



TUGAS AKHIR - EE234899

**ALGORITMA PEMBAGIAN TUGAS TERDISTRIBUSI
MULTI KAMIKAZE UAV DENGAN PENGHINDARAN
HALANGAN DINAMIS MENGGUNAKAN DYNAMIC
ARTIFICIAL POTENTIAL FIELD**

MUHAMMAD FAIZ RAMADHAN
NRP 5022211013

Dosen Pembimbing
Dr. Mochammad Sahal, S.T., M.Sc.
NIP 197011191998021002
Yusuf Bilfaqih, ST., MT.
NIP 197203251999031001

Program Studi Sarjana Teknik Elektro
Departemen Teknik Elektro
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya
2025



TUGAS AKHIR - EE234899

**ALGORITMA PEMBAGIAN TUGAS TERDISTRIBUSI
MULTI KAMIKAZE UAV DENGAN PENGHINDARAN
HALANGAN DINAMIS MENGGUNAKAN DYNAMIC
ARTIFICIAL POTENTIAL FIELD**

**MUHAMMAD FAIZ RAMADHAN
NRP 5022211013**

Dosen Pembimbing
Dr. Mochammad Sahal, S.T., M.Sc.
NIP 197011191998021002
Yusuf Bilfaqih, ST., MT.
NIP 197203251999031001

Program Studi Sarjana Teknik Elektro
Departemen Teknik Elektro
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya
2025

Halaman ini sengaja dikosongkan



FINAL PROJECT - EE234899

**DISTRIBUTED ASSIGNMENT SWITCH FOR MULTIPLE
KAMIKAZE UAVS WITH DYNAMIC OBSTACLE
AVOIDANCE USING DYNAMIC ARTIFICIAL POTENTIAL
FIELDS**

MUHAMMAD FAIZ RAMADHAN
NRP 5022211013

Advisor
Dr. Mochammad Sahal, S.T., M.Sc.
NIP 197011191998021002
Yusuf Bilfaqih, ST., MT.
NIP 197203251999031001

Study Program Bachelor of Electrical Engineering
Department of Electrical Engineering
Faculty of intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya
2025

Halaman ini sengaja dikosongkan

LEMBAR PENGESAHAN

ALGORITMA PEMBAGIAN TUGAS TERDISTRIBUSI MULTI KAMIKAZE UAV DENGAN PENGHINDARAN HALANGAN DINAMIS MENGGUNAKAN DYNAMIC ARTIFICIAL POTENTIAL FIELD

TUGAS AKHIR

Diajukan untuk memenuhi salah satu syarat
memperoleh gelar Sarjana Teknik pada
Program Studi S-1 Teknik Elektro
Departemen Teknik Elektro
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember

Oleh : **MUHAMMAD FAIZ RAMADHAN**
NRP. 5022211013

Disetujui oleh Tim Penguji Tugas Akhir :

- | | | |
|---|---------------|---|
| 1. Dr. Mochammad Sahal, S.T., M.Sc. | Pembimbing |  |
| 2. Yusuf Bilfaqih, ST., MT. | Ko-pembimbing |  |
| 3. Prof. Ir. Abdullah Alkaf, M.Sc., Ph.D. | Penguji |  |
| 4. Zulkifli Hidayat, S.T., M.Sc. | Penguji |  |
| 5. Nurlita Gamayanti, S.T., M.T. | Penguji |  |

SURABAYA
Juli, 2025

Halaman ini sengaja dikosongkan

PERNYATAAN ORISINALITAS

Yang bertanda tangan di bawah ini:

Nama mahasiswa / NRP : Muhammad Faiz Ramadhan / 5022211013
Program studi : Teknik Elektro
Dosen Pembimbing / NIP : 1. Dr. Mochammad Sahal, S.T., M.Sc. /
197011191998021002
2. Yusuf Bilfaqih, ST., MT./ 197203251999031001

dengan ini menyatakan bahwa Tugas Akhir dengan judul "Algoritma Pembagian Tugas Terdistribusi Multi Kamikaze UAV dengan Penghindaran Halangan Dinamis menggunakan Dynamic Artificial Potential Field" adalah hasil karya sendiri, bersifat orisinal, dan ditulis dengan mengikuti kaidah penulisan ilmiah.

Bilamana di kemudian hari ditemukan ketidaksesuaian dengan pernyataan ini, maka saya bersedia menerima sanksi sesuai dengan ketentuan yang berlaku di Institut Teknologi Sepuluh Nopember.

Surabaya, 18 Juli 2025

Mengetahui
Dosen Pembimbing

Dr. Mochammad Sahal, S.T., M.Sc.
NIP. 197011191998021002

Mengetahui
Dosen Ko-Pembimbing

Yusuf Bilfaqih, ST., MT.
NIP. 197203251999031001

Mahasiswa

Muhammad Faiz Ramadhan
NRP. 5022211013

Halaman ini sengaja dikosongkan

ABSTRAK

ALGORITMA PEMBAGIAN TUGAS TERDISTRIBUSI MULTI KAMIKAZE UAV DENGAN PENGHINDARAN HALANGAN DINAMIS MENGGUNAKAN DYNAMIC ARTIFICIAL POTENTIAL FIELD

Nama Mahasiswa / NRP : Muhammad Faiz Ramadhan / 5022211013
Departemen : Teknik Elektro FTEIC - ITS
Dosen Pembimbing : Dr. Mochammad Sahal, S.T., M.Sc.
Yusuf Bilfaqih, ST., MT.

Abstrak

Operasi multi-UAV di ruang tiga dimensi selalu berhadapan dengan trade off antara efisiensi lintasan dan waktu tempuh di satu sisi serta margin keselamatan terhadap tabrakan antarpesawat maupun rintangan di sisi lain. Penelitian ini menawarkan kerangka terpadu yang menggabungkan Dynamic Artificial Potential Field dengan Distributed Assignment Switch sehingga penghindaran rintangan dan pembagian tugas berlangsung serempak tanpa koordinator pusat, lalu dievaluasi terhadap Modified Artificial Potential Field sebagai pembanding. Hasil pengujian menunjukkan DAPF menjaga pemisahan antar agen dan mempertahankan jarak ke rintangan sekitar 2.5 kali. Pada lingkungan dengan rintangan statis total lintasan berkurang sekitar sepuluh persen meskipun waktu tempuh sedikit lebih panjang dibanding MAPF. Sebaliknya pada rintangan dinamis MAPF menyelesaikan misi sekitar 26% lebih cepat dan lintasannya hanya 3.6% lebih pendek namun margin keselamatan turun tajam. Integrasi DAPF dan DAS dengan demikian menghasilkan sistem multi UAV yang konsisten mencapai target sekaligus mempertahankan jarak aman dengan kebutuhan komputasi yang tetap realistik untuk operasi daring.

Kata kunci: *Dynamic Artificial Potential Field, Distributed Assignment Switch, MultiAgent, Collision Avoidance*

Halaman ini sengaja dikosongkan

ABSTRACT

DISTRIBUTED ASSIGNMENT SWITCH FOR MULTIPLE KAMIKAZE UAVS WITH DYNAMIC OBSTACLE AVOIDANCE USING ARTIFICIAL POTENTIAL FIELDS

Student Name / NRP : Muhammad Faiz Ramadhan / 5022211013
Department : Teknik Elektro FTEIC - ITS
Advisor : Dr. Mochammad Sahal, S.T., M.Sc.
Yusuf Bilfaqih, ST., MT.

Abstract

Multi UAV operation in three dimensional space faces a persistent trade off between path and time efficiency on one hand and safety margins against inter vehicle and obstacle collisions on the other. This study proposes an integrated framework that combines Dynamic Artificial Potential Field with Distributed Assignment Switch so obstacle avoidance and task allocation proceed simultaneously without a central coordinator, and evaluates it against Modified Artificial Potential Field as a baseline. Experiments show that DAPF maintains inter agent separation and obstacle clearance about two and a half times larger. In static obstacle environments the total path length is reduced by roughly 10% although mission time becomes slightly longer than with MAPF. In dynamic obstacle settings MAPF completes the mission about 26% faster and its paths are only 3.6% shorter but the safety margin drops sharply. The integration of DAPF and DAS therefore yields a multi UAV system that consistently reaches its targets while preserving safe separation with computational demands that remain practical for online operation.

Keywords: *Dynamic Artificial Potential Field, Distributed Assignment Switch, Multi-Agent, Collision Avoidance.*

Halaman ini sengaja dikosongkan

KATA PENGANTAR

Puji syukur ke hadirat Allah SWT atas rahmat dan karunia-Nya, sehingga penulis dapat menyelesaikan Tugas Akhir dengan judul “Igoritma Pembagian Tugas Terdistribusi Multi Kamikaze UAV dengan Penghindaran Halangan Dinamis Menggunakan Artificial Potential Field” ini dengan baik dan tepat waktu.

Penulisan Tugas Akhir ini merupakan salah satu syarat untuk meraih gelar Sarjana S1 pada Program Studi Teknik Sistem Kontrol, Departemen Teknik Elektro, Fakultas Teknologi Elektro dan Informatika Cerdas, Institut Teknologi Sepuluh Nopember.

Penyelesaian Tugas Akhir ini tidak lepas dari dukungan, bimbingan, dan bantuan dari berbagai pihak. Oleh karena itu, penulis menyampaikan rasa terima kasih yang tulus kepada:

1. Keluarga tercinta, atas doa, dukungan moral, serta kepercayaan yang tak ternilai selama proses penyusunan karya ini.
2. Bapak Dr. Mochammad Sahal, S.T., M.Sc., selaku Pembimbing I, dan Bapak Yusuf Bilfaqih, ST., MT., selaku Pembimbing II, atas bimbingan, arahan, koreksi, dan semangat yang senantiasa diberikan hingga Tugas Akhir ini dapat terselesaikan.
3. Segenap rekan-rekan Program Studi Teknik Sistem Kontrol dan Asisten Laboratorium Kontrol dan Otomasi atas kerja sama dan dukungan fasilitas yang diberikan.
4. Semua pihak yang tidak dapat disebutkan satu per satu, atas dukungan dan kontribusi, baik secara langsung maupun tidak langsung, dalam penyelesaian Tugas Akhir ini.

Penulis menyadari bahwa Tugas Akhir ini masih memiliki kekurangan. Oleh karena itu, saran dan kritik yang membangun sangat diharapkan demi penyempurnaan karya ini dan kontribusi bagi pengembangan ilmu pengetahuan di masa mendatang. Penulis berharap Tugas Akhir ini dapat memberikan manfaat serta informasi yang berguna bagi pembaca.

Surabaya, 23 Juli 2025

Muhammad Faiz Ramadhan
50222111013

Halaman ini sengaja dikosongkan

DAFTAR ISI

| | |
|--|-------|
| LEMBAR PENGESAHAN | vii |
| PERNYATAAN ORISINALITAS | ix |
| ABSTRAK | xi |
| ABSTRACT | xiii |
| KATA PENGANTAR | xv |
| DAFTAR ISI | xvii |
| DAFTAR GAMBAR | xix |
| DAFTAR TABEL | xxiii |
| DAFTAR SIMBOL | xxiv |
| BAB 1 PENDAHULUAN | 26 |
| 1.1 Latar Belakang | 26 |
| 1.2 Rumusan Masalah | 27 |
| 1.3 Batasan Masalah | 27 |
| 1.4 Tujuan | 27 |
| 1.5 Manfaat | 27 |
| BAB 2 TINJAUAN PUSTAKA | 28 |
| 2.1 Hasil Penelitian Terdahulu | 28 |
| 2.2 Dasar Teori | 28 |
| 2.2.1 Quadcopter | 28 |
| 2.2.2 State feedback controller | 35 |
| 2.2.3 Dynamic Artificial Potential Field (DAPF)) | 35 |
| 2.2.4 Algoritma pembagian tugas terdistribusi | 37 |
| BAB 3 METODOLOGI | 40 |
| 3.1 Metode yang Digunakan | 40 |
| 3.1.1 Pemodelan sistem | 41 |
| 3.1.2 Perancangan dan Implementasi Sistem | 43 |
| 3.2 Bahan dan Peralatan yang Digunakan | 48 |
| 3.2.1 <i>Personal computer</i> | 48 |
| 3.2.2 MATLAB | 48 |
| 3.3 Urutan Pelaksanaan Penelitian | 49 |
| 3.3.1 Studi literatur | 49 |
| 3.3.2 Perancangan dan implementasi sistem | 49 |
| 3.3.3 Pengujian sistem dan evaluasi | 49 |
| 3.3.4 Penyusunan laporan tugas akhir | 50 |
| BAB 4 HASIL DAN PEMBAHASAN | 52 |

| | | |
|-------|--|-----|
| 4.1 | Simulasi Perencanaan Lintasan UAV dengan <i>Artificial Potential Field</i> | 52 |
| 4.1.1 | Penghindaran halangan statis | 52 |
| 4.1.2 | Penghindaran halangan dinamis | 68 |
| 4.2 | Simulasi Pembagian Tugas Terdistribusi UAV..... | 93 |
| 4.2.1 | Konfigurasi jumlah UAV dan target | 93 |
| 4.3 | Simulasi Pembagian Tugas dan Penghindaran Halangan..... | 95 |
| BAB 5 | KESIMPULAN DAN SARAN..... | 104 |
| 5.1 | Kesimpulan | 104 |
| 5.2 | Saran | 104 |
| | DAFTAR PUSTAKA | 105 |
| | LAMPIRAN | 107 |
| | BIODATA PENULIS | 200 |

DAFTAR GAMBAR

| | |
|---|----|
| Gambar 2.1 Pergerakan Quadcopter..... | 29 |
| Gambar 2.2 Ilustrasi Gerakan <i>Roll</i> pada Quadcopter | 30 |
| Gambar 2.3 Ilustrasi Gerakan <i>Pitch</i> pada Quadcopter | 31 |
| Gambar 2.4 Ilustrasi Gerakan <i>Yaw</i> pada Quadcopter | 31 |
| Gambar 2.6 Batas jarak aman dinamis | 36 |
| Gambar 3.1 Skema Keseluruhan Sistem | 40 |
| Gambar 3.2 Quanser Qdrone | 41 |
| Gambar 3.3 Lingkungan Simulasi $200 \times 200 \times 100$ unit | 42 |
| Gambar 3.4 Halangan Statis dengan variasi radius dan ketinggian | 42 |
| Gambar 3.5 Halangan Dinamis dengan radius aman | 43 |
| Gambar 3.6 Diagram Alir Pelaksanaan Tugas Akhir..... | 49 |
| Gambar 4.1 Simulasi Pengujian <i>Obstacle</i> Terpusat dalam Lintasan DAPF | 52 |
| Gambar 4.2 Tampilan 3D Pengujian <i>Obstacle</i> Terpusat dalam Lintasan DAPF | 53 |
| Gambar 4.3 Simulasi Pengujian <i>Obstacle</i> Terpusat dalam Lintasan MAPF..... | 53 |
| Gambar 4.4 Tampilan 3D Pengujian <i>Obstacle</i> Terpusat dalam Lintasan MAPF | 53 |
| Gambar 4.5 Jarak Agen 1 ke Setiap Obstacle DAPF | 54 |
| Gambar 4.6 Jarak Agen 2 ke Setiap Obstacle DAPF | 54 |
| Gambar 4.7 Jarak Agen 3 ke Setiap Obstacle DAPF | 54 |
| Gambar 4.8 Jarak Agen 4 ke Setiap Obstacle DAPF | 55 |
| Gambar 4.9 Jarak Agen 5 ke Setiap Obstacle DAPF | 55 |
| Gambar 4.5 Jarak Agen 1 ke Setiap Obstacle MAPF..... | 55 |
| Gambar 4.6 Jarak Agen 2 ke Setiap Obstacle MAPF..... | 56 |
| Gambar 4.7 Jarak Agen 3 ke Setiap Obstacle MAPF..... | 56 |
| Gambar 4.8 Jarak Agen 4 ke Setiap Obstacle MAPF..... | 56 |
| Gambar 4.9 Jarak Agen 5 ke Setiap Obstacle MAPF..... | 57 |
| Gambar 4.10 Jarak Antar Agen | 57 |
| Gambar 4.11 Jarak Agen ke Target terhadap Waktu..... | 57 |
| Gambar 4.12 Simulasi Pengujian 15 Obstacle Tersebar dalam Lintasan..... | 58 |
| Gambar 4.13 Tampilan 3D Pengujian 15 Obstacle Tersebar dalam Lintasan..... | 58 |
| Gambar 4.14 Jarak Agen 1 ke Setiap Obstacle | 59 |
| Gambar 4.15 Jarak Agen 2 ke Setiap Obstacle | 59 |
| Gambar 4.16 Jarak Agen 3 ke Setiap Obstacle | 59 |
| Gambar 4.17 Jarak Agen 4 ke Setiap Obstacle | 60 |
| Gambar 4.18 Jarak Agen 5 ke Setiap Obstacle | 60 |
| Gambar 4.19 Jarak Agen ke Target terhadap Waktu..... | 60 |
| Gambar 4.20 Jarak Antar Agen | 61 |
| Gambar 4.21 Simulasi Pengujian 25 Obstacle Tersebar dalam Lintasan..... | 61 |
| Gambar 4.22 Tampilan 3D Pengujian 25 Obstacle Tersebar dalam Lintasan..... | 62 |
| Gambar 4.23 Jarak Agen 1 ke Setiap Obstacle | 62 |
| Gambar 4.24 Jarak Agen 2 ke Setiap Obstacle | 62 |
| Gambar 4.25 Jarak Agen 3 ke Setiap Obstacle | 63 |
| Gambar 4.26 Jarak Agen 4 ke Setiap Obstacle | 63 |
| Gambar 4.27 Jarak Agen 5 ke Setiap Obstacle | 63 |
| Gambar 4.28 Jarak Agen ke Target terhadap Waktu..... | 64 |
| Gambar 4.29 Jarak Antar Agen | 64 |
| Gambar 4.30 Simulasi Pengujian 35 Obstacle Tersebar dalam Lintasan..... | 65 |
| Gambar 4.31 Tampilan 3D Pengujian 35 Obstacle Tersebar dalam Lintasan..... | 65 |

| | |
|--|----|
| Gambar 4.32 Jarak Agen 1 ke Setiap Obstacle | 65 |
| Gambar 4.33 Jarak Agen 2 ke Setiap Obstacle | 66 |
| Gambar 4.34 Jarak Agen 3 ke Setiap Obstacle | 66 |
| Gambar 4.35 Jarak Agen 4 ke Setiap Obstacle | 66 |
| Gambar 4.36 Jarak Agen 5 ke Setiap Obstacle | 67 |
| Gambar 4.37 Jarak Agen ke Target terhadap Waktu..... | 67 |
| Gambar 4.38 Jarak Antar Agen | 68 |
| Gambar 4.39 Respons Agen Obstacle dari Depan (Sebelum)..... | 69 |
| Gambar 4.40 Respons Agen Obstacle dari Depan (Saat)..... | 69 |
| Gambar 4.41 Respons Agen Obstacle dari Depan (Setelah)..... | 69 |
| Gambar 4.42 Jarak Agen 1 ke Setiap Obstacle | 70 |
| Gambar 4.43 Jarak Agen 2 ke Setiap Obstacle | 70 |
| Gambar 4.44 Jarak Agen 3 ke Setiap Obstacle | 71 |
| Gambar 4.45 Jarak Agen 4 ke Setiap Obstacle | 71 |
| Gambar 4.46 Jarak Agen 5 ke Setiap Obstacle | 71 |
| Gambar 4.47 Jarak Agen ke Target..... | 72 |
| Gambar 4.48 Jarak Antar Agen | 72 |
| Gambar 4.49 Respons Agen Obstacle dari Samping (Sebelum)..... | 73 |
| Gambar 4.50 Respons Agen Obstacle dari Samping (Saat)..... | 73 |
| Gambar 4.51 Respons Agen Obstacle dari Samping (Sesudah)..... | 73 |
| Gambar 4.52 Jarak Agen 1 ke Setiap Obstacle | 74 |
| Gambar 4.53 Jarak Agen 2 ke Setiap Obstacle | 74 |
| Gambar 4.54 Jarak Agen 3 ke Setiap Obstacle | 74 |
| Gambar 4.55 Jarak Agen 4 ke Setiap Obstacle | 75 |
| Gambar 4.56 Jarak Agen 5 ke Setiap Obstacle | 75 |
| Gambar 4.57 Jarak antar Agen dan Target..... | 75 |
| Gambar 4.58 Jarak Antar Agen | 76 |
| Gambar 4.59 Respons Agen Obstacle dari Belakang (Sebelum) | 76 |
| Gambar 4.60 Respons Agen Obstacle dari Belakang (Saat) | 77 |
| Gambar 4.61 Respons Agen Obstacle dari Belakang (Sesudah)..... | 77 |
| Gambar 4.62 Jarak Agen 1 ke Setiap Obstacle | 77 |
| Gambar 4.63 Jarak Agen 2 ke Setiap Obstacle | 78 |
| Gambar 4.64 Jarak Agen 3 ke Setiap Obstacle | 78 |
| Gambar 4.65 Jarak Agen 4 ke Setiap Obstacle | 78 |
| Gambar 4.66 Jarak Agen 5 ke Setiap Obstacle | 79 |
| Gambar 4.67 Jarak Agen ke Target | 79 |
| Gambar 4.68 Jarak Antar Agen | 79 |
| Gambar 4.69 Respons Agen saat Obstacle dari Depan Pertama | 80 |
| Gambar 4.70 Respons Agen saat Obstacle dari Depan Kedua..... | 81 |
| Gambar 4.71 Lintasan Agen Akhir | 81 |
| Gambar 4.72 Jarak Agen 1 ke Setiap Obstacle | 82 |
| Gambar 4.73 Jarak Agen 2 ke Setiap Obstacle | 82 |
| Gambar 4.74 Jarak Agen 3 ke Setiap Obstacle | 82 |
| Gambar 4.75 Jarak Agen 4 ke Setiap Obstacle | 83 |
| Gambar 4.76 Jarak Agen 5 ke Setiap Obstacle | 83 |
| Gambar 4.77 Jarak Agen ke Target | 83 |
| Gambar 4.78 Jarak Antar Agen | 84 |
| Gambar 4.79 Respons Agen bertemu Obstacle dari Samping (pertama) | 85 |
| Gambar 4.80 Respons Agen bertemu Obstacle dari Samping (kedua) | 85 |

| | |
|---|-----|
| Gambar 4.81 Kesuluruhan Lintasan Agen untuk Obstacle dari Samping | 85 |
| Gambar 4.82 Jarak Agen 1 ke Setiap Obstacle | 86 |
| Gambar 4.83 Jarak Agen 2 ke Setiap Obstacle | 86 |
| Gambar 4.84 Jarak Agen 3 ke Setiap Obstacle | 87 |
| Gambar 4.85 Jarak Agen 4 ke Setiap Obstacle | 87 |
| Gambar 4.86 Jarak Agen 5 ke Setiap Obstacle | 87 |
| Gambar 4.87 Jarak Agen ke Target | 88 |
| Gambar 4.88 Jarak Antar Agen | 88 |
| Gambar 4.89 Respons Agen bertemu Obstacle dari Belakang (pertama) | 89 |
| Gambar 4.90 Respons Agen bertemu Obstacle dari Belakang (kedua) | 89 |
| Gambar 4.91 Kesuluruhan Lintasan Agen untuk Obstacle dari Belakang | 89 |
| Gambar 4.92 Jarak Agen 1 ke Setiap Obstacle | 90 |
| Gambar 4.93 Jarak Agen 2 ke Setiap Obstacle | 90 |
| Gambar 4.94 Jarak Agen 3 ke Setiap Obstacle | 91 |
| Gambar 4.95 Jarak Agen 4 ke Setiap Obstacle | 91 |
| Gambar 4.96 Jarak Agen 5 ke Setiap Obstacle | 91 |
| Gambar 4.97 Jarak Agen ke Target | 92 |
| Gambar 4.98 Jarak Antar Agen | 92 |
| Gambar 4.99 Lintasan Agen berjumlah sama dengan Target | 93 |
| Gambar 4.100 Pergantian Penugasan Agen | 93 |
| Gambar 4.102 Jarak Semua Agen ke Semua Target terhadap Waktu | 94 |
| Gambar 4.103 Lintasan Agen berjumlah lebih dari Target | 94 |
| Gambar 4.104 Pergantian Penugasan Agen | 95 |
| Gambar 4.106 Jarak Semua Agen ke Semua Target terhadap Waktu | 95 |
| Gambar 4.107 Respons Agen saat Obstacle dari Depan Pertama | 96 |
| Gambar 4.108 Respons Agen saat Obstacle dari Depan Kedua | 96 |
| Gambar 4.109 Kesuluruhan Lintasan Agen untuk Obstacle dari Depan DAPF | 96 |
| Gambar 4.110 Riwayat Penugasan DAPF | 97 |
| Gambar 4.111 Respons Agen saat Obstacle dari Depan Pertama MAPF | 97 |
| Gambar 4.112 Respons Agen saat Obstacle dari Depan Kedua MAPF | 97 |
| Gambar 4.113 Kesuluruhan Lintasan Agen untuk Obstacle dari Depan MAPF | 98 |
| Gambar 4.114 Riwayat Penugasan MAPF | 98 |
| Gambar 4.115 Jarak Agen ke Semua Target DAPF | 99 |
| Gambar 4.116 Jarak antar Agen DAPF | 99 |
| Gambar 4.117 Jarak antar Agen MAPF | 99 |
| Gambar 4.118 Jarak Agen ke Semua Target MAPF | 99 |
| Gambar 4.119 Jarak Agen 1 ke Setiap Obstacle DAPF | 100 |
| Gambar 4.120 Jarak Agen 2 ke Setiap Obstacle DAPF | 100 |
| Gambar 4.121 Jarak Agen 3 ke Setiap Obstacle DAPF | 101 |
| Gambar 4.122 Jarak Agen 4 ke Setiap Obstacle DAPF | 101 |
| Gambar 4.123 Jarak Agen 5 ke Setiap Obstacle DAPF | 101 |
| Gambar 4.124 Jarak Agen 1 ke Setiap Obstacle MAPF | 101 |
| Gambar 4.125 Jarak Agen 2 ke Setiap Obstacle MAPF | 102 |
| Gambar 4.126 Jarak Agen 3 ke Setiap Obstacle MAPF | 102 |
| Gambar 4.127 Jarak Agen 4 ke Setiap Obstacle MAPF | 102 |
| Gambar 4.128 Jarak Agen 5 ke Setiap Obstacle MAPF | 103 |

Halaman ini sengaja dikosongkan

DAFTAR TABEL

| | |
|--|----|
| Tabel 2.1 Hasil Penelitian Terdahulu | 28 |
| Tabel 3.1 Tabel Parameter Quadcopter | 41 |
| Tabel 3.2 Tabel Parameter Environment..... | 43 |
| Tabel 3.3 Parameter DAPF..... | 46 |
| Tabel 3.4 Spesifikasi Personal Computer..... | 48 |
| Tabel 4.1 Panjang lintasan masing-masing agen menuju target..... | 52 |
| Tabel 4.2 Jarak Lintasan Agen dengan 15 Obstacle Statis..... | 60 |
| Tabel 4.3 Jarak Lintasan Agen dengan 25 Obstacle Statis..... | 64 |
| Tabel 4.4 Jarak Lintasan Agen dengan 35 Obstacle Statis..... | 67 |
| Tabel 4.5 Jarak Lintasan Agen Obstacle Depan..... | 72 |
| Tabel 4.6 Jarak Lintasan Agen Obstacle Samping | 76 |
| Tabel 4.7 Jarak Lintasan Agen Obstacle Belakang | 80 |
| Tabel 4.8 Jarak Lintasan Agen | 84 |
| Tabel 4. 9 Jarak Lintasan Agen | 88 |
| Tabel 4. 10 Jarak Lintasan Agen | 92 |
| Tabel 4.11 Jarak Lintasan Agen | 95 |

DAFTAR SIMBOL

| Simbol | Keterangan |
|----------------------|--------------------------------------|
| m | Massa |
| F | Gaya |
| U_1 | Gaya <i>thrust</i> |
| U_2 | Gaya <i>roll</i> |
| U_3 | Gaya <i>pitch</i> |
| U_4 | Gaya <i>yaw</i> |
| ξ | Posisi linier quadcopter |
| θ | Posisi angular quadcopter |
| J | Momen inersia |
| τ | Torsi |
| ϕ, θ, ψ | Posisi sudut euler, roll, pitch, yaw |
| p, q, r | Kecepatan sudut euler |
| x, y, z | Posisi linier koordinat kartesian |
| ω | Kecepatan angular |
| X_B, Y_B, Z_B | Posisi quadcopter body frame |
| F_{att} | Gaya atraktif |
| F_{rep} | Gaya repulsif |
| F_{rot} | Gaya rotasi |

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Pesatnya perkembangan teknologi di berbagai sektor telah mendorong inovasi signifikan dalam pengembangan sistem otonom, termasuk *Unmanned Aerial Vehicle* (UAV) atau yang dikenal sebagai pesawat tanpa awak. UAV menawarkan berbagai keunggulan dibandingkan pesawat konvensional, terutama dari segi biaya produksi yang lebih efisien, fleksibilitas operasional yang tinggi, dan kemampuan bergerak yang adaptif (Shah Alam & Oluoch, 2021). Keunggulan ini menjadikan UAV sebagai platform yang sangat diminati dan telah diaplikasikan secara luas dalam berbagai bidang, mulai dari pemantauan lingkungan, pengiriman barang, pencarian hingga operasi militer (Geng et al., 2013; Hessel et al., 2019; Waharte & Trigoni, 2010).

Aplikasi drone yang semakin beragam menjadi teknologi yang menarik minat luas dan terus berkembang sehingga terbukti sangat ampuh bahkan dalam konflik di Ukraina (Prieto et al., 2023). Dalam konteks militer, salah satu jenis UAV yang mendapatkan perhatian khusus adalah UAV Kamikaze, atau sering disebut drone bunuh diri. UAV jenis ini dirancang secara spesifik untuk membawa muatan eksplosif dan menghancurkan target dengan cara menabrakkan diri (Kurnianto et al., 2024). Penggunaan UAV Kamikaze memberikan keuntungan strategis, seperti pengurangan risiko korban jiwa bagi operator dan biaya operasional yang lebih rendah dibandingkan penggunaan pesawat tempur berawak dalam misi serupa (Austin, 2010). Seiring dengan semakin kompleksnya medan pertempuran modern, kemampuan UAV Kamikaze untuk beroperasi secara efektif dalam skenario yang dinamis menjadi sangat krusial.

Meskipun demikian, implementasi UAV Kamikaze dalam operasi militer menghadapi sejumlah tantangan signifikan, terutama terkait navigasi, pengambilan keputusan, dan koordinasi, terlebih dalam skenario multi-UAV yang melibatkan banyak unit dalam lingkungan yang berubah-ubah. Salah satu permasalahan mendasar yang harus diatasi adalah kemampuan penghindaran rintangan dinamis. Dalam lingkungan pertempuran yang padat, UAV Kamikaze dituntut untuk mampu menavigasi melalui berbagai objek statis maupun dinamis seperti bangunan, pepohonan, bahkan UAV musuh yang bergerak. Kegagalan dalam menghindari rintangan dapat mengakibatkan UAV hancur sebelum mencapai target, yang pada akhirnya mengurangi efektivitas misi dan menimbulkan kerugian operasional.

Untuk menjawab tantangan tersebut, berbagai algoritma telah dikembangkan untuk meningkatkan kapabilitas navigasi dan pengambilan keputusan UAV Kamikaze. Salah satu metode yang menjanjikan adalah *Dynamic Artificial Potential Field* (DAPF). DAPF bekerja dengan memodelkan lingkungan sebagai medan potensial yang dinamis, di mana UAV diarahkan menuju target oleh gaya tarik (*attractive force*) dan dijauhkan dari rintangan oleh gaya tolak (*repulsive force*) (Wahab, 2022). Dengan pendekatan ini, UAV Kamikaze dapat menghitung jalur optimal sambil menghindari rintangan yang muncul secara tiba-tiba. Selain itu, dalam skenario multi-UAV, koordinasi dan pembagian tugas antar-UAV Kamikaze menjadi aspek krusial. Algoritma seperti *Distributed Assignment Switch* (DAS) memungkinkan UAV untuk secara mandiri dan terdesentralisasi memilih serta mengalokasikan target, sekaligus berkoordinasi dengan unit lain untuk menghindari redudansi dan meningkatkan efektivitas serangan.

Berdasarkan permasalahan yang telah diuraikan, tugas akhir ini berfokus pada pengembangan algoritma pembagian tugas terdistribusi multi-Kamikaze UAV dengan penghindaran halangan dinamis menggunakan *Artificial Potential Field* (APF). Tujuan utama dari penelitian ini adalah untuk menghasilkan algoritma yang dapat meningkatkan kinerja operasional UAV Kamikaze dalam berbagai aplikasi militer, dengan menekankan pada adaptasi navigasi terhadap

lingkungan yang berubah-ubah, kemampuan pengambilan keputusan yang mandiri, serta koordinasi antar-UAV yang terdesentralisasi.

1.2 Rumusan Masalah

Permasalahan pada UAV kamikaze mencakup tantangan signifikan dalam meningkatkan keberhasilan misi, terutama isu pembagian tugas yang dinamis dan penghindaran halangan berkecepatan tinggi. Gangguan terhadap *leader* menyebabkan kegagalan koordinasi, membuat sistem yang bergantung pada *leader (centralized)* rentan terhadap kegagalan. Oleh karena itu, diperlukan pendekatan desentralisasi tanpa *leader* yang mengandalkan komunikasi antar UAV untuk pembagian tugas yang optimal. Selain itu, UAV kamikaze menghadapi kesulitan dalam menghindari halangan dinamis seperti *interceptor* yang dapat menyebabkan kegagalan misi jika tidak diatasi dengan cepat. Pendekatan menggunakan *Dynamic Artificial Potential Field* (APF) dengan gaya repulsif dan batas aman yang beradaptasi dengan kecepatan dari halangan yang bergerak. Penelitian ini akan mengembangkan algoritma yang mengintegrasikan pembagian tugas terdistribusi dan penghindaran halangan dinamis menggunakan *Dynamic APF* untuk UAV kamikaze berhasil mencapai multi-target.

1.3 Batasan Masalah

Adapun batasan masalah dari pembuatan tugas ini adalah sebagai berikut:

1. Setiap UAV kamikaze dilengkapi dengan sensor yang mampu mendeteksi halangan dalam radius tertentu.
2. Setiap UAV dianggap memiliki pengetahuan penuh tentang lokasi dan kecepatan masing-masing UAV dalam sistem.
3. Implementasi simulasi dan pengujian dilakukan menggunakan aplikasi MATLAB.
4. Diasumsikan bahwa komunikasi antar UAV kamikaze berlangsung dengan baik tanpa adanya keterlambatan (*delay*).

1.4 Tujuan

Tujuan dari penelitian ini adalah mengembangkan algoritma untuk menghindari benturan dengan halangan berkecepatan tinggi serta dapat melakukan pembagian tugas bagi UAV kamikaze untuk menyerang banyak target, dengan mengintegrasikan penghindaran halangan dinamis menggunakan *Dynamic Artificial Potential Field* (APF).

1.5 Manfaat

Penelitian ini diharapkan dapat diterapkan dan dijadikan referensi dalam menyelesaikan permasalahan riil. Manfaat dari penelitian ini adalah UAV kamikaze dapat mencapai dan mengeksekusi banyak target, dengan mengurangi risiko kerusakan dan kegagalan misi.

BAB 2 TINJAUAN PUSTAKA

2.1 Hasil Penelitian Terdahulu

Dalam beberapa tahun terakhir, penelitian mengenai sistem multiagen telah berkembang pesat, terutama dalam konteks aplikasi UAV. UAV telah digunakan secara luas dalam berbagai misi, mulai dari pengintaian dan pemetaan hingga operasi militer maupun digunakan untuk bantuan bencana. Salah satu tantangan utama dalam aplikasi UAV adalah memastikan UAV dapat beroperasi secara aman dalam lingkungan yang dapat dipenuhi dengan rintangan. Untuk itu, berbagai pendekatan telah diusulkan dan diuji dalam upaya mengatasi masalah ini, mulai dari perencanaan jalur dan pembagian tugas hingga penghindaran tabrakan secara *real-time*.

Dalam penelitian oleh Hage (2023), model quadcopter UAV digunakan sebagai dasar untuk pembagian tugas dan penghindaran rintangan. Pada penelitian ini digunakan algoritma Modified *Artificial Potential Field* untuk menghindari halangan statis berupa gedung. Pada penelitian oleh Hu et al. (2020) memperkenalkan algoritma *Distributed Assignment Switch* untuk kontrol formasi multiagen. Algoritma ini memungkinkan UAV untuk berbagi informasi dan berkoordinasi, memastikan bahwa setiap agen dapat menyelesaikan tugas yang diberikan tanpa perlu kendali terpusat. Selain itu, Du et al. (2019) mengembangkan strategi penghindaran tabrakan waktu nyata menggunakan algoritma *Dynamic Artificial Potential Field (Dynamic APF)*. Berbeda dengan algoritma APF konvensional yang kurang responsif terhadap perubahan lingkungan yang cepat, *Dynamic APF* mempertimbangkan dinamika dan pergerakan rintangan. Implementasi *Dynamic APF* memberikan kemampuan bagi UAV untuk merespons perubahan lingkungan secara *real-time* lebih baik dibandingkan dengan APF konvensional, memastikan jalur yang aman dan bebas dari tabrakan.

Tabel 2.1 Hasil Penelitian Terdahulu

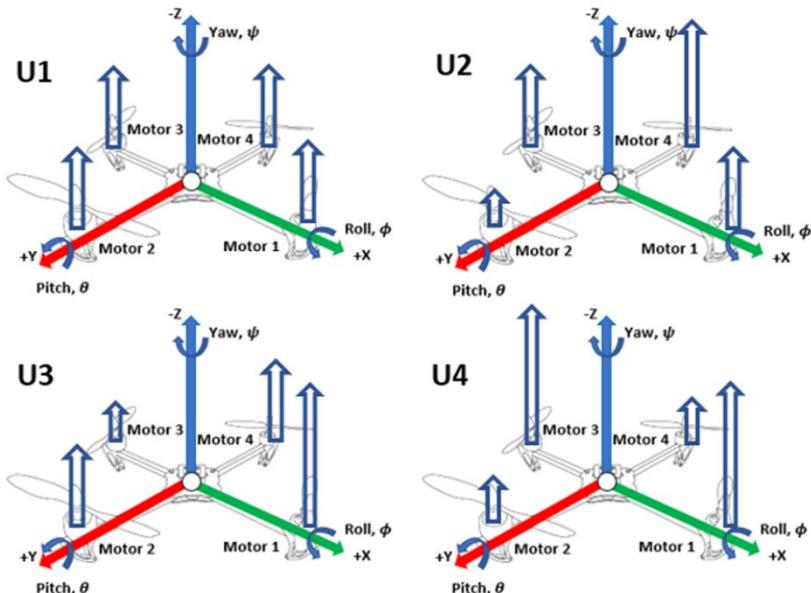
| Tahun | Judul Penelitian | Penulis | Keterangan |
|-------|---|--|---|
| 2023 | Kontrol Pembagian Tugas Multiagen Menuju Multitarget Dengan Penghindaran Halangan Menggunakan Artificial Potential Field | G. Hage | Penggunaan MAPF dan <i>Hybrid Task Allocation</i> untuk Pembagian Tugas Multi UAV dengan Penghindaran Halangan Statis |
| 2020 | <i>Convergent Multiagent Formation Control With Collision Avoidance</i> | J. Hu, H. Zhang, L. Liu, X. Zhu, C. Zhao, dan Q. Pan | Algoritma pembagian tugas terdistribusi untuk koordinasi dalam formasi multiagen. |
| 2019 | <i>A Real-Time Collision Avoidance Strategy in Dynamic Airspace Based on Dynamic Artificial Potential Field Algorithm</i> | Y. Du, X. Zhang, dan Z. Nie | Strategi penghindaran tabrakan waktu nyata dengan algoritma medan potensial buatan dinamis untuk mengatasi halangan yang dinamis. |

2.2 Dasar Teori

2.2.1 Quadcopter

Quadcopter merupakan salah satu jenis *Unmanned Aerial Vehicle* (UAV) yang menghasilkan daya angkat melalui empat rotor, dengan dua pasang rotor yang berputar saling berlawanan

arah. Komponen yang bergerak pada quadcopter hanyalah putaran *propeller*. Setiap *propeller* pada quadcopter digerakkan oleh motor, sebagai aktuator untuk menghasilkan gaya angkat. Untuk menjaga stabilitasnya saat terbang, biasanya keempat *propeller* berputar pada kecepatan yang sama, menggunakan motor dan *propeller* yang serupa (Magnussen & Skjønhaug, 2011). Namun, untuk mengatasi efek torsi yang dapat mempengaruhi gerakan horizontal, dua *propeller* dipasang berputar searah jarum jam dan dua lainnya berputar berlawanan arah jarum jam. Setiap poros pada quadcopter memiliki propeller yang berputar dalam arah yang seragam. Saat terbang, quadcopter dapat bergerak bebas terbang dalam 3 dimensi. Dimensi pergerakan dari quadcopter ini disimbolkan dengan arah *pitch* (θ), arah *roll* (Φ), dan arah *yaw* (Ψ). Arah *pitch* mewakili gerak quadcopter pada sumbu Y, arah *roll* mewakili gerak pada sumbu X, dan arah *yaw* mewakili gerak pada sumbu Z (Asti et al., 2020). Dalam gerak quadcopter, terdapat istilah kondisi *hover* atau melayang. Kondisi tersebut dihasilkan dengan motor depan dan belakang bergerak searah jarum jam, dan motor kanan dan kiri bergerak berlawanan arah jarum jam. Dari kondisi *hover* ini, terdapat beberapa gerakan yang dapat dihasilkan dengan mengatur kecepatan motor (Bresciani, 2008), antara lain dapat dilihat pada Gambar 2.1, terlihat bahwa quadcopter memiliki empat jenis gerakan: *roll*, *pitch*, *yaw*, dan *thrust*.



Gambar 2.1 Pergerakan Quadcopter

Kemudian, akan diuraikan analisis penurunan persamaan matematika untuk quadcopter. Analisis ini berasal dari pemahaman kinematika dan dinamika quadcopter. Hasil penurunan persamaan model matematika ini akan merepresentasikan semua gerakan quadcopter, dan persamaan yang dihasilkan akan berfungsi sebagai panduan untuk merancang kontroler yang sesuai dengan tujuan yang diinginkan.

Kinematika merupakan salah satu cabang ilmu mekanika yang mempelajari gerakan suatu objek tanpa mempertimbangkan gaya yang menyebabkannya. Dalam konteks quadcopter, analisis kinematika mencakup gerakan *roll*, *pitch*, dan *yaw* yang merujuk pada koordinat B-frame. Untuk menentukan posisi linear terhadap bumi, diperlukan sebuah matriks transformasi dari koordinat B-frame ke koordinat E-frame. Dengan demikian, posisi linear dan angular dapat didefinisikan sebagai berikut:

$$\begin{aligned}\xi^E &= [X \ Y \ Z]^T \\ \theta^E &= [\phi \ \theta \ \psi]^T\end{aligned}\tag{2.1}$$

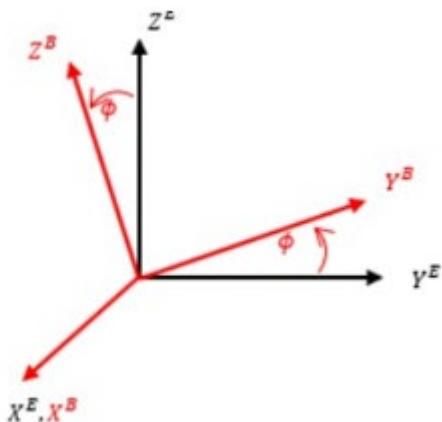
Dengan melakukan pengaturan kecepatan putaran *propeller* maka dihasilkan beberapa komando *input* sebagai berikut,

2.2.1.1 Pergerakan gaya *thrust* (U1)

Gaya *thrust* membuat quadcopter bergerak naik dan turun searah sumbu z. Jika ingin naik, gaya ini dihasilkan dengan keempat kecepatan motor yang sama, lalu kecepatan motor dipercepat. Begitu juga sebaliknya, quadcopter akan turun apabila kecepatan motor diperlambat secara bersamaan.

2.2.1.2 Pergerakan sudut *roll* (U2)

Pergerakan pada sudut *roll* adalah gerak quadcopter pada sumbu X. Gerak ini dipengaruhi oleh perubahan kecepatan motor bagian kiri dan kanan, sedangkan motor bagian depan dan belakang tetap berada pada kecepatannya. Apabila motor bagian kiri dipercepat, dan motor bagian kanan diperlambat, maka quadcopter akan bergerak ke kanan. Begitu juga sebaliknya, apabila motor bagian kanan dipercepat, dan motor bagian kiri diperlambat, maka quadcopter akan bergerak ke arah kiri.



Gambar 2.2 Ilustrasi Gerakan *Roll* pada Quadcopter

Berdasarkan gerak *roll* pada quadcopter didapatkan persamaan kinematika.

$$X^E = X^B \cos 0^\circ + Y^B \cos 90^\circ + Z^B \cos 90^\circ \quad (2.2)$$

$$Y^E = X^B \cos 90^\circ + Y^B \cos \phi + Z^B \cos(90^\circ + \phi)$$

$$Z^E = X^B \cos 90^\circ + Y^B \cos(90^\circ - \phi) + Z^B \cos \phi$$

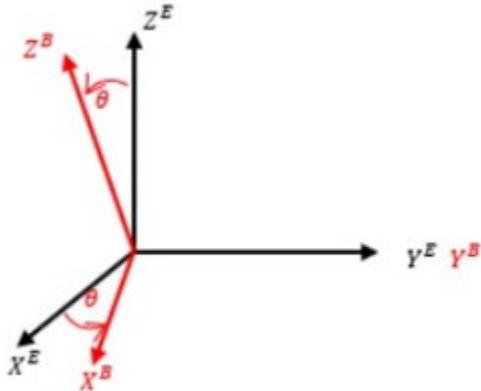
Persamaan (2.1) dapat diubah dalam bentuk matriks seperti (2.3) dan (2.4).

$$\begin{bmatrix} X^E \\ Y^E \\ Z^E \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi \\ 0 & \sin \phi & \cos \phi \end{bmatrix} \begin{bmatrix} X^B \\ Y^B \\ Z^B \end{bmatrix} \quad (2.3)$$

$$R^{BE}(\phi, x) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi \\ 0 & \sin \phi & \cos \phi \end{bmatrix}, \phi = \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] \quad (2.4)$$

2.2.1.3 Pergerakan sudut *pitch* (U3)

Pergerakan pada sudut *pitch* adalah gerak quadcopter pada sumbu Y. Gerak ini dipengaruhi oleh perubahan kecepatan motor bagian depan dan belakang, sedangkan motor bagian kanan dan kiri tetap berada pada kecepatannya. Apabila motor bagian belakang dipercepat, dan motor bagian depan diperlambat, maka quadcopter akan bergerak ke depan. Begitu juga sebaliknya, apabila motor bagian depan dipercepat, dan motor bagian belakang diperlambat, maka quadcopter akan bergerak ke arah belakang.



Gambar 2.3 Ilustrasi Gerakan *Pitch* pada Quadcopter
 $X^E = X^B \cos\theta + Y^B \cos 90^\circ + Z^B \cos(90^\circ - \theta)$ 2.5)

$$Y^E = X^B \cos 90^\circ + Y^B \cos 0^\circ + Z^B \cos 90^\circ$$

$$Z^E = X^B \cos(90^\circ + \theta) + Y^B \cos 90^\circ + Z^B \cos\theta$$

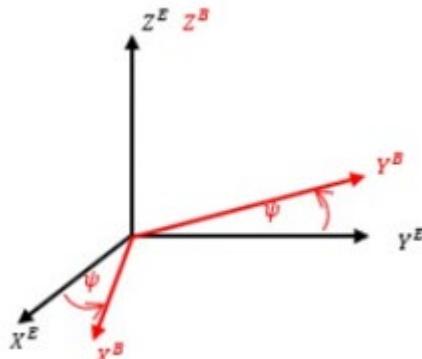
Persamaan (2.4) dapat diubah dalam bentuk matriks seperti (2.6) dan (2.7).

$$\begin{bmatrix} X^E \\ Y^E \\ Z^E \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\phi \end{bmatrix} \begin{bmatrix} X^B \\ Y^B \\ Z^B \end{bmatrix} \quad 2.6)$$

$$R^{BE}(\theta, y) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \theta = \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] \quad 2.7)$$

2.2.1.4 Pergerakan sudut *yaw* (U4)

Pergerakan pada sudut *yaw* adalah gerak quadcopter pada sumbu Z. Gerak ini dipengaruhi oleh perubahan pada semua kecepatan motor pada quadcopter. Apabila ingin memutar quadcopter berlawanan arah jarum jam, maka motor bagian depan dan belakang dipercepat, dan motor bagian kanan dan kiri diperlambat. Begitu juga sebaliknya, apabila ingin memutar quadcopter searah jarum jam, maka motor bagian kanan dan kiri dipercepat, dan motor bagian depan dan belakang diperlambat. Arah gerakan quadcopter akan bergerak searah dengan putaran motor yang diperlambat.



Gambar 2.4 Ilustrasi Gerakan *Yaw* pada Quadcopter

Berdasarkan gerak *yaw* pada quadcopter didapatkan persamaan kinematika seperti.

$$X^E = X^B \cos\psi + Y^B \cos(90^\circ + \psi) + Z^B \cos 90^\circ \quad (2.8)$$

$$Y^E = X^B \cos(90^\circ - \psi) + Y^B \cos\psi + Z^B \cos 90^\circ$$

$$Z^E = X^B \cos 90^\circ + Y^B \cos 90^\circ + Z^B \cos 0^\circ$$

Persamaan (2.7) dapat diubah dalam bentuk matriks seperti (2.9) dan (2.10).

$$\begin{bmatrix} X^E \\ Y^E \\ Z^E \end{bmatrix} = \begin{bmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X^B \\ Y^B \\ Z^B \end{bmatrix} \quad (2.9)$$

$$R^{BE}(\psi, z) = \begin{bmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \psi = \left[-\frac{\pi}{2}, \frac{\pi}{2} \right] \quad (2.10)$$

Untuk mendapatkan matriks transformasi R_{ξ}^{BE} dari koordinat B-frame ke koordinat E-frame maka analisa kinematika dari gerak *roll*, *pitch*, *yaw* dapat digabung kedalam satu matriks. Matriks R_{ξ}^{BE} diperoleh dengan mengalikan matriks rotasi dari tiap gerak quadcopter seperti pada persamaan berikut.

$$R_{\xi}^{BE} = R(\psi, z) R(\theta, y) R(\phi, x) \quad (2.11)$$

$$R_{\xi}^{BE} = \begin{bmatrix} c_{\psi}c_{\theta} & -s_{\psi}c_{\phi} + c_{\psi}s_{\theta}s_{\phi} & s_{\psi}s_{\phi} + c_{\psi}s_{\theta}c_{\phi} \\ s_{\psi}c_{\theta} & c_{\psi}c_{\phi} + s_{\psi}s_{\theta}c_{\phi} & -c_{\psi}s_{\phi} + s_{\psi}s_{\theta}c_{\phi} \\ -s_{\theta} & c_{\theta}s_{\phi} & c_{\theta}c_{\phi} \end{bmatrix}$$

Selain matriks transformasi R_{ξ}^{BE} , ada pula matriks transformasi T_{θ}^{BE} untuk mengubah besaran angular dari koordinat B-frame ke koordinat E-frame. Matriks transformasi T_{θ}^{BE} mengacu kecepatan Euler dalam B-frame dengan membalik pola perputaran sudut *roll*, *pitch*, dan *yaw*.

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = I \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + R(\phi, x)^{-1} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + R(\phi, x)^{-1} R(\theta, y)^{-1} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} \quad (2.12)$$

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = T_{\theta}^{BE}^{-1} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \quad (2.13)$$

Dengan menyelesaikan (2.12) maka didapatkan matriks transformasi T_{θ}^{BE} .

$$T_{\theta}^{BE}^{-1} = \begin{bmatrix} 1 & 0 & -\sin \theta \\ 0 & \cos \phi & \cos \theta \sin \phi \\ 0 & -\sin \phi & \cos \phi \cos \theta \end{bmatrix} = \begin{bmatrix} 1 & \sin \phi \tan \theta & \cos \phi \tan \theta \\ 0 & \cos \phi & -\sin \phi \\ 0 & \frac{\sin \phi}{\cos \theta} & \frac{\cos \phi}{\cos \theta} \end{bmatrix} \quad (2.14)$$

Sementara itu, dinamika adalah cabang ilmu mekanika yang mempelajari gerakan suatu objek dengan mempertimbangkan gaya yang memengaruhinya. Analisis dinamika dilakukan dengan menggunakan hukum Newton-Euler tentang gerak suatu benda. Gaya yang timbul akibat gerakan quadcopter dapat diungkapkan sebagai berikut:

$$\begin{bmatrix} F_{thrust} \\ F_{roll} \\ F_{pitch} \\ F_{yaw} \end{bmatrix} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} F_1 + F_2 + F_3 + F_4 \\ F_2 - F_4 \\ F_1 - F_3 \\ F_1 - F_2 + F_3 - F_4 \end{bmatrix} \quad (2.15)$$

Dimana gaya F_i merupakan gaya angkat yang dihasilkan oleh tiap *propeller* motor yang didefinisikan sebagai berikut:

$$F_i = K \frac{\omega}{s + \omega} u_i \quad (2.16)$$

Dengan K merupakan konstanta gaya tolak, ω bandwidth motor dan u adalah sinyal kontrol dari kontroler. Untuk memperoleh konstanta K maka dicari dengan menerangkan quadcopter pada posisi *hover*. Nilai K akan sebanding dengan total gaya angkat dari keempat motor quadcopter saat *hover*.

Pemodelan quadcopter dilakukan dengan menggunakan kombinasi koordinat *frame*, yang disebut sebagai Hybrid-*Frame* (*H-frame*). Penurunan model gerak translasi dilakukan terhadap koordinat bumi (*E-frame*), karena hal ini berkaitan dengan posisi dan kecepatan quadcopter terhadap bumi. Sementara itu, penurunan gerak rotasi dilakukan terhadap koordinat *body* (*B-frame*), karena gerak rotasi mempengaruhi pergerakan quadcopter itu sendiri. Dengan mengacu pada aksioma pertama Euler dari hukum II Newton, diperoleh persamaan gerak translasi sebagai berikut:

$$F_i = K \frac{\omega}{s + \omega} u_i \quad (2.17)$$

$$\sum F^E = \ddot{\xi}^E m$$

Dengan $\sum F^E$ merupakan resultan gaya yang bekerja pada quadcopter, m merupakan total massa quadcopter dan $\ddot{\xi}^E$ merupakan percepatan gerak quadcopter. Sehingga (2.16) dapat diubah menjadi (2.17).

$$F_f^E + F_g = \ddot{\xi}^E m \quad (2.18)$$

Dengan $\xi = [X \ Y \ Z]^T$ merupakan posisi quadcopter terhadap bumi, F_f^E merupakan resultan gaya pada tiap sumbu koordinat, dan $F_g = [0 \ 0 \ -mg]^T$ merupakan gaya gravitasi. Jika didefinisikan $F_f^B = [F_x \ F_y \ F_z]^T$ merupakan resultan gaya pada koordinat *B-frame*, maka harus ditransformasikan ke dalam koordinat bumi *E-frame*. Karena gaya yang bekerja hanya pada sumbu z maka resultan gaya yang muncul hanya F_z atau gaya *thrust* (U_1).

$$F_f^E = R_{\xi}^{BE} F_f^B \quad (2.19)$$

$$F_f^E = \begin{bmatrix} F_z(s_\psi s_\phi + c_\psi s_\theta c_\phi) \\ F_z(-c_\psi s_\phi + s_\psi s_\theta c_\phi) \\ F_z(c_\theta c_\phi) \end{bmatrix} = \begin{bmatrix} U_1(s_\psi s_\phi + c_\psi s_\theta c_\phi) \\ U_1(-c_\psi s_\phi + s_\psi s_\theta c_\phi) \\ U_1(c_\theta c_\phi) \end{bmatrix} \quad (2.20)$$

Substitusikan (2.19) ke (2.17), sehingga diperoleh

$$\begin{bmatrix} U_1(s_\psi s_\phi + c_\psi s_\theta c_\phi) \\ U_1(-c_\psi s_\phi + s_\psi s_\theta c_\phi) \\ U_1(c_\theta c_\phi) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} = \begin{bmatrix} \ddot{X} \\ \ddot{Y} \\ \ddot{Z} \end{bmatrix} m \quad (2.21)$$

Setelah (2.20) diselesaikan maka diperoleh persamaan gerak translasi quadcopter seperti pada (2.21).

$$\ddot{X} = (\sin \phi \sin \psi + \cos \phi \sin \theta \cos \psi) \frac{U_1}{m} \quad (2.22)$$

$$\ddot{Y} = (-\sin \phi \cos \psi + \cos \phi \sin \theta \sin \psi) \frac{U_1}{m}$$

$$\ddot{Z} = -g + (\cos \phi \cos \theta) \frac{U_1}{m}$$

Kemudian, berdasarkan aksioma kedua *Euler* pada hukum II Newton didapatkan persamaan gerak rotasi sebagai berikut:

$$\tau^E = J\ddot{\Theta}^E \quad (2.23)$$

Persamaan (2.23) merupakan persamaan di koordinat *E-frame*, sehingga perlu ditransformasi ke dalam koordinat *B-frame*. Jika persamaan tersebut dibawa ke dalam koordinat *B-frame* menjadi:

$$T_\theta \tau^B = J(T_\theta \omega^B) \quad (2.24)$$

$$T_\theta \tau^B = J(T_\theta \omega^B + T_\theta \dot{\omega}^B)$$

Jika turunan dari matriks transformasi T_θ adalah $T_\theta S(\omega^B)$, dengan $S(\omega^B)$ adalah matriks *skew-symmetric*, maka persamaan (2.23) dapat diubah menjadi (2.25).

$$T_\theta \tau^B = J(T_\theta S(\omega^B) \omega^B + T_\theta \dot{\omega}^B) \quad (2.25)$$

$$T_\theta \tau^B = T_\theta (S(\omega^B) J \omega^B + J \dot{\omega}^B)$$

$$T_\theta \tau^B = T_\theta (\omega^B \times J \omega^B + J \dot{\omega}^B)$$

Karena pada kedua ruas terdapat matriks transformasi T_θ , maka dapat dihilangkan atau persamaan (2.24) dapat dianggap sudah ditransformasikan ke dalam koordinat *B-frame*.

$$\tau^B = \omega^B \times J \omega^B + J \dot{\omega}^B \quad (2.26)$$

$$J \dot{\omega}^B = -(\omega^B \times J \omega^B) + \tau^B$$

Dengan $J = \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix}$ merupakan matriks diagonal yang berisi momen inersia tiap sumbu, $\omega = [p \ q \ r]^T = [\dot{\phi} \ \dot{\theta} \ \dot{\psi}]^T$ merupakan kecepatan sudut quadcopter, $\tau = [U_2 l \ U_3 l \ U_4 d]^T$ torsi yang bekerja pada quadcopter, l jarak motor terhadap pusat massa, dan d konstanta *drag* quadcopter. Untuk mempermudah proses penghitungan maka diselesaikan dulu perkalian *cross product* didalam kurung.

$$J \dot{\omega}^B = -(\omega^B \times J \omega^B) + \tau^B \quad (2.27)$$

$$\begin{bmatrix} J_{xx}\dot{p} \\ J_{yy}\dot{q} \\ J_{zz}\dot{r} \end{bmatrix} = \begin{bmatrix} (J_{yy} - J_{zz})qr \\ (J_{zz} - J_{xx})pr \\ (J_{xx} - J_{yy})pq \end{bmatrix} + \begin{bmatrix} U_2 l \\ U_3 l \\ U_4 d \end{bmatrix}$$

$$\dot{p} = \frac{J_{yy} - J_{zz}}{J_{xx}} qr + \frac{U_2 l}{J_{xx}} \quad (2.28)$$

$$\dot{q} = \frac{J_{zz} - J_{xx}}{J_{yy}} pr + \frac{U_3 l}{J_{yy}}$$

$$\dot{r} = \frac{J_{xx} - J_{yy}}{J_{zz}} pq + \frac{U_4 d}{J_{zz}}$$

Dengan mengambil persamaan gerak translasi (2.21) dan persamaan gerak rotasi (2.27), maka keseluruhan model dinamika quadcopter dapat dinyatakan sebagai:

$$\ddot{X} = (\sin \phi \sin \psi + \cos \phi \sin \theta \cos \psi) \frac{U_1}{m} \quad (2.29)$$

$$\ddot{Y} = (-\sin \phi \cos \psi + \cos \phi \sin \theta \sin \psi) \frac{U_1}{m}$$

$$\ddot{Z} = -g + (\cos \phi \cos \theta) \frac{U_1}{m}$$

$$\dot{p} = \frac{J_{yy} - J_{zz}}{J_{xx}} qr + \frac{U_2 l}{J_{xx}}$$

$$\dot{q} = \frac{J_{zz} - J_{xx}}{J_{yy}} pr + \frac{U_3 l}{J_{yy}}$$

$$\dot{r} = \frac{J_{xx} - J_{yy}}{J_{zz}} pq + \frac{U_4 d}{J_{zz}}$$

X, Y, Z merupakan posisi quadcopter dan p, q, r merupakan kecepatan *roll* (ϕ), *pitch* (θ), dan *yaw* (ψ)

2.2.2 State feedback controller

State feedback controller merupakan metode yang digunakan dalam teori sistem kontrol feedback (umpang balik) untuk menempatkan pole loop tertutup dari suatu plant di lokasi yang telah ditentukan di bidang s (Nurjanah et al., 2022). Sedangkan feedback linearization merupakan pendekatan umum yang digunakan untuk mengendalikan sistem nonlinier. Pendekatan ini melibatkan transformasi sistem non linier menjadi sistem linier yang ekuivalen melalui perubahan variabel dan input kontrol yang sesuai. Feedback linierization dapat diterapkan pada sistem non linier dengan bentuk:

$$\begin{aligned}\dot{x} &= f(x) + g(x)u \\ y &= hx\end{aligned}\tag{2.30}$$

dengan $x \in \mathbb{R}^n$ adalah vektor *state*, $u \in \mathbb{R}^p$ adalah vektor masukan (*input*), dan $y \in \mathbb{R}^m$ adalah vektor keluaran (*output*). Tujuannya adalah untuk mengembangkan *control input* u sebagai berikut,

$$\begin{aligned}u &= a(x) + b(x)v \\ v &= -Kx\end{aligned}\tag{2.31}$$

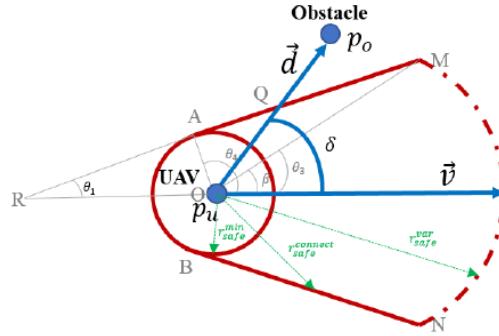
yang membuat peta *input-output* linier antara input baru v . Perubahan variabel

$$z = T(x)\tag{2.32}$$

yang mengubah sistem non linier menjadi sistem linier ekuivalen. Strategi kontrol *outer loop* untuk sistem kontrol linier yang dihasilkan kemudian dapat diterapkan..

2.2.3 Dynamic Artificial Potential Field (DAPF)

Dynamic Artificial Potential Field (DAPF) adalah algoritma perencanaan jalur yang dirancang untuk meningkatkan keamanan navigasi UAV di lingkungan yang dinamis (Du et al., 2019). Algoritma ini bekerja dengan menciptakan medan potensial virtual di sekitar UAV, di mana tujuan menarik UAV dengan gaya tarik, sementara rintangan menghasilkan gaya tolak. UAV akan bergerak mengikuti jalur yang merupakan hasil dari resultan kedua gaya ini.



Gambar 2.5 Batas jarak aman dinamis

DAPF memperkenalkan beberapa peningkatan penting dibandingkan APF tradisional. Pertama, DAPF menggunakan ambang batas jarak aman yang dinamis, yang dapat disesuaikan secara *real-time* berdasarkan kecepatan serta posisi relatif UAV dan rintangan ($\vec{v} = v_u - v_o$ dan $\vec{d} = p_o - p_u$, serta kemampuan manuver UAV. Jarak aman dinamis dirumuskan sebagai:

$$r_{safe} = \begin{cases} r_{safe}^{min} + \frac{k_v}{k_a + w_{max}} * \|\vec{v}(t)\| * \cos \delta, & \text{jika } \delta \in [0, \beta) \cup (-\beta, 0] \\ |OM| * \frac{\sin(\beta - \theta_1)}{\sin \angle OQM}, & \text{jika } \delta \in [\beta, \frac{\pi}{2} + \theta_1] \cup [-\frac{\pi}{2} - \theta_1, -\beta] \\ r_{safe}^{min}, & \text{lainnya} \end{cases} \quad (2.33)$$

di mana r_{safe}^{min} adalah jarak aman minimum, w_{max} adalah tingkat belok maksimum UAV, dan δ adalah sudut antara vektor kecepatan relatif (\vec{v}) dan vektor jarak relatif (\vec{d}) antara UAV dan rintangan. Hal ini memungkinkan UAV untuk merespons secara dinamis terhadap perubahan lingkungan yang cepat dan menjaga jarak aman yang optimal dari rintangan. Selain itu, DAPF memisahkan gaya tolak menjadi dua komponen, yaitu gaya tolak posisi dan gaya tolak kecepatan:

$$F_{rep}^p = \begin{cases} k_p^{rep} * TH_{level} * -\frac{\vec{d}}{\|\vec{d}\|}, & \text{jika } \|\vec{d}\| < r_{safe} \text{ dan } \delta \in [0, \frac{\pi}{2}) \cup (-\frac{\pi}{2}, 0] < \frac{\pi}{2} \\ 0, & \text{lainnya} \end{cases} \quad (2.34)$$

$$F_{rep}^v = \begin{cases} k_v^{rep} * TH_{level} \left(-\frac{\vec{d}}{\|\vec{d}\|} * \frac{1}{\cos(\delta)} + \frac{\vec{v}}{\|\vec{v}\|} \right), & \text{jika } \|\vec{d}\| < r_{safe} \text{ dan } \delta \in [\frac{\pi}{2}, \frac{3\pi}{2}] \\ 0, & \text{lainnya} \end{cases} \quad (2.35)$$

$$F_{rep}^i = F_{rep}^p + F_{rep}^v \quad (2.36)$$

Gaya tolak posisi berfungsi untuk menjaga UAV tetap berada pada jarak aman dari rintangan, sedangkan gaya tolak kecepatan membantu UAV mengubah arah kecepatannya untuk menghindari tabrakan. Kedua gaya tolak tersebut mempertimbangkan Tingkat ancaman agen oleh halangan. DAPF juga membagi gaya tarik menjadi gaya tarik posisi dan gaya tarik kecepatan:

$$F_{att} = F_{att_p} + F_{att_v} \quad (2.37)$$

$$F_{att_p} = k_p (p_g(t) - p(t))$$

$$F_{att_v} = k_v (V_0 - v(t))$$

Gaya tarik posisi berfungsi untuk menarik UAV menuju tujuan sesuai dengan jalur yang telah direncanakan, sedangkan gaya tarik kecepatan membantu UAV mencapai kecepatan yang diinginkan. Ditambahkan juga gaya redaman untuk mencegah osilasi UAV saat kembali ke jalur yang telah direncanakan setelah menghindari rintangan. Gaya redaman ini berlawanan arah dengan kecepatan UAV dan besarnya sebanding dengan kecepatan UAV.

$$\rho_{damp} = -k_{damp} * v_u(t) \quad (2.38)$$

Gaya tolak dalam DAPF juga mengalami modifikasi dengan mempertimbangkan tingkat ancaman (*threat level*) dari halangan bergerak. Tingkat ancaman suatu rintangan terhadap UAV dihitung berdasarkan jarak relatif, kecepatan relatif, dan tren gerakan relatif antara UAV dan rintangan. Jika rintangan mendekati UAV, tingkat ancamannya akan meningkat, dan gaya tolak yang dibutuhkan untuk menghindar juga akan meningkat. Tingkat ancaman ini dihitung dengan persamaan berikut:

$$\vec{v} = v_u - v_o^i \quad (2.39)$$

$$\vec{d} = p_o^i - p_u \quad (2.40)$$

$$TH_{level} = \begin{cases} \left(\frac{1}{\|\vec{d}\|} - \frac{1}{r_{safe}} \right) * \|\vec{v}\| * \cos \delta, \\ 0 \end{cases} \quad (2.41)$$

Dimana v_o^i dan p_o^i adalah kecepatan dan posisi dari rintangan ke-i di sekitar UAV, \vec{v} adalah vektor kecepatan relatif antara UAV dan rintangan. \vec{d} adalah vektor jarak relatif antara UAV dan rintangan, yang mengarah dari UAV menuju rintangan. $\|\vec{v}\| * \cos \delta$ adalah komponen posisi relatif dari kecepatan relatif antara UAV dan rintangan, yaitu kecepatan mendekat antara dua objek. Jika rintangan dan UAV saling mendekat ($\cos(\delta)>0$), semakin cepat kecepatan mendekat, semakin tinggi tingkat ancaman rintangan, dan semakin besar gaya tolak yang diperlukan untuk membantu UAV menghindari tabrakan. Sebaliknya, jika $\cos(\delta)\leq 0$, kedua objek saling menjauh. Dalam hal ini, tingkat ancaman rintangan adalah nol, dan tidak diperlukan penghindaran tabrakan.

2.2.4 Algoritma pembagian tugas terdistribusi

Distributed Assignment Switch (DAS) adalah mekanisme yang memungkinkan agen-agen dalam sistem multi-agen untuk secara dinamis dan mandiri mengubah tujuan mereka (Hu et al., 2020). DAS sangat penting ketika agen-agen ini memiliki keterbatasan dalam berkomunikasi dan beroperasi dalam lingkungan yang terbatas. Mekanisme DAS dimulai dengan pertukaran informasi secara berkala antara agen-agen yang berada dalam jangkauan komunikasi terbatas (R_C). Informasi yang dipertukarkan meliputi posisi agen ($p_{i,k}$), tujuan yang sedang dituju ($\phi_{i,k}$), dan nomor prioritas tujuan ($\theta_{i,k}$). Posisi agen merupakan vektor yang menunjukkan lokasi dari agen dalam suatu lingkungan, tujuan adalah titik yang ingin dicapai, dan nomor prioritas merupakan bilangan bulat yang menunjukkan tingkat kepentingan tujuan (semakin kecil nomornya, semakin tinggi prioritasnya).

Setelah menerima informasi dari tetangga-tetangganya, setiap agen akan mengevaluasi apakah perlu untuk mengganti tujuannya saat ini. Keputusan ini didasarkan pada beberapa faktor, termasuk apakah jalur agen menuju tujuannya saat ini akan menyebabkan tabrakan dengan agen tetangga, dan apakah ada tujuan lain yang lebih baik dan lebih aman yang dapat dicapai. Jika agen mengidentifikasi adanya potensi tabrakan, ia akan mencari tujuan baru yang lebih aman.

Salah satu faktor yang digunakan yaitu menghitung $(B_{i,k})$, yaitu tetangga yang dapat menghalangi jalur mereka menuju tujuan saat ini. Secara matematis, didefinisikan sebagai:

$$B_{i,k} = \{j \in N_{i,k} | \phi_{i,k} \in S_{j,i,k}, \|p_{i,k} - p_{j,k}\| \leq R_{SWITCH}\} \quad (2.42)$$

di mana $N_{i,k}$ adalah himpunan tetangga agen i pada waktu k. $\|p_{i,k} - p_{j,k}\|$ adalah jarak Euclidean antara agen i dan j pada waktu k. R_{SWITCH} adalah ambang batas jarak untuk pergantian tujuan. Jika jarak antara dua agen kurang dari atau sama dengan R_{SWITCH} , dan tujuan agen i berada dalam wilayah tugas agen j ($S_{j,i,k}$), maka agen j dianggap sebagai tetangga pemblokir bagi agen i. Klasifikasi *blocker* selanjutnya dipecah menjadi dua subset yang memiliki karakteristik dan implikasi strategis yang berbeda. Subset pertama didefinisikan sebagai

$$B_{i,k}^+ = B_{i,k} \cap \{j \in N_{i,k} | \theta_{j,k} > \theta_{i,k}, p_{j,k} \notin A_{j,k}\} \quad (2.43)$$

yang mencakup *blocker* yang berada di luar tetangga atraktif mereka namun memiliki nomor prioritas yang lebih besar daripada agen i. Subset kedua adalah

$$B_{i,k}^{++} = B_{i,k} \cap \{j \in N_{i,k} | p_{i,k} \in A_{j,k}, p_{j,k} \in A_{j,k}\} \quad (2.44)$$

yang terdiri dari *blocker* yang berada dalam tetangga atraktif mereka dan secara simultan mengurung agen i. Tetangga atraktif didefinisikan sebagai

$$A_{i,k} = \{s \in S | \|s - \varphi_{i,k}\| < R_{ATTR}\} \quad (2.45)$$

yaitu region berbentuk hipersfera dengan radius R_{ATTR} yang berpusat pada destinasi $\varphi_{i,k}$. Parameter R_{ATTR} ditetapkan sedemikian rupa sehingga memenuhi kondisi

$$d^* = \min_{m,n \in V, m \neq n} \|q_m - q_n\| \geq 2R_{ATTR} \quad (2.46)$$

untuk menjamin bahwa tetangga atraktif dari destinasi yang berbeda tidak saling berpotongan. Untuk menangani situasi yang lebih kompleks dalam penugasan multi-target, algoritma mendefinisikan himpunan *auxiliary* yang memfasilitasi pengambilan keputusan pada kondisi *dependency* yang rumit.

$$D_{i,k} = \{j \in N_{i,k} | \theta_{j,k} < \theta_{i,k}, p_{j,k} \notin A_{j,k}, i \in B_{j,k}\} \cup \{j \in N_{i,k} | i \in G_{j,k}\} \quad (2.47)$$

Himpunan $D_{i,k}$ mengidentifikasi tetangga dengan priority lebih tinggi yang tidak berada dalam attractive neighborhood mereka namun menganggap agen i sebagai *blocker*. Himpunan $G_{i,k}$ yang lebih kompleks didefinisikan sebagai

$$\begin{aligned} G_{i,k} = \{j \in N_{i,k} | \theta_{j,k} > \theta_{i,k}, & \|p_{i,k} - p_{j,k}\| \leq R_{SWITCH}, \|p_{j,k} - p_{g,k}\| \\ & \leq R_{STOP}, j \in B_{g,k}, g \in B_{i,k}, p_{g,k} \in A_{g,k}\} \end{aligned} \quad (2.48)$$

Definisi ini mengidentifikasi situasi rantai *dependency transitive* dimana agen j dengan prioritas lebih rendah terhambat oleh agen g yang telah mencapai tetangga atraktif destinasi, menciptakan *cascading effect* yang memerlukan resolusi melalui pertukaran tugas. Aturan penentuan permintaan bertukar sebagai berikut. aturan 2 menetapkan bahwa jika kondisi $p_{i,k} \notin A_{i,k}, D_{i,k} = \emptyset$ dan $B_{i,k}^+ \cup B_{i,k}^{++} \neq \emptyset$ terpenuhi, maka agen i mengirimkan permintaan bertukar kepada agen j^* yang ditentukan melalui

$$j^* = \operatorname{argmin}_{j \in B_{i,k}^+ \cup B_{i,k}^{++}} \|\varphi_{i,k} - p_{j,k}\| \quad (2.49)$$

Kriteria minimasi jarak Euclidean ini memastikan efisiensi dalam redistribusi tugas dengan memilih tetangga yang secara spasial paling dekat dengan destinasi yang diinginkan. aturan 3 menyediakan mekanisme alternatif ketika aturan 2 tidak terpenuhi, yaitu jika $p_{i,k} \notin A_{i,k}, D_{i,k} = \emptyset, B_{i,k}^+ \cup B_{i,k}^{++} = \emptyset$, dan $G_{i,k} \neq \emptyset$, maka agen i mengirimkan permintaan bertukar kepada

$$j^* = \operatorname{argmin}_{j \in G_{i,k}} \|\varphi_{i,k} - p_{j,k}\| \quad (2.50)$$

Mekanisme *update* destinasi yang diatur dalam aturan 4 mengimplementasikan transaksi atomik yang memastikan konsistensi *state* dalam sistem terdistribusi. Ketika agen i menerima

banyak permintaan bertukar, algoritma menyeleksi tetangga j dengan nomor prioritas terkecil untuk melakukan pertukaran tugas dan mengeksekusi update $\varphi_{i,k+1} = \varphi_{j,k}, \theta_{i,k+1} = \theta_{j,k}$, diikuti dengan pengiriman reply message kepada agen j. Setelah menerima konfirmasi reply, agen j melakukan update $\varphi_{j,k+1} = \varphi_{i,k}, \theta_{j,k+1} = \theta_{i,k}$. Proses pertukaran bilateral ini mempertahankan invariant bahwa jumlah total agen dan destinasi tetap konstan, sambil memungkinkan redistribusi yang optimal berdasarkan kondisi lokal.

Algoritma mengimplementasikan mekanisme *mutual exclusion* yang mencegah terjadinya kondisi perlombaan dan *deadlock* dalam sistem terdistribusi. Aturan 5 menetapkan bahwa dalam semua kondisi lain dimana kriteria aturan 2 dan aturan 3 tidak terpenuhi, agen tidak mengirimkan permintaan bertukar. Lebih fundamental lagi, desain protokol memastikan bahwa jika agen memutuskan untuk mengirimkan permintaan bertukar, maka agen tersebut tidak akan menerima permintaan bertukar dari tetangga pada interval waktu yang sama. Properti *mutual exclusion* ini mencegah pembentukan *dependency cycles* yang dapat menyebabkan *deadlock*, seperti situasi dimana agen i mengirimkan permintaan kepada agen j sementara agen j secara simultan mengirimkan permintaan kepada agen i, yang berpotensi menciptakan *circular waiting* yang tidak dapat diselesaikan.

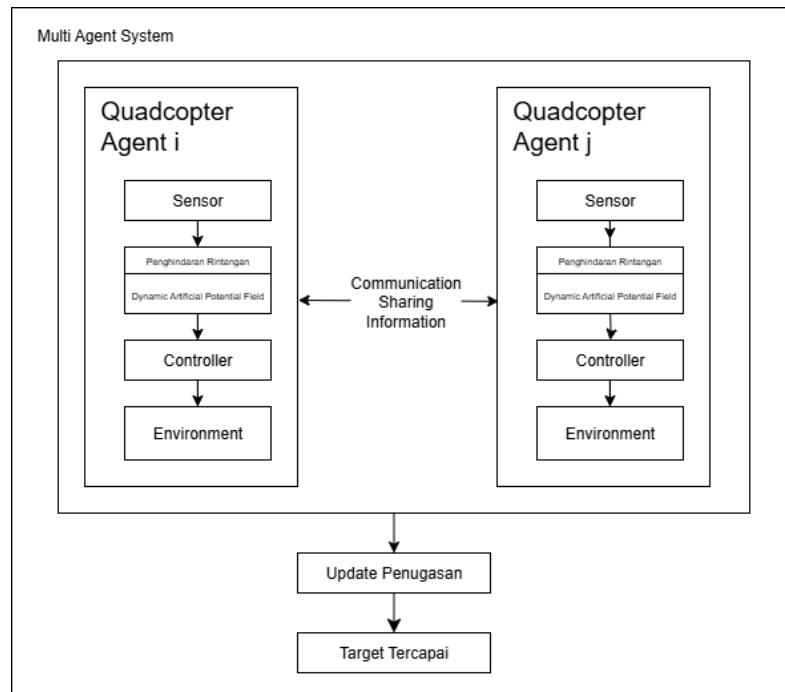
BAB 3 METODOLOGI

Penelitian ini menggunakan metode kuantitatif, yang merupakan proses sistematis yang berbasis pada prinsip-prinsip ilmiah untuk mengumpulkan data untuk tujuan dan keuntungan tertentu. Metode kuantitatif melibatkan penggunaan prosedur statistik berdasarkan hasil pengukuran untuk mendapatkan data yang diperlukan.

3.1 Metode yang Digunakan

Metodologi pada Tugas Akhir ini berfokus pada simulasi multi-UAV di lingkungan dinamis dengan dua tugas utama, yaitu pembagian tugas multi-target menggunakan Distributed Assignment Switch, serta penghindaran rintangan menggunakan Dynamic Artificial Potential Field (DAPF). Pendekatan ini dirancang agar setiap UAV dapat bekerja secara otonom dan kolaboratif, memanfaatkan desentralisasi untuk meningkatkan keandalan dan fleksibilitas sistem. Pada tahap pembagian tugas multi-target, setiap UAV menerapkan mekanisme Distributed Assignment Switch untuk saling bertukar informasi mengenai posisi target dan ancaman. Dengan demikian, tidak ada ketergantungan pada satu UAV sebagai leader, mengeliminasi risiko *single point of failure*, serta memastikan bahwa setiap agen dapat mengambil keputusan alokasi target berdasarkan kondisi misi secara *real-time*.

Pada metode DAPF, setiap target memunculkan gaya tarik (*attractive force*) yang membawa UAV menuju sasaran, sementara setiap halangan, baik statis maupun dinamis, menghasilkan dua jenis gaya tolak (*repulsive forces*). Pertama, gaya repulsif posisi yang besarnya berbanding terbalik dengan jarak ke rintangan, dan gaya repulsif kecepatan yang semakin besar apabila terdapat perbedaan kecepatan relatif tinggi antara UAV dan *obstacle* bergerak. Selain itu, radius aman DAPF bersifat dinamis, menyesuaikan parameter kecepatan dan posisi rintangan sehingga UAV mampu melakukan penghindaran rintangan secara adaptif dalam waktu nyata, khususnya terhadap halangan berkecepatan tinggi. Skema keseluruhan sistem dapat dilihat pada Gambar 3.1



Gambar 3.1 Skema Keseluruhan Sistem

3.1.1 Pemodelan sistem

3.1.1.1 Pemodelan quadcopter

Pada tugas akhir ini, tipe quadcopter yang digunakan adalah Quanser Qdrone. *Plant* ini telah banyak digunakan untuk penelitian yang dilakukan di luar ruangan. Quanser Qdrone mampu diaplikasikan meskipun memiliki risiko kerusakan yang tinggi dan mudah bermanuver karena memiliki kerangka badan dari serat karbon. Kerangka dengan dimensi $40\text{ cm} \times 40\text{ cm} \times 15\text{ cm}$ tersebut tahan lama dan ringan.



Gambar 3.2 Quanser Qdrone

Model dinamika quadcopter yang digunakan pada Persamaan 2.29. Quadcopter digunakan dengan parameter pada Tabel 3.1 (Agustinah et al., 2016).

Tabel 3.1 Tabel Parameter Quadcopter

| Parameter | Nilai |
|---|--------------------------------|
| Massa (m) | 1 kg |
| Percepatan gravitasi (g) | 9.81 m/s^2 |
| Momen inersia pada sumbu X (J_{xx}) | 0.03 kg. m^2 |
| Momen inersia pada sumbu Y (J_{yy}) | 0.03 kg. m^2 |
| Momen inersia pada sumbu Z (J_{zz}) | 0.04 kg. m^2 |
| Jarak rotor dari pusat massa (l) | 0.2 m |
| Konstanta <i>drag</i> (d) | $3.13 \times 10^{-5}\text{ N}$ |

Sehingga didapatkan persamaan model matematis quadcopter pada persamaan (2.29) dengan memasukkan nilai dari Tabel 3.2 sebagai berikut

$$\ddot{X} = (\sin \phi \sin \psi + \cos \phi \sin \theta \cos \psi)U_1 \quad (3.1)$$

$$\ddot{Y} = (-\sin \phi \cos \psi + \cos \phi \sin \theta \sin \psi)U_1$$

$$\ddot{Z} = -9.81 + (\cos \phi \cos \theta)U_1$$

$$\dot{p} = -0.333qr + 6.67 U_2$$

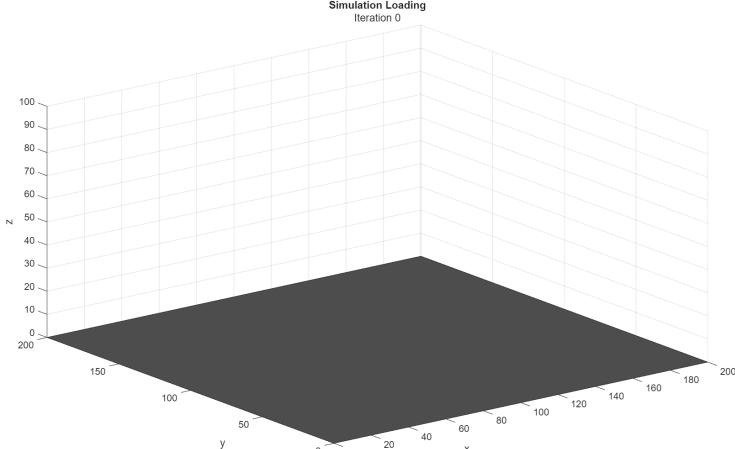
$$\dot{q} = 0.333pr + 6.67 U_3$$

$$\dot{r} = 7.825 \times 10^{-4} U_4$$

3.1.1.2 Pemodelan lingkungan

Pada simulasi Tugas Akhir ini, lingkungan kerja didefinisikan sebagai ruang kartesian tiga dimensi dengan batasan $x, y \in [0,200]\text{m}$ dan $z \in [0,100]\text{m}$. Untuk setiap eksperimen dapat ditentukan jumlah UAV sebanyak 5, yang akan memiliki posisi awal $p_{i(0)}$ dan kecepatan awal $v_{i(0)}$. Target dalam simulasi ditetapkan sebanyak M titik tujuan, divariasikan bergantung skenario, dengan setiap posisi target $p_{tgt}^j = [x_j, y_j, z_j]^T$ dipilih secara manual atau acak, dan

memiliki radius keberhentian r_{goal} sebagai ambang jarak minimum agar agen dianggap telah mencapai target.

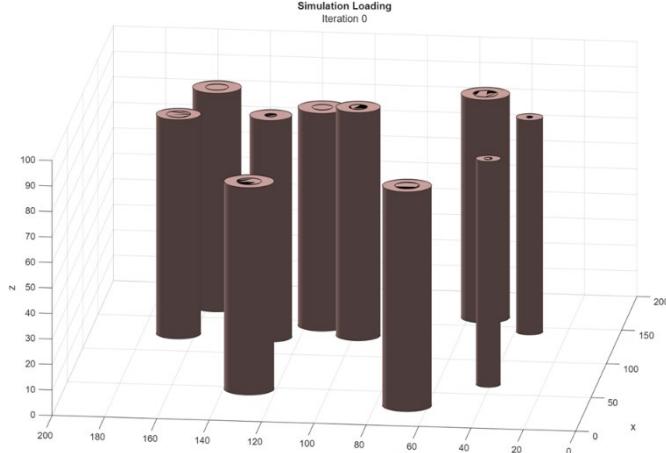


Gambar 3.3 Lingkungan Simulasi 200×200x100 unit

Sementara itu, *obstacle* statis dimodelkan sebagai silinder tegak dengan parameter pusat $c_i = [x_i, y_i, z_i]^T$, radius R_i , dan tinggi H_i . Suatu titik (x, y, z) termasuk di dalam volume silinder ke-i jika

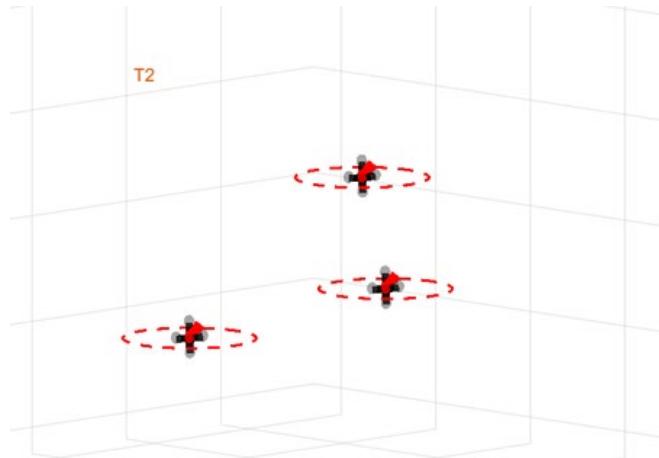
$$\begin{aligned} (x - x_i)^2 + (y - y_i)^2 &\leq R_i^2 \\ z &\leq z_i + H_i \end{aligned} \quad (3.2)$$

dengan banyaknya *obstacle* statis n_s yang dapat untuk menguji kemampuan agen dalam menghindar.



Gambar 3.4 Halangan Statis dengan variasi radius dan ketinggian

Halangan dinamis dimodelkan sebagai UAV Quadcopter lain dengan model 6DOF, dengan persamaan model pada persamaan (3.1). Posisi awal dari halangan dinamis diinisiasi sesuai skenario yang digunakan dan posisi tujuan p_{goal} ditetapkan acak di dalam area misi, dengan memastikan jalurnya memotong jalur agen-agen dalam mencapai target. Dengan halangan dinamis, kondisi terjadinya tumbukan antara agen dan halangan terjadi ketika $\|p_{UAV} - p_{UAVobs}\| \leq r_{quad}$, yang ditentukan berdasarkan dimensi quadcopter yang digunakan, sekitar 0.25m dan banyaknya *obstacle* dinamis n_d dapat diatur untuk menambah kompleksitas skenario. Kecepatan rintangan dinamis bernilai konstan sebesar 15 m/s dengan ditetapkan lebih cepat dibandingkan kecepatan *agen* yang sebesar 10 m/s.



Gambar 3.5 Halangan Dinamis dengan radius aman

Tabel 3.2 merangkum parameter *environment* yang akan digunakan

Tabel 3.2 Tabel Parameter *Environment*

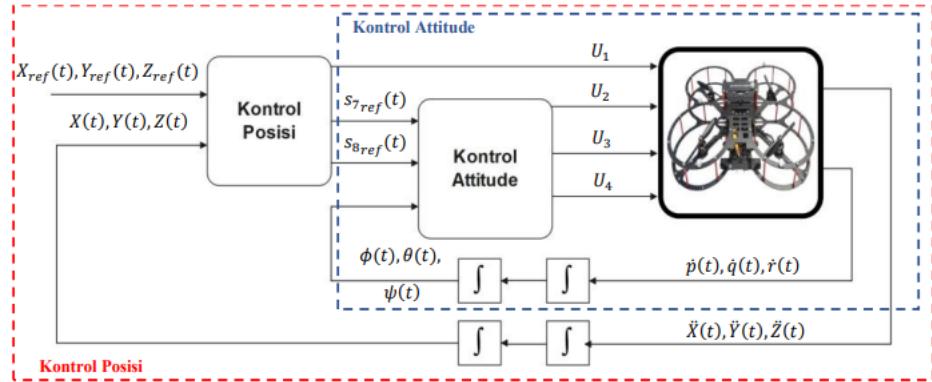
| Parameter | Nilai |
|---|----------------|
| Iterasi maksimal | 8000 iterasi |
| <i>Timestep</i> (dt) | 0.05 s |
| Rentang X | [0, 200] |
| Rentang Y | [0, 200] |
| Rentang Z | [0, 100] |
| Jumlah UAV | 5 agen |
| Jumlah target | 3-5 target |
| Radius penghentian (R_{Goal}) | 1 meter |
| Jumlah <i>obstacle</i> statis | 15-35 silinder |
| Jari-jari silinder | 3–15 meter |
| Tinggi silinder | 20–80 meter |
| Jumlah <i>obstacle</i> dinamis | 10-20 UAV |
| Jari-jari aman <i>obstacle</i> dinamis (r_{obsdvn}) | 0.25 meter |
| Kecepatan <i>obstacle</i> dinamis | 15 unit/s |

3.1.2 Perancangan dan Implementasi Sistem

3.1.2.1 Perancangan kontroler quadcopter

Skema kontrol quadcopter yang digunakan sebagai kontroler dibagi menjadi dua, yaitu kontrol posisi sebagai outer loop dan kontrol attitude sebagai inner loop yang dapat dilihat pada Gambar 3.6 (Nurjanah et al., 2022). Masukan pada kontrol posisi, merupakan nilai kesalahan dari posisi referensi $[X_{ref}(t) \ Y_{ref}(t) \ Z_{ref}(t)]^T$ terhadap posisi quadcopter yang sesungguhnya $[X(t) \ Y(t) \ Z(t)]^T$. Keluaran dari kontrol posisi adalah U_1 , $s_{7ref}(t)$, dan $s_{8ref}(t)$. Selanjutnya, $s_{7ref}(t)$, $s_{8ref}(t)$, posisi sudut quadcopter $[\phi(t) \ \theta(t) \ \psi(t)]^T$ menjadi masukan bagi kontrol attitude yang menghasilkan keluaran U_2 , U_3 , dan U_4 . State feedback controller digunakan untuk mengontrol ketinggian (Z) dan heading (ψ). Kombinasi state feedback controller dan fuzzy logic controller tipe Sugeno digunakan untuk mengontrol posisi X serta Y dan attitude ϕ serta θ . Sistem quadcopter (1, 2, ..., n) yang digunakan untuk semua quadcopter adalah identik. Secara keseluruhan fuzzy state feedback controller digunakan untuk mengontrol gerakan

quadcopter mulai dari titik UAV mulai bergerak menuju ke titik target yang dituju sesuai dengan lintasan yang telah dihasilkan.



Gambar 3.6 Blok Diagram Kontrol Quadcopter

Blok diagram untuk kontrol satu quadcopter dapat dilihat pada Gambar 3.6, ketika mendapat informasi target, maka quadcopter akan mencapai target tersebut menggunakan fuzzy state feedback controller yang dipakai sebagai kontrol posisi dan attitude. Untuk melakukan kontrol ketinggian, heading, posisi serta attitude dari quadcopter, persamaan (3.1) harus dimodifikasi dengan mempertimbangkan gaya gesek udara untuk kontrol ketinggian, nilai $\psi \rightarrow 0$ untuk kontrol heading. Sehingga diketahui persamaan yang digunakan dalam kontrollernya

$$\ddot{X} = \cos \phi \sin \theta \frac{U_1}{m} \quad (3.3)$$

$$\ddot{Y} = (-\sin \phi) \frac{U_1}{m} \quad (3.4)$$

$$\ddot{Z} = K_1(Z_{ref} - Z) + L_1(\dot{Z}_{ref} - \dot{Z}) + \ddot{Z}_{ref} \quad (3.5)$$

$$\ddot{s}_7 \approx \dot{p} = K_2 s_7_{ref} - K_2 s_7 - L_2 \dot{s}_7_{ref} - L_2 \dot{s}_7 + \ddot{s}_7_{ref} \quad (3.6)$$

$$\dot{r} = -K_4 \psi - L_4 \dot{\psi} \quad (3.7)$$

$$\ddot{s}_8 \approx \dot{q} = K_3 s_8_{ref} - K_3 s_8 - L_3 \dot{s}_8_{ref} - L_3 \dot{s}_8 + \ddot{s}_8_{ref} \quad (3.8)$$

dengan

$$\cos \phi \sin \theta = s_7 \quad (3.9)$$

$$\sin \phi = s_8 \quad (3.10)$$

Nilai parameter state feedback controller didapatkan melalui proses trial and error dengan memberikan titik-titik target atau waypoint, pada saat quadcopter berhasil mencapai semua titik target yang ditentukan, maka parameter dapat digunakan. Nilai parameter yang digunakan tertulis pada tabel 3.2.

Tabel 3.1 Nilai Parameter State Feedback Controller

| Parameter | Nilai | Parameter | Nilai |
|-----------|-------|-----------|-------|
| K_1 | 16 | L_1 | 9 |
| K_2 | 100 | L_2 | 21 |
| K_3 | 100 | L_3 | 21 |
| K_4 | 0.09 | L_4 | 0.61 |

Pada penelitian ini, fuzzy controller memiliki dua masukan, yaitu jarak dan kecepatan dari quadcopter menuju target. Sedangkan untuk keluarannya merupakan hasil dari sudut quadcopter. Diasumsikan jarak adalah selisih antara posisi (X dan Y) quadcopter saat ini

terhadap posisi targetnya. Jika selisih antara posisi quadcopter dan target lebih besar dari 25 meter maka masuk ke kategori far (jauh). Untuk kategori middle (sedang), jika jarak antara quadcopter dan target berada diantara 6 meter dan 25 meter. Untuk jarak yang lebih kecil dari 6 meter akan masuk ke kategori close (dekat). Masukan yang kedua adalah kecepatan, yaitu selisih antara kecepatan (\dot{X} dan \dot{Y}) quadcopter dan target. Membership function didesain sedemikian rupa sehingga jika selisih antara kecepatan quadcopter dan target lebih besar dari 4 m/s maka termasuk kategori fast (kencang). Sedangkan untuk kategori slow (pelan), jika selisih kecepatan quadcopter dan target lebih kecil dari 4 m/s. Keluaran yang diinginkan yang akan menjadi masukan untuk kontroler attitude (ϕ dan θ), dimana nilai keluaran adalah $-0.5 < ref < 0.5$. Dengan asumsi kemiringan maksimal UAV sebesar 30° .

Adapun rule base yang digunakan adalah,

1. Jika jaraknya jauh, $ref = 0.5$
2. Jika jaraknya sedang, dan kecepatannya kencang, $ref = 0.3$
3. Jika jaraknya dekat, dan kecepatannya kencang, $ref = -0.5$
4. Jika jaraknya dekat, dan kecepatannya pelan, $ref = 0$

3.1.2.2 Perancangan algoritma dynamic artificial potential fields

Dalam perancangan algoritma *Artificial Potential Field* (APF), baik dalam ruang dua dimensi maupun tiga dimensi, konsepnya yaitu setiap target yang terdefinisi baik statis maupun dinamis, akan menghasilkan gaya tarik yang membuat agen menuju dirinya. Sedangkan halangan statis (seperti gedung bangunan, pohon, dan kontur tanah) dan dinamis (UAV lain) akan menghasilkan gaya tolak yang menyebabkan agen menjauhi halangan-halangan tersebut, sehingga tidak menyebabkan tabrakan. Adapun dalam penelitian ini APF yang digunakan merupakan metode APF yang sudah dimodifikasi, Dynamic APF, untuk meningkatkan adaptabilitas terhadap lingkungan dinamis dan stabilitas jalur yang dihasilkan. Dikembangkan algoritma untuk melalui sebuah skenario bahwa quadcopter terbang dalam ruang tiga dimensi dan posisinya adalah $X = (x, y, z)^T$ dan Dynamic APF yang digunakan memiliki 3 komponen pada sumbu (x, y, z) .

Dalam skenario penghindaran rintangan statis, digunakan rentang radius *obstacle* 6-9 meter, maka ditetapkan parameter $r_{min}^{safe} = 30$ meter. Hal ini agar memastikan penghindaran rintangan untuk radius *obstacle* yang besar. Untuk skenario penghindaran halangan dinamis, digunakan parameter $r_{min}^{safe} = 5$ meter. Hal ini agar penghindaran rintangan tidak terlalu dini untuk *obstacle* dinamis yang ditetapkan yang memiliki kecepatan 15m/s. Dengan quadcopter dainggap memiliki kecepatan maksimal 18 m/s dan *pitch angle* maksimal 35° , ditetapkan parameter DAPF yaitu $\omega_{max} = 15^\circ/s$, $\beta = 60^\circ$, dan $\theta_1 = 15^\circ$ yang menetapkan kemampuan UAV untuk menghindari halangan dinamis, maupun statis dengan mendefinisikan batas pengindaran *head-on* dan *lateral*. Kemudian nilai-nilai tersebut akan digunakan pada persamaan 2.31 untuk menghitung r_{safe} untuk kemudian dilakukan perhitungan Gaya tolak posisi dan kecepatan berdasarkan tingkat ancaman yang bergantung dengan r_{safe} .

Ditetapkan sistem penghindaran rintangan berdasarkan ketinggian berdasarkan sifat hambatan: rintangan statis hanya ditangani secara horizontal, sedangkan rintangan dinamis memerlukan manuver tiga dimensi. Rintangan berbentuk tabung pada tugas akhir ini akan diperlakukan berbeda, untuk hambatan statis, gaya tolakan APF hanya dihitung pada bidang X-Y karena repulsif 2D sudah cukup mengamankan jalur terbang. Sebaliknya, ketika berhadapan dengan rintangan dinamis, repulsif 3D diaktifkan begitu $\rho(X) \leq \rho_0$ dan ketinggian pusat massa quadcopter ($z = 0,075$ m), memungkinkan drone menambah ketinggian untuk menghindar. Pengurangan 0.075 m, setengah tinggi Quanser Qdrone diperlukan karena ketinggian dihitung dari pusat masa, agar tubuh drone tidak menabrak meski baling-baling atau

kaki menonjol lebih rendah. Dengan skema ini 2D untuk statis, 3D untuk dinamis sistem penghindaran mampu memastikan keamanan terbang tanpa membebani sumber daya.

Pada simulasi yang dijalankan, selain menghindari halangan yang ada. Agen diharuskan untuk menghindari sesama agen dalam sistem multi-agenn, agar tidak terjadi tabrakan intra agen. Oleh karena itu ditetapkan $k_p^{repagent}$ dan $k_v^{repagent}$ dari agen bernilai setengah dari k_p^{rep} dan k_v^{rep} dari *obstacle*. Dilakukan modifikasi pada r_{safe} menjadi nilai parameter konstan pada persamaan pada gaya intra agen sebesar 1 meter digunakan untuk menjaga batas minimal antar UAV sekaligus mencegah adanya UAV yang terlalu berjauhan. Keluaran akhir dari algoritma penghindaran rintangan adalah gaya total kombinasi dari gaya atraktif dan repulsif.

$$F_{total}(X) = F_{att}(X) + F_{rep}(X) + F_{agent} \quad (3.11)$$

Gaya total ini memiliki tiga nilai, yaitu terhadap sumbu x, y, dan z. Kemudian gaya dijumlahkan dengan posisi quadcopter sekarang dan menjadi referensi posisi bagi kontroler penerbangan. Kemudian ditetapkan nilai-nilai parameter DAPF pada tabel 3.3

Tabel 3.3 Parameter DAPF

| Parameter | Nilai |
|---|------------|
| Konstanta Repulsif Posisi (k_p^{rep}) | 20 |
| Konstanta Repulsif Kecepatan (k_v^{rep}) | 10 |
| Konstanta Repulsif Posisi ($k_p^{repagent}$) | 10 |
| Konstanta Repulsif Kecepatan ($k_v^{repagent}$) | 5 |
| Konstanta Atraktif Posisi (k_p^{att}) | 1 |
| Konstanta Atraktif Kecepatan (k_v^{att}) | 1 |
| Konstanta Damping (k_{damp}) | 0.6 |
| Konstanta laju putar (k_a) | 10 |
| Laju putar maksimum (ω_{max}) | 14 |
| Koefisien laju putar (k_v) | 80 |
| Jarak aman minimum (Statis) | 30 meter |
| Jarak aman minimum (Dinamis) | 5 meter |
| Jarak aman agen | 1 meter |
| Radius UAV(r_u) | 0.25 meter |
| Desired Speed UAV (v_u) | 10 m/s |
| Sudut pusat CFR sektor (β) | 60° |
| Treshold Statis | 2 meter |
| Treshold Dinamis | 1 meter |

Algorithm 1 Dynamic Artificial Potential Field

```

1: procedure DYNAMIC_APF(p_obj, v_obj, obstacles, p_goal, v_goal)
2:   Initialize DAPF parameters (r_min_safe, k_v, k_a, w_max, k_rep, k_att,
   k_damp, β)
3:   F_total ← [0, 0, 0]
4:
5:   ▷ ===== REPULSIVE FORCE COMPUTATION =====
6:   for each obstacle_i ∈ obstacles do
7:     d_rel ← obstacle_i.position - p_obj
8:     v_rel ← v_obj - obstacle_i.velocity
9:     dist ← ||d_rel||
10:    δ ← angle_between(v_rel, d_rel)

```

```

11:           ▷ Dynamic safety distance calculation
12:           v_rel_component ← ||v_rel|| × cos(δ)
13:           if δ ∈ threat_sector then
14:               r_safe ← r_min_safe + (k_v/(k_a + w_max)) × max(0, v_rel_component)
15:           elseif δ ∈ lateral_zone then
16:               r_safe ← compute_lateral_safety_distance(v_rel_component)
17:           else
18:               r_safe ← r_min_safe
19:           end if
20:
21:           ▷ Threat level evaluation
22:           if obstacle_approaching(δ) then
23:               threat_level ← (1/dist - 1/r_safe) × ||v_rel|| × cos(δ)
24:           else
25:               threat_level ← 0
26:           end if
27:
28:           ▷ Generate repulsive force if in danger zone
29:           if dist < r_safe AND threat_level > 0 then
30:               if δ ∈ frontal_approach then
31:                   F_rep_pos ← k_rep_p × threat_level × (-normalize(d_rel))
32:               end if
33:               if δ ∈ lateral_approach then
34:                   steering_dir ← compute_steering_direction(d_rel, v_rel, δ)
35:                   F_rep_vel ← k_rep_v × threat_level × steering_dir
36:               end if
37:               F_total ← F_total + F_rep_pos + F_rep_vel
38:           end if
39:       end for
40:   end procedure

```

3.1.2.3 Perancangan algoritma Distributed Assignment Switch

Metode Distributed Assignment Switch atau DAS bekerja sepenuhnya terdistribusi. Setiap UAV menyiarkan paket informasi yang berisi posisi terkini, tujuan aktif, dan prioritas θ kepada tetangga dalam radius komunikasi. Sebelum misi dimulai penugasan awal dilakukan melalui mekanisme *market-based* yang mendukung kuota, setiap target j memiliki kapasitas Q_j , yang memastikan semua target dituju oleh minimal 1 agen. Sedangkan setiap agen i menghitung biaya C_{ij} yang mempertimbangkan jarak antara agen dan target kemudian menawar target dengan biaya terendah. Target menerima penawar terbaik hingga kuotanya terpenuhi, proses diulang sampai konvergen, dan urutan θ yang terbentuk dipakai sebagai prioritas lokal.

Selama penerbangan setiap UAV terus memeriksa kemungkinan lintasannya berpotongan dengan lintasan tetangga. Tetangga semacam itu dikategorikan sebagai *blocker* dan diurutkan menurut letak geometris terhadap zona atraktif serta prioritas θ . Permintaan pergantian tujuan dipicu hanya bila terpenuhi tiga syarat secara bersamaan yaitu jarak ke zona atraktif sudah lebih kecil dari R_{STOP} 1 m, terdapat *blocker* di depan, dan agen tersebut menghalangi tetangga dengan prioritas lebih tinggi. Permintaan dikirim ke *blocker* terdekat agar penambahan panjang lintasan sekecil mungkin. Protokol mutual exclusion menjamin satu agen tidak menerima dua permintaan pada saat yang sama sehingga deadlock akibat circular wait tidak muncul. Jika permintaan diterima kedua agen menukar tujuan dan prioritas secara atomik lalu menyiarkan pembaruan. Apabila grafik permintaan membentuk siklus agen dengan θ tertinggi dipaksa menuju target alternatif terdekat dan nilai θ ditambah sedikit supaya siklus tidak terulang. Nilai parameter ditetapkan sebagai berikut R_{SWITCH} 15 m menandai jarak mulai mempertimbangkan pergantian tugas, kemudian ditambahkan parameter $Commitment_{Distance}$ 10 m memastikan agen yang sudah dekat wajib menyelesaikan target, dan $Min_{ReassignBenefit}$ 5 m menjamin pertukaran tujuan hanya dilakukan jika pengurangan jarak cukup berarti.

Kombinasi penugasan awal berbasis *market-based* dengan kuota dan mekanisme pertukaran terkontrol selama penerbangan membuat DAS mampu mendistribusikan beban

secara adil, meminimalkan *deadlock*, serta menjaga jalur setiap UAV tetap efisien di lingkungan dinamis.

Algorithm 2 Distributed Assignment Switch

```

1: procedure DISTRIBUTED_ASSIGNMENT_SWITCH( $p_1 \dots p_N, \phi_1 \dots \phi_N$ )
2: for each agent  $i \in \{1..N\}$  do
3:    $N_i \leftarrow$  neighbors of  $i$  within  $R_{COMMS}$ 
4:    $B_i \leftarrow \{j \in N_i : \text{dist}(p_i, p_j) \leq R_{SWITCH}$ 
5:   AND  $\phi_i \in S_{\{j,i\}}$ 
6:    $B^+_i \leftarrow \{j \in B_i : \theta_j > \theta_i \text{ AND } p_j \notin A_j\}$ 
7:    $B^{++}_i \leftarrow \{j \in B_i : p_i \in A_j \text{ AND } p_j \in A_j\}$ 
8:    $G_i \leftarrow \{j \in N_i : \theta_j > \theta_i$ 
10:  AND  $\text{dist}(p_i, p_j) \leq R_{SWITCH}$ 
11:  AND  $\exists g : j \in B_g \text{ AND } p_g \in A_g$ 
12:  AND  $\text{dist}(p_j, p_g) \leq R_{STOP}\}$ 
14:
15: if  $(B^+_i \cup B^{++}_i \neq \emptyset)$  then  $\triangleright$  Rule 2
16:  $j^* \leftarrow \operatorname{argmin}_{\{j \in B^+_i \cup B^{++}_i\}} \text{dist}(\phi_i, p_j)$ 
17: sendSwitchRequest( $i \rightarrow j^*$ )
18:
19: elseif  $(G_i \neq \emptyset)$  then  $\triangleright$  Rule 3
20:  $j^* \leftarrow \operatorname{argmin}_{\{j \in G_i\}} \text{dist}(\phi_i, p_j)$ 
21: sendSwitchRequest( $i \rightarrow j^*$ )
22:
23: end if  $\triangleright$  Rule 5
24: end for
25:
26: upon agent  $j$  receiving requests  $R_j$  do
27:  $i^* \leftarrow \operatorname{argmin}_{\{i \in R_j\}} \theta_i \triangleright$  Rule 4
28: swap( $\phi_j, \phi_{\{i^*\}}$ )
29: end upon
30: end procedure

```

3.2 Bahan dan Peralatan yang Digunakan

Pada penelitian tugas akhir ini akan menggunakan beberapa *hardware* dan *software* untuk keperluan komputasi dan simulasi.

3.2.1 Personal computer

Pada penelitian tugas akhir ini akan dikerjakan dengan menggunakan spesifikasi *Personal Computer* (PC) pada Tabel 3.4:

Tabel 3.4 Spesifikasi *Personal Computer*

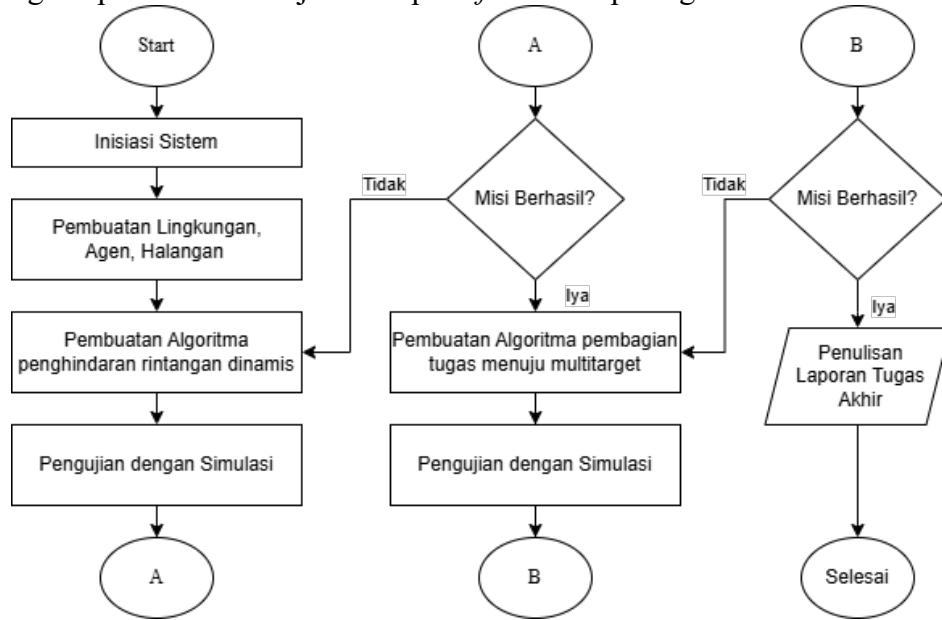
| Model Komputer | HP Pavilion Aero 13 |
|----------------|---|
| Processor | AMD Ryzen 5 5625U 2.30 GHz |
| RAM | 16 GB |
| Kartu Grafis | Integrated Graphics AMD Radeon Graphics |
| Penyimpanan | 512GB SSD |
| Sistem Operasi | Windows 11 |

3.2.2 MATLAB

MATLAB (*Matrix Laboratory*) adalah perangkat lunak komputasi numerik yang kuat dan serbaguna, banyak digunakan dalam penelitian dan pengembangan di berbagai bidang. Pada penelitian ini, MATLAB akan digunakan untuk merancang, mengimplementasikan algoritma *Dynamic Artificial Potential Field* dan *Distributed Assignment Switch*. Selain itu, MATLAB juga akan digunakan untuk menghasilkan visualisasi untuk dapat memahami kinerja sistem secara keseluruhan.

3.3 Urutan Pelaksanaan Penelitian

Tahapan pelaksanaan penelitian dalam tugas akhir ini mencakup beberapa langkah. Urutan langkah-langkah penelitian ini dijelaskan pada *flowchart* pada gambar 3.6



Gambar 3.7 Diagram Alir Pelaksanaan Tugas Akhir

3.3.1 Studi literatur

Tahap studi literatur dilakukan dengan melakukan tinjauan terhadap teori-teori yang relevan dan mengumpulkan data penelitian terkait perancangan sistem pembagian tugas dan penghindaran rintangan dinamis untuk multi-UAV kamikaze. Tinjauan ini mencakup studi mendalam tentang algoritma *Dynamic Artificial Potential Field* (DAPF), *Distributed Assignment Switch*, serta pemodelan dan kontrol UAV. Sumber-sumber yang digunakan dalam studi literatur ini dipastikan memiliki kredibilitas dan keakuratan informasi tinggi, seperti jurnal-jurnal IEEE, Google Scholar, serta penelitian-penelitian tugas akhir sebelumnya yang relevan baik ITS maupun perguruan tinggi lainnya.

3.3.2 Perancangan dan implementasi sistem

Pada tahap ini, quadcopter Quanser Qdrone dimodelkan sebagai *plant* dengan persamaan dinamik pada (2.29) dan parameter fisik pada Tabel 3.2, sedangkan lingkungan simulasi didefinisikan sebagai ruang 3D berukuran $200 \times 200 \times 100$ meter dengan himpunan target, *obstacle* statis berbentuk silinder, dan *obstacle* dinamis UAV lain sesuai Tabel 3.3. Selanjutnya algoritma Dynamic Artificial Potential Field (DAPF) diprogram untuk menghasilkan gaya total berupa kombinasi gaya atraktif menuju target dan gaya repulsif terhadap *obstacle* statis, *obstacle* dinamis, serta antar agen. Untuk koordinasi multi-UAV dan eliminasi konflik tujuan digunakan algoritma Distributed Assignment Switch yang menangani proses identifikasi *blocker* dan pertukaran destinasi secara terdistribusi. Terakhir, seluruh rancangan ini diimplementasikan dalam bentuk kode program (terlampir) sehingga sistem mampu melakukan navigasi multi-UAV secara aman dan adaptif.

3.3.3 Pengujian sistem dan evaluasi

Pengujian kinerja sistem dilaksanakan melalui berbagai skenario evaluasi yang dirancang untuk menganalisis kemampuan algoritma *dynamic APF* secara komprehensif. Pengujian penghindaran halangan statis dilakukan dengan konfigurasi *obstacle* tersebar acak. Evaluasi konfigurasi tersebar acak dibagi menjadi tiga sub-kasus berdasarkan tingkat kepadatan

halangan, yaitu 15 halangan yang menggambarkan kepadatan rendah, 25 halangan untuk kepadatan sedang, dan 35 halangan yang merepresentasikan kepadatan tinggi. Pengujian ini bertujuan menganalisis performa algoritma *dynamic* APF terhadap variasi kompleksitas lingkungan operasi dengan lintasan stasioner. Evaluasi penghindaran halangan dinamis dilakukan melalui tiga skenario pergerakan *obstacle*: konfigurasi *head-on* dimana *obstacle* bergerak berlawanan arah dengan UAV, konfigurasi side dimana *obstacle* bergerak dari samping melintasi jalur UAV, dan konfigurasi *back* dimana *obstacle* mengejar dari belakang. Simulasi pembagian tugas terdistribusi UAV diimplementasikan dalam dua konfigurasi: skenario dengan jumlah UAV dan target yang sama yaitu 5 agen dan 5 target untuk mengevaluasi efisiensi alokasi tugas seimbang, serta skenario dengan jumlah UAV dan target yang berbeda yaitu 5 agen dan 3 target untuk menganalisis kemampuan sistem dalam mengelola redundansi agen. Kemudian dilakukan pengujian keseluruhan sistem, dengan penghindaran lintangan serta pembagian tugas. Metrik evaluasi yang ditetapkan adalah panjang lintasan, waktu simulasi, dan jarak minimum.

3.3.4 Penyusunan laporan tugas akhir

Setelah penelitian telah selesai dilakukan, hasil-hasil yang diperoleh akan disusun menjadi sebuah laporan untuk keperluan Tugas Akhir dalam bentuk buku Tugas Akhir

BAB 4 HASIL DAN PEMBAHASAN

Pada bagian ini akan dibahas mengenai hasil dari rencana pengujian yang telah dibahas pada bab sebelum dengan menggunakan parameter yang telah ditentukan serta pembahasan terkait hasil simulasi yang telah dilakukan dengan konfigurasi yang berbeda.

4.1 Simulasi Perencanaan Lintasan UAV dengan *Artificial Potential Field*

Pada subbab ini akan dijelaskan bagaimana algoritma Artificial Potential Field (APF) digunakan untuk merencanakan lintasan UAV yang memengaruhi navigasi dalam berbagai skenario.

4.1.1 Penghindaran halangan statis

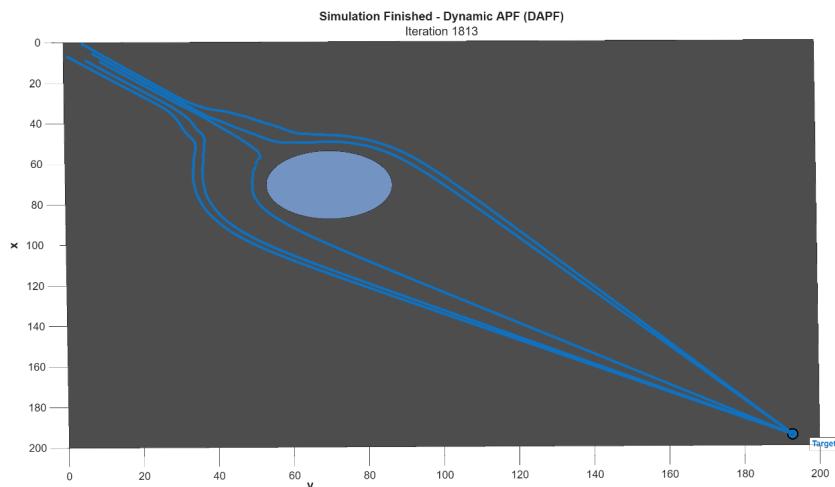
Pada subbab ini akan dilakukan simulasi penghindaran halangan statis, di mana UAV hanya menghadapi rintangan yang tidak bergerak sehingga APF cukup mempertimbangkan posisi *obstacle* dalam perhitungan lintasan.

4.1.1.1 Konfigurasi *obstacle* dalam lintasan menuju target

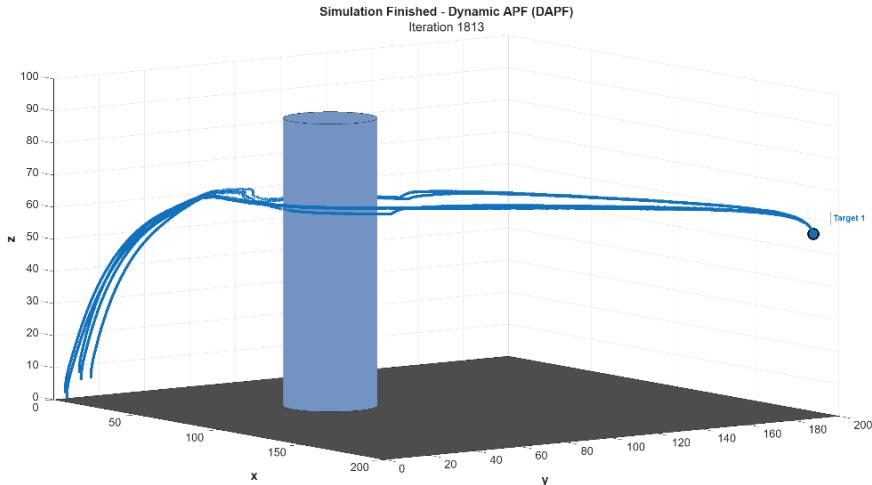
Pada skenario ini, kelima UAV diuji dalam konfigurasi dengan satu rintangan statis yang diposisikan tepat di sepanjang garis lurus antara masing-masing agen dan target. Gambar 4.1 menampilkan lintasan 2D dan Gambar 4.2 menampilkan lintasan 3D masing-masing UAV untuk algoritma DAPF, yang menunjukkan pola pergerakan melengkung untuk menghindari rintangan silindris tanpa terjadi tabrakan selama durasi simulasi 90.65 detik. Gambar 4.3 dan Gambar 4.4 menampilkan lintasan 2D dan 3D masing-masing UAV untuk algoritma MAPF, yang menunjukkan pola pergerakan yang kaku untuk menghindari rintangan silindris tanpa terjadi tabrakan selama durasi simulasi 86.5 detik. Panjang lintasan yang dilalui masing-masing agen dirangkum pada Tabel 4.11.

Tabel 4.1 Panjang lintasan masing-masing agen menuju target

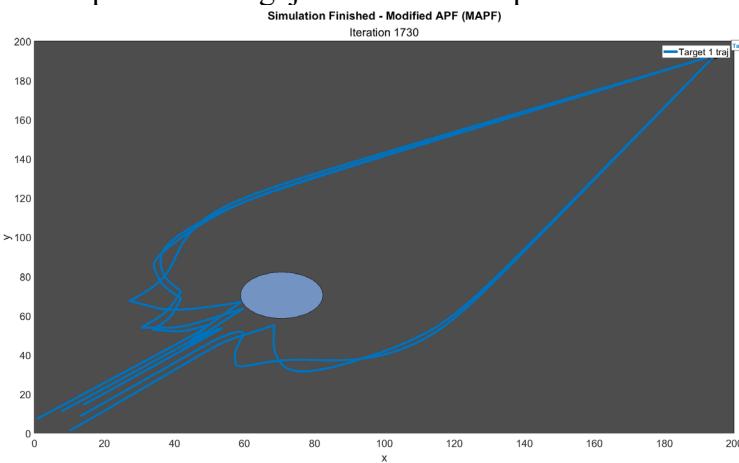
| Metode | Agen 1 | Agen 2 | Agen 3 | Agen 4 | Agen 5 | Total |
|--------|--------|--------|--------|--------|--------|----------|
| DAPF | 320.55 | 312.18 | 321.14 | 315.99 | 313.80 | 1 583.66 |
| MAPF | 334.56 | 331.53 | 354.17 | 383.91 | 360.97 | 1 765.14 |



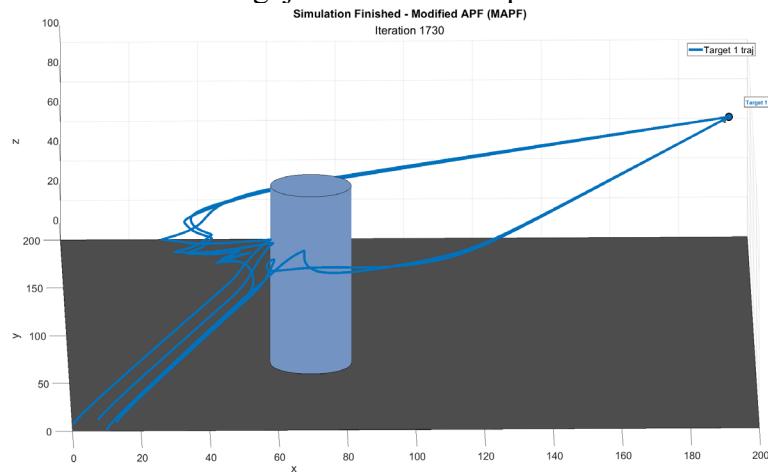
Gambar 4.1 Simulasi Pengujian *Obstacle* Terpusat dalam Lintasan DAPF



Gambar 4.2 Tampilan 3D Pengujian *Obstacle* Terpusat dalam Lintasan DAPF



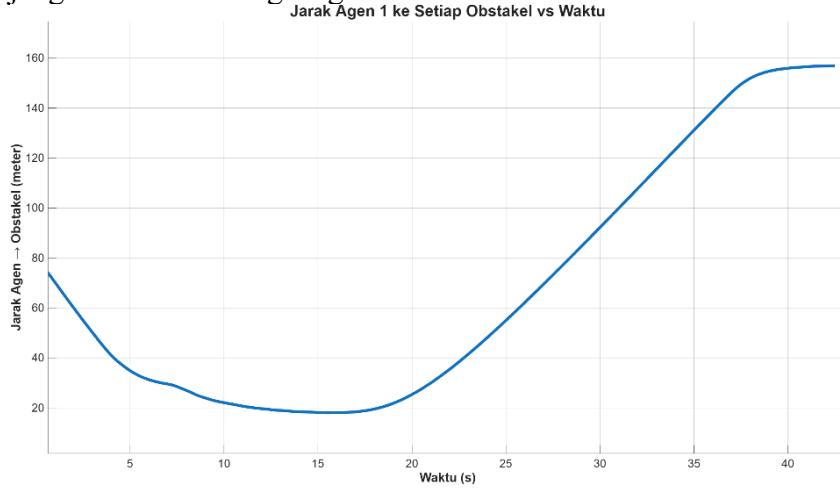
Gambar 4.3 Simulasi Pengujian *Obstacle* Terpusat dalam Lintasan MAPF



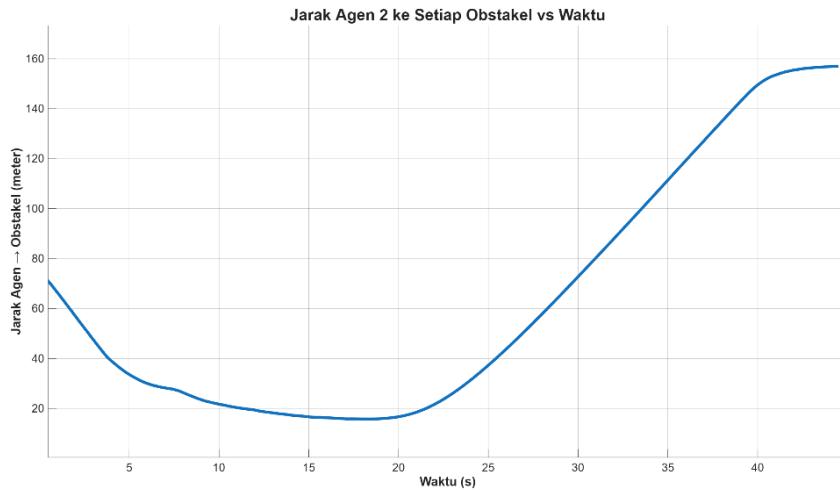
Gambar 4.4 Tampilan 3D Pengujian *Obstacle* Terpusat dalam Lintasan MAPF

Seluruh agen berhasil berhenti dalam radius toleransi $R_{STOP} = 1$ meter dengan rata-rata jarak akhir sebesar untuk DAPF 0.980 meter dan MAPF 0.999 meter. Gambar 4.5 hingga Gambar 4.9 menunjukkan grafik jarak masing-masing agen terhadap rintangan. Jarak minimum untuk Algoritma DAPF terhadap *obstacle* tidak pernah mencapai nol, yaitu 3.5 meter. Gambar 4.5 hingga Gambar 4.9 menunjukkan grafik jarak masing-masing agen terhadap rintangan. Jarak minimum untuk Algoritma MAPF terhadap *obstacle* tidak pernah mencapai nol, namun memiliki jarak minimum yang cukup ekstrem yaitu 0.5 meter. Hal ini menunjukkan bahwa tidak

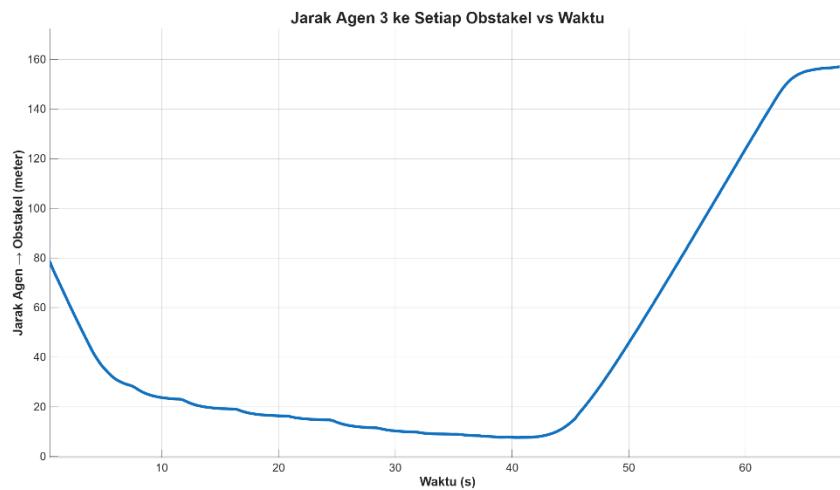
ada agen yang menyentuh rintangan dan untuk jarak minimum algoritma DAPF diatas batas *threshold* sepanjang simulasi berlangsung.



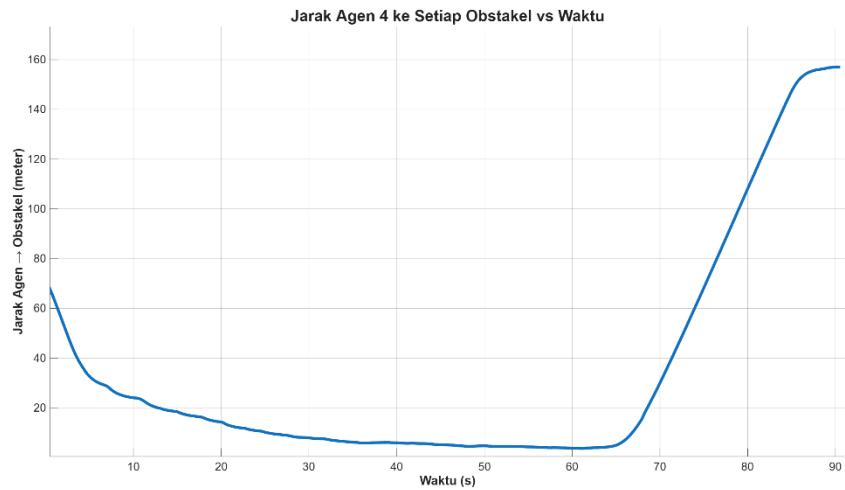
Gambar 4.5 Jarak Agen 1 ke Setiap *Obstacle* DAPF



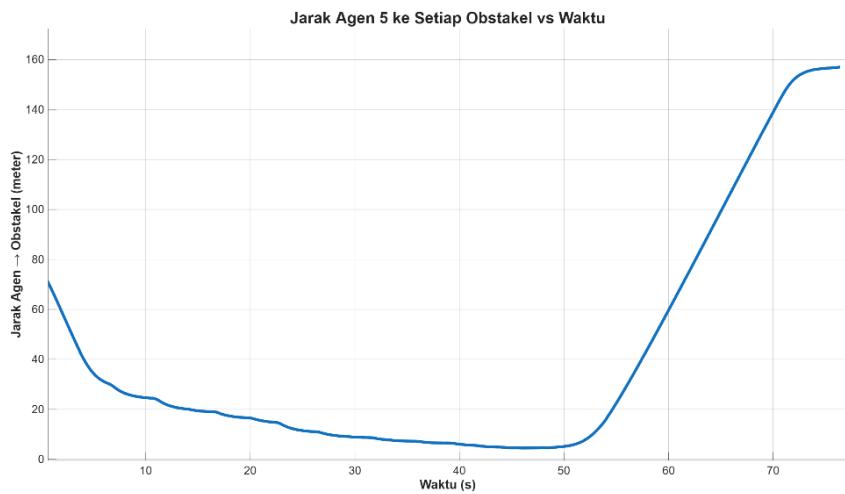
Gambar 4.6 Jarak Agen 2 ke Setiap *Obstacle* DAPF



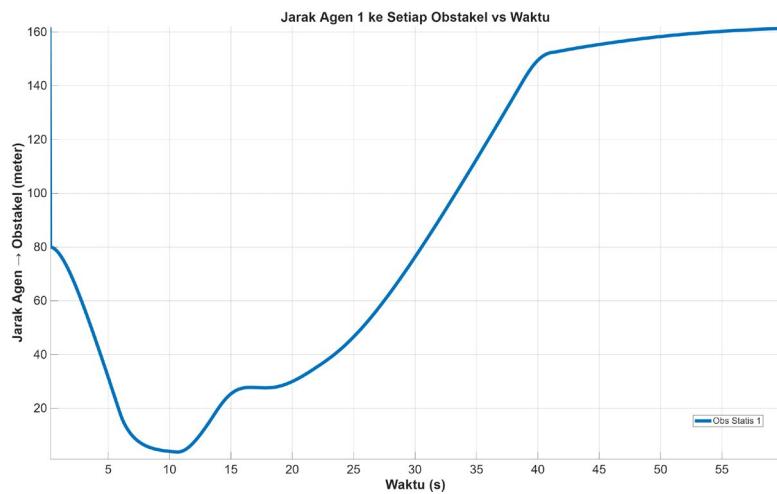
Gambar 4.7 Jarak Agen 3 ke Setiap *Obstacle* DAPF



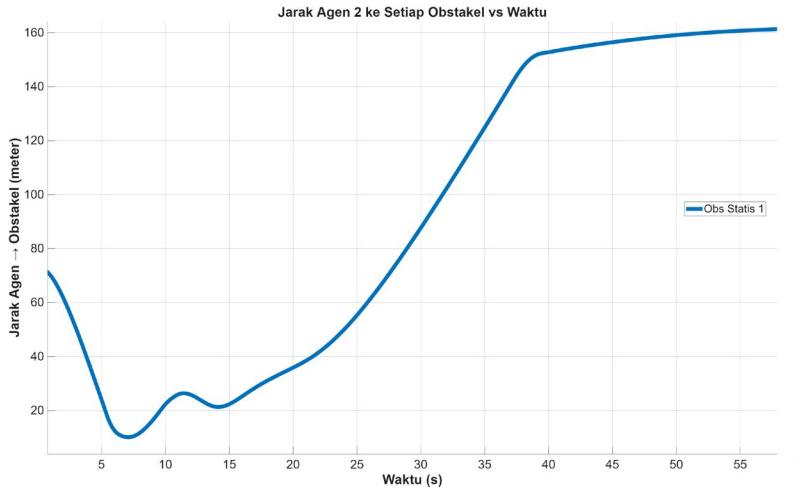
Gambar 4.8 Jarak Agen 4 ke Setiap *Obstacle* DAPF



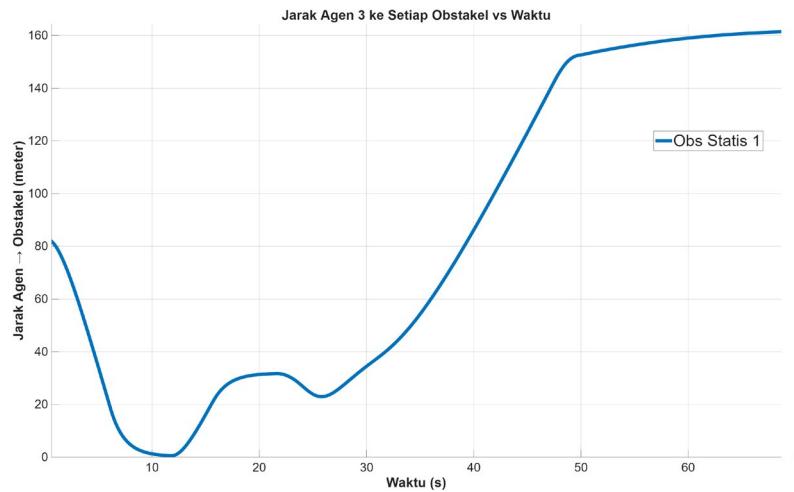
Gambar 4.9 Jarak Agen 5 ke Setiap *Obstacle* DAPF



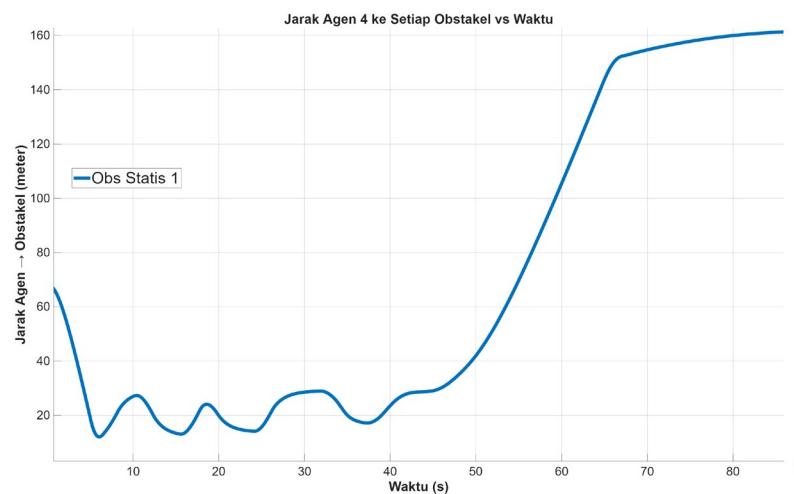
Gambar 4.10 Jarak Agen 1 ke Setiap *Obstacle* MAPF



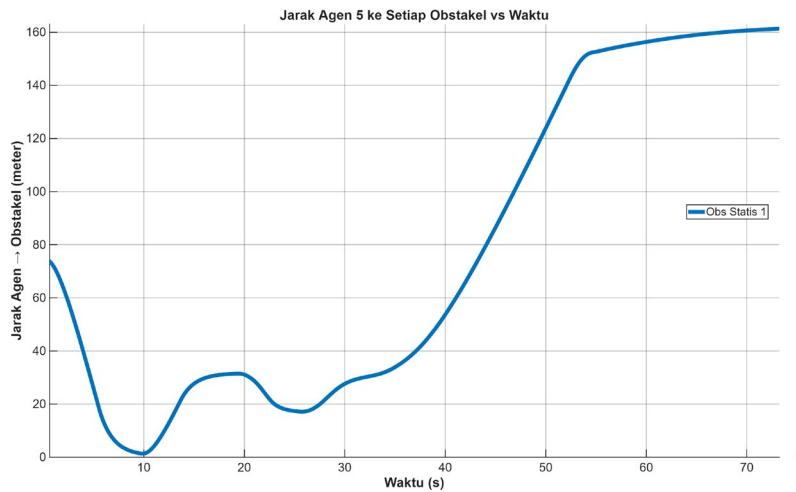
Gambar 4.11 Jarak Agen 2 ke Setiap *Obstacle* MAPF



Gambar 4.12 Jarak Agen 3 ke Setiap *Obstacle* MAPF

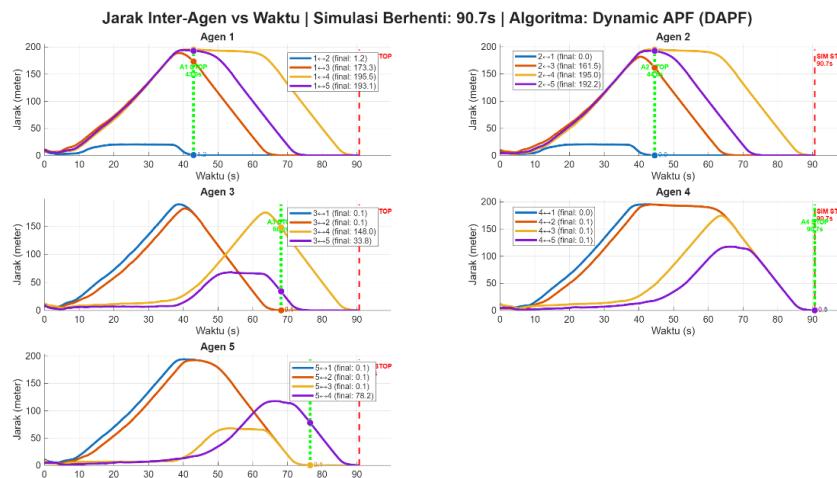


Gambar 4.13 Jarak Agen 4 ke Setiap *Obstacle* MAPF

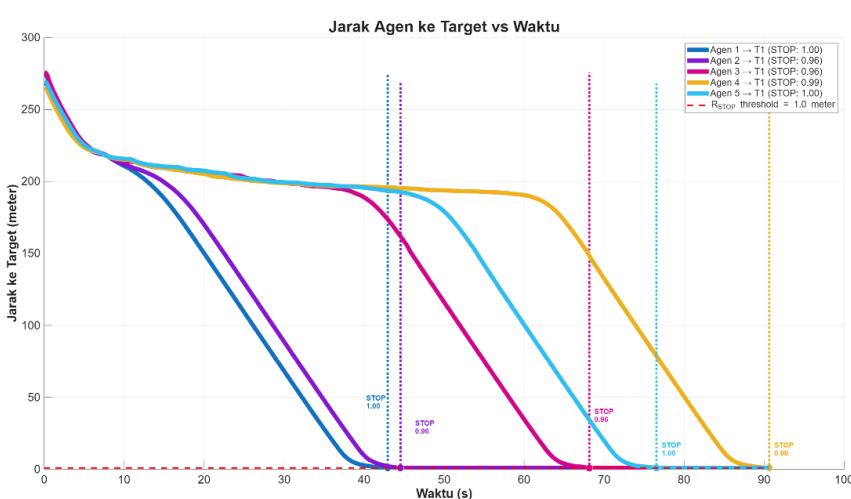


Gambar 4.14 Jarak Agen 5 ke Setiap *Obstacle* MAPF

Selanjutnya, Gambar 4.155 menunjukkan jarak antar-agen terhadap waktu. Kurva tidak pernah mencapai nol, yang bernilai minimal 1.6 meter, yang berarti tidak terjadi tabrakan antar-UAV. Gambar 4.16 memperlihatkan jarak masing-masing agen terhadap target seiring waktu, yang menunjukkan tren konvergen menuju titik



Gambar 4.15 Jarak Antar Agen



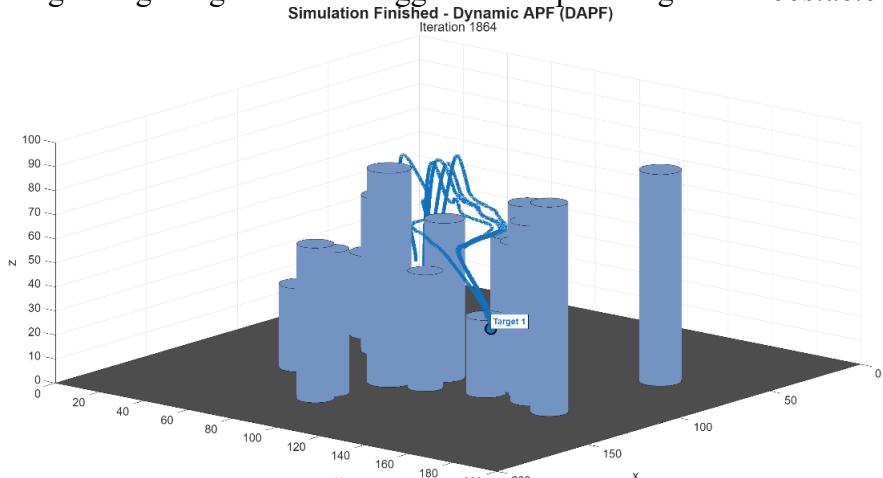
Gambar 4.16 Jarak Agen ke Target terhadap Waktu

4.1.1.2 Konfigurasi *obstacle* tersebar acak

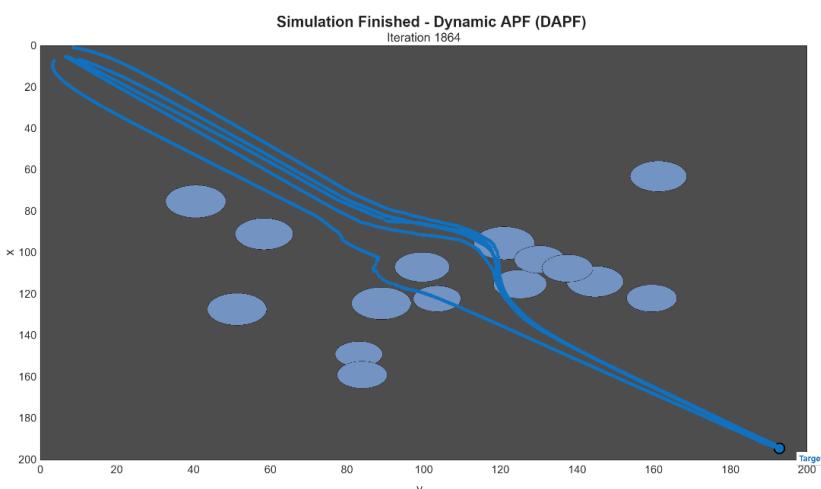
Pada skenario ini, rintangan statis ditempatkan secara acak di seluruh area operasi untuk menguji performa algoritma *Dynamic Artificial Potential Field* (DAPF) terhadap variasi kepadatan halangan. Pengujian dilakukan dalam tiga sub-kasus, yaitu: 15 *obstacle*, 25 *obstacle*, dan 35 *obstacle*.

a. Jumlah 15 halangan

Pada konfigurasi ini, sebanyak 15 rintangan statis tersebar secara acak dalam area operasi. Gambar 4.17 dan Gambar 4.18 memperlihatkan hasil simulasi dalam tampilan 2D dan 3D. Kerapatan halangan tergolong rendah sehingga UAV dapat menghindari *obstacle* dengan baik.

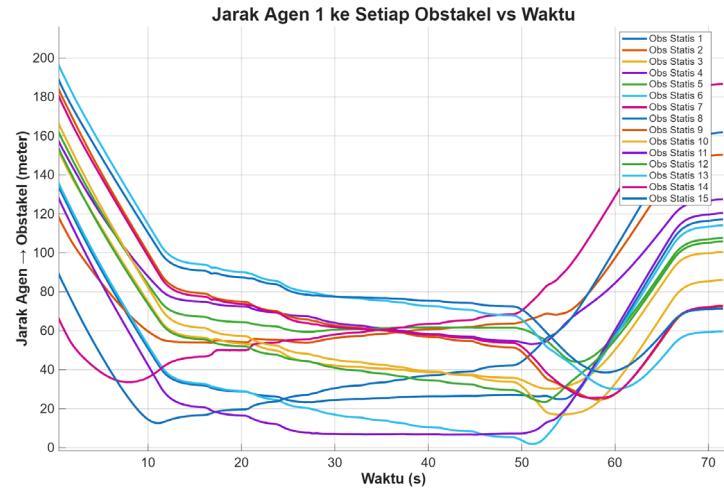


Gambar 4.17 Simulasi Pengujian 15 *Obstacle* Tersebar dalam Lintasan

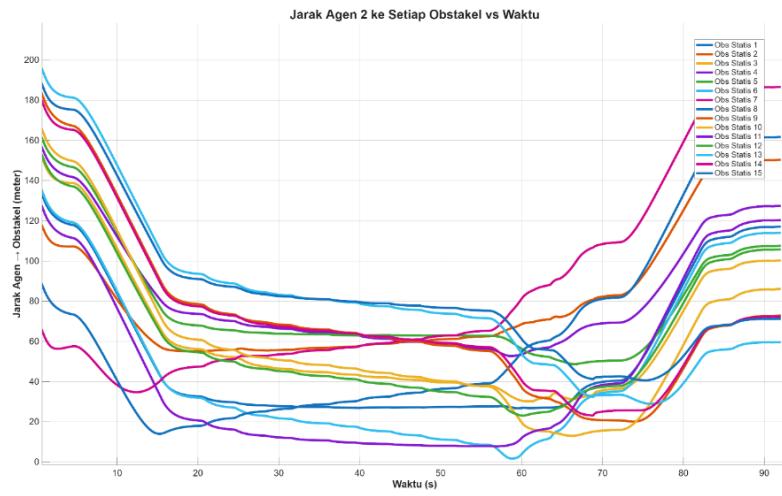


Gambar 4.18 Tampilan 3D Pengujian 15 *Obstacle* Tersebar dalam Lintasan

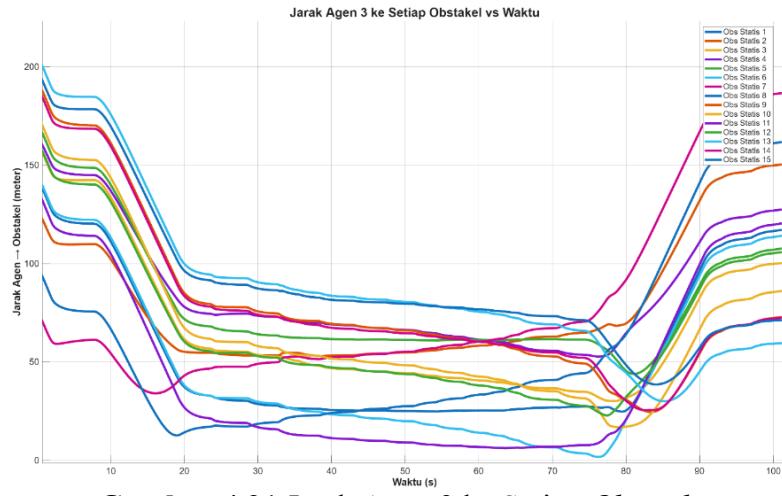
Seluruh agen berhasil berhenti dalam radius toleransi $R_{STOP} = 1 \text{ meter}$ dengan rata-rata jarak akhir sebesar 0.955 meter. Selain itu, kelima agen berhasil berhenti secara penuh dalam waktu yang bervariasi antara 87.3 hingga 103.7 detik. Grafik pada Gambar 4.19 hingga Gambar 4.23 menunjukkan jarak masing-masing agen terhadap *obstacle*, sedangkan Gambar 4.24 menampilkan jarak agen ke target terhadap waktu. Dari seluruh grafik tersebut, tidak ditemukan tabrakan antar agen dengan *obstacle*, dengan jarak minimum 1.54 meter dan jarak terhadap target menunjukkan tren menurun secara stabil.



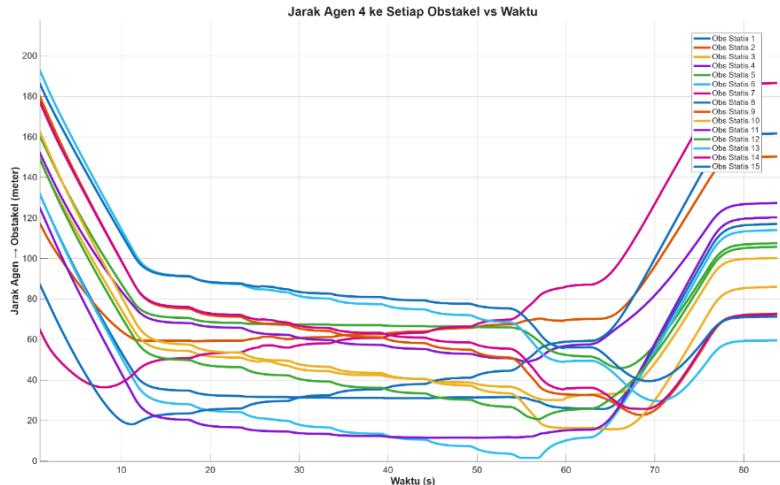
Gambar 4.19 Jarak Agen 1 ke Setiap *Obstacle*



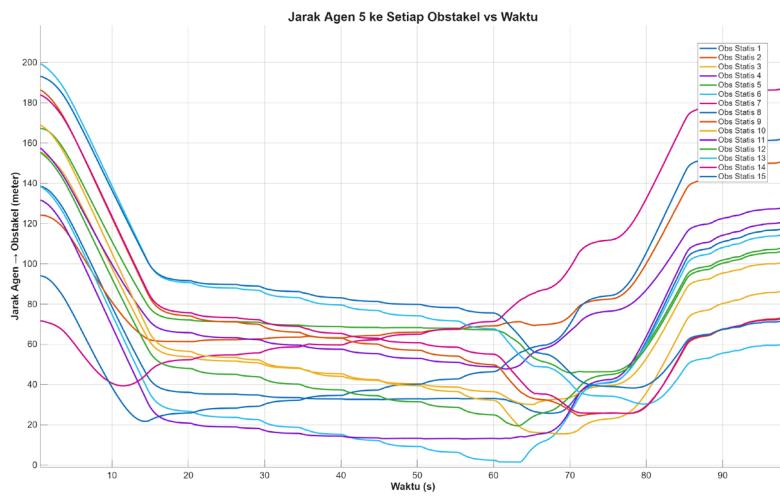
Gambar 4.20 Jarak Agen 2 ke Setiap *Obstacle*



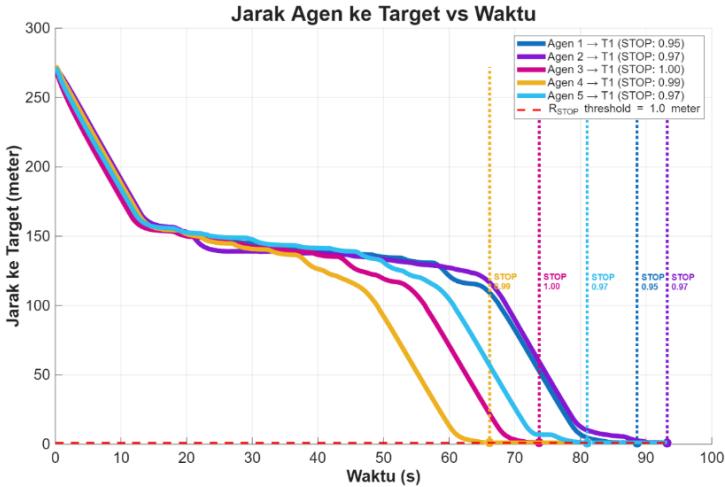
Gambar 4.21 Jarak Agen 3 ke Setiap *Obstacle*



Gambar 4.22 Jarak Agen 4 ke Setiap *Obstacle*



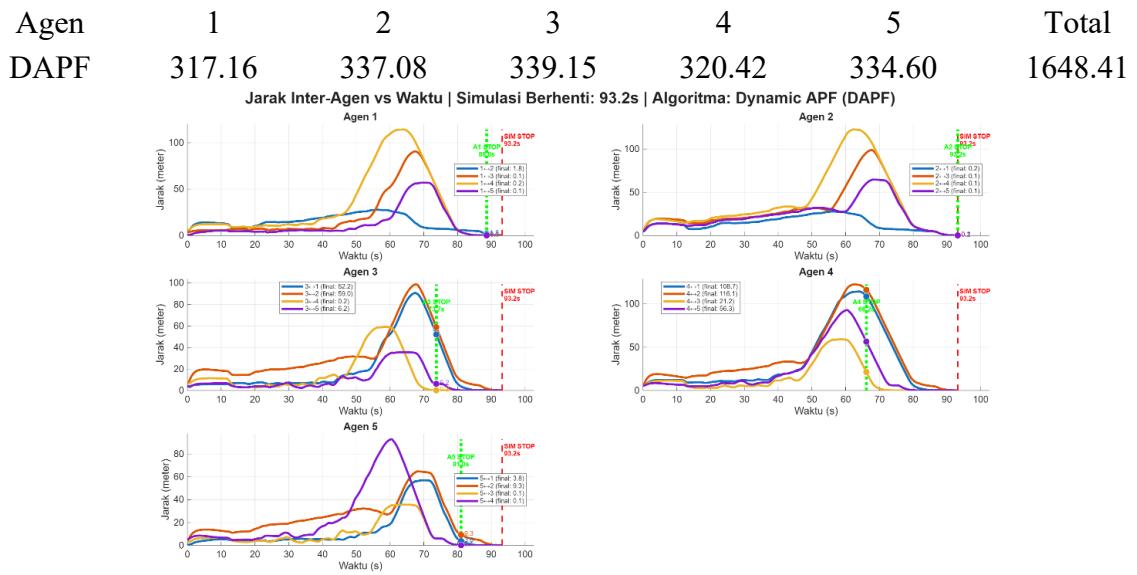
Gambar 4.23 Jarak Agen 5 ke Setiap *Obstacle*



Gambar 4.24 Jarak Agen ke Target terhadap Waktu

Panjang lintasan masing-masing agen disajikan dalam Tabel 4.2. Total lintasan yang ditempuh seluruh UAV adalah 1648.41 meter.

Tabel 4.2 Jarak Lintasan Agen dengan 15 *Obstacle* Statis

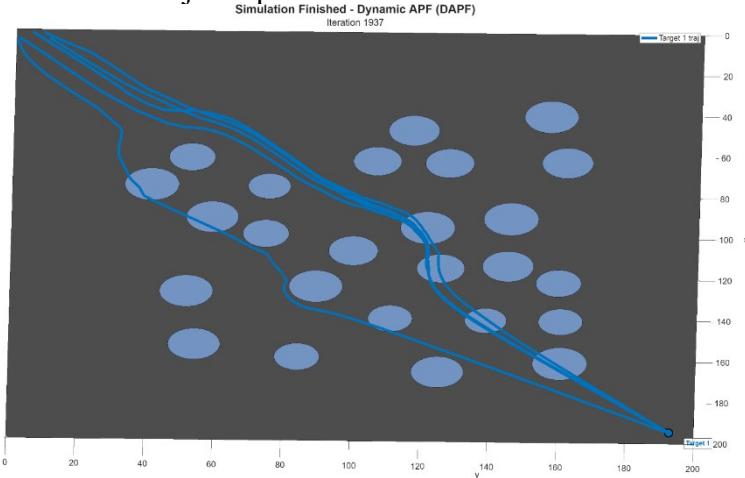


Gambar 4.25 Jarak Antar Agen

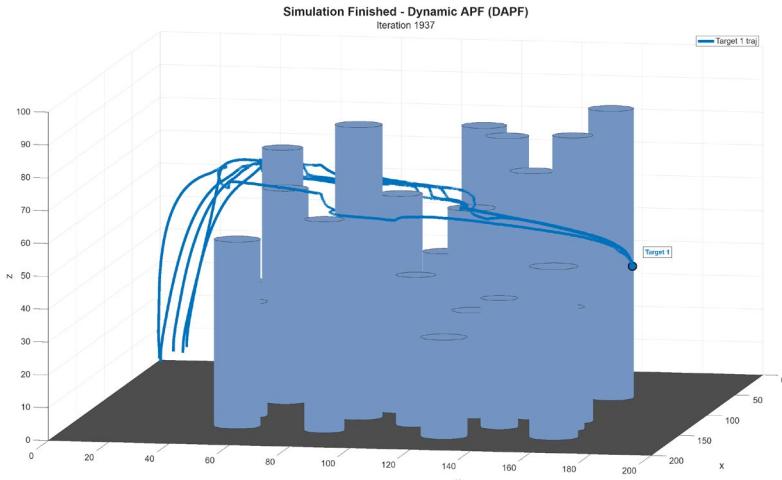
Gambar 4.25 menunjukkan jarak antar agen. Tidak ditemukan nilai mendekati nol, yang berarti agen tidak saling bertabrakan dan mampu menjaga jarak minimal lebih dari 4 meter.

b. Jumlah 25 halangan

Pada konfigurasi ini, sebanyak 25 *obstacle* tersebar secara acak dengan tingkat kepadatan sedang. Visualisasi lintasan disajikan pada Gambar 4.26 dan Gambar 4.27.

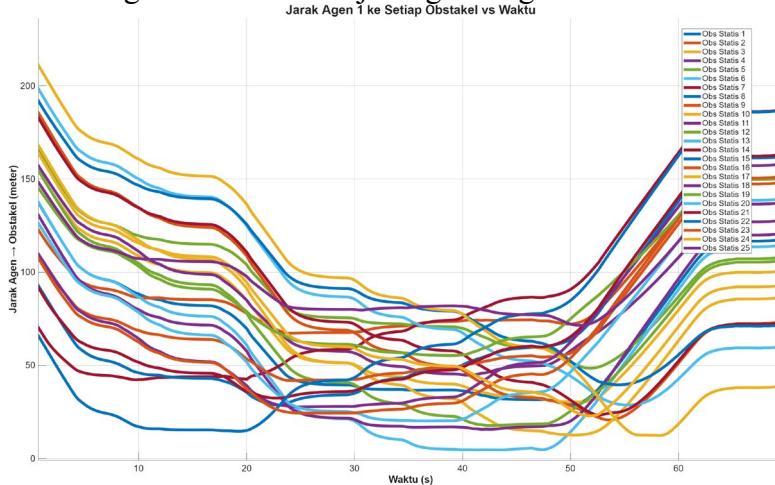


Gambar 4.26 Simulasi Pengujian 25 *Obstacle* Tersebar dalam Lintasan

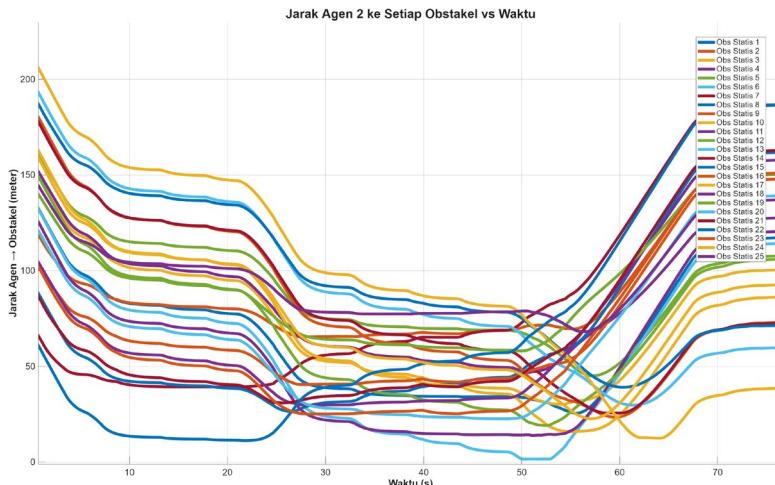


Gambar 4.27 Tampilan 3D Pengujian 25 *Obstacle* Tersebar dalam Lintasan

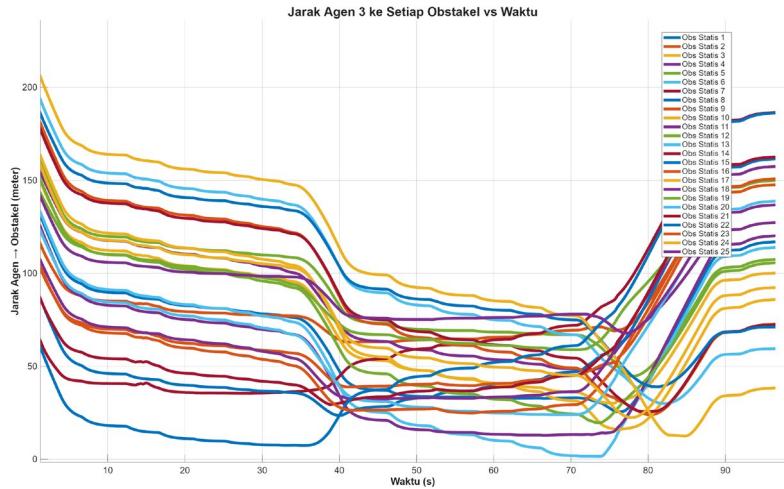
Seluruh agen berhasil berhenti dalam radius toleransi $R_{STOP} = 1 \text{ meter}$ dengan rata-rata jarak akhir sebesar 0.985 meter. Selain itu, kelima agen berhasil berhenti secara penuh dalam waktu yang bervariasi antara 69.15 hingga 96.85 detik. Grafik pada Gambar 4.28 hingga Gambar 4.32 menampilkan jarak masing-masing agen ke *obstacle*, sedangkan Gambar 4.33 menunjukkan jarak agen ke target terhadap waktu. Tidak terjadi tabrakan dengan dapat menjaga jarak minimal di 1.5 meter dan seluruh agen berhasil menuju target dengan aman.



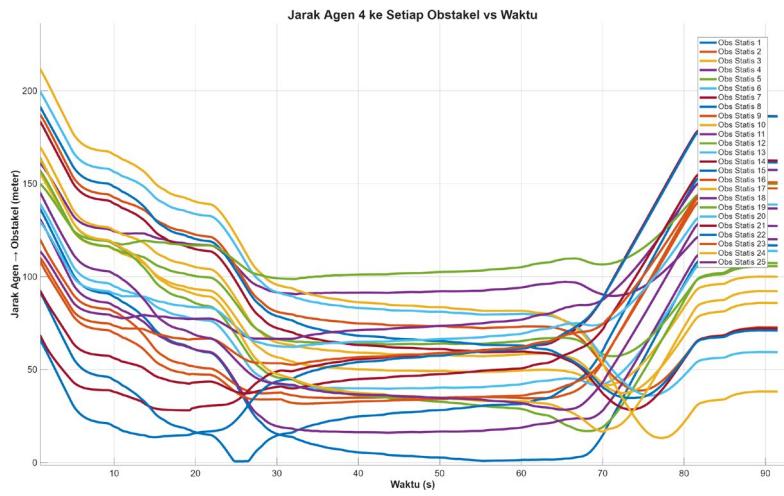
Gambar 4.28 Jarak Agen 1 ke Setiap *Obstacle*



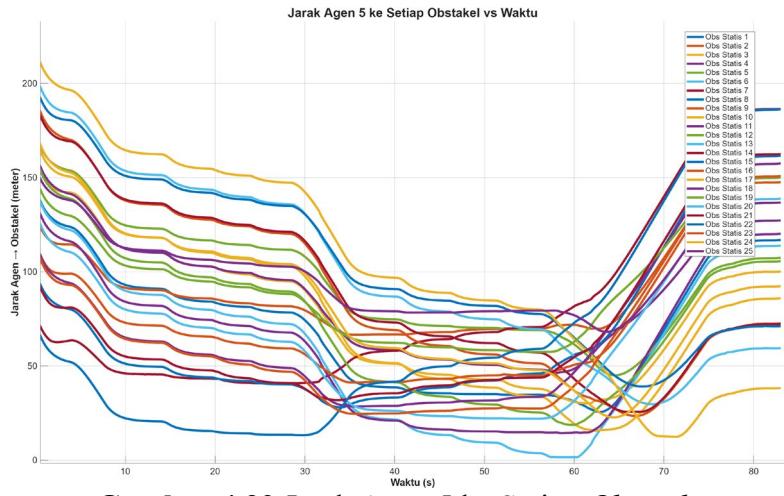
Gambar 4.29 Jarak Agen 2 ke Setiap *Obstacle*



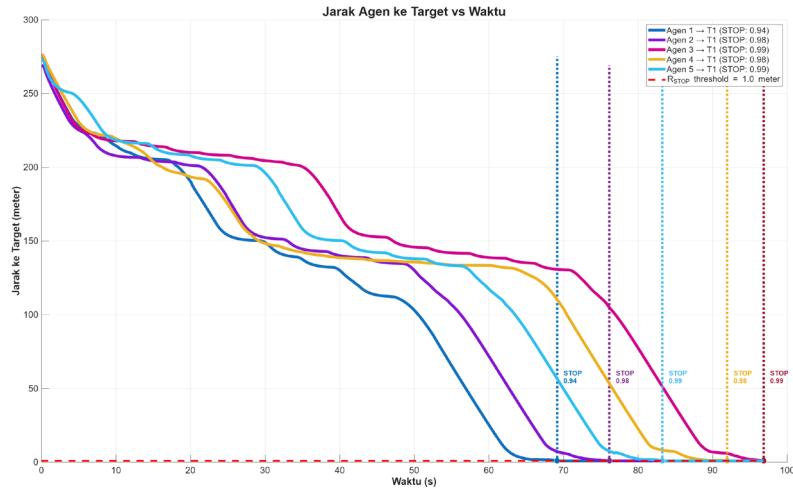
Gambar 4.30 Jarak Agen 3 ke Setiap *Obstacle*



Gambar 4.31 Jarak Agen 4 ke Setiap *Obstacle*



Gambar 4.32 Jarak Agen 5 ke Setiap *Obstacle*



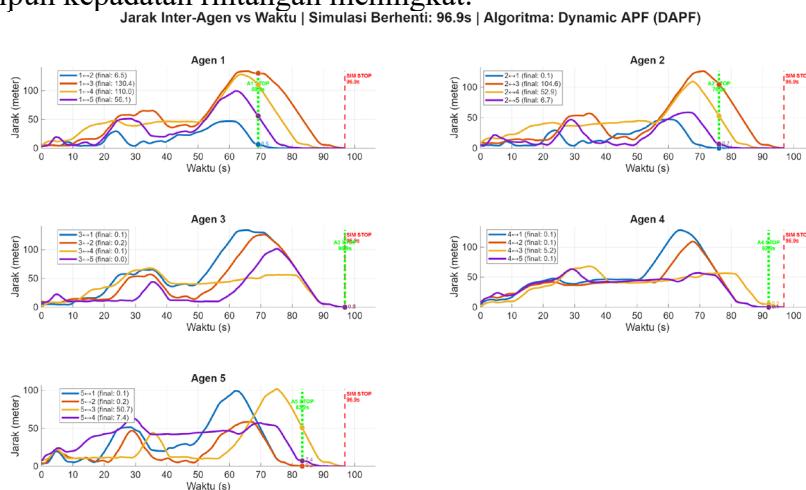
Gambar 4. 33 Jarak Agen ke Target terhadap Waktu

Panjang lintasan yang ditempuh oleh masing-masing agen disajikan pada Tabel 4.3 menunjukkan panjang lintasan masing-masing agen. Seluruh agen memiliki total lintasan sebesar 1641.59 meter.

Tabel 4.3 Jarak Lintasan Agen dengan 25 Obstacle Statis

| Agen | 1 | 2 | 3 | 4 | 5 | Total |
|------|--------|--------|--------|--------|--------|---------|
| DAPF | 323.67 | 316.33 | 332.62 | 337.16 | 331.81 | 1641.59 |

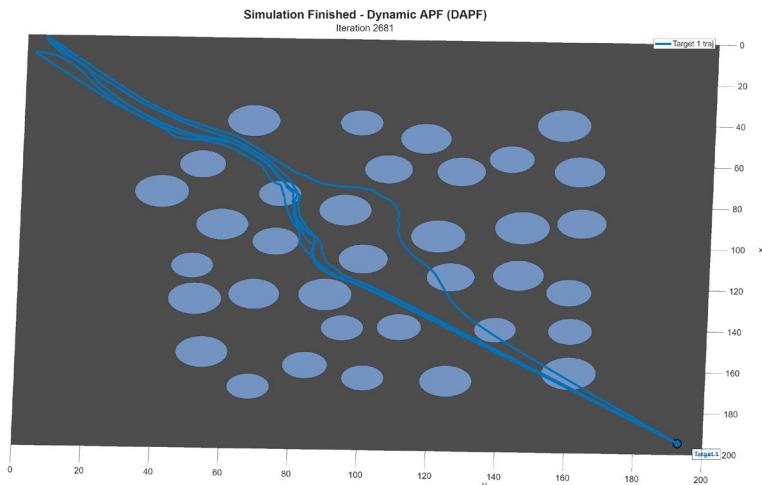
Gambar 4.34 memperlihatkan jarak antar agen yang tetap terjaga di atas 4 meter selama proses simulasi, meskipun kepadatan rintangan meningkat.



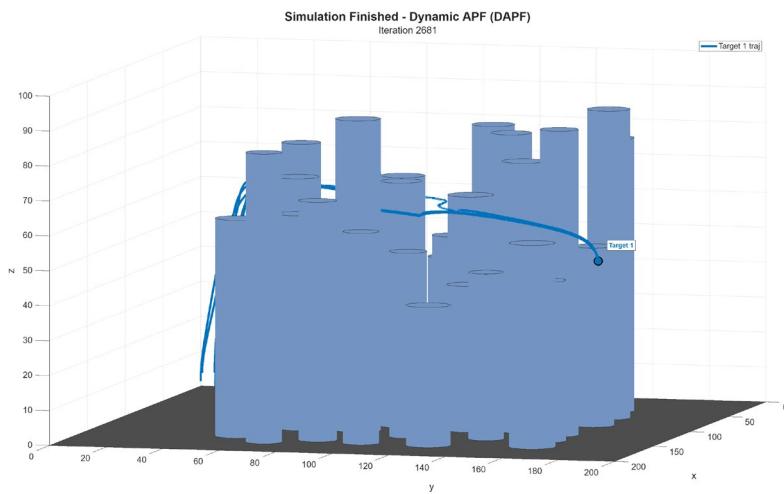
Gambar 4.34 Jarak Antar Agen

c. Jumlah 35 halangan

Pada konfigurasi terakhir, 35 *obstacle* tersebar secara acak dengan tingkat kepadatan lebih tinggi. Gambar 4. 35 dan Gambar 4. 36 menunjukkan visualisasi 2D dan 3D dari lintasan UAV.

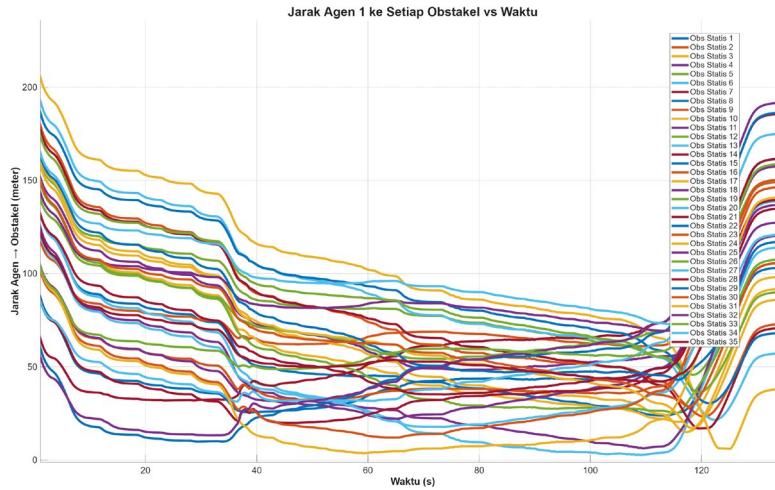


Gambar 4.35 Simulasi Pengujian 35 *Obstacle* Tersebar dalam Lintasan

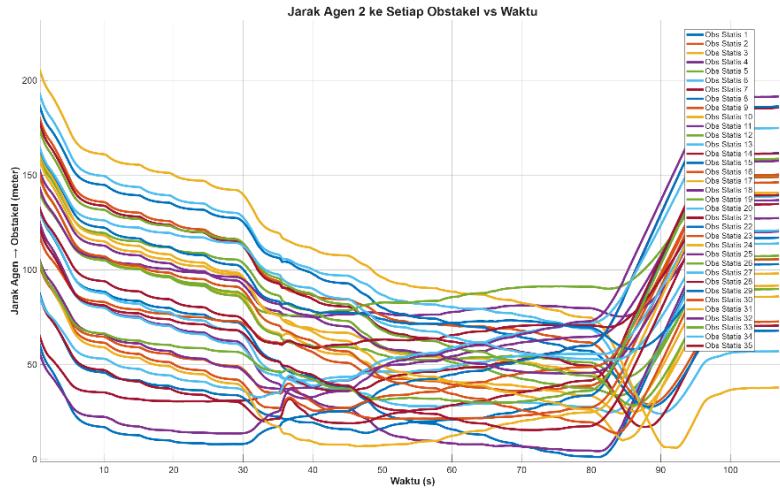


Gambar 4.36 Tampilan 3D Pengujian 35 *Obstacle* Tersebar dalam Lintasan

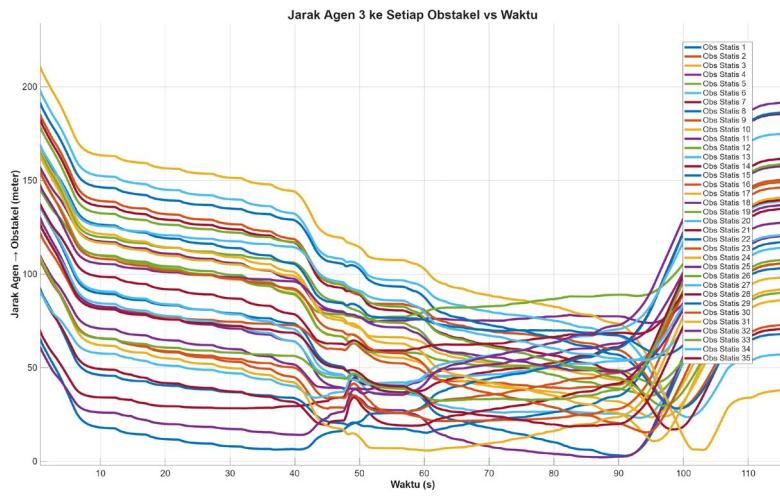
Seluruh agen berhasil berhenti dalam radius toleransi $R_{STOP} = 1 \text{ meter}$ dengan rata-rata jarak akhir sebesar 0.980 meter. Selain itu, kelima agen berhasil berhenti secara penuh dalam waktu yang bervariasi antara 96.8 hingga 134.1 detik. Grafik jarak agen ke *obstacle* ditampilkan pada Gambar 4.37 hingga Gambar 4.41, sedangkan Gambar 4.42 memperlihatkan jarak ke target. Tidak terdapat indikasi tabrakan atau kegagalan manuver, dengan jarak minimum yang terjadi yaitu 1.24 meter.



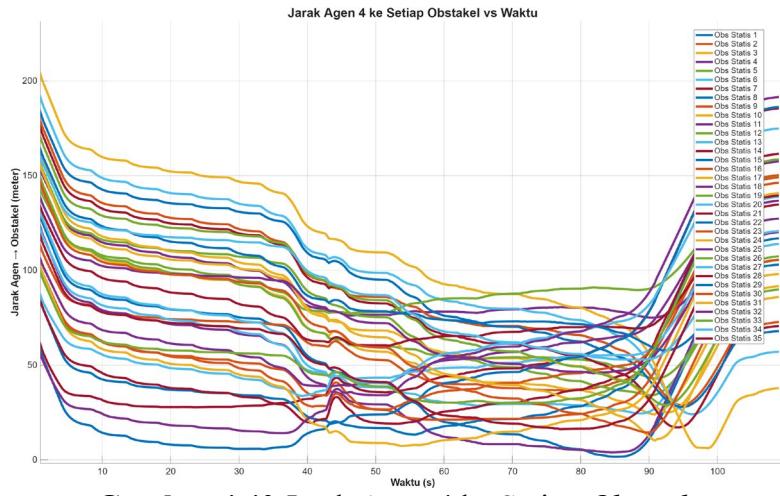
Gambar 4.37 Jarak Agen 1 ke Setiap *Obstacle*



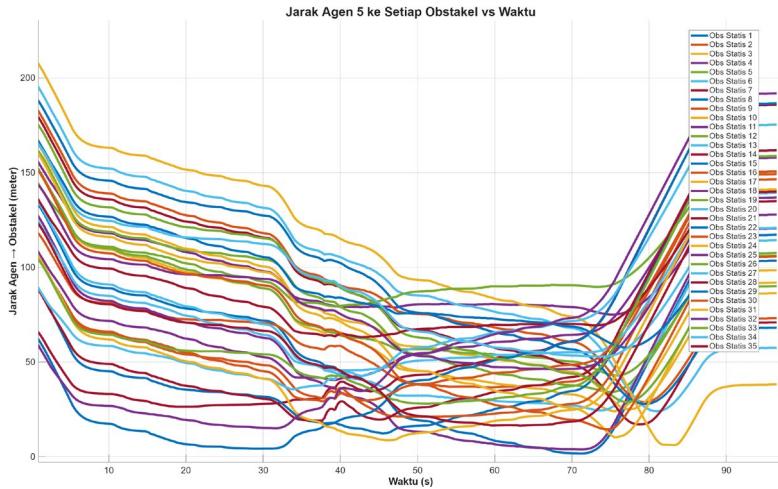
Gambar 4.38 Jarak Agen 2 ke Setiap *Obstacle*



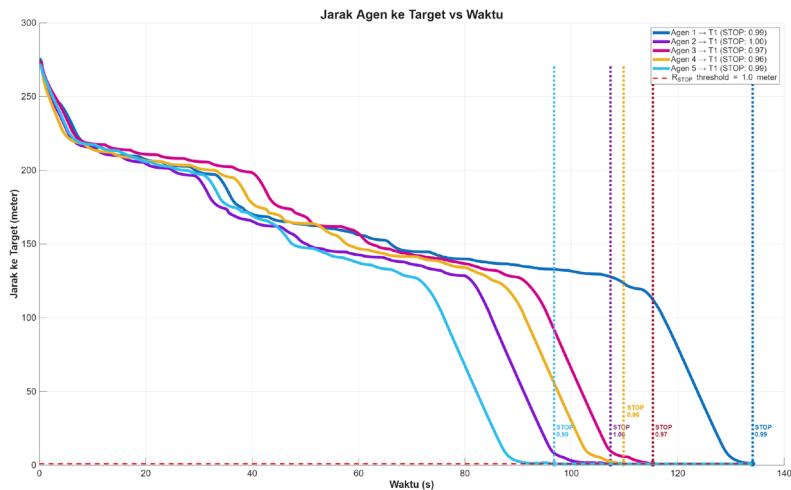
Gambar 4.39 Jarak Agen 3 ke Setiap *Obstacle*



Gambar 4.40 Jarak Agen 4 ke Setiap *Obstacle*



Gambar 4.41 Jarak Agen 5 ke Setiap *Obstacle*



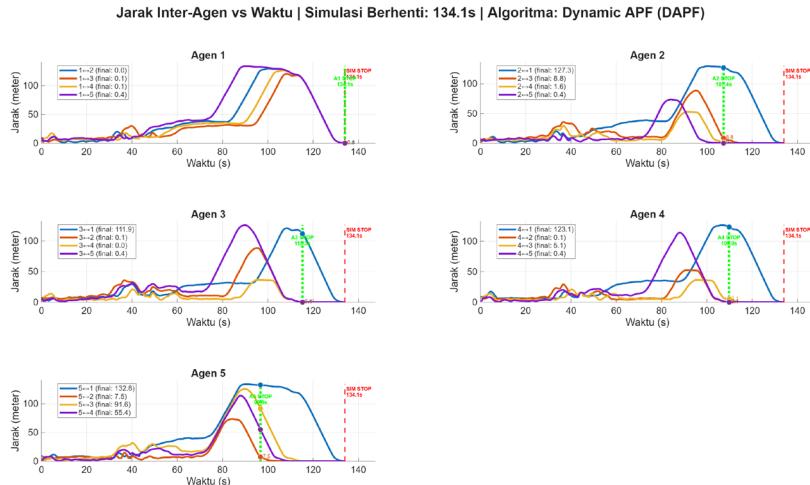
Gambar 4.42 Jarak Agen ke Target terhadap Waktu

Panjang lintasan yang ditempuh oleh masing-masing agen identik, seperti disajikan pada Tabel 4.4. Total lintasan seluruh agen mencapai 1724.37 meter.

Tabel 4.4 Jarak Lintasan Agen dengan 35 *Obstacle* Statis

| Agen | 1 | 2 | 3 | 4 | 5 | Total |
|------|--------|--------|--------|--------|--------|---------|
| DAPF | 349.43 | 343.19 | 349.46 | 343.66 | 338.63 | 1724.37 |

Jarak antar agen ditampilkan pada Gambar 4.43, yang menunjukkan bahwa semua agen tetap menjaga jarak aman selama navigasi meskipun dalam kondisi rintangan padat, dengan jarak minimal 1.3 meter.



Gambar 4.43 Jarak Antar Agen

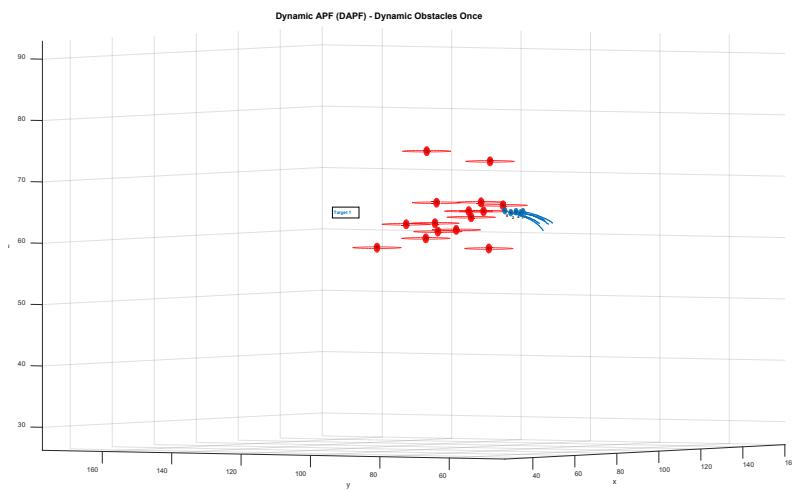
4.1.2 Penghindaran halangan dinamis

Untuk menangani skenario ini, digunakan algoritma Dynamic Artificial Potential Field (DAPF) yang menggabungkan komponen gaya tarik menuju target dan gaya tolak terhadap halangan. Parameter pengaturan yang diterapkan dalam simulasi meliputi gain tarik posisi dan kecepatan (k_{att}) senilai 1, gain tolak posisi dan kecepatan (k_{rep}) senilai 5. Simulasi dilaksanakan dalam dua jenis skenario utama. Skenario pertama menyimulasikan kemunculan *obstacle* dinamis secara bersamaan dalam satu waktu. Sementara itu, skenario kedua menguji ketahanan sistem terhadap gangguan yang muncul secara bertahap (sekuensial).

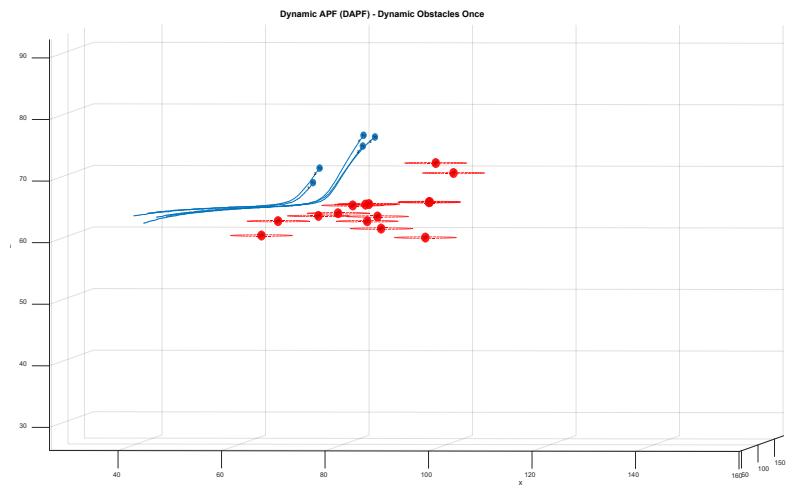
4.1.2.1 Skenario muncul bersamaan

a. Konfigurasi *obstacle head-on*

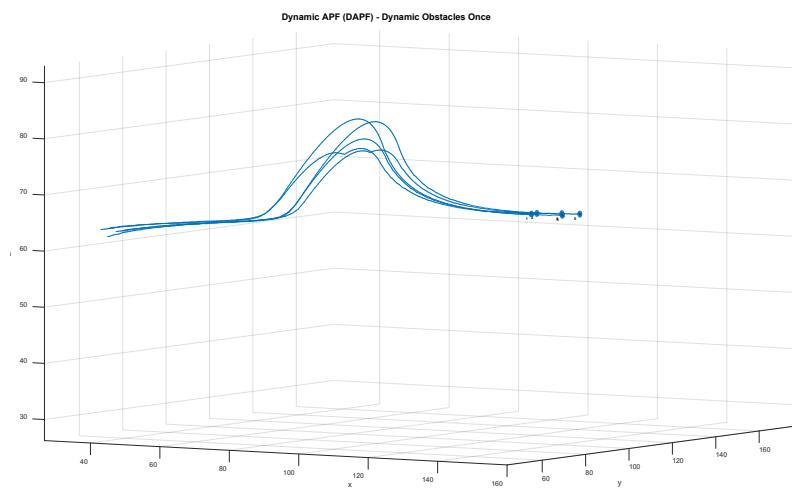
Pada konfigurasi ini, *obstacle* dinamis bergerak dari arah depan secara langsung menuju lintasan UAV. Skema ini dirancang untuk menguji kemampuan sistem *Dynamic Artificial Potential Field* (DAPF) dalam menghadapi potensi tabrakan frontal. Respons UAV terhadap skenario ini ditampilkan dalam Gambar 4.44 hingga Gambar 4.46, yang menggambarkan situasi sebelum, saat, dan setelah berdekatan dengan halangan dinamis. Pada urutan gambar tersebut terlihat bahwa UAV melakukan manuver penghindaran terhadap *obstacle* yang bergerak dari arah depan. Pergerakan UAV menunjukkan perubahan arah dan lintasan untuk menghindari potensi tabrakan, lalu kembali mengarahkan diri menuju target yang telah ditentukan.



Gambar 4.44 Respons Agen *Obstacle* dari Depan (Sebelum)



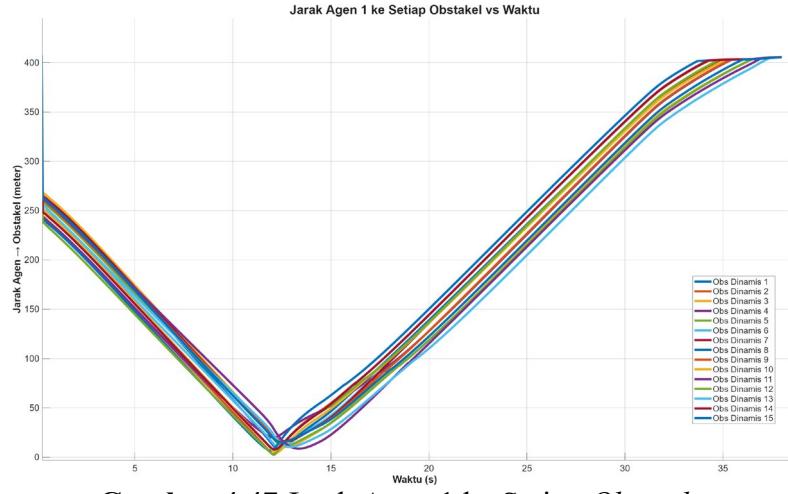
Gambar 4.45 Respons Agen *Obstacle* dari Depan (Saat)



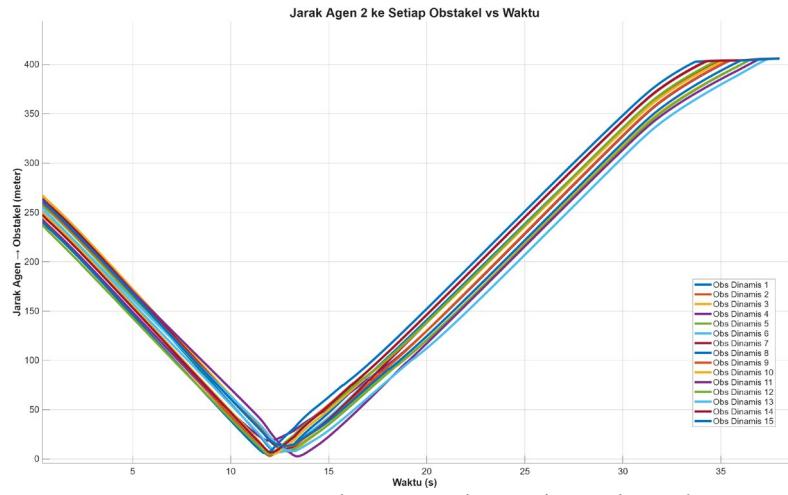
Gambar 4.46 Respons Agen *Obstacle* dari Depan (Setelah)

Untuk mengevaluasi kemampuan penghindaran, dilakukan analisis terhadap jarak antara masing-masing UAV dan *obstacle* dinamis sepanjang waktu simulasi. Hasilnya ditampilkan dalam Gambar 4.47 hingga Gambar 4.51. Semua grafik memperlihatkan kurva berbentuk "V", di mana jarak antara UAV dan *obstacle* awalnya terus menurun hingga mencapai titik minimum

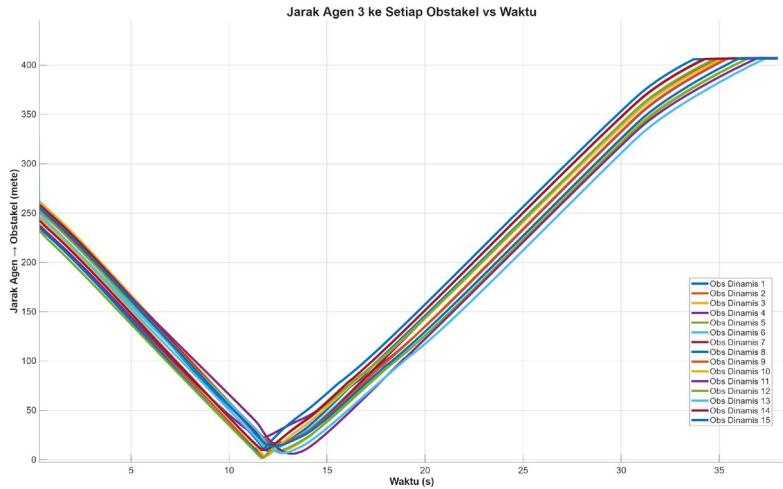
pada sekitar detik ke-10 hingga 15, yang mengindikasikan momen kritis pertemuan jarak antara agen dan *obstacle*. Setelah itu, jarak kembali meningkat seiring dengan dilakukannya manuver penghindaran oleh UAV terhadap halangan yang bergerak dari arah depan. Tidak terdapat titik nol pada grafik, yang mengonfirmasi bahwa tidak terjadi tabrakan antara UAV dan *obstacle* dinamis selama simulasi berlangsung. Serta dapat memenuhi *threshold* batas minimum dari *obstacle* yaitu 1.6 meter.



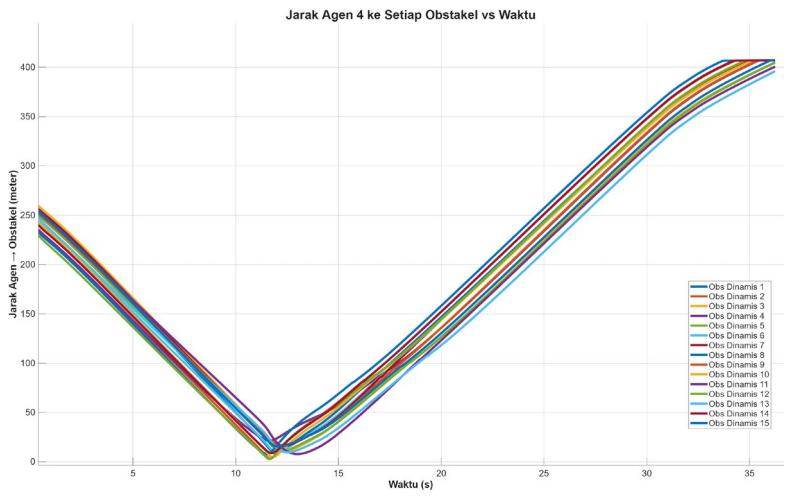
Gambar 4.47 Jarak Agen 1 ke Setiap *Obstacle*



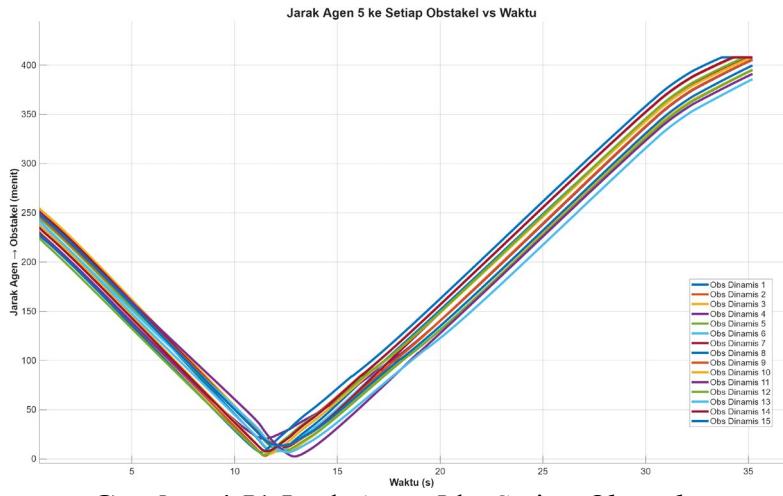
Gambar 4.48 Jarak Agen 2 ke Setiap *Obstacle*



Gambar 4.49 Jarak Agen 3 ke Setiap *Obstacle*



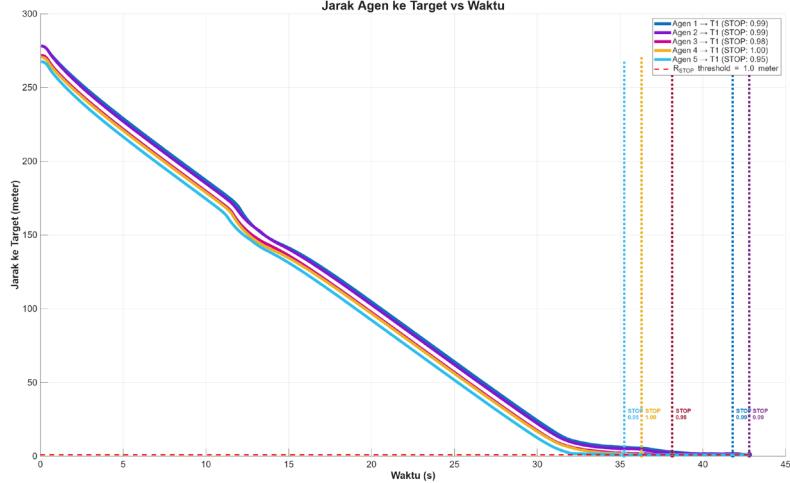
Gambar 4.50 Jarak Agen 4 ke Setiap *Obstacle*



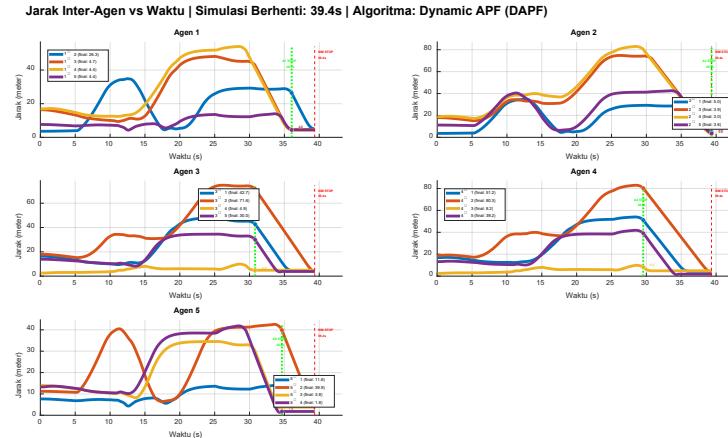
Gambar 4.51 Jarak Agen 5 ke Setiap *Obstacle*

Pada Gambar 4.52 memperlihatkan grafik jarak masing-masing UAV terhadap target sepanjang waktu simulasi. Pola penurunan yang konsisten menunjukkan bahwa setiap agen secara progresif mendekati target hingga mencapai titik tujuan. Sementara itu, Gambar 4.53 menunjukkan jarak antar agen selama manuver berlangsung. Tidak ditemukan kedekatan

ekstrem antar UAV, dimana seluruh agen mampu menjaga jarak aman satu sama lain sepanjang simulasi, masih dalam *threshold* 0.8 meter.



Gambar 4.52 Jarak Agen ke Target



Gambar 4.53 Jarak Antar Agen

Dari hasil simulasi, dapat disimpulkan bahwa seluruh agen berhasil mencapai target dengan aman tanpa terjadi tabrakan, baik terhadap *obstacle* maupun sesama agen. Sistem DAPF menunjukkan kemampuan dalam menghadapi ancaman frontal dan mengatur navigasi multi-agenn secara aman. Total waktu simulasi tercatat sebesar 39.45 detik, dengan jarak minimum antara agen dan *obstacle* sebesar 1.8 meter. Panjang lintasan yang ditempuh oleh masing-masing UAV dalam skenario ini ditampilkan pada Tabel 4.5.

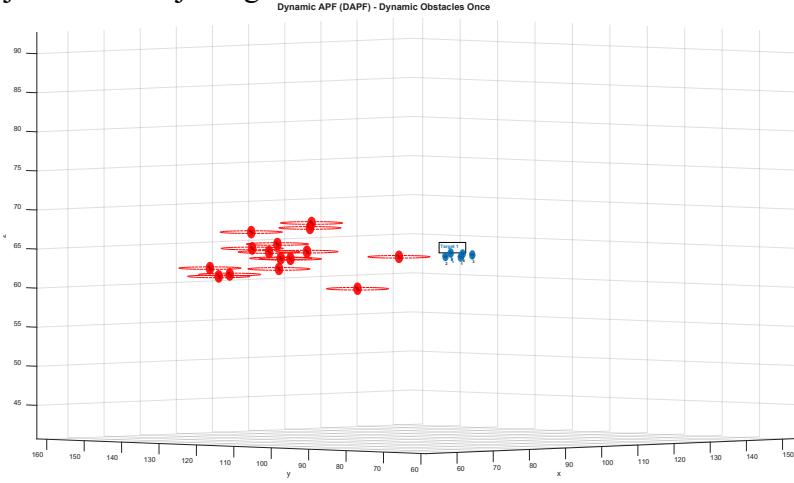
Tabel 4.5 Jarak Lintasan Agen *Obstacle* Depan

| Agen | 1 | 2 | 3 | 4 | 5 | Total |
|------|--------|--------|--------|--------|--------|----------|
| DAPF | 279.40 | 283.97 | 285.28 | 281.80 | 279.33 | 1 409.78 |

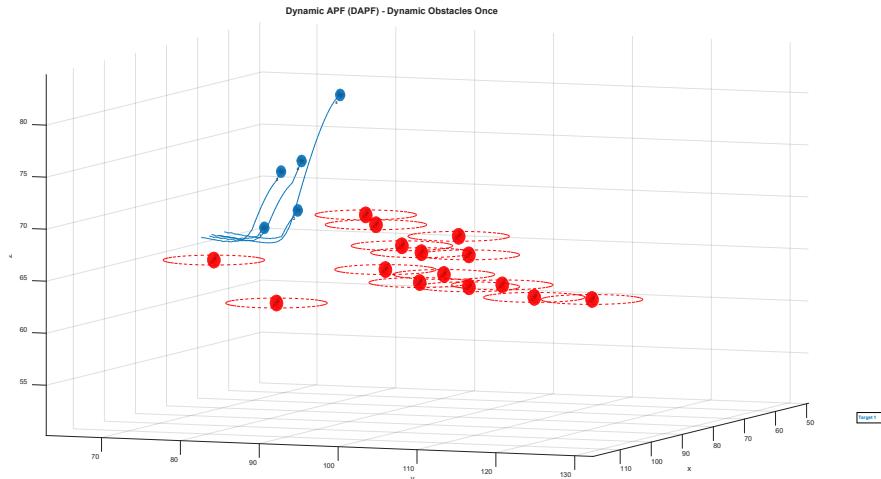
b. Konfigurasi *obstacle side*

Pada konfigurasi ini, *obstacle* dinamis bergerak dari arah samping dan melintasi jalur UAV secara lateral. Tujuan skenario ini adalah untuk menguji performa *Dynamic Artificial Potential Field* (DAPF) dalam menghadapi ancaman tabrakan dari sisi lintasan UAV. Visualisasi pergerakan agen sebelum, saat, dan sesudah pertemuan dengan *obstacle* ditampilkan pada Gambar 4.54 hingga Gambar 4.56. Pada gambar tersebut, tampak bahwa UAV melakukan

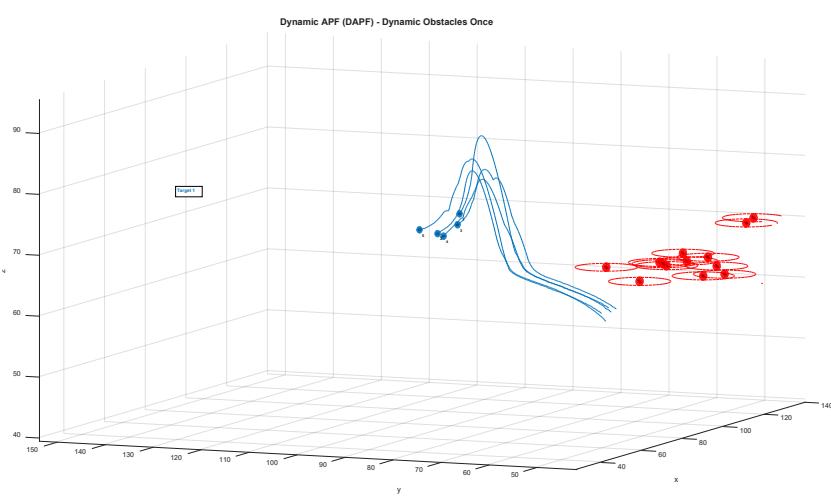
manuver penghindaran *lateral* terhadap *obstacle* yang datang dari samping sebelum kembali melanjutkan perjalanan menuju target.



Gambar 4.54 Respons Agen *Obstacle* dari Samping (Sebelum)



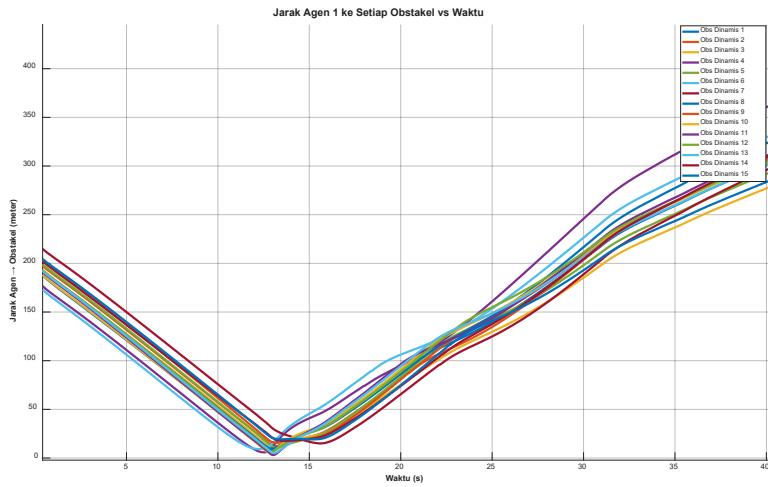
Gambar 4.55 Respons Agen *Obstacle* dari Samping (Saat)



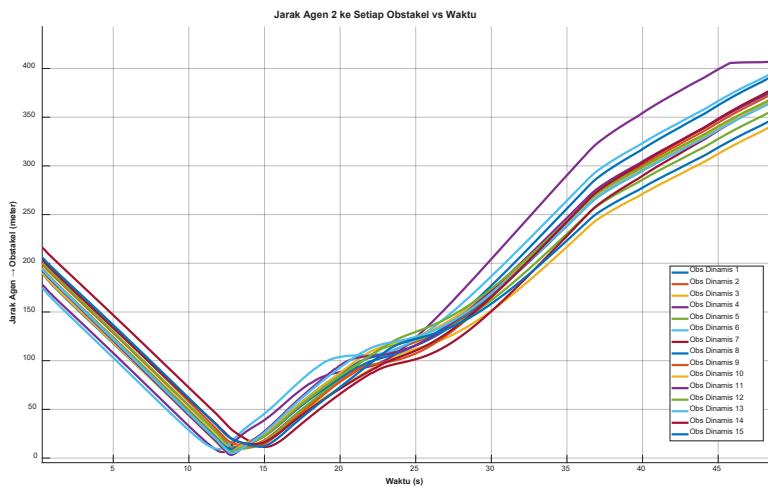
Gambar 4.56 Respons Agen *Obstacle* dari Samping (Sesudah)

Gambar 4.57 hingga Gambar 4.61 menampilkan grafik jarak masing-masing UAV terhadap *obstacle* dinamis. Kurva yang ditunjukkan mengikuti pola mendekat, mencapai titik minimum, dan kemudian menjauh. Tidak ditemukan titik dengan nilai nol pada grafik, dengan jarak

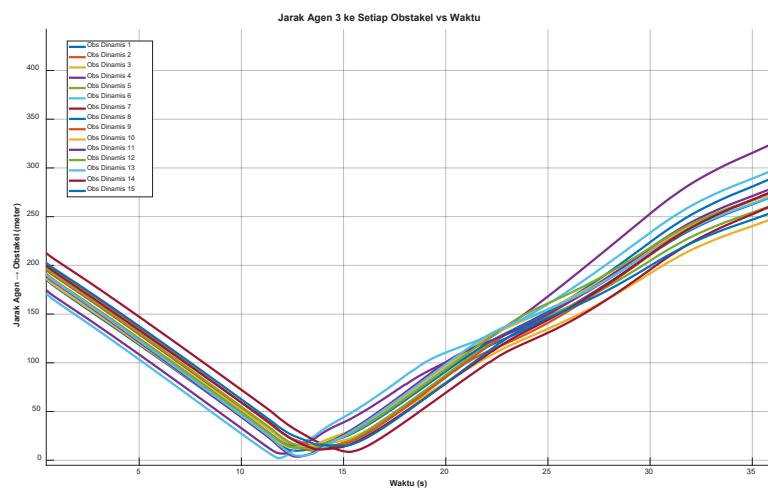
minimum yaitu 3 meter, yang mengindikasikan bahwa tidak terjadi kontak langsung antara UAV dan *obstacle*. Pada skenario *obstacle* dari samping, jarak minimum yang terjadi lebih jauh dibandingkan dengan *obstacle* dari depan, dengan batas minimum lebih jauh 1.2 meter.



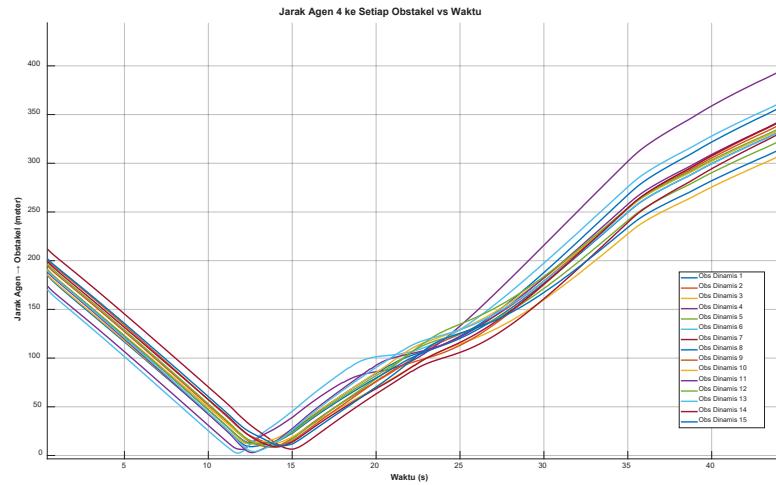
Gambar 4.57 Jarak Agen 1 ke Setiap *Obstacle*



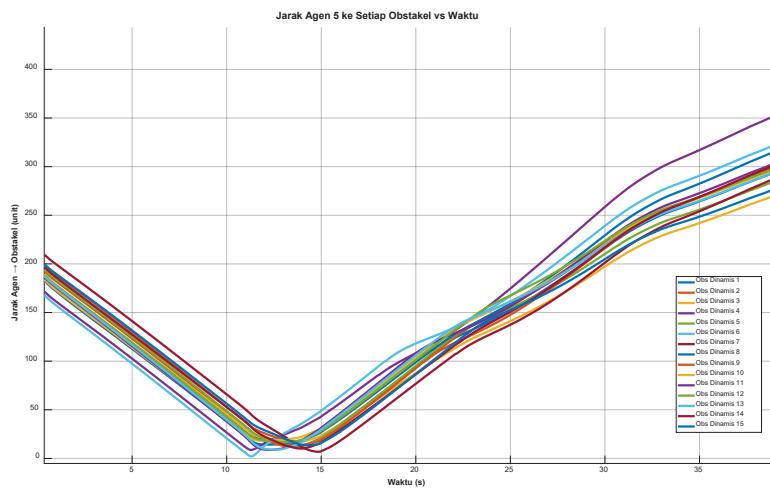
Gambar 4.58 Jarak Agen 2 ke Setiap *Obstacle*



Gambar 4.59 Jarak Agen 3 ke Setiap *Obstacle*

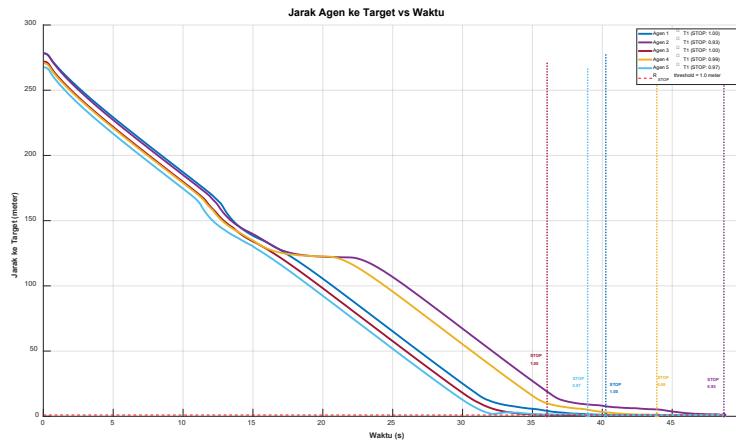


Gambar 4.60 Jarak Agen 4 ke Setiap *Obstacle*

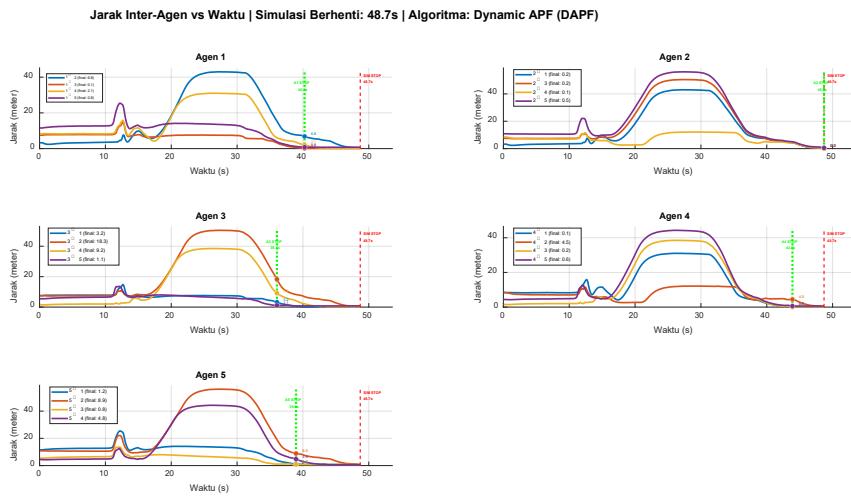


Gambar 4.61 Jarak Agen 5 ke Setiap *Obstacle*

Selanjutnya, Gambar 4.62 menunjukkan jarak UAV terhadap target sepanjang waktu simulasi. Grafik memperlihatkan tren penurunan yang konsisten, yang mengindikasikan bahwa seluruh agen secara bertahap mendekati target hingga berhasil mencapainya. Sementara itu, Gambar 4.63 menampilkan jarak antar UAV. Selama simulasi, tidak terdapat kedekatan ekstrem yang dapat memicu potensi tabrakan antar agen, dengan jarak minimum 1 meter.



Gambar 4.62 Jarak antar Agen dan Target



Gambar 4.63 Jarak Antar Agen

Berdasarkan hasil simulasi, dapat disimpulkan bahwa seluruh UAV berhasil mencapai target tanpa mengalami tabrakan dengan *obstacle* maupun dengan agen lainnya dengan jarak minimum ke *obstacle* sebesar 3 meter dan antar agen sebesar 1 meter. Sistem DAPF menunjukkan kinerja dalam menangani ancaman dari arah lateral. Total waktu simulasi tercatat sebesar 48.7 detik. Total lintasan kumulatif yang ditempuh oleh lima agen adalah 1583.84 meter sebagaimana ditunjukkan pada Tabel 4.6.

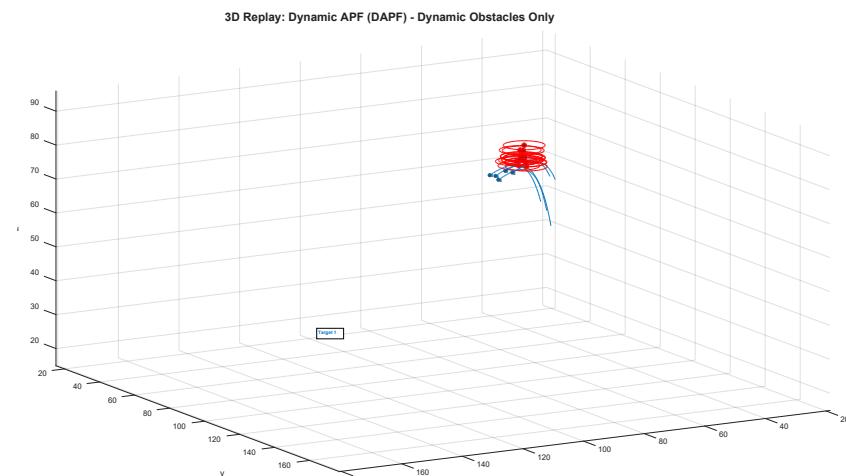
Tabel 4.6 Jarak Lintasan Agen *Obstacle* Samping

| Agen | 1 | 2 | 3 | 4 | 5 | Total |
|------|--------|--------|--------|--------|--------|---------|
| DAPF | 324.06 | 322.16 | 311.44 | 310.62 | 315.56 | 1583.84 |

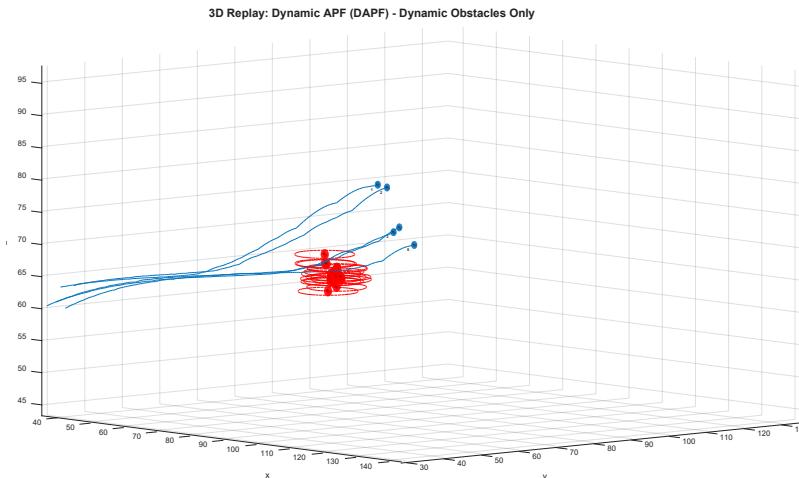
Pada skenario ini, waktu simulasi dan panjang lintasan yang terjadi lebih besar dibandingkan dengan skenario *obstacle* dari depan.

c. Konfigurasi *obstacle* back

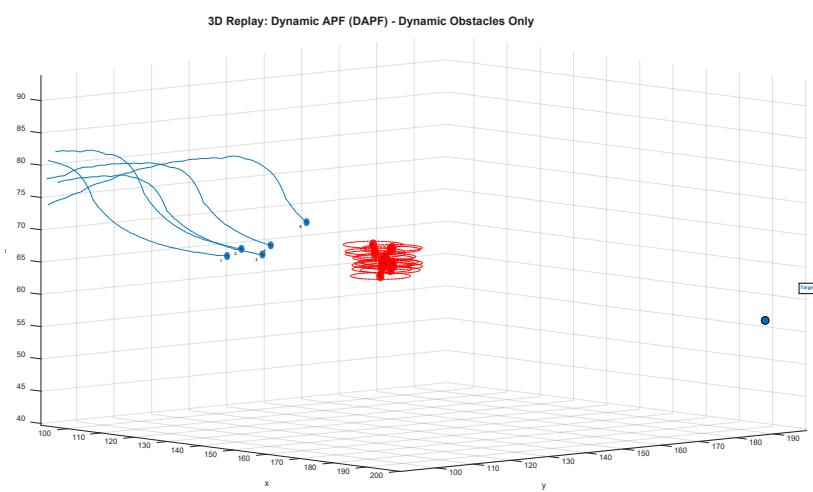
Pada konfigurasi dengan *obstacle* dinamis yang bergerak dari belakang (*back*) menuju UAV, sistem DAPF diuji kemampuannya dalam menangani ancaman tabrakan yang datang dari arah belakang. Dapat terlihat pada gambar simulasi yang menampilkan respons UAV saat konfigurasi ini.



Gambar 4.64 Respons Agen *Obstacle* dari Belakang (Sebelum)

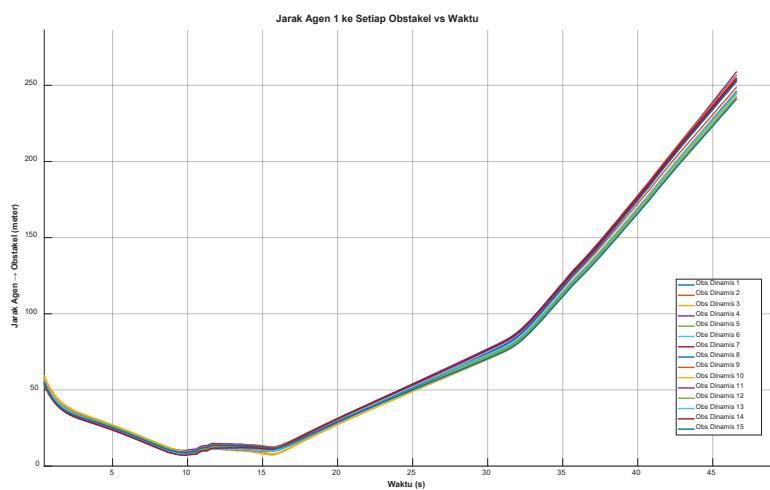


Gambar 4.65 Respons Agen *Obstacle* dari Belakang (Saat)

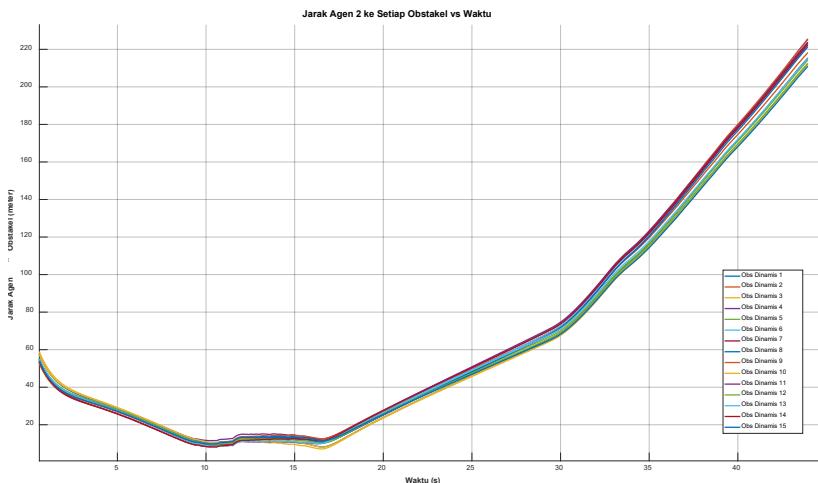


Gambar 4.66 Respons Agen *Obstacle* dari Belakang (Sesudah)

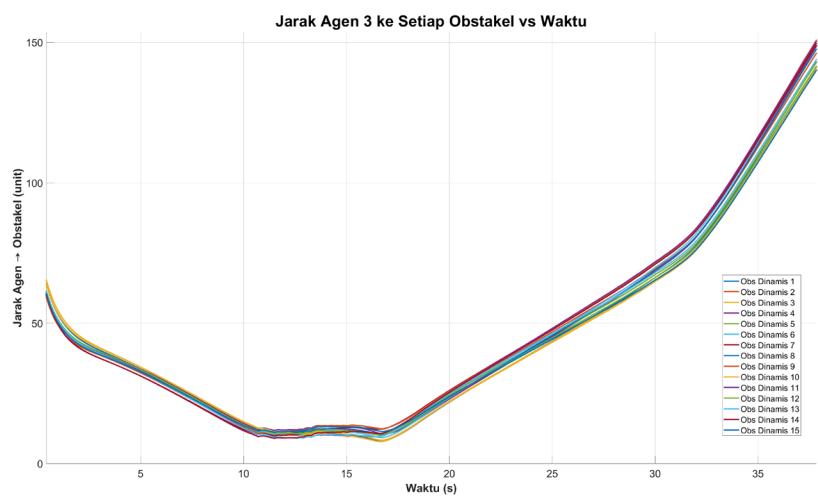
Pada gambar 4.28 terlihat Jarak UAV ke *Obstacle* terhadap Waktu. Grafik menunjukkan fluktuasi jarak setiap UAV terhadap *obstacle* dinamis. Terlihat pola penurunan jarak saat *obstacle* mendekat dari belakang, dengan jarak terdekat ke *obstacle* sebesar 12 meter, namun tidak pernah mencapai titik tabrakan.



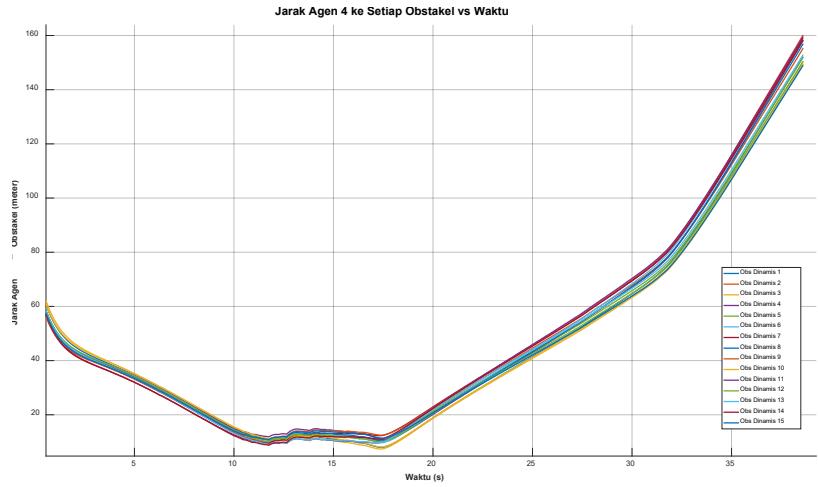
Gambar 4.67 Jarak Agen 1 ke Setiap *Obstacle*



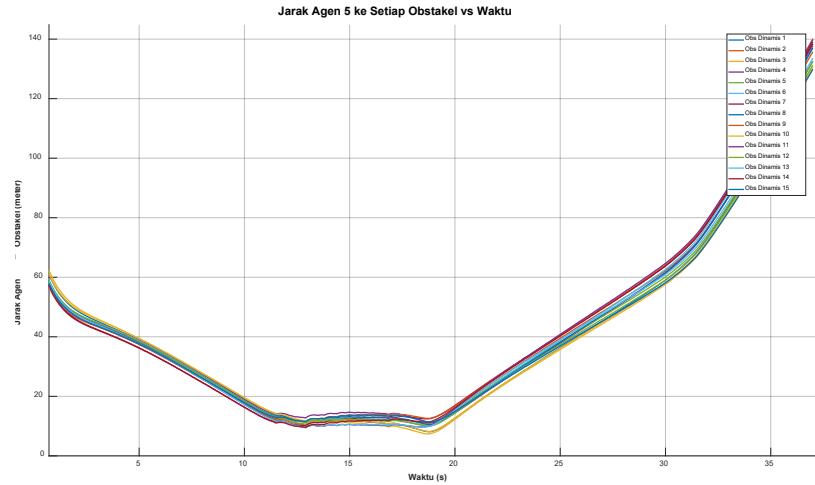
Gambar 4.68 Jarak Agen 2 ke Setiap *Obstacle*



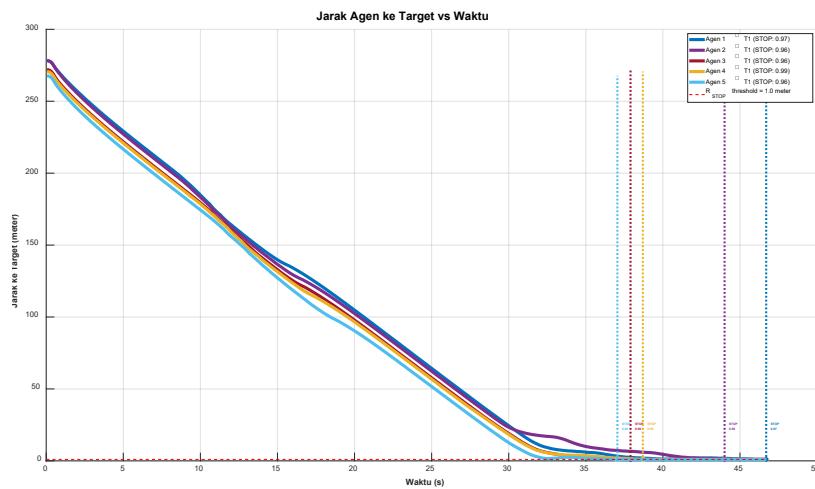
Gambar 4.69 Jarak Agen 3 ke Setiap *Obstacle*



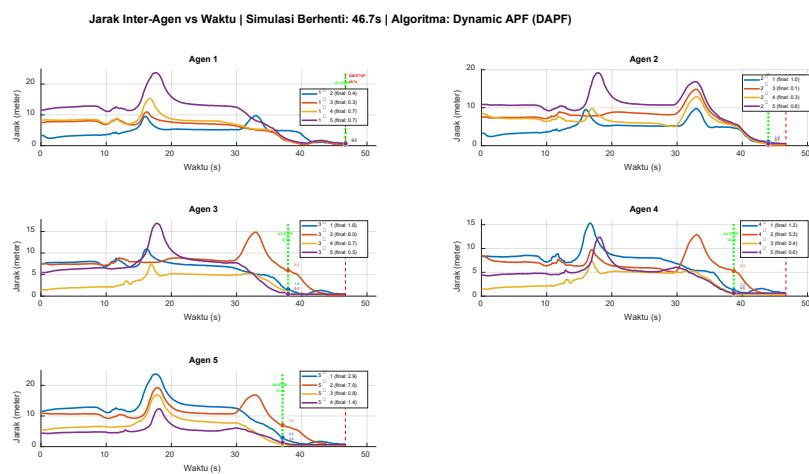
Gambar 4.70 Jarak Agen 4 ke Setiap *Obstacle*



Gambar 4.71 Jarak Agen 5 ke Setiap *Obstacle*



Gambar 4.72 Jarak Agen ke Target



Gambar 4.73 Jarak Antar Agen

Berdasarkan hasil simulasi, didapatkan bahwa seluruh UAV berhasil mencapai target tanpa mengalami tabrakan dengan *obstacle* maupun dengan agen lainnya dengan jarak minimum ke *obstacle* sebesar 12 meter dan antar agen sebesar 1.4 meter. Sistem DAPF menunjukkan kinerja dalam menangani ancaman dari arah lateral. Total waktu simulasi tercatat sebesar 46.7 detik.

Hal ini dapat terlihat bahwa waktu simulasi lebih cepat dibandingkan dengan skenario yang lainnya, dikarenakan pada skenario *obstacle* dari belakang, mengaktifkan gaya tolak berdasarkan kecepatan sehingga UAV menjadi lebih cepat melakukan penghindaran dan menuju target. Total lintasan kumulatif yang ditempuh oleh lima agen adalah 1577.14 meter sebagaimana ditunjukkan pada Tabel 4.6.

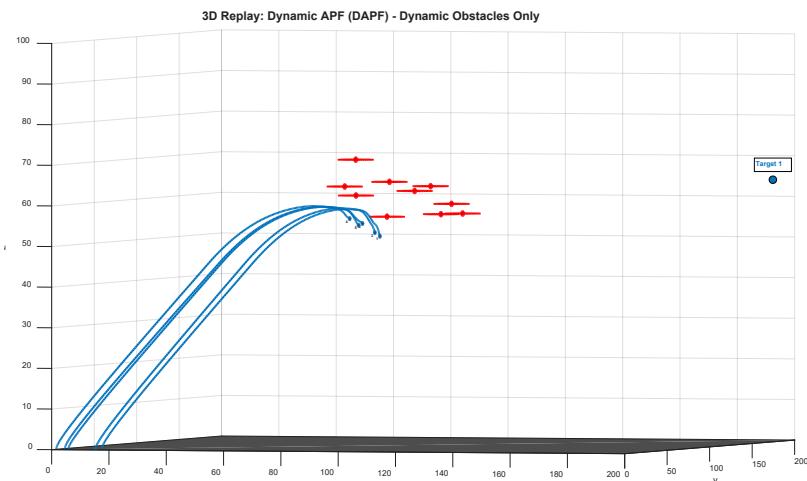
Tabel 4.7 Jarak Lintasan Agen *Obstacle* Belakang

| Agen | 1 | 2 | 3 | 4 | 5 | Total |
|------|--------|--------|--------|--------|--------|----------|
| DAPF | 323.58 | 320.19 | 311.72 | 311.25 | 310.40 | 1 577.14 |

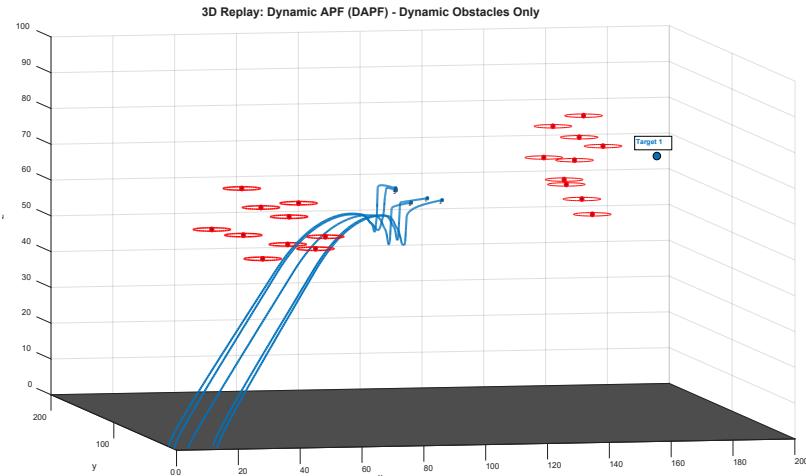
4.1.2.2 Skenario muncul sekuensial

a. Konfigurasi *obstacle head-on*

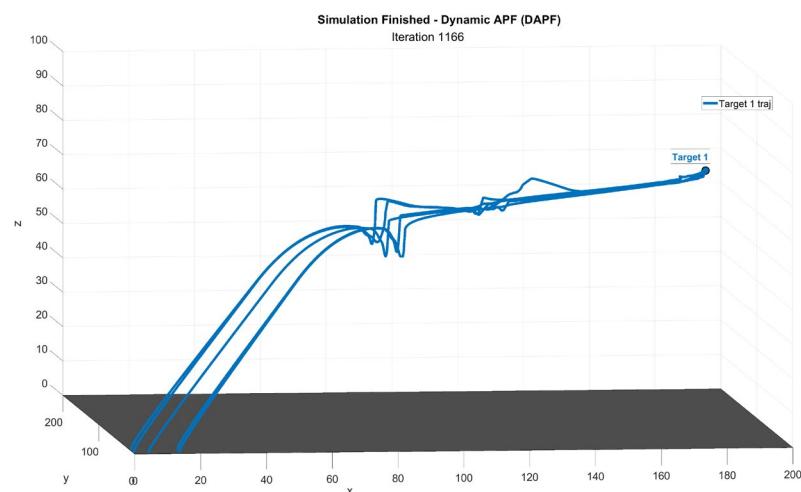
Pada konfigurasi dengan *obstacle* dinamis yang bergerak dari depan (*head-on*) menuju UAV secara sekuensial, sistem DAPF diuji kemampuannya dalam menangani ancaman tabrakan dari dua kali *obstacle* yang muncul pada simulasi yang terjadi. Respons UAV terhadap skenario ini ditampilkan pada Gambar 4.74 hingga Gambar 4.776, yang menggambarkan situasi pertama bertemu *obstacle dinamis*, saat bertemu kemunculan *obstacle* dinamis kedua, dan lintasan akhir setelah berdekatan dengan halangan dinamis. Pada urutan gambar tersebut terlihat bahwa UAV secara aktif melakukan manuver penghindaran terhadap *obstacle* yang bergerak dari arah depan. Pergerakan UAV menunjukkan perubahan arah dan lintasan untuk menghindari potensi tabrakan, lalu kembali mengarahkan diri menuju target yang telah ditentukan.



Gambar 4.74 Respons Agen saat *Obstacle* dari Depan Pertama

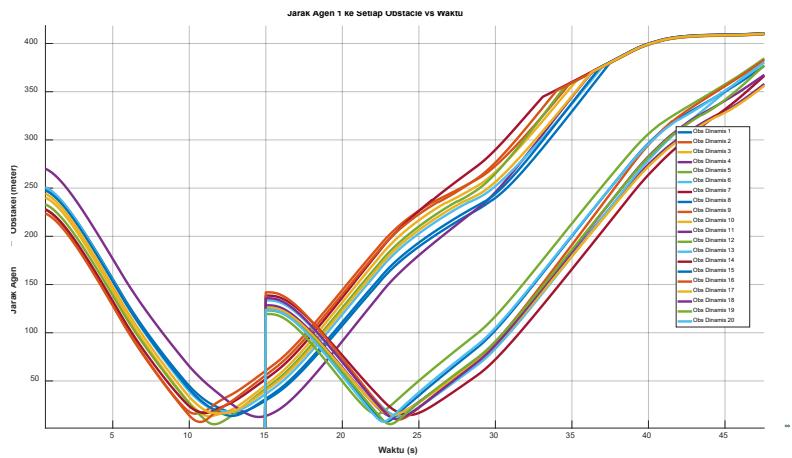


Gambar 4.75 Respons Agen saat *Obstacle* dari Depan Kedua

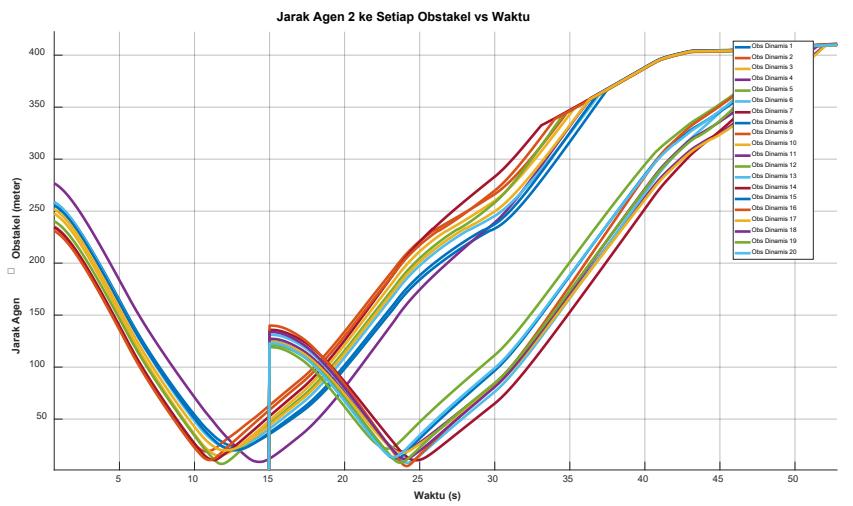


Gambar 4.76 Lintasan Agen Akhir

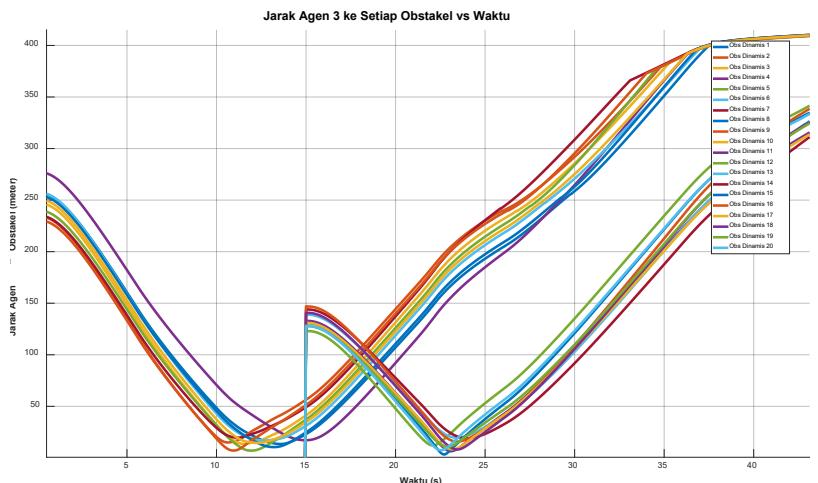
Untuk mengevaluasi kemampuan penghindaran, dilakukan analisis terhadap jarak antara masing-masing UAV dan *obstacle* dinamis sepanjang waktu simulasi. Hasilnya ditampilkan dalam Gambar 4.77 hingga Gambar 4.81 Jarak Agen 5 ke Setiap Obstacle. Semua grafik memperlihatkan beberapa kurva berbentuk "V", di mana jarak antara UAV dan *obstacle* awalnya terus menurun hingga mencapai titik minimum pada sekitar detik ke-9 hingga 15 dan 20 hingga 25, yang mengindikasikan momen kritis pertemuan jarak antara agen dan *obstacle*. Setelah itu, jarak kembali meningkat seiring dengan dilakukannya manuver penghindaran oleh UAV terhadap halangan yang bergerak dari arah depan. Tidak terdapat titik nol pada grafik, dengan kedekatan minimum yang terjadi adalah 3.6 meter.



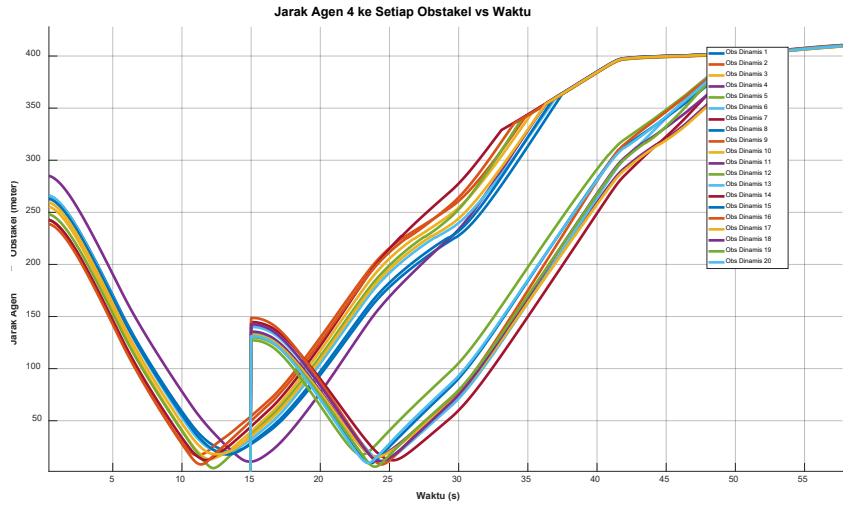
Gambar 4.77 Jarak Agen 1 ke Setiap *Obstacle*



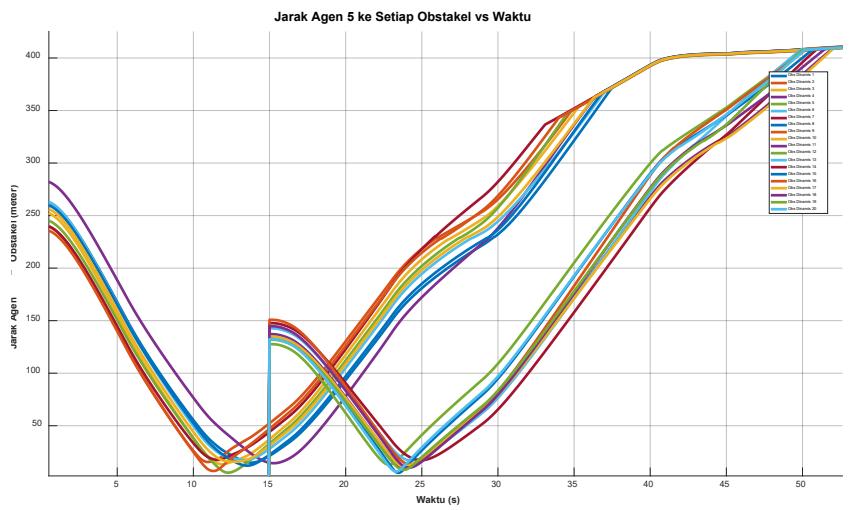
Gambar 4.78 Jarak Agen 2 ke Setiap *Obstacle*



Gambar 4.79 Jarak Agen 3 ke Setiap *Obstacle*

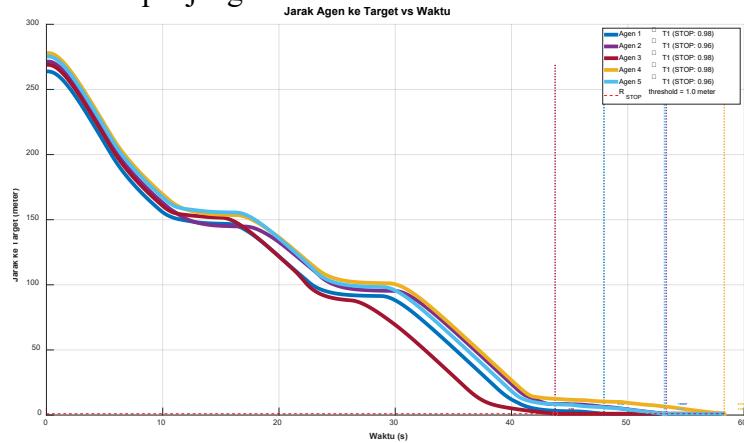


Gambar 4.80 Jarak Agen 4 ke Setiap *Obstacle*

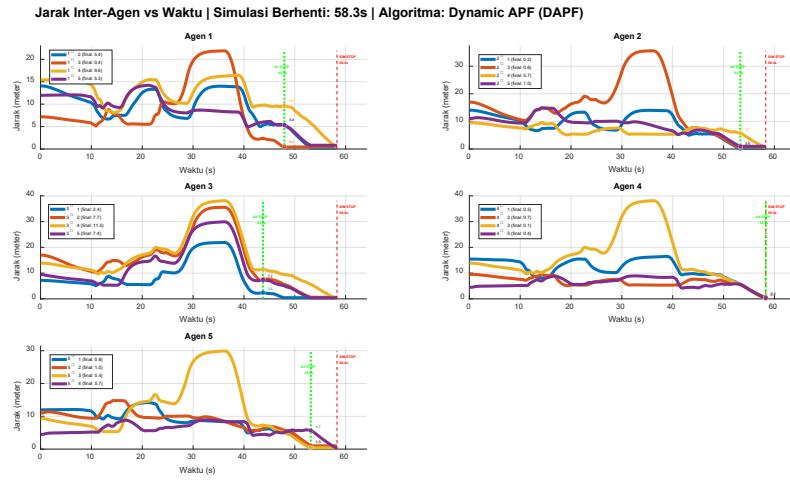


Gambar 4.81 Jarak Agen 5 ke Setiap *Obstacle*

Pada Gambar 4.82 memperlihatkan grafik jarak masing-masing UAV terhadap target sepanjang waktu simulasi. Pola penurunan yang konsisten menunjukkan bahwa setiap agen secara progresif mendekati target hingga mencapai titik tujuan. Sementara itu, Gambar 4.83 menunjukkan jarak antar agen selama manuver berlangsung. Seluruh agen mampu menjaga jarak aman satu sama lain sepanjang simulasi.



Gambar 4.82 Jarak Agen ke Target



Gambar 4.83 Jarak Antar Agen

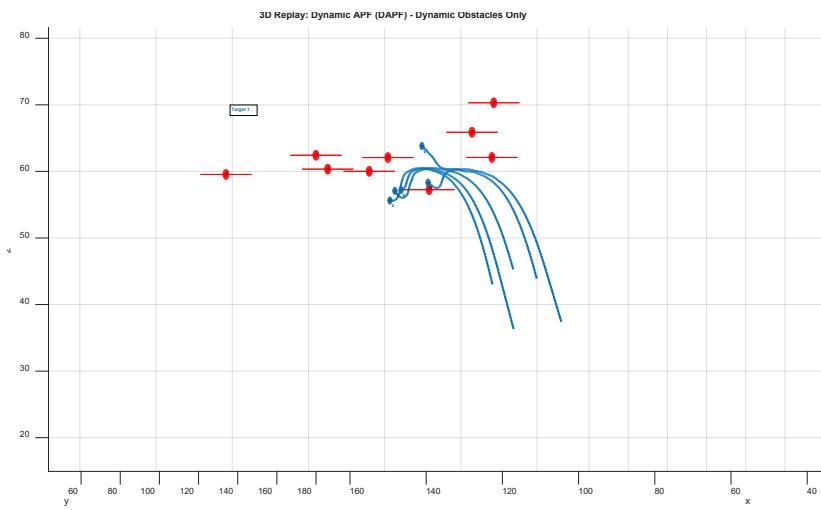
Dari hasil simulasi, dapat disimpulkan bahwa seluruh agen berhasil mencapai target dengan aman tanpa terjadi tabrakan, baik terhadap *obstacle* maupun sesama agen. Sistem DAPF menunjukkan kemampuan dalam menghadapi ancaman frontal sekuensial. Total waktu simulasi tercatat sebesar 58.3 detik, dengan jarak minimum antara agen dan *obstacle* sebesar 3.6 meter dan jarak antar agen minimum sebesar 2.4 meter. Panjang lintasan yang ditempuh oleh masing-masing UAV dalam skenario ini ditampilkan pada Tabel 4.5.

Tabel 4.8 Jarak Lintasan Agen

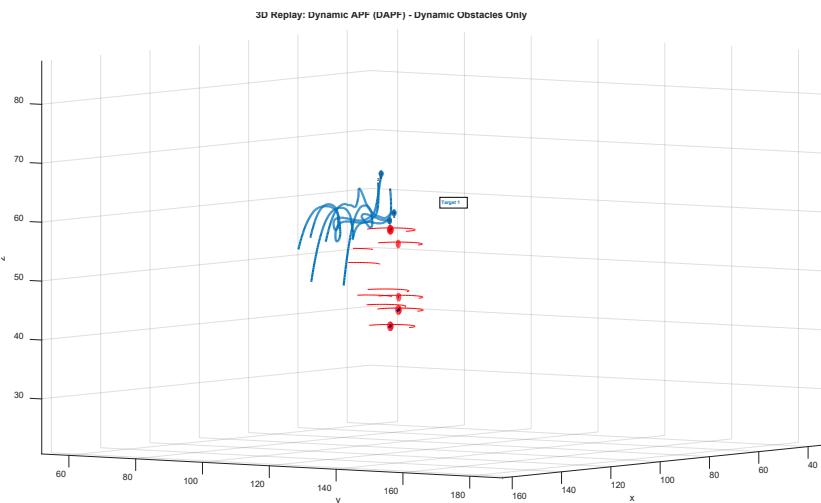
| Agen | 1 | 2 | 3 | 4 | 5 | Total |
|------|--------|--------|--------|--------|--------|---------|
| DAPF | 288.48 | 295.92 | 294.34 | 305.85 | 302.20 | 1486.79 |

b. Konfigurasi *obstacle side*

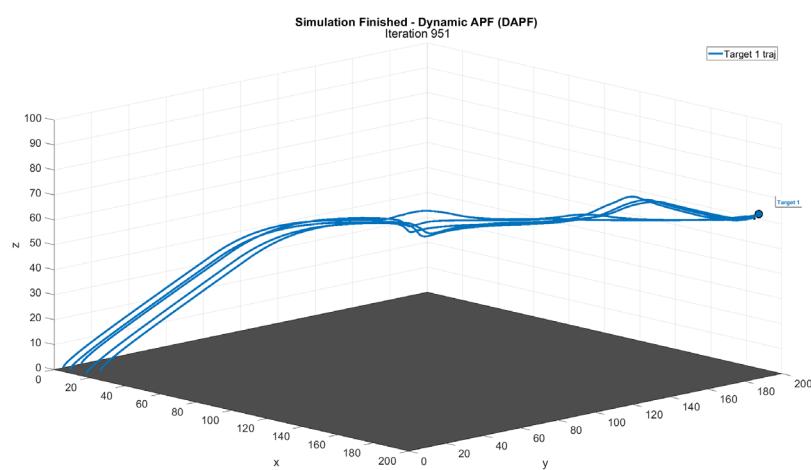
Pada konfigurasi dengan *obstacle* dinamis yang bergerak dari samping (*side*) menuju UAV secara sekuensial, sistem DAPF diuji kemampuannya dalam menangani ancaman tabrakan dari dua kali *obstacle* yang muncul pada simulasi yang terjadi. Respons UAV terhadap skenario ini ditampilkan pada Gambar 4.74 hingga Gambar 4.7786, yang menggambarkan situasi pertama bertemu *obstacle dinamis*, saat bertemu kemunculan *obstacle* dinamis kedua, dan lintasan akhir setelah berdekatan dengan halangan dinamis. Pada urutan gambar tersebut terlihat bahwa UAV secara aktif melakukan manuver penghindaran terhadap *obstacle* yang bergerak dari arah samping. Pergerakan UAV menunjukkan perubahan arah dan lintasan untuk menghindari potensi tabrakan, lalu kembali mengarahkan diri menuju target yang telah ditentukan.



Gambar 4.84 Respons Agen bertemu *Obstacle* dari Samping (pertama)



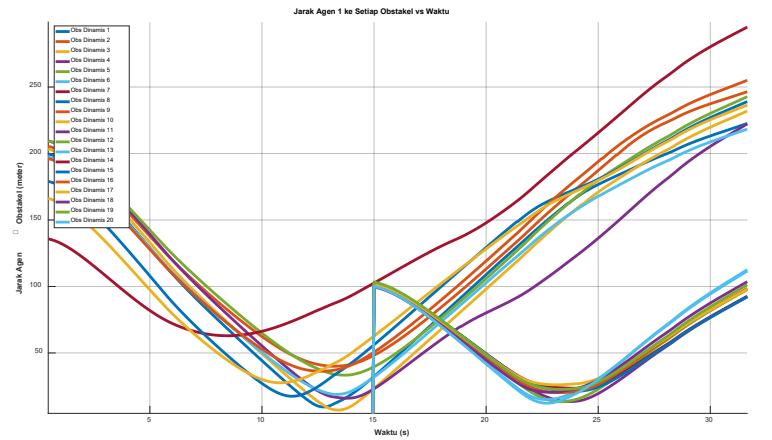
Gambar 4.85 Respons Agen bertemu *Obstacle* dari Samping (kedua)



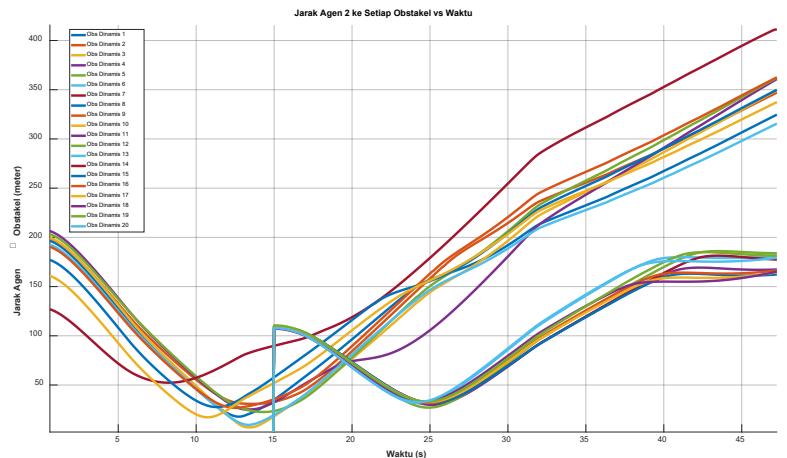
Gambar 4.86 Kesuluruhan Lintasan Agen untuk *Obstacle* dari Samping

Untuk mengevaluasi kemampuan penghindaran, dilakukan analisis terhadap jarak antara masing-masing UAV dan *obstacle* dinamis sepanjang waktu simulasi. Hasilnya ditampilkan

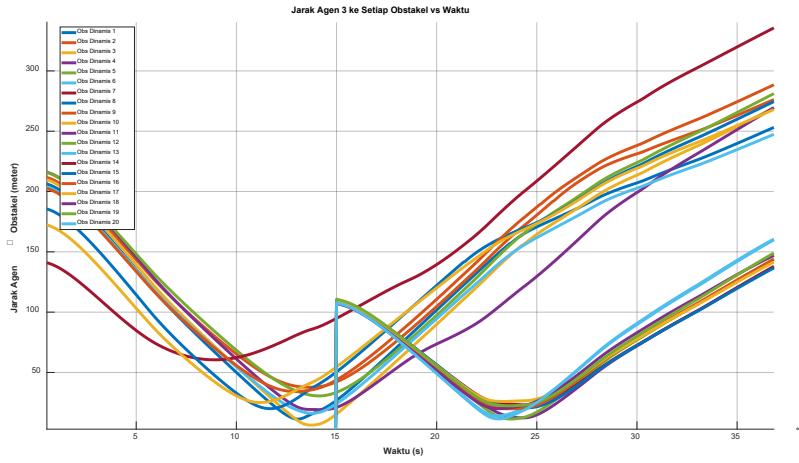
dalam Gambar 4.77 hingga Gambar 4.77. Semua grafik memperlihatkan beberapa kurva berbentuk "V", di mana jarak antara UAV dan *obstacle* awalnya turun terus menurun hingga mencapai titik minimum pada sekitar detik ke-11 hingga 15 dan 20 hingga 25, yang mengindikasikan momen kritis pertemuan jarak antara agen dan *obstacle*. Setelah itu, jarak kembali meningkat seiring dengan dilakukannya manuver penghindaran oleh UAV terhadap halangan yang bergerak dari arah samping. Tidak terdapat titik nol pada grafik, dengan kedekatan minimum yang terjadi antara UAV ke *Obstacle*, yaitu dengan jarak minimal 5 meter pada agen 4 dan 5.



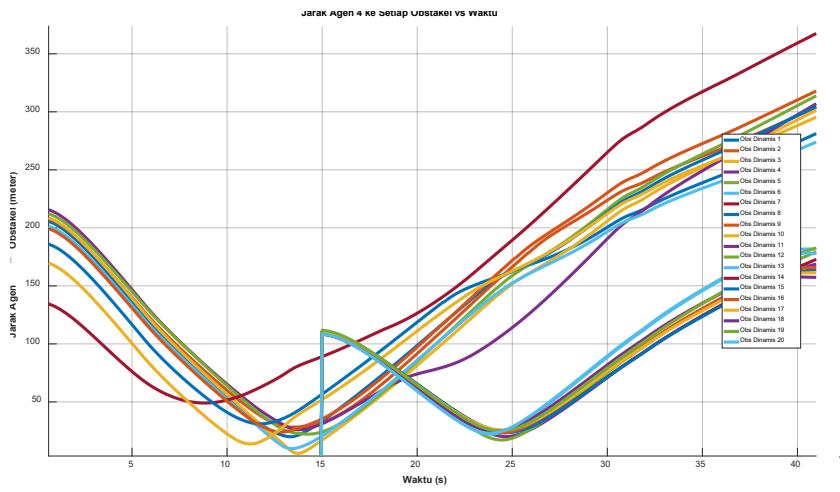
Gambar 4.87 Jarak Agen 1 ke Setiap *Obstacle*



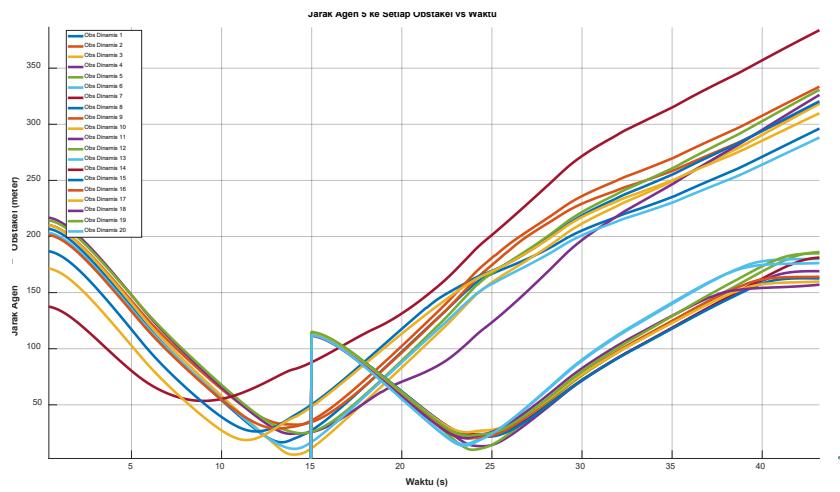
Gambar 4.88 Jarak Agen 2 ke Setiap *Obstacle*



Gambar 4.89 Jarak Agen 3 ke Setiap *Obstacle*



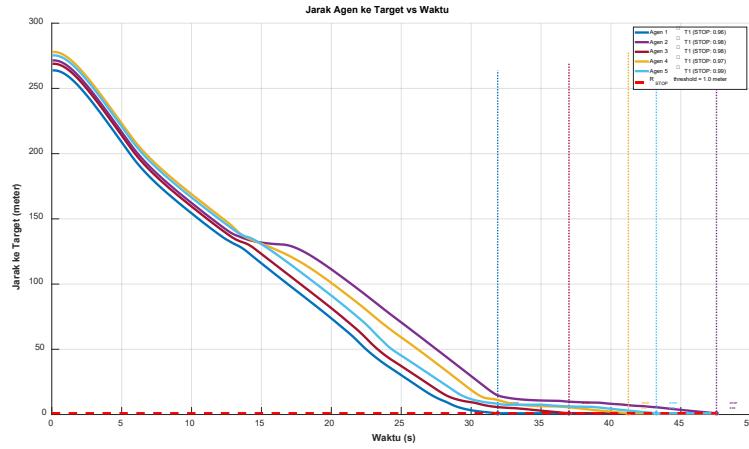
Gambar 4.90 Jarak Agen 4 ke Setiap *Obstacle*



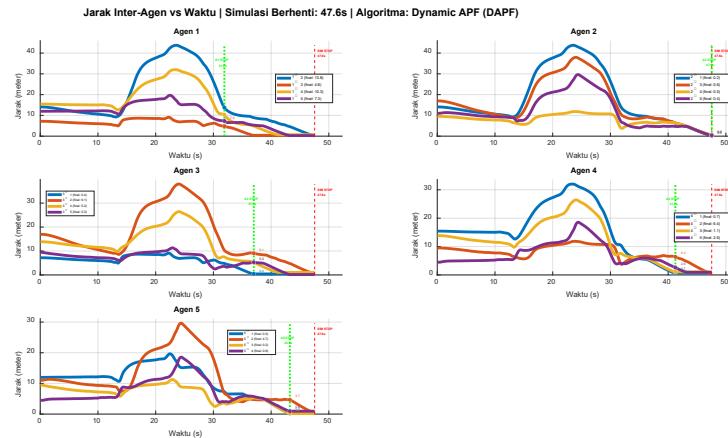
Gambar 4.91 Jarak Agen 5 ke Setiap *Obstacle*

Pada Gambar 4.82 memperlihatkan grafik jarak masing-masing UAV terhadap target sepanjang waktu simulasi. Pola penurunan yang konsisten menunjukkan bahwa setiap agen secara

progresif mendekati target hingga mencapai titik tujuan. Sementara itu, Gambar 4.83 menunjukkan jarak antar agen selama manuver berlangsung. Seluruh agen mampu menjaga jarak aman satu sama lain sepanjang simulasi.



Gambar 4.92 Jarak Agen ke Target



Gambar 4.93 Jarak Antar Agen

Dari hasil simulasi, dapat disimpulkan bahwa seluruh agen berhasil mencapai target dengan aman tanpa terjadi tabrakan, baik terhadap *obstacle* maupun sesama agen. Sistem DAPF menunjukkan kemampuan dalam menghadapi ancaman *lateral* sekuensial. Total waktu simulasi tercatat sebesar 47.6 detik, dengan jarak minimum antara agen dan *obstacle* sebesar 5 meter dan jarak antar agen minimum sebesar 2.4 meter. Panjang lintasan yang ditempuh oleh masing-masing UAV dalam skenario ini ditampilkan pada Tabel 4.5.

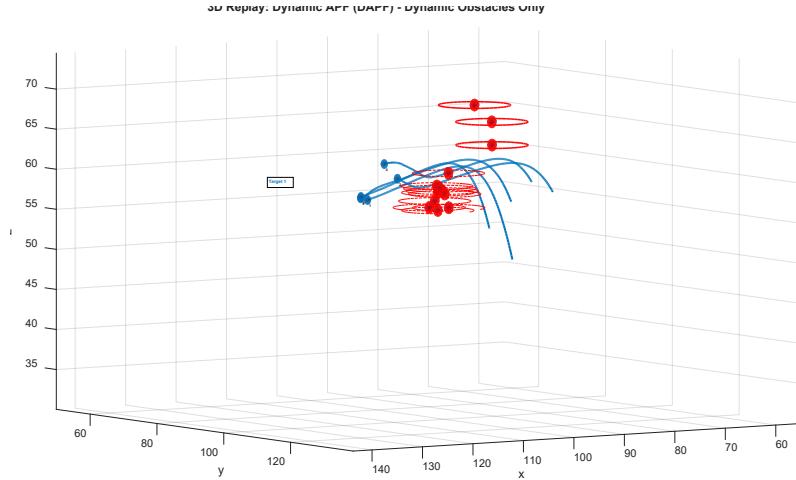
Tabel 4. 9 Jarak Lintasan Agen

| Agen | 1 | 2 | 3 | 4 | 5 | Total |
|------|--------|--------|--------|--------|--------|----------|
| DAPF | 278.99 | 285.16 | 286.97 | 291.88 | 294.02 | 1 437.02 |

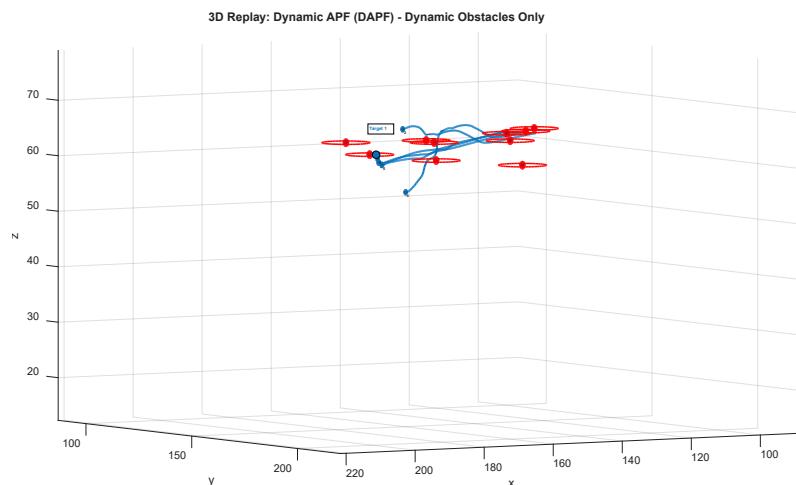
c. Konfigurasi *obstacle back*

Pada konfigurasi dengan *obstacle* dinamis yang bergerak dari belakang (*back*) menuju UAV secara sekuensial, sistem DAPF diuji kemampuannya dalam menangani ancaman tabrakan dari dua kali *obstacle* yang muncul pada simulasi yang terjadi. Respons UAV terhadap skenario ini ditampilkan pada Gambar 4.74 hingga Gambar 4.7796, yang menggambarkan situasi sebelum,

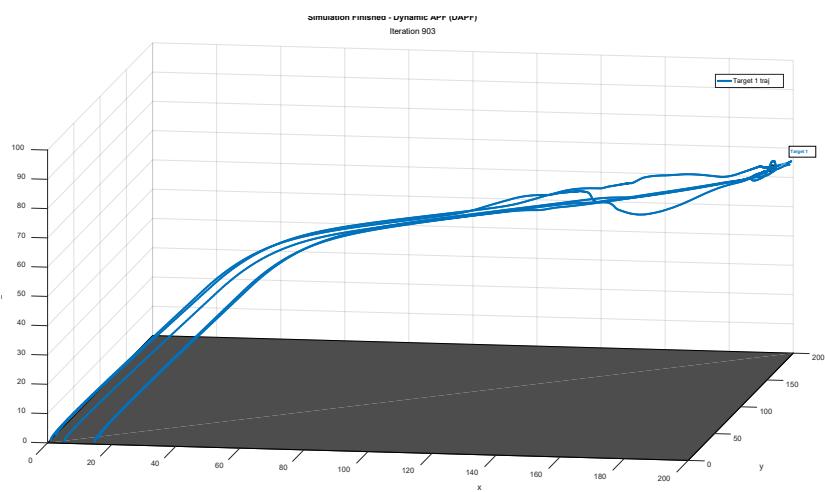
saat, dan setelah berdekatan dengan halangan dinamis. Pada urutan gambar tersebut terlihat bahwa UAV secara aktif melakukan manuver penghindaran terhadap *obstacle* yang bergerak dari arah depan. Pergerakan UAV menunjukkan perubahan arah dan lintasan untuk menghindari potensi tabrakan, lalu kembali mengarahkan diri menuju target yang telah ditentukan.



Gambar 4.94 Respons Agen bertemu *Obstacle* dari Belakang (pertama)

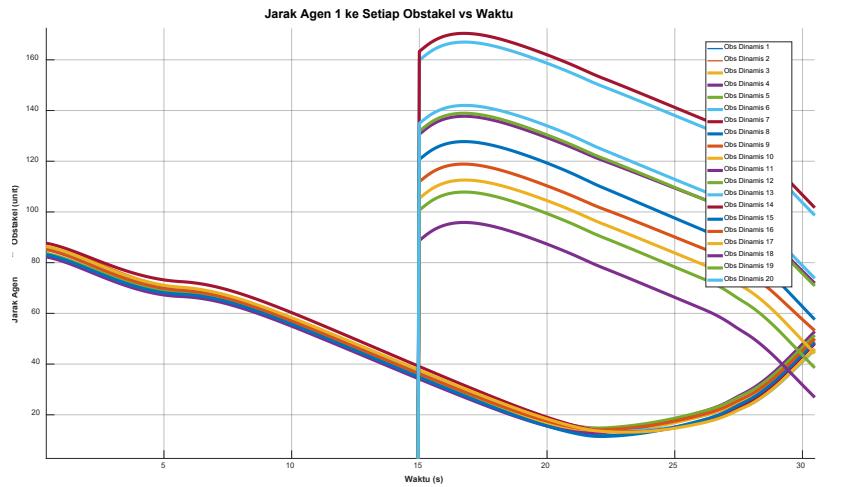


Gambar 4.95 Respons Agen bertemu *Obstacle* dari Belakang (kedua)

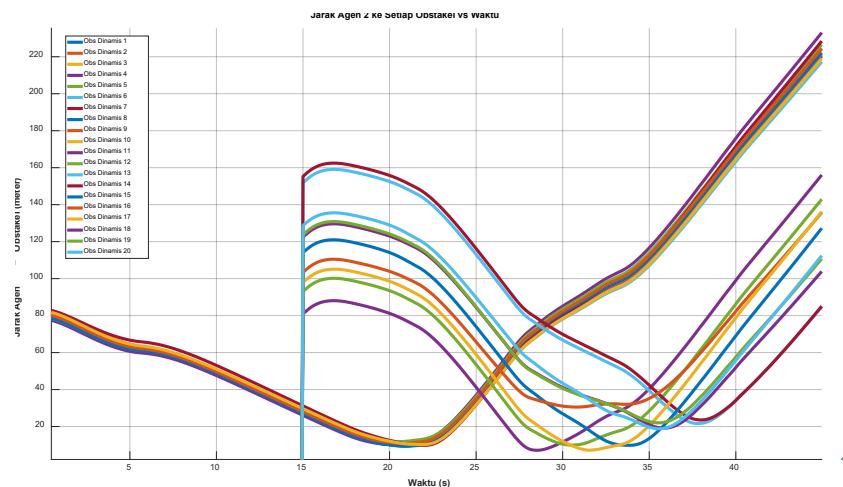


Gambar 4.96 Keseluruhan Lintasan Agen untuk *Obstacle* dari Belakang

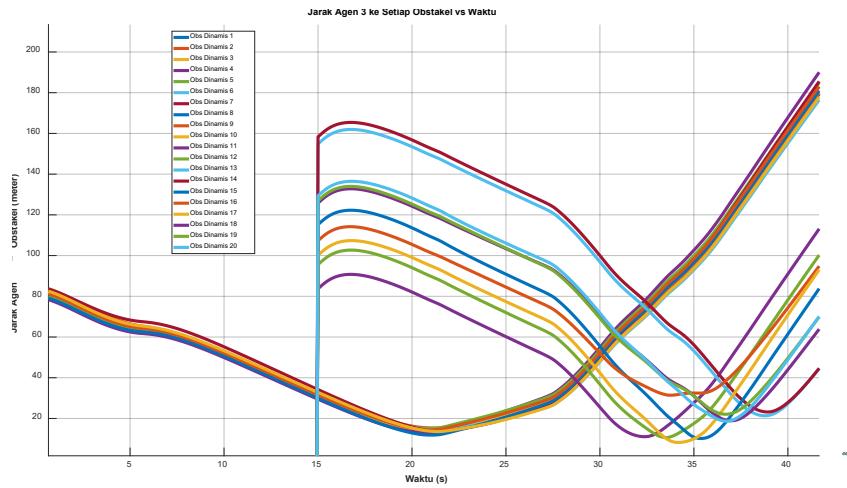
Untuk mengevaluasi kemampuan penghindaran, dilakukan analisis terhadap jarak antara masing-masing UAV dan *obstacle* dinamis sepanjang waktu simulasi. Hasilnya ditampilkan dalam Gambar 4.97 hingga Gambar 4.101. Semua grafik memperlihatkan beberapa kurva berbentuk "V", di mana jarak antara UAV dan *obstacle* awalnya turun terus menuju titik minimum pada sekitar detik ke-20 hingga 25, dan 25 hingga 35 yang mengindikasikan momen kritis pertemuan jarak antara agen dan *obstacle*. Setelah itu, jarak kembali meningkat seiring dengan dilakukannya manuver penghindaran oleh UAV terhadap halangan yang bergerak dari arah depan. Tidak terdapat titik nol pada grafik, dengan jarak minimal 7 meter antara agen dan *obstacle* dinamis.



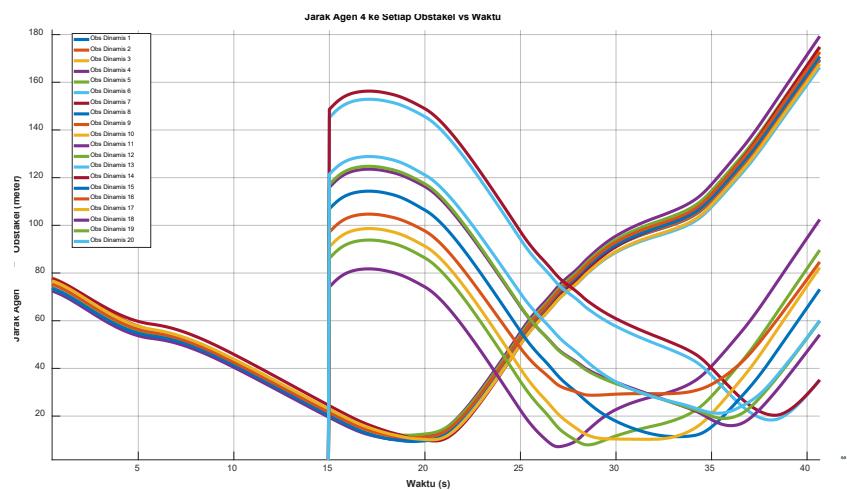
Gambar 4.97 Jarak Agen 1 ke Setiap *Obstacle*



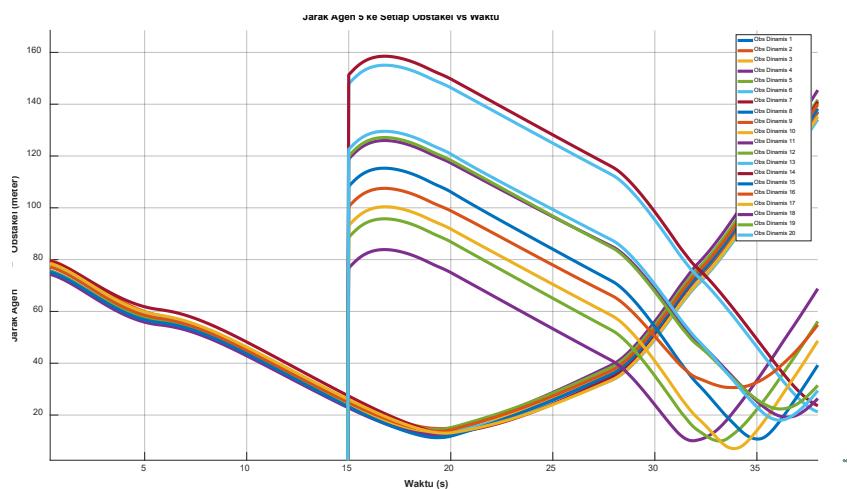
Gambar 4.98 Jarak Agen 2 ke Setiap *Obstacle*



Gambar 4.99 Jarak Agen 3 ke Setiap *Obstacle*



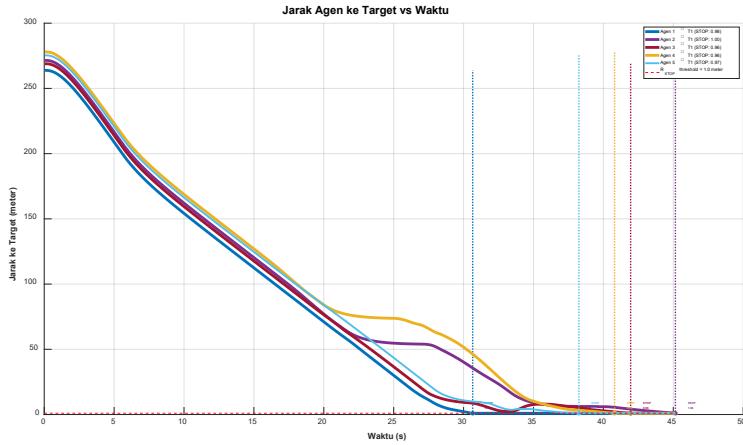
Gambar 4.100 Jarak Agen 4 ke Setiap *Obstacle*



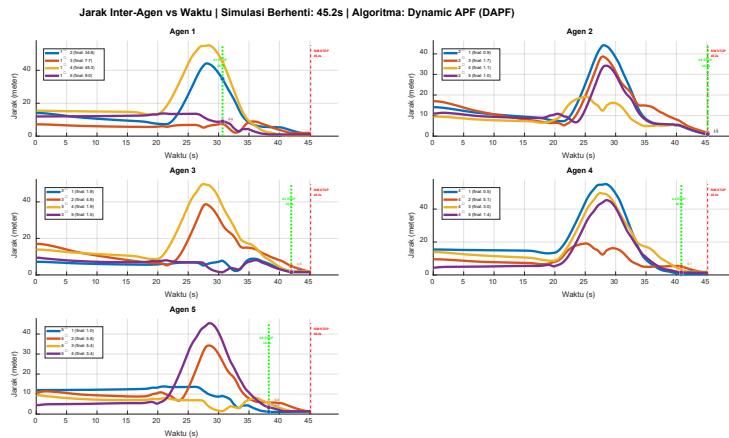
Gambar 4.101 Jarak Agen 5 ke Setiap *Obstacle*

Pada Gambar 4.82 memperlihatkan grafik jarak masing-masing UAV terhadap target sepanjang waktu simulasi. Pola penurunan yang konsisten menunjukkan bahwa setiap agen secara

progresif mendekati target hingga mencapai titik tujuan. Sementara itu, Gambar 4.83 menunjukkan jarak antar agen selama manuver berlangsung. Seluruh agen mampu menjaga jarak aman satu sama lain sepanjang simulasi.



Gambar 4.102 Jarak Agen ke Target



Gambar 4.103 Jarak Antar Agen

Dari hasil simulasi, dapat disimpulkan bahwa seluruh agen berhasil mencapai target dengan aman tanpa terjadi tabrakan, baik terhadap *obstacle* maupun sesama agen. Sistem DAPF menunjukkan kemampuan dalam menghadapi ancaman back sekvensial. Total waktu simulasi tercatat sebesar 55.76 detik, dengan jarak minimum antara agen dan *obstacle* sebesar 7 meter dan jarak antar agen minimum sebesar 2.1 meter. Panjang lintasan yang ditempuh oleh masing-masing UAV dalam skenario ini ditampilkan pada Tabel 4.510.

Tabel 4. 10 Jarak Lintasan Agen

| Agen | 1 | 2 | 3 | 4 | 5 | Total |
|------|--------|--------|--------|--------|--------|----------|
| DAPF | 274.63 | 286.37 | 297.92 | 295.17 | 294.92 | 1 449.01 |

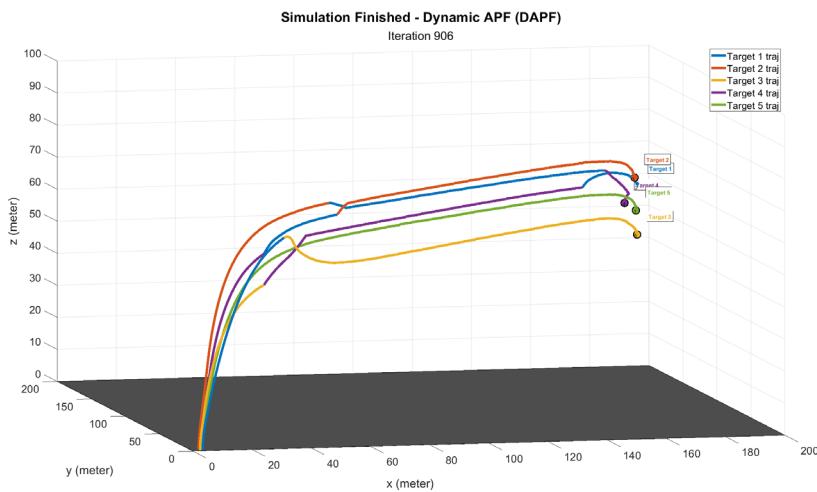
4.2 Simulasi Pembagian Tugas Terdistribusi UAV

4.2.1 Konfigurasi jumlah UAV dan target

Pada subbab ini akan ditampilkan dua konfigurasi Jumlah UAV dan Jumlah Target, konfigurasi UAV dan Target dengan jumlah yang sama (5 Agen dan 5 Target), serta UAV berjumlah lebih banyak dibandingkan Target (5 Agen dan 3 Target)

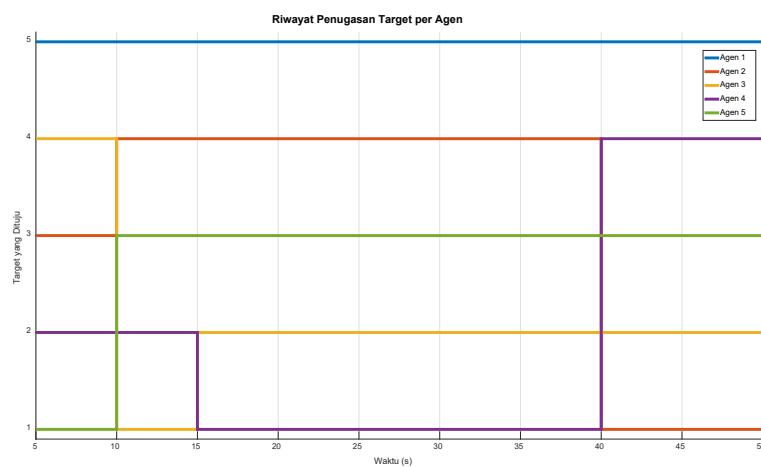
- a. UAV dan target berjumlah sama (5 agen dan 5 target)

Pada konfigurasi ini, digunakan lima UAV dan lima target dengan distribusi jumlah yang seimbang. Sistem *Distributed Assignment Switch* (DAS) melakukan pembagian tugas secara dinamis dengan rasio agen dan target yang sama. Jalur lintasan tiap UAV ditunjukkan pada Gambar 4.104, di mana kelima agen bergerak menuju target masing-masing.



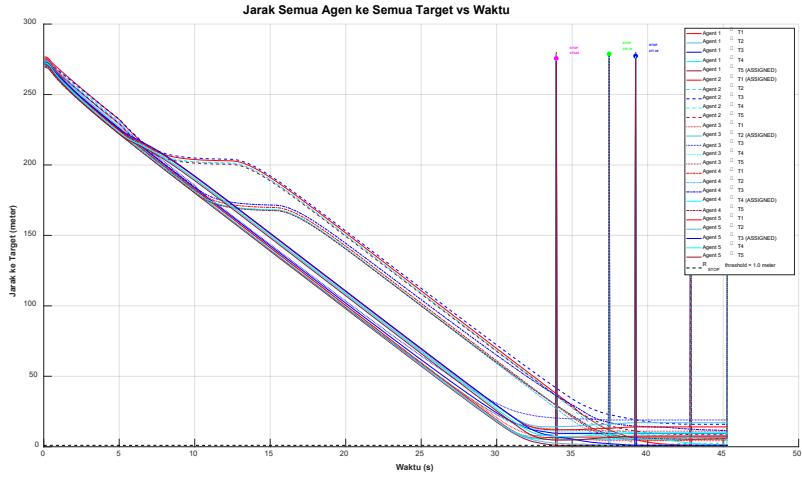
Gambar 4.104 Lintasan Agen berjumlah sama dengan Target

Visualisasi pada Gambar 4.105 menunjukkan bahwa setiap UAV diberikan penugasan target tertentu secara *real-time* dan bersifat dinamis. Terlihat bahwa beberapa agen mengalami perubahan target di tengah simulasi, yang mempertimbangkan jarak dan prioritas dari target. Pada Gambar 4.104 dapat terlihat bahwa seluruh UAV terdistribusi secara merata terhadap lima target yang tersedia.



Gambar 4. 105 Pergantian Penugasan Agen

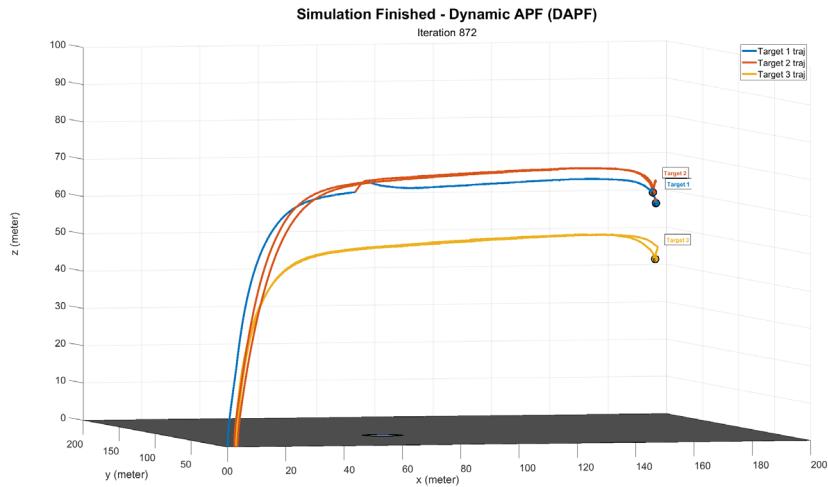
Pada Gambar 4.106, grafik menunjukkan tren penurunan jarak seluruh agen terhadap seluruh target secara progresif. Seluruh UAV berhasil mencapai target masing-masing dalam waktu kurang dari 45 detik, dengan jarak akhir yang menyentuh ambang minimum.



Gambar 4.106 Jarak Semua Agen ke Semua Target terhadap Waktu

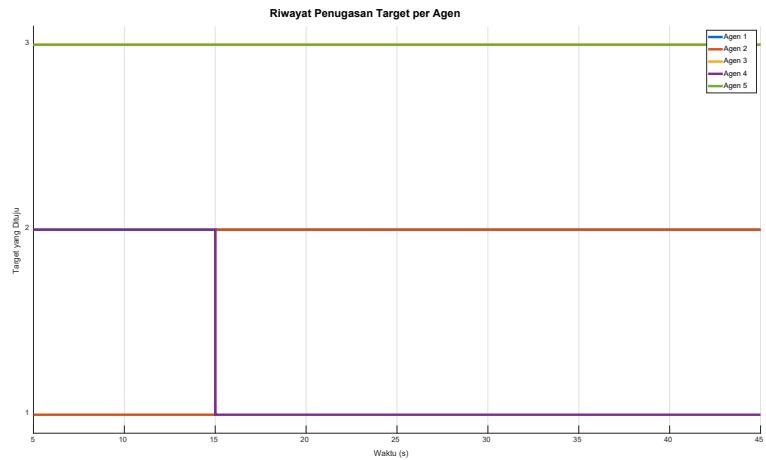
b. UAV dan target berjumlah beda (5 Agen dan 3 Target)

Pada konfigurasi ini, jumlah UAV lebih banyak dibanding jumlah target, yaitu lima agen dan hanya tiga target. Ketidakseimbangan ini menghasilkan kondisi kompetitif dalam pembagian target. Jalur lintasan tiap UAV ditunjukkan pada Gambar 4.107, di mana kelima agen bergerak menuju target masing-masing, dimana juga menunjukkan bahwa dua target teralokasikan oleh lebih dari satu UAV secara bersamaan, yang memicu pengelompokan lintasan.



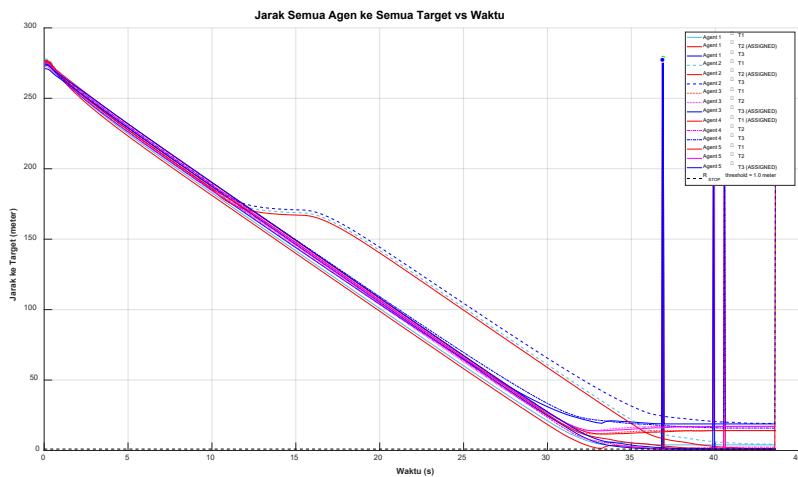
Gambar 4.107 Lintasan Agen berjumlah lebih dari Target

Gambar 4.108 memperlihatkan riwayat penugasan target, di mana tiga dari lima agen menerima penugasan ke tiga target berbeda secara permanen. Sementara dua agen lainnya harus berbagi target dengan UAV lain, dan mengalami perubahan penugasan di tengah simulasi.



Gambar 4.108 Pergantian Penugasan Agen

Gambar 4.109 memperlihatkan tren penurunan jarak yang tetap konvergen meskipun ada kompetisi target. Terlihat bahwa beberapa agen membutuhkan waktu lebih lama untuk menyelesaikan tugasnya, terutama agen yang mengalami perubahan target



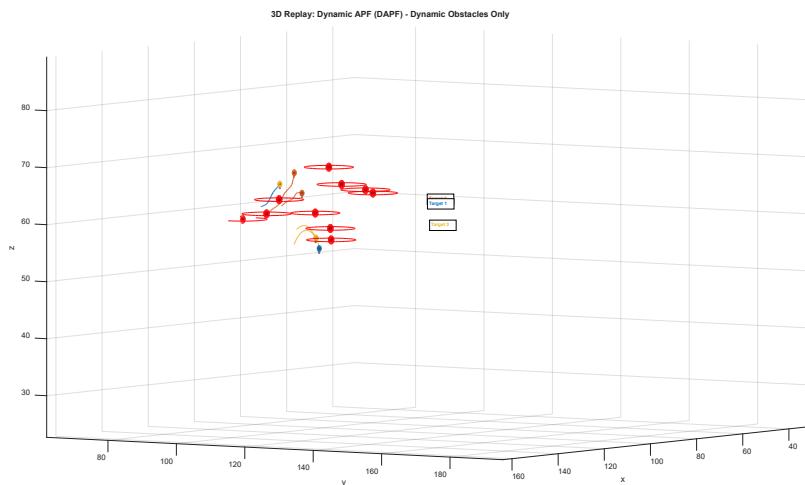
Gambar 4.109 Jarak Semua Agen ke Semua Target terhadap Waktu

4.3 Simulasi Pembagian Tugas dan Penghindaran Halangan

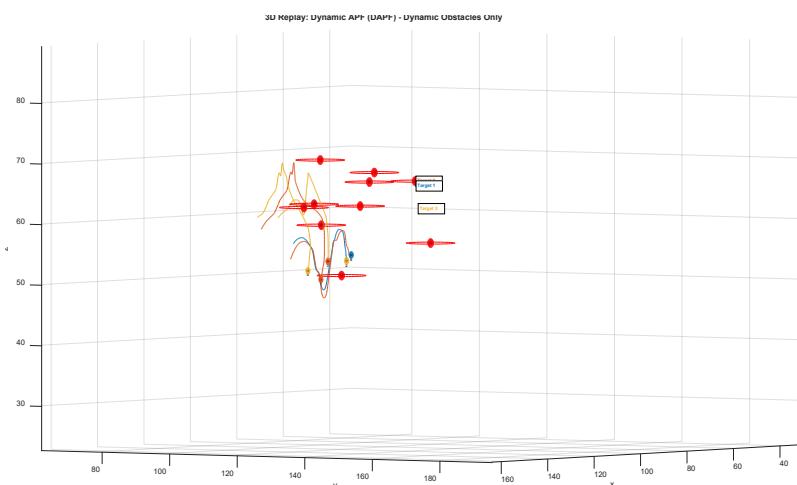
Dilakukan pengujian dengan menggabungkan pembagian tugas dan penghindaran rintangan, dengan mengambil skenario yaitu penghindaran 20 halangan dinamis, yang datangnya dari depan agen secara sekuensial. Dengan menggunakan agen berjumlah 5 dan target berjumlah 3, didapatkan hasil simulasi menggunakan DAPF pada Gambar 4.74 hingga Gambar 4.763 sedangkan menggunakan MAPF pada Gambar 4.114 hingga Gambar 4.117. Terlihat terdapat manuver yang tidak diperlukan pada lintasan MAPF setelah menghindari rintangan, terlihat terjadinya gerakan belok untuk mencapai target. Dapat dilihat juga kelima agen bergerak menuju target masing-masing, dimana target terdekat dialokasikan 2 UAV, terlihat untuk DAPF target 1 dan target 2 teralokasikan oleh dua UAV secara bersamaan dan target 3 dialokasikan 1 UAV, sedangkan MAPF target 2 dan target 3 teralokasikan oleh dua UAV secara bersamaan dan target 3 dialokasikan 1 UAV. Diketahui panjang lintasan kelima agen pada dengan panjang lintasan MAPF lebih pendek 3.58% dari DAPF.

Tabel 4.11 Jarak Lintasan Agen

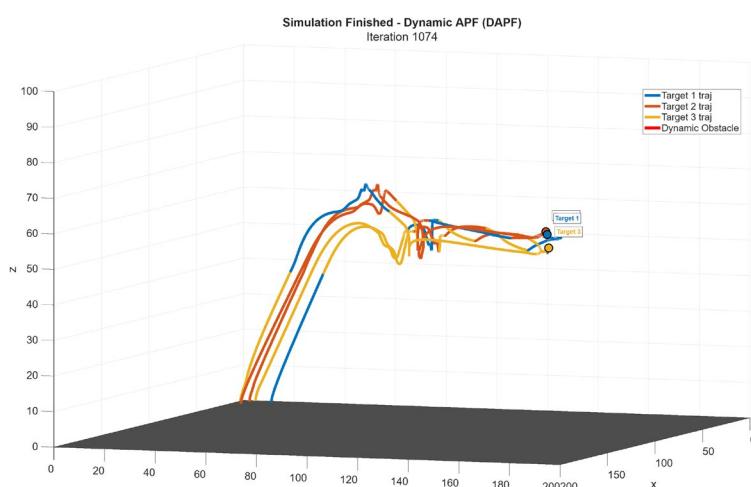
| Metode | Agen 1 | Agen 2 | Agen 3 | Agen 4 | Agen 5 | Total |
|--------|--------|--------|--------|--------|--------|----------|
| DAPF | 297.41 | 307.28 | 304.31 | 301.67 | 302.55 | 1 513.22 |
| MAPF | 280.42 | 296.66 | 297.10 | 294.14 | 290.79 | 1 459.11 |



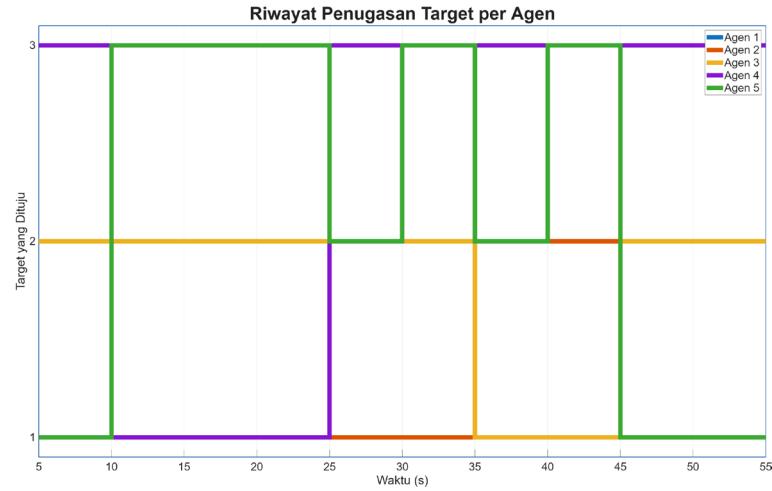
Gambar 4.110 Respons Agen saat *Obstacle* dari Depan Pertama



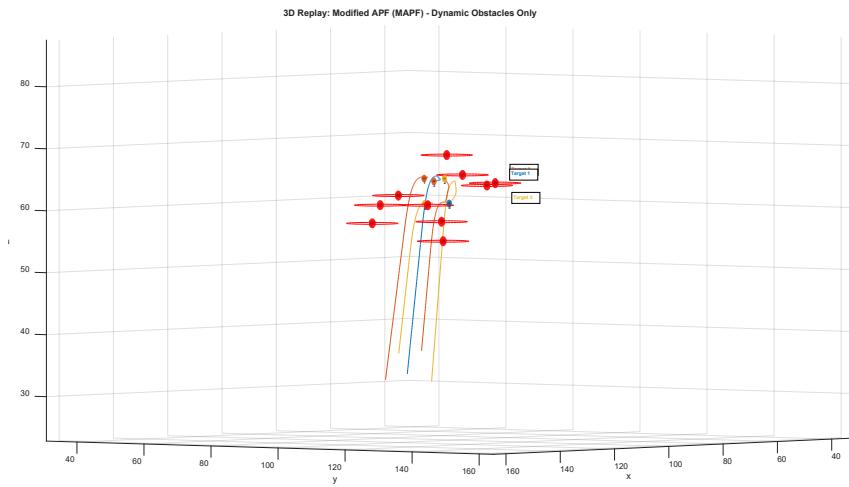
Gambar 4.111 Respons Agen saat *Obstacle* dari Depan Kedua



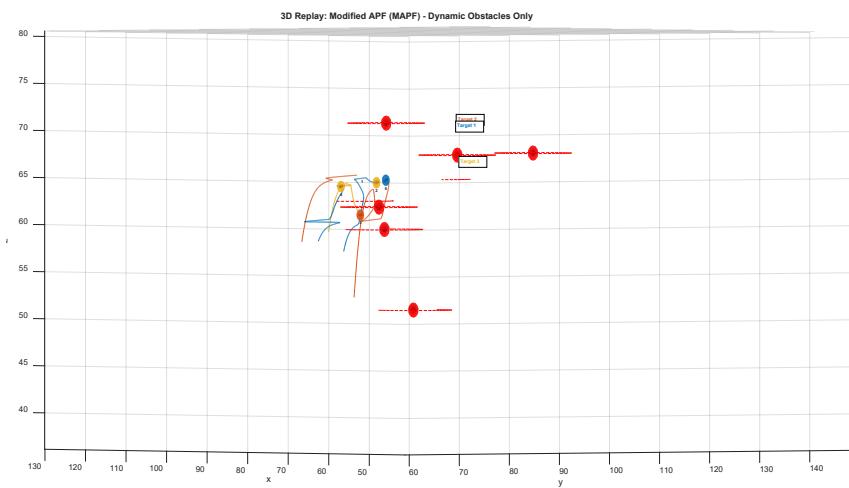
Gambar 4.112 Kesuluruhan Lintasan Agen untuk *Obstacle* dari Depan DAPF



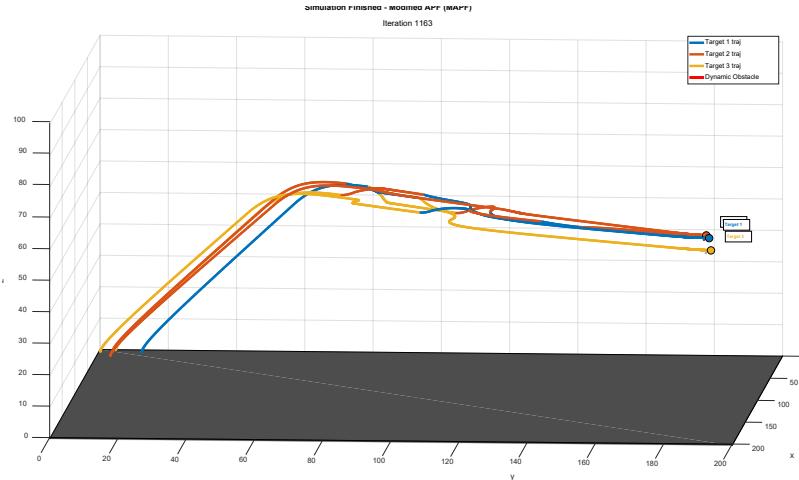
Gambar 4.113 Riwayat Penugasan DAPF



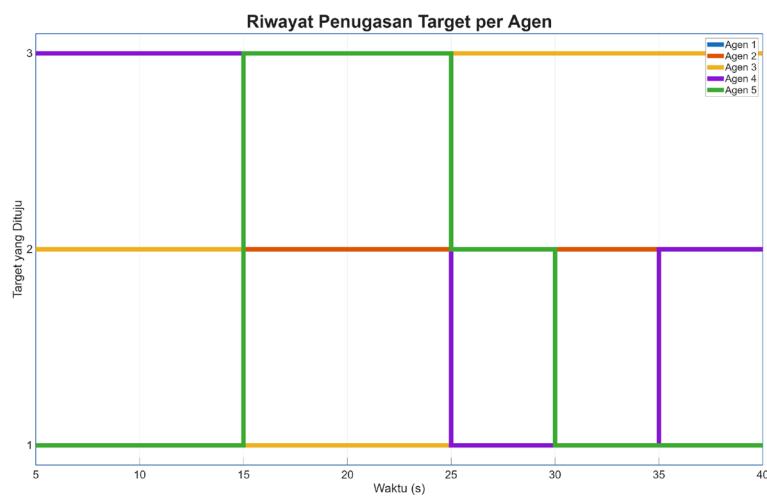
Gambar 4.114 Respons Agen saat *Obstacle* dari Depan Pertama MAPF



Gambar 4.115 Respons Agen saat *Obstacle* dari Depan Kedua MAPF

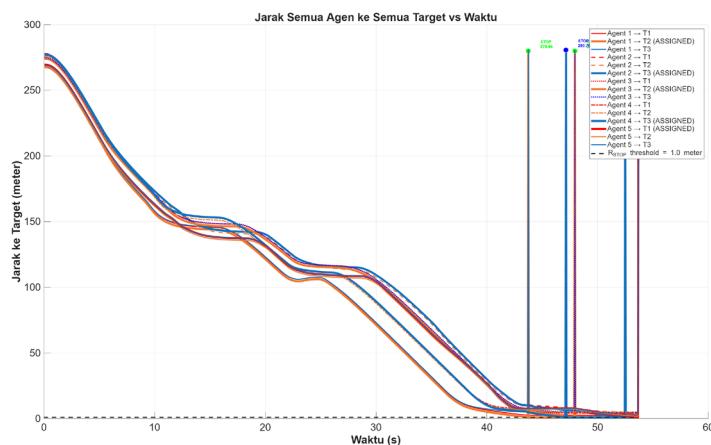


Gambar 4.116 Kesuluruhan Lintasan Agen untuk *Obstacle* dari Depan MAPF



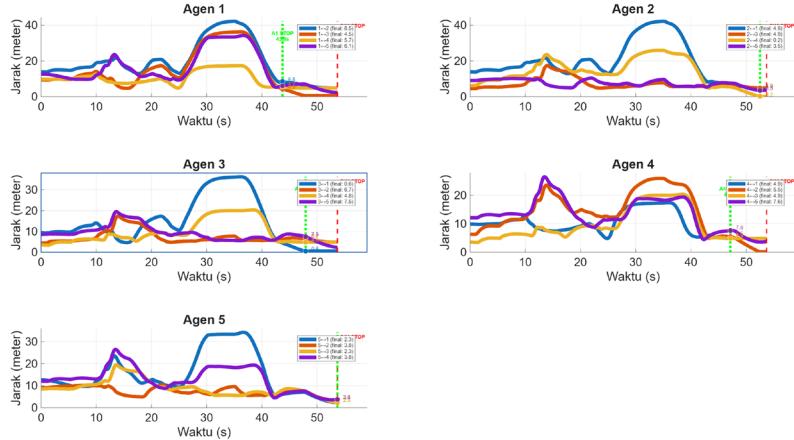
Gambar 4.117 Riwayat Penugasan MAPF

Pada Gambar 4.82118 dan Gambar 4.121 memperlihatkan grafik jarak masing-masing UAV terhadap target sepanjang waktu simulasi. Pola penurunan yang konsisten menunjukkan bahwa setiap agen secara progresif mendekati target hingga mencapai titik tujuan baik MAPF maupun DAPF. Sementara itu, Gambar 4.83 dan Gambar 4.120 menunjukkan jarak antar agen selama manuver berlangsung. Seluruh agen mampu menjaga jarak aman satu sama lain sepanjang simulasi, dengan jarak minimum MAPF yaitu 1.2 meter sedangkan untuk DAPF yaitu 4.5 meter. MAPF memiliki waktu simulasi yang lebih singkat sebesar 39.5 detik, sedangkan DAPF selama 53.7 detik.



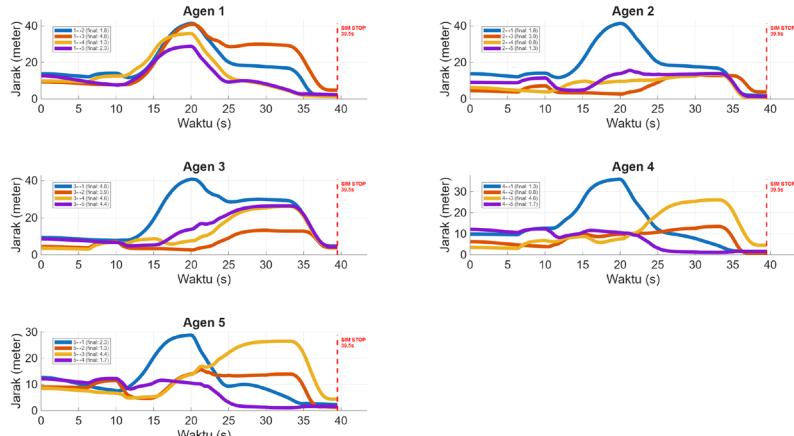
Gambar 4.118 Jarak Agen ke Semua Target DAPF

Jarak Inter-Agen vs Waktu | Simulasi Berhenti: 53.7s | Algoritma: Dynamic APF (DAPF)



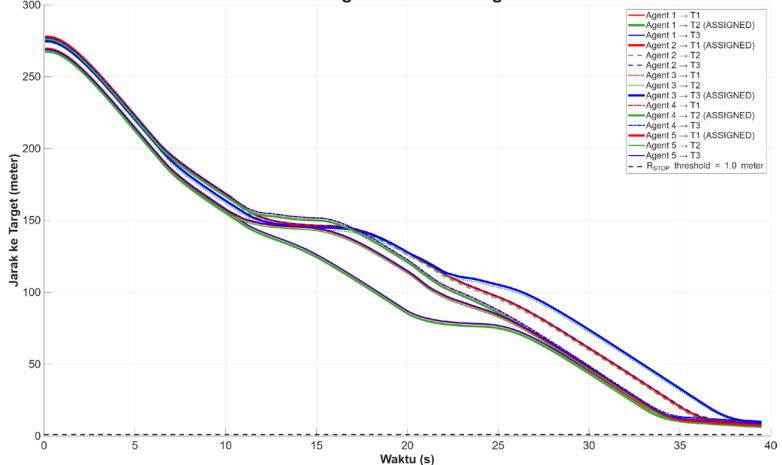
Gambar 4.119 Jarak antar Agen DAPF

Jarak Inter-Agen vs Waktu | Simulasi Berhenti: 39.5s | Algoritma: Modified APF (MAPF)



Gambar 4.120 Jarak antar Agen MAPF

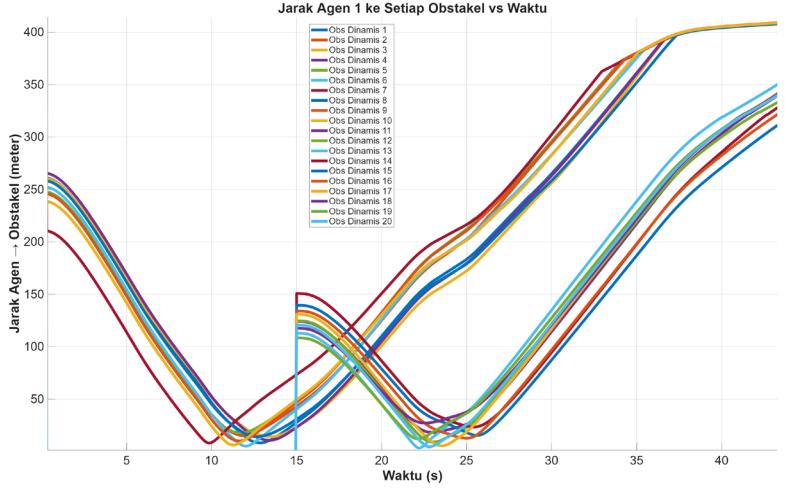
Jarak Semua Agen ke Semua Target vs Waktu



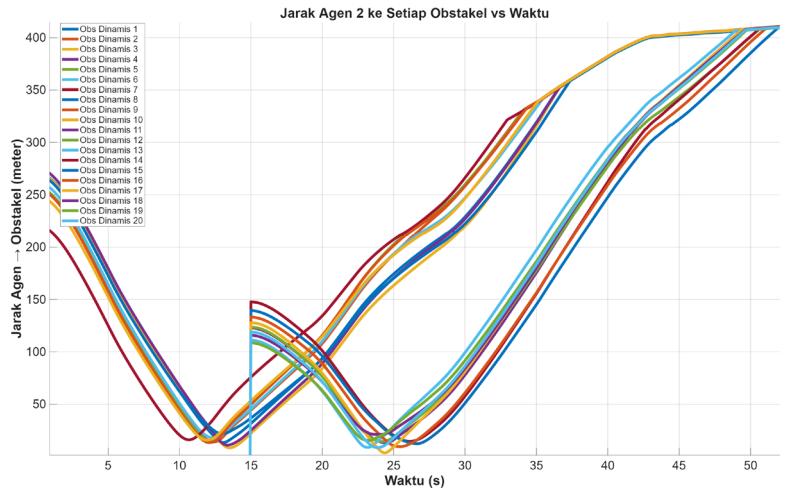
Gambar 4.121 Jarak Agen ke Semua Target MAPF

Untuk mengevaluasi kemampuan penghindaran, dilakukan analisis terhadap jarak antara masing-masing UAV dan *obstacle* dinamis sepanjang waktu simulasi. Hasilnya ditampilkan dalam Gambar 4.77 hingga Gambar 4.77. Semua grafik memperlihatkan beberapa kurva

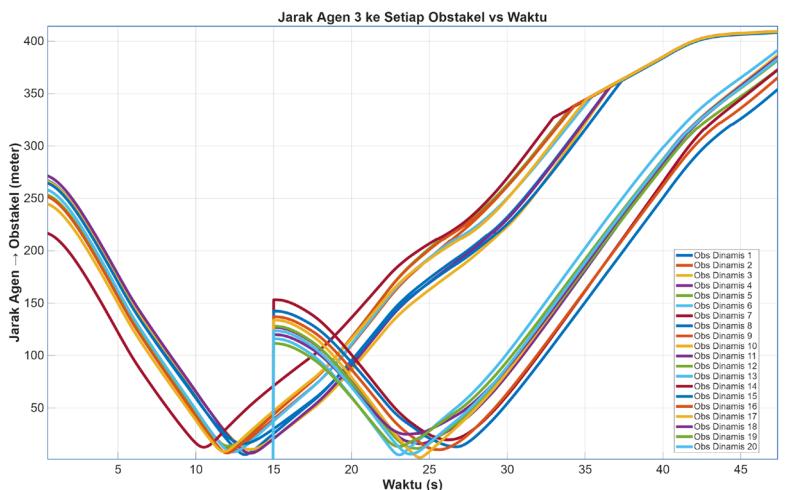
berbentuk "V", di mana jarak antara UAV dan *obstacle* awalnya terus menurun hingga mencapai titik minimum untuk DAPF dan MAPF pada sekitar detik ke-8 hingga 15 dan 20 hingga 25, yang mengindikasikan momen kritis pertemuan jarak antara agen dan *obstacle*. Setelah itu, jarak kembali meningkat seiring dengan dilakukannya manuver penghindaran oleh UAV terhadap halangan yang bergerak dari arah depan. Pada DAPF jarak minimum antara *obstacle* dan semua agen yaitu 2.5 meter, sedangkan MAPF memiliki nilai jarak minimum lebih kecil yaitu 0.7 meter.



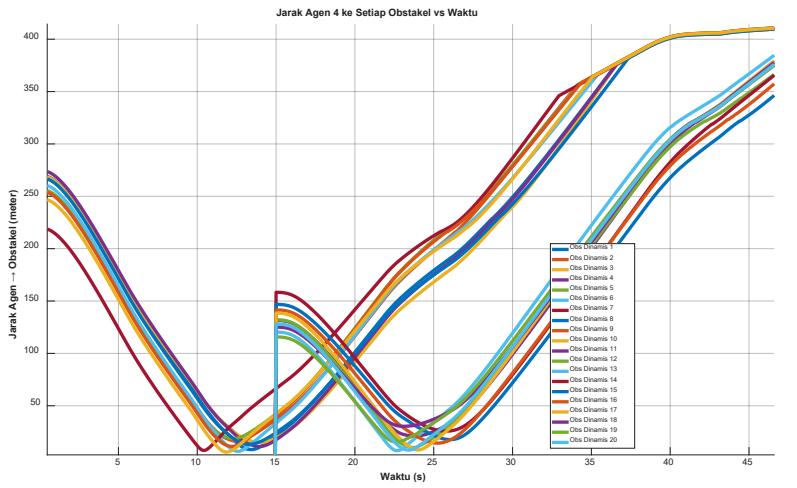
Gambar 4.122 Jarak Agen 1 ke Setiap *Obstacle* DAPF



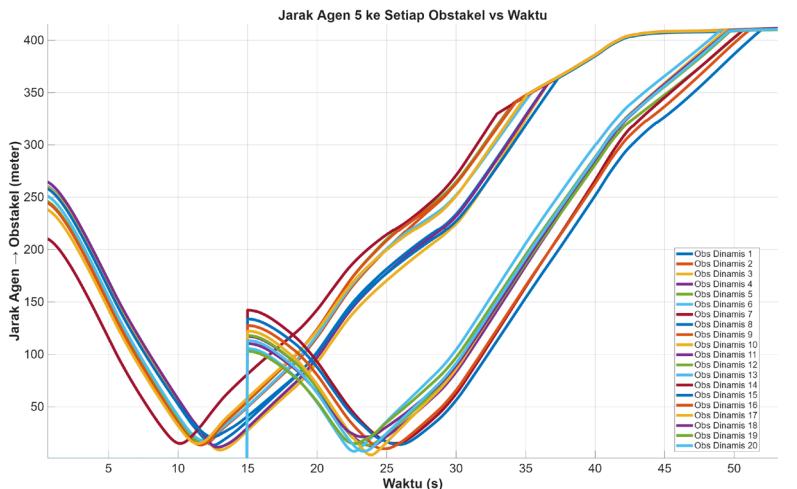
Gambar 4.123 Jarak Agen 2 ke Setiap *Obstacle* DAPF



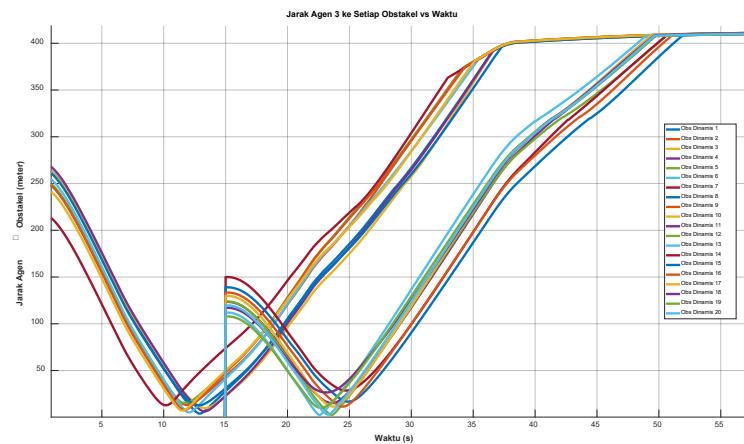
Gambar 4.124 Jarak Agen 3 ke Setiap *Obstacle* DAPF



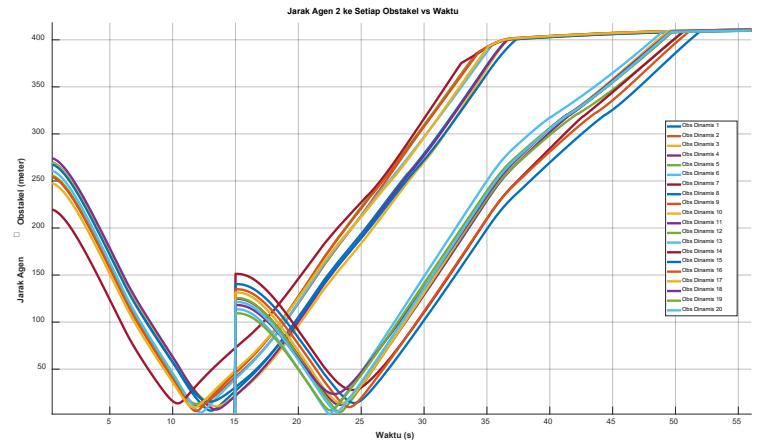
Gambar 4.125 Jarak Agen 4 ke Setiap *Obstacle* DAPF



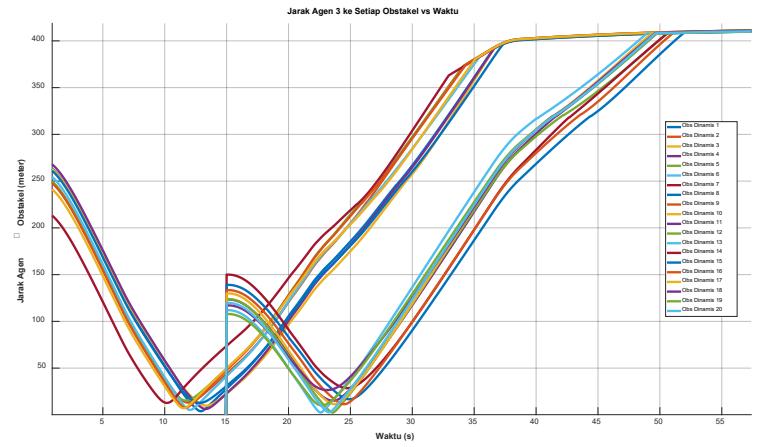
Gambar 4.126 Jarak Agen 5 ke Setiap *Obstacle* DAPF



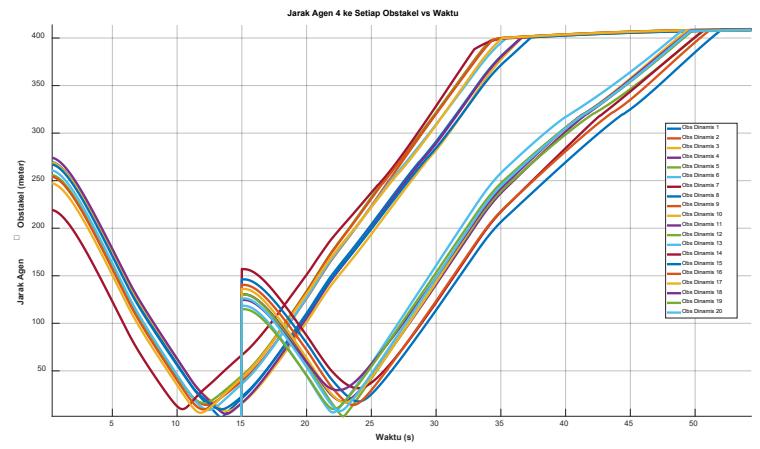
Gambar 4.127 Jarak Agen 1 ke Setiap *Obstacle* MAPF



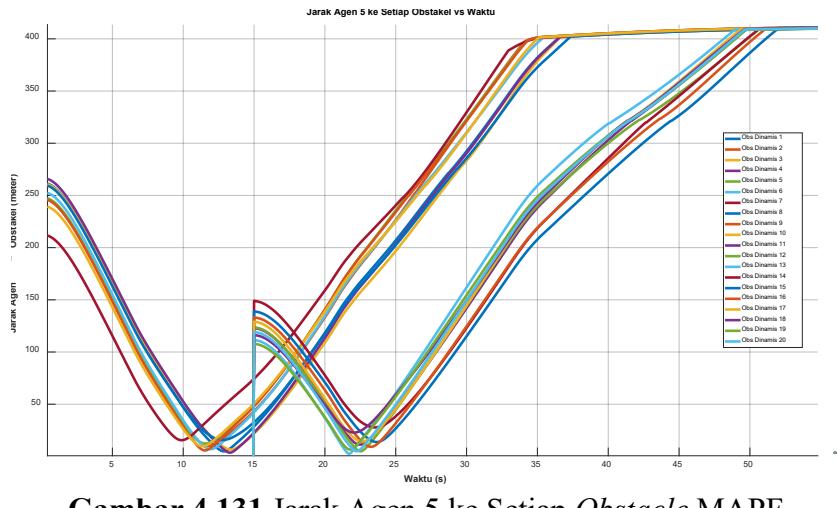
Gambar 4.128 Jarak Agen 2 ke Setiap *Obstacle* MAPF



Gambar 4.129 Jarak Agen 3 ke Setiap *Obstacle* MAPF



Gambar 4.130 Jarak Agen 4 ke Setiap *Obstacle* MAPF



Gambar 4.131 Jarak Agen 5 ke Setiap *Obstacle* MAPF

BAB 5 KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan hasil simulasi pembagian tugas multi UAV menuju multi target dengan penghindaran rintangan menggunakan DAPF dan MAPF diperoleh beberapa kesimpulan berikut.

1. DAPF menjaga jarak antar UAV lebih besar 2.5 kali dibanding MAPF dan mempertahankan jarak ke rintangan lebih tinggi lebih dari 2.5 kali. Pada skenario rintangan statis DAPF menghasilkan lintasan total 10% lebih pendek meskipun waktu tempuh sedikit lebih lama dibanding MAPF
2. MAPF menyelesaikan skenario rintangan dinamis lebih cepat sekitar 26% dan lintasannya lebih pendek sekitar 3.6% dibanding DAPF, tetapi margin keselamatan berkang signifikan.
3. Integrasi DAPF–DAS menghasilkan kerangka kerja multi-UAV yang mampu menggabungkan penghindaran halangan dan distribusi tugas secara simultan, mempertahankan pemisahan antar-UAV dan mencapai target dalam waktu komputasi

5.2 Saran

Saran yang diberikan oleh penulis untuk penelitian selanjutnya terkait dengan topik ini sehingga diharapkan dapat memperbaiki dan mengembangkan penelitian selanjutnya yaitu:

1. Mengembangkan penelitian dengan menambahkan algoritma *global path planning*.
2. Mengimplementasikan *machine learning* untuk proses tuning parameter yang lebih efisien.
3. Penerapan sistem dengan plant yang heterogenMengintegrasikan algoritma DAPF dengan Distributed Assignment Switch pada *fixed-wing* UAV dengan kemampuan terbatas

DAFTAR PUSTAKA

- Agustinah, T., Isdaryani, F., & Nuh, M. (2016). Tracking control of quadrotor using static output feedback with modified command-generator tracker. *International Review of Automatic Control*, 9(4), 242–251. <https://doi.org/10.15866/ireaco.v9i4.9431>
- Asti, I. S., Agustinah, T., & Santoso, A. (2020). Obstacle Avoidance with Energy Efficiency and Distance Deviation Using KNN Algorithm for Quadcopter. *Proceedings - 2020 International Seminar on Intelligent Technology and Its Application: Humanification of Reliable Intelligent Systems, ISITIA 2020*, 285–291. <https://doi.org/10.1109/ISITIA49792.2020.9163788>
- Austin, R. (2010). *Unmanned aircraft systems : UAVs design, development, and deployment*. American Institue of Aeronautics and Astronautics : Wiley. <https://doi.org/10.1002/9780470664797>
- Bresciani, T. (2008). *Modelling, Identification and Control of a Quadrotor Helicopter*. Department of Automatic Control, Lund University. <https://books.google.co.id/books?id=PTjnjkEACAAJ>
- Du, Y., Zhang, X., & Nie, Z. (2019). A Real-Time Collision Avoidance Strategy in Dynamic Airspace Based on Dynamic Artificial Potential Field Algorithm. *IEEE Access*, 7, 169469–169479. <https://doi.org/10.1109/ACCESS.2019.2953946>
- Geng, L., Zhang, Y. F., Wang, J. J., Fuh, J. Y. H., & Teo, S. H. (2013). Mission planning of autonomous UAVs for urban surveillance with evolutionary algorithms. *2013 10th IEEE International Conference on Control and Automation (ICCA)*, 828–833. <https://doi.org/10.1109/ICCA.2013.6564992>
- Hage, G. (2023). *Kontrol Pembagian Tugas Multiagen Menuju Multitarget Dengan Penghindaran Halangan Menggunakan Artificial Potential Field*.
- Hessel, M., Soyer, H., Espeholt, L., Czarnecki, W., Schmitt, S., & Van Hasselt, H. (2019). Multi-Task Deep Reinforcement Learning with PopArt. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01), 3796–3803. <https://doi.org/10.1609/aaai.v33i01.33013796>
- Hu, J., Zhang, H., Liu, L., Zhu, X., Zhao, C., & Pan, Q. (2020). Convergent Multiagent Formation Control With Collision Avoidance. *IEEE Transactions on Robotics*, 36(6), 1805–1818. <https://doi.org/10.1109/TRO.2020.2998766>
- Kurnianto, A., Guyana, D., & Widiarto, M. A. (2024). Fungsi Drone Kamikaze dalam Membantu TNI Angkatan Laut dalam Menghadapi Ancaman di Laut Guna Menjaga Pertahanan Negara Indonesia. *JIIP - Jurnal Ilmiah Ilmu Pendidikan*, 7(10), 11745–11750. <https://doi.org/10.54371/jiip.v7i10.6127>
- Magnussen, Ø., & Skjønhaug, K. (2011). Modeling, design and experimental study for a quadcopter system construction. *University of Agder*, 1–155. http://brage.bibsys.no/hia/handle/URN:NBN:no-bibsys_brage_20668%5Cnbrage.bibsys.no/hia/retrieve/3900/uiareport.pdf
- Nurjanah, S., Agustinah, T., & Fuad, M. (2022). Cooperative Position-based Formation-pursuit of Moving Targets by Multi-UAVs with Collision Avoidance. *JAREE (Journal on Advanced Research in Electrical Engineering)*, 6(2), 82–90. <https://doi.org/10.12962/jaree.v6i2.310>
- Prieto, M., Escarti-Guillem, M. S., & Hoyas, S. (2023). Aerodynamic optimization of a VTOL drone using winglets. *Results in Engineering*, 17, 100855. <https://doi.org/10.1016/j.rineng.2022.100855>

- Shah Alam, M., & Oluoch, J. (2021). A survey of safe landing zone detection techniques for autonomous unmanned aerial vehicles (UAVs). *Expert Systems with Applications*, 179, 115091. [https://doi.org/https://doi.org/10.1016/j.eswa.2021.115091](https://doi.org/10.1016/j.eswa.2021.115091)
- Wahab, F. (2022). Metode Potential Field sebagai Kendali Robot Roda Omni untuk Menuju Target dan Menghindari Rintangan. *ELKOMIKA: Jurnal Teknik Energi Elektrik, Teknik Telekomunikasi, & Teknik Elektronika*, 10, 177. <https://doi.org/10.26760/elkomika.v10i1.177>
- Waharte, S., & Trigoni, N. (2010). Supporting Search and Rescue Operations with UAVs. *2010 International Conference on Emerging Security Technologies*, 142–147. <https://doi.org/10.1109/EST.2010.31>

LAMPIRAN

```
https://github.com/aaiiizr/FinalProject.git
% Muhammad Faiz Ramadhan
% Update 7/3/2025
current_time = datetime('now', 'Format', 'yyyy-MM-dd HH-mm-ss');
log_filename = sprintf('simulation_log_%s.txt', char(current_time));

% Start logging ALL command window output
diary(log_filename);
diary on;

fprintf('🎬 SIMULATION STARTED: %s\n', char(datetime('now')));
fprintf('📝 Command window output being saved to: %s\n', log_filename);
fprintf('=====\\n\\n');

%% Environment code

clf;
close all;
clear all;
clc;

%% simulation parameter
iteration = 8000;           %iteration limit
dt=0.05;                    %time step
Time=dt.*[1:iteration];    %time limit
tpause=0.0001;              %pause duration after each iteration
speed=2;                     %camera orbit speed
tic

%rng generator rng(seed,'type'), change seed to randomly change
building
rng(1460,"v5uniform");
randomize=1 %random or Fixed
static = 1 % Set to 0 for dynamic obstacles
mixed = 0 % Campur Dinamis Statis (Belum Bisa)
APF_type = 2 % Try different algorithms (1-4)
datang = 1 % Coba 1 -3
% datang =1 obstacle dari depan
% datang = 2 obstacles dari belakang
% datang = 3 obstacles dari samping
centered = 1
% posisi obstacle ditengah pastikan sama randomize =1

% Display simulation settings
apf_names = {'Traditional APF', 'Modified APF (MAPF)', 'Velocity APF
(VAPF)', 'Dynamic APF (DAPF)'};
mode_str = '';
if static == 1
    mode_str = 'Static Obstacles';
elseif mixed == 1
    mode_str = 'Mixed (Static + Dynamic) Obstacles';
else
    mode_str = 'Dynamic Obstacles Only';
end

%% System Parameter
% Path Planning gain
Krep = 50; % Gain Repulsive
Katt = 1;
% Gain Attractive
Kform = 0; %Gain Formasi (Belum Dipake)
Kcost = 1; %Cost
```

```

alocfreq = 100; % Cek Switching Setiap 10 Iterasi
targetrad = 5; % Radius Target
desired_speed = 10; % Agent Speed
obstacle_speed = 15; % Obstacle speed
%%

%% Inisiasi Halangan
sped = obstacle_speed;
radii = 10; % Radius dy
% namic obstacle
nObsVec = 1; % Jumlah Halangan
r_dyn = radii;
M_obs = nObsVec;

%% Parameter DAPF
params_dist.rminsafe = 30; % radius aman
params_dist.kw = 40;
params_dist.ka = 10;
params_dist.wmax = 15;
params_dist.beta = pi/6;
params_dist.theta1 = pi/12;
params.Kcost = Kcost;
params.MinDistance= targetrad
params.STABILITY_DISTANCE = 15; % Don't reassign if within this
distance
params.COMMITMENT_DISTANCE = 10; % Fully committed if within this
distance
params.MIN_REASSIGN_BENEFIT = 5; % Only reassign if significantly
beneficial

% gains untuk attraction/repulsion formasi
gains_att.kp = Katt;
gains_att.kv = Katt;
gains_att.kdamp = 0; % Increased damping to prevent oscillation
gains_rep.krep = Krep;
gains_rep.krepv = Krep;
%% Agen Dan Target Inisiasi
%Defining agent & target
N=5; %number of agent
M=1; %target is 3
final_target_assignment = zeros(1, N); % Track final target for each
agent

for t = 1:M
    x_rand = 190 + (195 - 190) * rand;
    y_rand = 190 + (195 - 190) * rand;
    z_rand = 60 + (65 - 60) * rand;
    target{t} = [ x_rand; y_rand; z_rand ];
end
%% Formasi Belum Dipake
% Define formation patterns relative to each target
R = 2; % Formation radius
f = cell(N,1);
Delta = cell(N,1); % Formation offsets for each agent
for i=1:N
    angle = 2*pi*(i-1)/N;
    Delta{i} = [ R*cos(angle);
                 R*sin(angle);
                 0 ]; % Keep formation in same z-plane
    f{i} = Delta{i}; % Store for compatibility
end

d_des = cell(N,N);
for i=1:N
    for j=1:N
        d_des{i,j} = f{j} - f{i};
    end
end

```

```

    end
end
%%
%view angle view(xy plane, z plane) in degrees
xyp=45;
zp=90;
view(xyp,zp)
%% Resolusi Sphere
if static ==0
nRes = 10;
[XS0, YS0, ZS0] = sphere(nRes);
end
%% Buildings Position

% Inisialisasi posisi gedung (Cpos) - 50x3
Cpos = [
80,80,100;
120,130,80
120,100,90
% 1.573276287912332e+02, 1.378946000869402e+02, 66.363534063880167;
% 1.220482473888913e+02, 72.922377362135506, 47.618375837277867;
% 7.361177385694313e+01, 8.115383312930386e+01, 72.406970515365515;
% 8.819028346568220e+01, 1.499734300865982e+02, 79.469622105133908;
% 1.597862906134208e+02, 1.105014199862938e+02, 70.025477598063702;
% 6.159951608429921e+01, 1.569554992912268e+02, 73.069941746938326;
% 6.783081579899851e+01, 7.844358579822623e+01, 38.044413896617669;
% 8.072262672982433e+01, 1.189531604409474e+02, 64.408485099343920;
% 1.463994888052817e+02, 8.777200988196230e+01, 46.493517336543391;
% 1.536616738978522e+02, 6.672340490407777e+01, 23.207872893391407;
% 5.322359666034333e+01, 9.765154800942299e+01, 20.254444954636842;
% 1.410457777462161e+02, 1.538210264511630e+02, 76.625309465556043;
% 1.132997890371578e+02, 1.033481569585932e+02, 63.125625972731122;
% 5.065373181088210e+01, 1.624948951289558e+02, 66.784530402276204;
% 1.657596417009627e+02, 9.294229330596747e+01, 54.230649438973259;
% 1.222348255003406e+02, 6.734440811858958e+01, 73.614953154623720;
% 4.992481000345745e+01, 1.434973022997406e+02, 53.248253154374183;
% 6.045796165598497e+01, 9.317799551847328e+01, 32.971330335181079;
% 1.642121446856819e+02, 9.876946451729935e+01, 73.561523537512116;
% 5.228367048679788e+01, 8.016717768565396e+01, 58.863044189240448;
% 1.681187555292010e+02, 1.659427187567823e+02, 52.986570506108421;
% 4.110188322046096e+01, 1.231035279979451e+02, 46.837424237331291;
% 1.019707998110143e+02, 9.537307042764756e+01, 72.902490531615626;
% 1.323444324299894e+02, 1.446113218422077e+02, 75.386084605828117;
% 1.161039185093896e+02, 1.289591450145620e+02, 88.605646621606127;
% 5.453230591769799e+01, 1.283129694141639e+02, 22.934205907411897;
% 5.629942778193472e+01, 1.567364676475588e+02, 71.131648795729205;
% 1.356404034240190e+02, 1.409244384201849e+02, 64.919460775576255;
% 1.353980200550252e+02, 8.733888968818356e+01, 76.113958180576574;
% 1.562358399405659e+02, 4.957577768955932e+01, 76.447218816318340;
% 6.908643745102425e+01, 1.137448124471752e+02, 66.794166366824356;
% 1.099193456072939e+02, 1.656662514721841e+02, 78.129872863194009;
% 9.784866777427357e+01, 4.854531402547812e+01, 57.713186644019700;
% 1.467210078948024e+02, 1.550299303103431e+02, 20.143246090024437;
% 5.036310148912456e+01, 7.363493417739761e+01, 89.485543641140424;
% 6.809939147950307e+01, 5.058684262757231e+01, 39.224598495382153;
% 8.461566199495947e+01, 1.666784879895814e+02, 50.665848219792480;
% 1.672034524426207e+02, 6.539244389263217e+01, 22.213398876297642;
% 1.342640898899008e+02, 8.154995140003953e+01, 83.969226367125316;
% 4.497989431257338e+01, 9.483208816317341e+01, 29.595425987966216;
% 9.112245024343316e+01, 1.030947457885754e+02, 32.083086622248246;
% 1.694744987549772e+02, 1.510121538816217e+02, 28.085054979298548;
% 1.676175598775339e+02, 8.518659156217305e+01, 52.784477685246443;
% 7.827058219085836e+01, 4.164762248412715e+01, 87.022702449975228;
% 6.122399537328910e+01, 1.312854511385135e+02, 49.468523034523074;
% 5.933472569401872e+01, 8.622013334436897e+01, 44.832387854396501;
% 1.214000937591466e+02, 1.596316321668983e+02, 37.162873499323034;

```

```

% 1.697417848176984e+02, 5.453504852254214e+01, 24.656151897055132;
% 9.796334027015541e+01, 1.144536725674922e+02, 20.783375339746961;
% 1.199890572642272e+02, 1.635534136404759e+02, 40.344071960353233
];

% -----
% Inisialisasi radius (50x1)
radius = [
    7;
    6;
    9
    % 6.185872968728452;
    % 8.964777724273965;
    % 5.485747135843992;
    % 9.037518297143492;
    % 6.008904071349907;
    % 9.235594138539918;
    % 8.538654486202947;
    % 6.037835329366217;
    % 8.126538563083551;
    % 8.247119790472057;
    % 4.472953753848000;
    % 10.570936834379474;
    % 9.179010286357329;
    % 7.456135694904914;
    % 5.249681114365560;
    % 8.499316985105903;
    % 6.948839887779886;
    % 7.796712152048944;
    % 6.894786768783632;
    % 5.605224824704907;
    % 7.543407775395499;
    % 9.372789648181408;
    % 10.320786578246002;
    % 8.812422183856746;
    % 7.535837756950957;
    % 5.405198509683015;
    % 8.116378111187737;
    % 6.247466717337204;
    % 9.212976639978034;
    % 5.907951127615620;
    % 7.436736997897347;
    % 8.386368725566832;
    % 6.101062151079636;
    % 5.706359280856537;
    % 7.273562645133646;
    % 5.615773461848578;
    % 7.334084224570097;
    % 6.657780059093303;
    % 9.858501863730547;
    % 9.057918755858047;
    % 6.593862973526205;
    % 5.310714982307363;
    % 8.655301180046145;
    % 7.677012065903208;
    % 8.541646704193894;
    % 6.270589509444984;
    % 7.377924528942020;
    % 8.190405633985000;
    % 8.113725408404168;
    % 5.312221181635445
];

```

% Jika Anda juga butuh radius1 (radius "asli" sebelum diskalakan 1.1):

```

%% radnomize Obstacles

if randomize == 1 % Static
nb = nObsVec; % number of obstacles

% Initialize arrays to store obstacle properties
cposx = zeros(1, nb);
cposy = zeros(1, nb);
cposz = zeros(1, nb);
radius = zeros(nb, 1);

% Define bounds based on centered flag
if centered == 0
    x_min = 40; x_max = 170;
    y_min = 40; y_max = 170;
    z_min = 40; z_max = 90;
else
    x_min = 70; x_max = 71; % Note: this gives no variation in original
code
    y_min = 70; y_max = 71;
    z_min = 70; z_max = 90;
end

% Minimum distance between obstacle centers (can be adjusted)
min_separation = 2; % Adjust this value based on your needs
max_attempts = 1000; % Maximum attempts to place each obstacle

% Place obstacles one by one
for i = 1:nb
    placed = false;
    attempts = 0;

    while ~placed && attempts < max_attempts
        attempts = attempts + 1;

        % Generate random position
        temp_x = x_min + (x_max - x_min) * rand();
        temp_y = y_min + (y_max - y_min) * rand();
        temp_z = z_min + (z_max - z_min) * rand();

        % Generate random radius
        temp_radius = radii + (5 - 3) * rand();

        % Check if this position overlaps with existing obstacles
        valid_position = true;

        for j = 1:i-1
            % Calculate distance between centers
            distance = sqrt((temp_x - cposx(j))^2 + (temp_y - cposy(j))^2);

            % Check if obstacles would overlap (distance < sum of radii
            + minimum separation)
            required_distance = radius(j) + temp_radius +
            min_separation;

            if distance < required_distance
                valid_position = false;
                break;
            end
        end

        % If position is valid, place the obstacle
        if valid_position
            cposx(i) = temp_x;
            cposy(i) = temp_y;
        end
    end
end

```

```

        cposz(i) = temp_z;
        radius(i) = temp_radius;

        placed = true;
    end
end

% If we couldn't place the obstacle after max attempts
if ~placed
    warning('Could not place obstacle %d without overlap after %d
attempts', i, max_attempts);
    % Place it anyway with last attempted position (or handle as
needed)
    cposx(i) = temp_x;
    cposy(i) = temp_y;
    cposz(i) = temp_z;
    radius(i) = temp_radius;

end
end

% Create position matrix
Cpos = [cposx' cposy' cposz'];

% Optional: Display placement statistics
fprintf('Successfully placed %d obstacles\n', nb);

% Optional: Verify no overlaps (for debugging)
overlap_count = 0;
for i = 1:nb
    for j = i+1:nb
        distance = sqrt((cposx(i) - cposx(j))^2 + (cposy(i) -
cposy(j))^2);
        if distance < (radius(i) + radius(j))
            overlap_count = overlap_count + 1;
        end
    end
end

if overlap_count > 0
    warning('Found %d overlapping obstacle pairs', overlap_count);
else
    fprintf('No overlaps detected\n');
end
end

radius1 = radius / 1.1;
SEQUENTIAL_SPAWN_INTERVAL = 300; % Spawn every 150 iterations
MIN_GOAL_DISTANCE_THRESHOLD = 100; % Don't spawn if agent within 10 units
of goal
next_spawn_iteration = SEQUENTIAL_SPAWN_INTERVAL; % First spawn at
iteration 150
sequential_spawn_count = 0; % Track how many spawn events occurred
obstacles_per_spawn = nObsVec; % Number of obstacles to spawn each time
(same as initial)
original_M_obs = M_obs;

if static == 1
    % Static mode only
    obstacles = Cpos';
    M_obs = size(obstacles,2);
    radius = radius; % Use static radius
    obs_speeds = zeros(3, size(Cpos,1)); % Zero velocity
    obstacle_types = ones(1, M_obs); % 1 = static
    static_obs_indices = 1:M_obs;
    dynamic_obs_indices = [];

```

```

else
    % Dynamic mode or mixed mode
    if mixed == 1
        %% Belum Jadi
        % Mixed mode: both static and dynamic obstacles

        % Static obstacles
        static_obstacles = Cpos';
        static_radius = radius;
        n_static = size(static_obstacles, 2);

        % Dynamic obstacles
        M_obs_dynamic = nObsVec;

        % Initialize dynamic obstacle states
        obs_states = zeros(12, M_obs_dynamic);
        obs_goals = zeros(3, M_obs_dynamic);

        for k = 1:M_obs_dynamic
            % Start near a random target
            t = randi(M);
            start_pos = [180+randn*10; 180+randn*10; 40+randn*10];

            % Set goal to another position
            end_pos = [40; 40; 40];
            % Initialize state
            obs_states(1:3, k) = start_pos;
            obs_states(4:12, k) = zeros(9,1);
            obs_goals(:, k) = end_pos;
        end

        % Combine all obstacles
        obstacles = [static_obstacles, obs_states(1:3, :)];
        radius = [static_radius; r_dyn * ones(M_obs_dynamic, 1)];
        M_obs = n_static + M_obs_dynamic;
        obs_speeds = zeros(3, M_obs);

        % Track obstacle types: 1=static, 2=dynamic
        obstacle_types = [ones(1, n_static), 2*ones(1, M_obs_dynamic)];
        static_obs_indices = 1:n_static;
        dynamic_obs_indices = (n_static+1):M_obs;
    else
        %% Pure dynamic mode
        M_obs = nObsVec;

        % Initialize dynamic obstacle states
        obs_states = zeros(12, M_obs);
        obs_goals = zeros(3, M_obs);

        for k = 1:M_obs
            % Start near a random target
            t = randi(M);
            % start_pos = target{t} + [0;80;0] - [randn*20; randn*20;
            randn*5];
            %% Penentuan Arah Datang Obstacle Dinamis
            if datang == 1 % Dari Depan
                start_pos = [180+randn*10; 180+randn*10; 65+randn*3];

                % Set goal to another position
                end_pos = [-40+randn*1; -40+randn*1; 60+randn*10];
            elseif datang == 2
                end_pos = [580+randn*10; 580+randn*10; 61+randn*2];

                % Set goal to another position
                start_pos = [-30+randn*1; -30+randn*1; 61+randn*2];
            end
        end
    end
end

```

```

        else
            end_pos = [150+randn*20; 10+randn*20; 64+randn*7];
            % Set goal to another position
            start_pos = [-10+randn*20; 190+randn*20; 64+randn*4];
        end

        % Initialize state
        obs_states(1:3, k) = start_pos;
        obs_states(4:12, k) = zeros(9,1);
        obs_goals(:, k) = end_pos;
    end

    % Only dynamic obstacles
    obstacles = obs_states(1:3, :);
    radius = r_dyn * ones(M_obs, 1);
    obs_speeds = zeros(3, M_obs);
    obstacle_types = 2*ones(1, M_obs); % All dynamic
    static_obs_indices = [];
    dynamic_obs_indices = 1:M_obs;
end
end

%% Random initial UAV pos outside any obstacle
start = cell(N,1);
for i = 1:N
    valid = false;
    while ~valid
        xr = 15*rand(1);
        yr = 15*rand(1);
        zr = 0*rand(1);
        valid = true;
        % cek setiap obstacle (dinamis atau statis) via 'obstacles' &
        'radius'
        for k = 1:size(obstacles,2)
            dx = xr - obstacles(1,k);
            dy = yr - obstacles(2,k);
            dz = zr - obstacles(3,k);
            if static == 1
                % jika (x,y) masuk lingkaran obstacle dan z di bawah tinggi
                obstacle
                if dx^2 + dy^2 < radius(k)^2 && zr < obstacles(3,k)
                    valid = false;
                    break;
                end
                else
                    if dx^2 + dy^2 +dz^2 < radius(k)^2
                        valid = false;
                        break;
                    end
                end
            end
        end
        start{i} = [xr; yr; zr];
    end
%% Formasi Nyoba
% % Virtual leader state (add after target definition)
% xr = cell(M,1);
% vr = cell(M,1);
% xr_init = cell(M,1);
% xr_history = cell(M,1); % To store trajectory
%
% %% Communication topology matrices (add after line ~58)
% % These define which agents can communicate
% A_comm = zeros(N,N); % Adjacency matrix for agent communication

```

```

% L_comm = zeros(N,N); % Laplacian matrix
% B_comm = zeros(N,N); % Leader communication matrix
%
% % Example topology (modify based on your needs)
% % This creates a connected graph
% for i = 1:N
%     for j = 1:N
%         if i ~= j && norm(start{i} - start{j}) < 100 % Neighbors
within 100 units
%             A_comm(i,j) = 1;
%         end
%     end
% end
%
% % Compute Laplacian
% for i = 1:N
%     L_comm(i,i) = sum(A_comm(i,:));
%     for j = 1:N
%         if i ~= j
%             L_comm(i,j) = -A_comm(i,j);
%         end
%     end
% end
%
% % Leader communication (at least one agent can see the leader)
% B_comm = diag(rand(N,1) > 0.7); % Random agents can see leader
% if sum(diag(B_comm)) == 0
%     B_comm(1,1) = 1; % Ensure at least agent 1 sees leader
% end

%% setting up simulation
fill3([0 200 200 0],[0 0 200 200],[0 0 0 0],[0.3 0.3 0.3]);
hold on
grid on
x = 0:4:200;
y = 0:4:200;
xlabel("x");
ylabel("y");
zlabel("z");
xlim([0 200]);
ylim([0 200]);
zlim([0 100]);
fhandle=figure(1);
hObs = gobjects(0,1);

% Initialize static obstacles
if static == 1
    hObs = gobjects(M_obs,1);
    for i = 1:M_obs
        % create_cylinder(radius1(i), Cpos(i,:), [0.45, 0.58, 0.76]);
        [h1,~,~] = create_cylinder(radius(i), Cpos(i,:), [0.45, 0.58,
0.76]);
        hObs(i) = h1;
    end
else
    % Dynamic or mixed mode
    if mixed == 1
        % Draw static obstacles as cylinders
        hObs = gobjects(length(static_obs_indices),1);
        for idx = 1:length(static_obs_indices)
            i = static_obs_indices(idx);
            % create_cylinder(radius1(idx), Cpos(idx,:), [0 0 0]);
            [h1,~,~] = create_cylinder(radius(idx), Cpos(idx,:), [0.45,
0.58, 0.76]);
            hObs(idx) = h1;
        end
    end
end

```

```

else
    hObs = [];
end

% Initialize dynamic obstacle graphics as quadcopters
if ~isempty(dynamic_obs_indices)
    hObsQuad = cell(length(dynamic_obs_indices), 1);
    hObsCircle = gobjects(length(dynamic_obs_indices), 1);
    hObsCirclex = gobjects(length(dynamic_obs_indices), 1);

    for idx = 1:length(dynamic_obs_indices)
        k = dynamic_obs_indices(idx);
        if mixed == 1
            % In mixed mode, use local index for obs_states
            obs_idx = idx;
        else
            % In pure dynamic mode, use k directly
            obs_idx = k;
        end

        pos_k = obs_states(1:3, obs_idx);
        % Create quadcopter graphic
        hObsQuad{idx} = create_quadcopter_graphics(pos_k,
obs_states(7:9, obs_idx), [1 0 0], 2);
        radiussafe = 0.25

        % Create safety circle around obstacle
        theta_circle = linspace(0, 2*pi, 50);
        circle_x = pos_k(1) + r_dyn * cos(theta_circle);
        circle_y = pos_k(2) + r_dyn * sin(theta_circle);
        circle_z = pos_k(3) * ones(size(theta_circle));
        hObsCircle(idx) = plot3(circle_x, circle_y, circle_z, 'r--',
', 'LineWidth', 1.5);

        circle_xx = pos_k(1) + radiussafe * cos(theta_circle);
        circle_yy = pos_k(2) + radiussafe * sin(theta_circle);
        circle_zz = pos_k(3) * ones(size(theta_circle));
        hObsCirclex(idx) = plot3(circle_xx, circle_yy, circle_zz,
'b', 'LineWidth', 3);
    end
else
    hObsQuad = [];
    hObsCircle = [];
end

Nfeas=100000;
feas_point=zeros(Nfeas,3);
for j = 1:Nfeas
    flag = 0;
    while ~flag
        flag = 1;
        feas_point(j,:) = [200*rand 200*rand 100*rand];
        % Loop melalui setiap obstacle (dinamis atau statis, sesuai
toggle)
        for k = 1 : size(obstacles,2)
            dx = feas_point(j,1) - obstacles(1,k);
            dy = feas_point(j,2) - obstacles(2,k);
            horiz_dist2 = dx^2 + dy^2;
            % obstacles(3,k) = tinggi obstacle; radius(k) = jari-jari
            obstacle
            if horiz_dist2 < radius(k)^2 && feas_point(j,3) <
            obstacles(3,k)
                flag = 0;
                break;
            end
        end
    end
end

```

```

        end
    end
end

%Defining agent & target
colors = lines(M); % M warna berbeda
cmap = lines(max(M,N)); % Define cmap early to avoid errors with
dynamic obstacles
final_stopping_pos = cell(N, 1);

for t = 1:M

    plot3(target{t}(1), target{t}(2), target{t}(3), 'o', ...
        'MarkerSize', 15, 'MarkerFaceColor', colors(t,:), ...
        'MarkerEdgeColor', 'black', 'LineWidth', 2);

    % Target label
    text(target{t}(1)+5, target{t}(2)+5, target{t}(3)+5,
        sprintf('Target %d', t), 'FontSize', 12, 'FontWeight', 'bold', ...
        'Color', colors(t,:), 'BackgroundColor', 'white', ...
        'EdgeColor', 'black');
end
legend(arrayfun(@(t) sprintf('Target %d',t), 1:M,
'UniformOutput',false));

fprintf('\n==== Simulation Settings ====\n');
fprintf('APF Algorithm: %s\n', apf_names{APF_type});
fprintf('Obstacle Mode: %s\n', mode_str);
fprintf('Number of Agents: %d\n', N);
fprintf('Number of Targets: %d\n', M);
if static == 0
    fprintf('Dynamic Obstacles: %d\n', length(dynamic_obs_indices));
    if mixed == 1
        fprintf('Static Obstacles: %d\n', length(static_obs_indices));
    end
end
fprintf('=====\\n\\n');

titlehandle=title(sprintf("Simulation Loading - %s with %s",
apf_names{APF_type}, mode_str));
disp('Simulation Loading');
subtitlehandle=subtitle("Iteration 0");
%
h=plot3(target{1}(1),target{1}(2),target{1}(3),"Marker","*", "Color", "black"
,"LineStyle", ":");

% Defining intial position of the UAV.
state0=zeros(9,1);

for j=1:N
    state{j}=[start{j}; state0];
    all_state{j}=[start{j}; state0];
    current_pos{j}=start{j};
    previous_pos{j}=current_pos{j};
    data_points{j}=zeros(iteration,3);
    all_Ft{j}=zeros(3,1);
    Ftp{j} = zeros(3,1);
end

figure(1)
hold on
grid on
axis([0 200 0 200 0 100]); % set limit x,y,z
axis manual; % kunci agar tidak autoscale

```

```

%Path Planning
F = zeros(3,length(obstacles));

%% --- Task Allocation: nearest-first with equal quotas + theta ---

% 0) Inisiasi theta untuk deadlock-free swap (makalah Hu et al.)
theta = (1:N) + rand(1,N)*0.1;
fprintf('Initial theta: '); fprintf('%.2f ', theta); fprintf('\n');

% 1) Hitung jarak tiap agen ke tiap target & tampilkan
D = zeros(N, M);
for j = 1:N
    for t = 1:M
        D(j,t) = norm(current_pos{j} - target{t});
    end
    fprintf('Agent %d distances:', j);
    for t = 1:M
        fprintf(' T%d=% .2f', t, D(j,t));
    end
    fprintf('\n');
end

% 1) hitung rata2 jarak ke tiap target
avgD = mean(D,1); % D adalah matriks NxM jarak agen→target

% 2) urutkan target berdasarkan jarak naik
[~, idx] = sort(avgD, 'ascend');

% 3) buat kuota
base = floor(N/M);
rem = mod(N, M);
quota = base * ones(1, M);

% 4) berikan +1 ke rem target paling kecil avg jaraknya
quota(idx(1:rem)) = quota(idx(1:rem)) + 1;

fprintf('Quotas per target: ');
disp(quota);

% 3) Inisiasi assigned_goal & counter
assigned_goal = zeros(1, N);
counts = zeros(1, M);

% 4) Buat daftar pasangan (agent, target) & urutkan by jarak naik
[pj, pt] = ndgrid(1:N, 1:M);
pairs = [pj(:), pt(:)];
[~, order] = sort(D(:));
sorted_pairs = pairs(order, :);

% 5) Greedy nearest-first assign sesuai kuota
for k = 1:size(sorted_pairs,1)
    j = sorted_pairs(k,1);
    t = sorted_pairs(k,2);
    if assigned_goal(j)==0 && counts(t) < quota(t)
        assigned_goal(j) = t;
        counts(t) = counts(t) + 1;
    end
end
fprintf('Initial assignment: ');
disp(assigned_goal);

% Initialize agent quadcopter graphics
hAgentQuad = cell(N,1); % Use cell array instead of gobjects
for i=1:N
    if assigned_goal(i) > 0 && assigned_goal(i) <= size(cmap,1)
        col = cmap(assigned_goal(i),:);
    else

```

```

        col = [0 0 1]; % Default blue for agents
    end
    hAgentQuad{i} = create_quadcopter_graphics(start{i}, zeros(3,1),
col, 1.5); % Scale 1.5 for agents
    htext{i}=text(start{i}(1)-1,start{i}(2)-1,start{i}(3)-
1,(sprintf('%d', i))); %Callout name
end

% Initial quota distribution
initcounts = histcounts(assigned_goal,1:M+1);

% Redistribution for under-quota targets
under = find(initcounts < quota);
over = find(initcounts > quota);

for t = under
    need = quota(t) - initcounts(t);
    while need>0 && ~isempty(over)
        s = over(1);
        donors = find(assigned_goal==s);
        [~, idxs] = sort( arrayfun(@(j) norm(current_pos{j}-target{t}),
donors ), 'descend');
        move = donors(idxs(1));
        assigned_goal(move) = t;
        need = need - 1;
        initcounts = histcounts(assigned_goal,1:M+1);
        if initcounts(s) <= quota(s)
            over(1) = [];
        end
    end
end

% message queues untuk handshake swap
for j=1:N
    requests.in{j} = [];
    requests.ack{j} = [];
    requests.nack{j} = [];
    requests.sent(j).to = [];
end

SAVE_3D_REPLAY = true; % Set to true to enable 3D replay data saving

if SAVE_3D_REPLAY
    fprintf('Initializing 3D replay recording...\n');

    % Initialize replay data structure
    replay_data = struct();
    replay_data.settings = struct();
    replay_data.settings.APF_type = APF_type;
    replay_data.settings.apf_name = apf_names{APF_type};
    replay_data.settings.mode_str = mode_str;
    replay_data.settings.static = static;
    replay_data.settings.mixed = mixed;
    replay_data.settings.datang = datang;
    replay_data.settings.N = N;
    replay_data.settings.M = M;
    replay_data.settings.M_obs = M_obs;
    replay_data.settings.iteration = iteration;
    replay_data.settings.dt = dt;
    replay_data.settings.xyp = xyp;
    replay_data.settings.zp = zp;

    % Save static elements
    replay_data.static_elements = struct();
    replay_data.static_elements.targets = target;
    replay_data.static_elements.start_positions = start;

```

```

replay_data.static_elements.colors = cmap;
replay_data.static_elements.target_colors = colors;
replay_data.static_elements.params = params;

% Save obstacles
replay_data.static_elements.obstacles = struct();
replay_data.static_elements.obstacles.positions = obstacles;
replay_data.static_elements.obstacles.radii = radius;
if static == 1
    replay_data.static_elements.obstacles.Cpos = Cpos;
    replay_data.static_elements.obstacles.type = 'static';
else
    replay_data.static_elements.obstacles.initial_states = obs_states;
    replay_data.static_elements.obstacles.goals = obs_goals;
    replay_data.static_elements.obstacles.type = 'dynamic';
    if mixed == 1
        replay_data.static_elements.obstacles.static_indices =
static_obs_indices;
        replay_data.static_elements.obstacles.dynamic_indices =
dynamic_obs_indices;
    end
end

% Pre-allocate arrays for dynamic data
max_iter = iteration;
replay_data.dynamic = struct();
replay_data.dynamic.agent_positions = zeros(N, 3, max_iter);
replay_data.dynamic.agent_orientations = zeros(N, 3, max_iter);
replay_data.dynamic.agent_assignments = zeros(N, max_iter);
replay_data.dynamic.agent_status = false(N, max_iter); % done/not done
replay_data.dynamic.agent_crashed = false(N, max_iter);
replay_data.dynamic.target_reached_time = zeros(N, 1);

if static == 0
% Calculate maximum possible obstacles including sequential spawns
max_spawns = ceil(iteration / SEQUENTIAL_SPAWN_INTERVAL);
max_possible_obstacles = M_obs + (max_spawns * obstacles_per_spawn);

    replay_data.dynamic.obstacle_positions = zeros(max_possible_obstacles,
3, max_iter);
    replay_data.dynamic.obstacle_orientations =
zeros(max_possible_obstacles, 3, max_iter);
    replay_data.dynamic.obstacle_states = zeros(12, max_possible_obstacles,
max_iter);

    % Track actual number of obstacles per frame
    replay_data.dynamic.num_obstacles_per_frame = zeros(max_iter, 1);
end

% Save trajectory data structure
replay_data.dynamic.trajectories = cell(N, 1);
for j = 1:N
    replay_data.dynamic.trajectories{j} = [];
end

% Save initial assignment
replay_data.dynamic.initial_assignment = assigned_goal;

fprintf('3D Replay recording initialized\n');
replay_frame_count = 0;
end

%% parameters DAS
params.R_STOP      = 1;
params.R_SWITCH    = 15;
params.delta       = 5;      % minimal distance δ

```

```

params.dtheta    = 0.1;
params.comm_range= 50;
params.timeout   = 2; % atau detik maksimal, misal 5
params.quota    = quota; % vektor 1xM
done = false(1,N);
params.assigngoal = []; % matriks kosong: tiap baris satu snapshot
assigned_goal
target_reached_time = zeros(N, 1); % Time when each agent reached
target
crashed_agents = false(1, N); % Track crashed agents
target_filled = false(1, M); % Track which targets are
completely filled
crash_recovery_points = cell(N, 1); % Store recovery points for crashed
agents

% neighbour list (update statis pertama kali)
for j=1:N
    neighbors{j} = find_neighbors(j, current_pos, params.comm_range);
end

flag_alo=1;
target_dilacak = 1:M
idx_dkt=zeros(M,1);
idx_hnt=zeros(M,1);
goalI=start;

titlehandle.String=sprintf("Simulation Running - %s",
apf_names{APF_type});
disp('Running');
disp('Click the figure and press any key to stop iteration')

%% Initialize logging variables
done_all = false;
num_steps = ceil(iteration/allocfreq);
assigned_history = zeros(N, num_steps);
theta_history    = zeros(N, num_steps);
max_possible_obstacles = M_obs + (iteration / 100) * nObsVec + 50; %
Add buffer
distance_obs_all = zeros(N, max_possible_obstacles, iteration);

time_steps      = (1:num_steps) * allocfreq * dt;
params.assigned_goal = assigned_goal;

% Initialize history
distance_obs = zeros(N, iteration);
crash = false(N, iteration);
posA_log      = zeros(N, 3, iteration);
pos0_log      = zeros(max_possible_obstacles, 3, iteration);
% All obstacles (static + dynamic)
Fatt_log      = zeros(N, 3, iteration);
Frep_log      = zeros(N, 3, iteration);
iter_log      = (1:iteration)';

% Initialize speed tracking
agent_speeds = zeros(N, iteration);
if static == 0
obstacle_speeds = zeros(max_possible_obstacles, iteration);
else
obstacle_speeds = [];
end

if static==1
    for i=1:iteration
        % Fix: Only assign to the rows corresponding to actual
obstacles
        pos0_log(1:M_obs,:,:,i) = obstacles';
    end
end

```

```

        % Fill remaining rows with NaN if needed
        if M_obs < size(pos0_log,1)
            pos0_log(M_obs+1:end,:,:,:) = NaN;
        end
    end
end

% Initialize handles for visualization
% Initialize trajectory handles for both static and dynamic modes
hTraj = cell(N,1);
for jj = 1:N
    hTraj{jj} = []; % Will store multiple line handles for color
segments
end

% Track assignment changes
prev_assigned_goal = assigned_goal;
segment_start_idx = ones(N,1); % Starting index of current color
segment

% Initialize distance matrix
distance_agents = zeros(N, N, iteration);

detected_agents = cell(N, iteration); % Which other agents each
agent detects

%% MAIN ITERATION
for i=1:iteration
    % Update dynamic obstacles FIRST
    if static == 0
        for k = 1:M_obs
            % Get current obstacle state
            obs_pos = obs_states(1:3, k);
            obs_goal = obs_goals(:, k);

            % Simple proportional navigation toward goal
            direction = obs_goal - obs_pos;
            dist_to_goal = norm(direction);

            if dist_to_goal > 5 % If not at goal
                % Desired acceleration toward goal (using configurable
speed)
                desired_velocity = obstacle_speed * direction /
current_velocity = obs_states(4:6, k);
                desired_acc = (desired_velocity - current_velocity);

                % Update obstacle state using quadcopter dynamics
                obs_states(:, k) = obs_state_update(obs_states(:, k),
desired_acc, dt);

            else
                % Reached goal, set new goal
                obs_goals(1:3, k) = [-100;-100;50];
            end

            % Track obstacle speed
            obstacle_speeds(k, i) = norm(obs_states(4:6, k));
        end

        % UPDATE
        obstacles = obs_states(1:3, :);
        obs_speeds = obs_states(4:6, :);

        % Update visualization ( if dynamic obstacles)
    end

```

```

        if exist('hObsQuad', 'var') && iscell(hObsQuad) &&
~isempty(hObsQuad)
    for k = 1:min(M_obs, length(hObsQuad))
        if k <= size(obs_states, 2) && ~isempty(hObsQuad{k}) &&
isValid(hObsQuad{k})
            pos_k = obs_states(1:3, k);
            angles_k = obs_states(7:9, k);

            % Update quadcopter graphic
            update_quadcopter_graphics(hObsQuad{k}, pos_k, angles_k);

            % Update safety circles if they exist
            if exist('hObsCircle', 'var') && length(hObsCircle) >= k &&
isValid(hObsCircle(k))
                theta_circle = linspace(0, 2*pi, 50);
                circle_x = pos_k(1) + r_dyn * cos(theta_circle);
                circle_y = pos_k(2) + r_dyn * sin(theta_circle);
                circle_z = pos_k(3) * ones(size(theta_circle));
                set(hObsCircle(k), 'XData', circle_x, 'YData', circle_y,
'ZData', circle_z);
            end

            if exist('hObsCirclex', 'var') && length(hObsCirclex) >= k &&
isValid(hObsCirclex(k))
                radiussafe = 0.25;
                circle_xx = pos_k(1) + radiussafe * cos(theta_circle);
                circle_yy = pos_k(2) + radiussafe * sin(theta_circle);
                circle_zz = pos_k(3) * ones(size(theta_circle));
                set(hObsCirclex(k), 'XData', circle_xx, 'YData', circle_yy,
'ZData', circle_zz);
            end
        end
    end

    % Log obstacle positions
    if size(obstacles, 2) > 0
        pos0_log(1:size(obstacles,2), :, i) = obstacles';
        % Fill remaining slots with NaN
        if size(obstacles,2) < size(pos0_log,1)
            pos0_log(size(obstacles,2)+1:end, :, i) = NaN;
        end
    end
end
elseif static == 1

    end
    %% Sequential Obstacle Batch Spawning Logic
if static == 0 && i == next_spawn_iteration
    % Check if any agent is close to goal
    agent_near_goal = check_agent_near_goal(current_pos, assigned_goal,
target, MIN_GOAL_DISTANCE_THRESHOLD);

    if ~agent_near_goal
        % Safe to spawn new batch of obstacles
        [obs_states, obs_goals, obstacles, radius, M_obs, obstacle_types] =
...
        spawn_obstacle_batch(obs_states, obs_goals, obstacles, radius,
M_obs, obstacle_types, datang, obstacles_per_spawn, r_dyn);

        sequential_spawn_count = sequential_spawn_count + 1;

        % Create graphics for all new obstacles in the batch
        if exist('hObsQuad', 'var') && iscell(hObsQuad)
            % Calculate range of new obstacles
            start_idx = M_obs - obstacles_per_spawn + 1;
            end_idx = M_obs;

```

```

for new_k = start_idx:end_idx
    if new_k <= size(obs_states, 2)
        new_pos = obs_states(1:3, new_k);
        new_angles = obs_states(7:9, new_k);

        % Expand graphics arrays if needed
        if length(hObsQuad) < new_k
            hObsQuad{new_k} = [];
        end
        hObsQuad{new_k} = create_quadcopter_graphics(new_pos,
new_angles, [1 0 0], 2);

        % Create safety circles
        if exist('hObsCircle', 'var')
            if length(hObsCircle) < new_k
                hObsCircle(new_k) = gobjects(1);
            end

            theta_circle = linspace(0, 2*pi, 50);
            circle_x = new_pos(1) + r_dyn * cos(theta_circle);
            circle_y = new_pos(2) + r_dyn * sin(theta_circle);
            circle_z = new_pos(3) * ones(size(theta_circle));
            hObsCircle(new_k) = plot3(circle_x, circle_y,
circle_z, 'r--', 'LineWidth', 1.5);
        end

        if exist('hObsCirclex', 'var')
            if length(hObsCirclex) < new_k
                hObsCirclex(new_k) = gobjects(1);
            end

            radiussafe = 0.25;
            circle_xx = new_pos(1) + radiussafe *
cos(theta_circle);
            circle_yy = new_pos(2) + radiussafe *
sin(theta_circle);
            circle_zz = new_pos(3) * ones(size(theta_circle));
            hObsCirclex(new_k) = plot3(circle_xx, circle_yy,
circle_zz, 'b', 'LineWidth', 3);
        end
    end
end

fprintf('Batch spawn #%d completed at iteration %d (Total
obstacles: %d)\n', sequential_spawn_count, i, M_obs);
else
    fprintf('Obstacle batch spawn blocked - agent near goal at
iteration %d\n', i);
end

% Schedule next spawn
next_spawn_iteration = i + SEQUENTIAL_SPAWN_INTERVAL;
end

% Task allocation
if mod(i, alocfreq)==1

%% 1. CRASH DETECTION AND HANDLING
for j = 1:N
    if ~crashed_agents(j) % Only check non-crashed agents
        agent_crashed = false;

        % Check crash with obstacles
        for b = 1:size(obstacles,2)

```

```

        dx = current_pos{j}(1) - obstacles(1,b);
        dy = current_pos{j}(2) - obstacles(2,b);
        dz = current_pos{j}(3) - obstacles(3,b);

        if static == 1 || (exist('obstacle_types', 'var') &&
obstacle_types(b) == 1)
            % Cylinder (static)
            if dx^2 + dy^2 < radius(b)^2 && current_pos{j}(3) <
obstacles(3,b)
                agent_crashed = true;
                break;
            end
        else
            % Sphere (dynamic)
            dist_3d = sqrt(dx^2 + dy^2 + dz^2);
            radiussafe = 0.25;
            if dist_3d < radiussafe
                agent_crashed = true;
                break;
            end
        end
    end
end

% Handle newly crashed agent
if agent_crashed
    crashed_agents(j) = true;
    old_target = assigned_goal(j);

    % Find nearest feasible point for crashed agent
    distances_to feas = arrayfun(@(idx) norm(current_pos{j} -
feas_point(idx,:)'), 1:size(feas_point,1));
    [~, nearest_idx] = min(distances_to feas);
    crash_recovery_points{j} = feas_point(nearest_idx,:');

    fprintf('CRASH: Agent %d crashed, was targeting Target %d,
now going to recovery point\n', j, old_target);

    % Trigger reallocation flag
    need_reallocation = true;
end
end

%% 2. TARGET COMPLETION TRACKING {FIXED}
%% 2. TARGET COMPLETION TRACKING {FIXED}
for j = 1:N
    if ~crashed_agents(j) && ~done(j)
        % Check if agent reached target
        target_pos = target{assigned_goal(j)};
        if norm(current_pos{j} - target_pos) < params.R_STOP
            if target_reached_time(j) == 0 % First time reaching - ONLY set
if zero
                target_reached_time(j) = i * dt; % Use iteration time, not
reset
                fprintf('SUCCESS: Agent %d reached Target %d at time %.2f
seconds (iteration %d)\n', ...
j, assigned_goal(j), target_reached_time(j), i);

                % CRITICAL: FREEZE the target assignment for this agent
                final_target_assignment(j) = assigned_goal(j); % Save
final assignment
            end
            done(j) = true;
        end
    end
end

```

```

%% 3. CHECK TARGET FILLING STATUS
active_agents = ~crashed_agents;
target_counts = zeros(1, M);
targets_reached_count = zeros(1, M);
targets_assigned_count = zeros(1, M);

for t = 1:M
    % Count agents that actually REACHED this target
    agents_on_target = find(assigned_goal == t & ~crashed_agents);
    targets_reached_count(t) = sum(done(agents_on_target));

    % Count agents currently assigned (including those still moving)
    targets_assigned_count(t) = sum(assigned_goal(~crashed_agents) == t);
end

% Find targets that have NO agents reached yet (even if assigned)
targets_not_reached = find(targets_reached_count == 0);
% Find targets that have at least one agent reached but not filled
targets_partially_reached = find(targets_reached_count > 0 &
targets_reached_count < params.quota);

% fprintf('Debug: Targets not reached: %s\n',
mat2str(targets_not_reached));
% fprintf('Debug: Targets partially reached: %s\n',
mat2str(targets_partially_reached));

for j = 1:N
    if ~crashed_agents(j) && ~done(j)
        current_target = assigned_goal(j);
        dist_to_current_target = norm(current_pos{j} -
target{current_target});

        % ADD THESE STABILITY CHECKS:
        % Don't reassign if agent is committed to current target
        if dist_to_current_target < params.COMMITMENT_DISTANCE
            continue; % Skip all reassignment logic
        end

        % Don't reassign if agent is close and target still needs them
        if dist_to_current_target < params.STABILITY_DISTANCE
            agents_close_to_current = sum(arrayfun(@(k) ~crashed_agents(k) && assigned_goal(k) == current_target && ...
norm(current_pos{k} - target{current_target}) < params.COMMITMENT_DISTANCE, 1:N));
            agents_reached_current = sum(done(find(assigned_goal == current_target)) & ~crashed_agents(find(assigned_goal == current_target)));

            if (agents_reached_current + agents_close_to_current) <=
params.quota(current_target)
                continue; % Don't reassign stable agents
            end
        end

        % PRIORITY 1: If there are targets with NO agents reached yet
        if ~isempty(targets_not_reached)
            % Check if current target already has someone who reached it
            if targets_reached_count(current_target) > 0
                % Current target already has someone, can move this agent
                % Find closest unreached target
                distances_to_unreached = arrayfun(@(t) norm(current_pos{j} -
target{t}), targets_not_reached);
                [min_dist, closest_idx] = min(distances_to_unreached);
                new_target = targets_not_reached(closest_idx);
            end
        end
    end
end

```

```

    % Only move if this won't leave current target with no
assigned agents    if targets_assigned_count(current_target) > 1
        old_target = assigned_goal(j);
        assigned_goal(j) = new_target;
        theta(j) = theta(j) + params.dtheta;

        % Update counts
        targets_assigned_count(old_target) =
targets_assigned_count(old_target) - 1;
        targets_assigned_count(new_target) =
targets_assigned_count(new_target) + 1;

        fprintf('PRIORITY 1: Agent %d moved from reached Target
%d to unreached Target %d (dist: %.2f)\n', ...
            j, old_target, new_target, min_dist);
        continue; % Skip normal allocation
    end
end
end

% PRIORITY 2: If all targets have at least one reached, fill quotas
for partially reached
    if isempty(targets_not_reached) &&
~isempty(targets_partially_reached)
        % Current target is over-quota and there are under-quota
targets
        if targets_reached_count(current_target) >=
params.quota(current_target)
            % Find closest partially reached target
            distances_to_partial = arrayfun(@(t) norm(current_pos{j} -
target{t}), targets_partially_reached);
            [min_dist, closest_idx] = min(distances_to_partial);
            new_target = targets_partially_reached(closest_idx);

            old_target = assigned_goal(j);
            benefit = dist_to_current_target -
distances_to_partial(closest_idx);
            if benefit < params.MIN_REASSIGN_BENEFIT
                continue; % Not worth reassigning
            end
            assigned_goal(j) = new_target;
            theta(j) = theta(j) + params.dtheta;

            fprintf('PRIORITY 2: Agent %d moved from full Target %d to
partial Target %d (dist: %.2f)\n', ...
                j, old_target, new_target, min_dist);
            continue; % Skip normal allocation
        end
end

% PRIORITY 3: Emergency - ensure no target has zero assigned agents
for t = 1:M
    if targets_assigned_count(t) == 0
        fprintf('EMERGENCY: Target %d has no assigned agents!\n',
t);
        old_target = assigned_goal(j);
        assigned_goal(j) = t;
        theta(j) = theta(j) + params.dtheta;

        fprintf('EMERGENCY ASSIGN: Agent %d moved from Target %d to
empty Target %d\n', ...
            j, old_target, t);
        break;
    end
end

```

```

% Only call normal allocation if agent is not close to target
if dist_to_current_target > params.STABILITY_DISTANCE
    [assigned_goal, theta, requests] = alokasi_tugasnew(j,
current_pos, assigned_goal, theta, neighbors, target, params, requests,
done);
end
end

%% 5. REALLOCATION AFTER CRASHES
if exist('need_reallocation', 'var') && need_reallocation
    fprintf('==> REALLOCATING AFTER CRASH ==\n');
%
% Get active agents and recalculate optimal assignment
active_agent_indices = find(~crashed_agents);
N_active = length(active_agent_indices);
%
if N_active > 0
    % Recalculate quotas for active agents
    base_quota = floor(N_active / M);
    remainder = mod(N_active, M);
    new_quotas = base_quota * ones(1, M);
%
    % Distribute remainder to unfilled targets first
    unfilled_indices = find(~target_filled);
    if length(unfilled_indices) >= remainder
        new_quotas(unfilled_indices(1:remainder)) =
new_quotas(unfilled_indices(1:remainder)) + 1;
    %
    else
        new_quotas(unfilled_indices) =
new_quotas(unfilled_indices) + 1;
    %
    remaining = remainder - length(unfilled_indices);
    if remaining > 0
        other_targets = setdiff(1:M, unfilled_indices);
        new_quotas(other_targets(1:remaining)) =
new_quotas(other_targets(1:remaining)) + 1;
    %
    end
    %
    % Create distance matrix for active agents
    distances = [];
    for idx = 1:N_active
        j = active_agent_indices(idx);
        if ~done(j) % Only reassign agents that haven't reached
target
            for t = 1:M
                distances = [distances; j, t, norm(current_pos{j}
- target{t})];
            end
        end
    end
%
% Sort by distance and reassign
if ~isempty(distances)
    [~, sort_idx] = sort(distances(:, 3));
    sorted_distances = distances(sort_idx, :);
%
% Reset assignment counts
new_assignment_counts = zeros(1, M);
%
% Greedy reassignment
for k = 1:size(sorted_distances, 1)
    j = sorted_distances(k, 1); % agent index
    t = sorted_distances(k, 2); % target index
%

```

```

%
%           if new_assignment_counts(t) < new_quotas(t)
%               old_target = assigned_goal(j);
%               assigned_goal(j) = t;
%               new_assignment_counts(t) =
new_assignment_counts(t) + 1;
%
%           if old_target ~= t
%               fprintf('REALLOC: Agent %d moved from Target
%d to Target %d\n', j, old_target, t);
%
%           end
%
%       end
%
%   end
%
%   clear need_reallocation; % Reset flag
%   fprintf('== RELOCATION COMPLETE ==\n');
% end
%
%% 6. ENSURE ALL TARGETS COVERED (Safety check)
for t = 1:M
    if ~any(assigned_goal(~crashed_agents) == t) && ~target_filled(t)
        % Find nearest active agent to assign to this target
        active_agents_list = find(~crashed_agents);
        if ~isempty(active_agents_list)
            distances = arrayfun(@(j) norm(current_pos{j} - target{t}), active_agents_list);
            [~, min_idx] = min(distances);
            reassign_agent = active_agents_list(min_idx);

            old_target = assigned_goal(reassign_agent);
            assigned_goal(reassign_agent) = t;
            theta(reassign_agent) = theta(reassign_agent) +
params.dtheta;
            fprintf('SAFETY: Agent %d assigned to uncovered Target
%d\n', reassign_agent, t);
        end
    end
end

% Update neighbors for active agents only
for j=1:N
    if ~crashed_agents(j) && ~done(j)
        neighbors{j} = find_neighbors(j, current_pos,
params.comm_range);
    end
end
all_agent_goals = zeros(3, N);
for agent_idx = 1:N
    target_id = assigned_goal(agent_idx);
    if target_id >= 1 && target_id <= length(target)
        all_agent_goals(:, agent_idx) = target{target_id};
    else
        % Fallback to first target if invalid assignment
        all_agent_goals(:, agent_idx) = target{1};
    end
end

step = floor(i/aLocFreq)+1;
assigned_history(:, step) = assigned_goal(:);
theta_history(:, step) = theta(:);

```

```

alloc_step = floor(i/aLocFreq)+1;
assigned_history(:,alloc_step) = assigned_goal';
theta_history(:,alloc_step) = theta';

% Agent control loop
% Replace the agent control loop section (around line 800+) with
this modified version:

% Agent control loop
% Agent control loop - REPLACE YOUR EXISTING LOOP WITH THIS
for j=1:N

    % ===== CALCULATE DISTANCE TO TARGET FOR ALL AGENTS =====
    target_pos = target{assigned_goal(j)};
    distance_to_target = norm(current_pos{j} - target_pos);

    %  CRITICAL: Skip all processing for done agents except
    % visualization and logging
    if done(j)
        % Agent is completely done - maintain stopped state
        state{j}(4:6) = zeros(3,1); % Zero velocity
        state{j}(10:12) = zeros(3,1); % Zero angular velocity
        state{j}(1:3) = current_pos{j}; % Keep actual position
        Ft{j} = zeros(3,1);

        %  FREEZE the target assignment
        if exist('final_target_assignment', 'var') &&
final_target_assignment(j) == 0
            final_target_assignment(j) = assigned_goal(j);
        end

        %  LOG DISTANCE for stopped agents too
        fprintf('Stopped Agent %d: distance to target = %.3f at iteration
%d\n', j, distance_to_target, i);

        % Update visualization and logging
        posA_log(j,:,:) = current_pos{j};
        data_points{j}(i,:) = transpose(current_pos{j});

        % Update visualization
        if exist('hAgentQuad', 'var') && iscell(hAgentQuad) &&
length(hAgentQuad) >= j && isvalid(hAgentQuad{j})
            update_quadcopter_graphics(hAgentQuad{j}, current_pos{j},
state{j}(7:9));
        end
        set(htext{j}, 'Position', current_pos{j} - [1;1;1]);

        continue; %  CRITICAL: Skip all other processing for done agents
    end

    % Handle crashed agents differently
    if crashed_agents(j)
        % Crashed agents go to recovery points, not targets
        if ~isempty(crash_recovery_points{j})
            goal{j} = crash_recovery_points{j};
        else
            % Fallback to original behavior
            if jarak3(start{j},current_pos{j})<8
                goal{j}=feas_point(randi(Nfeas),:)';
            end
        end
    else
        % Normal agents go to assigned targets
        if jarak3(start{j},current_pos{j})<8
            goal{j}=feas_point(randi(Nfeas),:)';
        end
    end
end

```

```

        end
    goal{j} = target{ assigned_goal(j) };

end

% ===== AUTO-STOP FUNCTIONALITY (IMPROVED) =====
if ~done(j)
    %  LOG DISTANCE for moving agents
    % fprintf('Moving Agent %d: distance to target = %.3f at iteration
%d\n', j, distance_to_target, i);

    if distance_to_target <= params.R_STOP
        % Mark as done and record the ACTUAL stopping position
        if target_reached_time(j) == 0
            target_reached_time(j) = i * dt;
            fprintf('SUCCESS: Agent %d reached Target %d at time %.2f
seconds (iteration %d)\n', ...
                j, assigned_goal(j), target_reached_time(j), i);

        % SAVE THE ACTUAL STOPPING POSITION (not target position)
        final_stopping_pos{j} = current_pos{j}; % Save where it
actually stopped
        final_target_assignment(j) = assigned_goal(j); % Freeze
assignment

        fprintf(' Actual stopping position: [%.2f, %.2f, %.2f]\n',
...
current_pos{j}(1), current_pos{j}(2),
current_pos{j}(3));
        fprintf(' Target position: [%.2f, %.2f, %.2f]\n', ...
target_pos(1), target_pos(2), target_pos(3));
        fprintf(' Final distance to target: %.3f units\n',
distance_to_target);
    end
    done(j) = true;

    % STOP MOVEMENT - but keep actual position
    state{j}(4:6) = zeros(3,1); % Zero velocity
    state{j}(10:12) = zeros(3,1); % Zero angular velocity

    % Keep agent at LAST REAL POSITION (not target position)
    state{j}(1:3) = current_pos{j}; % Maintain actual stopping
position

    % Skip force calculation for stopped agents
    Ft{j} = zeros(3,1);

    % Update visualization at ACTUAL position
    if assigned_goal(j) > 0 && assigned_goal(j) <= size(cmap,1)
        col = cmap(assigned_goal(j),:);
    else
        col = [0 0 1];
    end
    if exist('hAgentQuad', 'var') && iscell(hAgentQuad) &&
length(hAgentQuad) >= j && isvalid(hAgentQuad{j})
        update_quadcopter_graphics(hAgentQuad{j}, current_pos{j},
state{j}(7:9));
    end

    % Update text position at actual stopping location
    set(htext{j}, 'Position', current_pos{j} - [1;1;1]);

    % Continue to next agent (skip force calculation)
    continue;
end
end

```

```

% ===== ORIGINAL FORCE CALCULATION (for moving agents only) =====
group = find(assigned_goal == assigned_goal(j));
N_agents = numel(current_pos);
idx_other = setdiff(1:N_agents, j);
agenlain = current_pos(idx_other);
pos = current_pos{:,j};
vcur = state{j}(4:6);
min_obs_dist = inf;

for k = 1:size(obstacles,2)
    if static == 1
        dx = pos(1:2) - obstacles(1:2,k);
        dist = norm(dx);
        if pos(3) < obstacles(3,k)
            min_obs_dist = min(min_obs_dist, dist - radius(k));
        end
    else
        radiussafe = 0.25;
        dist = norm(pos - obstacles(:,k)) - radiussafe;
        min_obs_dist = min(min_obs_dist, dist);
    end
end

% NEW: Add other agents as detected obstacles (within communication
range)
detected_agent_indices = [];
detected_agent_pos = [];
detected_agent_speeds = [];
params.comm_range = 100;
radiussafe = 0.25; % Obstacles Safe Radius
for k = 1:N
    if k ~= j && ~done(k)
        agent_dist = norm(pos - current_pos{k});
        if agent_dist <= params.comm_range % Only detect agents
within range
            detected_agent_indices = [detected_agent_indices, k];
            detected_agent_pos = [detected_agent_pos, current_pos{k}];
            detected_agent_speeds = [detected_agent_speeds,
state{k}(4:6)];
        end
    end
end

% Create fake obstacles for other targets
other_tg = setdiff(1:M, assigned_goal(j));
tg_obs = cell2mat(target(other_tg))';
tg_rad = params.MinDistance * ones(numel(other_tg),1);
tg_speeds = zeros(3, numel(other_tg));

% Combine all obstacles
all_obs = [obstacles];
all_radius = [radius];
all_speeds = [obs_speeds];
speed_agent = [detected_agent_speeds];
agent_radius = [radiussafe * ones(length(detected_agent_indices), 1)];
agent_pos = [detected_agent_pos];

% Get additional forces based on selected APF algorithm
switch APF_type
    case 1
        % Traditional APF
        [Ft_existing, Fatt_j, Frep_j] = traditional_APF( ...
            j, target, current_pos{j}, static, ...

```

```

goal{j}, ...
all_obs, all_radius, ...
gains_att, gains_rep, ...
params);

case 2
    % Modified APF (MAPF)
    [Ft_existing, Fatt_j, Frep_j] = modified_APF( ...
        j, target, current_pos{j}, static, ...
        goal{j}, ...
        vcur, ...
        all_obs, all_radius, ...
        gains_att, gains_rep, ...
        params, ...
        desired_speed);

case 3
    % Velocity APF (VAPF)
    [Ft_existing, Fatt_j, Frep_j] = velocity_APF( ...
        j, target, current_pos{j}, static, ...
        goal{j}, ...
        vcur, ...
        all_obs, all_radius, ...
        all_speeds, ...
        gains_att, gains_rep, ...
        params);

case 4
    % Dynamic APF (DAPF) - Current implementation
    if size(all_speeds, 2) < size(all_obs, 2)
        all_speeds = [all_speeds, zeros(3, size(all_obs, 2) - size(all_speeds,
2))];
    end
    [Ft_existing, Fatt_j, Frep_j] = avoidance_multiRobot_pure_dapf(
...
        j, target, current_pos{j}, static, ...
        goal{j}, ...
        vcur, ...
        Kform, ...
        all_obs, all_radius, ...
        F, ...
        agenlain, grouppp, ...
        params, ...
        params_dist, ...
        gains_att, gains_rep, ...
        desired_speed, ...
        all_speeds, speed_agent, agent_radius, agent_pos,
        all_agent_goals);

    otherwise
        error('Invalid APF_type. Use 1=Traditional, 2=MAPF, 3=VAPF,
4=DAPF');
    end

    % Combine forces
    Ft{j} = Ft_existing;

    % Store forces for logging
    posA_log(j,:,:) = current_pos{j}';
    Fatt_log(j,:,:) = Fatt_j';
    Frep_log(j,:,:) = Frep_j';

    % Update state
    previous_pos{j} = current_pos{j};

% Initialize Ftp (previous force) if it doesn't exist

```

```

if ~exist('Ftp', 'var') || length(Ftp) < j || isempty(Ftp{j})
    Ftp{j} = zeros(3,1); % Initialize previous force
end

% Update state using new dynamics
state{j} = UAV_input_Ft(state{j}, Ft{j}, Ftp{j},dt);
current_pos{j} = state{j}(1:3);

% Update previous force for next iteration
Ftp{j} = Ft{j};
if j == 3 && mod(i, 50) == 0 % Every 50 iterations for Agent 3
    target_pos = target{assigned_goal(j)};
    dist_to_target_debug = norm(current_pos{j} - target_pos);
    fprintf('Iter %d: Agent 3 at [%f, %f, %f], dist to target: %f, velocity: [%f, %f, %f]\n', ...
        i, current_pos{j}(1), current_pos{j}(2), current_pos{j}(3), ...
        dist_to_target_debug, state{j}(4), state{j}(5), state{j}(6));
end

% Track agent speed
agent_speeds(j, i) = norm(state{j}(4:6));

% CRASH DETECTION - Fixed to use current obstacle positions
for b = 1:size(obstacles,2)
    dx = current_pos{j}(1) - obstacles(1,b);
    dy = current_pos{j}(2) - obstacles(2,b);
    dz = current_pos{j}(3) - obstacles(3,b);

    % Check obstacle type
    if static == 1 || (static == 0 && mixed == 1 && obstacle_types(b)
== 1)
        % Cylinder (static)
        if dx^2 + dy^2 < radius(b)^2 && current_pos{j}(3) <
obstacles(3,b)
            fprintf('Crash detected (static): UAV %d with obstacle %d
at iteration %d\n', j, b, i);
            crash(j,i) = true;
            break;
        end
    else
        % Sphere (dynamic) - proper 3D distance check
        dist_3d = sqrt(dx^2 + dy^2 + dz^2);
        radiussafe = 0.25;
        if dist_3d < radiussafe;
            fprintf('Crash detected (dynamic): UAV %d with obstacle %d
at iteration %d\n', j, b, i);
            fprintf('UAV pos: [%f, %f, %f], Obs pos: [%f,
%f, %f], Dist: %f, Radius: %f\n', ...
                current_pos{j}(1), current_pos{j}(2),
                current_pos{j}(3), ...
                obstacles(1,b), obstacles(2,b), obstacles(3,b),
                dist_3d, radius(b));
            crash(j,i) = true;
            break;
        end
    end
end

% Store data
data_points{j}(i,:) = transpose(current_pos{j});
all_state{j}(:,i+1) = state{j}';
all_Ft{j}(:,i+1) = Ft{j};

% Update agent quadcopter visualization
if assigned_goal(j) > 0 && assigned_goal(j) <= size(cmap,1)

```

```

        col = cmap(assigned_goal(j),:);
    else
        col = [0 0 1];
    end
    if exist('hAgentQuad', 'var') && iscell(hAgentQuad) &&
length(hAgentQuad) >= j && isvalid(hAgentQuad{j})
        update_quadcopter_graphics(hAgentQuad{j}, current_pos{j},
state{j}(7:9));
    end

    % Update trajectory with color changes based on assignment (works for
both static and dynamic)
    if exist('hTraj', 'var') && ~isempty(hTraj)
        % Check if assignment changed
        if prev_assigned_goal(j) ~= assigned_goal(j)
            % Assignment changed - finish current segment and start new one
            if ~isempty(hTraj{j}) && segment_start_idx(j) < i
                % Update the last segment to current position
                set(hTraj{j}(end), ...
                    'XData', data_points{j}(segment_start_idx(j):i,1), ...
                    'YData', data_points{j}(segment_start_idx(j):i,2), ...
                    'ZData', data_points{j}(segment_start_idx(j):i,3));
            end

            % Start new segment with new color
            segment_start_idx(j) = i;
            prev_assigned_goal(j) = assigned_goal(j);

            % Create new line segment with target color
            if assigned_goal(j) > 0 && assigned_goal(j) <= size(cmap,1)
                col = cmap(assigned_goal(j),:);
            else
                col = [0.5 0.5 0.5];
            end

            new_line = plot3(NaN, NaN, NaN, 'LineWidth', 1.5, 'Color',
col);
            if isempty(hTraj{j})
                hTraj{j} = new_line;
            else
                hTraj{j}(end+1) = new_line;
            end
        else
            % Same assignment - update current segment
            if isempty(hTraj{j})
                % Create first segment
                if assigned_goal(j) > 0 && assigned_goal(j) <= size(cmap,1)
                    col = cmap(assigned_goal(j),:);
                else
                    col = [0.5 0.5 0.5];
                end
                hTraj{j} = plot3(NaN, NaN, NaN, 'LineWidth', 1.5, 'Color',
col);
            end

            % Update the current segment
            set(hTraj{j}(end), ...
                'XData', data_points{j}(segment_start_idx(j):i,1), ...
                'YData', data_points{j}(segment_start_idx(j):i,2), ...
                'ZData', data_points{j}(segment_start_idx(j):i,3));
        end
    end

    % Update text position
    set(htext{j}, 'Position', current_pos{j} - [1;1;1]);

```

```

% Distance logging
% ✓ FIXED: Distance logging with proper freezing for stopped agents
if ~done(j) || i == 1
    % Calculate distance to ALL obstacles for this agent
    min_dist_to_any_obstacle = inf;

    for k = 1:M_obs
        if k <= size(obstacles, 2)
            obs_pos = obstacles(:, k);
            obs_radius = radius(k);
            agent_pos = current_pos{j};

            if static == 1
                % 🔧 FIXED: Static cylinder distance calculation
                % Distance from agent to cylinder surface

                % Horizontal distance to cylinder center
                dx_2d = agent_pos(1) - obs_pos(1);
                dy_2d = agent_pos(2) - obs_pos(2);
                horizontal_dist = sqrt(dx_2d^2 + dy_2d^2);

                if agent_pos(3) <= obs_pos(3)
                    % Agent is below or at obstacle height
                    if horizontal_dist <= obs_radius
                        % Agent is INSIDE the cylinder - calculate
                        distance to edge
                        horizontal_dist;
                        dist_to_cylinder_edge = obs_radius -
                            % Very close but not zero
                        else
                            % Agent is OUTSIDE the cylinder - distance to
                            surface
                            actual_distance = horizontal_dist - obs_radius;
                        end
                    else
                        % Agent is ABOVE the obstacle
                        if horizontal_dist <= obs_radius
                            % Agent is directly above cylinder
                            actual_distance = agent_pos(3) - obs_pos(3);
                        else
                            % Agent is above and to the side - 3D distance
                            to corner
                            edge_point = [obs_pos(1) + obs_radius *
                                (dx_2d/horizontal_dist);
                                obs_pos(2) + obs_radius *
                                (dy_2d/horizontal_dist);
                                obs_pos(3)];
                            actual_distance = norm(agent_pos - edge_point);
                        end
                    end
                else
                    % 🔧 FIXED: Dynamic sphere distance calculation
                    center_to_center = norm(agent_pos - obs_pos);
                    actual_distance = center_to_center - 0.25;
                end
            end
        end
    end
end

% 🔧 Ensure minimum distance but don't artificially
inflate

% Track minimum distance to any obstacle
min_dist_to_any_obstacle = min(min_dist_to_any_obstacle,
actual_distance);

% Store individual obstacle distance
if ~exist('distance_obs_all', 'var')

```

```

        distance_obs_all = zeros(N, M_obs, iteration);
    end
    distance_obs_all(j, k, i) = actual_distance;

    % 🔪 DEBUG: Print suspicious distances
    if actual_distance < 0.1 && mod(i, 100) == 0
        fprintf('⚠ Agent %d very close to Obstacle %d:
dist=%.3f at iter %d\n', ...
            j, k, actual_distance, i);
    end
end

% Store the minimum distance to any obstacle
distance_obs(j, i) = min_dist_to_any_obstacle;

else
    % 🔪 AGENT HAS STOPPED: Freeze distances properly
    if target_reached_time(j) > 0
        stop_iteration = round(target_reached_time(j) / dt);
        stop_iteration = max(1, min(stop_iteration, i-2));

        if stop_iteration > 0 && stop_iteration <= size(distance_obs,
2)
            % Copy the ACTUAL calculated distance from when it stopped
            frozen_distance = distance_obs(j, stop_iteration);

            % 🔪 Only use frozen distance if it's reasonable
            if frozen_distance > 0 && isinfinite(frozen_distance)
                distance_obs(j, i) = frozen_distance;

                % Also freeze individual obstacle distances
                if exist('distance_obs_all', 'var')
                    for k = 1:M_obs
                        if stop_iteration <= size(distance_obs_all, 3)
                            frozen_k = distance_obs_all(j, k,
stop_iteration);
                            if frozen_k > 0 && isinfinite(frozen_k)
                                distance_obs_all(j, k, i) = frozen_k;
                            end
                        end
                    end
                else
                    fprintf('⚠ Invalid frozen distance for Agent %d:
%.6f\n', j, frozen_distance);
                    distance_obs(j, i) = 1.0; % Reasonable fallback
                end
            end
        end
    end
end

% Record inter-agent distances
for a = 1:N
    for b = a+1:N
        d = norm(current_pos{a} - current_pos{b});
        distance_agents(a,b,i) = d;
        distance_agents(b,a,i) = d;
    end
end

% Update display
subtitlehandle.String = sprintf('Iteration %d', i);
% Save Replay Data

```

```

    %% Save Replay Data (FIXED VERSION - Replace your existing section)
if SAVE_3D_REPLAY
    replay_frame_count = replay_frame_count + 1;

    % Save agent data
    for j = 1:N
        replay_data.dynamic.agent_positions(j, :, replay_frame_count) =
current_pos{j}';
        replay_data.dynamic.agent_orientations(j, :, replay_frame_count) =
state{j}(7:9)';
        replay_data.dynamic.agent_assignments(j, replay_frame_count) =
assigned_goal(j);
        replay_data.dynamic.agent_status(j, replay_frame_count) = done(j);
        replay_data.dynamic.agent_crashed(j, replay_frame_count) =
crashed_agents(j);

        % Save trajectory points
        if replay_frame_count == 1
            replay_data.dynamic.trajectories{j} = current_pos{j}';
        else
            replay_data.dynamic.trajectories{j}(end+1, :) =
current_pos{j}';
        end
    end

    % Save target reached times
    replay_data.dynamic.target_reached_time = target_reached_time;

    %% FIXED: Save dynamic obstacle data with sequential spawning
support
    if static == 0
        %% CRITICAL: Save the current number of obstacles
        replay_data.dynamic.num_obstacles_per_frame(replay_frame_count) =
M_obs;

        %% FIXED: Save obstacle data for ALL possible obstacles
        % (including future spawns)
        max_obs_slots = size(replay_data.dynamic.obstacle_positions, 1);

        for k = 1:max_obs_slots
            if k <= M_obs && k <= size(obs_states, 2)
                % This obstacle exists - save its state
                replay_data.dynamic.obstacle_positions(k, :, replay_frame_count) =
obs_states(1:3, k)';
                replay_data.dynamic.obstacle_orientations(k, :, replay_frame_count) =
obs_states(7:9, k)';
                replay_data.dynamic.obstacle_states(:, k, replay_frame_count) =
obs_states(:, k);

                % Debug output for sequential spawns
                if k > original_M_obs && replay_frame_count > 1
                    prev_pos = replay_data.dynamic.obstacle_positions(k, :, replay_frame_count-1);
                    if all(prev_pos == 0) && any(obs_states(1:3, k) ~= 0)
                        fprintf(' RECORDING: Obstacle %d spawned at frame %d, pos [% .1f, %.1f, %.1f]\n', ...
                                k, replay_frame_count, obs_states(1,k), obs_states(2,k), obs_states(3,k));
                    end
                end
            else
                % This obstacle doesn't exist yet - fill with zeros
                replay_data.dynamic.obstacle_positions(k, :, replay_frame_count) =
[0, 0, 0];
                replay_data.dynamic.obstacle_orientations(k, :, replay_frame_count) =
[0, 0, 0];
            end
        end
    end

```

```

        if size(replay_data.dynamic.obstacle_states, 2) >= k
            replay_data.dynamic.obstacle_states(:, k,
replay_frame_count) = zeros(12, 1);
        end
    end
end

% 🔧 Debug: Print obstacle count changes
if replay_frame_count > 1
    prev_count =
replay_data.dynamic.num_obstacles_per_frame(replay_frame_count-1);
    if M_obs > prev_count
        fprintf('📈 SEQUENTIAL SPAWN: Obstacle count increased
from %d to %d at frame %d\n', ...
                prev_count, M_obs, replay_frame_count);
    end
end
end

drawnow;

agents_reached_target = 0;
total_active_agents = 0;

% Debug output every 100 iterations
if mod(i, 100) == 0
    fprintf('\n--- Completion Check at Iteration %d ---\n', i);
end

for j = 1:N
    if ~crashed_agents(j) % Only count non-crashed agents
        total_active_agents = total_active_agents + 1;

        % Check current distance to assigned target
        target_pos = target{assigned_goal(j)};
        current_dist = norm(current_pos{j} - target_pos);

        % Agent is considered "reached" if marked as done
        if done(j)
            agents_reached_target = agents_reached_target + 1;

            % Debug output every 100 iterations for reached agents
            if mod(i, 100) == 0
                fprintf(' Agent %d: REACHED Target %d (dist=%2f,
time=%2f)\n', ...
                        j, assigned_goal(j), current_dist,
target_reached_time(j));
            end
        else
            % Debug output for agents still moving (only if close)
            if current_dist < 10 && mod(i, 100) == 0
                fprintf(' Agent %d: MOVING to Target %d (dist=%2f)\n',
...
                        j, assigned_goal(j), current_dist);
            end
        end
    end
end

% Primary completion condition: All active agents have reached their
targets
mission_complete = (agents_reached_target == total_active_agents) &&
(total_active_agents > 0);

```

```

% Debug summary every 100 iterations
if mod(i, 100) == 0
    fprintf(' Summary: %d/%d agents reached targets\n',
agents_reached_target, total_active_agents);
    % fprintf(' All targets covered: %s\n', char(all_targets_covered +
"false"));
    fprintf(' Mission complete: %s\n', char(mission_complete + "false"));
end
%%
% ===== STOP SIMULATION IF MISSION COMPLETE =====
if mission_complete
    % Calculate final timing
    final_simulation_time = toc;
    final_iteration_time = i * dt;

    fprintf('\n==== MISSION COMPLETED ===\n');

    % ADD THIS HERE:
    % Save replay data
    if SAVE_3D_REPLAY
        replay_data.actual_iterations = replay_frame_count;
        replay_data.final_simulation_time = final_simulation_time;
        replay_data.final_iteration_time = final_iteration_time;
        save_replay_data(replay_data, 'completed');
    end

    % Your existing completion code continues...
    if mission_complete
        fprintf('✓ All active agents (%d/%d) reached their targets\n',
agents_reached_target, total_active_agents);
    end

    fprintf('Simulation stopped at iteration %d\n', i);
    fprintf('Total simulation time: %.2f seconds (real time: %.2f
seconds)\n', final_iteration_time, final_simulation_time);

% ===== DETAILED TARGET COMPLETION ANALYSIS =====
fprintf('\n==== TARGET COMPLETION DETAILS ===\n');
for t = 1:M
    agents_on_target = find(assigned_goal == t & ~crashed_agents);
    agents_reached_target_t = agents_on_target(done(agents_on_target));

    quota_t = 1; % Default quota
    if exist('params', 'var') && isfield(params, 'quota') && t <=
length(params.quota)
        quota_t = params.quota(t);
    end

    fprintf('Target %d (Quota: %d):\n', t, quota_t);

    if ~isempty(agents_reached_target_t)
        % Get completion times for this target
        completion_times =
target_reached_time(agents_reached_target_t);
        valid_times = completion_times(completion_times > 0);
        valid_agents = agents_reached_target_t(completion_times > 0);

        if ~isempty(valid_times)
            [sorted_times, sort_idx] = sort(valid_times);
            sorted_agents = valid_agents(sort_idx);

            for idx = 1:length(sorted_agents)
                ag = sorted_agents(idx);

```

```

        final_dist = norm(current_pos{ag} - target{t});
        fprintf('  → Agent %d reached at %.2f seconds (final
dist: %.3f)\n', ...
            ag, sorted_times(idx), final_dist);
    end

    % Target completion statistics
    if length(sorted_times) > 1
        fprintf('  Time span: %.2f seconds (%.2f to %.2f)\n',
...
            max(sorted_times) - min(sorted_times),
min(sorted_times), max(sorted_times));
    else
        fprintf('  Single agent completion at %.2f seconds\n',
sorted_times(1));
    end
    else
        fprintf('  → Agents reached but timing data missing\n');
    end
else
    fprintf('  → No agents reached this target\n');
end
fprintf('\n');

% ===== OVERALL MISSION STATISTICS =====
successful_times = [];
for j = 1:N
    if j <= length(crashed_agents) && j <= length(target_reached_time)
        if ~crashed_agents(j) && target_reached_time(j) > 0
            successful_times(end+1) = target_reached_time(j);
        end
    end
end

if ~isempty(successful_times)
    fprintf('==> OVERALL MISSION STATISTICS ==>\n');
    [min_time, min_idx] = min(successful_times);
    [max_time, max_idx] = max(successful_times);

    % Find agent indices for min/max times
    successful_agents = [];
    for j = 1:N
        if j <= length(crashed_agents) && j <=
length(target_reached_time)
            if ~crashed_agents(j) && target_reached_time(j) > 0
                successful_agents(end+1) = j;
            end
        end
    end

    if ~isempty(successful_agents)
        first_agent = successful_agents(min_idx);
        last_agent = successful_agents(max_idx);

        fprintf('First completion: %.2f seconds (Agent %d → Target
%d)\n', ...
            min_time, first_agent, assigned_goal(first_agent));
        fprintf('Last completion: %.2f seconds (Agent %d → Target
%d)\n', ...
            max_time, last_agent, assigned_goal(last_agent));
        fprintf('Average completion: %.2f seconds\n',
mean(successful_times));
        fprintf('Mission duration: %.2f seconds\n', max_time -
min_time);
        fprintf('Success rate: %.1f%% (%d/%d agents)\n', ...

```

```

N); (length(successful_times)/N)*100, length(successful_times),
    end
    end
    fprintf('=====\\n\\n');

    % Set iterend before breaking
    iterend = i;
    break; % Exit the main simulation loop
end

% ===== CHECK FOR USER INTERRUPTION =====
if exist('fhandle', 'var') && isvalid(fhandle) &&
~isempty(fhandle.CurrentCharacter)
    fprintf('\\nSimulation stopped by user at iteration %d\\n', i);

    % ADD THIS HERE:
    % Save replay data even if interrupted
    if SAVE_3D_REPLAY
        replay_data.actual_iterations = replay_frame_count;
        save_replay_data(replay_data, 'interrupted');
    end

    iterend = i;
    break;
end

% This should be the END of your main iteration loop
end % End of main "for i=1:iteration" loop

if ~exist('iterend', 'var')
    iterend = iteration; % If loop completed normally
end

disp('Simulation Finished');
if exist('titlehandle', 'var') && isvalid(titlehandle)
    titlehandle.String = sprintf('Simulation Finished - %s',
apf_names{APF_type});
end
elapsedSim = toc;

1. Output Command Windows

% ===== FINAL MISSION RESULTS (Safe Array Access) =====
fprintf('\\n== FINAL MISSION RESULTS ==\\n');

% Ensure all arrays are the correct size
required_length = N;
if length(crashed_agents) ~= required_length
    fprintf('Warning: Resizing crashed_agents array from %d to %d\\n',
length(crashed_agents), required_length);
    if length(crashed_agents) < required_length
        crashed_agents(end+1:required_length) = false;
    else
        crashed_agents = crashed_agents(1:required_length);
    end
end

if length(target_reached_time) ~= required_length
    fprintf('Warning: Resizing target_reached_time array from %d to %d\\n',
length(target_reached_time), required_length);
    if length(target_reached_time) < required_length
        target_reached_time(end+1:required_length) = 0;
    else

```

```

        target_reached_time = target_reached_time(1:required_length);
    end
end

if length(done) ~= required_length
    fprintf('Warning: Resizing done array from %d to %d\n', length(done),
    required_length);
    if length(done) < required_length
        done(end+1:required_length) = false;
    else
        done = done(1:required_length);
    end
end

if length(assigned_goal) ~= required_length
    fprintf('Warning: Resizing assigned_goal array from %d to %d\n',
    length(assigned_goal), required_length);
    if length(assigned_goal) < required_length
        assigned_goal(end+1:required_length) = 1; % Default to target 1
    else
        assigned_goal = assigned_goal(1:required_length);
    end
end

% Display final status for each agent
fprintf('Final Agent Status:\n');
for j = 1:N
    if crashed_agents(j)
        fprintf(' Agent %d: CRASHED\n', j);
    elseif done(j) && target_reached_time(j) > 0
        final_dist = norm(current_pos{j} - target{assigned_goal(j)});
        fprintf(' Agent %d: REACHED Target %d at %.2f seconds (final dist:
        %.3f)\n', ...
            j, assigned_goal(j), target_reached_time(j), final_dist);
    else
        fprintf(' Agent %d: DID NOT REACH target\n', j);
    end
end

% Final statistics
total_crashed = sum(crashed_agents);
total_successful = sum(~crashed_agents & done & target_reached_time > 0);

fprintf('\nFinal Mission Summary:\n');
fprintf(' Total agents: %d\n', N);
fprintf(' Successful: %d\n', total_successful);
fprintf(' Crashed: %d\n', total_crashed);
fprintf(' Success rate: %.1f%%\n', (total_successful/N)*100);
fprintf(' Total simulation time: %.2f seconds\n', elapsedSim);
fprintf('=====\\n');

% Save crash data
if exist('crash', 'var')
    save('crash_data.mat', 'crash');
end

% ANALYSIS OF STOPPING ACCURACY
fprintf('\\n== STOPPING ACCURACY ANALYSIS ==\\n');
fprintf('Target stopping threshold (R_STOP): %.3f units\\n', params.R_STOP);
fprintf('%-8s %-12s %-15s %-15s %-15s\\n', 'Agent', 'Stop Time', 'Actual
Dist', 'Target Pos', 'Stop Pos');
fprintf('%s\\n', repmat('-', 1, 80));

for j = 1:N

```

```

if done(j) && ~isempty(final_stopping_pos{j})
    target_pos = target{assigned_goal(j)};
    actual_dist = norm(final_stopping_pos{j} - target_pos);

    fprintf('-%8d %-12.2f %-15.3f [% .1f,% .1f,% .1f] [% .1f,% .1f,% .1f]\n',
...
    j, target_reached_time(j), actual_dist, ...
    target_pos(1), target_pos(2), target_pos(3), ...
    final_stopping_pos{j}(1), final_stopping_pos{j}(2),
final_stopping_pos{j}(3));
    end
end
%%

% Statistics
stopped_agents = find(done);
if ~isempty(stopped_agents)
    actual_distances = zeros(length(stopped_agents), 1);
    for idx = 1:length(stopped_agents)
        j = stopped_agents(idx);
        if ~isempty(final_stopping_pos{j})
            actual_distances(idx) = norm(final_stopping_pos{j} -
target{assigned_goal(j)}));
        end
    end

    fprintf('\n==== STOPPING STATISTICS ====\n');
    fprintf('Number of agents that stopped: %d\n', length(stopped_agents));
    fprintf('Mean stopping distance: %.3f units\n',
mean(actual_distances));
    fprintf('Std stopping distance: %.3f units\n', std(actual_distances));
    fprintf('Min stopping distance: %.3f units\n', min(actual_distances));
    fprintf('Max stopping distance: %.3f units\n', max(actual_distances));
    fprintf('Agents within R_STOP threshold: %d/%d (%.1f%%)\n',
...
    sum(actual_distances <= params.R_STOP), length(stopped_agents), ...
    100 * sum(actual_distances <= params.R_STOP) /
length(stopped_agents));
end

%% Fixed Table Creation Section
numRows = N * iterend;

Iterasi = repmat((1:iterend)', N, 1);
Agen = repelem((1:N)', iterend, 1);

Ax = reshape(posA_log(:,1,1:iterend), [], 1);
Ay = reshape(posA_log(:,2,1:iterend), [], 1);
Az = reshape(posA_log(:,3,1:iterend), [], 1);

if static == 0
    Ox = repmat(reshape(pos0_log(1,1,1:iterend), [], 1), N, 1);
    Oy = repmat(reshape(pos0_log(1,2,1:iterend), [], 1), N, 1);
    Oz = repmat(reshape(pos0_log(1,3,1:iterend), [], 1), N, 1);
else
    tmp = obstacles';
    Ox = repmat(tmp(1,1), numRows, 1);
    Oy = repmat(tmp(1,2), numRows, 1);
    Oz = repmat(tmp(1,3), numRows, 1);
end

FattX = reshape(Fatt_log(:,1,1:iterend), [], 1);
FattY = reshape(Fatt_log(:,2,1:iterend), [], 1);
FattZ = reshape(Fatt_log(:,3,1:iterend), [], 1);

```

```

FrepX = reshape(Frep_log(:,1,1:iterend), [], 1);
FrepY = reshape(Frep_log(:,2,1:iterend), [], 1);
FrepZ = reshape(Frep_log(:,3,1:iterend), [], 1);

T = table(Iterasi, Agen, ...
          Ax, Ay, Az, ...
          Ox, Oy, Oz, ...
          FattX, FattY, FattZ, ...
          FrepX, FrepY, FrepZ);

writetable(T, 'simulation_log.csv');
save('simulation_log.mat','T');

% Legend handling for trajectory plots
if exist('hTraj', 'var') && ~isempty(hTraj)
    % Collect all trajectory line handles for legend
    all_traj_handles = [];
    traj_labels = {};

    % Get one representative line per target
    for t = 1:M
        % Find an agent with this target
        agents_with_target = find(assigned_goal == t);
        if ~isempty(agents_with_target)
            ag = agents_with_target(1);
            if ~isempty(hTraj{ag})
                % Find a line segment with this target's color
                for seg = 1:length(hTraj{ag})
                    if isvalid(hTraj{ag}(seg))
                        line_color = get(hTraj{ag}(seg), 'Color');
                        if all(abs(line_color - cmap(t,:)) < 0.01)
                            all_traj_handles(end+1) = hTraj{ag}(seg);
                            traj_labels{end+1} = sprintf('Target %d', t);
                            break;
                        end
                    end
                end
            end
        end
    end

    % Add legend based on obstacle type
    if static == 0 && exist('h0bsQuad', 'var') && iscell(h0bsQuad) && ~isempty(h0bsQuad)
        % Dynamic obstacles - get first obstacle handle
        if ~isempty(all_traj_handles) && length(h0bsQuad) >= 1 && isvalid(h0bsQuad{1})
            legend([all_traj_handles, h0bsQuad{1}], [traj_labels, {'Dynamic Obstacle'}], 'Location', 'northeast');
        elseif length(h0bsQuad) >= 1 && isvalid(h0bsQuad{1})
            legend(h0bsQuad{1}, {'Dynamic Obstacle'}, 'Location', 'northeast');
        elseif ~isempty(all_traj_handles)
            legend(all_traj_handles, traj_labels, 'Location', 'northeast');
        end
        elseif ~isempty(all_traj_handles)
            % Static obstacles or no obstacles visible
            legend(all_traj_handles, traj_labels, 'Location', 'northeast');
        end
    end

    for t=1:M
        grp = find(assigned_goal==t);
        fprintf('Agen di Target %d: %s\n', t, mat2str(grp));
    end
end

```

```

end

%% Plotting Section
%% → Hitung panjang lintasan tiap agen (HANYA sampai agen berhenti)
path_lengths = zeros(N,1);

for j = 1:N
    % Tentukan sampai iterasi mana agen ini bergerak
    if done(j) && target_reached_time(j) > 0
        % Agen sudah berhenti - hitung sampai waktu berhenti saja
        stop_iteration = round(target_reached_time(j) / dt);
        end_iter = min([stop_iteration, iterend, size(data_points{j}, 1)]);
        fprintf('Agent %d stopped at iteration %d (time %.2fs)\n', j,
stop_iteration, target_reached_time(j));
    else
        % Agen belum berhenti - hitung sampai akhir simulasi
        end_iter = min([iterend, size(data_points{j}, 1)]);
        fprintf('Agent %d did not stop - calculating until simulation end
(iter %d)\n', j, end_iter);
    end

    if end_iter > 1
        % Ambil posisi dari awal sampai agen berhenti
        pts = data_points{j}(1:end_iter, :); % ukuran end_iterx3

        % Hapus posisi yang invalid (zeros atau NaN)
        valid_rows = ~any(isnan(pts), 2) & ~all(pts == 0, 2);
        pts = pts(valid_rows, :);

        if size(pts, 1) > 1
            deltas = diff(pts, 1, 1); % (end_iter-1)×3
perbedaan posisi
            dists = sqrt(sum(deltas.^2, 2)); % jarak Euclid per
langkah
            path_lengths(j) = sum(dists); % total jarak tempuh
        else
            path_lengths(j) = 0; % Tidak ada gerakan
        end
        else
            path_lengths(j) = 0; % Tidak ada data
        end
    end

    % Tampilkan hasil dengan informasi tambahan
fprintf('\n==== Panjang Lintasan per Agen (Sampai Berhenti) ====\n');
fprintf('%-8s %-15s %-12s %-15s\n', 'Agent', 'Path Length', 'Stop Time',
'Status');
fprintf('%s\n', repmat('-', 1, 60));

for j = 1:N
    if done(j) && target_reached_time(j) > 0
        status = sprintf('STOPPED (%.1fs)', target_reached_time(j));
        stop_time_str = sprintf('.%2f s', target_reached_time(j));
    else
        status = 'NOT STOPPED';
        stop_time_str = 'N/A';
    end

    fprintf('%-8d %-15.2f %-12s %-15s\n', j, path_lengths(j),
stop_time_str, status);
end

```

1. Plot Grafik

```

% Fixed Inter-Agent Distance Plot with Simulation Stop Indicators
figure;
nrow = ceil(sqrt(N));
ncol = ceil(N/nrow);

% Calculate simulation stop time
simulation_stop_time = iterend * dt;

for i = 1:N
    subplot(nrow, ncol, i);
    hold on; grid on;

    % FIX: Properly initialize line_handles as cell array
    line_handles = {};
    line_colors = {};
    other_agents = setdiff(1:N, i);

    for idx = 1:length(other_agents)
        k = other_agents(idx);
        dik = squeeze(distance_agents(i, k, 1:iterend));

        % FIX: Store handle and color properly
        h = plot(Time(1:iterend), dik, 'LineWidth', 3);
        line_handles{idx} = h;
        line_colors{idx} = h.Color; % Store the color
    end

    % Add simulation stop indicator
    ymax = 0;
    for idx = 1:length(other_agents)
        k = other_agents(idx);
        current_max = max(squeeze(distance_agents(i, k, 1:iterend)));
        ymax = max(ymax, current_max);
    end

    if ymax == 0
        ymax = 100; % Default max if no data
    end

    % Vertical line at simulation stop
    plot([simulation_stop_time, simulation_stop_time], [0, ymax], 'r--',
...
        'LineWidth', 2);

    % Add text annotation for simulation stop
    text(simulation_stop_time + 0.5, ymax * 0.9, ...
        sprintf('SIM STOP\n%.1fs', simulation_stop_time), ...
        'FontSize', 9, 'Color', 'red', 'FontWeight', 'bold', ...
        'HorizontalAlignment', 'left');

    % Mark individual agent stopping times
    if done(i) && target_reached_time(i) > 0
        agent_stop_time = target_reached_time(i);

        % Mark agent stop time with different color
        plot([agent_stop_time, agent_stop_time], [0, ymax], 'g:', ...
            'LineWidth', 4);

        % Add agent stop annotation
        text(agent_stop_time, ymax * 0.8, ...
            sprintf('A%d STOP\n%.1fs', i, agent_stop_time), ...
            'FontSize', 9, 'Color', 'green', 'FontWeight', 'bold', ...
            'HorizontalAlignment', 'center');

    % Mark final distances to other agents at stopping time

```

```

stop_iteration = round(agent_stop_time / dt);
if stop_iteration <= iterend
    for idx = 1:length(other_agents)
        k = other_agents(idx);
        final_distance = distance_agents(i, k, stop_iteration);

        % FIX: Use stored color instead of dot indexing
        if idx <= length(line_colors)
            marker_color = line_colors{idx};
        else
            marker_color = [0 0 1]; % Default blue
        end

        % Mark final distance point
        plot(agent_stop_time, final_distance, 'o', ...
            'Color', marker_color, 'MarkerSize', 9, ...
            'MarkerFaceColor', marker_color, ...
            'MarkerEdgeColor', 'white');

        % Add distance value annotation (only for very close
agents)
        if final_distance < 10
            text(agent_stop_time + 1, final_distance + 1, ...
                sprintf('%1f', final_distance), ...
                'FontSize', 9, 'Color', marker_color);
        end
    end
end

% Set plot properties
title(sprintf('Agen %d', i));
% ADD LABELS TO ALL SUBPLOTS
xlabel('Waktu (s)', 'FontSize', 12);
ylabel('Jarak (meter)', 'FontSize', 12); % Changed from "unit" to
"meter"

% FIX: Enhanced legend with proper handle access
legend_labels = {};
legend_handles_array = [];

for idx = 1:length(other_agents)
    k = other_agents(idx);
    final_dist = distance_agents(i, k, iterend);

    if done(i) && target_reached_time(i) > 0
        stop_iter = round(target_reached_time(i) / dt);
        if stop_iter <= iterend && stop_iter > 0
            stop_dist = distance_agents(i, k, stop_iter);
            legend_labels{idx} = sprintf('%d-%d (final: %.1f)', i, k,
stop_dist);
        else
            legend_labels{idx} = sprintf('%d-%d (final: %.1f)', i, k,
final_dist);
        end
    else
        legend_labels{idx} = sprintf('%d-%d (final: %.1f)', i, k,
final_dist);
    end

    % Store handle for legend
    if idx <= length(line_handles)
        legend_handles_array{idx} = line_handles{idx};
    end

```

```

    end

    % Add legend with smaller font
    if ~isempty(legend_labels) && ~isempty(legend_handles_array)
        legend(legend_handles_array, legend_labels, 'Location', 'best',
        'FontSize', 9);
    end

    % Set axis limits
    xlim([0, max(Time(iterend), simulation_stop_time) * 1.1]);
    ylim([0, ymax * 1.05]);
end

% Overall title with simulation info
sgtitle(sprintf('Jarak Inter-Agen vs Waktu | Simulasi Berhenti: %.1fs | '
    'Algoritma: %s', ...
    simulation_stop_time, apf_names{APF_type}));

%% Additional summary information (FIXED)
figure;
hold on; grid on;

% Plot showing when each agent stopped
if exist('cmap', 'var') && size(cmap, 1) >= N
    agent_colors = cmap(1:N, :);
else
    agent_colors = lines(N);
end

for j = 1:N
    if done(j) && target_reached_time(j) > 0
        % Bar showing agent stop time
        bar(j, target_reached_time(j), 'FaceColor', agent_colors(j,:), ...
            'EdgeColor', 'black', 'LineWidth', 1);

        % Add text annotation
        text(j, target_reached_time(j) + 1, ...
            sprintf('%.1fs', target_reached_time(j)), ...
            'HorizontalAlignment', 'center', 'FontWeight', 'bold');
    else
        % Agent didn't stop - show simulation end time
        bar(j, simulation_stop_time, 'FaceColor', [0.7 0.7 0.7], ...
            'EdgeColor', 'black', 'LineWidth', 1);

        text(j, simulation_stop_time + 1, 'N/A', ...
            'HorizontalAlignment', 'center', 'FontWeight', 'bold');
    end
end

% Add simulation stop line
plot([0.5, N+0.5], [simulation_stop_time, simulation_stop_time], 'r--', ...
    'LineWidth', 2);

text(N/2, simulation_stop_time + 2, ...
    sprintf('Simulation Stop: %.1fs', simulation_stop_time), ...
    'HorizontalAlignment', 'center', 'Color', 'red', 'FontWeight', 'bold');

xlabel('Agent');
ylabel('Waktu Berhenti (s)');
title('Waktu Berhenti per Agen');
xlim([0.5, N+0.5]);
ylim([0, simulation_stop_time * 1.2]);

% Add agent labels
xticks(1:N);
xticklabels(arrayfun(@(j) sprintf('A%d', j), 1:N, 'UniformOutput', false));

```

```

%% Final distance matrix at simulation end (FIXED)
figure;

% Create distance matrix at simulation end
final_distances = zeros(N, N);
for i = 1:N
    for j = 1:N
        if i ~= j
            final_distances(i, j) = distance_agents(i, j, iterend);
        end
    end
end

% Plot as heatmap
imagesc(final_distances);
colorbar;
colormap('hot');

% Add text annotations
for i = 1:N
    for j = 1:N
        if i ~= j
            text(j, i, sprintf('%1f', final_distances(i, j)), ...
                  'HorizontalAlignment', 'center', 'FontWeight', 'bold', ...
                  'Color', 'white');
        end
    end
end

title(sprintf('Jarak Akhir Antar-Agen (t=%.1fs)', simulation_stop_time));
xlabel('Agent');
ylabel('Agent');

% Set axis labels
xticks(1:N);
yticks(1:N);
xticklabels(arrayfun(@(j) sprintf('A%d', j), 1:N, 'UniformOutput', false));
yticklabels(arrayfun(@(j) sprintf('A%d', j), 1:N, 'UniformOutput', false));

%% Final distance matrix at simulation end
figure;

% Create distance matrix at simulation end
final_distances = zeros(N, N);
for i = 1:N
    for j = 1:N
        if i ~= j
            final_distances(i, j) = distance_agents(i, j, iterend);
        end
    end
end

% Plot as heatmap
imagesc(final_distances);
colorbar;
colormap('hot');

% Add text annotations
for i = 1:N
    for j = 1:N
        if i ~= j
            text(j, i, sprintf('%1f', final_distances(i, j)), ...
                  'HorizontalAlignment', 'center', 'FontWeight', 'bold', ...
                  'Color', 'white');
        end
    end
end

```

```

    end
end

title(sprintf('Jarak Akhir Antar-Agen (t=%.1fs)', simulation_stop_time));
xlabel('Agent');
ylabel('Agent');

% Set axis labels
xticks(1:N);
yticks(1:N);
xticklabels(arrayfun(@(j) sprintf('A%d', j), 1:N, 'UniformOutput', false));
yticklabels(arrayfun(@(j) sprintf('A%d', j), 1:N, 'UniformOutput', false));

%%

% Plot tiap agen berdasarkan assigned_goal mereka:
if ~exist('cmap', 'var')
    cmap = lines(M);
end

% Only plot simple trajectories if we don't have colored segments
% (This is a fallback in case trajectory recording was disabled)
if isempty(hTraj{1})
    for t = 1:M
        agents_on_t = find(assigned_goal == t);
        for ag = agents_on_t
            plot3( data_points{ag}(1:iterend-1,1), ...
                    data_points{ag}(1:iterend-1,2), ...
                    data_points{ag}(1:iterend-1,3), ...
                    'Color', cmap(t,:), 'LineWidth', 1.5 );
        end
    end
    legend(arrayfun(@(t) sprintf('Target %d',t), 1:M,'Uni',false));
end

num_steps_valid = floor(iterend / alocfreq) + 1;

% Limit num_steps_valid to actual array sizes
max_available_steps = size(assigned_history, 2);
max_time_steps = length(time_steps);
num_steps_valid = min(num_steps_valid, max_available_steps);
num_steps_valid = min(num_steps_valid, max_time_steps);

% Now safely trim arrays
if num_steps_valid > 0 && num_steps_valid <= size(assigned_history, 2)
    assigned_history = assigned_history(:, 1:num_steps_valid);
end

if num_steps_valid > 0 && num_steps_valid <= size(theta_history, 2)
    theta_history = theta_history(:, 1:num_steps_valid);
end

if num_steps_valid > 0 && num_steps_valid <= length(time_steps)
    time_steps = time_steps(1:num_steps_valid);
end

%% FIX: Maintain Constant Distance to Obstacles After Agent Stops
% Replace your obstacle distance calculation section with this:
%% Calculate distance from each agent to ALL obstacles separately
%% → Kode Pengecekan Jarak Semua Obstacle ke Agen vs Waktu (FIXED)
% Calculate distance from each agent to ALL obstacles separately
% Initialize distance matrix: [N agents × M_obs obstacles × iterations]

```

```

%% ✓ FIXED: Distance Calculation for All Obstacles
% Calculate distance from each agent to ALL obstacles separately
% Initialize distance matrix: [N agents x M_obs obstacles x iterations]
if ~exist('distance_obs_all', 'var')
    distance_obs_all = zeros(N, M_obs, iterend);
end

% Calculate distances for all agents and all obstacles
for j = 1:N
    for k = 1:M_obs
        % Only calculate new distances for moving agents
        if ~done(j) || (j == 1 && k == 1 && i == 1) % Always calculate for
first iteration
            if k <= size(obstacles, 2)
                obs_pos = obstacles(:, k);
                obs_radius = radius(k);

                if static == 1
                    % Static cylinder
                    dx = current_pos{j}(1:2) - obs_pos(1:2);
                    dist_2d = norm(dx);

                    if current_pos{j}(3) < obs_pos(3)
                        % Below obstacle height
                        dist_to_obs = max(0.1, dist_2d - obs_radius);
                    else
                        % Above obstacle
                        dist_to_obs = norm([dx; current_pos{j}(3) -
obs_pos(3)]) - obs_radius;
                    end
                else
                    % Dynamic sphere
                    if exist('obstacle_types', 'var') && mixed == 1 &&
obstacle_types(k) == 1
                        % Static cylinder in mixed mode
                        dx = current_pos{j}(1:2) - obs_pos(1:2);
                        dist_2d = norm(dx);
                        if current_pos{j}(3) < obs_pos(3)
                            dist_to_obs = dist_2d - obs_radius;
                        else
                            dist_to_obs = norm([dx; current_pos{j}(3) -
obs_pos(3)]) - obs_radius;
                        end
                    else
                        % Dynamic sphere
                        dist_to_obs = norm(current_pos{j} - obs_pos) -
0.25;
                    end
                end
            end
            distance_obs_all(j, k, i) = max(0.1, dist_to_obs);
        end
    else
        % ✓ Agent has stopped - freeze distance at stopping value
        % ✓ FIXED: Agent has stopped - freeze distance properly
        if target_reached_time(j) > 0
            actual_stop_iter = round(target_reached_time(j) / dt);
            actual_stop_iter = max(1, min(actual_stop_iter, iterend));

            % Check if we have valid data at stopping iteration
            if actual_stop_iter > 0 && actual_stop_iter <= size(distance_obs_all,
3)
                frozen_dist = distance_obs_all(j, k, actual_stop_iter);

                % Only use frozen distance if it's valid (> 0.01)
            end
        end
    end
end

```

```

if frozen_dist > 0.01 && isnan(frozen_dist)
    distance_obs_all(j, k, i) = frozen_dist;
else
    % Recalculate if frozen distance is invalid
    obs_pos = obstacles(:, k);
    obs_radius = radius(k);

    if static == 1
        dx = current_pos{j}(1:2) - obs_pos(1:2);
        dist_2d = norm(dx);
        if current_pos{j}(3) < obs_pos(3)
            new_dist = max(0.1, dist_2d - obs_radius);
        else
            new_dist = norm([dx; current_pos{j}(3) - obs_pos(3)]) -
obs_radius;
        end
    else
        new_dist = norm(current_pos{j} - obs_pos) - obs_radius;
    end

    distance_obs_all(j, k, i) = max(0.1, new_dist);

    % Debug output
    fprintf('Recalculated distance for stopped Agent %d, Obstacle
%d: %.3f\n', j, k, new_dist);
end
else
    % Fallback: recalculate if can't access stopping iteration
    obs_pos = obstacles(:, k);
    obs_radius = radius(k);

    if static == 1
        dx = current_pos{j}(1:2) - obs_pos(1:2);
        dist_2d = norm(dx);
        if current_pos{j}(3) < obs_pos(3)
            new_dist = max(0.1, dist_2d - obs_radius);
        else
            new_dist = norm([dx; current_pos{j}(3) - obs_pos(3)]) -
obs_radius;
        end
    else
        new_dist = norm(current_pos{j} - obs_pos) - obs_radius;
    end

    distance_obs_all(j, k, i) = max(0.1, new_dist);
end
end
end
end

%% Plot untuk setiap agen: jarak ke semua obstacle vs waktu
%%  FIXED: Plot untuk setiap agen dengan legend yang benar
obstacle_colors = lines(M_obs); % Different color for each obstacle

for j = 1:N
    figure;
    hold on; grid on;

    %  Store line handles properly for correct legend
    h_lines = [];
    obstacle_labels = {};

    for k = 1:M_obs
        distances_to_k = squeeze(distance_obs_all(j, k, 1:iterend));

```

```

% Plot line for this obstacle and STORE the handle
h_k = plot(Time(1:iterend), distances_to_k, '-', ...
    'Color', obstacle_colors(k, :), 'LineWidth', 2.5);
h_lines(end+1) = h_k; % ✓ Store actual handle

% Create label for this obstacle
if static == 0
    obstacle_labels{end+1} = sprintf('Obs Dinamis %d', k);
else
    obstacle_labels{end+1} = sprintf('Obs Statis %d', k);
end
end

% Mark stopping point if agent stopped
if done(j) && target_reached_time(j) > 0
    stop_time = target_reached_time(j);
    stop_iter = round(stop_time / dt);

    if stop_iter > 0 && stop_iter <= iterend
        % Calculate ymax from all obstacles
        ymax = 0;
        for k = 1:M_obs
            ymax = max(ymax, max(squeeze(distance_obs_all(j, k,
1:iterend)))); % Add vertical line at stopping time
        end
        plot([stop_time, stop_time], [0, ymax], 'r--', 'LineWidth', 2);

        % Mark stopping points for each obstacle
        for k = 1:M_obs
            stop_distance_k = distance_obs_all(j, k, stop_iter);

            % Mark stopping point for this obstacle
            plot(stop_time, stop_distance_k, 'o',...
                'Color', obstacle_colors(k, :), 'MarkerSize', 8, ...
                'MarkerFaceColor', obstacle_colors(k, :), ...
                'MarkerEdgeColor', 'white', 'LineWidth', 1.5);

            % ✓ Highlight the flat section after stopping for each
            obstacle
            if stop_iter < iterend
                flat_section_time = Time(stop_iter:iterend);
                flat_section_dist = squeeze(distance_obs_all(j, k,
stop_iter:iterend));

                % Plot flat section with thicker line
                plot(flat_section_time, flat_section_dist, '-',...
                    'Color', obstacle_colors(k, :), 'LineWidth', 4);
            end
        end
    end
end

% Add distance value annotation for close obstacles
if stop_distance_k < 50 % Only annotate if relatively
close
    text(stop_time + 1, stop_distance_k + 2, ...
        sprintf('.1f', stop_distance_k), ...
        'FontSize', 8, 'Color', obstacle_colors(k, :), ...
        'FontWeight', 'bold');
end
end
end

```

```

% Enhanced plot formatting
xlabel('Waktu (s)', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('Jarak Agen → Obstakel (unit)', 'FontSize', 12, 'FontWeight',
'bold');
title(sprintf('Jarak Agen %d ke Setiap Obstakel vs Waktu', j), ...
'FontSize', 14, 'FontWeight', 'bold');

% Add grid and better formatting
grid on;
set(gca, 'GridAlpha', 0.3, 'FontSize', 11);

% Enhanced legend with obstacle information
if ~isempty(h_lines)
    legend_entries = obstacle_labels;

    % Add final distances to legend
    for k = 1:M_obs
        final_dist = distance_obs_all(j, k, iterend);
        legend_entries{k} = sprintf('%s ', obstacle_labels{k});
    end

    legend(h_lines, legend_entries, 'Location', 'best', 'FontSize',
10);
end
F
% Set y-axis to start from 0 for better visualization
all_distances = squeeze(distance_obs_all(j, :, 1:iterend));
ymax = max(all_distances(:));
ylim([0, ymax * 1.1]);
xlim([0, Time(iterend) * 1.05]);
end
%% Summary Plot: All Agents vs Closest Obstacle
figure;
hold on; grid on;

agent_colors = lines(N);
for j = 1:N
    % Find minimum distance to any obstacle at each time step
    min_distances = zeros(1, iterend);
    for i = 1:iterend
        min_distances(i) = min(squeeze(distance_obs_all(j, :, i)));
    end

    % Plot minimum distance line
    h_agent = plot(Time(1:iterend), min_distances, '-',
    'Color', agent_colors(j, :), 'LineWidth', 2);

    % Mark stopping point
    if done(j) && target_reached_time(j) > 0
        stop_time = target_reached_time(j);
        stop_iter = round(stop_time / dt);

        if stop_iter > 0 && stop_iter <= iterend
            stop_min_distance = min_distances(stop_iter);

            plot(stop_time, stop_min_distance, 'o', ...
            'Color', agent_colors(j, :), 'MarkerSize', 8, ...
            'MarkerFaceColor', agent_colors(j, :), ...
            'MarkerEdgeColor', 'white', 'LineWidth', 1.5);
        end
    end
end

xlabel('Waktu (s)', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('Jarak Minimum ke Obstakel (unit)', 'FontSize', 12, 'FontWeight',
'bold');

```

```

title('Jarak Minimum Setiap Agen ke Obstakel Terdekat', 'FontSize', 14,
'FontWeight', 'bold');
grid on;

% ✓ FIXED: Create proper legend with handles and status
summary_handles = [];
summary_labels = {};

% Collect handles and create labels with stop status
for j = 1:N
    if done(j) && target_reached_time(j) > 0
        stop_time = target_reached_time(j);
        stop_iter = round(stop_time / dt);

        if stop_iter > 0 && stop_iter <= iterend
            stop_min_distance = min_distances(stop_iter);
            summary_labels{j} = sprintf('Agen %d (stop: %.1f)', j,
stop_min_distance);
        else
            summary_labels{j} = sprintf('Agen %d (berhenti)', j);
        end
    else
        final_min_distance = min_distances(iterend);
        summary_labels{j} = sprintf('Agen %d (final: %.1f)', j,
final_min_distance);
    end
end

% Note: You need to store handles when creating the plot lines above
% Add this after each plot() command: summary_handles(end+1) = h_agent;
legend(summary_handles, summary_labels, 'Location', 'best', 'FontSize',
10);

%% Verification: Check that distances stay constant after stopping
fprintf('\n==> VERIFICATION: All Distances Constant After Stopping ==>\n');
for j = 1:N
    if done(j) && target_reached_time(j) > 0
        stop_iter = round(target_reached_time(j) / dt);

        if stop_iter > 0 && stop_iter < iterend
            fprintf('Agent %d stopped at iteration %d:\n', j, stop_iter);

            for k = 1:M_obs
                % Check if distance to obstacle k remains constant
                check_end = min(iterend, stop_iter + 10);
                distances_after = squeeze(distance_obs_all(j, k,
stop_iter:check_end));

                is_constant = all(abs(distances_after - distances_after(1)) < 0.001);
                fprintf('  Obstacle %d: Distance = %.3f, Constant = %s\n',
...
k, distances_after(1), char(is_constant + "false"));

                if ~is_constant
                    fprintf('  ⚠ Range: %.3f to %.3f\n',
min(distances_after), max(distances_after));
                end
            end
        end
    end
end

```

```

%% Agent Speed Plot
figure;
subplot(2,1,1);
hold on; grid on;
for jj = 1:N
    plot(Time(1:iterend), agent_speeds(jj,1:iterend), 'LineWidth',1.5);
end
xlabel('Waktu (s)');
ylabel('Kecepatan (m/s)');
title('Kecepatan Agen vs Waktu');
legend(arrayfun(@(j) sprintf('Agen %d',j), 1:N, 'UniformOutput',false),
...
'Location','northeastoutside');
ylim([0, max(max(agent_speeds(:,1:iterend)))*1.1]);

% Obstacle Speed Plot (if dynamic)
if static == 0 && ~isempty(dynamic_obs_indices)
    subplot(2,1,2);
    hold on; grid on;
    for k = 1:length(dynamic_obs_indices)
        plot(Time(1:iterend), obstacle_speeds(k,1:iterend), 'r-',
'LineWidth',2);
    end
    xlabel('Waktu (s)');
    ylabel('Kecepatan (m/s)');
    title(sprintf('Kecepatan Obstacle Dinamis (Target: %.1f m/s)', obstacle_speed));
    if mixed == 1
        legend(arrayfun(@(k) sprintf('Dynamic Obs %d',k),
1:length(dynamic_obs_indices), 'UniformOutput',false), ...
'Location','northeastoutside');
    else
        legend(arrayfun(@(k) sprintf('Obstacle %d',k),
1:length(dynamic_obs_indices), 'UniformOutput',false), ...
'Location','northeastoutside');
    end
    ylim([0, obstacle_speed*1.5]);
    % Add reference line for target speed
    plot([0, Time(iterend)], [obstacle_speed, obstacle_speed], 'k--',
'LineWidth',1);
end
%%
% Target assignment history
figure;
hold on; grid on;
for j = 1:N
    stairs(time_steps, assigned_history(j,:), 'LineWidth', 2);
end
xlabel('Waktu (s)');
ylabel('Target yang Dituju');
title('Riwayat Penugasan Target per Agen');
legend(arrayfun(@(j) sprintf('Agen %d', j), 1:N, 'UniformOutput', false));
yticks(1:M);
ylim([0.9, M + 0.1]);

%% FINAL FIX: Distance to Target Calculation
actual_iterations = min(iterend, iteration);
dist_to_target = zeros(N, actual_iterations);

% Create agent target history
agent_target_history = zeros(N, actual_iterations);

```

```

% Fill the target history based on allocation frequency
for i = 1:actual_iterations
    if exist('assigned_history', 'var') && size(assigned_history, 2) > 0
        alloc_step = min(floor((i-1)/alocfreq) + 1, size(assigned_history, 2));
    else
        alloc_step = max(1, alloc_step);
    end
    for j = 1:N
        agent_target_history(j, i) = assigned_history(j, alloc_step);
    end
end

% Calculate distances using the correct position data
for j = 1:N
    for i = 1:actual_iterations
        % FIX: Use the correct position source
        if done(j) && target_reached_time(j) > 0
            stop_iteration = round(target_reached_time(j) / dt);

            if i >= stop_iteration
                % For stopped agents AFTER stopping: use
                final_stopping_pos
                if exist('final_stopping_pos', 'var') &&
                    ~isempty(final_stopping_pos{j})
                    pos = final_stopping_pos{j}'; % Convert to row vector
[1x3]
                else
                    % Fallback to data_points if final_stopping_pos not
                    available
                    if i <= size(data_points{j}, 1) &&
                        ~isempty(data_points{j}(i,:)) && any(data_points{j}(i,:) ~= 0)
                        pos = data_points{j}(i, :);
                    else
                        % Use last valid position
                        if i > 1
                            pos = final_stopping_pos{j}';
                        else
                            pos = start{j}';
                        end
                    end
                end
            else
                % For agents BEFORE stopping: use data_points
                if i <= size(data_points{j}, 1) &&
                    ~isempty(data_points{j}(i,:)) && any(data_points{j}(i,:) ~= 0)
                    pos = data_points{j}(i, :);
                else
                    % Use last valid position or interpolate
                    if i > 1 && i-1 <= size(dist_to_target, 2)
                        % Keep previous distance
                        dist_to_target(j, i) = dist_to_target(j, i-1);
                        continue;
                    else
                        pos = start{j}';
                    end
                end
            end
        else
            % For agents that haven't stopped: use data_points
            if i <= size(data_points{j}, 1) &&
                ~isempty(data_points{j}(i,:)) && any(data_points{j}(i,:) ~= 0)

```

```

        pos = data_points{j}(i, :);
    else
        % Use last valid distance or start position
        if i > 1 && i-1 <= size(dist_to_target, 2)
            dist_to_target(j, i) = dist_to_target(j, i-1);
            continue;
        else
            pos = start{j}';
        end
    end
end

% Get target assignment for this iteration
current_target_id = agent_target_history(j, i);

% Validate target ID
if current_target_id < 1 || current_target_id > length(target)
    current_target_id = 1;
end

%  Calculate distance with correct dimensions
tgt = target{current_target_id}'; % Convert to row vector [1x3]
calculated_distance = norm(pos - tgt); % [1x3] - [1x3] = correct!

dist_to_target(j, i) = calculated_distance;
end
end

%% FIXED DISTANCE TO TARGET PLOT WITH PROPER LEGEND
figure;
hold on; grid on;

% Store line handles for proper legend
line_handles = [];
legend_labels = {};

for j = 1:N
    line_handle = plot(Time(1:actual_iterations), dist_to_target(j, :),
'LineWidth', 4);

    %  STORE THE ACTUAL LINE HANDLE
    line_handles(end+1) = line_handle;

    % Mark the actual stopping point
    if done(j) && target_reached_time(j) > 0
        time_reached = target_reached_time(j);
        if time_reached <= Time(actual_iterations)
            time_reached_iter = round(time_reached / dt);
            if time_reached_iter <= actual_iterations && time_reached_iter
> 0
                actual_stopping_distance = dist_to_target(j,
time_reached_iter);

                % Mark stopping point
                plot(time_reached, actual_stopping_distance, 'o', ...
                    'Color', line_handle.Color, 'MarkerSize', 10, ...
                    'MarkerFaceColor', line_handle.Color,
                    'MarkerEdgeColor', 'white');

                % Add text annotation
                text(time_reached + (Time(actual_iterations) * 0.02),
actual_stopping_distance + 5, ...
                    sprintf('STOP\n%.2f', actual_stopping_distance), ...
                    'FontSize', 8, 'Color', line_handle.Color,
                    'FontWeight', 'bold');
            end
        end
    end
end

```

```

        % Add vertical line to show stopping time
        ymax = max(dist_to_target(j, :));
        plot([time_reached, time_reached], [0, ymax], ':', ...
              'Color', line_handle.Color, 'LineWidth', 3);
    end
end
end

% Add reference line for R_STOP threshold
threshold_handle = plot([0, Time(actual_iterations)], [params.R_STOP,
params.R_STOP], 'r--', 'LineWidth', 2);

%  CREATE PROPER LEGEND WITH LINE HANDLES AND CLEAN LABELS
for j = 1:N
    if done(j) && target_reached_time(j) > 0
        stop_iter = round(target_reached_time(j) / dt);
        if stop_iter <= actual_iterations && stop_iter > 0
            actual_distance = dist_to_target(j, stop_iter);
            stop_target = agent_target_history(j, stop_iter);
            %  CLEAN LEGEND FORMAT
            legend_labels{j} = sprintf('Agen %d → T%d (STOP: %.2f)', j,
stop_target, actual_distance);
        else
            legend_labels{j} = sprintf('Agen %d → T%d (ERROR)', j,
assigned_goal(j));
        end
    else
        legend_labels{j} = sprintf('Agen %d → T%d (MOVING)', j,
assigned_goal(j));
    end
end

% Add threshold to legend
line_handles(end+1) = threshold_handle;
legend_labels{end+1} = sprintf('R_{STOP} threshold = %.1f meter',
params.R_STOP);

%  CREATE LEGEND WITH ACTUAL LINE HANDLES
legend(line_handles, legend_labels, 'Location', 'northeast', 'FontSize',
10);

xlabel('Waktu (s)', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('Jarak ke Target (meter)', 'FontSize', 12, 'FontWeight', 'bold');
title('Jarak Agen ke Target vs Waktu', 'FontSize', 14, 'FontWeight',
'bold');
grid on;

%%
%% PLOT DISTANCE FROM EACH AGENT TO ALL TARGETS
% This code works with your actual variable structure

% Your variables:
% - data_points{j}(i,:) = position history of agent j at iteration i
% - target{t} = position of target t
% - assigned_goal(j) = which target is assigned to agent j
% - actual_iterations = number of iterations that actually ran

figure;
hold on; grid on;

% Store line handles for legend
line_handles = [];
legend_labels = {};

% Define colors and line styles

```

```

target_colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k'];
line_styles = {'-', '--', ':', '-.'};

% For each agent
for j = 1:N
    % For each target
    for target_idx = 1:M
        % Calculate distance from agent j to target target_idx at each
        % iteration
        distances_to_target = zeros(1, actual_iterations);

        for iter = 1:actual_iterations
            % Get agent position at this iteration
            agent_pos = data_points{j}(iter, :); % This is your position
            history

            % Get target position
            target_pos = target{target_idx}; % This is your target array

            % Calculate distance
            distances_to_target(iter) = norm(agent_pos - target_pos');
        end

        % Choose color and line style
        color_idx = mod(target_idx - 1, length(target_colors)) + 1;
        style_idx = mod(j - 1, length(line_styles)) + 1;

        % Plot the line
        if target_idx == assigned_goal(j)
            % This is the assigned target - use thick solid line
            line_handle = plot(Time(1:actual_iterations),
distances_to_target, ...
                    'Color', target_colors(color_idx), ...
                    'LineStyle', '-', ...
                    'LineWidth', 4);
            legend_text = sprintf('Agent %d → T%d (ASSIGNED)', j,
target_idx);
        else
            % This is not the assigned target - use thin dashed line
            line_handle = plot(Time(1:actual_iterations),
distances_to_target, ...
                    'Color', target_colors(color_idx), ...
                    'LineStyle', line_styles{style_idx}, ...
                    'LineWidth', 2);
            legend_text = sprintf('Agent %d → T%d', j, target_idx);
        end

        % Store handle for legend
        line_handles(end+1) = line_handle;
        legend_labels{end+1} = legend_text;

        % Mark stopping point if this is the assigned target
        if target_idx == assigned_goal(j) && done(j) &&
target_reached_time(j) > 0
            time_reached = target_reached_time(j);
            if time_reached <= Time(actual_iterations)
                time_reached_iter = round(time_reached / dt);
                if time_reached_iter <= actual_iterations &&
time_reached_iter > 0
                    actual_stopping_distance =
distances_to_target(time_reached_iter);

                    % Mark stopping point
                    plot(time_reached, actual_stopping_distance, 'o',...
                        'Color', target_colors(color_idx), 'MarkerSize',
10, ...

```

```

        'MarkerFaceColor', target_colors(color_idx),
'MarkerEdgeColor', 'white');

        % Add text annotation
        text(time_reached + (Time(actual_iterations) * 0.02),
actual_stopping_distance + 5, ...
sprintf('STOP\n%.2f', actual_stopping_distance),
...
'FontSize', 8, 'Color', target_colors(color_idx),
'FontWeight', 'bold');
    end
end
end

% Add reference line for R_STOP threshold
threshold_handle = plot([0, Time(actual_iterations)], [params.R_STOP,
params.R_STOP], 'k--', 'LineWidth', 2);
line_handles(end+1) = threshold_handle;
legend_labels{end+1} = sprintf('R_STOP threshold = %.1f meter',
params.R_STOP);

% Create legend
legend(line_handles, legend_labels, 'Location', 'northeast', 'FontSize',
8);
xlabel('Waktu (s)', 'FontSize', 12, 'FontWeight', 'bold');
ylabel('Jarak ke Target (meter)', 'FontSize', 12, 'FontWeight', 'bold');
title('Jarak Semua Agen ke Semua Target vs Waktu', 'FontSize', 14,
'FontWeight', 'bold');
grid on;
%%
%% SEPARATE SUBPLOT FOR EACH AGENT
figure;

for j = 1:N
    subplot(ceil(sqrt(N)), ceil(sqrt(N)), j);
    hold on; grid on;

    line_handles = [];
    legend_labels = {};

    % Plot distance to each target
    for target_idx = 1:M
        % Calculate distance from agent j to target target_idx
        distances_to_target = zeros(1, actual_iterations);

        for iter = 1:actual_iterations
            % Get agent position at this iteration
            agent_pos = data_points{j}(iter, :); % Your position history

            % Get target position
            target_pos = target{target_idx}; % Your target array

            % Calculate distance
            distances_to_target(iter) = norm(agent_pos - target_pos');
        end

        % Different style for assigned vs non-assigned targets
        if target_idx == assigned_goal(j)
            line_handle = plot(Time(1:actual_iterations),
distances_to_target, ...
'LineWidth', 4, 'Color', 'r');
            legend_labels{end+1} = sprintf('T%d (ASSIGNED)', target_idx);
        else

```

```

        line_handle = plot(Time(1:actual_iterations),
distances_to_target, ...
    'LineWidth', 2, 'Color', [0.5, 0.5, 0.5]);
    legend_labels{end+1} = sprintf('T%d', target_idx);
end

line_handles(end+1) = line_handle;

% Mark stopping point for assigned target
if target_idx == assigned_goal(j) && done(j) &&
target_reached_time(j) > 0
    time_reached = target_reached_time(j);
    if time_reached <= Time(actual_iterations)
        time_reached_iter = round(time_reached / dt);
        if time_reached_iter <= actual_iterations &&
time_reached_iter > 0
            actual_stopping_distance =
distances_to_target(time_reached_iter);
            plot(time_reached, actual_stopping_distance, 'ro', ...
'MarkerSize', 8, 'MarkerFaceColor', 'r',
'MarkerEdgeColor', 'white');
        end
    end
end
end

% Add threshold line
threshold_handle = plot([0, Time(actual_iterations)], [params.R_STOP,
params.R_STOP], 'k--', 'LineWidth', 1);
line_handles(end+1) = threshold_handle;
legend_labels{end+1} = sprintf('R_{STOP} = %.1f', params.R_STOP);

% Formatting
title(sprintf('Agent %d', j), 'FontSize', 12, 'FontWeight', 'bold');
xlabel('Waktu (s)', 'FontSize', 10);
ylabel('Jarak (m)', 'FontSize', 10);
legend(line_handles, legend_labels, 'Location', 'best', 'FontSize', 8);
grid on;
end

sgtitle('Jarak Setiap Agen ke Semua Target', 'FontSize', 14, 'FontWeight',
'bold');

%% Verification
fprintf('\n== DISTANCE CALCULATION VERIFICATION ==\n');
for j = 1:N
    if done(j) && target_reached_time(j) > 0
        stop_iter = round(target_reached_time(j) / dt);
        if stop_iter <= actual_iterations && stop_iter > 0
            calculated_distance = dist_to_target(j, stop_iter);
            target_at_stop = agent_target_history(j, stop_iter);

            % Manual verification
            if exist('final_stopping_pos', 'var') &&
~isempty(final_stopping_pos{j})
                manual_distance = norm(final_stopping_pos{j} -
target{target_at_stop});

                fprintf('Agent %d:\n', j);
                fprintf(' Calculated distance: %.3f units\n',
calculated_distance);
                fprintf(' Manual verification: %.3f units\n',
manual_distance);
                fprintf(' Match: %s\n', char((abs(calculated_distance -
manual_distance) < 0.01) + "false"));
                fprintf(' Status: %s\n', char((calculated_distance <=
params.R_STOP) + "OVER" + "\u2225 PASS"));
            end
        end
    end
end

```

```

        fprintf('\n');
    end
end
end

% LIntasan Lebih Tebal
% Alternative Timeline Visualization (using thick lines instead of
rectangles)
figure;
hold on; grid on;
for j = 1:N
    assignment_changes = find(diff([0, assigned_history(j,:)]) ~= 0);

    y_pos = j;
    for k = 1:length(assignment_changes)
        start_time = time_steps(assignment_changes(k));
        if k < length(assignment_changes)
            end_time = time_steps(assignment_changes(k+1));
        else
            end_time = Time(iterend);
        end

        target_id = assigned_history(j, assignment_changes(k));
        if target_id > 0 && target_id <= size(cmap,1)
            col = cmap(target_id,:);
        else
            col = [0.5 0.5 0.5];
        end

        % Use thick line instead of rectangle
        if start_time < end_time
            plot([start_time, end_time], [y_pos, y_pos], 'Color', col,
'LineWidth', 8);
        end
    end
end

% Add target labels and formatting
for t = 1:M
    text(Time(iterend)*1.02, 0.5-t*0.3, sprintf('Target %d', t), ...
'Color', cmap(t,:), 'FontWeight', 'bold', 'FontSize', 10);
end

xlabel('Time (s)');
ylabel('Agent');
title('Agent Target Assignment Timeline (Color = Assigned Target)');
yticks(1:N);
yticklabels(arrayfun(@(j) sprintf('Agent %d', j), 1:N, 'UniformOutput',
false));
ylim([0.5, N+0.5]);
xlim([0, Time(iterend)*1.05]);

    %% COMPLETE TRAJECTORY PLOT - REVISED VERSION (PLACE AROUND LINE 1800-
1900)
figure;
clf; % Clear figure completely to avoid artifacts
hold on; grid on;

% Plot obstacles based on mode
if static == 1
    % Static mode - all obstacles are cylinders
    for i = 1:M_obs
        [h1,~,~] = create_cylinder(radius(i), Cpos(i,:), [0.45, 0.58,
0.76]);
    end
end

```

```

    end
else
% Dynamic or mixed mode
if mixed == 1
    % Plot static obstacles as cylinders
    for idx = 1:length(static_obs_indices)
        i = static_obs_indices(idx);
        [h1,~,~] = create_cylinder(radius(i), Cpos(idx,:), [0.45, 0.58,
0.76]);
    end
end

% Plot targets as POINTS ONLY (no cylinders)
% FIXED TARGET PLOTTING - Replace your existing target section
for t = 1:M
    % Large circular marker for target
    plot3(target{t}(1), target{t}(2), target{t}(3), 'o', ...
        'MarkerSize', 20, 'MarkerFaceColor', colors(t,:), ...
        'MarkerEdgeColor', 'black', 'LineWidth', 3);

    % X marker for emphasis
    plot3(target{t}(1), target{t}(2), target{t}(3), 'kx', ...
        'MarkerSize', 15, 'LineWidth', 3);

    % Target label
    text(target{t}(1)+5, target{t}(2)+5, target{t}(3)+5, ...
        sprintf('T%d', t), 'FontSize', 12, 'FontWeight', 'bold', ...
        'Color', colors(t,:), 'BackgroundColor', 'white', ...
        'EdgeColor', 'black');
end
% Remove the target_handles array completely - it's not needed

% Plot agent trajectories with proper color coding
trajectory_handles = [];
trajectory_labels = {};
used_targets = [];

for j = 1:N
    % Get trajectory data up to actual end iteration
    actual_end = min(iterend, size(data_points{j}), 1));
    traj_data = data_points{j}(1:actual_end, :);

    % Remove any NaN or invalid data points
    valid_idx = ~any(isnan(traj_data), 2) & ~any(isinf(traj_data), 2);
    traj_data = traj_data(valid_idx, :);

    if size(traj_data, 1) < 2
        continue; % Skip if not enough valid points
    end

    % Get agent's final target assignment color
    final_target = assigned_goal(j);
    if final_target > 0 && final_target <= size(cmap,1)
        col = cmap(final_target,:);
    else
        col = [0.5 0.5 0.5]; % Gray for unassigned
    end

    % Plot trajectory as single colored line
    h_traj = plot3(traj_data(:,1), traj_data(:,2), traj_data(:,3), ...
        'Color', col, 'LineWidth', 5);

    % Mark starting position (white square)
    plot3(start{j}(1), start{j}(2), start{j}(3),
        's', 'MarkerSize', 10, 'MarkerFaceColor', 'white', ...

```

```

'MarkerEdgeColor', 'black', 'LineWidth', 2);

% Mark final position (black circle)
if length(current_pos) >= j && ~isempty(current_pos{j})
    plot3(current_pos{j}(1), current_pos{j}(2), current_pos{j}(3), ...
        'o', 'MarkerSize', 10, 'MarkerFaceColor', 'black', ...
        'MarkerEdgeColor', 'white', 'LineWidth', 2);

    % Agent number label
    text(current_pos{j}(1)+2, current_pos{j}(2)+2, current_pos{j}(3)+2,
...
        sprintf('A%d', j), 'FontSize', 10, 'FontWeight', 'bold', ...
        'Color', 'black', 'BackgroundColor', 'white');
end

% Store handles for legend (one per target)
if ~ismember(final_target, used_targets) && final_target > 0
    trajectory_handles(end+1) = h_traj;
    trajectory_labels{end+1} = sprintf('Target %d Trajectory',
final_target);
    used_targets(end+1) = final_target;
end
end

% Set up the plot appearance
xlabel('X (m)');
ylabel('Y (m)');
zlabel('Z (m)');

% Dynamic title based on APF type and obstacle mode
apf_str = sprintf('%s', apf_names{APF_type});
if static == 1
    title_str = sprintf('Complete Agent Trajectories with Target Assignment
Colors %s', apf_str);
elseif mixed == 1
    title_str = sprintf('Complete Agent Trajectories %s - Mixed Obstacles',
apf_str);
else
    title_str = sprintf('Complete Agent Trajectories %s - Dynamic
Obstacles', apf_str);
end
title(title_str);

% Set view and limits
view(45, 30);
axis equal;
xlim([0 200]);
ylim([0 200]);
zlim([0 100]);

% Clean legend (only if we have trajectory handles)
if ~isempty(trajectory_handles) && ~isempty(trajectory_labels)
    legend(trajectory_handles, trajectory_labels, 'Location', 'northeast',
...
        'FontSize', 10);
end

% Add grid for better visualization
grid on;
set(gca, 'GridAlpha', 0.3);

%%
% Get all figure handles
figs = findall(0, 'type', 'figure');

% Save each figure

```

```

for i = 1:length(figs)
    figure(figs(i)); % Make the figure active
    savefig(figs(i), sprintf('figure_%d.fig', figs(i).Number));
end
% Helper function to check if any agent is close to goal
function agent_near_goal = check_agent_near_goal(current_pos,
assigned_goal, target, threshold)
    agent_near_goal = false;
    for j = 1:length(current_pos)
        if ~isempty(current_pos{j}) && assigned_goal(j) > 0
            goal_pos = target{assigned_goal(j)};
            distance_to_goal = norm(current_pos{j} - goal_pos);
            if distance_to_goal < threshold
                agent_near_goal = true;
                return;
            end
        end
    end
end
%% Function to spawn batch of sequential obstacles
function [obs_states, obs_goals, obstacles, radius, M_obs, obstacle_types] = spawn_obstacle_batch(obs_states, obs_goals, obstacles, radius, M_obs, obstacle_types, datang, obstacles_per_spawn, r_dyn)

    fprintf('Spawning batch of %d obstacles...\n', obstacles_per_spawn);

    for batch_idx = 1:obstacles_per_spawn
        % Add new obstacle state
        new_obs_idx = M_obs + 1;

        % Expand arrays if needed
        if size(obs_states, 2) < new_obs_idx
            obs_states = [obs_states, zeros(12, 1)];
            obs_goals = [obs_goals, zeros(3, 1)];
        end

        % Set spawn position based on datang parameter (same logic as original)
        if datang == 1 % Dari Depan
            start_pos = [180+randn*10; 180+randn*10; 65+randn*10];

            % Set goal to another position
            end_pos = [-40+randn*1; -40+randn*1; 60+randn*10];
        elseif datang == 2
            end_pos = [580+randn*20; 580+randn*20; 62+randn*2];

            % Set goal to another position
            start_pos = [30+randn*20; 30+randn*20; 62+randn*2];
        else
            end_pos = [300+randn*10; 60+randn*10; 55+randn*8];

            % Set goal to another position
            start_pos = [100+randn*1; 210+randn*1; 55+randn*8];
        end

        % Initialize new obstacle
        obs_states(1:3, new_obs_idx) = start_pos;
        obs_states(4:12, new_obs_idx) = zeros(9,1);
        obs_goals(:, new_obs_idx) = end_pos;

        % Update obstacle arrays
        obstacles = [obstacles, start_pos];
        radius = [radius; r_dyn];
        M_obs = new_obs_idx;
    end
end

```

```

% Update obstacle types if in mixed mode
if exist('obstacle_types', 'var')
    obstacle_types = [obstacle_types, 2]; % 2 = dynamic
end

fprintf(' - Obstacle %d/%d spawned at [% .1f, %.1f, %.1f]\n', ...
    batch_idx, obstacles_per_spawn, start_pos(1), start_pos(2),
start_pos(3));
end
end

function save_replay_data(replay_data, end_type)
    % Save the replay data with timestamp
    timestamp = datestr(now, 'yyyymmdd_HHMMSS');
    filename = sprintf('UAV_3D_Replay_%s_%s_%s.mat', ...
        replay_data.settings.apf_name, replay_data.settings.mode_str,
    timestamp);

    % Remove spaces and special characters
    filename = regexp替换成(filename, '[^\w\.\.]', '_');

    % Trim arrays to actual size
    actual_iter = replay_data.actual_iterations;
    replay_data.dynamic.agent_positions =
replay_data.dynamic.agent_positions(:, :, 1:actual_iter);
    replay_data.dynamic.agent_orientations =
replay_data.dynamic.agent_orientations(:, :, 1:actual_iter);
    replay_data.dynamic.agent_assignments =
replay_data.dynamic.agent_assignments(:, 1:actual_iter);
    replay_data.dynamic.agent_status = replay_data.dynamic.agent_status(:, 1:actual_iter);
    replay_data.dynamic.agent_crashed =
replay_data.dynamic.agent_crashed(:, 1:actual_iter);

    if isfield(replay_data.dynamic, 'obstacle_positions')
        replay_data.dynamic.obstacle_positions =
replay_data.dynamic.obstacle_positions(:, :, 1:actual_iter);
        replay_data.dynamic.obstacle_orientations =
replay_data.dynamic.obstacle_orientations(:, :, 1:actual_iter);
        replay_data.dynamic.obstacle_states =
replay_data.dynamic.obstacle_states(:, :, 1:actual_iter);
    end

    % Save additional info
    replay_data.save_info = struct();
    replay_data.save_info.end_type = end_type;
    replay_data.save_info.save_time = datestr(now);
    replay_data.save_info.total_frames = actual_iter;

    save(filename, 'replay_data', '-v7.3');

    file_info = dir(filename);
    fprintf('\n3D Replay data saved: %s\n', filename);
    fprintf('File size: %.2f MB\n', file_info.bytes / (1024*1024));
    fprintf('Total frames: %d\n', actual_iter);
    fprintf('To replay with EXACT same visualization:\n');
    replay_uav_simulation(['%s'\n], filename);
end

function replay_uav_simulation(filename)
    % Replay with EXACT same visualization as original simulation

    if nargin < 1
        [file, path] = uigetfile('UAV_3D_Replay_*.mat', 'Select 3D Replay
File');
        if isequal(file, 0)
            return;
    end

```

```

        end
    filename = fullfile(path, file);
end

fprintf('Loading 3D replay data: %s\n', filename);
load(filename, 'replay_data');

% Create new figure for replay - EXACT same setup as original
replay_fig = figure('Name', sprintf('3D Replay: %s - %s', ...
    replay_data.settings.apf_name, replay_data.settings.mode_str), ...
    'Position', [100, 100, 1400, 900]);

% Setup environment EXACTLY like original
setup_exact_environment(replay_data);

% Create control panel
create_replay_controls_exact(replay_fig, replay_data);

fprintf('3D Replay loaded with EXACT original visualization!\n');
fprintf('- Same quadcopter graphics for agents\n');
fprintf('- Same cylinder/obstacle graphics\n');
fprintf('- Same trajectory colors and styles\n');
fprintf('- Same target markers and labels\n');
end

function setup_exact_environment(replay_data)
    % Setup EXACTLY Like your original simulation

    % Floor - exact same as your code
    fill3([0 200 200 0],[0 0 200 200],[0 0 0 0],[0.3 0.3 0.3]);
    hold on; grid on;

    % Axes - exact same
    xlabel("x"); ylabel("y"); zlabel("z");
    xlim([0 200]); ylim([0 200]); zlim([0 100]);

    % Use your exact view settings
    view(replay_data.settings.xyp, replay_data.settings.zp);

    % Targets - EXACT same as your code
    colors = replay_data.static_elements.target_colors;
    for t = 1:replay_data.settings.M
        target_pos = replay_data.static_elements.targets{t};

        plot3(target_pos(1), target_pos(2), target_pos(3), 'o', ...
            'MarkerSize', 15, 'MarkerFaceColor', colors(t,:), ...
            'MarkerEdgeColor', 'black', 'LineWidth', 2);

        % Target label - exact same
        text(target_pos(1)+5, target_pos(2)+5, target_pos(3)+5, ...
            sprintf('Target %d', t), 'FontSize', 12, 'FontWeight', 'bold', ...
            'Color', colors(t,:), 'BackgroundColor', 'white', ...
            'EdgeColor', 'black');
    end

    % Obstacles - EXACT same as your original code
    if strcmp(replay_data.static_elements.obstacles.type, 'static')
        % Static obstacles (cylinders) - using your exact function
        Cpos = replay_data.static_elements.obstacles.Cpos;
        radius = replay_data.static_elements.obstacles.radii;

        for i = 1:size(Cpos, 1)
            create_cylinder(radius(i), Cpos(i,:), [0.45, 0.58, 0.76]);
        end
    end

```

```

elseif replay_data.settings.mixed == 1
    % Mixed mode - static cylinders only (dynamic obstacles animated
separately)
    if isfield(replay_data.static_elements.obstacles, 'static_indices')
        static_indices =
replay_data.static_elements.obstacles.static_indices;
        Cpos = replay_data.static_elements.obstacles.Cpos;
        radius = replay_data.static_elements.obstacles.radii;

        for idx = 1:length(static_indices)
            i = static_indices(idx);
            create_cylinder(radius(idx), Cpos(idx,:), [0.45, 0.58,
0.76]);
        end
    end
end

title(sprintf('3D Replay: %s - %s', replay_data.settings.apf_name,
replay_data.settings.mode_str));
end

function create_replay_controls_exact(fig, replay_data)
    % Create control panel for replay with EXACT same graphics

    % Control panel
    panel = uipanel('Parent', fig, 'Title', 'Replay Controls', ...
                    'Position', [0.02, 0.02, 0.96, 0.15]);

    % Current frame
    current_frame = 1;
    max_frames = replay_data.actual_iterations;

    % Control elements
    frame_slider = uicontrol('Parent', panel, 'Style', 'slider', ...
                            'Position', [50, 60, 300, 20], ...
                            'Min', 1, 'Max', max_frames, 'Value', 1, ...
                            'SliderStep', [1/max_frames, 10/max_frames]);

    frame_text = uicontrol('Parent', panel, 'Style', 'text', ...
                           'Position', [50, 35, 100, 20], ...
                           'String', sprintf('Frame: %d/%d', 1,
max_frames));

    time_text = uicontrol('Parent', panel, 'Style', 'text', ...
                           'Position', [160, 35, 100, 20], ...
                           'String', sprintf('Time: %.2fs', 0));

    play_button = uicontrol('Parent', panel, 'Style', 'pushbutton', ...
                           'Position', [370, 50, 60, 30], ...
                           'String', 'Play', 'FontWeight', 'bold');

    uicontrol('Parent', panel, 'Style', 'text', ...
              'Position', [450, 65, 50, 15], 'String', 'Speed:');
    speed_slider = uicontrol('Parent', panel, 'Style', 'slider', ...
                            'Position', [450, 45, 100, 20], ...
                            'Min', 0.1, 'Max', 5, 'Value', 1);

    uicontrol('Parent', panel, 'Style', 'pushbutton', ...
              'Position', [570, 50, 60, 30], ...
              'String', 'Reset', ...
              'Callback', @(~,~) reset_animation());

    % Initialize graphics - EXACT same as your original
    agent_handles = cell(replay_data.settings.N, 1);
    agent_text_handles = cell(replay_data.settings.N, 1);
    trajectory_handles = cell(replay_data.settings.N, 1);

```

```

obstacle_handles = [];
obstacle_circle_handles = [];

% Create agent quadcopters - using YOUR exact function
cmap = replay_data.static_elements.colors;
for j = 1:replay_data.settings.N
    initial_pos = squeeze(replay_data.dynamic.agent_positions(j,:,:1));
    initial_assignment = replay_data.dynamic.agent_assignments(j,1);

    if initial_assignment > 0 && initial_assignment <= size(cmap,1)
        col = cmap(initial_assignment,:);
    else
        col = [0 0 1];
    end

    % Use YOUR exact quadcopter creation function
    agent_handles{j} = create_quadcopter_graphics(initial_pos,
zeros(3,1), col, 1.5);

    % Agent text labels - exact same
    agent_text_handles{j} = text(initial_pos(1)-1, initial_pos(2)-1,
initial_pos(3)-1, ...
                                sprintf('%d', j), 'FontSize', 10,
'FontWeight', 'bold');

    % Initialize trajectory handles
    trajectory_handles{j} = [];
end

% Create dynamic obstacle quadcopters if needed
if strcmp(replay_data.static_elements.obstacles.type, 'dynamic') ||
replay_data.settings.mixed == 1
    M_obs = replay_data.settings.M_obs;
    if isfield(replay_data.dynamic, 'obstacle_positions') && M_obs > 0
        obstacle_handles = cell(M_obs, 1);
        obstacle_circle_handles = gobjects(M_obs, 1);

        for k = 1:M_obs
            initial_pos =
squeeze(replay_data.dynamic.obstacle_positions(k,:,:1));
            initial_angles =
squeeze(replay_data.dynamic.obstacle_orientations(k,:,:1));

            % Dynamic obstacle quadcopter - YOUR exact function
            obstacle_handles{k} =
create_quadcopter_graphics(initial_pos, initial_angles, [1 0 0], 2);

            % Safety circle - YOUR exact code
            r_dyn = 6; % Same as your code
            theta_circle = linspace(0, 2*pi, 50);
            circle_x = initial_pos(1) + r_dyn * cos(theta_circle);
            circle_y = initial_pos(2) + r_dyn * sin(theta_circle);
            circle_z = initial_pos(3) * ones(size(theta_circle));
            obstacle_circle_handles(k) = plot3(circle_x, circle_y,
circle_z, 'r--', 'LineWidth', 1.5);
        end
    end
end

% Animation variables
is_playing = false;
timer_obj = [];

% Update function - maintains EXACT same visualization
function update_frame(frame_num)
    frame_num = round(max(1, min(frame_num, max_frames)));

```

```

current_frame = frame_num;

% Update UI
set(frame_slider, 'Value', frame_num);
set(frame_text, 'String', sprintf('Frame: %d/%d', frame_num,
max_frames));
set(time_text, 'String', sprintf('Time: %.2fs', (frame_num-1) *
replay_data.settings.dt));

% Update agents - EXACT same as your original
for j = 1:replay_data.settings.N
    pos =
squeeze(replay_data.dynamic.agent_positions(j,:,:frame_num));
    angles =
squeeze(replay_data.dynamic.agent_orientations(j,:,:frame_num));
    assignment = replay_data.dynamic.agent_assignments(j,
frame_num);
    is_done = replay_data.dynamic.agent_status(j, frame_num);
    is_crashed = replay_data.dynamic.agent_crashed(j, frame_num);

    % Color based on assignment - EXACT same logic
    if assignment > 0 && assignment <= size(cmap,1)
        col = cmap(assignment,:);
    else
        col = [0 0 1];
    end

    % Update quadcopter - YOUR exact function
    if isvalid(agent_handles{j})
        update_quadcopter_graphics(agent_handles{j}, pos, angles);
    end

    % Update text position - EXACT same
    set(agent_text_handles{j}, 'Position', pos - [1;1;1]);

    % Update trajectories with color changes - EXACT same as your
code
    if frame_num > 1
        prev_assignment = replay_data.dynamic.agent_assignments(j,
frame_num-1);
        if assignment ~= prev_assignment ||
isempty(trajetory_handles{j})
            % New trajectory segment
            traj_data =
squeeze(replay_data.dynamic.agent_positions(j,:,:1:frame_num));
            if assignment > 0 && assignment <= size(cmap,1)
                traj_col = cmap(assignment,:);
            else
                traj_col = [0.5 0.5 0.5];
            end

            new_line = plot3(traj_data(1,:), traj_data(2,:),
traj_data(3,:), ...
                            'LineWidth', 1.5, 'Color', traj_col);

            if isempty(trajetory_handles{j})
                trajetory_handles{j} = new_line;
            else
                trajetory_handles{j}(end+1) = new_line;
            end
        end
    end
end

% Update dynamic obstacles - EXACT same as your original
% Update dynamic obstacles with variable count

```

```

if ~isempty(obstacle_handles) || (static == 0)
    % Get number of obstacles for this frame
    if isfield(replay_data.dynamic, 'num_obstacles_per_frame')
        current_num_obstacles =
replay_data.dynamic.num_obstacles_per_frame(frame_num);
    else
        current_num_obstacles =
size(replay_data.dynamic.obstacle_positions, 1);
    end

    % Expand obstacle handles if needed
    while length(obstacle_handles) < current_num_obstacles
        k = length(obstacle_handles) + 1;

        % Create new obstacle graphics
        initial_pos =
squeeze(replay_data.dynamic.obstacle_positions(k,:,frame_num));
        initial_angles =
squeeze(replay_data.dynamic.obstacle_orientations(k,:,frame_num));

        if size(initial_pos, 1) == 1
            initial_pos = initial_pos';
        end
        if size(initial_angles, 1) == 1
            initial_angles = initial_angles';
        end

        % Create new obstacle
        obstacle_handles{k} = create_quadcopter_graphics(initial_pos,
initial_angles, [1 0 0], 2);

        % Create safety circle
        r_dyn = 6;
        theta_circle = linspace(0, 2*pi, 50);
        circle_x = initial_pos(1) + r_dyn * cos(theta_circle);
        circle_y = initial_pos(2) + r_dyn * sin(theta_circle);
        circle_z = initial_pos(3) * ones(size(theta_circle));

        if length(obstacle_circle_handles) < k
            obstacle_circle_handles(k) = plot3(circle_x, circle_y,
circle_z, 'r--', 'LineWidth', 1.5);
        end
    end

    % Update all current obstacles
    for k = 1:current_num_obstacles
        if k <= length(obstacle_handles) && k <=
size(replay_data.dynamic.obstacle_positions, 1)
            pos =
squeeze(replay_data.dynamic.obstacle_positions(k,:,frame_num));
            angles =
squeeze(replay_data.dynamic.obstacle_orientations(k,:,frame_num));

            if size(pos, 1) == 1
                pos = pos';
            end
            if size(angles, 1) == 1
                angles = angles';
            end

            % Hide obstacles that don't exist yet (all zeros)
            if all(pos == 0)
                if isvalid(obstacle_handles{k})
                    set(obstacle_handles{k}, 'Visible', 'off');
                end
            end
        end
    end
end

```

```

        if k <= length(obstacle_circle_handles) &&
isValid(obstacle_circle_handles(k))
            set(obstacle_circle_handles(k), 'Visible', 'off');
        end
        continue;
    else
        if isValid(obstacle_handles{k})
            set(obstacle_handles{k}, 'Visible', 'on');
        end
        if k <= length(obstacle_circle_handles) &&
isValid(obstacle_circle_handles(k))
            set(obstacle_circle_handles(k), 'Visible', 'on');
        end
    end
end

% Update obstacle quadcopter
if isValid(obstacle_handles{k})
    update_quadcopter_graphics(obstacle_handles{k}, pos,
angles);
end

% Update safety circle
if k <= length(obstacle_circle_handles) &&
isValid(obstacle_circle_handles(k))
    r_dyn = 6;
    theta_circle = linspace(0, 2*pi, 50);
    circle_x = pos(1) + r_dyn * cos(theta_circle);
    circle_y = pos(2) + r_dyn * sin(theta_circle);
    circle_z = pos(3) * ones(size(theta_circle));
    set(obstacle_circle_handles(k), 'XData', circle_x, 'YData',
circle_y, 'ZData', circle_z);
end
end
end

drawnow;
end

% Play/pause and other control functions (same as before)
function play_pause_callback(~, ~)
    if is_playing
        if ~isempty(timer_obj) && isValid(timer_obj)
            stop(timer_obj);
            delete(timer_obj);
        end
        set(play_button, 'String', 'Play');
        is_playing = false;
    else
        speed = get(speed_slider, 'Value');
        timer_period = 0.05 / speed;

        timer_obj = timer('Period', timer_period, 'ExecutionMode',
'fixedRate', ...
                           'TimerFcn', @(~,~) advance_frame());
        start(timer_obj);
        set(play_button, 'String', 'Pause');
        is_playing = true;
    end
end

function advance_frame()
    if current_frame < max_frames
        update_frame(current_frame + 1);
    else
        if ~isempty(timer_obj) && isValid(timer_obj)

```

```

        stop(timer_obj);
        delete(timer_obj);
    end
    set(play_button, 'String', 'Play');
    is_playing = false;
end
end

function reset_animation()
    if is_playing
        play_pause_callback([], []);
    end
    update_frame(1);
end

% Set callbacks
set(frame_slider, 'Callback', @(src,~) update_frame(get(src,
'Value')));
set(play_button, 'Callback', @play_pause_callback);

% Initial update
update_frame(1);
end

% Convenience function
function replay_latest_uav()
    files = dir('UAV_3D_Replay_*.mat');
    if isempty(files)
        fprintf('No UAV replay files found!\n');
        return;
    end

    [~, idx] = max([files.datenum]);
    latest_file = files(idx).name;

    fprintf('Loading latest UAV replay: %s\n', latest_file);
    replay_uav_simulation(latest_file);
end

%% Function for creating quadcopter graphics
function hgroup = create_quadcopter_graphics(pos, angles, color, scale)
    if nargin < 4
        scale = 1;
    end

    % Quadcopter parameters
    arm_length = 0.2 * scale;
    body_radius = 0.4 * scale;
    prop_radius = 0.2 * scale;

    % Create graphics group
    hgroup = hgtransform('Parent', gca);

    % Body (central sphere)
    [xs, ys, zs] = sphere(10);
    body = surf(xs*body_radius, ys*body_radius, zs*body_radius, ...
        'Parent', hgroup, ...
        'FaceColor', color, ...
        'EdgeColor', 'none', ...
        'FaceAlpha', 0.8);

    % Arms and propellers
    angles_arm = [45, 135, 225, 315] * pi/180;
    for i = 1:4
        % Arm
        x_arm = [0, arm_length*cos(angles_arm(i))];
        y_arm = [0, arm_length*sin(angles_arm(i))];

```

```

        z_arm = [0, 0];
        plot3(x_arm, y_arm, z_arm, 'k-', 'LineWidth', 2*scale,
'Parent', hgroup);

        % Propeller circle
        theta_prop = linspace(0, 2*pi, 20);
        x_prop = arm_length*cos(angles_arm(i)) +
prop_radius*cos(theta_prop);
        y_prop = arm_length*sin(angles_arm(i)) +
prop_radius*sin(theta_prop);
        z_prop = zeros(size(theta_prop)) + 0.05*scale;
        fill3(x_prop, y_prop, z_prop, [0.3 0.3 0.3], ...
'Parent', hgroup, 'FaceAlpha', 0.5, 'EdgeColor', 'none');
    end

    % Direction indicator (front)
    plot3([0, arm_length*1.2], [0, 0], [0, 0], 'r-', ...
'LineWidth', 3*scale, 'Parent', hgroup);

    % Apply position and rotation
    update_quadcopter_graphics(hgroup, pos, angles);
end

%% Function to update quadcopter position and orientation
function update_quadcopter_graphics(hgroup, pos, angles)
    % Check if hgroup is valid
    if ~isValid(hgroup)
        warning('Invalid graphics handle');
        return;
    end

    % Create transformation matrix
    roll = angles(1);
    pitch = angles(2);
    yaw = angles(3);

    % Rotation matrices
    Rx = [1 0 0; 0 cos(roll) -sin(roll); 0 sin(roll) cos(roll)];
    Ry = [cos(pitch) 0 sin(pitch); 0 1 0; -sin(pitch) 0 cos(pitch)];
    Rz = [cos(yaw) -sin(yaw) 0; sin(yaw) cos(yaw) 0; 0 0 1];

    % Combined rotation
    R = Rz * Ry * Rx;

    % Create 4x4 transformation matrix
    T = eye(4);
    T(1:3, 1:3) = R;
    T(1:3, 4) = pos;

    % Apply transformation
    set(hgroup, 'Matrix', T);
end

%% Function for creating cylinders
function [h1, h2, h3] = create_cylinder(Radius, pos,color)
nSides = 100;
[X,Y,Z] = cylinder(Radius, nSides);
Height = pos(3);
Z = Z*Height;
Xpos = pos(1);
Ypos = pos(2);
X = X + Xpos;
Y = Y + Ypos;
h1 = surf(X,Y,Z,'facecolor',color,'LineStyle','none');
h2 = fill3(X(1,:),Y(1,:),Z(1,:),color);
h3 = fill3(X(2,:),Y(2,:),Z(2,:),color);

```

```

end
%%
function x=UAV_input_Ft(x,Ft,Ftp,dt)

% vx=x(4)+0.1*Ft(1);
% vy=x(5)+0.1*Ft(2);
% vz=x(6)+0.1*Ft(3);
vx=(Ft(1)-Ftp(1))/dt;
vy=(Ft(2)-Ftp(2))/dt;
vz=(Ft(3)-Ftp(3))/dt;
% vx=saturasi(vx,-2.5,2.5);
% vy=saturasi(vy,-2.5,2.5);
% vz=saturasi(vz,-2.5,2.5);

r=zeros(9,1);
r(1)=x(1)+Ft(1);
r(2)=x(2)+Ft(2);
r(3)=x(3)+Ft(3);
r(4)=vx;
r(5)=vy;
r(6)=vz;
dx=pers_state_uavnkontroller(x,r); %dinamika robot + kontroller
% disp([a_v]);
% if (mod(i,100)==0)
% r=[80;80;100].*rand(3,1)-[40;40;0];
% end
% disp([x(1:3) x(4:6) x(7:9)]);
x=x+dt*dx; % dinamika diskritisasi
end

function z=saturasi(x,lb,ub)
if (x>ub)
z=ub;
elseif (x<lb)
z=lb;
else
z=x;
end
end
%%
function dx=pers_state_uavnkontroller(x,r)
dx=[0;0;0;0;0;0;0;0;0];
%3 posisi x y z
%3 kecepatan translasi x y z
%3 orientasi roll pitch yaw
%3 kecepatan sudut roll pitch yaw
%parameter
mm=1;
g=10;
Ix=0.03;
Iy=0.03;
Iz=0.05;
dd=0.7; %gaya gesek udara/drag
d=3.13e-3; %koefisien gerak yaw
L=0.2; %koefisien gerak roll/pitch

%Parameter Kontroller
K2=100;
L2=21;
K3=K2;
L3=L2;
K4=0.09;
L4=0.61;
K1=20;
L1=9;

```

```

%Kontroller Ketinggian dulu aja deh
%xr= [1 2 3 4      5      6      7      8      9      10 11 12]
%x = [x y z u      v      w      roll    pitch   yaw    p    q    r]
%r = [x y z xdot ydot zdot xddot yddot zddot]
u1=mm*(g+K1*(r(3)-x(3))+L1*(r(6)-
x(6))+r(9)+dd/mm*r(6))/cos(x(7))/cos(x(8));
u1=saturasi(u1,0,20); %
Peb=[1 sin(x(7))*tan(x(8)) cos(x(7))*tan(x(8))
    0 cos(x(7)) -sin(x(7))
    0 sin(x(7))/cos(x(8)) cos(x(7))/cos(x(8))];
dx(1:3)=x(4:6);
dx(4)=(sin(x(9))*sin(x(7))+cos(x(9))*cos(x(7))*sin(x(8)))*u1/mm-
dd*x(4)/mm;
dx(5)=(-cos(x(9))*sin(x(7))+sin(x(9))*cos(x(7))*sin(x(8)))*u1/mm-
dd*x(5)/mm;
dx(6)=-g+cos(x(7))*cos(x(8))*u1/mm-dd*x(6)/mm;
% dx(7)=x(10);
% dx(8)=x(11);
% dx(9)=x(12);
dx(7:9)=Peb*x(10:12);

%Kontroller orientasi
s8=sin(x(8))*cos(x(7));
s7=sin(x(7));
% r7=0.4; r8=0.4;

ds8=dx(8)*cos(x(7))*cos(x(8))-dx(7)*sin(x(7))*sin(x(8));
ds7=dx(7)*cos(x(7));

if (x(9)>0.1 || x(9)<-0.1)
    u2=Iy/L*(-K2*s7-L2*ds7);
    u3=Ix/L*(-K3*s8-L3*ds8);
else
    [r7,r8]=outer_cascade_tracking(x,r);
    disp([r7 r8]);
    u2=Iy/L*(K2*r7-K2*s7-L2*ds7);
    u3=Ix/L*(K3*r8-K3*s8-L3*ds8);
end
u4=Iz/d*(-K4*x(9)-L4*x(12));
disp([u1 u2 u3 u4]);
%untuk pdot, qdot, rdot sama dengan teori di buku tesis
dx(10)=((Iy-Iz)*x(11)*x(12)+u2*L)/Ix;
dx(11)=((Iz-Ix)*x(10)*x(12)+u3*L)/Iy;
dx(12)=((Ix-Iy)*x(10)*x(11)+u4*d)/Iz;
end

function [r7,r8]=outer_cascade_tracking(x,r)
s7=(x(1)-r(1))*cos(x(9))+(x(2)-r(2))*sin(x(9));
s8=-(x(1)-r(1))*sin(x(9))+(x(2)-r(2))*cos(x(9));
v7=(x(4)-r(4))*cos(x(9))+(x(5)-r(5))*sin(x(9));
v8=-(x(4)-r(4))*sin(x(9))+(x(5)-r(5))*cos(x(9));
s=sqrt(s7*s7+s8*s8);
v=sqrt(v7*v7+v8*v8);
% disp([arah7 arah8]);
% disp([s7 s8 v7 v8]);
sa=25;
sb=6;
vc=4;
if (s>sa)
    a=1;
    b=0;
elseif (s>sb)
    a=(s-sb)/(sa-sb);
    b=1-a;
else

```

```

        a=0;
        b=s/sb;
    end
    if (v>vc)
        c=1;
    else
        c=v/vc;
    end
    rra=0.5;
    rrb=0.3;
    rrc=0.5;
    if(v>0 && s>0)
        r7=s8/s*rra*a+s8/s*rrb*b*(1-c)+v8/v*rrc*c*(1-a);
        r8=-(s7/s*rra*a+s7/s*rrb*b*(1-c)+v7/v*rrc*c*(1-a));
    elseif (s>0)
        r7=s8/s*rra*a+s8/s*rrb*b*(1-c);
        r8=-(s7/s*rra*a+s7/s*rrb*b*(1-c));
    elseif (v>0)
        r7=v8/v*rrc*c*(1-a);
        r8=-(v7/v*rrc*c*(1-a));
    else
        r7=0;
        r8=0;
    end
    if (r7~=0 && r8~=0)
        rrr=sqrt(r7^2+r8^2);
        if (rrr>0.5)
            r7=0.5*r7/rrr;
            r8=0.5*r8/rrr;
        end
    end
end

```

```

%% Main state update function
function x = obs_state_update(x, a_des, dt)
    g = 10;          % gravity (m/s^2)

    a_des(3) = a_des(3) + g;

    dx = obsquadcopter_dynamics_with_controller(x, a_des);
    x = x + dt*dx;
end

%% 6DOF Quadcopter Dynamics with Integrated Controller
function dx = obsquadcopter_dynamics_with_controller(x, a_des)
    dx = zeros(12,1);

    pos = x(1:3);
    vel = x(4:6);
    rpy = x(7:9);
    pqr = x(10:12);

    roll = rpy(1); pitch = rpy(2); yaw = rpy(3);
    p = pqr(1); q = pqr(2); r = pqr(3);

    m = 1;
    g = 10;
    Ix = 0.03;
    Iy = 0.03;
    Iz = 0.05;
    L = 0.2;
    d = 3.13e-3;
    drag = 0.7;

```

```

Kp_pos = 2.0;
Kd_pos = 3.0;
Kp_att = 100;
Kd_att = 21;
Kp_yaw = 0.09;
Kd_yaw = 0.61;

a_current = vel;

thrust_acc = a_des(3) + drag*vel(3)/m;
u1 = m * thrust_acc / (cos(roll)*cos(pitch));
u1 = saturate(u1, 0, 1000);

ax_des = a_des(1) + drag*vel(1)/m;
ay_des = a_des(2) + drag*vel(2)/m;

pitch_des = atan2(ax_des, g);
roll_des = atan2(-ay_des*cos(pitch_des), g);

pitch_des = saturate(pitch_des, -pi/6, pi/6);
roll_des = saturate(roll_des, -pi/6, pi/6);

% Hitung arah horizontal dari akselerasi yang diinginkan
yaw_des = 0;

roll_error = roll_des - roll;
u2 = Iy/L * (Kp_att*roll_error - Kd_att*p);

pitch_error = pitch_des - pitch;
u3 = Ix/L * (Kp_att*pitch_error - Kd_att*q);

yaw_error = wrapToPi(yaw_des - yaw);
u4 = Iz/d * (Kp_yaw*yaw_error - Kd_yaw*r);

dx(1:3) = vel;

dx(4) = (sin(yaw)*sin(roll) + cos(yaw)*cos(roll)*sin(pitch))*u1/m - drag*vel(1)/m;
dx(5) = (-cos(yaw)*sin(roll) + sin(yaw)*cos(roll)*sin(pitch))*u1/m - drag*vel(2)/m;
dx(6) = -g + cos(roll)*cos(pitch)*u1/m - drag*vel(3)/m;

dx(7) = p + sin(roll)*tan(pitch)*q + cos(roll)*tan(pitch)*r;
dx(8) = cos(roll)*q - sin(roll)*r;
dx(9) = sin(roll)/cos(pitch)*q + cos(roll)/cos(pitch)*r;

dx(10) = ((Iy-Iz)*q*r + u2*L)/Ix;
dx(11) = ((Iz-Ix)*p*r + u3*L)/Iy;
dx(12) = ((Ix-Iy)*p*q + u4*d)/Iz;

end
%% Utility function: Saturation
function y = saturate(x, lower, upper)
    if x > upper
        y = upper;
    elseif x < lower
        y = lower;
    else
        y = x;
    end
end

%%
function neigh = find_neighbors(i, current_pos, ~)

```

```

        neigh = voronoi_neighbors(i, current_pos);
    end
%%
function VN = voronoi_neighbors(i, current_pos)
    N = numel(current_pos);
    pos_i = current_pos{i}; VN = [];
    for j=1:N
        if j==i, continue; end
        mid = (pos_i+current_pos{j})/2;
        if norm(pos_i-mid)<norm(current_pos{j}-mid)
            VN(end+1)=j;
        end
    end
end
%%
function [assigned_goal, theta, requests,params] = alokasi_tugasnew(i,
current_pos, assigned_goal, theta, neighbors, target, params,
requests,done)

% Input:
% i : indeks agen
% current_pos : cell array posisi {1..N}
% assigned_goal : 1xN, tujuan tiap agen
% theta : 1xN, prioritas unik
% neighbors : cell array list tetangga per agen
% target : cell array posisi target {1..M}
% params : struct R_STOP, R_SWITCH, R_ATTR, dtheta, delta,
comm_range, timeout
% requests : struct berisi:
%     sent{i}.to : list permintaan kirim
%     in{i} : incoming request FIFO
%     ack{i} : incoming ack FIFO
%     timer{i}{j} : timestamp request i->j
if done(i)
    return;
end

% 1) Hitung neighbor Voronoi lokal via half-space
neighbors{i} = voronoi_neighbors(i, current_pos);
pos_i = current_pos{i};
g_i = assigned_goal(i);
cnt = histcounts(assigned_goal, 1:(numel(target)+1));

% 2) Proses timeout request
now = tic;
for j = requests.sent(i).to
    if now - requests.timer{i}{j} > params.timeout
        % batal request yg timeout
        requests.sent(i).to(requests.sent(i).to==j) = [];
    end
end

% 3) Proses incoming requests (FIFO)
while ~isempty(requests.in{i})
    sender = requests.in{i}(1);
    requests.in{i}(1) = [];
    % Rule 2/3: jika sender block dan tujuan berbeda
    if theta(sender)<theta(i) && assigned_goal(sender)~=g_i
        requests.ack{i}(end+1)=sender; % positif
    else
        requests.nack{i}(end+1)=sender; % negatif
    end
end

```

```

% 4) Ekskusi swap jika ada ACK sesuai FIFO
while ~isempty(requests.ack{i})
    j = requests.ack{i}(1);
    requests.ack{i}(1)=[];
    % swap assigned_goal dan update theta
    tmp = assigned_goal(i);
    assigned_goal(i) = assigned_goal(j);
    assigned_goal(j) = tmp;
    theta(i) = theta(i) + params.dtheta;
end

% 5) Inisiasi request baru ke blockers (B+ dan B++)
[Bp, Bpp] = classify_blockers(i, current_pos, neighbors{i},
target{g_i}, params);
candidates = [Bp, Bpp];
for j = candidates
    d_ij = norm(pos_i - current_pos{j});
    if d_ij<params.R_SWITCH && d_ij>params.R_STOP

        wz = 1;
        diff_i = pos_i - target{g_i};
        diff_j = current_pos{j} - target{g_i};
        cost_i = norm([diff_i(1:2); wz*diff_i(3)]);
        cost_j = norm([diff_j(1:2); wz*diff_j(3)]);

        % Softmax-compare
        pi = exp(-params.Kcost * cost_i);
        pj = exp(-params.Kcost * cost_j);
        if pj > pi
            % kirim request i->j jika j lebih "berhak"
            requests.sent(i).to(end+1) = j;
            requests.timer{i}{j} = now;
            requests.in{j}(end+1) = i;
        end
        % =====
    end
end

% -----
N = numel(current_pos);
M = numel(target);
counts = histcounts(assigned_goal, 1:(M+1)); % vektor 1xM

% 6) Rule 1+: stop hanya kalau belum penuh kuota
if norm(pos_i - target{g_i}) < params.R_STOP && counts(g_i) <
params.quota(g_i)
    % hanya hold kalau target g_i belum mencapai quota(g_i)
    return;
end
% ----

% 7) Deadlock resolution sesuai paper
% 7) Deadlock resolution sesuai paper
G = build_request_graph(requests.sent);
[found, cycle] = detect_cycle(G);
if found
    victim = cycle( find(theta(cycle)==max(theta(cycle)), 1) );

    % cek ada alternatif goal?
    alt_goals = setdiff(1:numel(target), assigned_goal(victim));
    if isempty(alt_goals)
        return; % no alternative → skip swap
    end

```

```

    % kalau ada, hitung cost dan swap
    costs = arrayfun(@(t) norm(current_pos{victim}-target{t}),
alt_goals);
    [~, m] = min(costs);
    assigned_goal(victim) = alt_goals(m);
    theta(victim)      = theta(victim) + params.dtheta;
end

% setelah semua swap / deadlock resolution selesai
params.assigngoal(end+1, :) = assigned_goal;

end

%%

function [Bp, Bpp] = classify_blockers(i, current_pos, neigh, goal_i,
params)
    pos_i = current_pos{i};
    dir   = (goal_i - pos_i) / norm(goal_i - pos_i);
    Bp = []; Bpp = [];
    for j = neigh
        v = current_pos{j} - pos_i;
        angle = acos( dot(v,dir) / (norm(v)*norm(dir) + eps) );
        if angle < pi/2
            Bp(end+1) = j;
        else
            Bpp(end+1) = j;
        end
    end
end
%%

%% Helper: build graph
function G = build_request_graph(sent)
    N=numel(sent); G = cell(N,1);
    for k=1:N, G{k}=sent(k).to; end
end
%%

function [found,cycle] = detect_cycle(G)
    N=numel(G);
    vis=false(1,N); st=false(1,N);
    for v=1:N
        [found,cycle]=dfs(v,vis,st,[],G);
        if found, return; end
    end
    found=false; cycle=[];
end
%%

function [found,cycle]=dfs(v,vis,st,path,G)
    vis(v)=true; st(v)=true; path(end+1)=v;
    for w=G{v}
        if ~vis(w)
            [found,cycle]=dfs(w,vis,st,path,G);
            if found, return; end
        elseif st(w)
            idx=find(path==w,1);
            cycle=path(idx:end); found=true; return;
        end
    end
    st(v)=false; found=false; cycle=[];
end

%% Revised avoidance_multiRobot function based on Du et al. 2019 paper
function [Ft, Fatt, Frep_total] = avoidance_multiRobot_pure_dapf( ...
j, target, pos, static, goal, vcur, ...
Kform, ...
all_obs, all_radius, Fcache, agenlain, group, ...

```

```

params, ...
params_dist, gains_att, gains_rep, ...
desired_speed, all_speeds, speed_agent, agent_radius, agent_pos,
all_agent_goals)

% 3) Attractive Force (Equations 17-21)
d_goal = norm(goal - pos);
r_min_safe = params_dist.rminsafe;
kv = params_dist.kw;
ka = params_dist.ka;
wmax = params_dist.wmax;
beta = params_dist.beta;
theta1 = params_dist.theta1;

% Position attractive force (Equation 18)
if d_goal > eps
    F_att_p = gains_att.kp * (goal - pos);
else
    F_att_p = zeros(3,1);
end

% Speed attractive force (Equation 19)
Vo = desired_speed;
if d_goal > eps
    desired_velocity = Vo * (goal - pos) / d_goal;
else
    desired_velocity = zeros(3,1);
end
F_att_v = gains_att.kv * (desired_velocity - vcur);

% Check if in collision avoidance mode
in_collision_avoidance = false;
for k = 1:size(all_obs,2)
    p_obs = all_obs(:,k);
    if k <= size(all_speeds, 2), v_obs = all_speeds(:,k); else, v_obs =
zeros(3,1); end
    r_obs = all_radius(k);

    % Calculate rsafe for this check
    Ev = vcur - v_obs;
    dE = p_obs - pos;
    norm_Ev = norm(Ev) + eps;
    norm_dE = norm(dE) + eps;
    cos_delta = dot(Ev, dE) / (norm_Ev * norm_dE);
    cos_delta = max(-1, min(1, cos_delta));
    delta = acos(cos_delta);

    r_min_safe = params_dist.rminsafe;
    % Variable part of safety distance (Equation 7)
    r_var_safe = r_min_safe + (kv/(ka + wmax)) * norm_Ev * cos(delta);

    % Calculate connection part parameters for equation (8)
    % Variable safety distance at boundary angle β
    r_var_safe_at_beta = r_min_safe + (kv/(ka + wmax)) * norm_Ev * cos(beta);

    % Connection part of safety distance (Equation 8)
    % Based on geometric construction from Figure 1
    OM_distance = r_var_safe_at_beta; % Distance at boundary angle β
    angle_OQM = beta + theta1; % Based on the geometric construction shown in
    Figure 1

    % Connection part of safety distance (Equation 8)
    if sin(angle_OQM) ~= 0
        r_connection_safe = OM_distance * sin(beta - theta1) / sin(angle_OQM);
    else
        r_connection_safe = r_min_safe; % Fallback for degenerate case
    end
end

```

```

end

% Ensure r_connection_safe is within reasonable bounds
r_connection_safe = max(r_connection_safe, r_min_safe);
r_connection_safe = min(r_connection_safe, r_var_safe_at_beta);

% Determine rsafe based on angle δ (Equation 9)
if abs(delta) <= beta
    % Use variable safety distance
    rsafe = r_var_safe;
elseif abs(delta) > beta && abs(delta) < (pi/2 + theta1)
    % Connection region - use equation (8)
    rsafe = r_connection_safe;
else
    % Use minimum safety distance (abs(delta) >= pi/2 + theta1)
    rsafe = r_min_safe;
end

% Add obstacle radius to safety distance
rsafe = rsafe + r_obs;

if static == 1
    dx = pos(1:2) - p_obs(1:2);
    dist_2d = norm(dx);
    if dist_2d < rsafe && pos(3) < p_obs(3)
        in_collision_avoidance = true;
        break;
    end
else
    dist_to_obs = norm(pos - p_obs);
    if dist_to_obs < agent_radius*2
        in_collision_avoidance = true;
        break;
    end
end
end

% Apply attractive force with damping (Equation 21)
if in_collision_avoidance
    % During collision avoidance - no damping
    Fatt = F_att_p + F_att_v;
else
    % Normal flight - add damping to prevent oscillation
    kdamp = gains_att.kdamp;
    damping_force = kdamp * vcur;
    Fatt = F_att_p + F_att_v - damping_force;
end

%% 1) Calculate Adaptive Safety Distance (Equation 9 from paper)
% This is the key difference - proper adaptive safety distance
calculation

Frep_total = zeros(3,1);

for k = 1:size(all_obs,2)
    p_obs = all_obs(:,k);
    v_obs = all_speeds(:,k);
    r_obs = all_radius(k);

    % Relative velocity and position (Equations 10-11)
    Ev = vcur - v_obs; % Relative velocity
    dE = p_obs - pos; % Relative position (from UAV
to obstacle)

    % Calculate angle δ between relative velocity and position
    norm_Ev = norm(Ev) + eps;

```

```

norm_dE = norm(dE) + eps;
cos_delta = dot(Ev, dE) / (norm_Ev * norm_dE);
cos_delta = max(-1, min(1, cos_delta)); % Clamp to [-1, 1]
delta = acos(cos_delta);

% Actual distance to obstacle
if static == 1
    % Cylinder obstacle
    dx = pos(1:2) - p_obs(1:2);
    dist_2d = norm(dx);
    if dist_2d < r_obs && pos(3) < p_obs(3)
        dE_actual = 0.1; % Very close
    else
        if pos(3) < p_obs(3)
            dE_actual = dist_2d - r_obs;
        else
            dE_actual = norm([dx; pos(3) - p_obs(3)]) - r_obs;
        end
    end
else
    % Sphere obstacle
    dE_actual = norm(dE) - r_obs;
    dE_actual = max(dE_actual, 0.1); % Avoid negative distances
end

% Calculate adaptive safety distance rsafe (Equation 9)

% Variable part of safety distance (Equation 7)
r_var_safe = r_min_safe + (kv/(ka + wmax)) * norm_Ev * cos(delta);

% Calculate connection part parameters for equation (8)
% Variable safety distance at boundary angle β
r_var_safe_at_beta = r_min_safe + (kv/(ka + wmax)) * norm_Ev * cos(beta);

% Connection part of safety distance (Equation 8)
% Based on geometric construction from Figure 1
OM_distance = r_var_safe_at_beta; % Distance at boundary angle β
angle_OQM = beta + theta1; % Based on the geometric construction shown in
Figure 1

% Connection part of safety distance (Equation 8)
if sin(angle_OQM) ~= 0
    r_connection_safe = OM_distance * sin(beta - theta1) / sin(angle_OQM);
else
    r_connection_safe = r_min_safe; % Fallback for degenerate case
end

% Ensure r_connection_safe is within reasonable bounds
r_connection_safe = max(r_connection_safe, r_min_safe);
r_connection_safe = min(r_connection_safe, r_var_safe_at_beta);

% Determine rsafe based on angle δ (Equation 9)
if abs(delta) <= beta
    % Use variable safety distance
    rsafe = r_var_safe;
elseif abs(delta) > beta && abs(delta) < (pi/2 + theta1)
    % Connection region - use equation (8)
    rsafe = r_connection_safe;
else
    % Use minimum safety distance (abs(delta) >= pi/2 + theta1)
    rsafe = r_min_safe;
end

% Add obstacle radius to safety distance
rsafe = rsafe+r_obs ;

```

```

% Check if within safety distance
if dE_actual >= rsafe
    continue; % No repulsion needed
end

% Calculate threat level (Equation 12)
if cos(delta) > 0
    TH_level = (1/dE_actual - 1/rsafe) * norm_Ev * cos(delta);
else
    TH_level = 0;
end

% Position repulsion (Equation 13)
if dE_actual < rsafe && delta >= 0 && delta <= pi/2
    F_p_rep = gains_rep.krep * TH_level * (-dE / norm_dE);
else
    F_p_rep = zeros(3,1);
end

% Speed repulsion (Equation 14)
if dE_actual < rsafe && delta > pi/2 && delta < 3*pi/2
    if abs(cos(delta)) > eps
        perp_component = -dE/norm_dE * (1/cos(delta)) + Ev/norm_Ev;
    else
        perp_component = Ev/norm_Ev;
    end
    perp_norm = norm(perp_component) + eps;
    perp_component = perp_component / perp_norm;

    F_v_rep = gains_rep.krepv * TH_level * perp_component;
else
    F_v_rep = zeros(3,1);
end

% Total repulsion from this obstacle (Equation 15)
if static == 1
    F_rep_k = (F_p_rep + F_v_rep)*norm(Fatt);
else
    F_rep_k = (F_p_rep + F_v_rep);
end

% For static obstacles, remove vertical component
if static == 1
    F_rep_k(3) = 0;
end

if static == 1 && pos(3)> p_obs(3) + 2
    F_rep_k = zeros(3,1);
end

Frep_total = Frep_total + Frep_k;
end

%% 2) Process other agents as obstacles via helper
% Filter out stopped agents before calculating agent interactions
active_agent_pos = [];
active_speed_agent = [];

if size(agent_pos, 2) > 0 % Check if there are other agents
    for i = 1:size(agent_pos, 2)
        % Calculate distance from each agent to its goal
        agent_goal = all_agent_goals(:, i); % Get goal for agent i
        agent_position = agent_pos(:, i); % Get position for agent i

```

```

        agent_to_goal_dist = norm(agent_goal - agent_position);

    % Only include agents that haven't stopped (distance to goal >=
R_STOP)
    if agent_to_goal_dist >= params.R_STOP
        active_agent_pos = [active_agent_pos, agent_position];
        active_speed_agent = [active_speed_agent, speed_agent(:, i)];
    end
end

% Calculate agent interaction only with active (non-stopped) agents
if ~isempty(active_agent_pos)
    F_agent = calculateAgentInteraction( ...
        pos, d_goal, vcur, active_agent_pos, active_speed_agent,
params_dist, gains_rep, gains_att);
else
    F_agent = zeros(3,1); % No active agents to interact with
end

% Also prevent current agent from receiving F_agent if it has stopped
if d_goal < 1.2* params.R_STOP
    F_agent = zeros(3,1);
end
F_agent(3) = 0;
%F_agent = zeros(3,1);

Frep_total = Frep_total + F_agent;
%
%% 4) Total Force (Equation 22)
Ft = Fatt + Frep_total;

end

function F_agent = calculateAgentInteraction( ...
    pos_uav, d_goal, vel_uav, agents_pos, agents_vel, params_dist,
gains_rep, gains_att)
% calculateAgentInteraction Compute repulsive or attractive force per
agent
    F_agent = zeros(3,1);
    nAgents = size(agents_pos,2);
    r_safe_agent = 5; % safe radius

    for a = 1:nAgents
        p_a = agents_pos(:,a);
        v_a = agents_vel(:,a);

        % relative vectors
        Ev = vel_uav - v_a;
        dE = p_a - pos_uav;
        normEv = norm(Ev) + eps;
        normdE = norm(dE) + eps;

        % distance from UAV surface to agent
        dE_actual = max(normdE - r_safe_agent, 0.1);

        % compute adaptive rsafe (simplified: variable or min)
        cos_d = dot(Ev,dE)/(normEv*normdE);
        cos_d = max(-1,min(1,cos_d));
        delta = acos(cos_d);

```

```

% Add obstacle radius to safety distance
rsafe = 1;

    % decide repulsion vs attraction
    if dE_actual < rsafe
        % --- repulsion ---
        if cos_d>0
            TH = (1/dE_actual - 1/rsafe)*normEv*cos_d;
        else
            TH = 0;
        end
        % position repulsion
        if delta>=0 && delta<=pi/2
            Fp = 15 * gains_rep.krepp * TH * (-dE / normdE);
        else
            Fp = zeros(3,1);
        end
        % speed repulsion
        if delta>pi/2 && delta<3*pi/2
            if abs(cos_d)>eps
                perp = -dE/normdE*(1/cos_d) + Ev/normEv;
            else
                perp = Ev/normEv;
            end
            perp = perp / (norm(perp)+eps);
            Fv = 15* gains_rep.krepv * TH * perp;
        else
            Fv = zeros(3,1);
        end
        F_int = Fp + Fv;
    else
        % --- attraction (weak) towards agent to maintain cohesion ---
        % F_int = gains_att.kp * (p_a - pos_uav);
        F_int = zeros(3,1);
    end

    if d_goal < 6
        F_int = zeros(3,1);
    end

    F_agent = F_agent + F_int;

end end

%% Compute safety distance according to Du et al. 2019 (Eq. 9)
function rsafe = compute_safety_distance_du2019(p, v, p_obs, v_obs,
r_obs, params_dist)
    % Relative velocity and position
    Ev = v - v_obs; % Relative velocity
    Ed = p_obs - p; % Relative position (from UAV to
obstacle)

    % Norms
    norm_Ev = norm(Ev) + eps;
    norm_Ed = norm(Ed) + eps;

    % Angle between relative velocity and position ( $\delta$  in paper)
    cos_delta = dot(Ev, Ed) / (norm_Ev * norm_Ed);
    cos_delta = max(-1, min(1, cos_delta)); % Clamp to [-1, 1]
    delta = acos(cos_delta);

    % Parameters from paper

```

```

r_min_safe = params_dist.rminsafe; % Include obstacle radius
kv = params_dist.kw;
ka = params_dist.ka;
wmax = params_dist.wmax;
beta = params_dist.beta; %  $\beta$  in paper
theta1 = params_dist.theta1; %  $\theta_1$  in paper

% Variable part (Eq. 7)
r_var_safe = r_min_safe + (kv / (ka + wmax)) * norm_Ev *
cos(delta);

% Safety distance based on angle (Eq. 9)
if abs(delta) >= 0 && abs(delta) < beta
    % Use variable safety distance
    rsafe = r_var_safe;
elseif abs(delta) >= beta && abs(delta) <= (pi/2 + theta1)
    % Connection region - linear interpolation
    t = (abs(delta) - beta) / ((pi/2 + theta1) - beta);
    rsafe = (1 - t) * r_var_safe + t * r_min_safe;
else
    % Use minimum safety distance
    rsafe = r_min_safe;
end
end
%% Compute repulsion according to Du et al. 2019 paper
function Frep = compute_repulsion_du2019(p, v, p_obs, v_obs, r_obs,
h_obs, params_dist, gains_rep, static)
    % Initialize
    Frep = zeros(3,1);

    % For static obstacles, check if UAV is above obstacle height
    if static == 1 && p(3) > h_obs
        return; % No repulsion if well above obstacle
    end

    % Relative velocity and position
    Ev = v - v_obs; % Relative velocity
    Ed = p_obs - p; % Relative position (from UAV to
obstacle)

    % Distance to obstacle surface
    if static == 1
        % Cylinder: project to nearest point on cylinder surface
        dx = p(1:2) - p_obs(1:2);
        d_horiz = norm(dx);
        if d_horiz > eps
            n_horiz = dx / d_horiz;
        else
            n_horiz = [1; 1];
        end
        q_xy = p_obs(1:2) + n_horiz * r_obs;
        z_clamp = min(max(p(3), 0), h_obs);
        q = [q_xy; z_clamp];
        dE_actual = norm(p - q);
    else
        % Sphere: distance to center minus radius
        dE_actual = norm(Ed) - r_obs;
        dE_actual = max(dE_actual, eps); % Avoid negative distances
    end

    % Safety distance
    rsafe = compute_safety_distance_du2019(p, v, p_obs, v_obs, r_obs,
params_dist);
    % Check if within safety distance
    if dE_actual >= rsafe
        return; % No repulsion needed
    end
end

```

```

    end

    % Norms
    norm_Ev = norm(Ev) + eps;
    norm_Ed = norm(Ed) + eps;

    % Angle between relative velocity and position ( $\delta$ )
    cos_delta = dot(Ev, Ed) / (norm_Ev * norm_Ed);
    cos_delta = max(-1, min(1, cos_delta));
    delta = acos(cos_delta);

    % Threat Level (Eq. 12)

    if cos_delta > 0
        if static == 0
            TH_level = (1/dE_actual - 1/rsafe) * norm_Ev * cos_delta;
        else
            TH_level = (1/dE_actual - 1/rsafe) * norm_Ev * cos_delta;
        end
    else
        TH_level = 0;
    end

    % Unit vector from UAV to obstacle
    e_dE = Ed / norm_Ed;

    % Position Repulsion (Eq. 13)
    if dE_actual < rsafe && delta >= 0 && delta <= pi/2
        Fp_rep = gains_rep.krep * TH_level * (-e_dE);
    else
        Fp_rep = zeros(3,1);
    end

    % Speed Repulsion (Eq. 14)
    if dE_actual < rsafe && delta > pi/2 && delta < 3*pi/2
        e_v = Ev / norm_Ev;
        % The perpendicular component from Eq. 14
        if abs(cos_delta) > eps
            perp = -e_dE * (1/cos_delta) + e_v;
        else
            % When  $\cos(\delta) \approx 0$ , use simplified perpendicular
            perp = e_v;
        end
        perp_norm = norm(perp) + eps;
        perp = perp / perp_norm;

        Fv_rep = gains_rep.krepv * TH_level * perp;
    else
        Fv_rep = zeros(3,1);
    end

    % Total repulsion (Eq. 15)
    Frep = Fp_rep + Fv_rep;

    % For static obstacles, optionally remove vertical component
    if static == 1 && norm(v_obs) < eps
        Frep(3) = 0;
    end
end

% Helper function to check if near obstacle (using Du et al. 2019
safety distance)
function flag = near_obstacle_du2019(p, all_obs, rad, params_dist, v,
v_obs_all)
    flag = false;
    for k = 1:size(all_obs,2)

```

```

p_obs = all_obs(:,k);
if k <= size(v_obs_all,2)
    v_obs = v_obs_all(:,k);
else
    v_obs = zeros(3,1); % Static obstacle
end
r_obs = rad(k);

% Compute safety distance for this obstacle
rsafe = compute_safety_distance_du2019(p, v, p_obs, v_obs,
r_obs, params_dist);

% Check actual distance
dist = norm(p - p_obs) - r_obs; % Distance to surface

if dist < rsafe
    flag = true;
    return;
end
end
end

%%
function d = jarak3(a,b)
    d = norm(a-b);
end

%% Traditional APF Implementation
function [Ft, Fatt, Frep_total] = traditional_APF(j, target, pos,
static, goal, all_obs, all_radius, gains_att, gains_rep, params)
    % Traditional Artificial Potential Field
    % Simple attractive force to goal and repulsive forces from
obstacles

    %% Attractive Force
    d_goal = norm(goal - pos);
    if d_goal > eps
        Fatt = gains_att.kp * (goal - pos) / d_goal;

        % Limit force near goal
        if d_goal < params.R_STOP * 3
            scale = d_goal / (params.R_STOP * 3);
            Fatt = Fatt * scale;
        end
    else
        Fatt = zeros(3,1);
    end

    %% Repulsive Force
    Frep_total = zeros(3,1);
    d0 = 10; % Influence distance

    for k = 1:size(all_obs,2)
        p_obs = all_obs(:,k);
        r_obs = all_radius(k);

        if static == 1 && k <= size(all_obs,2)
            % Cylinder obstacle
            dx = pos(1:2) - p_obs(1:2);
            dist_2d = norm(dx);
            if dist_2d < d0 && pos(3) < p_obs(3)
                if dist_2d < r_obs
                    dist_2d = 0.1; % Avoid division by zero
                end
            end
        end
    end
end

```

```

        Frep_k = gains_rep.krepp * (1/(dist_2d-r_obs) - 1/d0) *
(1/(dist_2d-r_obs))^2;
        if dist_2d > eps
            Frep_k = Frep_k * [dx/dist_2d; 0];
        else
            Frep_k = [100; 100; 0]; % Emergency repulsion
        end
        Frep_total = Frep_total + Frep_k;
    end
else
    % Sphere obstacle
    dist = norm(pos - p_obs) - r_obs;
    if dist < d0 && dist > 0
        % Traditional repulsive force
        Frep_k = gains_rep.krepp * (1/dist - 1/d0) *
(1/dist)^2;
        direction = (pos - p_obs) / norm(pos - p_obs);
        Frep_total = Frep_total + Frep_k * direction;
    elseif dist <= 0
        % Emergency repulsion
        direction = (pos - p_obs);
        if norm(direction) < eps
            direction = [1; 0; 0];
        else
            direction = direction / norm(direction);
        end
        Frep_total = Frep_total + 100 * direction;
    end
end
end

%% Total Force
Ft = Fatt + Frep_total;
end

%% Modified APF (MAPF) Implementation
function [Ft, Fatt, Frep_total] = modified_APF(j, target, pos, static,
goal, vcur, all_obs, all_radius, gains_att, gains_rep, params,
desired_speed)
% Modified APF with multi-robot avoidance implementation
% Mapping input parameters to new implementation variables
current_pos = pos;
obstacles = all_obs;
radius = all_radius;
Katt = gains_att.kp; % Use position gain as attractive gain
Krep = gains_rep.krepp; % Use repulsive gain
Kform = 0; % Formation gain (default value)

% Initialize variables
goalp = goal;
robot_height = current_pos(3,1);
goal_height = goal(3,1);
flag = 0;

% Initialize force matrix
F = zeros(3, size(obstacles,2) + 10); % Extra space for multi-agent forces

%% Attractive Force
Fatt = potential_attraction(Katt, current_pos, goal);

%% Repulsive Forces from Static Obstacles
for k = 1: length(obstacles(1,:))
    % Measuring the horizontal distance between UAV and centre axis of the
    building
    rou = sqrt((current_pos(1,1)-obstacles(1,k))^2+(current_pos(2,1)-
    obstacles(2,k))^2)-radius(k,1);

```

```

rou3d = sqrt((current_pos(1,1)-obstacles(1,k))^2+(current_pos(2,1)-
obstacles(2,k))^2+(current_pos(3,1)-obstacles(3,k))^2);

% differentiation of variable rou
d_rou = [current_pos(1,1)-obstacles(1,k); current_pos(2,1)-
obstacles(2,k)]/(rou+radius(k,1));

% Threshold value to judge whether the UAV near to building or not?
zeta = max(3*radius(k,1),5);
n = 2;

if rou<=0
    if static == 1
        if robot_height <= obstacles(3,k)
            disp("Crash Detected");
            F(:,k)=0;
        elseif robot_height <= obstacles(3,k)+zeta
            F(:,k) = (n/2)*Krep*(1/(robot_height-obstacles(3,k))-1/zeta)^2*dist_factor(current_pos, goal, n-1,
flag)*diff_distance_factor(current_pos, goal, n, flag);
        else
            F(:,k)=0;
        end
    elseif rou3d <= radius(k,1)
        disp("Crash Detected");
        F(:,k)=0;
    else
        F(:,k)=0;
    end
elseif rou<=zeta
    if robot_height <= obstacles(3,k)
        % Flag - that tells the UAV to move in xy plane
        % no increment in height.
        Frep1 = Krep*abs((1/rou)-(1/zeta))*d_rou;
        F(:,k) = vertcat(Frep1,0);
        F(3,k)=0;
    else
        F(:,k) = 0;
    end
elseif rou > zeta
    F(:,k) = 0;
end
end

%% Multi-Agent Avoidance (if other agents exist)
% For now, using empty agenlain and group since not provided in original
parameters
agenlain = {};% This would need to be passed as parameter for multi-agent
scenarios
group = [];% This would need to be passed as parameter for formation
control

[~,nkk]=size(agenlain);
for kk = 1: nkk
    % Measuring the horizontal distance between UAV and the other agent
    rou = sqrt((current_pos(1,1)-agenlain{kk}(1))^2+(current_pos(2,1)-
agenlain{kk}(2))^2 ...
    +(current_pos(3,1)-agenlain{kk}(3))^2);

    % differentiation of variable rou
    d_rou = [current_pos(1,1)-agenlain{kk}(1); current_pos(2,1)-
agenlain{kk}(2); ...
    current_pos(3,1)-agenlain{kk}(3)]/rou;

    % Threshold value to judge whether the UAV near to other agent or not?
    zeta = 1;

```

```

n = 2;

if rou<=zeta
    if robot_height > agenlain{kk}(3)
        goal=goalp;
        % Inter-agent repulsive force
        Frep1 = Kform*Krep*((1/rou)-
(1/zeta))*(1/rou^2)*dist_factor(current_pos, goal, n, flag)*d_rou;
        F(:,k+kk) = Frep1;
        F(3,k+kk) = 0;
    else
        F(:,k+kk) = 0;
    end
elseif (rou > zeta) && (ismember(kk, group))
    dd=[(current_pos(1,1)-agenlain{kk}(1));(current_pos(2,1)-
agenlain{kk}(2))];
    goal=agenlain{kk};
    F(:,k+kk)=vertcat(Katt*Kform*((1/rou)-(1/zeta))*dd/norm(dd),0);
    F(3,k+kk) = 0;
end
end

%% Calculate Total Forces
Frep_total = sum(F,2); % summation of all repulsive forces
Ft = Fatt + Frep_total*norm(Fatt);

end

%% Helper Functions
function Fatt = potential_attraction(Katt, current_pos, goal)
    % Calculate attractive potential field force
    Fatt = Katt * (goal - current_pos);
end

function factor = dist_factor(current_pos, goal, n, flag)
    % Distance factor for goal-aware repulsion
    d_goal = norm(goal - current_pos);
    if d_goal > eps
        factor = d_goal^n;
    else
        factor = 1;
    end
end

function diff_factor = diff_distance_factor(current_pos, goal, n, flag)
    % Derivative of distance factor
    d_goal = norm(goal - current_pos);
    if d_goal > eps
        diff_factor = n * d_goal^(n-1) * (goal - current_pos) / d_goal;
    else
        diff_factor = zeros(size(current_pos));
    end
end

%% VAPF Implementation
function [Ft, Fatt, Frep_total] = velocity_APF(j, target, pos, static,
goal, vcur, all_obs, all_radius, obs_speeds, gains_att, gains_rep, params)
    % Velocity-based APF considering relative velocities

    %% Attractive Force
    d_goal = norm(goal - pos);

    if d_goal > eps
        % Basic attractive force
        Fatt = gains_att.kp * (goal - pos);

        % Damping to prevent oscillation

```

```

Fatt = Fatt - gains_att.kdamp * vcur;

% Limit force near goal
if d_goal < params.R_STOP * 3
    scale = d_goal / (params.R_STOP * 3);
    Fatt = Fatt * scale;
end
else
    Fatt = zeros(3,1);
end

%% Velocity-based Repulsive Force
Frep_total = zeros(3,1);
d0 = 30; % Influence distance
collision_time_threshold = 5; % seconds

for k = 1:size(all_obs,2)
    p_obs = all_obs(:,k);
    r_obs = all_radius(k);
    v_obs = obs_speeds(:,k);

    % Relative position and velocity
    rel_pos = pos - p_obs;
    rel_vel = vcur - v_obs;

    if static == 1 && k <= size(all_obs,2)
        % Cylinder obstacle (static)
        dx = pos(1:2) - p_obs(1:2);
        dist_2d = norm(dx);

        if dist_2d < d0 && pos(3) < p_obs(3)
            % Check collision course
            if dist_2d > eps && dot(vcur(1:2), -dx) > 0
                % Moving toward obstacle
                time_to_collision = dist_2d / (norm(vcur(1:2)) +
eps);

                if time_to_collision < collision_time_threshold
                    % Velocity-based repulsion
                    urgency = 1 - time_to_collision /
collision_time_threshold;
                    Frep_k = gains_rep.krepp * urgency *
(1/(dist_2d-r_obs))^2;

                    if dist_2d > eps
                        Frep_k = Frep_k * [dx/dist_2d; 0];
                    else
                        Frep_k = [100; 100; 0];
                    end
                    Frep_total = Frep_total + Frep_k;
                end
            end
        end
    else
        % Sphere obstacle (possibly moving)
        dist = norm(rel_pos) - r_obs;

        if dist < d0 && dist > 0
            % Check if on collision course
            if norm(rel_vel) > eps && dot(rel_vel, -rel_pos) > 0
                % Calculate time to closest approach
                t_closest = -dot(rel_pos, rel_vel) /
(norm(rel_vel)^2);

                if t_closest > 0 && t_closest <
collision_time_threshold

```

```

        % Predicted closest distance
        closest_pos = rel_pos + t_closest * rel_vel;
        closest_dist = norm(closest_pos) - r_obs;

        if closest_dist < r_obs
            % Collision predicted
            urgency = 1 - t_closest /
collision_time_threshold;
            Frep_k = gains_rep.krepp * urgency *
(1/dist)^2;

            direction = rel_pos / norm(rel_pos);
            Frep_total = Frep_total + Frep_k *

direction;
        end
    end

    % Add basic repulsion for close obstacles
    if dist < r_obs * 2
        Frep_k = gains_rep.krepp * (1/dist - 1/d0) *
(1/dist)^2;
        direction = rel_pos / norm(rel_pos);
        Frep_total = Frep_total + Frep_k * direction;
    end
elseif dist <= 0
    % Emergency repulsion
    direction = rel_pos;
    if norm(direction) < eps
        direction = [1; 0; 0];
    else
        direction = direction / norm(direction);
    end
    Frep_total = Frep_total + 150 * direction;
end
end

%% Total Force
Ft = Fatt + Frep_total;

```


BIODATA PENULIS



Penulis dilahirkan ke dunia di kota Surabaya, 10 November 2003, merupakan anak pertama dari 2 bersaudara. Penulis telah menempuh pendidikan formal yaitu di SDN Ngagel Rejo 1, SMP Al-Hikmah Surabaya dan MAN Insan Cendekia Serpong. Setelah lulus dari MAN pada tahun 2021, Penulis mengikuti SNMPTN dan diterima di Departemen Teknik Elektro FTEIC - ITS pada tahun 2021 dan terdaftar dengan NRP 5022211013.

Selama berkuliah di Institut Teknologi Sepuluh Nopember, Penulis sempat aktif di beberapa kegiatan seperti IniLhoITS dan UKM Expo. Di Departemen Teknik Elektro Penulis sempat aktif di beberapa kegiatan yang diselenggarakan oleh Departemen, sebagai Kepala Biro Media Informasi di Himpunan Mahasiswa Teknik Elektro (HIMATEKTRO) dan aktif di Laboratorium Kontrol dan Otomasi sebagai Kepala Divisi Media, Asisten Praktikum Dasar Sistem Kontrol serta Asisten Praktikum Kontrol dan Figital Otomasi.