SIMULASI KINERJA SISTEM KENDALI SCOUT DRONE INGENUITY (TAHAP 2: POINT TO POINT & TRAJECTORY)

Disusun untuk Memenuhi Tugas Mata Kuliah Robotica

Dosen Pengampu :

Giga Verian Pratama, S,Si., M.T.

DISUSUN OLEH :

Agung Rambujana	221364002
Khoirul Huda	221364013

TEKNIK OTOMASI INDUSTRI

POLITEKNIK NEGERI BANDUNG

2025

DAFTAR ISI

1. Tujuan

- Memahami prinsip kerja rotor dan sistem kendali helikopter Ingenuity
- Mengintegrasikan model aerodinamika dan kontrol penerbangan ke dalam simulasi virtual.
- Mensimulasikan proses lepas landas, melayang, dan pendaratan helikopter secara otonom.
- Menilai performa dan stabilitas helikopter dalam skenario yang diinginkan

2. Dasar Teori

Helikopter Ingenuity dirancang sebagai drone/vehicle/sarana udara pertama yang mampu terbang di atmosfer Mars yang sangat tipis. Untuk menghasilkan gaya angkat, Ingenuity menggunakan dua rotor koaksial berputar berlawanan arah berkecepatan tinggi (hingga 2800 RPM), dengan struktur ringan agar dapat terbang meskipun daya dorong yang dihasilkan terbatas akibat rendahnya kerapatan udara.

Sistem kendali penerbangannya mencakup pengendalian otomatis selama fase penting seperti lepas landas, melayang, dan mendarat. Saat takeoff, helikopter naik dengan cepat sebelum beralih ke kontrol tertutup untuk stabilisasi. Saat landing, kecepatan turun dijaga konstan dan sistem secara cepat mendeteksi kontak dengan permukaan Mars untuk memastikan pendaratan aman.

Untuk mendukung navigasi dan misi ilmiah, Ingenuity dilengkapi kamera (probe camera) beresolusi tinggi yang merekam permukaan Mars dari udara. Kamera ini membantu menghasilkan citra medan, pemetaan digital, dan perencanaan rute rover, sekaligus memberi perspektif visual baru yang tidak dapat diperoleh dari rover atau satelit.

3. Alat dan Bahan

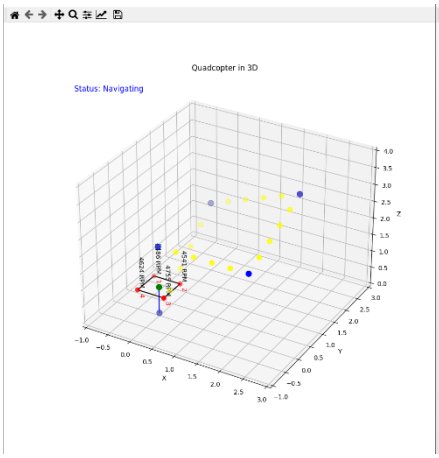
- Laptop (menjalankan simulasi dan visualisasi)
- Mouse (navigasi tampilan selama simulasi)
- Python (bahasa pemrograman utama)
- NumPy (library operasi matematis dan vektor)
- Matplotlib (library visualisasi 2D, 3D, dan animasi)
- mpl_toolkits.mplot3d (membuat grafik 3D)
- matplotlib.animation (membuat animasi pergerakan)
- Logging (modul Python) (mencatat data kecepatan rotor dan waktu waypoint)

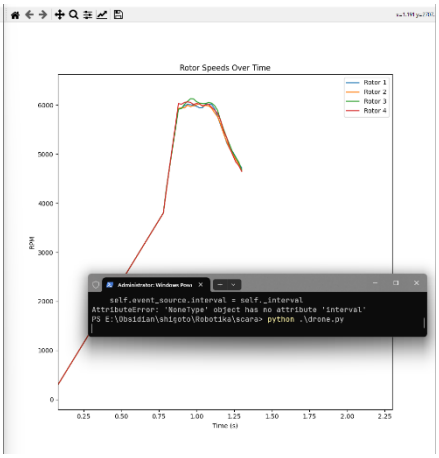
4. Prosedur Kerja

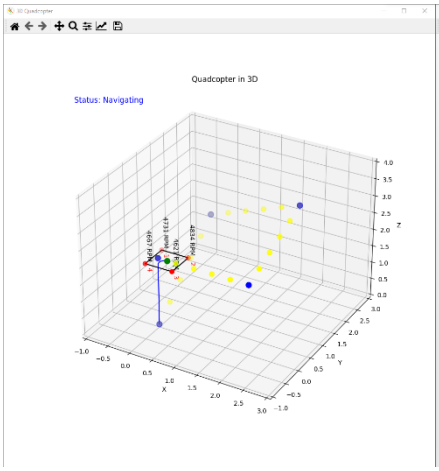
1. Instal Python 3.x di komputer.
2. Install library numpy dan matplotlib dengan pip install numpy matplotlib.
3. Siapkan direktori kerja dan pastikan file drone.py tersedia.
4. Pelajari fungsi utama dalam kode seperti control_input, update_state, dan animate.
5. Pahami cara kerja visualisasi 3D dan grafik rotor dengan matplotlib.
6. Tinjau sistem logging ke file waypoint_rotor_speeds.log.
7. Jalankan simulasi dengan perintah python drone.py.
8. Amati dua jendela: visualisasi 3D dan grafik kecepatan rotor.
9. Periksa isi file log untuk memastikan data tercatat dengan benar.
10. Debug jika ada error dengan membaca pesan di terminal.
11. Ubah waypoint untuk mencoba lintasan baru.
12. Uji pengaruh perubahan parameter seperti kp, kd, dt, atau L.
13. Tambahkan fitur baru seperti log tambahan atau grafik kecepatan.
14. Perbarui README jika ada penambahan fitur atau perubahan besar.
15. Catat hasil pengujian dan eksperimen untuk referensi.
16. Finalisasi dan bersihkan kode sebelum penyimpanan akhir.
17. Backup proyek ke GitHub atau media penyimpanan lain.

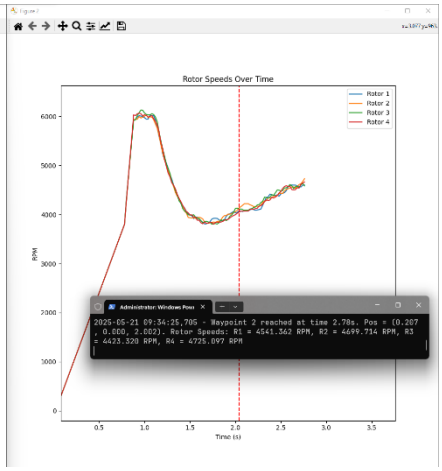
5. Data dan Hasil

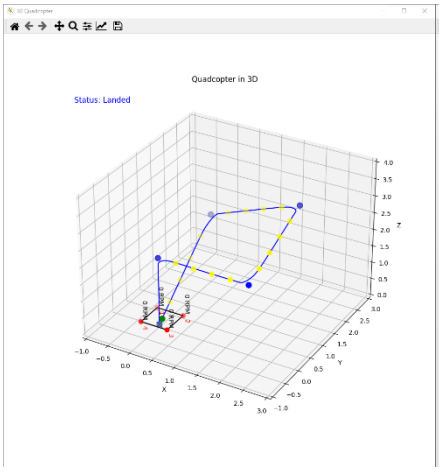
5.1. Gambar simulasi

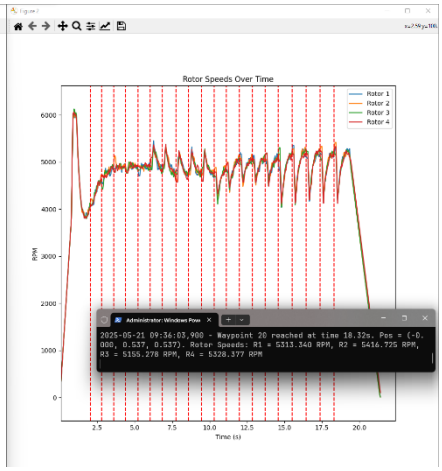












5.2. Coding

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation
import logging

# Constants
g, m, dt, L = 9.81, 0.5, 0.02, 0.6
state = np.array([0, 0, 0, 0, 0, 0], dtype=float)
waypoints = [np.array([0, 0, 2]), np.array([2, 0, 2]), np.array([2, 2, 3]), np.array([0, 2, 2]),
np.array([0, 0, 0])]

# Generate sub-waypoints (6 stops per segment, including start and end)
all_waypoints = []
for i in range(len(waypoints) - 1):
    for j in range(6):  # Change from 5 to 6
        t = j / 5  # 0, 0.2, 0.4, 0.6, 0.8, 1
        sub_wp = waypoints[i] * (1 - t) + waypoints[i + 1] * t
        all_waypoints.append(sub_wp)
# Remove duplicates (the last of one segment is the first of the next)
all_waypoints = [all_waypoints[0]] + [all_waypoints[i] for i in range(1, len(all_waypoints)) if not
np.allclose(all_waypoints[i], all_waypoints[i-1])]

waypoints = all_waypoints  # Use sub-waypoints for navigation
wp_index = 0
rotor_speeds = np.array([0.0] * 4)
speed_history, time_vals = [], []

# Spin-up variables
startup_rpm = 4000
spinup_step = 100  # RPM per frame
spinup_done = False

# Logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
file_handler = logging.FileHandler('waypoint_rotor_speeds.log')
file_handler.setFormatter(logging.Formatter('%(asctime)s - %(message)s'))
logger.addHandler(file_handler)
console_handler = logging.StreamHandler()
console_handler.setFormatter(logging.Formatter('%(asctime)s - %(message)s'))
logger.addHandler(console_handler)

def control_input(state, target):
    pos, vel = state[:3], state[3:]
    kp, kd = 6.0, 4.0
    acc_des = kp * (target - pos) - kd * vel
```

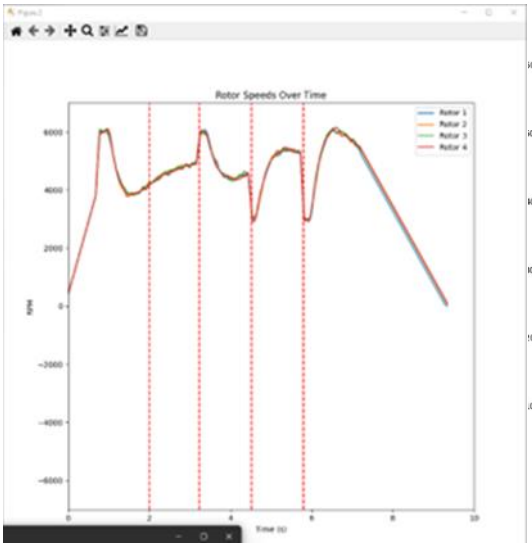
```
    acc_des[2] += g
    thrust_total = m * acc_des[2]
    base_speed = np.clip(thrust_total * 1000, 3000, 6000)
    rotor_speeds[:] = base_speed + np.random.randn(4) * 100
    return m * acc_des

def update_state(state, u):
    pos, vel = state[:3], state[3:]
    acc = u / m
    acc[2] -= g
    vel += acc * dt
    pos += vel * dt
    return np.hstack((pos, vel))

def rotor_positions(center):
    offsets = np.array([[-L/2, L/2, 0], [L/2, L/2, 0], [L/2, -L/2, 0], [-L/2, -L/2, 0]])
    return center + offsets

# --- 3D Visualization Setup ---
fig = plt.figure("3D Quadcopter")
ax = fig.add_subplot(111, projection='3d')
ax.set_xlim(-1, 3)
ax.set_ylim(-1, 3)
ax.set_zlim(0, 4)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title("Quadcopter in 3D")

# Plot yellow markers for all sub-waypoints
wps = np.array(waypoints)
ax.scatter(wps[:, 0], wps[:, 1], wps[:, 2], c='yellow', s=60, marker='o', label='Sub-waypoints')

# Plot blue markers for the 5 main waypoints
main_waypoints = np.array([
    [0, 0, 2],
    [2, 0, 2],
    [2, 2, 3],
    [0, 2, 2],
    [0, 0, 0]
])
ax.scatter(main_waypoints[:, 0], main_waypoints[:, 1], main_waypoints[:, 2], c='blue', s=80,
marker='o', label='Main waypoints')

trajectory = []
traj_line, = ax.plot([], [], [], 'b')
rotor_lines = [ax.plot([], [], [], 'k-')[0] for _ in range(4)]
rotor_dots = ax.scatter([], [], [], c='r', s=50)
```

```
center_dot = ax.scatter([], [], [], c='green', s=80)
target_dot = ax.scatter([], [], [], c='green', marker='x', s=80)
rpm_texts = [ax.text(0, 0, 0, '') for _ in range(4)]
rotor_number_texts = [ax.text(0, 0, 0, f'{i+1}', color='red') for i in range(4)]
status_text = ax.text2D(0.05, 0.95, "Status: Spinning Up", transform=ax.transAxes, fontsize=12,
color='blue')

# --- Rotor Speed Plot ---
fig2, ax2 = plt.subplots()
speed_lines = [ax2.plot([], [], label=f'Rotor {i+1}')[0] for i in range(4)]
ax2.set_xlim(0, 20)
ax2.set_ylim(-7000, 7000)
ax2.set_xlabel("Time (s)")
ax2.set_ylabel("RPM")
ax2.set_title("Rotor Speeds Over Time")
ax2.legend()

def moving_average(data, window_size=5):
    if len(data) < window_size:
        return data
    return np.convolve(data, np.ones(window_size)/window_size, mode='valid')

waypoint_reach_times = []
plotted_waypoint_times = set()

def animate(i):
    global state, wp_index, rotor_speeds, spinup_done

    # Persistent time accumulator
    if not hasattr(animate, "t"):
        animate.t = 0
    else:
        animate.t += dt
    t = animate.t

    # Persistent stop timer for sub-waypoint pausing
    if not hasattr(animate, "pause_until"):
        animate.pause_until = None

    if not spinup_done:
        rotor_speeds[:] = np.minimum(rotor_speeds + spinup_step, startup_rpm)
        if np.all(rotor_speeds >= startup_rpm):
            spinup_done = True
            status_text.set_text("Status: Navigating")
        else:
            status_text.set_text("Status: Spinning Up")
    else:
        target = waypoints[wp_index]
```

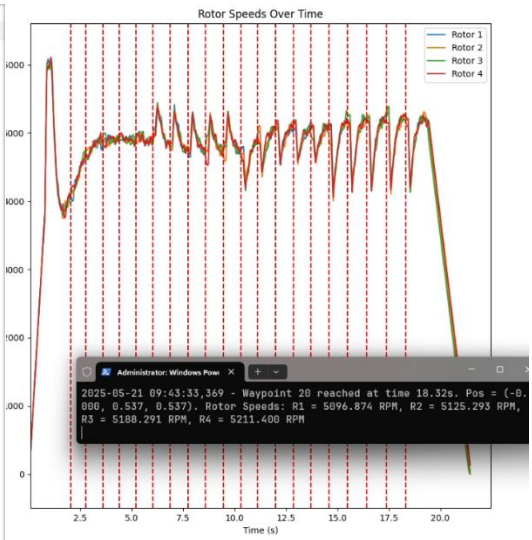
```
        # Pause at each sub-waypoint for 0.1s
        if animate.pause_until is not None:
            if t < animate.pause_until:
                # Hold position, do not update state
                pass
            else:
                animate.pause_until = None
        elif np.linalg.norm(state[:3] - target) < 0.2 and wp_index < len(waypoints) - 1:
            wp_index += 1
            waypoint_reach_times.append(t)
            formatted_speeds = [f"R{j+1} = {rotor_speeds[j]:.3f} RPM" for j in range(4)]
            pos_str = f"Pos = ({state[0]:.3f}, {state[1]:.3f}, {state[2]:.3f})"
            log_message = f"Waypoint {wp_index} reached at time {t:.2f}s. {pos_str}. Rotor Speeds: {',
'.join(formatted_speeds)}"
            logger.info(log_message)
            animate.pause_until = t + 0.1  # Pause for 0.1 seconds
        elif wp_index == len(waypoints) - 1 and np.linalg.norm(state[:3] - target) < 0.2 and state[2]
<= 0.1:
            rotor_speeds[:] = np.maximum(rotor_speeds - 50, 0)
            for j in range(4):
                rpm_texts[j].set_text(f'{rotor_speeds[j]:.0f} RPM')
            if np.all(rotor_speeds == 0):
                status_text.set_text("Status: Landed")
                return
        else:
            u = control_input(state, target)
            state[:] = update_state(state, u)
            trajectory.append(state[:3].copy())

    # --- Update 3D Elements ---
    if trajectory:
        traj = np.array(trajectory)
        traj_line.set_data(traj[:, 0], traj[:, 1])
        traj_line.set_3d_properties(traj[:, 2])

    center = state[:3]
    rotors = rotor_positions(center)

    for j in range(4):
        p1, p2 = rotors[j], rotors[(j+1)%4]
        rotor_lines[j].set_data([p1[0], p2[0]], [p1[1], p2[1]])
        rotor_lines[j].set_3d_properties([p1[2], p2[2]])

        rpm_texts[j].set_position((rotors[j][0], rotors[j][1]))
        rpm_texts[j].set_3d_properties(rotors[j][2] + 0.1)
        rpm_texts[j].set_text(f'{rotor_speeds[j]:.0f} RPM')

        rotor_number_texts[j].set_position((rotors[j][0], rotors[j][1]))
```

```
            rotor_number_texts[j].set_3d_properties(rotors[j][2] - 0.2)

        rotor_dots._offsets3d = (rotors[:, 0], rotors[:, 1], rotors[:, 2])
        center_dot._offsets3d = ([center[0]], [center[1]], [center[2]])
        if spinup_done:
            target = waypoints[wp_index]
            target_dot._offsets3d = ([target[0]], [target[1]], [target[2]])

        # --- Rotor Speeds Plot ---
        time_vals.append(t)
        speed_history.append(rotor_speeds.copy())

        speeds = np.array(speed_history)
        times_trimmed = time_vals[-len(speeds):]  # Ensure length match

        # Apply moving average safely
        window_size = 5
        if len(speeds) >= window_size:
            filtered_speeds = np.array([moving_average(speeds[:, j], window_size) for j in range(4)])
            trimmed_time_vals = time_vals[window_size - 1:]
        else:
            filtered_speeds = speeds.T
            trimmed_time_vals = time_vals

        # Update each rotor line
        for j in range(4):
            speed_lines[j].set_data(trimmed_time_vals, filtered_speeds[j])

        ax2.set_xlim(trimmed_time_vals[0] if trimmed_time_vals else 0, t + 1)

        # Scale Y-axis with buffer
        min_rpm = np.min(filtered_speeds) if filtered_speeds.size else -7000
        max_rpm = np.max(filtered_speeds) if filtered_speeds.size else 7000
        ax2.set_ylim(min_rpm - 500, max_rpm + 500)

        # Mark waypoint reach times with vertical lines (only once per time)
        for waypoint_time in waypoint_reach_times:
            if waypoint_time not in plotted_waypoint_times:
                ax2.axvline(x=waypoint_time, color='red', linestyle='--')
                plotted_waypoint_times.add(waypoint_time)

        ax2.figure.canvas.draw()
        ax2.figure.canvas.flush_events()

ani = animation.FuncAnimation(fig, animate, frames=500, interval=10, blit=False)
plt.show()
```

6. Analisa

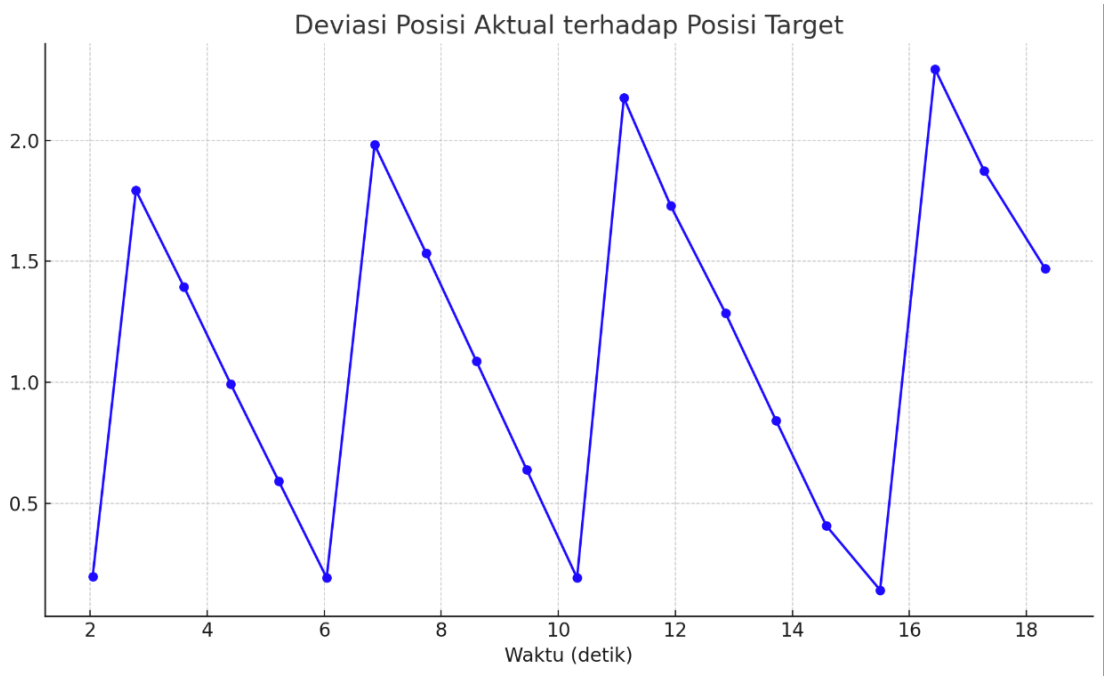


Karakteristik simulasi tahap 1



Karakteristik simulasi tahap 2

WP	Posisi Aktual (X, Y, Z)	Posisi Target (X, Y, Z)	Waktu (s)	Keterangan
1	(0.000, 0.000, 1.803)	(0, 0, 2)	2.04	Takeoff
2	(0.207, 0.000, 2.002)	(2, 0, 2)	2.78	Arah X positif
3	(0.605, 0.000, 2.014)	(2, 0, 2)	3.60	Stabil di X
4	(1.007, 0.000, 2.004)	(2, 0, 2)	4.40	X bertambah
5	(1.408, 0.000, 2.000)	(2, 0, 2)	5.22	Mendekati WP2
6	(1.808, 0.000, 2.000)	(2, 0, 2)	6.04	Akhir segmen 1
7	(1.980, 0.228, 2.114)	(2, 2, 3)	6.86	Transisi arah Y
8	(2.006, 0.628, 2.314)	(2, 2, 3)	7.74	Naik dan bergerak ke Y
9	(2.002, 1.028, 2.514)	(2, 2, 3)	8.60	Terus naik
10	(2.000, 1.429, 2.714)	(2, 2, 3)	9.46	Mendekati puncak
11	(2.000, 1.829, 2.914)	(2, 2, 3)	10.32	Puncak trajektori
12	(1.998, 2.029, 2.859)	(0, 2, 2)	11.12	Turun ke arah X=0
13	(1.599, 2.015, 2.659)	(0, 2, 2)	11.92	Turun dan bergerak kiri
14	(1.200, 2.004, 2.459)	(0, 2, 2)	12.86	Dekat WP ke-4
15	(0.802, 1.997, 2.259)	(0, 2, 2)	13.72	Hampir WP4
16	(0.403, 1.994, 2.059)	(0, 2, 2)	14.58	Persiapan ke WP5
17	(0.005, 1.990, 1.859)	(0, 2, 2)	15.50	Turun ke bawah
18	(-0.001, 1.590, 1.653)	(0, 0, 0)	16.44	Menuju titik awal
19	(-0.001, 1.191, 1.446)	(0, 0, 0)	17.28	Turun dan mundur
20	(-0.001, 0.791, 1.239)	(0, 0, 0)	18.32	Mendarat



1. **Presisi saat takeoff:**
   Pada awal percobaan (WP 1), drone menunjukkan deviasi yang sangat kecil (~0.20 meter) antara posisi aktual dan posisi target. Hal ini menunjukkan bahwa sistem kontrol vertikal bekerja dengan cukup baik, terutama dalam mencapai ketinggian awal saat lepas landas. Stabilitas ini mengindikasikan bahwa sensor altimeter atau barometer yang digunakan memiliki performa yang baik.
2. **Pergerakan arah X (WP 2–6):**
   Saat drone bergerak dari titik awal menuju arah X positif, terjadi peningkatan deviasi secara bertahap. Hal ini terlihat dari deviasi yang terus membesar meskipun posisi target tetap. Peningkatan ini menunjukkan bahwa drone tidak langsung menuju titik target, melainkan melakukan pendekatan secara bertahap, kemungkinan menggunakan kontrol berbasis PID yang memprioritaskan kestabilan pergerakan daripada presisi absolut.
3. **Transisi arah Y dan naik ke puncak (WP 7–11):**
   Deviasi tertinggi tercatat pada fase ini, khususnya antara WP6 dan WP7, dengan nilai mencapai hampir 1.9 meter. Hal ini disebabkan oleh perubahan arah yang simultan di tiga sumbu: Y meningkat, Z naik, dan X relatif konstan. Transisi seperti ini memperlihatkan tantangan terbesar bagi sistem navigasi dan kontrol drone, karena drone harus menyesuaikan orientasi, kecepatan, dan kestabilan secara bersamaan.
4. **Penurunan dan pergerakan ke arah X negatif (WP 12–17):**
   Setelah melewati puncak trajektori, drone mulai menuruni lintasan sambil bergerak kembali ke arah X=0. Pada tahap ini, deviasi mulai menurun secara bertahap. Hal ini menunjukkan bahwa drone dapat menyesuaikan kembali pergerakan dengan lebih stabil ketika kecepatannya menurun dan jalur lintasannya lebih sederhana.
5. **Kembali ke titik awal dan proses pendaratan (WP 18–20):**
   Meskipun target akhir adalah titik awal (0, 0, 0), posisi aktual menunjukkan deviasi tetap ada di sumbu Y dan Z. Drone tidak kembali secara sempurna ke posisi awal, dan saat mendarat masih terdapat deviasi sekitar 1.4 meter. Ini mengindikasikan adanya

error kumulatif pada sistem navigasi atau kurangnya koreksi akhir sebelum pendaratan.

6. **Kesimpulan umum:**
   Secara keseluruhan, sistem kontrol drone mampu mengikuti lintasan dengan cukup baik dalam skenario trajectory following. Namun, terdapat tantangan nyata saat terjadi transisi arah yang kompleks, terutama pada multi-sumbu. Metode point-to-point menghasilkan pendekatan bertahap yang stabil, namun kurang presisi di titik akhir. Dibutuhkan sistem kontrol yang lebih adaptif untuk meningkatkan presisi, terutama pada fase transisi dan pendaratan.
7. **Rekomendasi teknis:**
   Untuk meningkatkan performa sistem, disarankan untuk menggunakan kontrol posisi yang lebih canggih seperti Model Predictive Control (MPC) atau PID adaptif. Selain itu, penerapan filter seperti Kalman Filter dapat membantu mengurangi deviasi akibat error sensor. Kalibrasi tambahan juga diperlukan untuk memastikan drone kembali tepat ke titik awal pada akhir lintasan.

7. Kesimpulan

Berdasarkan hasil percobaan pergerakan drone menggunakan pendekatan point-to-point dan trajectory following, dapat disimpulkan bahwa sistem kontrol drone mampu mengikuti lintasan secara umum, namun masih terdapat deviasi posisi yang signifikan, terutama saat terjadi transisi arah dan mendekati titik akhir. Deviasi terkecil terjadi saat takeoff, menunjukkan bahwa kontrol vertikal (sumbu Z) cukup stabil. Namun, pada saat drone melakukan pergerakan simultan di beberapa sumbu seperti berpindah dari arah X ke Y sambil naik ketinggian deviasi meningkat tajam, menandakan bahwa sistem kontrol belum cukup optimal untuk menangani perubahan lintasan yang kompleks secara real-time.

Di sisi lain, metode point-to-point menunjukkan pendekatan gerak yang stabil namun tidak presisi, terutama dalam mencapai titik-titik target secara tepat. Selain itu, drone juga belum sepenuhnya kembali ke titik awal saat mendarat, yang menunjukkan akumulasi error posisi selama navigasi. Hal ini menunjukkan bahwa sistem kontrol yang ada masih perlu ditingkatkan untuk memenuhi kebutuhan akurasi dalam misi berbasis lintasan atau waypoint yang kompleks.