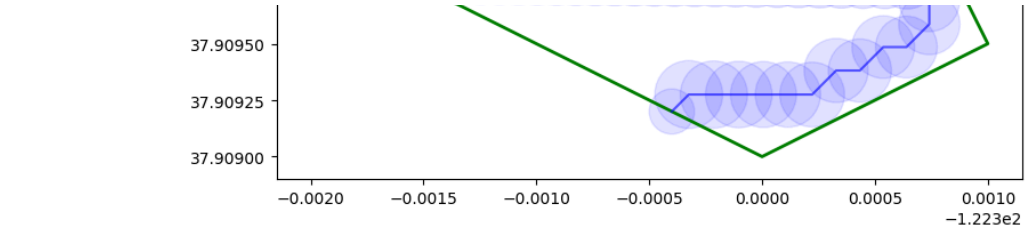


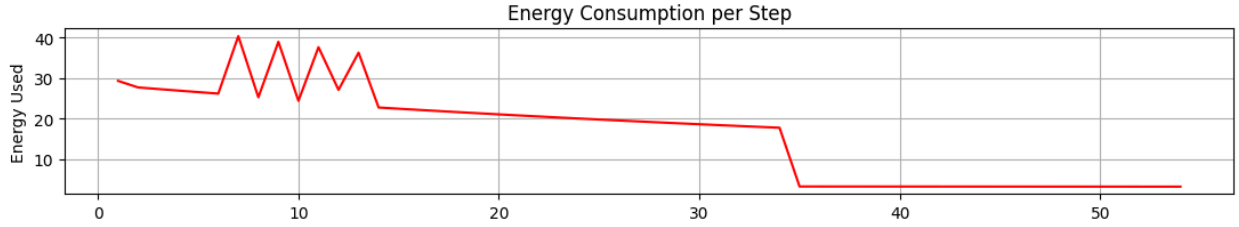


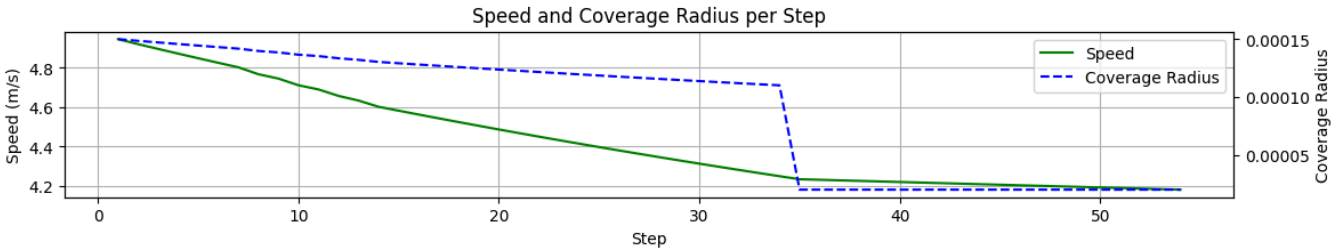



```
Step: Moved from (np.float64(-122.30096044097077), np.float64(37.90978598322125)) to (np.float64(-122.30094544097076), np.float64(37
Energy used: 3.17, remaining: 120.95
Coverage: 27.1%
Coverage radius: 0.0000
```

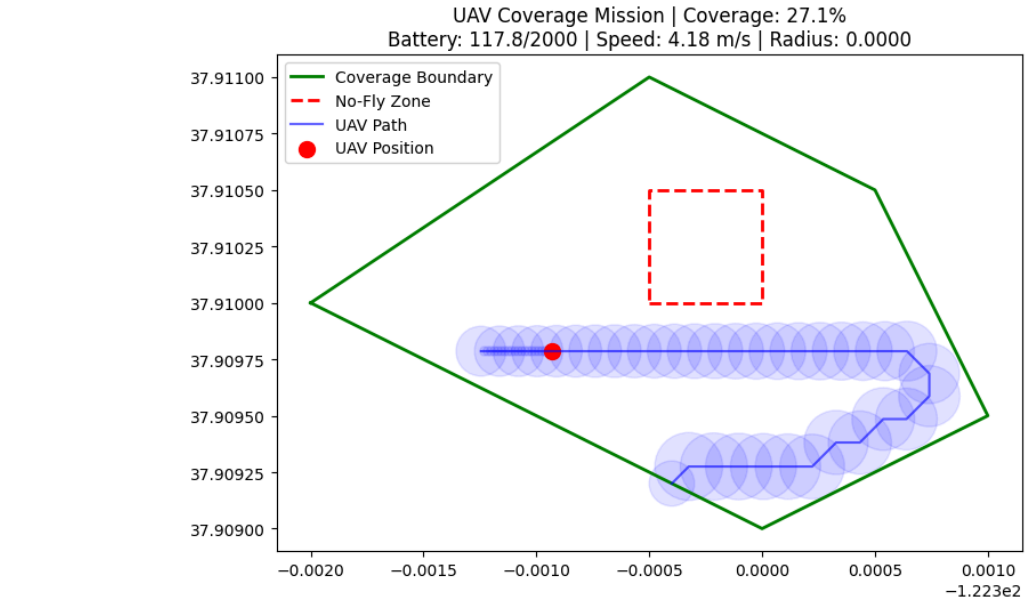


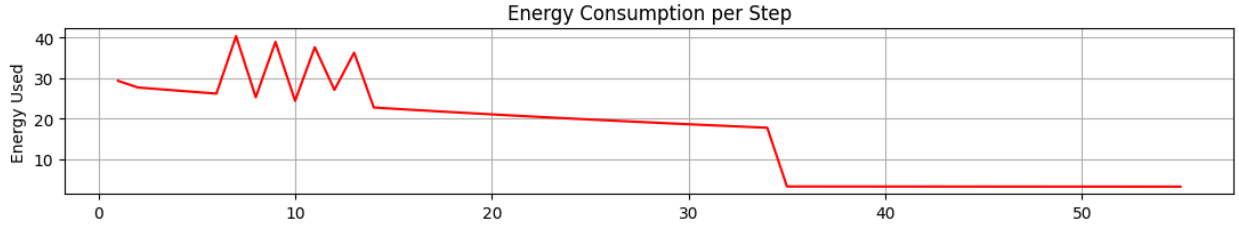


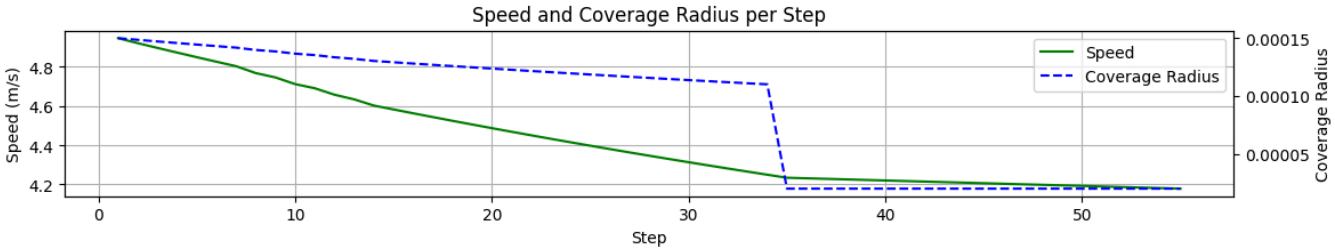





```
Step: Moved from (np.float64(-122.30094544097076), np.float64(37.90978598322125)) to (np.float64(-122.30093044097076), np.float64(37
Energy used: 3.16, remaining: 117.78
Coverage: 27.1%
Coverage radius: 0.0000
```

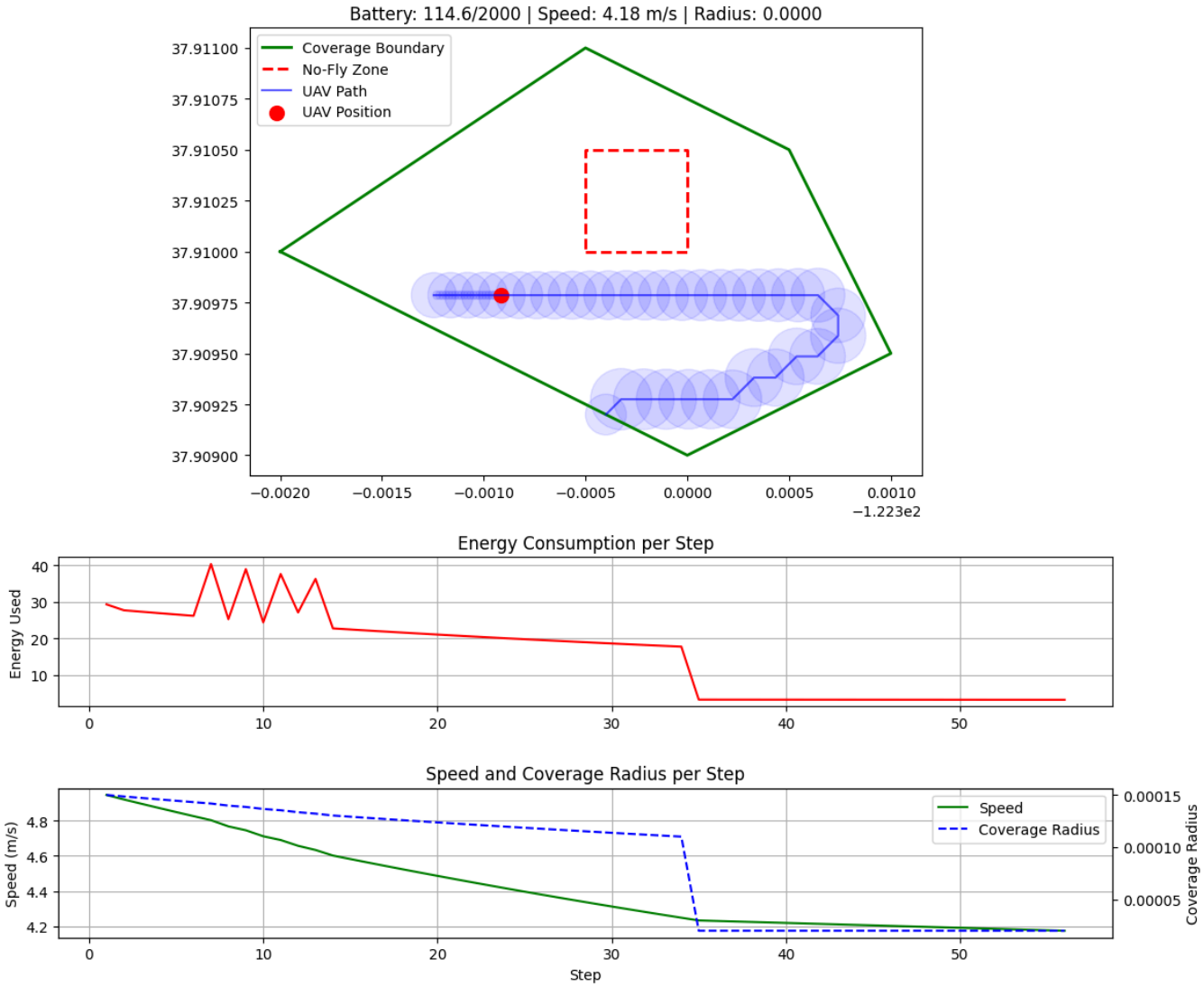




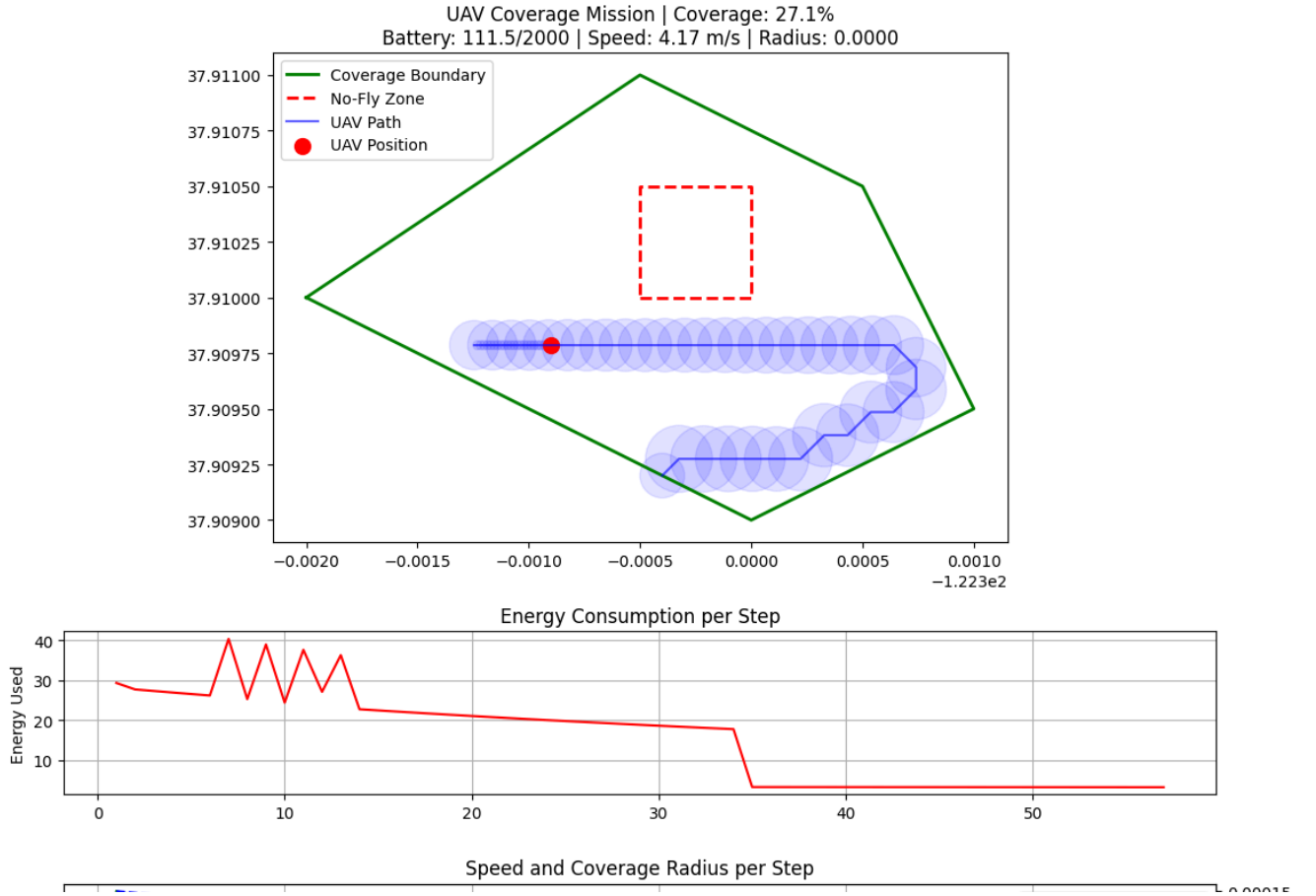


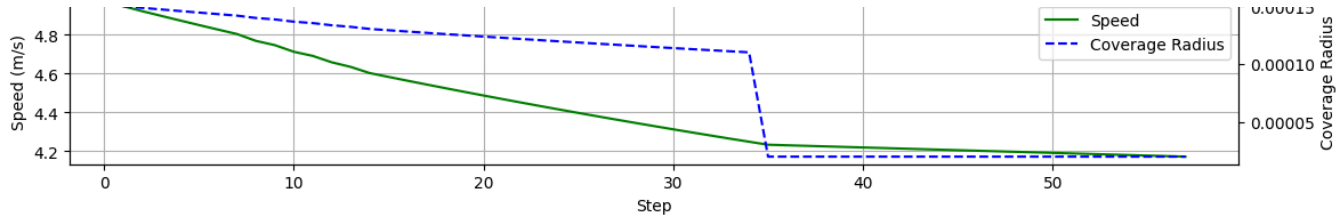

```
Step: Moved from (np.float64(-122.30093044097076), np.float64(37.90978598322125)) to (np.float64(-122.30091544097075), np.float64(37
Energy used: 3.16, remaining: 114.62
Coverage: 27.1%
Coverage radius: 0.0000
```

UAV Coverage Mission | Coverage: 27.1%

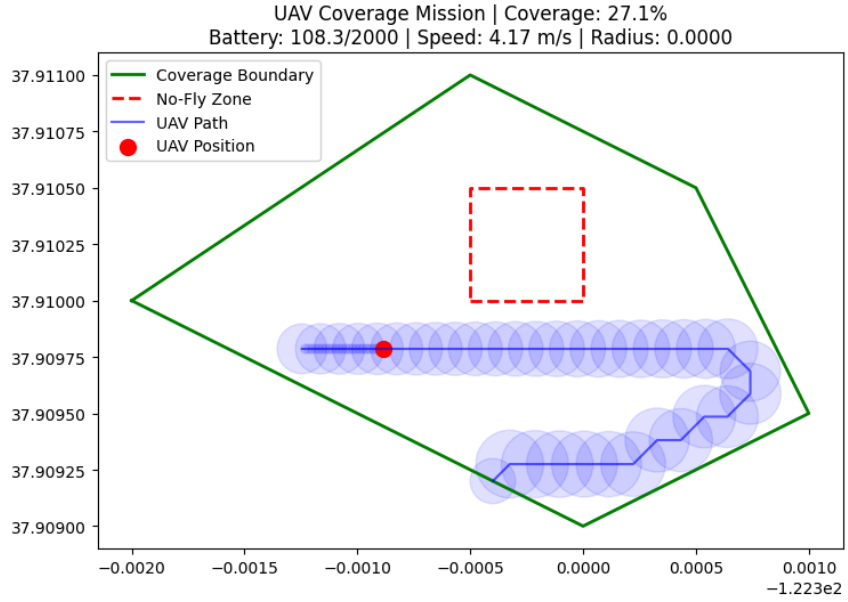


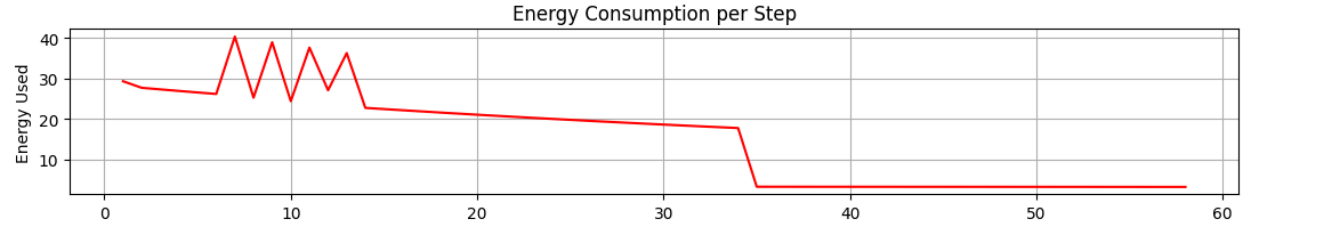

```
Step: Moved from (np.float64(-122.30091544097075), np.float64(37.90978598322125)) to (np.float64(-122.30090044097075), np.float64(37
Energy used: 3.16, remaining: 111.46
Coverage: 27.1%
Coverage radius: 0.0000
```

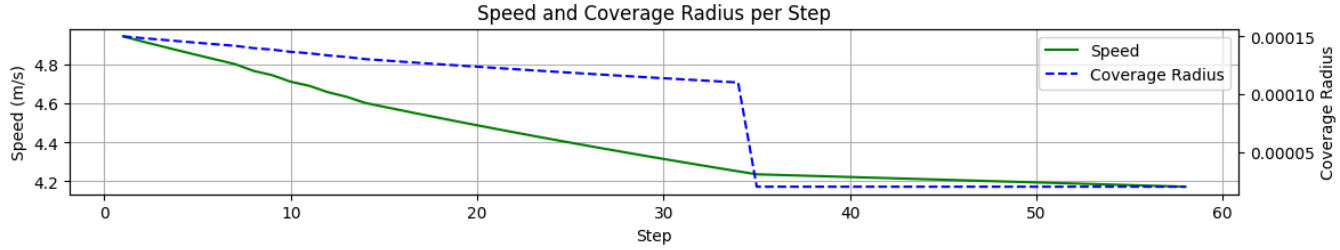




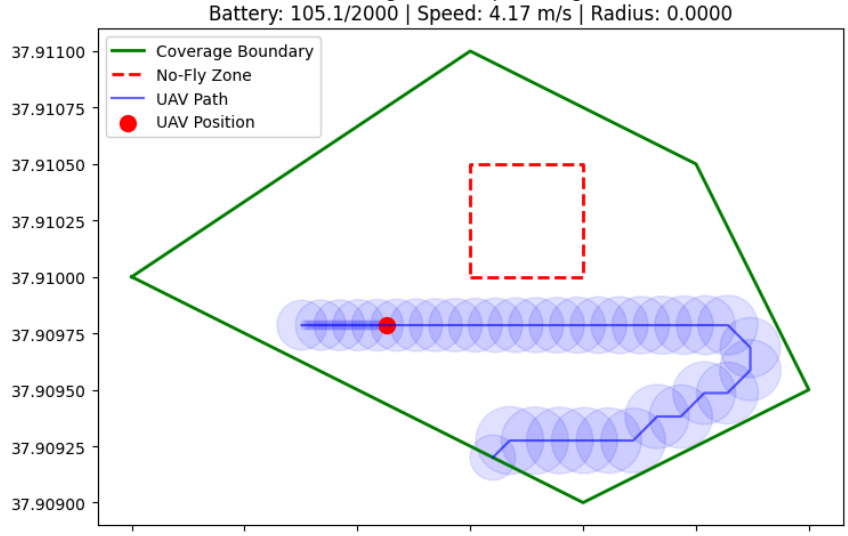

```
Step: Moved from (np.float64(-122.30090044097075), np.float64(37.90978598322125)) to (np.float64(-122.30088544097075), np.float64(37
Energy used: 3.16, remaining: 108.30
Coverage: 27.1%
Coverage radius: 0.0000
```

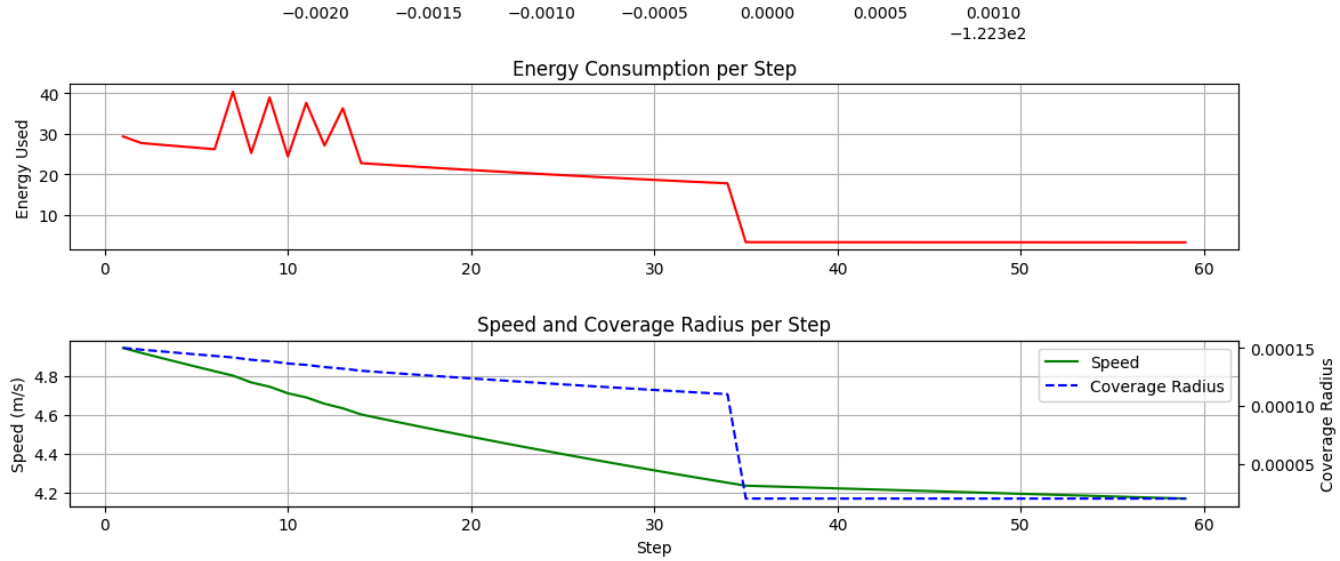




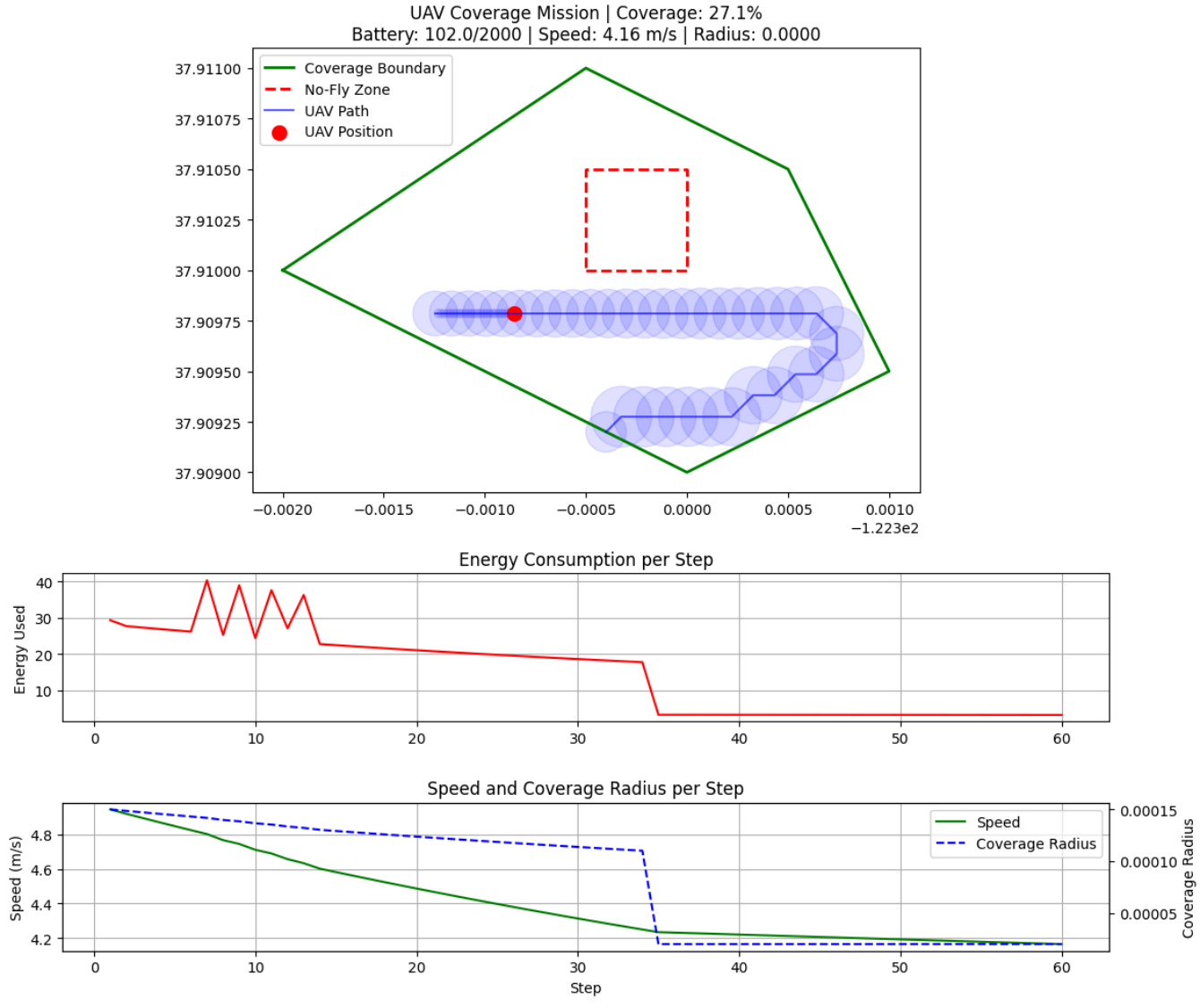



```
Step: Moved from (np.float64(-122.30088544097075), np.float64(37.90978598322125)) to (np.float64(-122.30087044097074), np.float64(37
Energy used: 3.16, remaining: 105.14
Coverage: 27.1%
Coverage radius: 0.0000
```

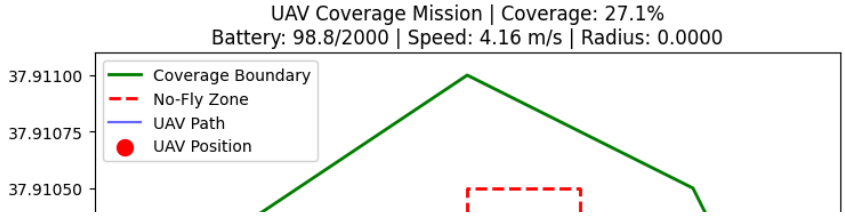


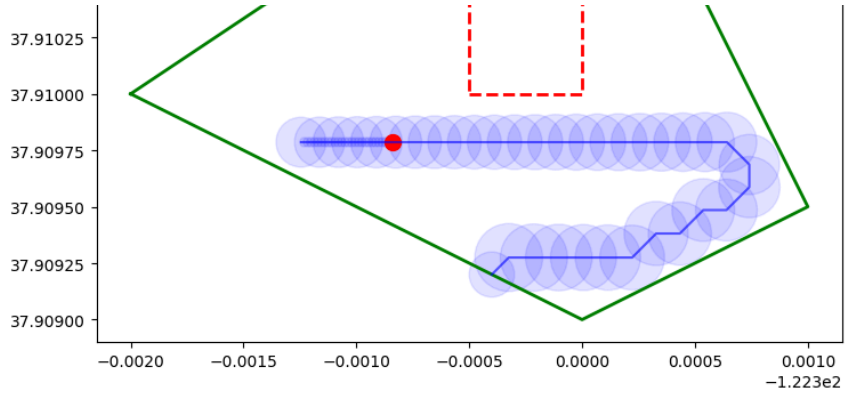


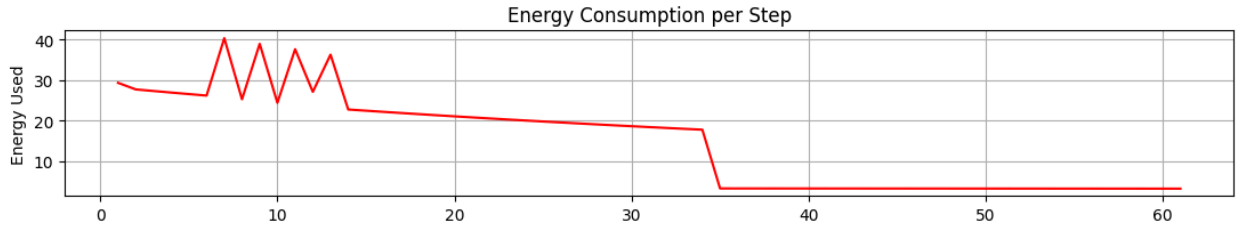

```
Step: Moved from (np.float64(-122.30087044097074), np.float64(37.90978598322125)) to (np.float64(-122.30085544097074), np.float64(37
Energy used: 3.15, remaining: 101.99
Coverage: 27.1%
Coverage radius: 0.0000
```

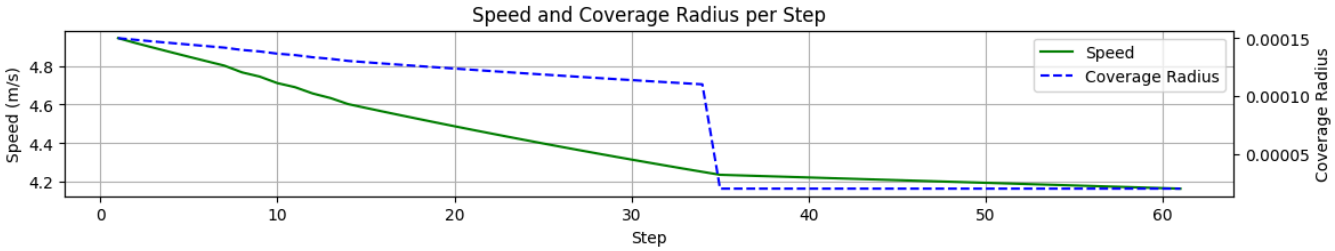



```
Step: Moved from (np.float64(-122.30085544097074), np.float64(37.90978598322125)) to (np.float64(-122.30084044097073), np.float64(37
Energy used: 3.15, remaining: 98.84
Coverage: 27.1%
Coverage radius: 0.0000
```

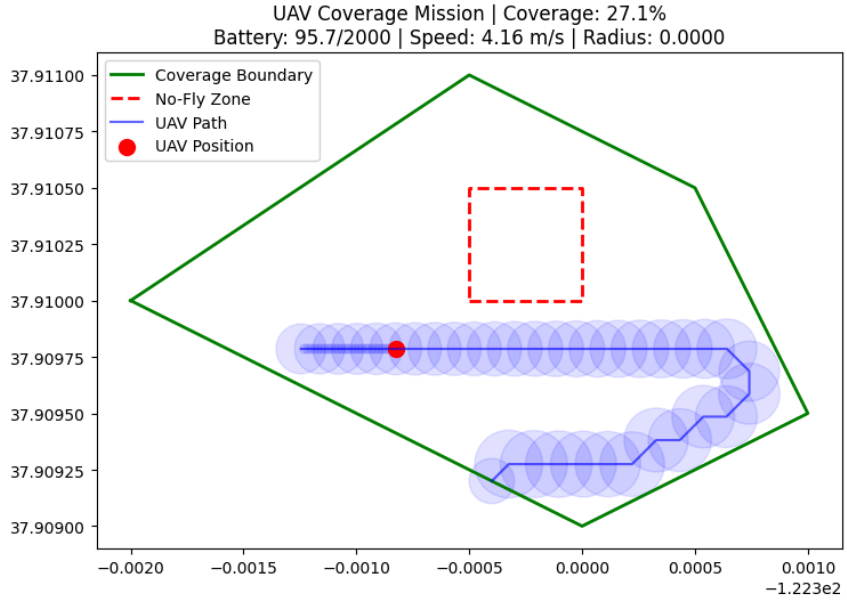


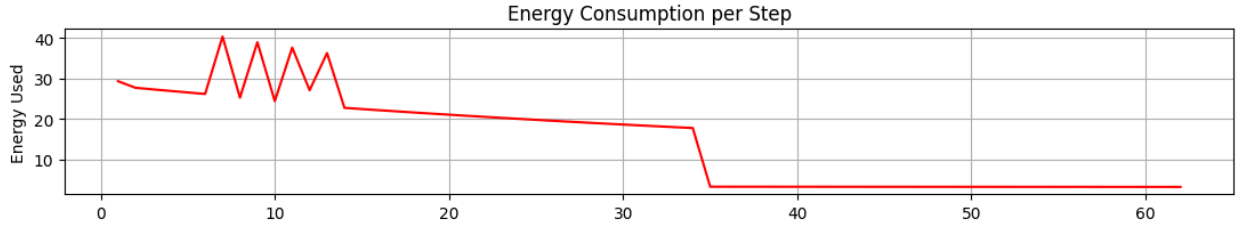


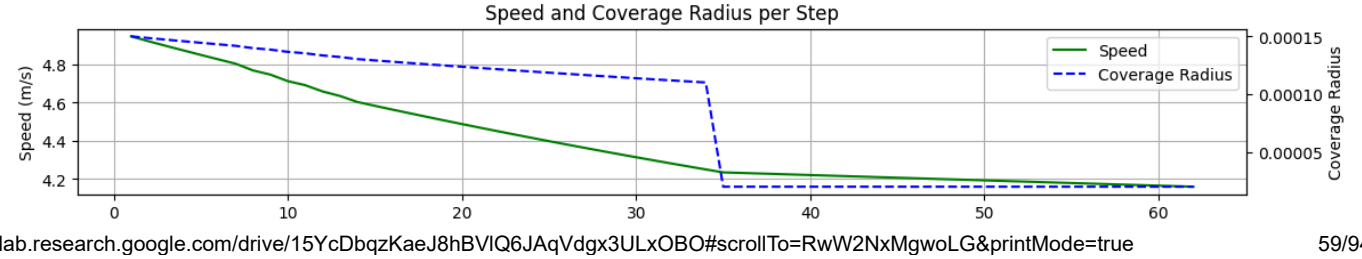





```
Step: Moved from (np.float64(-122.30084044097073), np.float64(37.90978598322125)) to (np.float64(-122.30082544097073), np.float64(37
Energy used: 3.15, remaining: 95.69
Coverage: 27.1%
Coverage radius: 0.0000
```

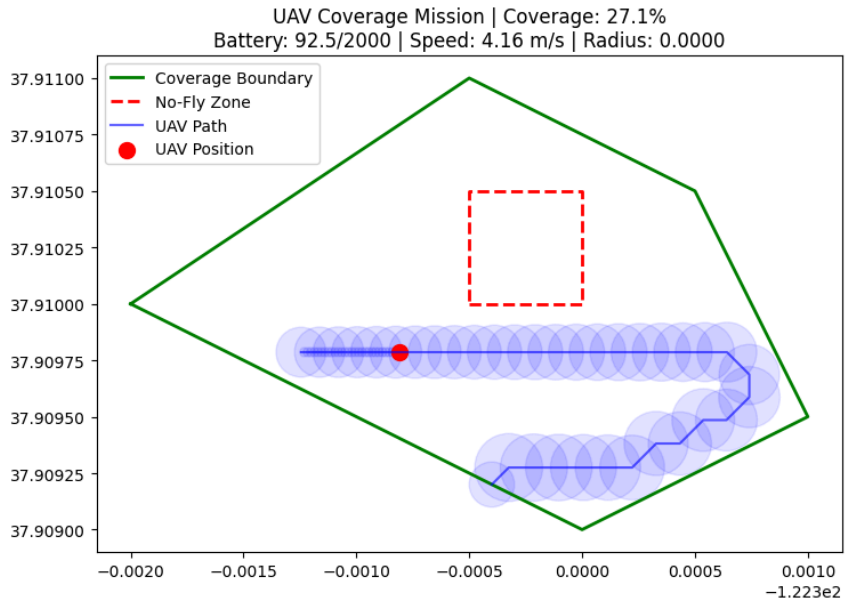


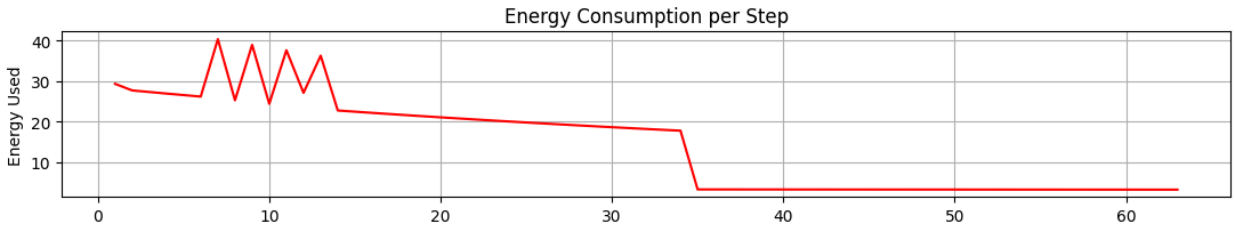


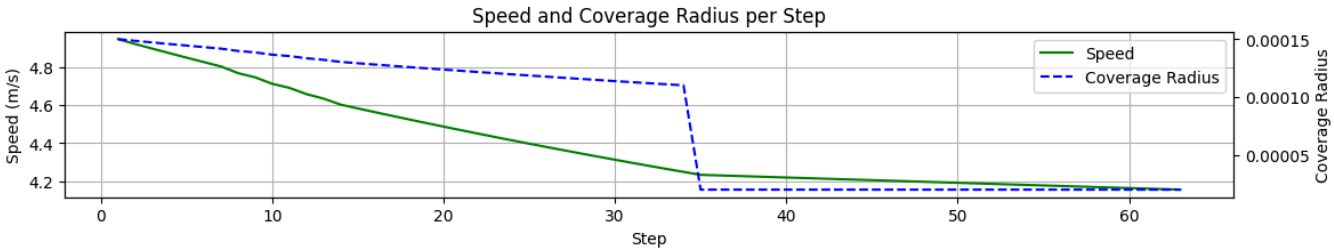


Step

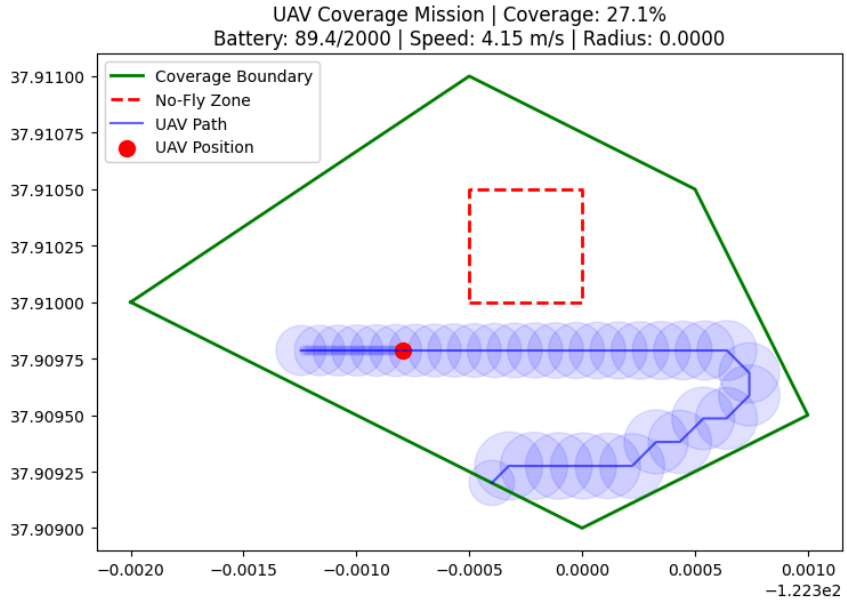
```
Step: Moved from (np.float64(-122.30082544097073), np.float64(37.90978598322125)) to (np.float64(-122.30081044097072), np.float64(37
Energy used: 3.15, remaining: 92.54
Coverage: 27.1%
Coverage radius: 0.0000
```

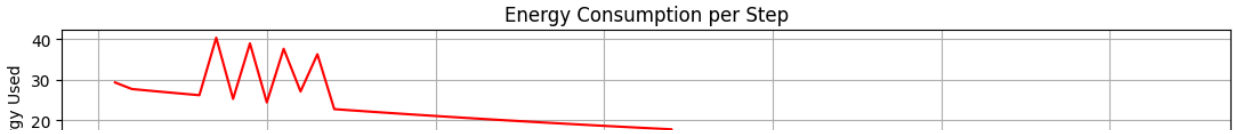


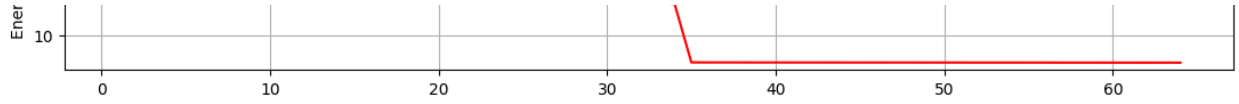


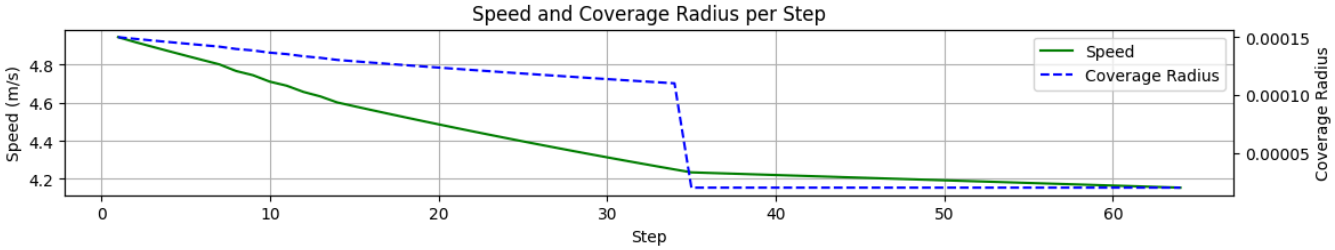



```
Step: Moved from (np.float64(-122.30081044097072), np.float64(37.90978598322125)) to (np.float64(-122.30079544097072), np.float64(37
Energy used: 3.15, remaining: 89.39
Coverage: 27.1%
Coverage radius: 0.0000
```

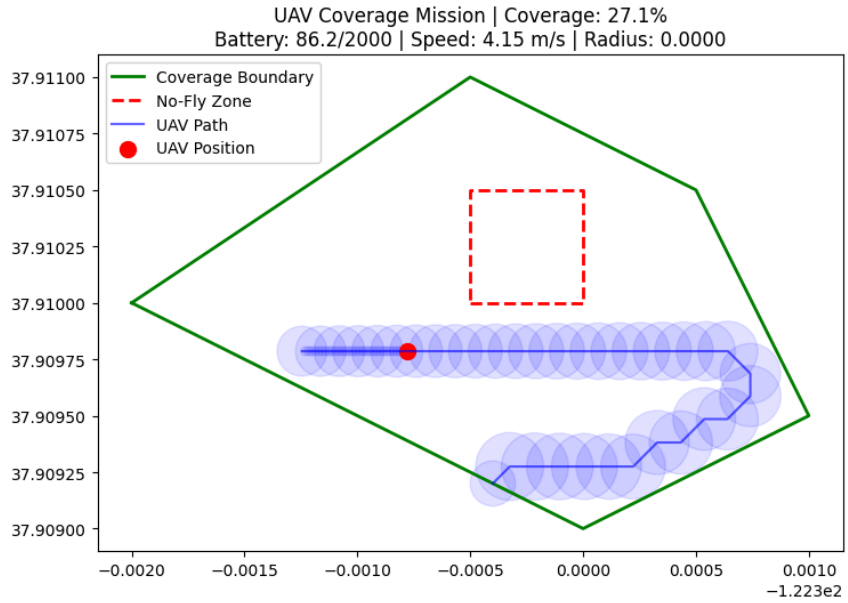


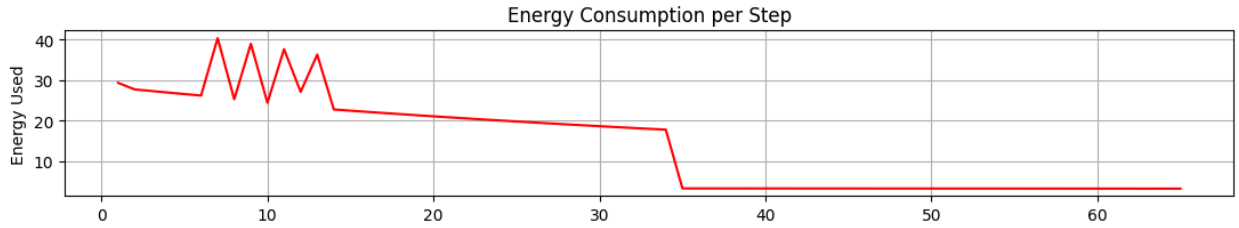


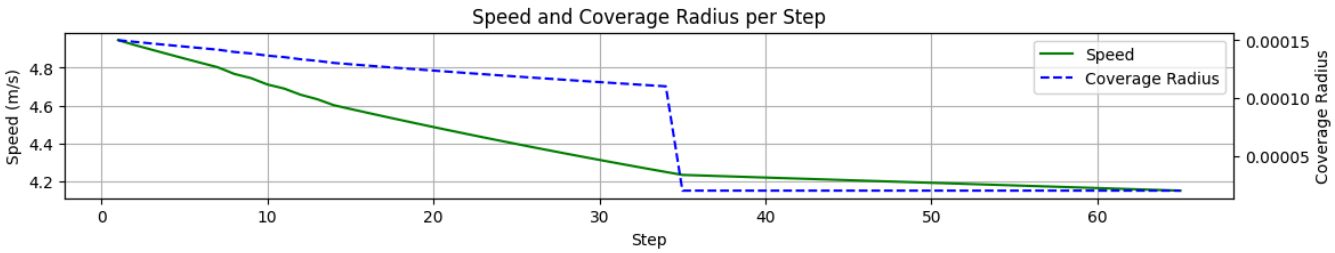




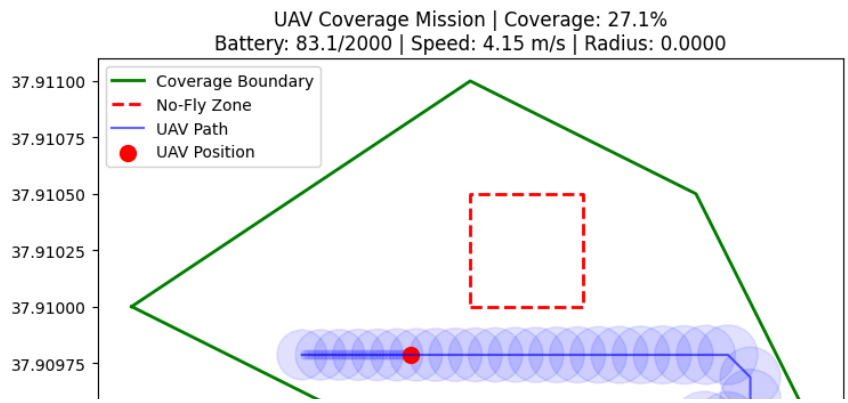

```
Step: Moved from (np.float64(-122.30079544097072), np.float64(37.90978598322125)) to (np.float64(-122.30078044097071), np.float64(37
Energy used: 3.14, remaining: 86.25
Coverage: 27.1%
Coverage radius: 0.0000
```

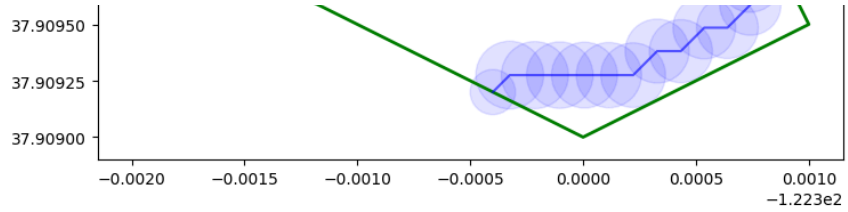


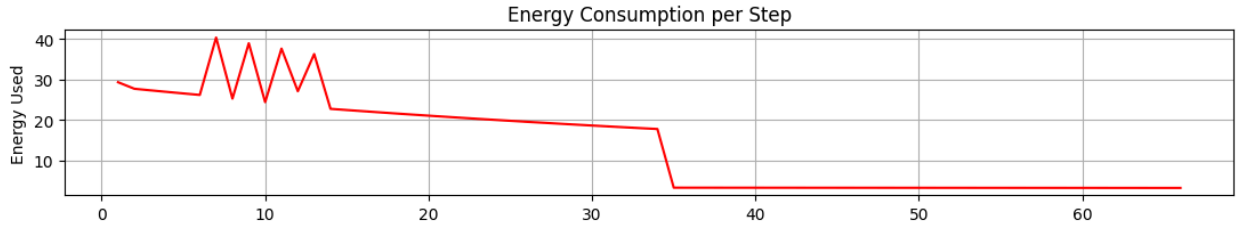


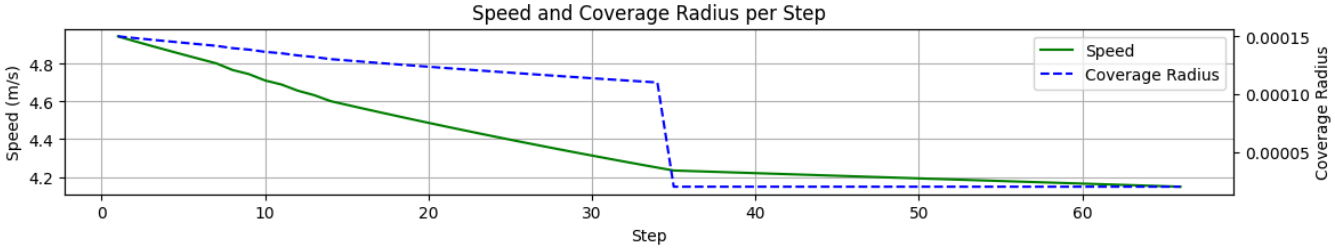



```
Step: Moved from (np.float64(-122.30078044097071), np.float64(37.90978598322125)) to (np.float64(-122.30076544097071), np.float64(37
Energy used: 3.14, remaining: 83.11
Coverage: 27.1%
Coverage radius: 0.0000
```

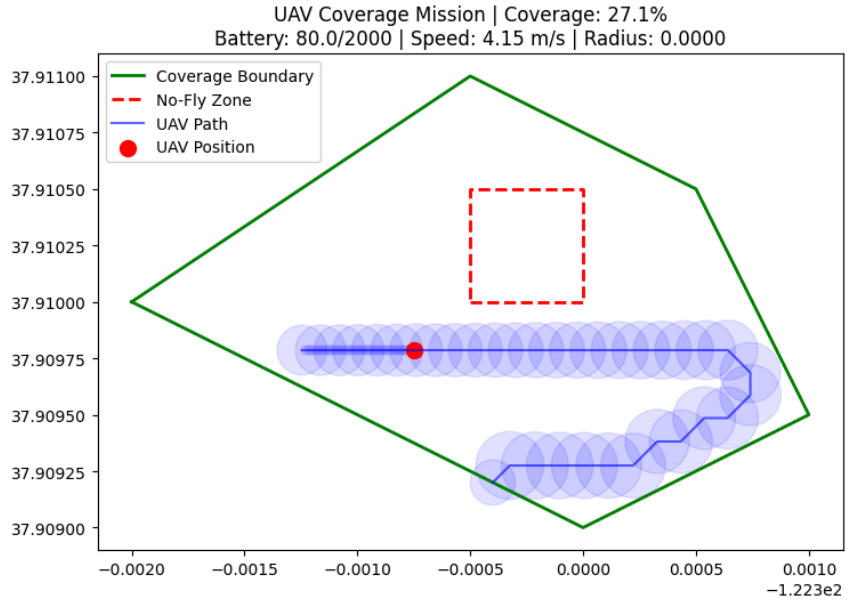


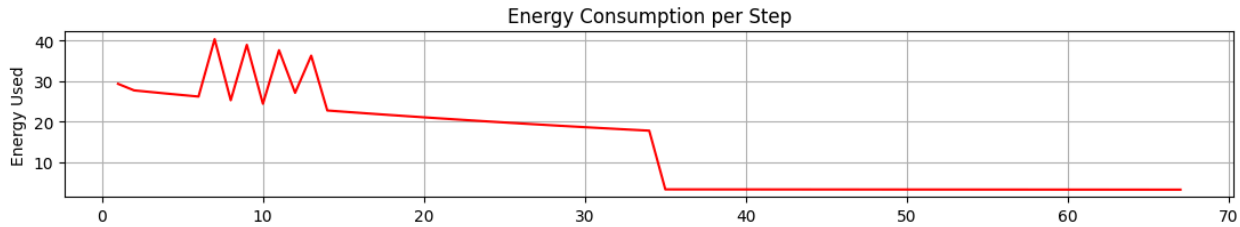


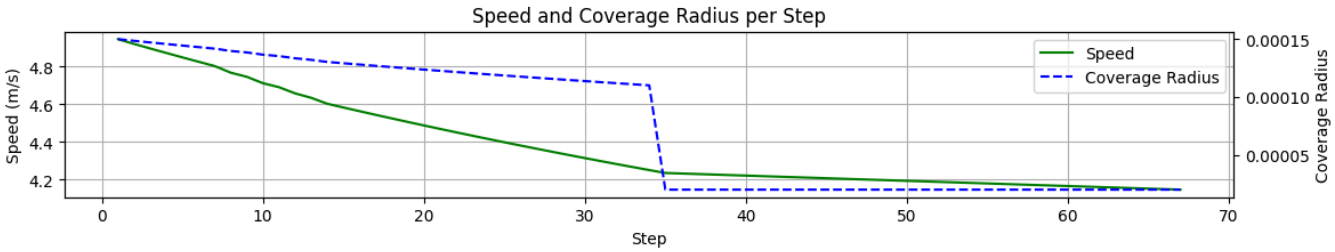




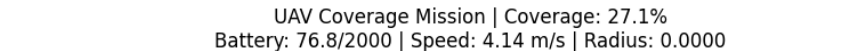

```
Step: Moved from (np.float64(-122.30076544097071), np.float64(37.90978598322125)) to (np.float64(-122.3007504409707), np.float64(37
Energy used: 3.14, remaining: 79.97
Coverage: 27.1%
Coverage radius: 0.0000
```

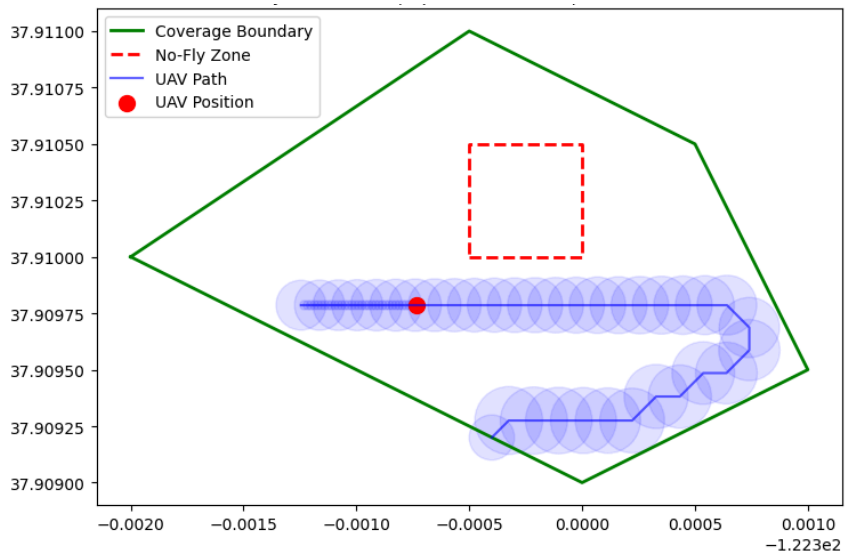


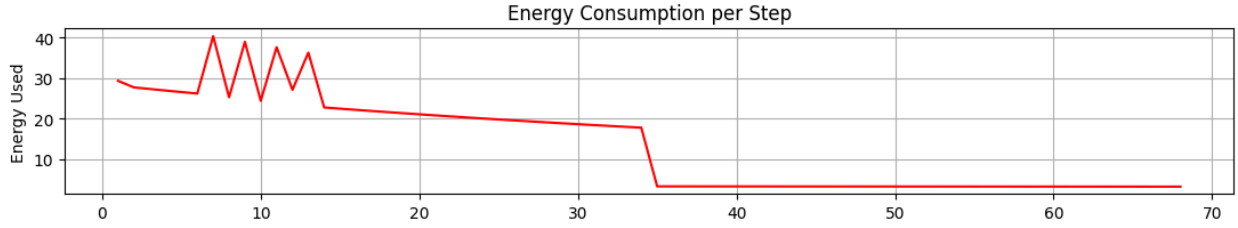


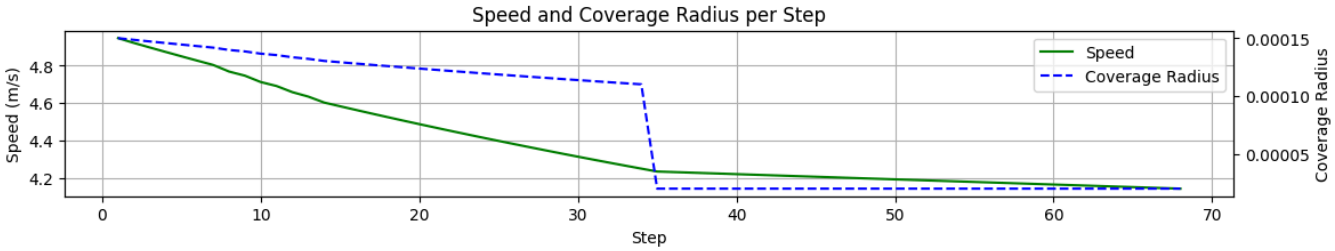



```
Step: Moved from (np.float64(-122.3007504409707), np.float64(37.90978598322125)) to (np.float64(-122.3007354409707), np.float64(37.9
Energy used: 3.14, remaining: 76.83
Coverage: 27.1%
Coverage radius: 0.0000
```

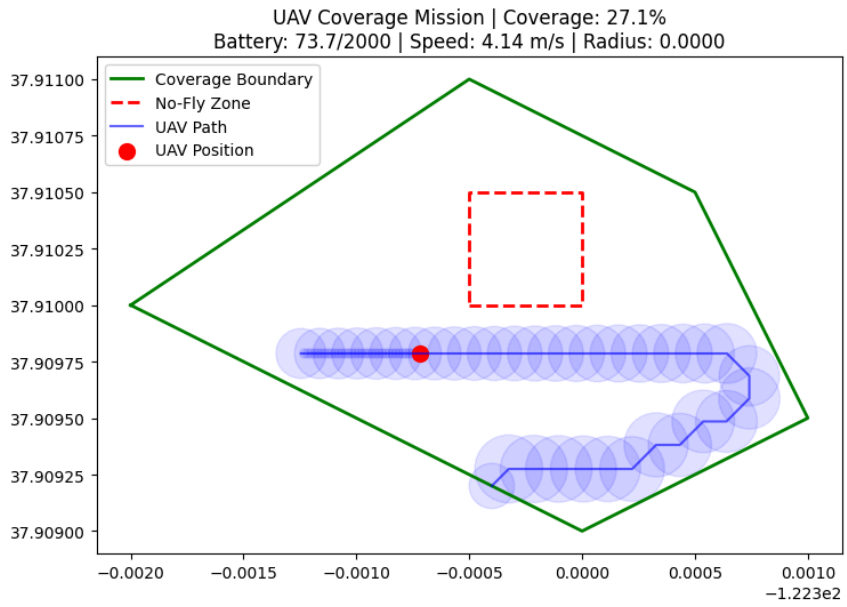


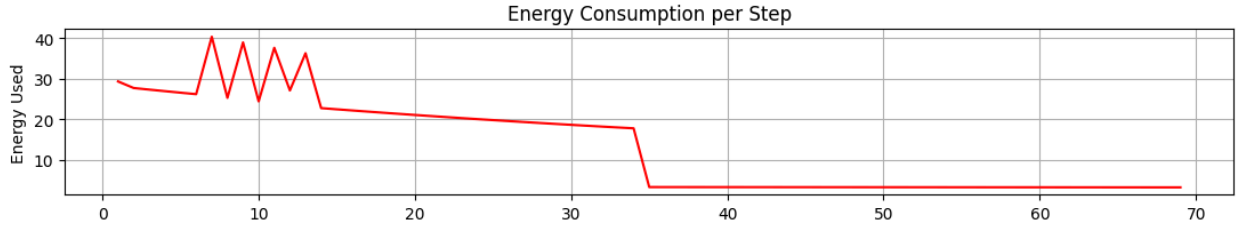


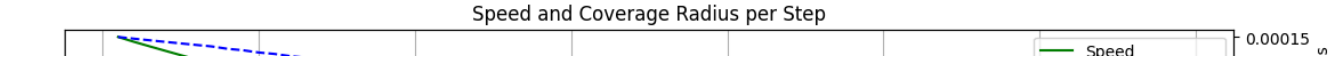


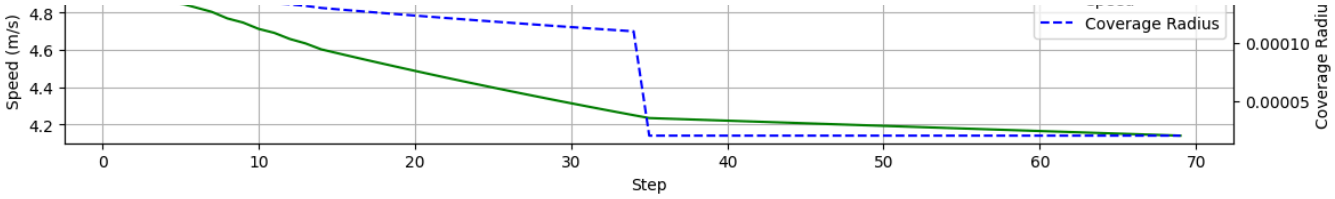



```
Step: Moved from (np.float64(-122.3007354409707), np.float64(37.90978598322125)) to (np.float64(-122.3007204409707), np.float64(37.9
Energy used: 3.14, remaining: 73.69
Coverage: 27.1%
Coverage radius: 0.0000
```

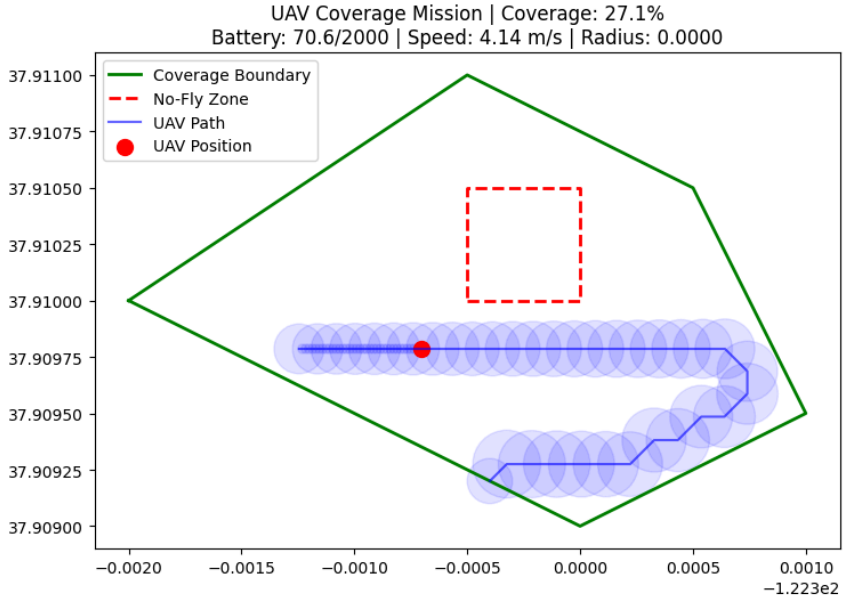


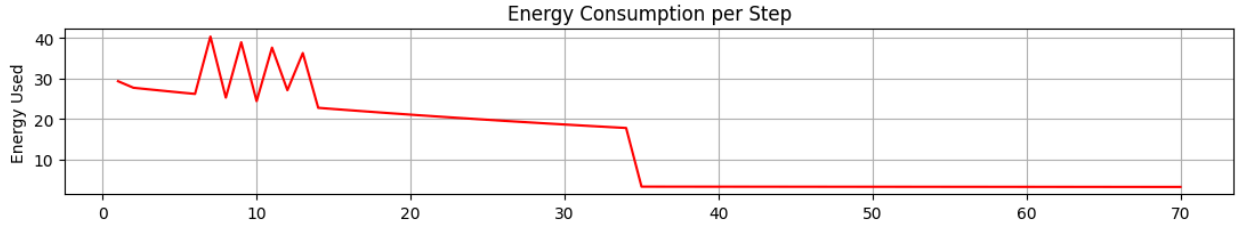


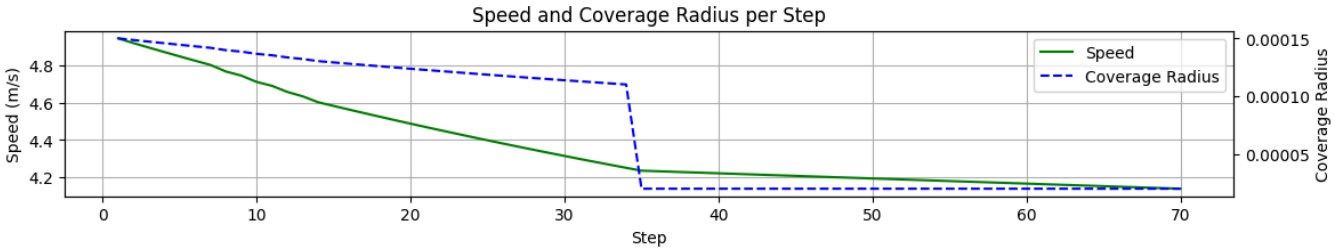




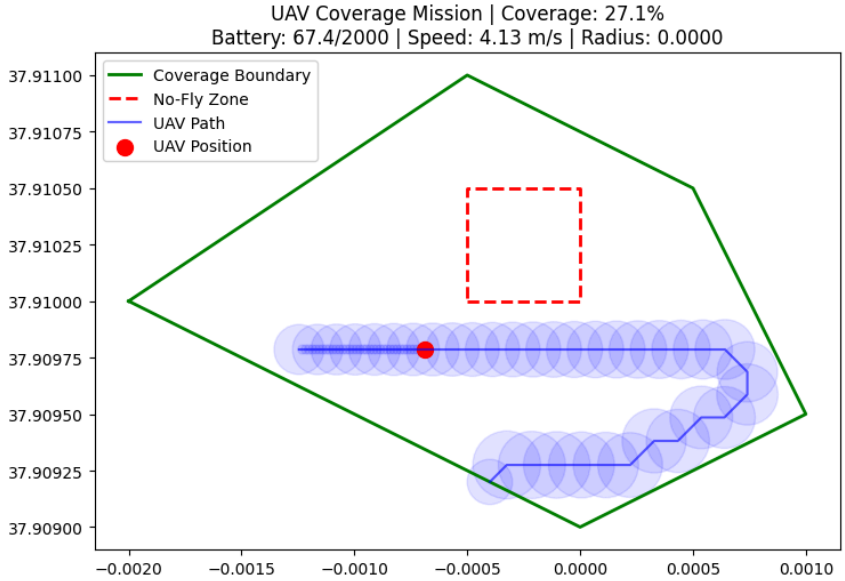

```
Step: Moved from (np.float64(-122.3007204409707), np.float64(37.90978598322125)) to (np.float64(-122.30070544097069), np.float64(37
Energy used: 3.13, remaining: 70.56
Coverage: 27.1%
Coverage radius: 0.0000
```

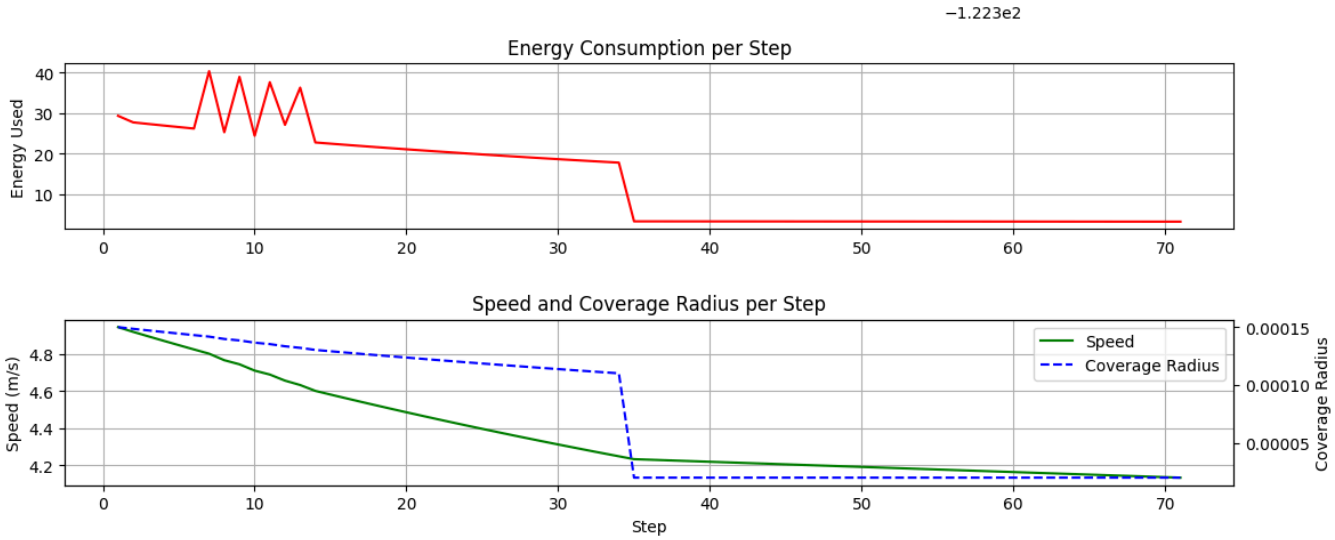




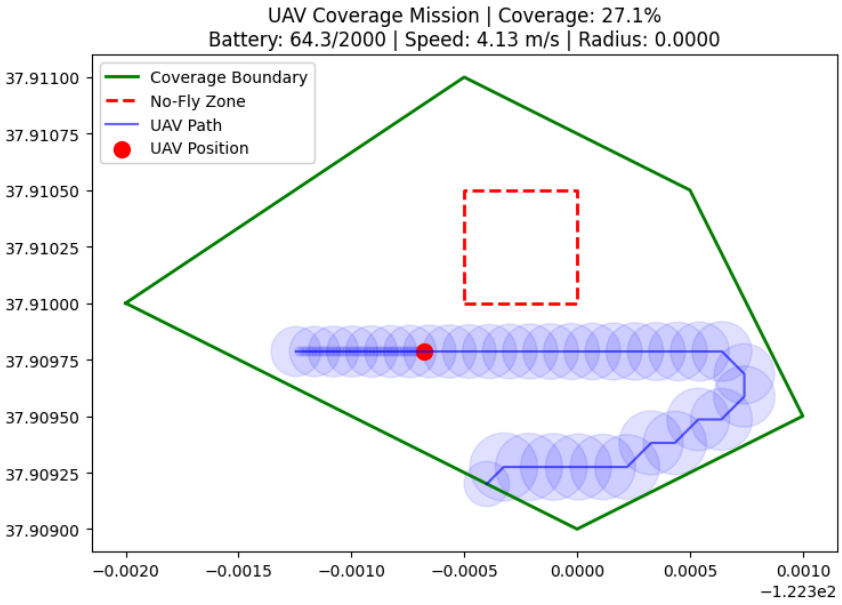


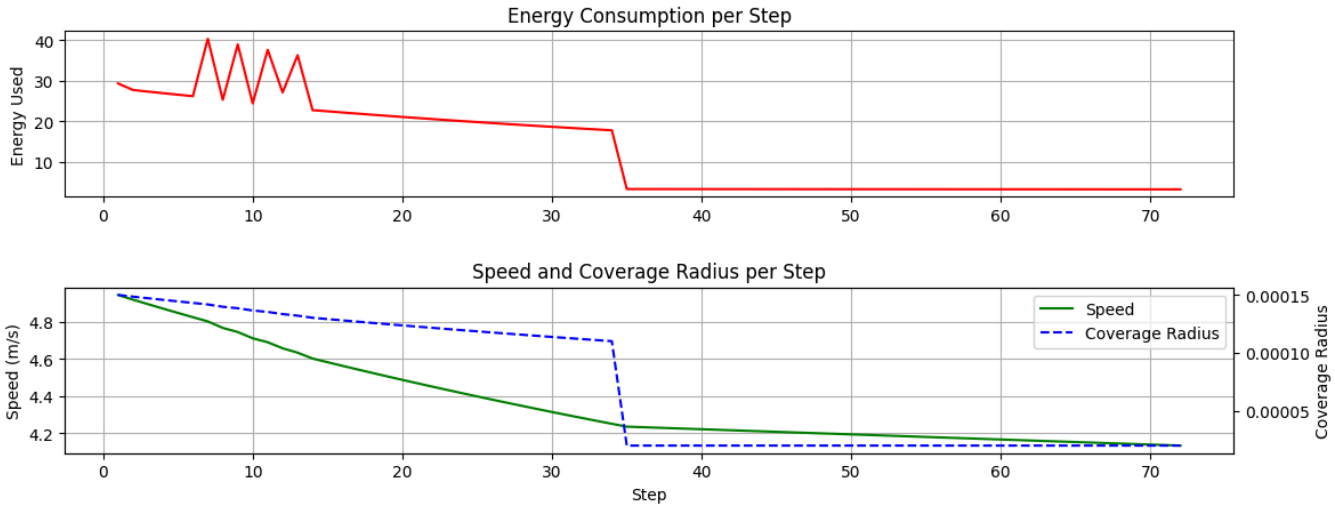

```
Step: Moved from (np.float64(-122.30070544097069), np.float64(37.90978598322125)) to (np.float64(-122.30069044097068), np.float64(37
Energy used: 3.13, remaining: 67.43
Coverage: 27.1%
Coverage radius: 0.0000
```



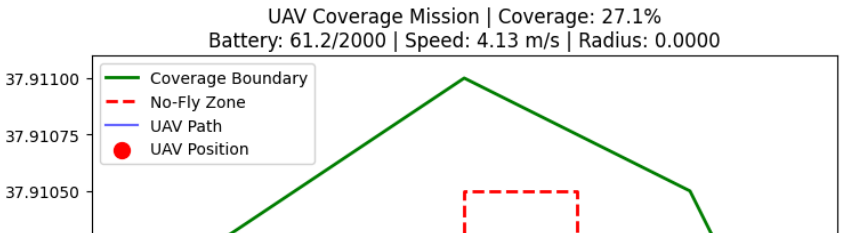


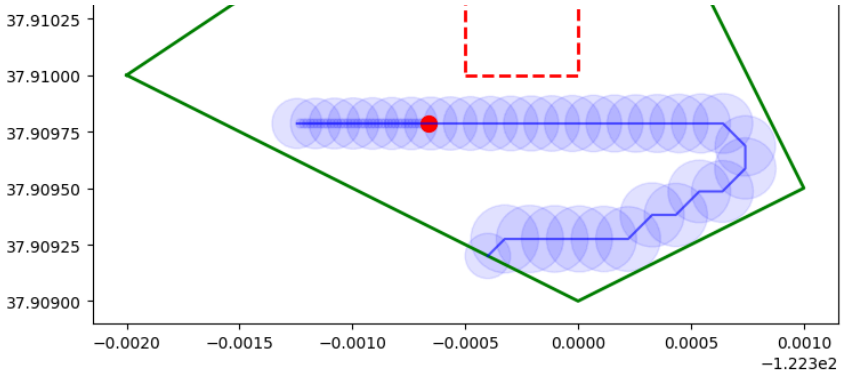

```
Step: Moved from (np.float64(-122.30069044097068), np.float64(37.90978598322125)) to (np.float64(-122.30067544097068), np.float64(37
Energy used: 3.13, remaining: 64.30
Coverage: 27.1%
Coverage radius: 0.0000
```

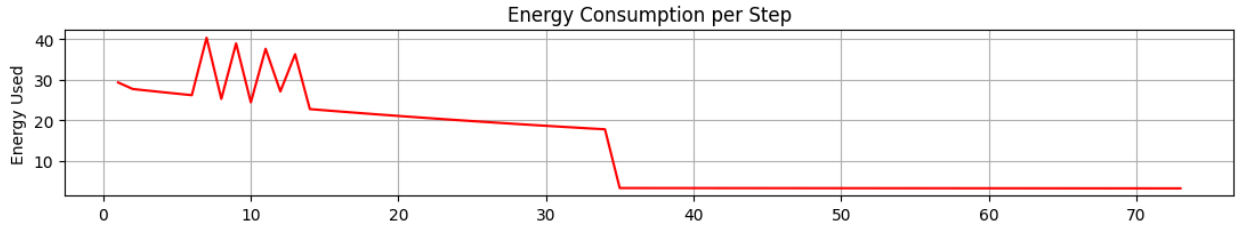


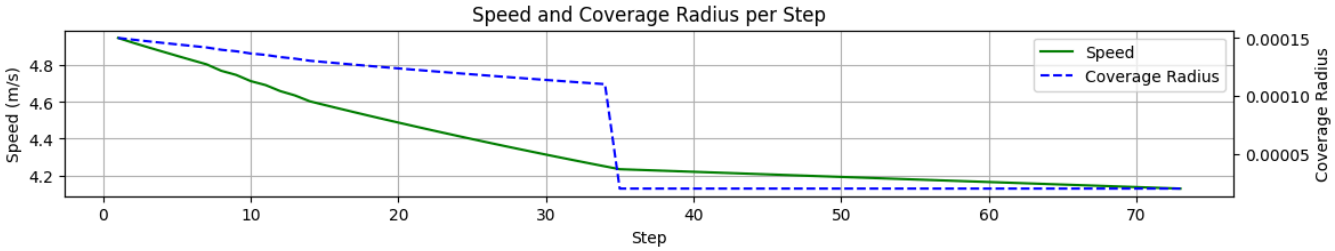



```
Step: Moved from (np.float64(-122.30067544097068), np.float64(37.90978598322125)) to (np.float64(-122.30066044097067), np.float64(37
Energy used: 3.13, remaining: 61.17
Coverage: 27.1%
Coverage radius: 0.0000
```

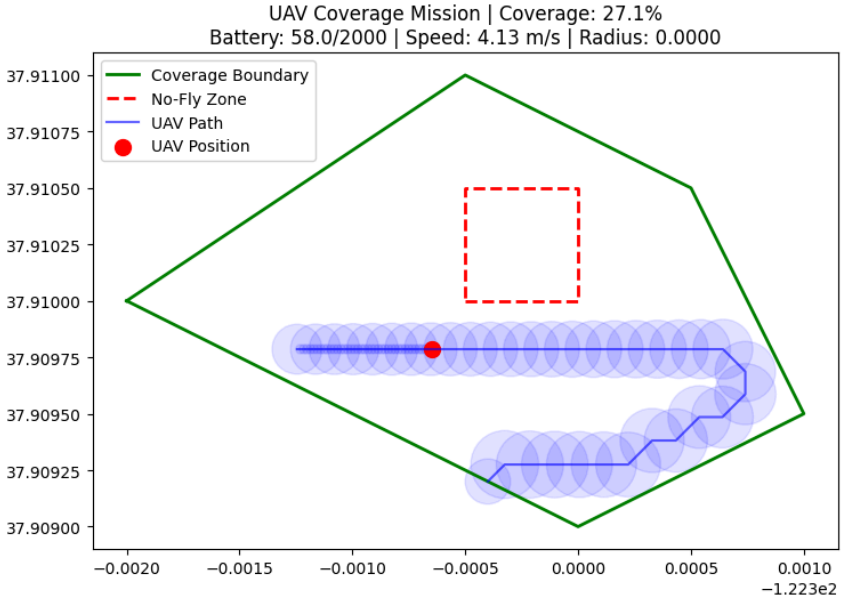


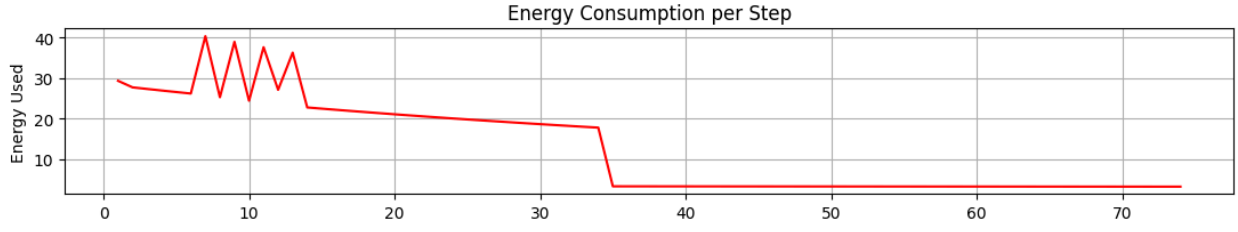


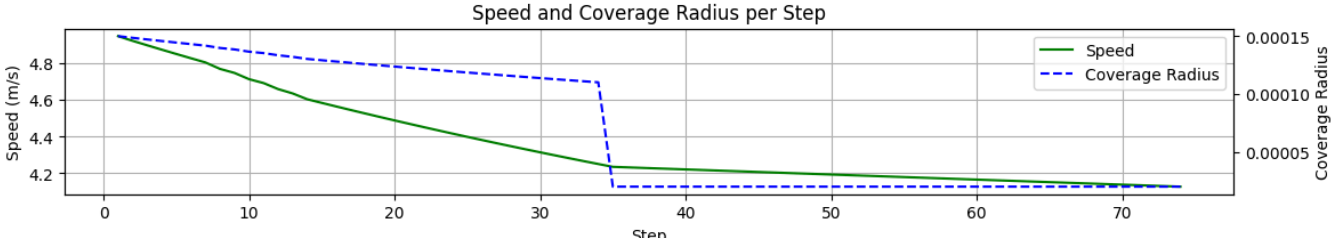




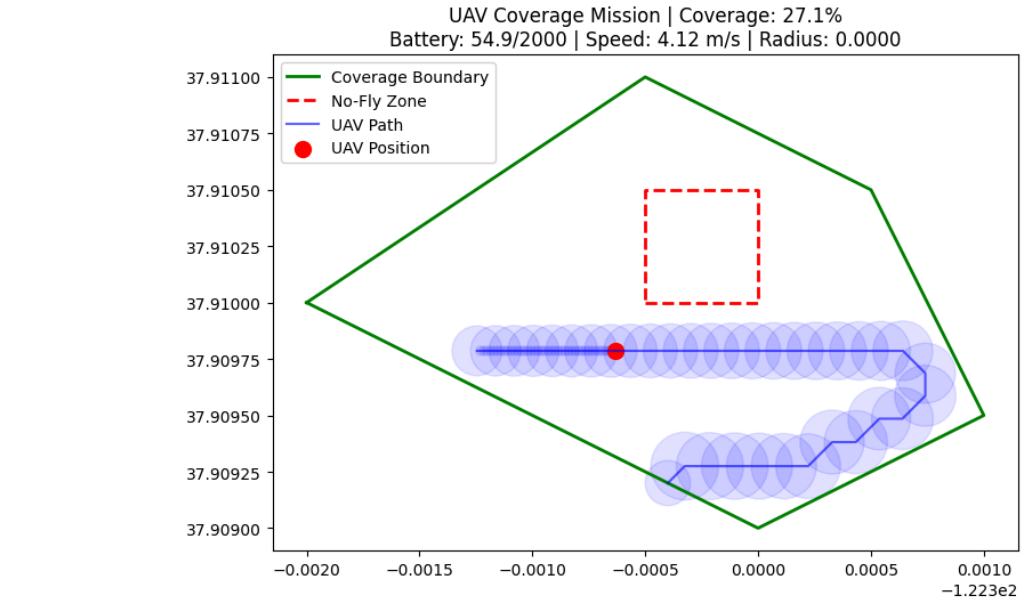

```
Step: Moved from (np.float64(-122.30066044097067), np.float64(37.90978598322125)) to (np.float64(-122.30064544097067), np.float64(37
Energy used: 3.13, remaining: 58.04
Coverage: 27.1%
Coverage radius: 0.0000
```

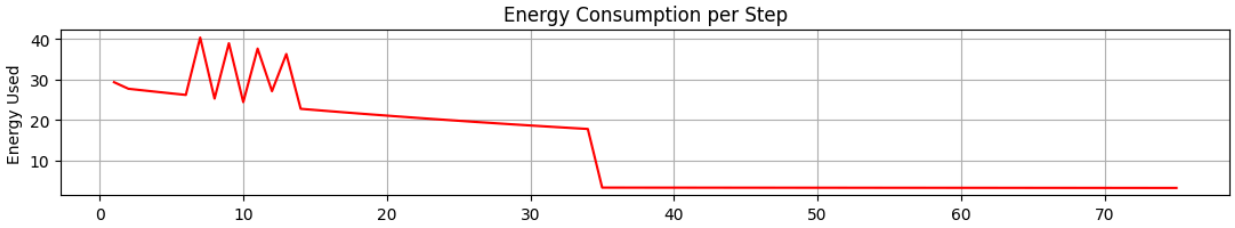


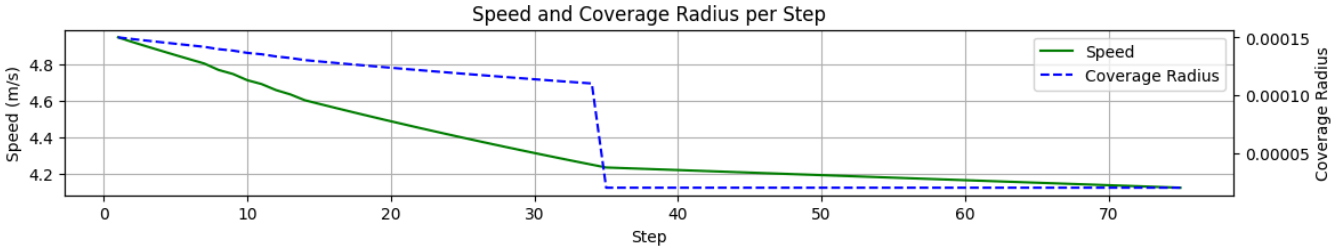




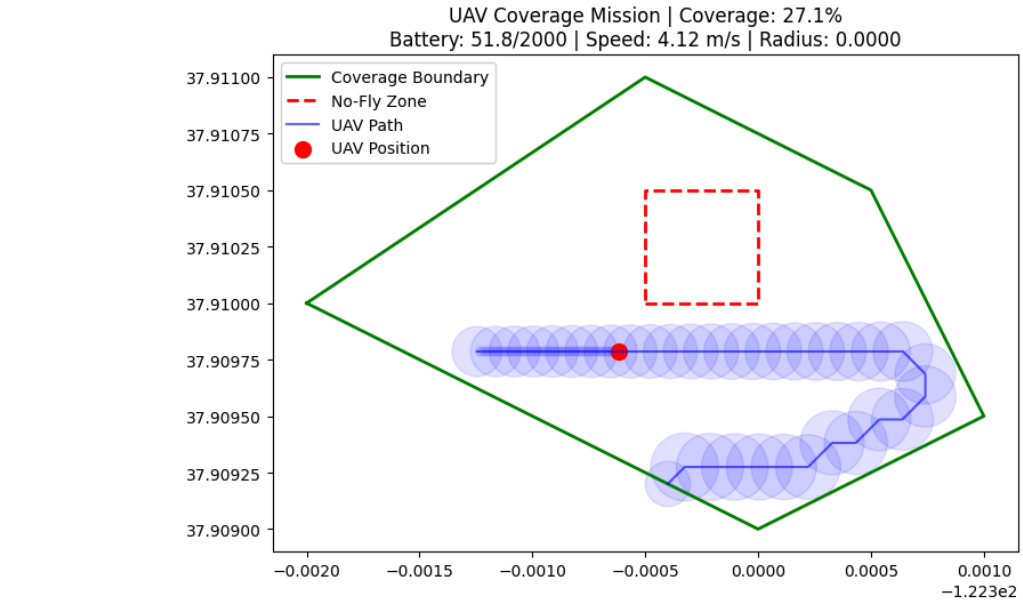
```
Step: Moved from (np.float64(-122.30064544097067), np.float64(37.90978598322125)) to (np.float64(-122.30063044097066), np.float64(37
Energy used: 3.12, remaining: 54.92
Coverage: 27.1%
Coverage radius: 0.0000
```

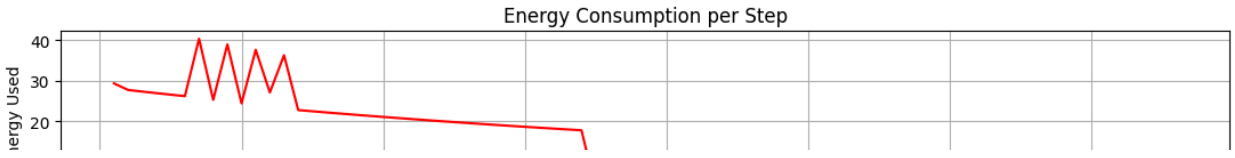


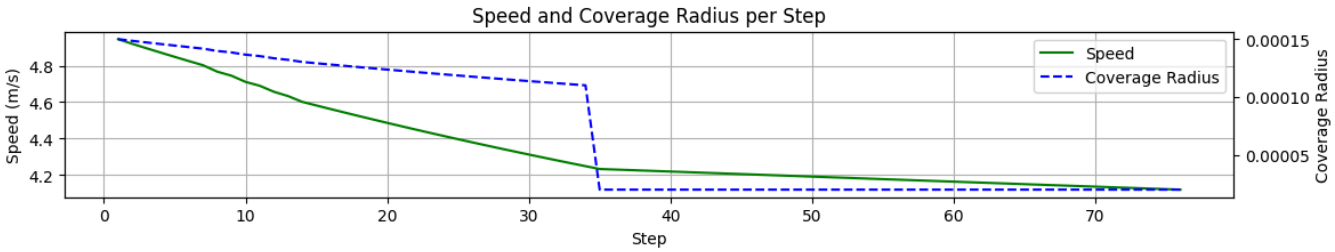




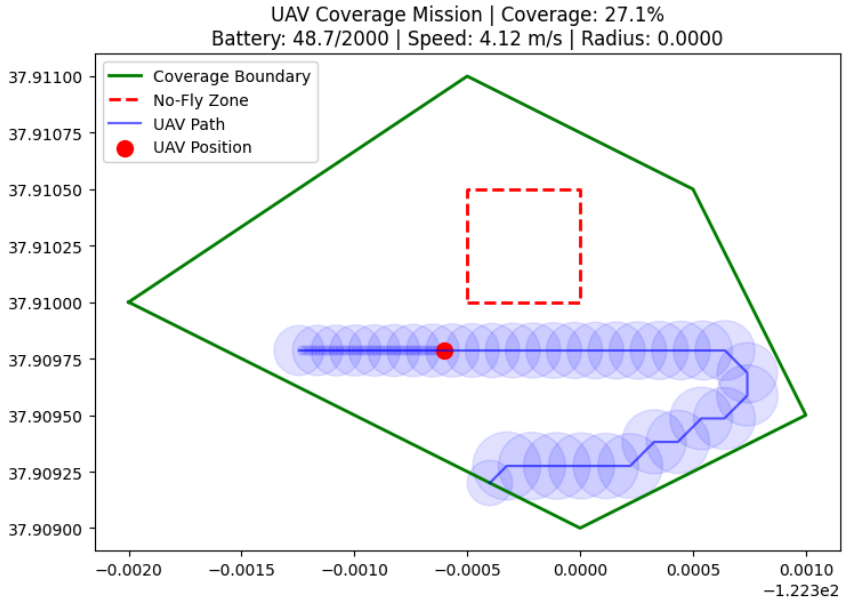

```
Step: Moved from (np.float64(-122.30063044097066), np.float64(37.90978598322125)) to (np.float64(-122.30061544097066), np.float64(37
Energy used: 3.12, remaining: 51.80
Coverage: 27.1%
Coverage radius: 0.0000
```

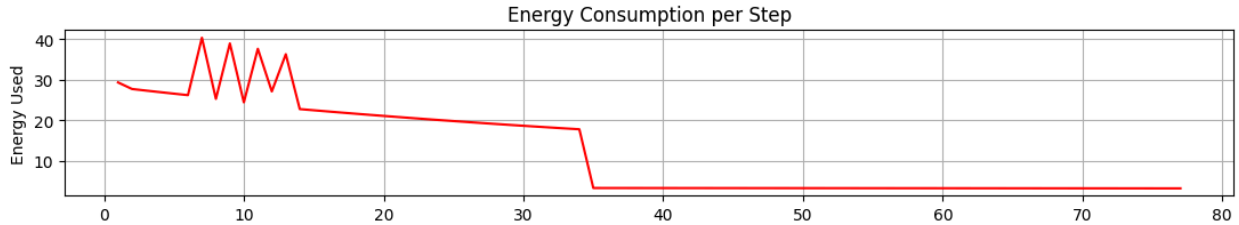


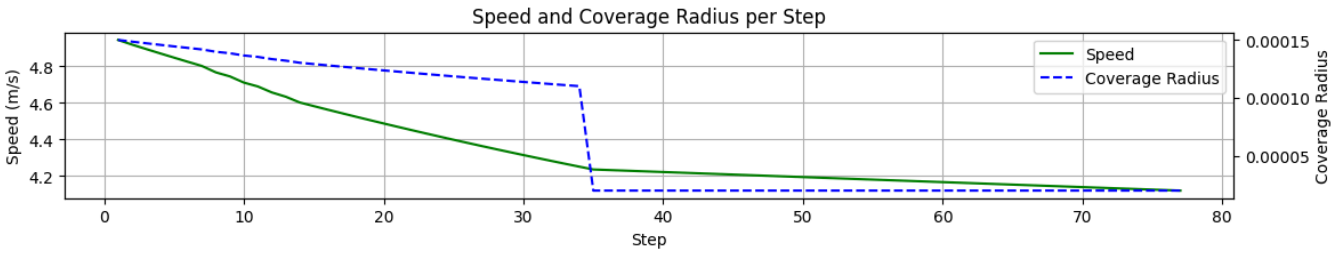




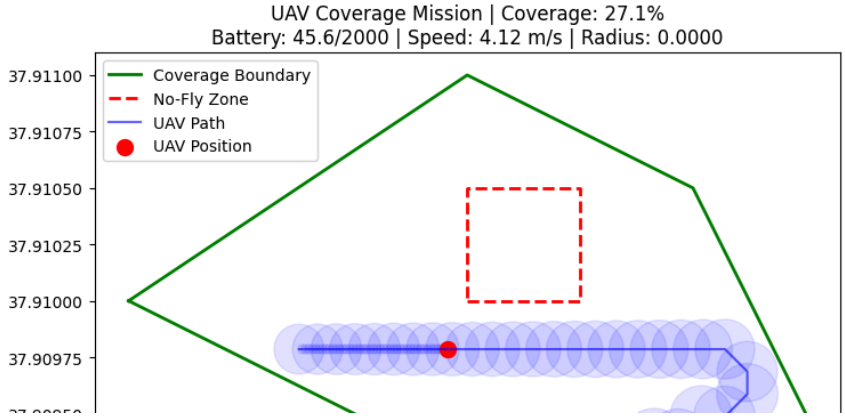

```
Step: Moved from (np.float64(-122.30061544097066), np.float64(37.90978598322125)) to (np.float64(-122.30060044097065), np.float64(37
Energy used: 3.12, remaining: 48.68
Coverage: 27.1%
Coverage radius: 0.0000
```

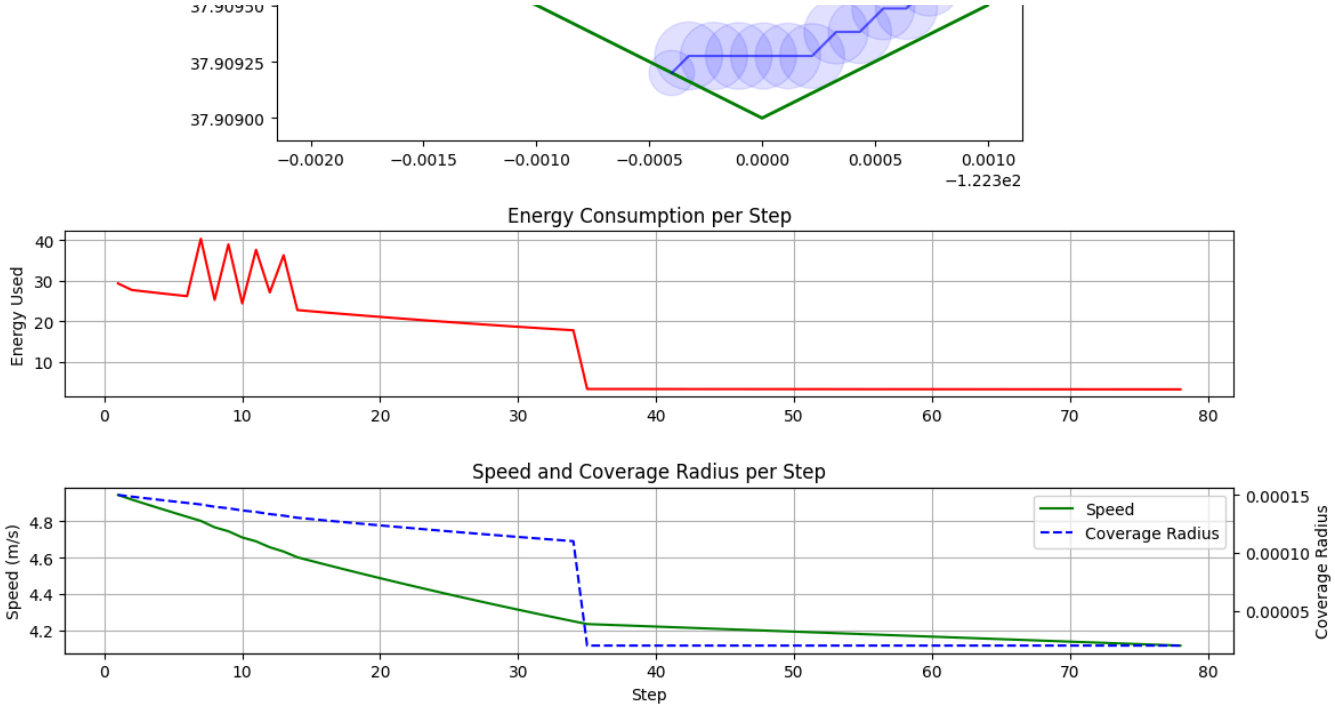




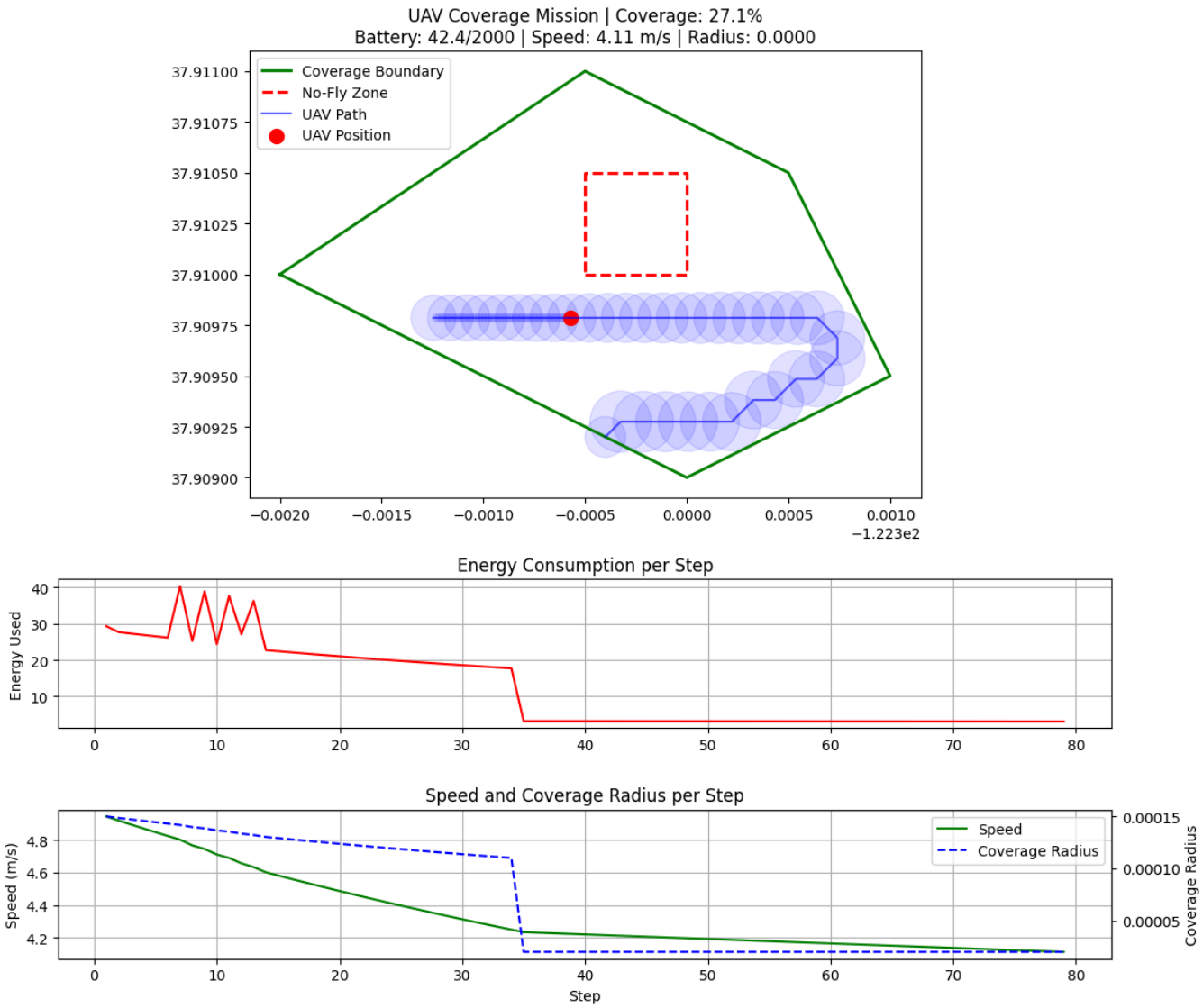



```
Step: Moved from (np.float64(-122.30060044097065), np.float64(37.90978598322125)) to (np.float64(-122.30058544097065), np.float64(37
Energy used: 3.12, remaining: 45.56
Coverage: 27.1%
Coverage radius: 0.0000
```

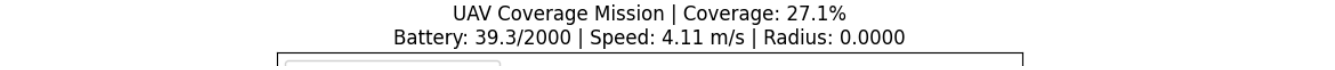


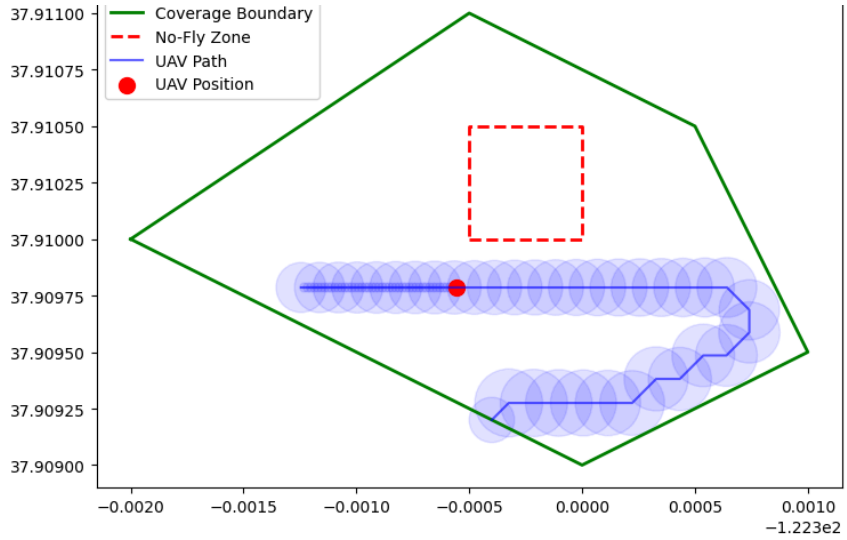


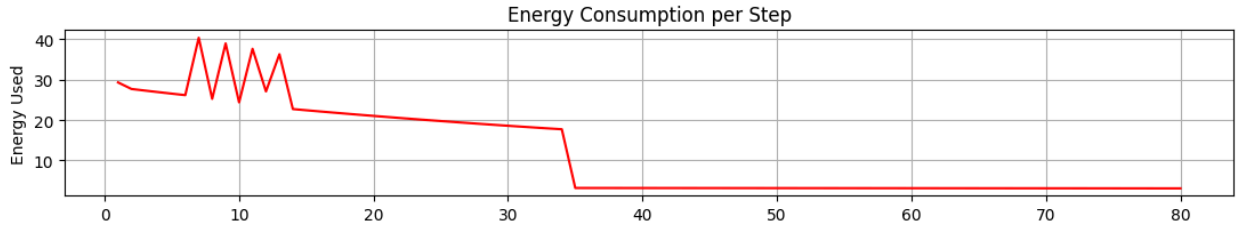

```
Step: Moved from (np.float64(-122.30058544097065), np.float64(37.90978598322125)) to (np.float64(-122.30057044097065), np.float64(37
Energy used: 3.12, remaining: 42.45
Coverage: 27.1%
Coverage radius: 0.0000
```

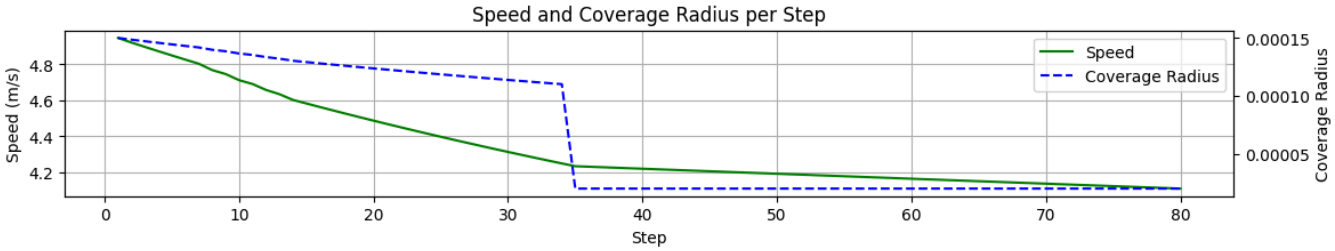



```
Step: Moved from (np.float64(-122.30057044097065), np.float64(37.90978598322125)) to (np.float64(-122.30055544097064), np.float64(37
Energy used: 3.11, remaining: 39.33
Coverage: 27.1%
Coverage radius: 0.0000
```

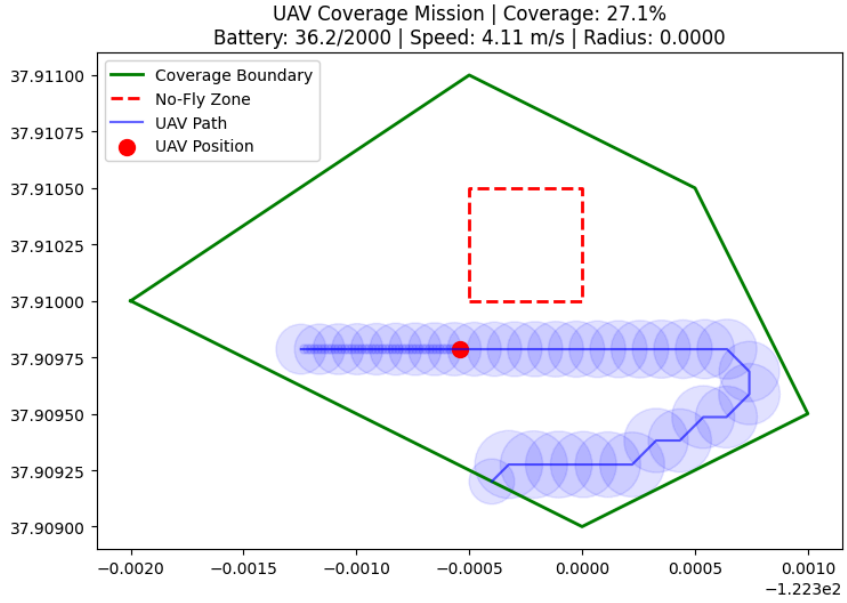


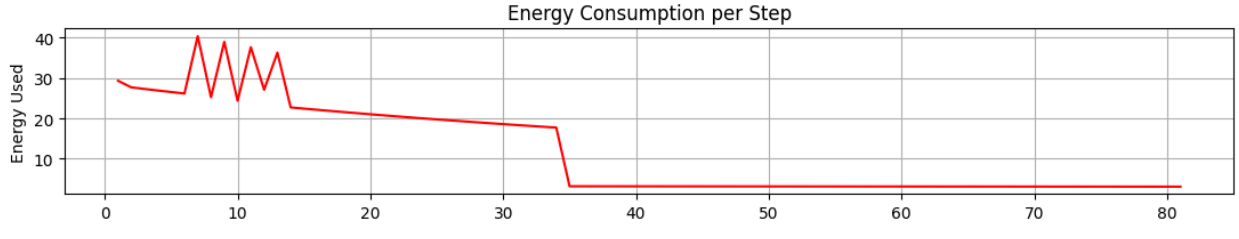


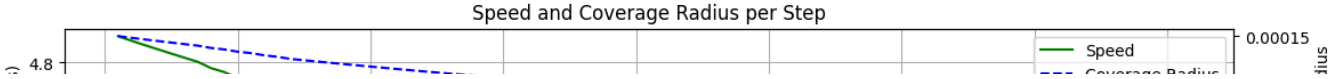


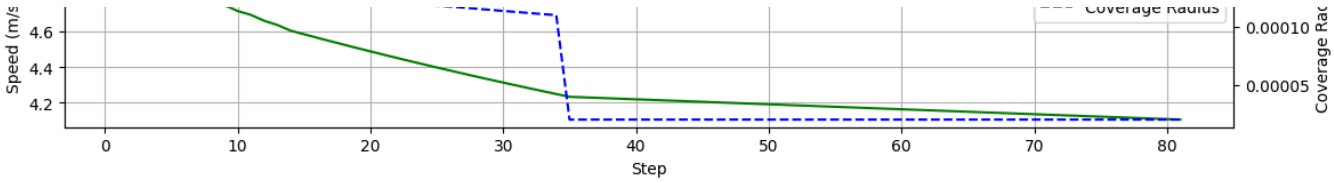



```
Step: Moved from (np.float64(-122.30055544097064), np.float64(37.90978598322125)) to (np.float64(-122.30054044097064), np.float64(37
Energy used: 3.11, remaining: 36.22
Coverage: 27.1%
Coverage radius: 0.0000
```

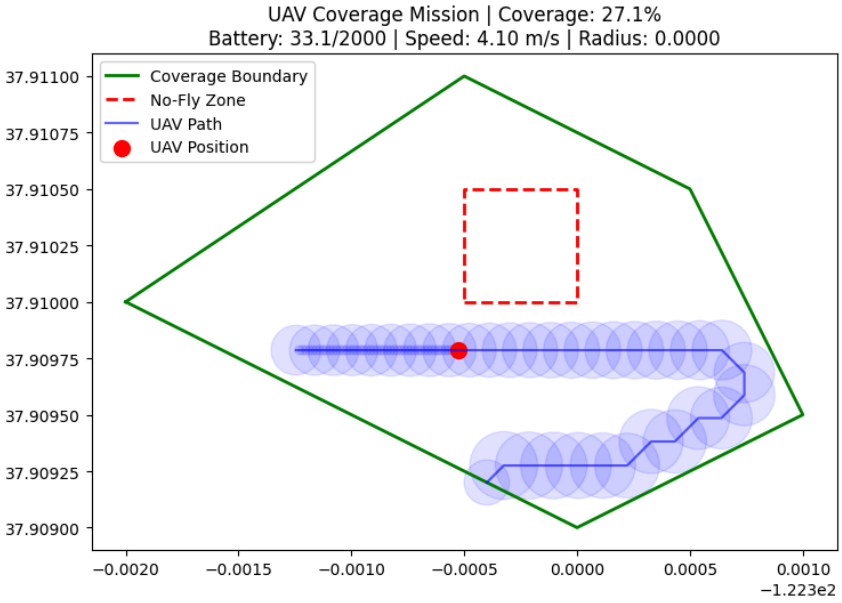


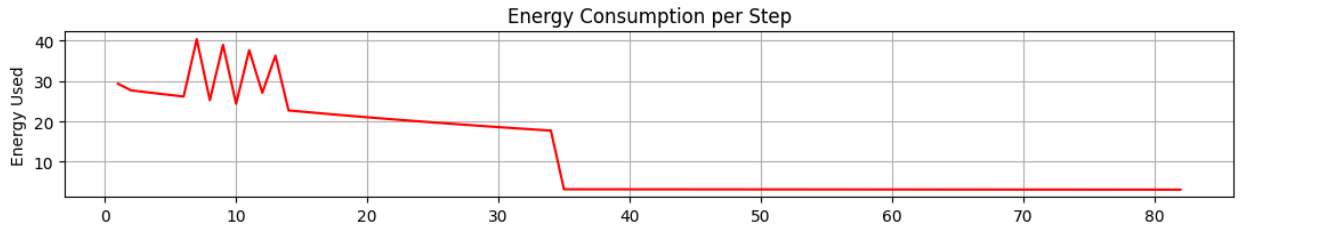


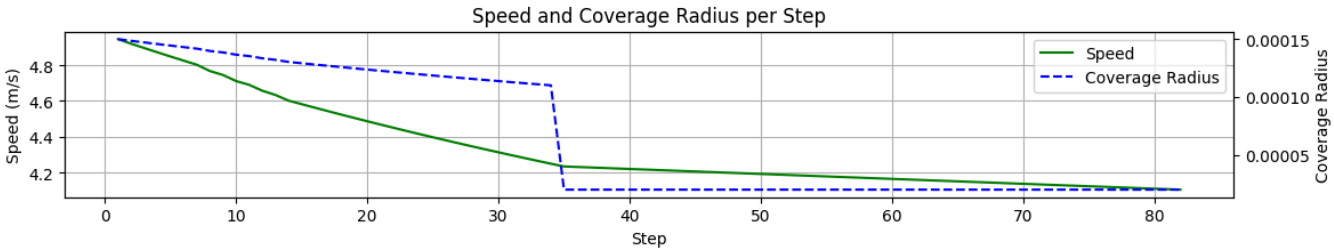




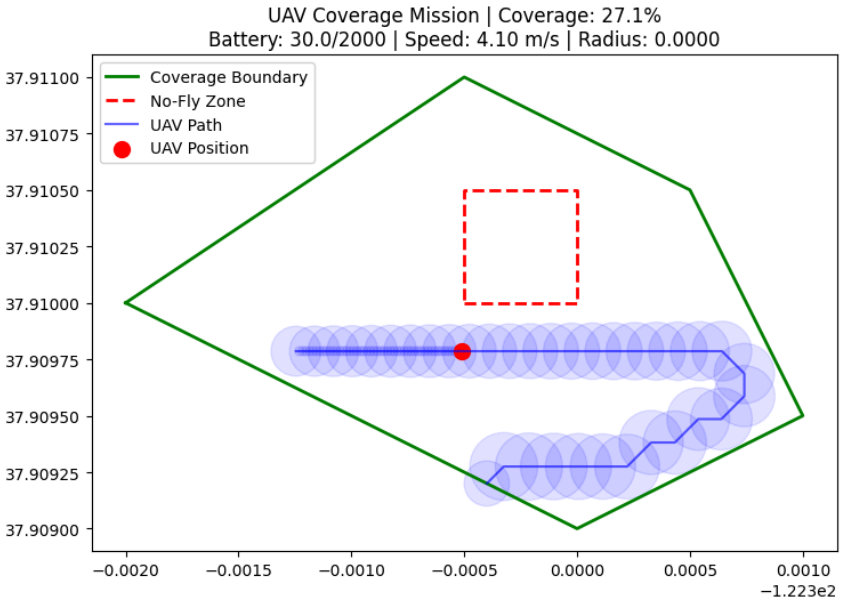

```
Step: Moved from (np.float64(-122.30054044097064), np.float64(37.90978598322125)) to (np.float64(-122.30052544097063), np.float64(37
Energy used: 3.11, remaining: 33.11
Coverage: 27.1%
Coverage radius: 0.0000
```

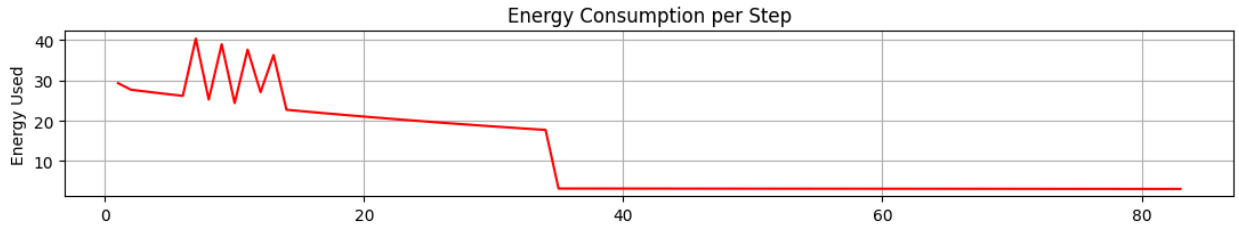


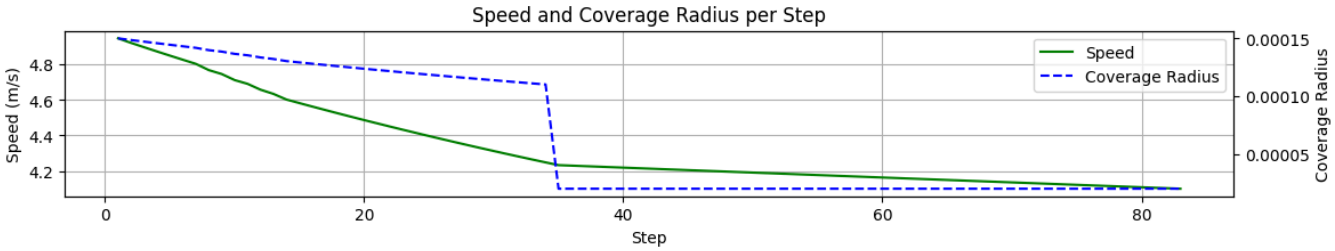




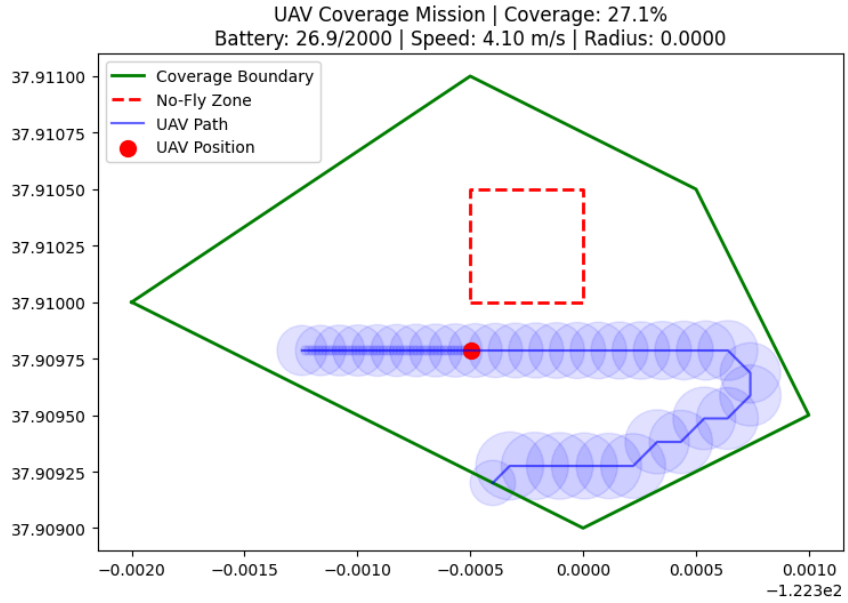

```
Step: Moved from (np.float64(-122.30052544097063), np.float64(37.90978598322125)) to (np.float64(-122.30051044097063), np.float64(37
Energy used: 3.11, remaining: 30.01
Coverage: 27.1%
Coverage radius: 0.0000
```

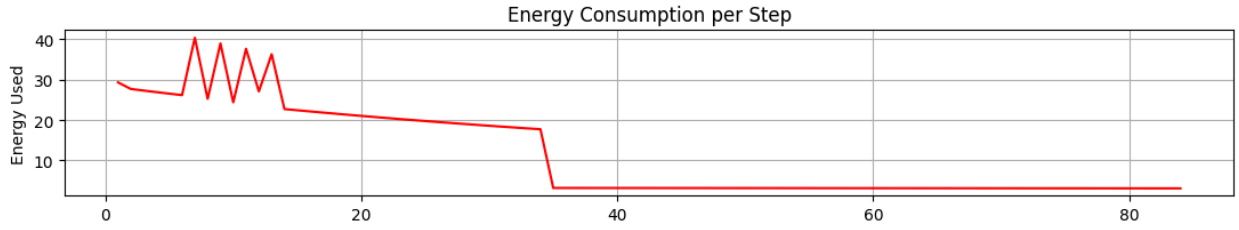


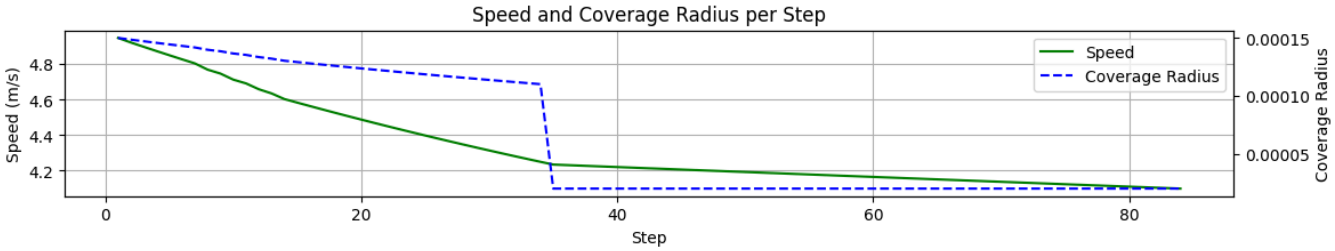




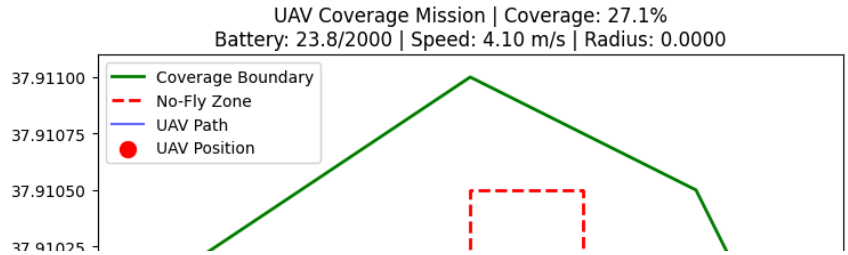

```
Step: Moved from (np.float64(-122.30051044097063), np.float64(37.90978598322125)) to (np.float64(-122.30049544097062), np.float64(37
Energy used: 3.10, remaining: 26.90
Coverage: 27.1%
Coverage radius: 0.0000
```

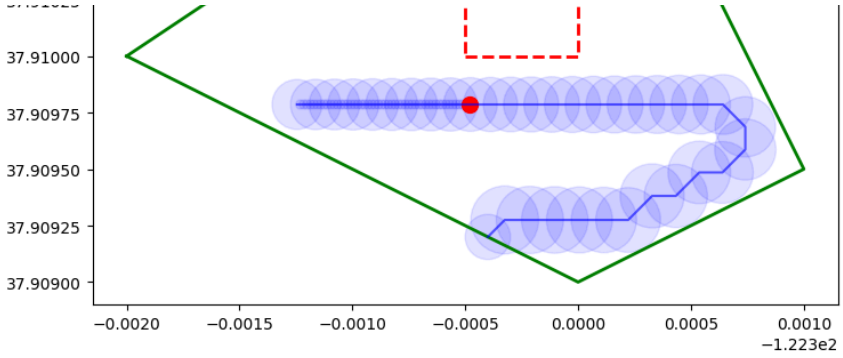


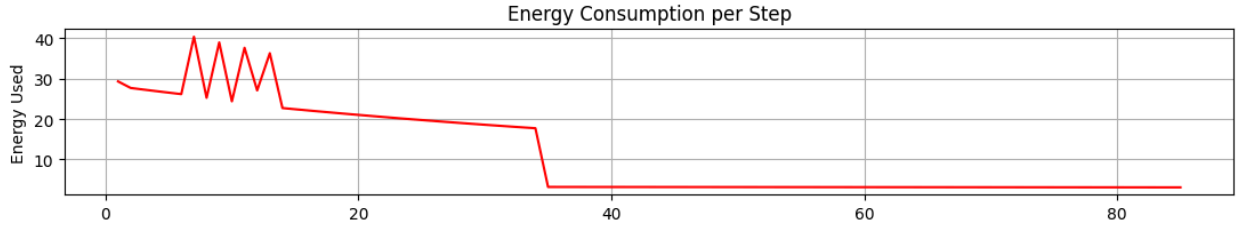


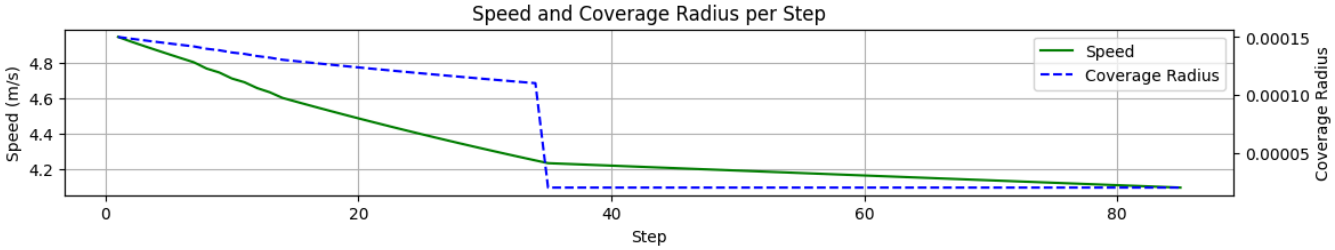



```
Step: Moved from (np.float64(-122.30049544097062), np.float64(37.90978598322125)) to (np.float64(-122.30048044097062), np.float64(37
Energy used: 3.10, remaining: 23.80
Coverage: 27.1%
Coverage radius: 0.0000
```

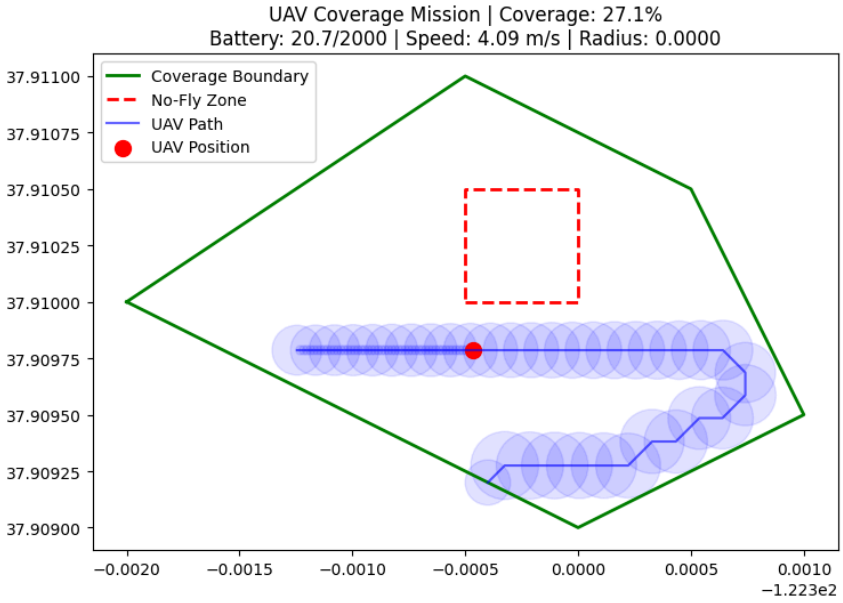


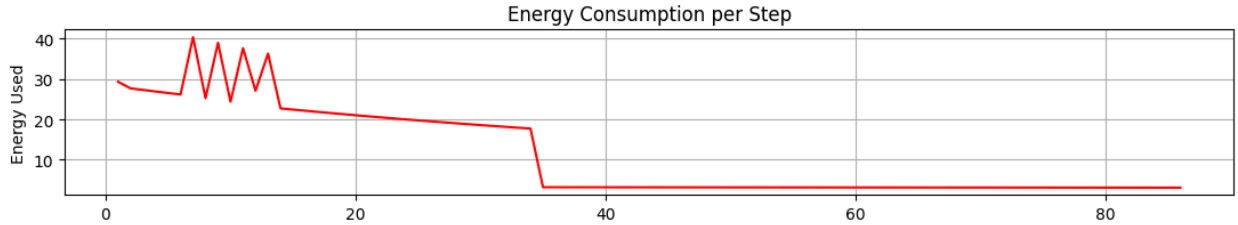


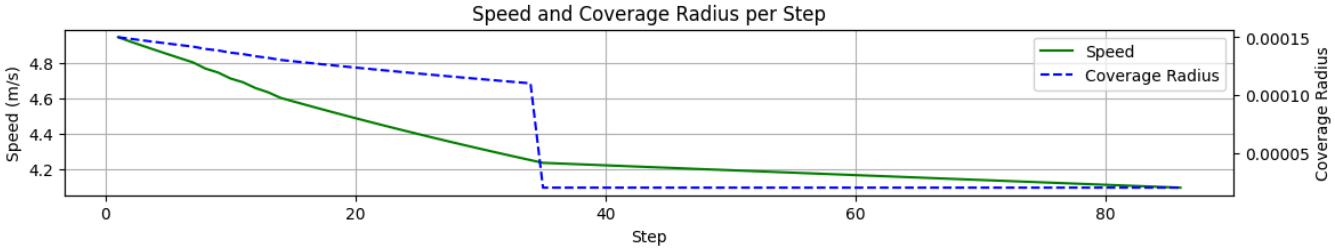




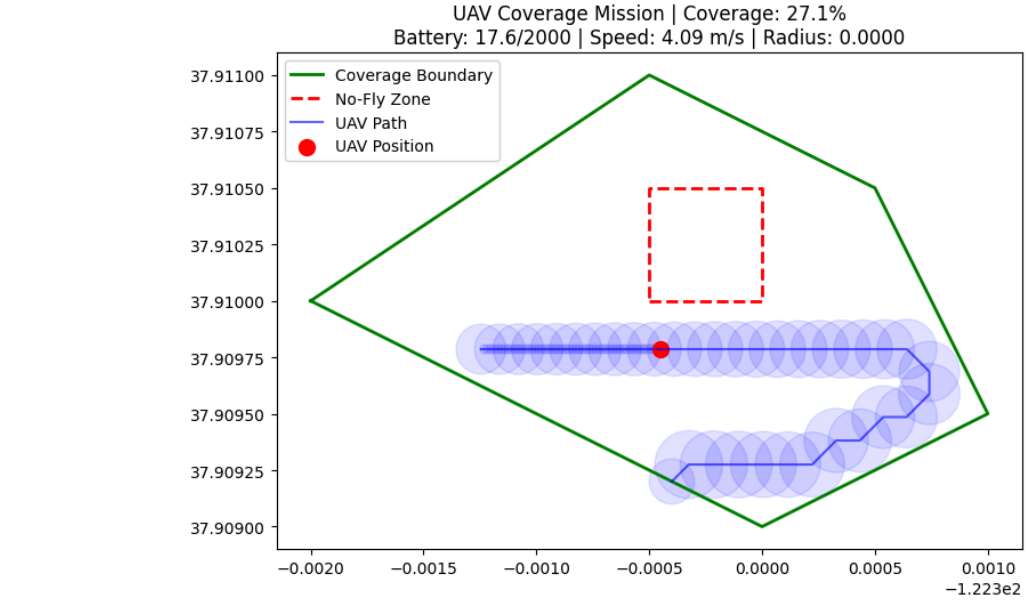

```
Step: Moved from (np.float64(-122.30048044097062), np.float64(37.90978598322125)) to (np.float64(-122.30046544097061), np.float64(37
Energy used: 3.10, remaining: 20.70
Coverage: 27.1%
Coverage radius: 0.0000
```

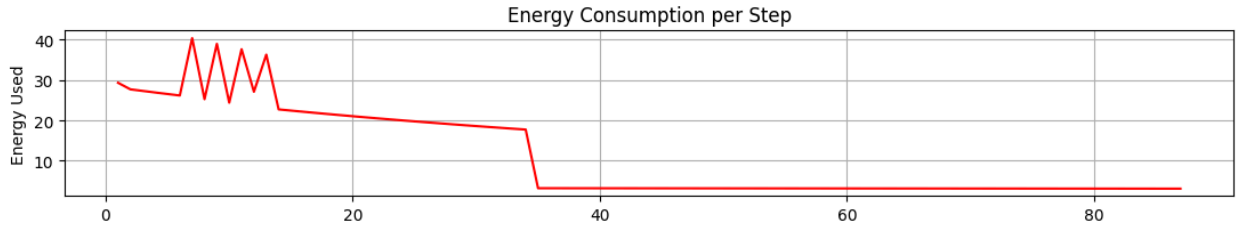


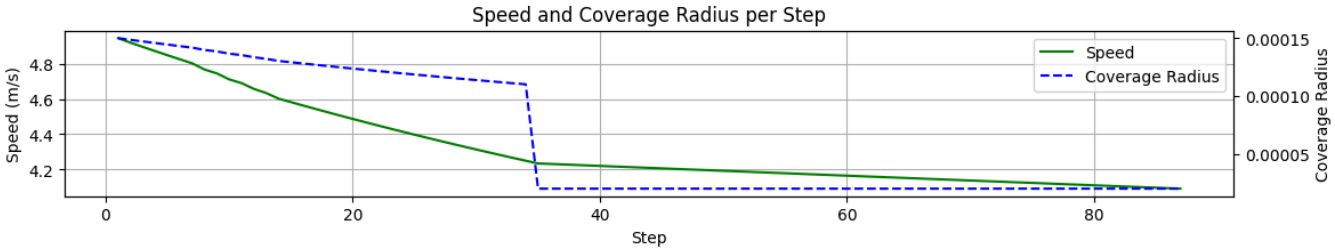




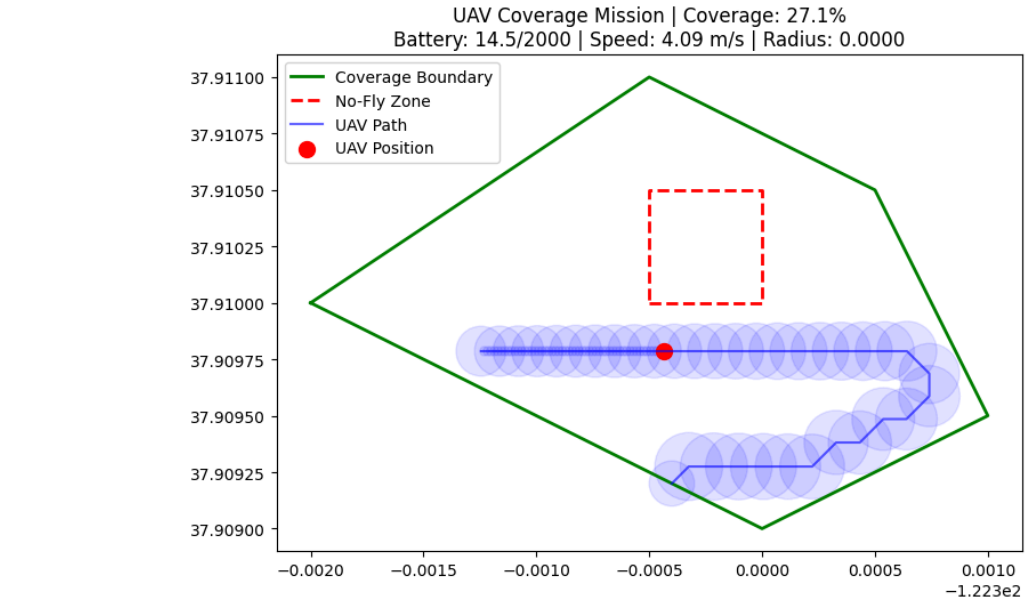
```
Step: Moved from (np.float64(-122.30046544097061), np.float64(37.90978598322125)) to (np.float64(-122.30045044097061), np.float64(37
Energy used: 3.10, remaining: 17.60
Coverage: 27.1%
Coverage radius: 0.0000
```

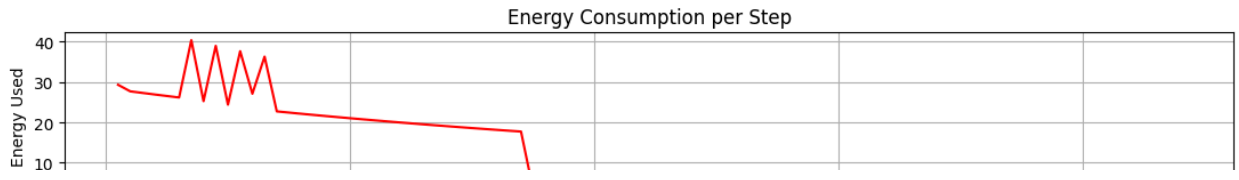


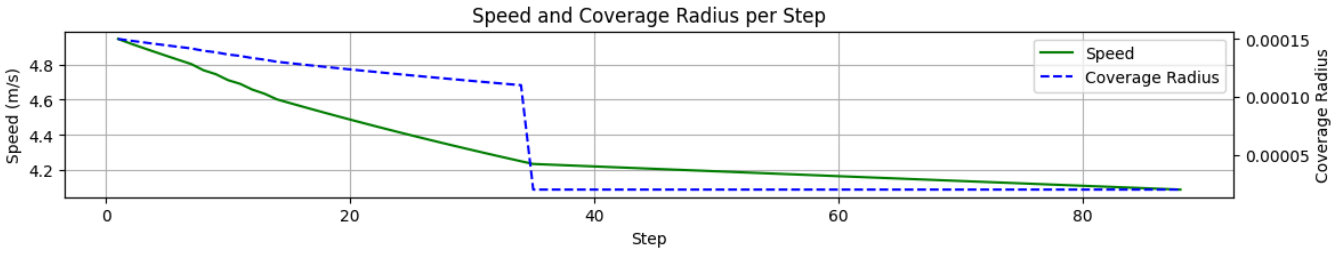




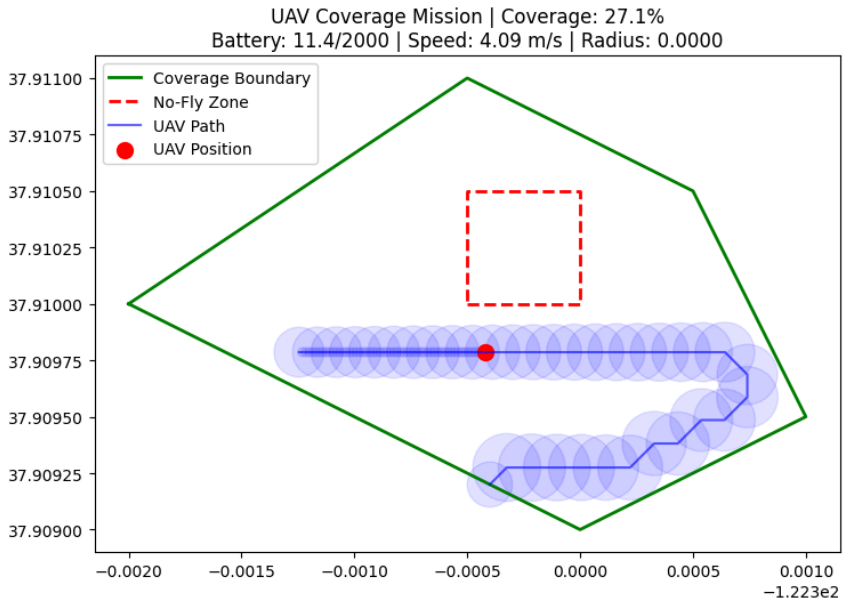

```
Step: Moved from (np.float64(-122.30045044097061), np.float64(37.90978598322125)) to (np.float64(-122.3004354409706), np.float64(37
Energy used: 3.10, remaining: 14.50
Coverage: 27.1%
Coverage radius: 0.0000
```

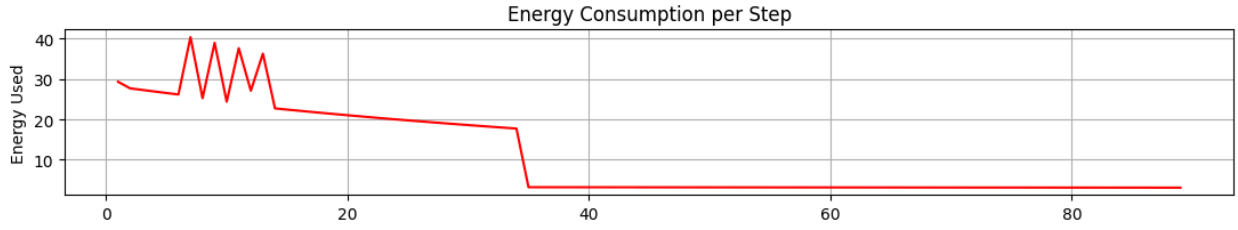


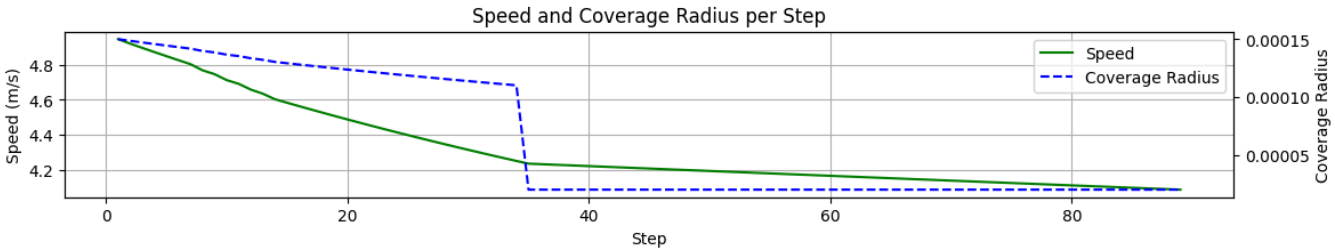




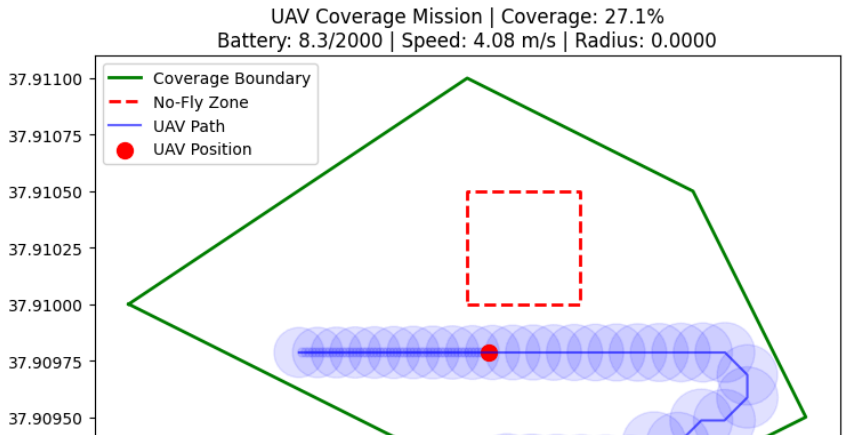

```
Step: Moved from (np.float64(-122.3004354409706), np.float64(37.90978598322125)) to (np.float64(-122.3004204409706), np.float64(37.9
Energy used: 3.09, remaining: 11.41
Coverage: 27.1%
Coverage radius: 0.0000
```

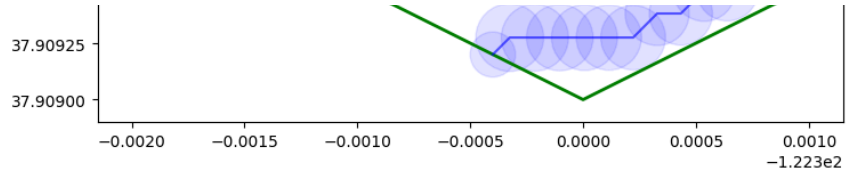


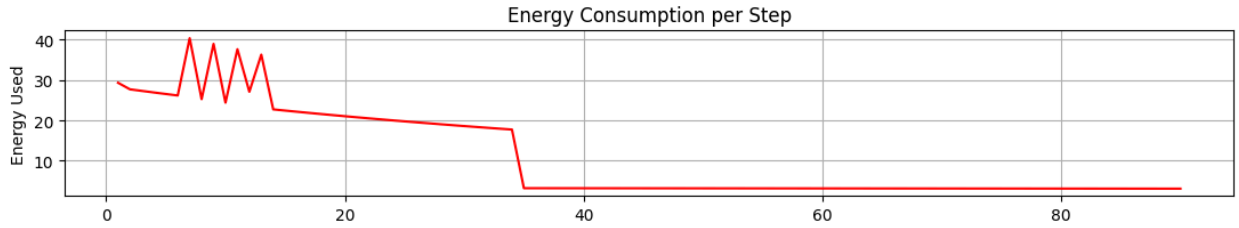


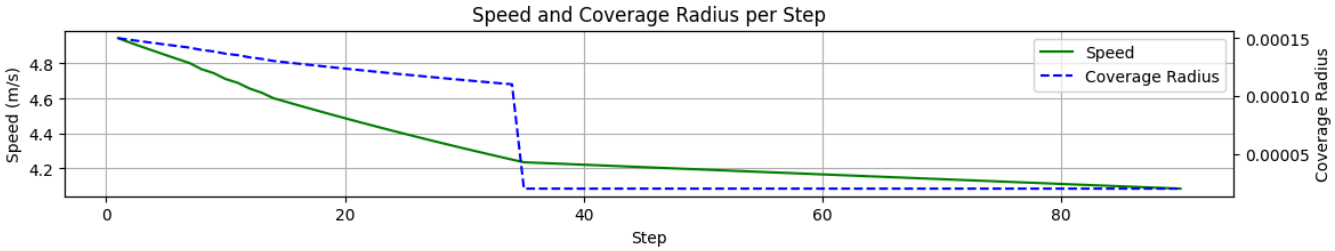



```
Step: Moved from (np.float64(-122.3004204409706), np.float64(37.90978598322125)) to (np.float64(-122.3004054409706), np.float64(37.9
Energy used: 3.09, remaining: 8.32
Coverage: 27.1%
Coverage radius: 0.0000
```

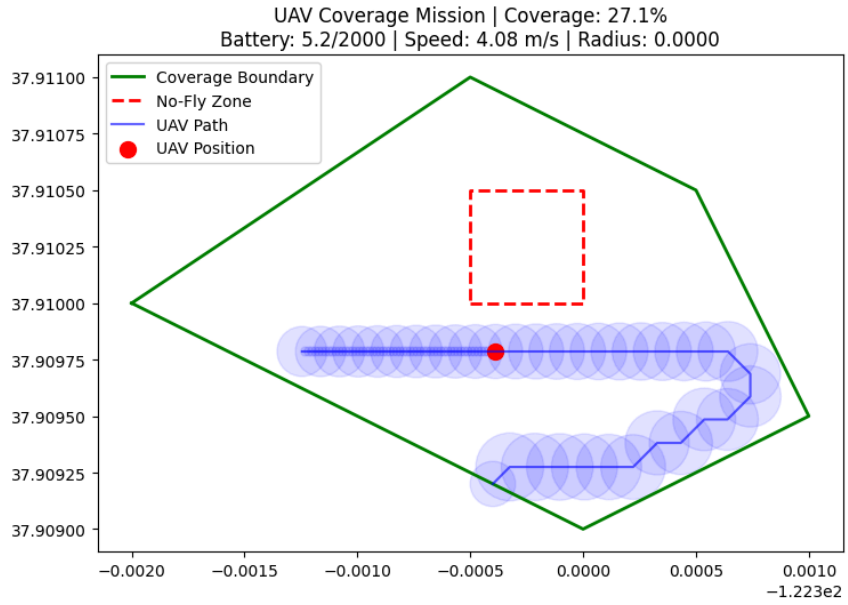


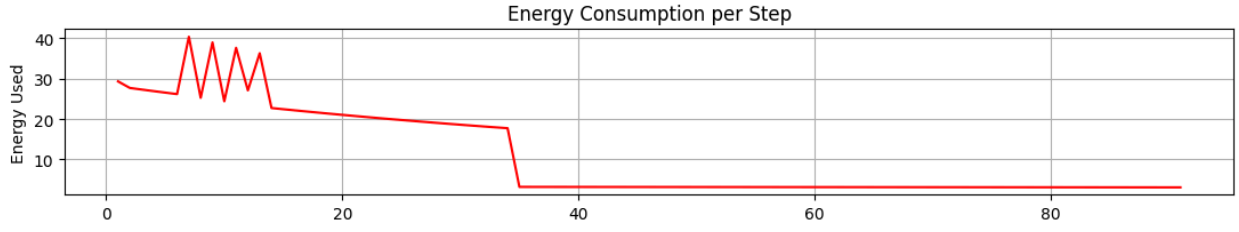


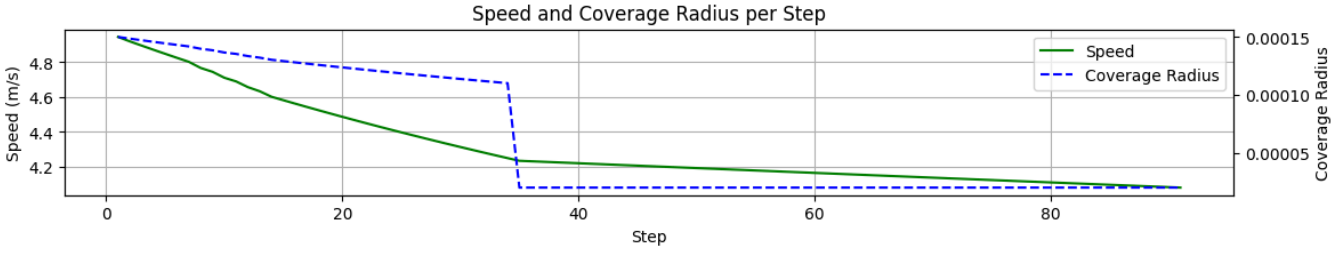




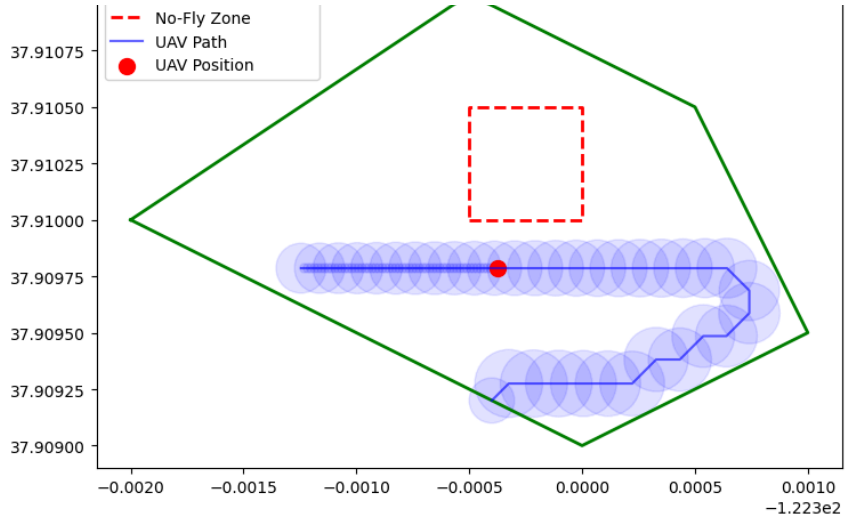

```
Step: Moved from (np.float64(-122.3004054409706), np.float64(37.90978598322125)) to (np.float64(-122.30039044097059), np.float64(37
Energy used: 3.09, remaining: 5.23
Coverage: 27.1%
Coverage radius: 0.0000
```

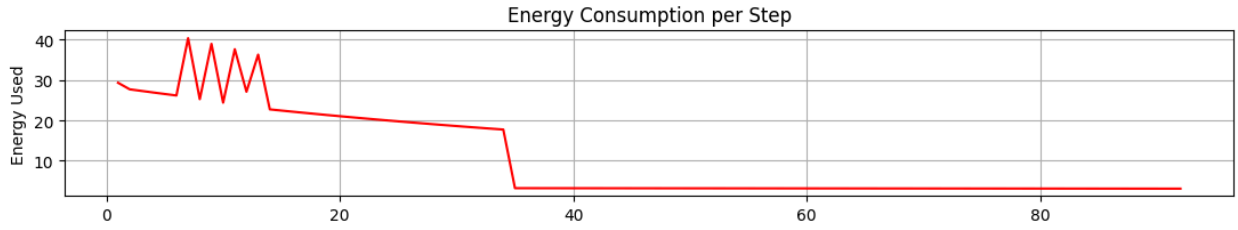


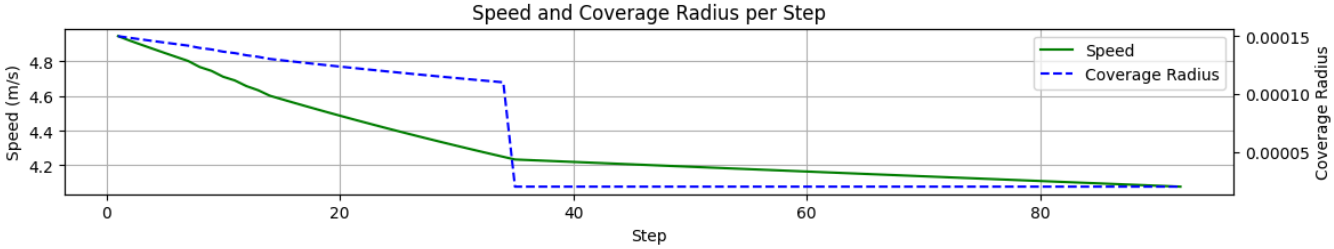




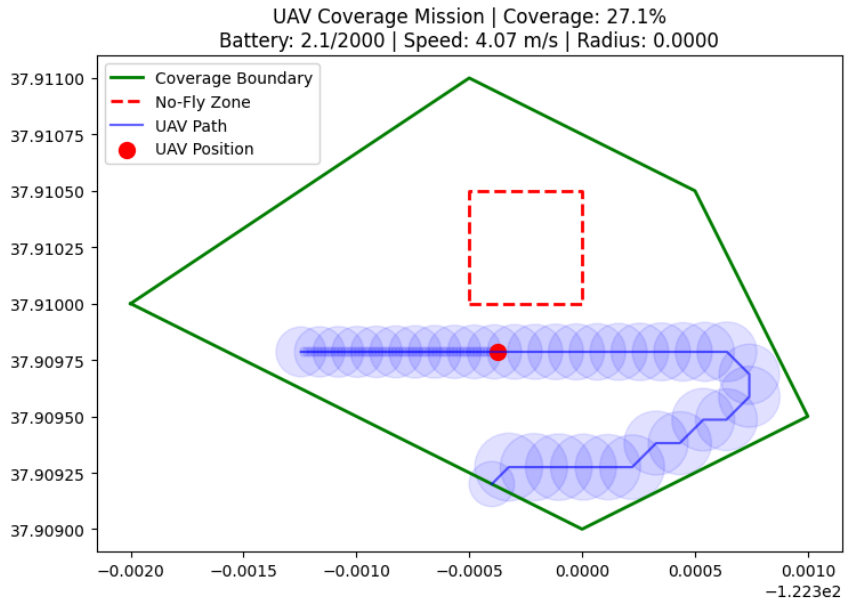

```
Step: Moved from (np.float64(-122.30039044097059), np.float64(37.90978598322125)) to (np.float64(-122.30037544097058), np.float64(37
Energy used: 3.09, remaining: 2.14
Coverage: 27.1%
Coverage radius: 0.0000
```

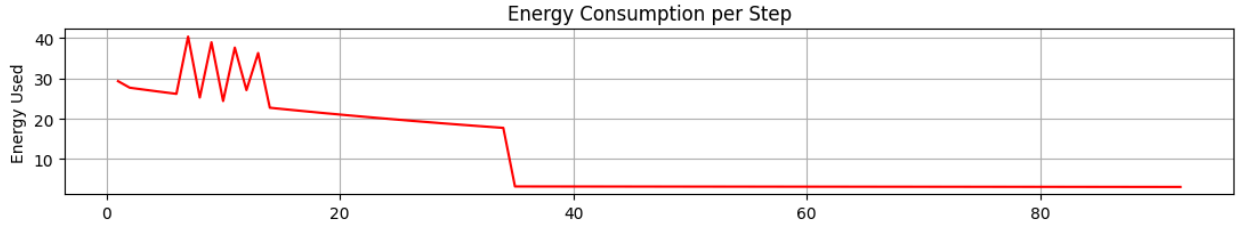


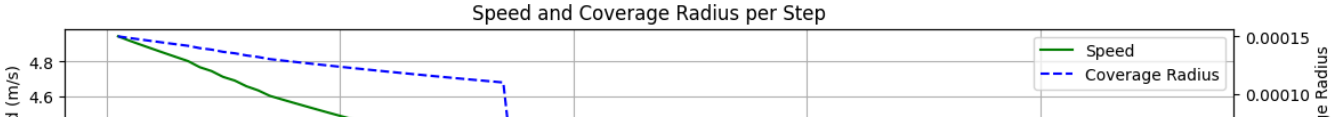


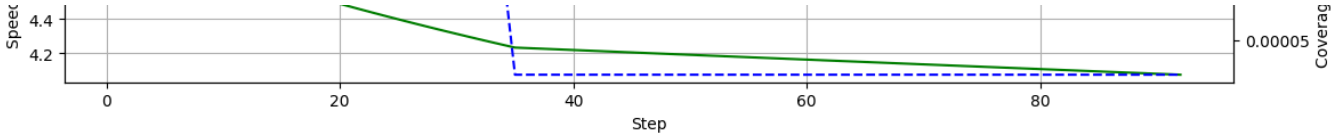



```
WARNING: Couldn't find any valid next points. Trying last resort measures.
No valid next position found. Coverage: 27.1%
Low battery! Unable to complete coverage.
```










```
Final coverage: 27.1%
Remaining battery: 2.1/2000
(False, 0.2714621210929025, np.float64(2.1377893483490915))
```

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from shapely.geometry import Polygon, Point, LineString
import random
import time
import math
from collections import deque
import heapq

class AdaptiveUAVCoverageEnv(gym.Env):
    def __init__(self, coverage_area, no_fly_zones, wind_data, payload_weight, battery_capacity, current_battery):
        super(AdaptiveUAVCoverageEnv, self).__init__()
        print("Initializing Adaptive UAV Environment...")

        # Convert coverage area and no-fly zones into polygons
        self.coverage_area = Polygon([(p["lon"], p["lat"]) for p in coverage_area])
        self.no_fly_zones = [Polygon([(p["lon"], p["lat"]) for p in zone]) for zone in no_fly_zones]

        # Wind data
        self.wind_data = wind_data

        self.energy_estimates = []

        # Calculate area dimensions and area
        minx, miny, maxx, maxy = self.coverage_area.bounds
        self.width = maxx - minx
        self.height = maxy - miny
        self.area = self.coverage_area.area

        # UAV Parameters
        self.payload_weight = payload_weight
        self.battery_capacity = battery_capacity
        self.current_battery = current_battery
        self.initial_battery = current_battery

        # Dynamic parameters
        self.energy_per_area_unit = None  # Will calculate after first movements
        self.coverage_radius = None       # Dynamic coverage radius

        # Initialize base grid size
        self.base_grid_size = min(self.width, self.height) / 10
        self.min_grid_size = self.base_grid_size / 5
        self.max_grid_size = self.base_grid_size * 2

        # Initialize UAV speed
        self.uav_speed = self._compute_dynamic_speed()

        # Generate initial grid with base size
        self.grid_size = self.base_grid_size
        self.grid = self._generate_grid(self.grid_size)

        # Make sure we have at least one point
        if len(self.grid) == 0:
            center_x = (minx + maxx) / 2
            center_y = (miny + maxy) / 2
            print(f"WARNING: No valid grid points generated. Adding center point at ({center_x}, {center_y}).")
            self.grid.append((center_x, center_y))

        # Initialize coverage tracking
        self.position = self.grid[0]
        self.visited = set([self.position])
        self.coverage_map = {}  # Maps points to their coverage radius
        self.coverage_map[self.position] = self._get_current_coverage_radius()

        # Visualization data
        self.trajectory = [self.position]
        self.energy_usage_log = []
        self.speed_log = []
        self.coverage_radius_log = []

        # Debug counters
        self.stuck_count = 0
        self.last_position = None
        self.last_coverage_percentage = 0
        self.energy_estimates = []

        # Set up observation and action spaces
        self.action_space = gym.spaces.Discrete(8)  # 8 directions
        self.observation_space = gym.spaces.Box(low=-1, high=1, shape=(6,), dtype=np.float32)

        # Algorithm settings
```

```
        self.adaptive_step_size = True
        self.dynamic_speed = True
        self.energy_reserve_percentage = 0.1  # Keep 10% battery reserve for emergencies

        print(f"Environment initialized with {len(self.grid)} grid points.")
        print(f"Coverage area: {self.area:.6f} square units")
        print(f"Initial grid size: {self.grid_size:.6f}")

    def _compute_dynamic_speed(self):
        """Adjust speed based on battery level, payload, and wind resistance."""
        base_speed = 10

        # Battery factor (higher speed when more battery available)
        battery_factor = 0.7 + (0.3 * self.current_battery / self.battery_capacity)

        # Payload factor (lower speed with heavier payload)
        payload_factor = max(0.5, 1 - (self.payload_weight / 20))

        # Wind effect
        wind_effect = self._compute_wind_effect()
        wind_factor = max(0.5, 1 - wind_effect)

        # Calculate speed
        speed = base_speed * battery_factor * payload_factor * wind_factor

        return max(2, min(15, speed))

    def _compute_wind_effect(self):
        """Calculate wind effect based on current conditions."""
        wind_direction = np.deg2rad(self.wind_data["direction"])
        move_angle = np.arctan2(1, 1)  # Default angle
        wind_effect = np.cos(wind_direction - move_angle) * self.wind_data["speed"] / 10
        return abs(wind_effect)

    def _generate_grid(self, grid_size):
        """Generate a grid within the coverage area, avoiding no-fly zones."""
        minx, miny, maxx, maxy = self.coverage_area.bounds
        grid = []
        x, y = minx, miny

        while y <= maxy:
            while x <= maxx:
                point = Point(x, y)
                if self.coverage_area.contains(point) and not any(zone.contains(point) for zone in self.no_fly_zones):
                    grid.append((x, y))
                x += grid_size
            x = minx
            y += grid_size

        print(f"Generated {len(grid)} grid points with grid size {grid_size}")
        return grid

    def _get_current_coverage_radius(self):
        """Calculate the coverage radius based on current energy and remaining area."""
        # Base coverage radius
        base_radius = self.grid_size / 2

        # Adjust radius based on battery percentage
        battery_percentage = self.current_battery / self.battery_capacity

        # If we have energy estimates, use them to predict needed radius
        if self.energy_estimates:
            avg_energy_per_unit = sum(self.energy_estimates) / len(self.energy_estimates)
            remaining_area = self.area * (1 - self._calculate_coverage_percentage())
            remaining_energy = self.current_battery - (self.battery_capacity * self.energy_reserve_percentage)

            # If we have plenty of energy, use larger coverage radius
            if remaining_energy > avg_energy_per_unit * remaining_area:
                adaptive_radius = base_radius * (1 + battery_percentage)
            else:
                # Calculate minimum radius needed to cover remaining area with available energy
                adaptive_radius = max(
                    self.min_grid_size / 2,
                    base_radius * min(1.0, remaining_energy / (avg_energy_per_unit * remaining_area))
                )
        else:
            # No energy estimates yet, use battery percentage
            adaptive_radius = base_radius * max(0.5, min(2.0, battery_percentage * 2))

        return max(self.min_grid_size / 2, min(adaptive_radius, self.max_grid_size))

    def _calculate_coverage_percentage(self):
```

```
            """Calculate the current coverage percentage."""
            if not self.coverage_map:
                return 0

            # Create a buffer around each visited point based on its coverage radius
            covered_area = None
            for point, radius in self.coverage_map.items():
                point_obj = Point(point)
                buffer = point_obj.buffer(radius)

                if covered_area is None:
                    covered_area = buffer
                else:
                    covered_area = covered_area.union(buffer)

            # Calculate intersection with coverage area to get actual coverage
            if covered_area:
                coverage_intersection = self.coverage_area.intersection(covered_area)
                return min(1.0, coverage_intersection.area / self.area)
            return 0

        def _is_point_covered(self, point):
            """Check if a point is already covered by existing coverage."""
            point_obj = Point(point)

            for visited_point, radius in self.coverage_map.items():
                visited_point_obj = Point(visited_point)
                if point_obj.distance(visited_point_obj) <= radius:
                    return True

            return False

        def _find_best_next_point(self):
            """Find the best next point to visit for optimal coverage."""
            coverage_percentage = self._calculate_coverage_percentage()

            # If we're already at 100% coverage, return None
            if coverage_percentage >= 0.99:
                return None

            # Get current position
            current_x, current_y = self.position

            # Calculate current coverage radius
            current_radius = self._get_current_coverage_radius()

            # Generate potential next positions based on current grid size
            directions = [
                (1, 0), (-1, 0), (0, 1), (0, -1),
                (1, 1), (-1, -1), (1, -1), (-1, 1)
            ]

            # Adaptive step size based on current energy and coverage
            if self.adaptive_step_size:
                # Smaller steps when battery is low or coverage is high
                step_multiplier = max(0.5, min(2.0, self.current_battery / self.battery_capacity))

                # Adjust step size based on coverage percentage
                if coverage_percentage > 0.8:
                    step_multiplier *= 0.5  # Use smaller steps for fine-tuning at high coverage
            else:
                step_multiplier = 1.0

            # Step size is determined by coverage radius and step multiplier
            step_size = current_radius * 1.5 * step_multiplier

            # Calculate potential next positions
            candidates = []
            for dx, dy in directions:
                next_x = current_x + dx * step_size
                next_y = current_y + dy * step_size
                next_point = (next_x, next_y)

                # Check if the point is valid (inside coverage area, outside no-fly zones)
                point_obj = Point(next_point)
                if not self.coverage_area.contains(point_obj) or any(zone.contains(point_obj) for zone in self.no_fly_zones):
                    continue

                # Calculate energy required to move to this point
                energy_required = self._compute_energy_usage(self.position, next_point)

                # Skip if we don't have enough energy
```

```
            if energy_required >= self.current_battery:
                continue

            # Calculate how much new area this point would cover
            coverage_gain = self._estimate_coverage_gain(next_point, current_radius)

            # Calculate a score based on energy efficiency and coverage gain
            if coverage_gain > 0:
                score = coverage_gain / energy_required
            else:
                score = 0

            candidates.append((next_point, score, energy_required, coverage_gain))

        # If we couldn't find any valid next points with the current step size,
        # try with a smaller step size (half the current one)
        if not candidates and step_size > self.min_grid_size:
            reduced_step_size = max(self.min_grid_size, step_size / 2)
            for dx, dy in directions:
                next_x = current_x + dx * reduced_step_size
                next_y = current_y + dy * reduced_step_size
                next_point = (next_x, next_y)

                # Check if the point is valid
                point_obj = Point(next_point)
                if not self.coverage_area.contains(point_obj) or any(zone.contains(point_obj) for zone in self.no_fly_zones)
                    continue

                energy_required = self._compute_energy_usage(self.position, next_point)

                if energy_required >= self.current_battery:
                    continue

                coverage_gain = self._estimate_coverage_gain(next_point, current_radius)

                if coverage_gain > 0:
                    score = coverage_gain / energy_required
                else:
                    score = 0

                candidates.append((next_point, score, energy_required, coverage_gain))

        # If we still can't find any valid next points, try desperate measures
        if not candidates:
            print("WARNING: Couldn't find any valid next points. Trying last resort measures.")

            # Find uncovered areas and try to reach them
            min_distance = float('inf')
            best_uncovered_point = None

            # Sample points throughout the coverage area
            sample_size = max(100, int(self.area * 10000))  # Adjust sampling density
            minx, miny, maxx, maxy = self.coverage_area.bounds

            for _ in range(sample_size):
                # Generate random point within bounds
                rand_x = random.uniform(minx, maxx)
                rand_y = random.uniform(miny, maxy)
                sample_point = (rand_x, rand_y)

                # Check if point is valid and not already covered
                point_obj = Point(sample_point)
                if not self.coverage_area.contains(point_obj) or any(zone.contains(point_obj) for zone in self.no_fly_zones)
                    continue

                if self._is_point_covered(sample_point):
                    continue

                # Calculate distance to this point
                distance = math.sqrt((self.position[0] - sample_point[0])**2 + (self.position[1] - sample_point[1])**2)

                # Calculate energy required
                energy_required = self._compute_energy_usage(self.position, sample_point)

                # If we can reach it with our energy, consider it
                if energy_required < self.current_battery and distance < min_distance:
                    min_distance = distance
                    best_uncovered_point = sample_point

            # If we found an uncovered point we can reach, use it
            if best_uncovered_point:
                energy_required = self._compute_energy_usage(self.position, best_uncovered_point)
```

```
                coverage_gain = self._estimate_coverage_gain(best_uncovered_point, current_radius)

                if coverage_gain > 0:
                    score = coverage_gain / energy_required
                else:
                    score = 0

                candidates.append((best_uncovered_point, score, energy_required, coverage_gain))

        # Sort candidates by score (highest first)
        candidates.sort(key=lambda x: x[1], reverse=True)

        # Return the best candidate, or None if there are no valid candidates
        return candidates[0][0] if candidates else None

    def _estimate_coverage_gain(self, point, radius):
        """Estimate how much new area would be covered by visiting this point."""
        point_obj = Point(point)
        new_buffer = point_obj.buffer(radius)

        # Calculate current covered area
        current_covered_area = None
        for visited_point, visited_radius in self.coverage_map.items():
            visited_point_obj = Point(visited_point)
            buffer = visited_point_obj.buffer(visited_radius)

            if current_covered_area is None:
                current_covered_area = buffer
            else:
                current_covered_area = current_covered_area.union(buffer)

        if current_covered_area is None:
            # Nothing covered yet, so this is all new coverage
            intersection_with_coverage_area = self.coverage_area.intersection(new_buffer)
            return intersection_with_coverage_area.area
        else:
            # Calculate the new area this point would add
            combined_area = current_covered_area.union(new_buffer)
            new_area = combined_area.area - current_covered_area.area

            # Ensure we only count area within the coverage boundary
            intersection_with_coverage_area = self.coverage_area.intersection(combined_area)
            current_intersection = self.coverage_area.intersection(current_covered_area)

            return intersection_with_coverage_area.area - current_intersection.area

    def _compute_energy_usage(self, from_pos, to_pos):
        """Calculate energy usage for movement between two positions."""
        # Calculate distance
        distance = math.sqrt((from_pos[0] - to_pos[0])**2 + (from_pos[1] - to_pos[1])**2)
        if distance == 0:
            return 0

        # Direction of movement
        dx, dy = to_pos[0] - from_pos[0], to_pos[1] - from_pos[1]
        move_angle = math.atan2(dy, dx) if (dx != 0 or dy != 0) else 0

        # Wind effect based on movement direction
        wind_direction = np.deg2rad(self.wind_data["direction"])
        wind_effect = math.cos(wind_direction - move_angle) * self.wind_data["speed"] / 10

        # Base power usage
        power_usage = 100 + (self.payload_weight * 0.5)

        # Wind resistance factor
        wind_resistance = 1 + abs(wind_effect) * 0.3

        # Speed factor (higher speed = more energy usage)
        speed_factor = self.uav_speed / 10

        # Distance factor
        distance_factor = distance / self.grid_size

        # Calculate final energy usage
        energy_usage = power_usage * wind_resistance * speed_factor * distance_factor

        # Store energy usage per unit distance for future estimates
        if distance > 0:
            energy_per_unit = energy_usage / distance
            self.energy_estimates.append(energy_per_unit)
            # Keep only the last 20 estimates
            if len(self.energy_estimates) > 20:
```

```
                self.energy_estimates = self.energy_estimates[-20:]

        return energy_usage

    def step(self, action=None):
        """Move UAV to next position for optimal coverage."""
        # Update UAV speed if dynamic speed is enabled
        if self.dynamic_speed:
            self.uav_speed = self._compute_dynamic_speed()

        # Find the best next point to visit
        next_position = self._find_best_next_point()

        # If no valid next point found, we're either done or out of energy
        if next_position is None:
            coverage_percentage = self._calculate_coverage_percentage()
            print(f"No valid next position found. Coverage: {coverage_percentage*100:.1f}%")

            # Check if we have satisfactory coverage
            if coverage_percentage >= 0.99:
                print("Achieved full coverage! Mission complete.")
                return self._get_observation(), 10, True, {"success": True}
            else:
                # Check if we're out of energy
                if self.current_battery < self.battery_capacity * 0.1:
                    print("Low battery! Unable to complete coverage.")
                    return self._get_observation(), -5, True, {"success": False, "reason": "low_battery"}
                else:
                    print("Unable to find path to uncovered areas.")
                    return self._get_observation(), -5, True, {"success": False, "reason": "no_path"}

        # Calculate energy for this move
        energy_used = self._compute_energy_usage(self.position, next_position)

        # Check if we have enough energy
        if energy_used <= self.current_battery:
            prev_position = self.position
            self.position = next_position
            self.trajectory.append(self.position)

            # Update coverage map with this position and its coverage radius
            current_radius = self._get_current_coverage_radius()
            self.coverage_map[self.position] = current_radius

            # Update tracking data
            self.visited.add(self.position)
            self.current_battery -= energy_used

            # Log data for visualization
            self.energy_usage_log.append(energy_used)
            self.speed_log.append(self.uav_speed)
            self.coverage_radius_log.append(current_radius)

            # Calculate reward
            coverage_percentage = self._calculate_coverage_percentage()
            coverage_gain = coverage_percentage - self.last_coverage_percentage
            self.last_coverage_percentage = coverage_percentage

            reward = coverage_gain * 100  # Scale reward based on coverage gain

            print(f"Step: Moved from {prev_position} to {next_position}")
            print(f"Energy used: {energy_used:.2f}, remaining: {self.current_battery:.2f}")
            print(f"Coverage: {coverage_percentage*100:.1f}%")
            print(f"Coverage radius: {current_radius:.4f}")
        else:
            # Not enough energy
            print(f"Not enough energy to move to {next_position}.")
            print(f"Required: {energy_used:.2f}, Available: {self.current_battery:.2f}")
            reward = -1

        # Check if done (full coverage or out of energy)
        done = (self._calculate_coverage_percentage() >= 0.99) or (self.current_battery <= 0)

        return self._get_observation(), reward, done, {}

    def _get_observation(self):
        """Get current observation state."""
        coverage_percentage = self._calculate_coverage_percentage()

        return np.array([
            self.position[0],
            self.position[1],
```

```
            self.current_battery / self.battery_capacity,
            self.uav_speed / 15,
            coverage_percentage,
            self._get_current_coverage_radius() / self.max_grid_size
        ], dtype=np.float32)

    def reset(self):
        """Reset UAV for a new mission."""
        # Reset UAV state
        minx, miny, maxx, maxy = self.coverage_area.bounds
        self.position = self.grid[0]
        self.visited = set([self.position])
        self.trajectory = [self.position]
        self.current_battery = self.initial_battery
        self.uav_speed = self._compute_dynamic_speed()

        # Reset coverage tracking
        current_radius = self._get_current_coverage_radius()
        self.coverage_map = {self.position: current_radius}
        self.last_coverage_percentage = 0

        # Reset logs
        self.energy_usage_log = []
        self.speed_log = []
        self.coverage_radius_log = []
        self.energy_estimates = []

        return self._get_observation()

    def render(self, mode='human'):
        """Visualize UAV coverage and path."""
        plt.figure(figsize=(12, 10))

        # Create subplots: main plot and resource usage
        gs = plt.GridSpec(3, 1, height_ratios=[3, 1, 1])
        ax1 = plt.subplot(gs[0])
        ax2 = plt.subplot(gs[1])
        ax3 = plt.subplot(gs[2])

        # Main coverage plot
        # Plot coverage area boundary
        x, y = self.coverage_area.exterior.xy
        ax1.plot(x, y, c="green", linewidth=2, label="Coverage Boundary")

        # Plot no-fly zones
        for i, zone in enumerate(self.no_fly_zones):
            x, y = zone.exterior.xy
            label = "No-Fly Zone" if i == 0 else ""
            ax1.plot(x, y, c="red", linestyle='--', linewidth=2, label=label)

        # Plot the coverage circles for each visited point
        for point, radius in self.coverage_map.items():
            circle = plt.Circle(point, radius, color='blue', fill=True, alpha=0.1)
            ax1.add_patch(circle)

        # Plot the trajectory path
        if len(self.trajectory) > 1:
            trajectory_x, trajectory_y = zip(*self.trajectory)
            ax1.plot(trajectory_x, trajectory_y, 'b-', linewidth=1.5, alpha=0.6, label="UAV Path")

        # Plot current position
        ax1.scatter(*self.position, c="red", s=100, label="UAV Position")

        # Set title with info
        coverage_percentage = self._calculate_coverage_percentage()
        ax1.set_title(f"UAV Coverage Mission | Coverage: {coverage_percentage*100:.1f}%\n"
                      f"Battery: {self.current_battery:.1f}/{self.battery_capacity} | "
                      f"Speed: {self.uav_speed:.2f} m/s | "
                      f"Radius: {self._get_current_coverage_radius():.4f}")

        ax1.set_aspect('equal')

        # Energy usage plot
        steps = range(1, len(self.energy_usage_log) + 1)
        if self.energy_usage_log:
            ax2.plot(steps, self.energy_usage_log, 'r-', label="Energy Usage")
            ax2.set_ylabel("Energy Used")
            ax2.set_title("Energy Consumption per Step")
            ax2.grid(True)

        # Speed and coverage radius plot
        if self.speed_log and self.coverage_radius_log:
```

```
            ax3.plot(steps, self.speed_log, 'g-', label="Speed")
            ax3.set_xlabel("Step")
            ax3.set_ylabel("Speed (m/s)")
            ax3.grid(True)

            # Twin axis for coverage radius
            ax3b = ax3.twinx()
            ax3b.plot(steps, self.coverage_radius_log, 'b--', label="Coverage Radius")
            ax3b.set_ylabel("Coverage Radius")

            # Combine legends
            lines1, labels1 = ax3.get_legend_handles_labels()
            lines2, labels2 = ax3b.get_legend_handles_labels()
            ax3.legend(lines1 + lines2, labels1 + labels2, loc='upper right')

            ax3.set_title("Speed and Coverage Radius per Step")

        # Handle legend for main plot
        handles, labels = ax1.get_legend_handles_labels()
        by_label = dict(zip(labels, handles))
        ax1.legend(by_label.values(), by_label.keys())

        plt.tight_layout()
        plt.show()
        plt.pause(0.2)
        plt.close()

# Example Usage with different coverage areas
def run_simulation(coverage_area, no_fly_zones, battery_capacity):
    wind_data = {"speed": 5, "direction": 90}

    # Initialize Environment
    env = AdaptiveUAVCoverageEnv(
        # coverage_area,
        # no_fly_zones,
        # wind_data,
        # payload_weight=2,
        # battery_capacity=battery_capacity,
        # current_battery=battery_capacity
        coverage_area=test_scenarios[0]["coverage_area"],
        no_fly_zones=test_scenarios[0]["no_fly_zones"],
        wind_data={"speed": 5, "direction": 90},  # Provide wind_data directly
        payload_weight=2,
        battery_capacity=2000,
        current_battery=1000
    )

    # Run Simulation
```