Deep Reinforcement Learning as Guidance for Aerospace Robotics

by

Kirk Charles Hovell

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Aerospace Engineering

Carleton University
Ottawa, Ontario

© 2022
Kirk Charles Hovell
This work is dedicated to my late friend Molly Milligan.
Abstract

The ability for a manipulator-equipped chaser spacecraft to autonomously capture a target spacecraft is an unsolved prerequisite for space debris removal and on-orbit servicing. This thesis investigates using deep reinforcement learning (DRL) to improve the capabilities of a manipulator-equipped chaser at this task. DRL allows for behaviour to be learned, rather than designed, according to a simple reward function. DRL uses trial-and-error to learn the behaviour, which is not feasible to perform on-board a spacecraft. Training must therefore be performed in simulation with the resulting behaviour transferred to the spacecraft. Transferring the learned-in-simulation behaviour to a real robot, however, is difficult due to dynamics differences between the simulator and the real world, i.e., the simulation-to-reality gap. This thesis develops, over the course of four increasingly-difficult applications, a solution to the simulation-to-reality gap by restricting DRL to exclusively learn the guidance portion of the guidance, navigation, and control system needed for autonomous spacecraft operations.

The first application is spacecraft proximity operations (without capture), where a DRL-based guidance strategy issuing desired velocity signals is designed, trained, and evaluated in simulation and experiment. Next, the DRL-based guidance strategy is improved upon and applied to a quadrotor proximity operations scenario. Here, it is demonstrated in simulation and experiment that desired acceleration signals lead to better performance compared to desired velocity signals. These two proof-of-concept results show the proposed DRL-based guidance strategy is viable for bringing DRL to real aerospace vehicles.

Next, the DRL-based guidance strategy is applied to a more difficult scenario: a multi-agent cooperative quadrotor runway inspection task, where fault-tolerant behaviour is successfully learned and demonstrated in both simulation and a real, outdoor, GPS-driven quadrotor facility.

Finally, with the now-developed DRL-based guidance strategy, the author returns to the central motivator for this research: autonomous manipulator-based capture of a
spinning spacecraft. The DRL-based guidance strategy learns this task in simulation and is successfully transferred to an experimental facility where similar results are obtained. Additionally, capture is successful in experiment despite large perturbations and initial conditions not seen during training. Improvements to the experimental facility were performed to enable this research.
Preface

This is an ‘integrated thesis’ that contains chapters which have already been published or have been prepared for publication as journal articles or conference proceedings.

Chapter 2 has been peer-reviewed and was published in the *Journal of Spacecraft and Rockets*, with the authors retaining copyright. The paper is included in this thesis with minor formatting changes and variable renaming (for consistency between chapters), along with improvements to the theory in Sec. 2.3. This paper was co-authored by the thesis author, Kirk Hovell, and his thesis supervisor, Prof. Steve Ulrich. Mr. Hovell set-up, developed, wrote, and evaluated the relevant code; obtained and analyzed simulated and experimental results; and wrote the manuscript. Theory development and manuscript editing were conducted jointly. It should be cited as:


Chapter 3 has been peer-reviewed and was presented in the *2021 AIAA Guidance, Navigation, and Control Conference*, with the authors retaining copyright. The paper is included in this thesis with minor formatting changes and variable renaming (for consistency between chapters). A theory section was removed to avoid repetition, with the reader being referred to Sec. 2.3. This paper was co-authored by the thesis author, Kirk Hovell, his thesis supervisor, Prof. Steve Ulrich, and a colleague, Prof. Murat Bronz. Mr. Hovell set-up, developed, wrote, and evaluated the relevant code; obtained and analyzed simulated results; prepared software for experimental demonstrations; analyzed experimental results; and wrote the manuscript. Theory development and manuscript editing were conducted jointly. Prof. Bronz was responsible for obtaining experimental results. It should be cited as:

Chapter 4 is under peer-review at the journal *Field Robotics*. The paper is included in this thesis with minor formatting changes and variable renaming (for consistency between chapters). A theory section was removed to avoid repetition, with the reader being referred to Sec. 2.3. This paper was co-authored by the thesis author, Kirk Hovell, his thesis supervisor, Prof. Steve Ulrich, and a colleague, Prof. Murat Bronz. Mr. Hovell set-up, developed, wrote, and evaluated the relevant code; obtained and analyzed simulated results; prepared software for experimental demonstrations; analyzed experimental results; and wrote the manuscript. Theory development and manuscript editing were conducted jointly. Prof. Bronz was responsible for obtaining experimental results. The following paper citation is preferred, and this thesis should only be cited if the appropriate citation for the included article cannot be found:


A video summary of the paper is available at https://youtu.be/Pu5rWnLgyZs.

Chapter 5 is under peer-review at the *Journal of Guidance, Control, and Dynamics*. The paper is included in this thesis with three minor modifications: 1) Eqs. (5.18) and (5.19) are included for completeness along with their elements defined in App. A; 2) minor formatting changes; and 3) a theory section was removed to avoid repetition, with the reader being referred to Sec. 2.3. This paper was co-authored by the thesis author, Kirk Hovell, and his thesis supervisor, Prof. Steve Ulrich. Mr. Hovell set-up, developed, wrote, and evaluated the relevant code; obtained and analyzed
simulated and experimental results; and wrote the manuscript. Theory development
and manuscript editing were conducted jointly. The following paper citation is pre-
ferred, and this thesis should only be cited if the appropriate citation for the included
article cannot be found:

Hovell, K., and Ulrich, S., “Spacecraft Robotic Capture and Simultaneous Stabi-
lization Using Deep Reinforcement Learning-based Guidance,” under peer-review at

A video summary of the paper is available at https://youtu.be/_oWpEH_dalo.

Statement of Student Involvement

I, Kirk Hovell, was fully involved and responsible for setting-up, developing, writing,
and evaluating the relevant code; obtaining and analyzing simulated results; obtain-
ing (with the exception of Chaps. 3 and 4) and analyzing experimental results; and
writing the manuscripts presented in this thesis.

Kirk Hovell

The student, Kirk Hovell, was fully involved and responsible for setting-up, devel-
oping, writing, and evaluating the relevant code; obtaining and analyzing simulated
results; obtaining (with the exception of Chaps. 3 and 4) and analyzing experimental
results; and writing the manuscripts presented in this thesis.

Steve Ulrich
Acknowledgements

I sincerely thank my supervisor, Professor Steve Ulrich. Without him, I would have never considered pursuing graduate studies, nor would I have fallen in love with research. I also would have missed out on many (pre-pandemic) squash games. He let me grow and explore my domain freely, providing thoughtful guidance when needed. Also, his establishment of the Spacecraft Robotics and Control Laboratory, along with his trust in me to take the lead on developing it, has helped significantly with my personal growth. His financial assistance was deeply appreciated. I’ve got a lifelong friend, colleague, and squash partner.

I also thank Professor Murat Bronz, who greatly impacted the course of my Ph.D. for the better after he approached me at a conference. His enthusiasm, flexibility, and dedication have been contagious. He remains extremely friendly despite my work causing his quadrotors to crash countless times.

I would also like to thank my family. My mom, Narda, my dad, Jeff, and my sister, Kelly, have all been incredibly supportive and curious throughout my studies. I can feel their love and excitement pushing me along, from 1,420 km away. My grandparents have supported me with mailed cookies; my Aunt Sonia with her handwritten letters. I thank my friends who were always eager for updates and problem solving. I also thank my cats, Homer and Gertie, for their support throughout the COVID-19 pandemic.

I invite readers to take a moment to acknowledge their own mental health; know that support is available if you need it.

This research was financially supported in part by the Natural Sciences and Engineering Research Council of Canada under the Postgraduate Scholarship-Doctoral PGSD3-503919-2017 and the Discovery Grant program. In addition, the Ontario
Graduate Scholarship, entrance scholarships to the Mechanical and Aerospace Engineering department, and Teaching Assistantships helped fund my research. Computations were enabled by support provided by Research Computing Services\(^1\) at Carleton University; specific thanks to Ryan Taylor for his help. Computations were also made on the supercomputer Béluga from Université du Québec, managed by Calcul Québec and Compute Canada. The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), Ministère de l’Économie et de l’Innovation du Québec (MEI) and le Fonds de recherche du Québec - Nature et technologies (FRQNT).

\(^1\)https://carleton.ca/rcs
# Table of Contents

Abstract iii  
Preface v  
Acknowledgements viii  
List of Tables xvi  
List of Figures xvii  
List of Symbols xxi  

Chapter 1  Introduction  
  1.1 Motivation 1  
  1.2 Problem Statement 2  
  1.3 Literature Review 2  
    1.3.1 Guidance and Control for On-orbit Servicing 3  
    1.3.2 Deep Reinforcement Learning for Aerospace Vehicles 4  
    1.3.3 Simulation-to-Reality Gap 7  
  1.4 Thesis Objective 8  
  1.5 Contributions 8  
  1.6 Thesis Overview 9  

Chapter 2  Deep Reinforcement Learning for Spacecraft Proximity Operations Guidance 11  
  2.1 Abstract 12  
  2.2 Introduction 12  
  2.3 Markov Decision Processes and Deep Reinforcement Learning 16  
  2.4 Deep Guidance 20  
  2.5 Problem Statement 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.1 Kinematics and Dynamics Models</td>
<td>23</td>
</tr>
<tr>
<td>2.5.2 Reward Function</td>
<td>24</td>
</tr>
<tr>
<td>2.5.3 Learning Algorithm Details</td>
<td>26</td>
</tr>
<tr>
<td>2.6 Simulation Results</td>
<td>27</td>
</tr>
<tr>
<td>2.6.1 Docking with a Stationary Target</td>
<td>27</td>
</tr>
<tr>
<td>2.6.2 Docking with a Spinning Target</td>
<td>30</td>
</tr>
<tr>
<td>2.6.3 Docking While Avoiding an Obstacle</td>
<td>32</td>
</tr>
<tr>
<td>2.7 Experimental Validation</td>
<td>36</td>
</tr>
<tr>
<td>2.7.1 Experiment Facility</td>
<td>36</td>
</tr>
<tr>
<td>2.7.2 Setup</td>
<td>38</td>
</tr>
<tr>
<td>2.7.3 Results</td>
<td>39</td>
</tr>
<tr>
<td>2.8 Conclusion</td>
<td>41</td>
</tr>
</tbody>
</table>

Chapter 3  Acceleration-based Quadrotor Guidance Under Time Delays Using Deep Reinforcement Learning  42

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Abstract</td>
<td>43</td>
</tr>
<tr>
<td>3.2 Introduction</td>
<td>43</td>
</tr>
<tr>
<td>3.3 Markov Decision Processes and Deep Reinforcement Learning</td>
<td>46</td>
</tr>
<tr>
<td>3.4 Problem Statement</td>
<td>46</td>
</tr>
<tr>
<td>3.4.1 Deep Guidance</td>
<td>47</td>
</tr>
<tr>
<td>3.4.2 Kinematics Model</td>
<td>49</td>
</tr>
<tr>
<td>3.4.3 Time Delay Consideration</td>
<td>49</td>
</tr>
<tr>
<td>3.4.4 Dynamics Model</td>
<td>50</td>
</tr>
<tr>
<td>3.4.5 Reward Function</td>
<td>51</td>
</tr>
<tr>
<td>3.4.6 Learning Algorithm Implementation Details</td>
<td>53</td>
</tr>
<tr>
<td>3.5 Simulation Results</td>
<td>55</td>
</tr>
<tr>
<td>3.5.1 Velocity-based Guidance Results</td>
<td>55</td>
</tr>
<tr>
<td>3.5.2 Acceleration-based Guidance Results</td>
<td>56</td>
</tr>
<tr>
<td>3.6 Experimental Validation</td>
<td>60</td>
</tr>
<tr>
<td>3.6.1 Experiment Facility</td>
<td>60</td>
</tr>
</tbody>
</table>
Chapter 5  Spacecraft Robotic Capture and Simultaneous Stabilization Using Deep Reinforcement Learning-based Guidance

5.1 Abstract .................................................... 104
5.2 Introduction ................................................ 104
5.3 Markov Decision Processes and Deep Reinforcement Learning ............ 107
5.4 Problem Statement .......................................... 107
  5.4.1 Assumptions and Improvements ...................... 108
  5.4.2 Deep Reinforcement Learning-based Guidance ......... 109
  5.4.3 Kinematics ............................................. 110
  5.4.4 Dynamics ............................................... 113
  5.4.5 Post-capture Angular Momentum ...................... 114
  5.4.6 Reward Function ..................................... 116
  5.4.7 Learning Algorithm Implementation Details ........... 118
5.5 Training in Simulation .................................... 119
  5.5.1 Control ................................................ 120
  5.5.2 Training Results ..................................... 121
  5.5.3 Simulation Example .................................. 123
  5.5.4 Policy Analysis ...................................... 123
5.6 Experimental Validation .................................. 126
  5.6.1 Experiment Facility .................................. 127
  5.6.2 Setup ............................................... 129
  5.6.3 Experimental Results ................................ 132
  5.6.4 Perturbations ....................................... 134
  5.6.5 Analysis ............................................. 134
5.7 Conclusion ............................................... 136
## Appendix D  Reinforcement Learning

| D.1 Markov Decision Processes and Deep Reinforcement Learning | 197 |
| D.2 Model-based Methods | 197 |
| D.3 Model-free Methods | 198 |
| D.3.1 Value Function Methods | 198 |
| D.3.2 Policy Gradient Methods | 201 |
| D.4 Benchmarking | 203 |
| D.5 Other Learning Methods | 203 |
| D.5.1 Imitation Learning | 203 |
| D.5.2 Iterative Learning Control | 204 |

## Appendix E  Example: Reinforcement Learning on Two-link Manipulator

| E.1 Deep Reinforcement Learning Background | 206 |
| E.2 DDPG Algorithm | 209 |
| E.3 Dynamics | 213 |
| E.4 Reward Function | 215 |
| E.5 Simulations | 216 |
| E.6 Robustness | 221 |
| E.7 Conclusion | 223 |

## Biographical Sketch

224
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>General characteristics for the Tawaki v1.0 autopilot board</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>General characteristics for the Tawaki autopilot board</td>
<td>93</td>
</tr>
<tr>
<td>5.1</td>
<td>Dynamics parameters</td>
<td>120</td>
</tr>
<tr>
<td>5.2</td>
<td>Position, velocity, and acceleration limits</td>
<td>120</td>
</tr>
<tr>
<td>5.3</td>
<td>Initial conditions and their uniform randomization</td>
<td>120</td>
</tr>
<tr>
<td>B.1</td>
<td>Chaser and target mass and inertia properties under different configurations</td>
<td>173</td>
</tr>
<tr>
<td>C.1</td>
<td>AND gate truth table</td>
<td>187</td>
</tr>
<tr>
<td>C.2</td>
<td>Outputs of initialized neural network</td>
<td>188</td>
</tr>
<tr>
<td>C.3</td>
<td>Calculating loss on initialized neural network</td>
<td>190</td>
</tr>
<tr>
<td>C.4</td>
<td>Calculating predictions and loss after one training iteration</td>
<td>191</td>
</tr>
<tr>
<td>C.5</td>
<td>Calculating predictions and loss after 100 training iterations</td>
<td>192</td>
</tr>
<tr>
<td>C.6</td>
<td>Calculating predictions and loss after 1,000 training iterations</td>
<td>192</td>
</tr>
<tr>
<td>D.1</td>
<td><em>n</em>-armed bandit lever statistics</td>
<td>199</td>
</tr>
<tr>
<td>D.2</td>
<td>Action-value table after 1 lever pull</td>
<td>200</td>
</tr>
<tr>
<td>D.3</td>
<td>Action-value table after 5 lever pulls</td>
<td>200</td>
</tr>
<tr>
<td>D.4</td>
<td>Action-value table after 100 lever pulls</td>
<td>200</td>
</tr>
<tr>
<td>D.5</td>
<td>Action-value table after 10,000 lever pulls</td>
<td>201</td>
</tr>
<tr>
<td>E.1</td>
<td>Manipulator physical parameters</td>
<td>216</td>
</tr>
<tr>
<td>E.2</td>
<td>Reinforcement learning hyperparameters</td>
<td>216</td>
</tr>
<tr>
<td>E.3</td>
<td>Stationary target initial conditions</td>
<td>217</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Policy and value neural networks used in the D4PG algorithm. 18
2.2 Proposed deep guidance strategy. 21
2.3 Proposed deep guidance with an ideal controller for training purposes in simulation. 22
2.4 Spacecraft pose tracking and docking task. 23
2.5 Training performance for chaser pose tracking and docking for a stationary target with constant initial conditions. 28
2.6 Visualization of chaser trajectories at various episodes during the training process. 29
2.7 Training performance for chaser pose tracking and docking for a stationary target with randomized initial conditions. 30
2.8 Examples of learned chaser trajectories with randomized initial conditions. 31
2.9 Training performance for chaser pose tracking and docking with a rotating target. 32
2.10 Examples of learned chaser trajectories with a rotating target. 33
2.11 Training performance for chaser pose tracking and docking while avoiding an obstacle. 34
2.12 Examples of learned chaser trajectories while avoiding an obstacle. 35
2.13 The Spacecraft Proximity Operations Testbed (SPOT). 37
2.14 Experimental trajectories of deep guidance in Carleton’s Spacecraft Proximity Operations Testbed. 40
3.1 Velocity-based deep guidance strategy [1]. 48
3.2 Acceleration-based deep guidance strategy. 48
3.3 Deep guidance with an ideal controller for training purposes in simulation. 48
3.4 Velocity-based deep guidance training progress. 56
3.5 Visualization of velocity-based follower trajectories at various episodes once training is complete. ........................................... 57
3.6 Acceleration-based deep guidance training progress. ................. 58
3.7 Visualization of acceleration-based follower trajectories at various episodes once training is complete. ................................. 59
3.8 Time-series scalar follower position error. .................................. 60
3.9 ENAC facility and flight hardware. ........................................ 61
3.10 Experimental results. .......................................................... 64
3.11 Hexacopter hardware and experimental trajectory. .................... 65
4.1 Runway shapes and inspection grids. ....................................... 74
4.2 Block-scheme diagrams of the deep guidance strategies used in experiment (with a real controller) and in training (with an ideal controller assumed). .......................................................... 75
4.3 Non-rectangular areas to inspect. .......................................... 83
4.4 The hand-crafted zig-zag approach. ....................................... 83
4.5 Indoor single-quadrotor runway deep guidance training progress. 85
4.6 Indoor simulated runway inspection results. ............................. 86
4.7 Outdoor single- and dual-quadrotor runway deep guidance training progress. ................................................................. 87
4.8 Visualization of sample trajectories from the outdoor simulated training scenarios. Quadrotors are enlarged to show their current position. ................................................................. 88
4.9 Learning to explore non-rectangular shapes. ............................ 89
4.10 Visualization of sample trajectories from the outdoor simulated non-rectangular area training scenarios. Quadrotors are enlarged to show their current position. ....................... 89
4.11 Training results with higher numbers of quadrotors. The exponential smoothing factor is increased to 0.99 for clarity. .......... 90
4.12 ENAC indoor facility and flight hardware. .............................. 92
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13</td>
<td>ENAC outdoor facility and flight hardware.</td>
</tr>
<tr>
<td>4.14</td>
<td>Communication between the various segments in the outdoor experiments. Dashed arrows between the quadrotor and ground computer represent wireless communication using XBee. Indoor experiments obtained $x$ and $v$ from a ground truth system instead of an on-board GPS module.</td>
</tr>
<tr>
<td>4.15</td>
<td>Indoor experimental results.</td>
</tr>
<tr>
<td>4.16</td>
<td>Outdoor experimental results.</td>
</tr>
<tr>
<td>5.1</td>
<td>The robotic capture and stabilization task to be learned. The chaser and manipulator are drawn in red and the target is drawn in gray.</td>
</tr>
<tr>
<td>5.2</td>
<td>Deep reinforcement learning-based guidance strategy.</td>
</tr>
<tr>
<td>5.3</td>
<td>Chaser and target reference frame and state notation.</td>
</tr>
<tr>
<td>5.4</td>
<td>Training progression in simulation.</td>
</tr>
<tr>
<td>5.5</td>
<td>Frames during the simulated capture scenario with a target spin of $6.8^\circ/s$.</td>
</tr>
<tr>
<td>5.6</td>
<td>Acceleration signals and angular rates corresponding to the motion shown in Fig. 5.5.</td>
</tr>
<tr>
<td>5.7</td>
<td>Understanding the learned policy. Red lines show example trajectories generated from following the guidance field.</td>
</tr>
<tr>
<td>5.8</td>
<td>The Spacecraft Proximity Operations Testbed.</td>
</tr>
<tr>
<td>5.9</td>
<td>Snapshots of three experimental trajectories.</td>
</tr>
<tr>
<td>5.10</td>
<td>Angular rate over time for three sets of initial conditions.</td>
</tr>
<tr>
<td>5.11</td>
<td>Snapshots of two perturbation experiments.</td>
</tr>
<tr>
<td>B.1</td>
<td>Mass property measurements with digital scales.</td>
</tr>
<tr>
<td>B.2</td>
<td>Moment of inertia measurements with a bifilar pendulum.</td>
</tr>
<tr>
<td>B.3</td>
<td>Thruster numbering, centre of mass offset, and variable definition.</td>
</tr>
<tr>
<td>B.4</td>
<td>Flow diagram on SPOT.</td>
</tr>
<tr>
<td>B.5</td>
<td>Experimentally-determined thrust decay.</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>C.1</td>
<td>A feedforward neural network with two hidden layers and 3 neurons in each layer.</td>
</tr>
<tr>
<td>C.2</td>
<td>One-neuron example neural network used to model an AND gate.</td>
</tr>
<tr>
<td>C.3</td>
<td>Computation graph for neural network used in the AND gate example.</td>
</tr>
<tr>
<td>C.4</td>
<td>Computation graph for neural network including the loss function.</td>
</tr>
<tr>
<td>C.5</td>
<td>Trained neural network, representing an AND logic gate.</td>
</tr>
<tr>
<td>D.1</td>
<td>Block scheme diagram of a policy gradient algorithm.</td>
</tr>
<tr>
<td>E.1</td>
<td>Block diagram of the DDPG algorithm.</td>
</tr>
<tr>
<td>E.2</td>
<td>Planar two-link manipulator.</td>
</tr>
<tr>
<td>E.3</td>
<td>Agent performance during training.</td>
</tr>
<tr>
<td>E.4</td>
<td>Learning curve with randomized initial conditions.</td>
</tr>
<tr>
<td>E.5</td>
<td>Training progress with simplified manipulator dynamics.</td>
</tr>
<tr>
<td>E.6</td>
<td>Performance as a function of link length.</td>
</tr>
</tbody>
</table>
List of Symbols

\( \alpha \) = policy network learning rate
\( A \) = action space
\( a \) = action
\( a, b \) = manipulator length properties, m
\( B \) = number of value distribution bins
\( \beta \) = value network learning rate
\( C \) = Coriolis matrix
\( d_t \) = docking port position on target, m
\( D \) = dynamics delay, number of time steps
\( e \) = error
\( E \) = episode number
\( E \) = expectation
\( \epsilon \) = weight-smoothing parameter
\( f \) = reward field
\( F \) = force, N
\( g \) = activation function
\( h \) = angular momentum, kg\( \cdot \)m\(^2\)/s
\( I \) = moment of inertia, kg\( \cdot \)m\(^2\)
\( I \) = identity matrix
\( \gamma \) = discount factor for future rewards
\( J \) = total expected rewards
\( J \) = Jacobian matrix
\( K \) = number of actors
\( K \) = reward weighting
\( K_p \) = proportional gain matrix
\( K_I \) = integral gain matrix
\( l \) = linear momentum, kg\( \cdot \)m/s
$L$ = loss function

$m$ = mass, kg

$M$ = mass matrix

$M$ = mini-batch size

$N$ = N-step return length

$N$ = normal distribution

$\eta, c_1$ = constants

$o$ = observation

$O$ = observation space

$p$ = position, m

$\pi_\theta$ = policy neural network with parameters $\theta$

$\phi'$ or $\theta'$ = exponentially-smoothed versions of $\phi$ or $\theta$

$\phi, q, \psi$ = angle, deg

$Q$ = number of quadrotors

$r$ = reward

$R$ = replay buffer size

$S$ = runway state

$\sigma$ = exploration noise standard deviation

$T$ = kinetic energy, J

$u$ = control effort

$v$ = velocity, m/s

$\omega$ = angular rate, rad/s

$x$ = state

$x$ = position in the x direction, m

$X$ = state space

$Y$ = target value distribution

$y$ = position in the y direction, m

$Z_\phi$ = value neural network with parameters $\phi$
Subscripts

c = chaser

cm = chaser-manipulator combination

cmt = chaser-manipulator-target combination

e = end-effector

n = time step number

o = obstacle

t = target

1, 2, 3 = manipulator joint numbers

Superscripts

d = desired

f = follower

i = the \( i^{th} \) quadrotor

t = target
Chapter 1

Introduction

1.1 Motivation

As of 2021, there were more than 28,000 objects larger than 10 cm in orbit around the Earth, but only 14% of them were operational spacecraft [2]. The defunct spacecraft that remain in orbit are named space debris. Due to their extremely high speeds, a collision between two pieces of space debris, or between a piece of space debris and a functional satellite, will cause both objects to break into many more pieces of debris, increasing the chance of further collisions. Leaving non-operational spacecraft in orbit was not considered a problem until 1978 when Kessler studied the collision frequency of spacecraft and found that while the density of debris in low Earth orbit is small, the collision frequency is non-zero and the debris population indeed has exponential growth; this phenomenon was later called the Kessler Syndrome [3]. A study by the European Space Agency revealed that five to ten properly-selected pieces of space debris must be actively removed per year for the current number of space debris to remain stable [4].

On-orbit servicing is an emerging industry that has the potential to transform the space economy and enable the sustainable use of space. On-orbit servicing refers to a servicer spacecraft refuelling, transporting to a new orbit, performing station keeping for, or providing hardware upgrades to a target spacecraft. This has the potential to dramatically extend spacecraft lifetimes and reduce their replacement frequency.

A prerequisite for space debris removal and on-orbit servicing to become a reality is the ability for a chaser spacecraft to rendezvous with and robotically capture a target satellite. In addition, this rendezvous and capture must be performed autonomously, since it is infeasible to always have an astronaut on-board or to have the capture remotely-operated from the ground, especially when the capture is being performed in geosynchronous orbit (GEO) or beyond due to signal delays. Most previous instances
of in-space capture have been manually-piloted, such as the Space Shuttle docking and servicing missions [5] and Apollo lunar rendezvous [6]. Despite a few proof-of-concept demonstrations of autonomous cooperative capture with ETS-VII in 1997 [7] and Orbital Express in 2008 [8], no wide-scale adoption of on-orbit servicing or robotic capture manoeuvres have occurred. This is due to: 1) the inherent risk associated with operating a manipulator-equipped spacecraft in close proximity to an uncooperative (and possibly tumbling) target; and 2) the requirement for autonomous operations. There is indeed a large technological gap between current single-use spacecraft paradigm and the on-orbit servicing future.

1.2 Problem Statement

A key enabling-technology for future robotic capture manoeuvres of tumbling objects in space is the need to determine, in real time, the desired path the chaser spacecraft and its manipulator should follow, i.e., guidance. Indeed, because of the time scale and separation distances involved, as well as the level of precision and accuracy required, autonomous real-time guidance is necessary. In this context, this thesis is concerned with the development of innovative guidance techniques that enable the autonomous robotic capture of uncooperative spinning targets.

1.3 Literature Review

First, some definitions:

- **Navigation** - the act of determining the state (positions, orientations, etc.) of a system. In simulation, the state is often fully known (i.e., perfect navigation is assumed); in experiment, sensors (cameras, GPS, radar, etc.) are needed to perform navigation.

- **Guidance** - the act of choosing a path to follow (using the navigation data) in order to complete a task or mission. This may take the form of desired position waypoints, desired velocities, or desired accelerations.
• **Control** - the act of calculating how much effort (thrust, torque, etc.) needs to be applied for the robot to track the guided (desired) path. Knowledge of the robot’s current state is obtained from the Navigation.

Guidance, Navigation, and Control (GNC) are three key elements in any robotic system, and their definitions will help in understanding the previous work in the field.

1.3.1 **Guidance and Control for On-orbit Servicing**

For a manipulator-equipped chaser spacecraft to capture an uncooperative spinning target, it must perform two tasks:

1. Target capture: the chaser must actually capture the target by moving itself around the target to enable its robotic manipulator to gently engage the target’s docking port; then

2. Stabilize the system: after capture, the chaser must then stabilize the combined chaser-manipulator-target system.

There are two main approaches to perform the first task (target capture): a) bring the chaser to rest near the target, stop actuating the chaser base, then actuate the manipulator to capture the target in a way that minimizes disturbance to the unactuated spacecraft base [9–12]; or b) actuate the chaser spacecraft base and robotic manipulator simultaneously. The latter approach is more flexible, and is more suitable to capturing uncooperative rotating targets, since the chaser can move as needed to avoid time-varying keep out zones. Optimization-based methods have been used to investigate manipulator-based capture with an actuated chaser [13–17]. Model-predictive control methods [18] and sample-based methods [19] have also been applied to the problem. After capture, the second task must now be executed (stabilize the system). Minimum-time formulations have been used [13, 20], among other advanced control methods [21–24].

Recently, Virgili-Llop and Romano [25] developed an optimization-based guidance method that performs both tasks (capture and stabilization) simultaneously. It does
so by considering both the capture and stabilization requirements in its optimization scheme. The result is a real-time guidance strategy that moves the chaser and its manipulator to capture the target with equal and opposite angular momentum (about the combined centre of mass of the post-capture system) such that upon capture, no residual angular momentum exists in the system. The authors demonstrated their approach can be executed on low-powered hardware, and in real-time, in the Space-craft Robotics Lab at the Naval Postgraduate School. While the work successfully demonstrated the capture and stabilization of a target spacecraft using a robotic manipulator, several assumptions were invoked to make the optimization tractable, such as: 1) prescribed capture times; 2) prescribed capture orientations; 3) constant chaser initial conditions; and 4) pre-set manipulator motion in the final seconds before capture. This work is impressive, although a fully-autonomous approach should be able to capture and stabilize a spinning target without these simplifications.

Learning-based approaches are a proposed alternative to traditional GNC techniques that do not rely upon as-strict assumptions for optimal behaviour to be realized. Of the many learning-based approaches available, deep reinforcement learning is most appropriate for the spacecraft capture problem, as discussed below.

### 1.3.2 Deep Reinforcement Learning for Aerospace Vehicles

Deep Reinforcement Learning (DRL) is a branch of artificial intelligence (AI) that has rapidly progressed in recent years due to a rise in available computing power and the development of new learning algorithms. DRL is a very general learning strategy where, starting from scratch, behaviours are learned in a reward-seeking manner. If rewards are given for completing a task, the agent will learn to solve the task via maximizing the rewards it receives. Using trial-and-error, over (potentially) millions of attempts, the algorithm learns which actions, from a given state, maximize the rewards received. DRL-based approaches give the agent the flexibility to discover this reward-seeking behaviour on its own, instead of the behaviour being hand-crafted by a designer. In addition, the agent may discover behaviour that outperforms a hand-crafted solution. The first major demonstration of this “agent learns best” notion was in a 2015 paper where DRL was used to learn, entirely on its own, how to play
a large variety of Atari video games at a super-human level [26]. Since this inaugural paper, numerous DRL algorithms and applications have been rapidly developed: the world champion at the game of Go was defeated using DRL in 2016 [27], followed by StarCraft II in 2019 [28]. In 2021, the protein folding problem (a grand challenge in biology) was solved using machine learning techniques [29].

Deep reinforcement learning can be applied to problems formulated as a Markov Decision Process (MDP). A MDP is any discrete-time process where the state vector contains all relevant information about the system (i.e., processes that do not require time histories to fully describe the current state). The DRL algorithm uses the state to calculate an action which is then executed on the environment. The environment is propagated forward one time step and the next state is returned. A scalar reward is also returned at each time step; the reward feedback is used by the DRL algorithm to learn the intended behaviour. Robotics tasks are good candidates to be formulated as MDPs to which DRL can be readily applied since the dynamical state fully describes the system. A number of different DRL algorithms have been developed, each with a particular structure and use case. Model-free policy gradient algorithms with the actor-critic architecture are applicable to robotics, since they can handle continuous state and action information. Actor-critic algorithms use two neural networks to learn behaviour that maximizes the rewards received. For background on neural networks and reinforcement learning, see Appendices C and D, respectively.

Deep reinforcement learning has been cited as a major influence in the future of aerospace engineering [30, 31], with applications to the guidance, navigation, and control aspects of autonomous robotics. For navigation, DRL has been used to map small asteroid bodies in simulation [32]. For control, reinforcement learning was used to replace a controller for the Apollo transposition and docking manoeuvre [33]. Model-based reinforcement learning was used to generate a dynamics model of a quadrotor, used in conjunction with optimal control, in experiment [34]. Others combined a DRL-based controller with classical proportional-derivative (PD) control in experiment [35]. Reinforcement learning was shown to outperform some classical control approaches in simulation [36].
DRL approaches have been used as guidance for multi-agent uncrewed aerial vehicle wildfire detection [37] and planetary landing in [38] (where guidance and navigation were combined) and [39] (where guidance and control were combined). Federici et al. also studied the use of behaviour cloning (BC) and reinforcement learning to perform guidance and control for spacecraft proximity operations [40]. Others combined guidance and control for planar spacecraft rendezvous [41].

Applying DRL to manipulator-based spacecraft to enable the capture of target spacecraft has received considerable attention in recent months, with two main papers being published in this area. First, DRL was used to issue guidance velocity commands that manoeuvre a dual-arm spacecraft to capture a target spacecraft. The base spacecraft was uncontrolled; only the manipulator joints were actuated. In simulation, this technique was able to capture both stationary and translating targets [42]. The second paper improved the approach by including manipulator collision-avoidance and velocity constraints to the problem. In addition, capture of a spinning target was demonstrated [43]. These results are preliminary: the criterion for capture was when the end-effector was within 0.2 m of the target (end-effector angle was ignored) and the spacecraft base was unactuated; however, it represents an introduction to DRL for manipulator-based capture.

Unless explicitly stated otherwise, all previous work has been in simulation only. However, for aerospace domains, DRL will ultimately have to be applied to real hardware. Aerospace applications for DRL have a unique requirement: no training is allowed on-board the hardware. A spacecraft cannot spend time, fuel, or compute power attempting to capture a target many times in orbit until it gets it right, for example. Despite modern quick-learning DRL algorithms [44,45] (requiring hundreds or thousands of attempts, down from millions with older DRL algorithms), on-board learning is still unacceptable for aerospace domains. This presents a problem, however, because problems arise from training DRL algorithms exclusively in simulation and transferring that learned behaviour to a real robot.
1.3.3 Simulation-to-Reality Gap

DRL algorithms are motivated to learn behaviour that collects as many rewards as possible. When training is performed in simulation, as it is typically done, this can lead to the DRL algorithm exploiting the simulation in such a way that the learned behaviour is excellent in simulation (with the simulated dynamics), but performs poorly when transferred to the real world (with slightly different dynamics). This is known as the *simulation-to-reality* gap [31, 46].

Currently, some simulation-to-reality transfer has been performed by training the behaviour in simulation and using real-world data to fine-tune the performance in experiment. A trained-in-simulation policy network was used alongside a second trained-in-experiment neural network that modelled the dynamics present in the real world. Then, the policy was adapted with respect to the dynamics model [47, 48]. Other work used real-world images to train a neural network to adapt simulated navigation images to appear more like real-world images, which led to a 50 times reduction in the number of real-world images to achieve acceptable performance [49]. Other researchers have attempted decomposing neural network policies into “task-specific” and “robot-specific” segments [50]. The robot-specific segment learns an individual robot’s dynamics (which could be done in experiment, though this work was performed entirely in simulation) while the task-specific segment learns the high-level behaviour needed to solve the task. Once trained, the policy segments can be mixed and matched together (i.e., any robot-specific segment could be combined with any task-specific segment to accomplish the behaviour on any robot). Others developed a quadrotor stabilization task that summed the DRL commands with a conventional PD controller to guide the learning process and help with the simulation-to-reality transfer [35]. All these approaches, however, require real-world data which make them infeasible for spacecraft applications.

Another method for bridging the simulation-to-reality gap is domain randomization. Domain randomization modifies the simulated dynamics parameters during each episode, such that the agent learns to generalize its behaviour to a range of dynamics parameters. Then, when the behaviour is transferred to the real world, the agent will be robust to whatever dynamics parameters it encounters. Domain randomization
has enabled real quadrotor flight after being trained entirely in simulation [51], has enabled drone racing [52], the manipulation of objects with a robotic hand [53], and others [54–58]. However, domain randomization significantly increases the training time since the policy must learn to become adept at all dynamic variations of the task. In addition, because the DRL algorithm has to learn all dynamic variations of the task, it may not become an expert in any single variation.

In most of the previously-mentioned approaches, DRL was used to learn an entire guidance, navigation, and control system (i.e., accept sensor inputs and calculate control efforts). However, it is possible that the poor simulation-to-reality transfer occurs in part due to the DRL agent overfitting a controller in simulation that is no longer valid on the real robot hardware. Restricting reinforcement learning to only learn guidance was suggested by Harris et al. [59] in 2019. They argued that reinforcement learning should not be tasked with learning both guidance and control since control theory already has great success.

1.4 Thesis Objective

The objective of this thesis is to modify the current paradigm of DRL (where it learns the entire GNC suite) and instead use DRL as a guidance-only system to: a) cross the simulation-to-reality gap; and b) enable the autonomous robotic capture of a spinning target spacecraft. To develop this DRL-for-guidance approach, a series of applications with increasing complexity will be used. This thesis introduces DRL-based guidance techniques to the aerospace GNC community.

1.5 Contributions

The contributions of this work to the aerospace robotics community are:

- A deep reinforcement learning-based guidance algorithm for aerospace vehicles that enables state-of-the-art behaviour and transfers well from simulation to reality. The algorithm is developed over four papers.

- Experimental demonstrations of deep reinforcement learning techniques being applied to quadrotors in a cooperative and fault-tolerant way.
• The first, to the best of the author’s knowledge, experimental demonstration of applying artificial intelligence techniques to a spacecraft platform in experiment for both proximity operations and manipulator-based capture.

• Improvements to the Spacecraft Proximity Operations Testbed laboratory at Carleton University, consisting of: 1) a complete mass property determination; and 2) an improved control allocation strategy that accounts for both the manipulator’s real-time orientation and thruster pressure drop. Details of this work can be found in App. B.

The desire for an “off the shelf” deep reinforcement learning algorithm for use in robotics is strong [31]; this research moves the community one step closer to that goal.

1.6 Thesis Overview

The ultimate goal of improving the manipulator-based capture of spinning spacecraft is developed over four papers, where each paper is included as a separate chapter. This thesis is organized as follows:

Chapter 2 first develops a deep reinforcement learning-based guidance approach for spacecraft proximity operations. It restricts deep reinforcement learning to exclusively learn the guidance (path planning) portion of the guidance, navigation, and control system. The DRL-based guidance algorithm issues desired velocity commands to a controller that tracks the desired velocity signal. Perfect navigation is assumed for all work. The DRL velocity-based guidance system is trained entirely in simulation for a spacecraft proximity operations task, where a chaser spacecraft has to approach a spinning target while avoiding a nearby obstacle, and is successfully transferred to a hardware experiment at Carleton University’s Spacecraft Proximity Operations Testbed.

Chapter 3 improves upon the DRL velocity-based guidance technique, developed in Chap. 2, by proposing that acceleration-based guidance signals are more appropriate. A comparison between velocity- and acceleration-based DRL guidance approaches is performed—both in simulation and in a quadrotor experimental facility.
at École Nationale de l’Aviation Civile. It was determined that acceleration-based guidance approaches are indeed a better strategy.

Chapter 4 further develops the acceleration-based DRL guidance strategy, by applying it to a multi-agent cooperative quadrotor runway exploration scenario. The DRL guidance technique is trained to be tolerant to both signal delays and (perceived) hardware failures present in the experimental facility. The result is a successful demonstration of an entirely trained-in-simulation application of deep reinforcement learning to solve a difficult multi-agent cooperative runway inspection task. Experiments were performed in an outdoor GPS-driven runway facility at École Nationale de l’Aviation Civile.

Chapter 5, armed with the now fully-developed DRL acceleration-based guidance technique, returns to the core motivation of this work: to improve the state-of-the-art of manipulator-based capture of spinning spacecraft, using artificial intelligence techniques, to enable on-orbit servicing and space debris removal missions. Simulated results show excellent manipulator-based spacecraft capture, which is confirmed with experimental results at Carleton University’s Spacecraft Proximity Operations Testbed. The system’s tolerance to initial conditions and perturbations not seen during training is investigated with promising results.

Finally, Chapter 6 summarizes the main conclusions of this thesis, discusses consequences of this work, and provides potential avenues for future work.
Chapter 2

Deep Reinforcement Learning for Spacecraft Proximity Operations Guidance

This chapter has been peer-reviewed and was published in the Journal of Spacecraft and Rockets. It should be cited as:


The paper is included in this thesis with minor formatting changes and variable renaming (for consistency between chapters), along with improvements to the DRL theory in Sec. 2.3. This paper was co-authored by the thesis author, Kirk Hovell, and his thesis supervisor, Prof. Steve Ulrich. Mr. Hovell set-up, developed, wrote, and evaluated the relevant code; obtained and analyzed simulated and experimental results; and wrote the manuscript. Theory development and manuscript editing were conducted jointly.

Paper Context

This first paper introduces deep reinforcement learning to the spacecraft robotics community. It considers the spacecraft guidance problem as a decision-making problem to which deep reinforcement learning is applied. The decision taken at each time step is: given the state of the system, what is the desired velocity the robot should follow to accomplish the task? A spacecraft proximity operations with obstacle avoidance proof-of-concept task, in simulation and experiment, is used to test this approach.
2.1 Abstract

This paper introduces a guidance strategy for spacecraft proximity operations which leverages deep reinforcement learning, a branch of artificial intelligence. This technique enables guidance strategies to be learned rather than designed. The learned guidance strategy feeds velocity commands to a conventional controller to track. Control theory is used alongside deep reinforcement learning in order to lower the learning burden and facilitate the transfer of the learned behaviour from simulation to reality. In this paper, a proof-of-concept spacecraft pose tracking and docking scenario is considered, in simulation and experiment, to test the feasibility of the proposed approach. Results show that such a system can be trained entirely in simulation and transferred to reality with comparable performance.

2.2 Introduction

Autonomous spacecraft rendezvous and docking operations have become an active research area in recent decades. Applications, such as on-orbit servicing, assembly, and debris capture [60] require the capability for a chaser spacecraft to autonomously and safely maneuver itself in proximity to a potentially uncooperative target object. A common strategy is pose tracking, i.e., synchronizing the translational and rotational motion of the chaser with respect to the target, such that there is no relative motion between the two objects. Only then does the chaser perform its final approach and capture or dock with the target. Guidance and control algorithms have been developed for this purpose. For example, a guidance and control scheme for capturing a tumbling debris with a robotic manipulator was developed by Aghili [61]. Wilde et al. [62] developed inverse dynamics models to generate guidance paths and included experimental validation. Ma et al. [63] applied feed-forward optimal control for orienting a chaser spacecraft at a constant relative position with respect to a tumbling target. A variety of dual quaternion approaches have been explored [64–66]. Pothen and Ulrich [67,68] used the Udwadia-Kalaba equation to formulate the close-range rendezvous problem and included experimental validation. Lyapunov vector fields have been used to command docking with an uncooperative target spacecraft.
by Hough and Ulrich [69].

The currently-developed guidance and control techniques presented above are hand-crafted to solve a particular task and require significant engineering effort. As more complex tasks are introduced, the engineering effort needed to hand-craft solutions may become infeasible. For example, developing a guidance law for a chaser spacecraft to detumble a piece of spinning space debris, when the two objects are connected via flexible tethers, does not have a clear solution that can be hand-crafted. Motivated by more difficult guidance tasks, this paper introduces a new approach that builds upon a branch of artificial intelligence, called deep reinforcement learning, to augment the guidance capabilities of spacecraft for difficult tasks.

Reinforcement learning is based on the idea of an agent trying to choose actions in order to maximize the rewards it receives over a period of time. The agent uses a policy that, when given an input, returns a suggested action to take. At each timestep, a scalar reward, which may be positive or negative, is given to the agent and corresponds to task completion. Through trial-and-error, the agent attempts to learn a policy that maps inputs to actions, such that the actions taken maximize the rewards received. By selecting an appropriate reward scheme, complex behaviours can be learned by the agent without being explicitly programmed. For a tethered space debris detumbling task, for example, rewards could be given for reducing the angular velocity of the debris and penalties could be given for fuel usage, forcing the agent to learn a fuel-efficient detumbling policy. The engineering effort is reduced to specifying the reward system rather than the complete logic required to complete the task; this is the main appeal of reinforcement learning. Neural networks have become a popular choice for representing the policy in reinforcement learning as they are universal function approximators [70]. When neural networks are used within reinforcement learning, the technique is called deep reinforcement learning. The core concepts in reinforcement learning have been around for decades, but have only become useful in recent years due to the rapid rise in computing power. Many notable papers have been published recently that use deep reinforcement learning to solve previously-unsolvable tasks. In 2015, Mnih et al. [26] applied deep reinforcement learning to play many Atari 2600 [71] games at a superhuman level by training a policy to select
button presses (the action) as a function of the screen pixels (the input). Silver et al. [27, 72] used deep reinforcement learning to master the game of Go in 2016, a decade earlier than expected.

Training deep reinforcement learning policies on physical robots is time consuming, expensive, and leads to significant wear-and-tear on the robot because, even with state-of-the-art learning algorithms, the task may have to be repeated hundreds or thousands of times before learning succeeds. Training a policy onboard a spacecraft is not viable, due to fuel, time, and computer limitations. An alternative is to train the policy in a simulated environment and transfer the trained policy to a physical robot. If the simulated model is sufficiently accurate and the task is not highly dynamic, policies trained in simulation may be directly transferrable to a real robot [31]. For example, Tai et al. [73] trained a room-navigating robot in simulation and deployed it to reality with success. While good results were obtained, this technique is unlikely to generalize well to more difficult or dynamic tasks due to the effect known as the simulation-to-reality gap [31, 46, 49]. This effect states that since the simulator within which the policy is trained cannot perfectly model the dynamics of the real world, such a policy will fail to perform well in a real-world environment due to overfitting of the simulated dynamics. Efforts to get around this problem often take advantage of domain randomization [53], i.e., randomizing the environmental parameters for each simulation to force the policy to become robust to environmental changes. Domain randomization has been used successfully in a drone racing task [52] and in the manipulation of objects with a robotic hand [53]. However, domain randomization significantly increases the training time since the policy must learn to become adept at all dynamic variations of the task. Other efforts to solve the simulation-to-reality problem involve continuing to train the policy once deployed to experiment. A drifting car [47] policy was partially trained in simulation and then fine-tuning training was performed once experimental data was collected. A quadrotor stabilization task [35] summed the policy output with a conventional PD controller to guide the learning process and help with the simulation-to-reality transfer.

Deep reinforcement learning has also been applied to aerospace applications, though mostly in simulation. A simulated fleet of wildfire surveillance aircraft used
deep reinforcement learning to command the flight-path of the aircraft [37]. Deep reinforcement learning has also been applied to spacecraft map generation while orbiting small bodies [32], spacecraft orbit control in unknown gravitational fields [74], and spacecraft orbital transfers [75]. Others have used reinforcement learning to train a policy that performs guidance and control for pinpoint planetary landing [76, 77]. Neural networks have been trained to approximate offline-generated optimal guidance paths for pinpoint planetary landing, such that the neural network approximates an optimal guidance algorithm that can be executed in real-time [78, 79].

Inspired by deep reinforcement learning’s ability to have behaviour be learned rather than hand-crafted, and motivated by the need for new simulation-to-reality transfer techniques, this paper introduces a novel technique that allows for reinforcement learning’s use on a real spacecraft platform. The proposed technique builds off of the planetary landing work [76, 77], where the neural networks were trained to approximate hand-crafted optimal guidance trajectories and a conventional controller was used to track the approximated trajectory. Here, deep reinforcement learning is used to train a guidance policy whose trajectories are fed to a conventional controller to track. Using reinforcement learning allows for an unbiased guidance policy to be discovered by the agent instead of being shown many hand-crafted guidance trajectories to mimic. The proposed technique is in contrast to typical reinforcement learning research, where the policy is responsible for learning both the guidance and control logic. By restricting the policy to learn only the guidance portion, we harness the high-level, unbiased, task-solving abilities of reinforcement learning while deferring the control aspect to the well-established control theory community. Control theories have been developed that are able to perform trajectory tracking well under dynamic uncertainty [80–82], and can therefore handle model discrepancies between simulation and reality. Harris et al. [59] recently presented a strategy that uses reinforcement learning to switch between a set of available controllers depending on the system state. The authors [59] suggest that reinforcement learning should not be tasked with learning guidance and control since control theory already has great success. By combining deep reinforcement learning for guidance with conventional control theory, the policy is prevented from learning a controller that overfits the
error-prone simulated dynamics. We call our deep reinforcement learning guidance strategy *deep guidance*. In light of the above, the novel contributions of this work are:

1. The *deep guidance* technique, that combines deep reinforcement learning as guidance with a conventional controller.

2. Experimental demonstrations showing this deep guidance strategy can be trained in simulation and deployed to reality without any fine-tuning.

3. The first, to the best of the authors’ knowledge, experimental demonstration of artificial intelligence commanding the motion of a spacecraft platform.

It should be noted that although the authors were motivated by difficult guidance tasks, such as manipulator-based spacecraft capture, in this paper a proof-of-concept task—a simple spacecraft pose tracking and docking scenario—is considered. Demonstrating the deep guidance technique on a simple task will highlight its potential for use on more difficult tasks.

This paper is organized as follows: Sec. 2.3 presents background on deep reinforcement learning and the specific learning algorithm used in this paper, Sec. 2.4 presents the novel guidance concept developed by the authors, Sec. 2.5 describes the pose tracking scenario considered, Sec. 2.6 presents numerical simulations demonstrating the effectiveness of the technique, Sec. 2.7 presents experimental results, and Sec. 2.8 concludes this paper.

### 2.3 Markov Decision Processes and Deep Reinforcement Learning

A Markov Decision Process (MDP) describes a discrete-time sequence of events to which reinforcement learning can be applied. For a state $x_n \in \mathcal{X}$ at time step $n$ upon which a chosen action $a_n \in \mathcal{A}$ is applied, a new state $x_{n+1}$ is returned according to a state transition function $P_a(x_n, x_{n+1})$, which may be stochastic, along with a corresponding scalar reward $r_n(x_n, a_n)$. The MDP must satisfy the Markov property, which requires the process be memoryless. In other words, the state $x$ must contain all information required to make an optimal decision. In many cases, the true state is
unknown and must be inferred from an observation of the state, \( o \in \mathcal{O} \), in which case the process is named a Partially Observable Markov Decision Process (POMDP). In the context of robotics, the state of the dynamic system is \( x \), the equations of motion represent the state transition function (which are deterministic), the chosen action is the control effort executed on the dynamics, and the reward is a designer-chosen function that corresponds to task completion and drives the learned behaviour. If camera images are used as inputs, they would represent an observation of the state from which the true state must be inferred. In this work, however, the system state is assumed to be completely known or measured.

Reinforcement learning acts on a MDP to learn a policy \( \pi \) that maps states \( x \) to actions \( a \) in order to maximize the rewards accumulated over time. A table may be used to represent the policy if the state and action spaces are small, but function approximators must be used for large or continuous spaces. Reinforcement learning is said to be “deep” when a deep neural network is used to represent the policy. In this case, the policy is written as \( \pi_\theta \) where \( \theta \) represents the deep neural network’s trainable parameters. The parameters are initialized randomly, yielding random initial behaviour from the policy. Simulations (called episodes in reinforcement learning) are run where the policy explores the environment and collects time steps of data \((x_n, a_n, r_n, x_{n+1})\) and stores them in a replay buffer. In parallel to the data collection, a learning algorithm is used to analyze the collected data to improve the policy \( \pi_\theta \) by updating \( \theta \) in the direction that increases the expected rewards. In contrast to optimal control, reinforcement learning only has access to the dynamics through sampling. The pursuit of rewards by the learning algorithm allows for complex behaviours to emerge according to the reward function; the designer does not need to know how to solve the problem, only when it is solved.

This work uses the Distributed Distributional Deep Deterministic Policy Gradient (D4PG) learning algorithm because it allows for continuous states and actions, yields deterministic behaviour, can be trained across many CPUs, and has excellent performance [83]. The D4PG algorithm has an actor-critic structure, meaning that it learns a policy (the actor) and a value approximation (the critic). Both the actor and critic are approximated by separate neural networks as shown in Fig. 2.1. The
actor predicts the best action to take based on the state, through

$$\mathbf{a}_n = \pi_{\theta}(\mathbf{x}_n)$$  \hspace{0.5cm} (2.1)

while the critic neural network predicts the amount of discounted rewards yet to be received during the remainder of the episode, \(Z_{\phi}(\mathbf{x}, \mathbf{a})\), with trainable parameters \(\phi\). Instead of predicting a scalar discounted reward, it predicts a probability distribution of the expected remaining discounted rewards [84]. The “Distributed” term in the algorithm name implies the algorithm can be distributed across a number of CPUs, while the “Distributional” term indicates that a value distribution prediction is learned rather than a scalar value prediction. Taking the expectation of the critic probability distribution yields the average predicted rewards for the current version of the policy, \(J(\theta)\)

$$J(\theta) = \mathbb{E} \{ Z_{\phi}(\mathbf{x}, \pi_{\theta}(\mathbf{x})) \}$$  \hspace{0.5cm} (2.2)

where \(\mathbb{E}\) denotes the expectation and Eq. (2.1) has been substituted for the action. The learning algorithm seeks to maximize \(J(\theta)\), as training progresses, through adjusting the policy parameters \(\theta\). Since neural networks are differentiable (i.e., the derivative of the neural network’s output can be analytically evaluated with respect to its parameters), the gradient of Eq. (2.2) can be taken with respect to \(\theta\)

$$\nabla_{\theta} J(\theta) = \mathbb{E} \left[ \nabla_{\theta} \pi_{\theta}(\mathbf{x}) \mathbb{E} \left[ \nabla_{\mathbf{a}} Z_{\phi}(\mathbf{x}, \mathbf{a}) |_{\mathbf{a}=\pi_{\theta}(\mathbf{x})} \right] \right]$$  \hspace{0.5cm} (2.3)

which can be numerically evaluated using batches of data drawn from the replay
buffer. The policy network parameters are then updated in the direction of that gradient, through
\[ \theta \leftarrow \theta + \nabla_{\theta} J(\theta) \alpha \] (2.4)
with learning rate \( \alpha \). In order for Eq. (2.3) to accurately calculate policy gradients in the direction of increased rewards, an accurate critic \( Z_\phi(x, a) \) is necessary.

The critic is responsible for predicting the amount of discounted rewards yet to be received during the remainder of the episode, for a given state and action. Since the replay buffer contains state, action, and reward data, supervised learning can be used to minimize the cross-entropy loss between the predicted reward distribution and the true reward given by
\[ L(\phi) = \mathbb{E}\{-Y \log(Z_\phi(x, a))\} \] (2.5)
where \( Y \) are the target reward distribution values the critic should predict, calculated through
\[ Y_n = \sum_{i=0}^{N-1} \gamma^i r_{n+i} + \gamma^N Z_\phi'(x_{n+N}, \pi'_\theta(x_{n+N})) \] (2.6)
where \( \gamma \) is the discount factor that weights future rewards lower than current rewards. To calculate the target values at a given time step \( n \), the rewards received at that time step plus the following \( N - 1 \) time steps (discounted by \( \gamma \)), plus the critic’s own prediction for the remaining rewards calculated at time step \( n + N \) are added together. Using \( N \)-steps of data in Eq. (2.6) has been shown to improve learning [26]. The policy and critic parameters \( \theta' \) and \( \phi' \) in Eq. (2.6) are exponentially-smoothed versions of the true parameters, calculated through
\[ \theta' = (1 - \epsilon) \theta' + \epsilon \theta \] (2.7)
\[ \phi' = (1 - \epsilon) \phi' + \epsilon \phi \] (2.8)
with \( \epsilon \ll 1 \). At each learning iteration, the critic loss in Eq. (2.5) is minimized by adjusting \( \phi \) with learning rate \( \beta \). The term “iteration” refers to the number of times the loss function in Eq. (2.5) is minimized. Minimizing the critic’s loss function improves the critic’s prediction for the expected reward distribution, which improves the policy update in Eq. (2.4).
Data is collected from $K$ parallel workers that run episodes (each individual simulation is called an episode) using the most up-to-date version of the policy. To force exploration of the action space, noise is applied to the chosen action according to

$$a_n = \pi_\theta(x_n) + \mathcal{N}(0, \sigma^2)$$

(2.9)

where $\mathcal{N}(0, \sigma^2)$ is the normal distribution with zero mean and $\sigma$ standard deviation. Once training is complete and the policy is being used, or if the policy’s performance is being evaluated during training, no exploration noise is added.

A summary of the D4PG algorithm is presented in Algorithm 1.

### 2.4 Deep Guidance

Using deep reinforcement learning for spacecraft guidance may allow for difficult tasks to be accomplished through learning an appropriate behaviour rather than hand-crafting such a behaviour. In order for the reinforcement learning algorithm presented in Sec. 2.3 to be trained in simulation and deployed to reality without any fine-tuning on the spacecraft, it is proposed that the learning algorithm cannot be responsible for the entire guidance, navigation, and control stack (GNC). This is to prevent the policy from overfitting the simulated dynamics and being unable to handle the transition to a dynamically-uncertain real world, i.e., the simulation-to-reality problem [31]. For this reason, the authors present a system, called *deep guidance*, which uses deep reinforcement learning as a guidance system along with a conventional controller. Conventional controllers are able to handle dynamic uncertainties and modelling errors that typically plague reinforcement learning policies that attempt to learn the entire GNC routine. It is assumed that perfect navigation is available. A block-scheme diagram of the proposed system is shown in Fig. 2.2. The learned *Deep Guidance* block has the current state $x_n$ as its input and the desired (denoted with the superscript $d$) velocity $\dot{x}^d_n$ as its output. The desired velocity is fed to a conventional controller, that also receives the current state $x_n$, and calculates a control effort $u_n$. The control effort is executed on the dynamics that generate a scalar reward $r_n$ and the next state $x_{n+1}$. 
Learner

Initialize policy network weights $\theta$ and value network weights $\phi$ randomly
Initialize smoothed policy and target value network weights $\theta' = \theta$ and $\phi' = \phi$
Launch $K$ actors and copy policy weights $\theta$ to each actor
repeat
  Sample a batch of $M$ data points from the replay buffer
  Compute the target value distribution used to train the value network
  $Y_n = \sum_{i=0}^{N-1} \gamma^i r_{n+i} + \gamma^N Z_{\phi'}(x_{n+N}, \pi_{\theta'}(x_{n+N}))$
  Update value network weights $\phi$ by minimizing the loss function
  $L(\phi) = \mathbb{E}\{-Y \log(Z(\phi(x, a)))\}$ using learning rate $\beta$
  Compute policy gradients using
  $\nabla_{\theta} J(\theta) = \mathbb{E}\{\nabla_{\theta} \pi_{\theta}(x) \mathbb{E}\{\nabla_a Z_{\phi}(x, a)\}|a=\pi_{\theta}(x)\}$ and update policy
  weights via $\theta = \theta + \nabla_{\theta} J(\theta) \alpha$
  Update the smoothed network weights slowly in the direction of the main
  policy and value network weights $\theta' = (1 - \epsilon)\theta' + \epsilon \theta$ and $\phi' = (1 - \epsilon)\phi' + \epsilon \phi$ for $\epsilon \ll 1$
until acceptable performance

Actor

repeat
  From the given state, use the policy to obtain an action and add
  exploration noise $a_n = \pi_{\theta}(x_n) + \mathcal{N}(0, \sigma^2)$
  Step environment forward one time step using action $a_n$
  Record $(x_n, a_n, r_n, x_{n+1})$ and store in the replay buffer $R$
  At the end of each episode, obtain the most up-to-date version of the
  policy $\pi_{\theta}$ from the Learner and reset the environment.
until acceptable performance

Algorithm 1: D4PG [83]

---

**Figure 2.2:** Proposed deep guidance strategy.
During training of the deep guidance model, an ideal controller is assumed, as shown in Fig. 2.3. This ensures the guidance model does not overfit to any specific controller, thereby making this approach controller-independent. Since the ideal controller perfectly commands the dynamics to move at the desired velocity $\dot{x}_d^n$, the ideal controller and the dynamics model may be combined into a single kinematics model.

Once trained, any controller may be combined with the deep guidance system for use on a real robot. Since the deep guidance system only experiences an ideal controller during training, it is possible that the guidance model will not experience any non-ideal states that may be encountered when using a real controller, which may harm performance. For this reason, Gaussian noise may be applied to the output of the kinematics to force the system into undesirable states during training that may be encountered by a non-ideal controller.

The proposed system is trained and tested on a spacecraft pose tracking and docking task detailed in the following section.

### 2.5 Problem Statement

This section presents the simulated spacecraft pose tracking and docking environment the deep guidance system will be trained within. Though the deep guidance technique may allow for more complex behaviours to be learned, a simple task considered here is presented as a proof-of-concept. Similar to other spacecraft researchers with experimental validation [62,85–87], a double-integrator planar dynamics model is used such that the available experimental facility can validate the simulated results. Including a full orbital model was deemed to be outside the scope of this paper, since the goal of this work is to demonstrate the simulation-to-reality ability of the proposed deep guidance approach. Furthermore, a double-integrator model is representative of
proximity operations in orbit over small distances and timescales [88].

A chaser spacecraft exists in a planar laboratory environment and is tasked with approaching and docking with a target spacecraft, as shown in Fig. 2.4. The chaser and target spacecraft start at rest. The chaser spacecraft is given some time to maneuver to the hold point in front of the target. Then, the chaser spacecraft is tasked with approaching the target such that the two may dock. Docking itself is not considered in this paper; only moving to the docking point is.

2.5.1 Kinematics and Dynamics Models

During training, a kinematics model is used which approximates a dynamics model and an ideal controller, as shown in Fig. 2.3. The deep guidance policy accepts the chaser state error $e_n$ and outputs the commanded action, which in this case is the desired velocity, $\dot{x}_n^d$.

$$\mathbf{x}_n = \begin{bmatrix} x & y & \psi \end{bmatrix}^T$$

$$e_n = \mathbf{x}_d - \mathbf{x}_n$$

$$\dot{x}_n^d = \pi_\theta(e_n)$$

where $\mathbf{x}_d$ is the desired state and $\mathbf{x}_n$ is the state of the system, where $x$ and $y$ represent the $X$ and $Y$ location of the chaser, respectively, and $\psi$ represents the orientation of the chaser. With the ideal controller assumption invoked, the desired velocity, $\dot{x}_n^d$, is

Figure 2.4: Spacecraft pose tracking and docking task.
numerically integrated using the Scipy [89] Adams/Backward differentiation formula
methods in Python to obtain $x_{n+1}$.

To evaluate the learning performance, the trained policy is periodically evaluated
on an environment with full dynamics and a controller, as shown in Fig. 2.2. In
other words, it is “deployed” to another simulation for evaluation in much the same
way that it will be deployed to an experiment in Sec. 2.7. The deep guidance policy
outputs the desired velocity as in Eq. (2.12) that is fed to a simple proportional
velocity-controller of the form

$$u_n = K_p (\dot{x}_n^d - \dot{x}_n)$$

where $K_p = \text{diag}(2, 2, 0.1)$ and $u_n$ is the control effort. The $K_p$ values were chosen
by trial-and-error until satisfactory performance was achieved.

A double-integrator dynamics model is used to simulate the motion of the chaser

$$\ddot{x} = \frac{F_x}{m}$$

$$\ddot{y} = \frac{F_y}{m}$$

$$\dot{\psi} = \frac{\tau}{I}$$

where $F_x$ and $F_y$ are the forces applied in the $X$ and $Y$ directions, respectively, $\tau$ is
the torque applied about the $Z$ axis, $m$ is the chaser spacecraft mass, $I$ is its moment
of inertia, $\ddot{x}$ is the acceleration in $X$, $\ddot{y}$ is its acceleration in $Y$, and $\dot{\psi}$ is the angular
acceleration about $Z$. The accelerations are numerically integrated twice to obtain
the position and orientation at the following timestep.

The following subsection discusses the reward function used to incentivize the
desired behaviour.

2.5.2 Reward Function

To calculate the reward given to the agent at each timestep, a reward field, $f$, is
generated according to

$$f(x_n) = -|e_n|$$
with $e_n$ obtained from Eq. (2.11). The reward field is zero at the desired state and becomes negative linearly as the chaser moves away from the desired state.

The reward given to the agent, $r_n(x_n, a_n)$, depends on the action taken in a given state. Therefore, the difference in the reward field between the current and previous timestep is used to calculate the reward given to the agent

$$r_n = \| K(f(x_n) - f(x_{n-1})) \|$$

(2.18)

the states are weighted with $K = \text{diag}(0.5, 0.5, 0.1)$ to ensure the rotational component does not dominate the reward field. By rewarding the change in reward field, a positive reward will be given to the agent if it chooses an action that moves the chaser closer to the desired state and a negative reward otherwise. Two additional penalties are included to encourage the desired behaviour: a velocity-error penalty and a collision penalty. To avoid chaser oscillations, velocity errors are penalized near the desired state. To discourage collisions, the reward is reduced by $r_{\text{collide}} = 15$ when the chaser collides with the target (chosen by trial-and-error until collisions were successfully deterred).

$$r_n = \begin{cases} 
\| K(f(x_n) - f(x_{n-1})) \| - c_1 \frac{\| \bar{x}_n - v_{\text{ref}} \|}{\| e_n \| + \eta} - r_{\text{collide}} & \text{for } \| x_t - x_n \| \leq 0.3 \\
\| K(f(x_n) - f(x_{n-1})) \| - c_1 \frac{\| \bar{x}_n - v_{\text{ref}} \|}{\| e_n \| + \eta} & \text{otherwise}
\end{cases}$$

(2.19)

Here, $v_{\text{ref}}$ is the velocity of the hold/docking point, $\eta = 0.01$ is a small constant to avoid division by zero, $c_1 = 0.5$ is used to weight the velocity penalty such that it does not dominate the reward function, and $x_t$ is the target position. All constants were hand-selected on the ground of: 1) yielding rewards/penalties proportional to the significance of the event; and 2) weighting the different reward/penalty terms such that one does not dominate the reward function (causing the learning algorithm to pay particular attention to that term).

The learning algorithm implementation details are presented in the following subsection.
2.5.3 Learning Algorithm Details

The policy and the value neural networks have 400 neurons in their first hidden layer and 300 neurons in their second layer—Fig. 2.1 is not to scale. The neural network size was replicated from the original DDPG paper [90]. The number and size of layers may have to be adjusted according to the complexity of the task, though that was not the case with this work. In the value network, the action is fed directly into the second layer of the network, as this was empirically shown to be beneficial [83,90]. The value network therefore has 138,951 trainable weights \( \phi \) and the policy network has 123,603 trainable weights \( \theta \). Each neuron in both hidden layers uses a rectified linear unit as its nonlinear activation function, shown below

\[
g(y) = \begin{cases} 
0 & \text{for } y < 0 \\
y & \text{for } y \geq 0 
\end{cases} \tag{2.20}
\]

In the output layer of the policy network, a \( g(y) = \tanh(y) \) nonlinear activation function is used to force the commanded velocity to be bounded. The output layer of the value neural network uses the softmax function, shown below, to force the output value distribution to indeed be a valid probability distribution

\[
g(y_i) = \frac{e^{y_i}}{\sum_{k=1}^{K} e^{y_k}} \quad \forall i = 1, \ldots, B \tag{2.21}
\]

for each element \( y_i \) and for \( B \) bins in the value distribution. Drawing from the original value distribution paper [84], \( B = 51 \) bins are used in this work. The value distribution bins are evenly spaced on the empirically-determined interval \([-1000, 100]\), as this is the range of accumulated rewards encountered during this pose tracking and docking task.

The policy and value networks are trained using the Adam stochastic optimization routine [91] with a learning rate of \( \alpha = \beta = 0.0001 \). The replay buffer \( R \) can contain \( 10^6 \) transition data points and a mini batch size \( M = 256 \) is used. The smoothed network parameters are updated on each training iteration with \( \epsilon = 0.001 \). The noise standard deviation applied to the action to force exploration during training is
\[ \sigma = \frac{1}{3} \left[ \max(a) - \min(a) \right] (0.9999)^E, \]

where \( E \) is the episode number (one simulated attempt at solving the task is called one \textit{episode}; the episode number increases as training progresses); having a standard deviation of one third the action range empirically leads to good exploration of the action space. The noise standard deviation is decayed exponentially as more episodes are performed to narrow the action search area, at a rate that halves roughly every 7,000 episodes. The decay rate was chosen after determining roughly how many episodes are needed for learning to succeed. Ten actors are used, \( K = 10 \), to maximize CPU usage during training, such that simulated data using the most up-to-date version of the policy is being collected by ten actors simultaneously. A discount factor of \( \gamma = 0.99 \) was used along with an N-step return length of \( N = 1 \). The Tensorflow [92] machine learning framework was used to generate and train the neural networks.

Every five training episodes, the current policy is “deployed” and run in a full dynamics environment with the proportional velocity-controller in Eq. (2.13) to evaluate its performance, as shown in Fig. 2.2. During deployment, \( \sigma = 0 \) in Eq. (2.9) such that no exploration noise is applied to the deep guidance velocity commands.

### 2.6 Simulation Results

To test the deep guidance approach, three variations of the spacecraft pose tracking and docking task are studied. The first uses a stationary target, the second uses a rotating target, and the third uses a rotating target with a stationary obstacle that must be avoided. The first scenario is run both with constant initial conditions and with randomized initial conditions, while the second and third scenarios use randomized initial conditions. The 30 cm cube spacecraft platform has a uniform simulated mass of 10 kg.

#### 2.6.1 Docking with a Stationary Target

The chaser and target nominal initial conditions are \([3 \text{ m}, 1 \text{ m}, 0 \text{ rad}]\) and \([1.85 \text{ m}, 0.6 \text{ m}, \frac{\pi}{2} \text{ rad}]\) for the chaser and target, respectively. The hold point is 1.0 m offset from the front-face of the target and the docking point is 0.5 m offset. The
simulation is run for 90 seconds with a 0.2 second timestep. For the first 45 seconds, the desired state is the hold point and afterwards it is the docking point. The commanded velocity bounds are $\pm 0.05 \text{ m/s}$ and $\pm \frac{\pi}{18} \text{ rad/s}$.

Results of the first spacecraft pose tracking and docking task are shown in Fig. 2.5, and are the result of 47 hours of training on an Intel® i7-8700K CPU. The learning curve, shown in Fig. 2.5a, plots the total rewards received on each episode which increases, as expected, during training. The deep guidance strategy successfully learned the desired behaviour after roughly 11,000 episodes. The loss function, calculated using Eq. (2.5) and shown in Fig. 2.5b, decreases as anticipated, indicating that the value network output distribution is approaching the target values calculated using Eq. (2.6), on average. Sample trajectories during training are shown in Fig. 2.6. The gray object represents the initial pose of the chaser, the dashed line represents its trajectory, and the solid black object represents its final pose. The pose tracking and docking task was successfully learned.

Next, the chaser and target initial conditions were randomized at the beginning of each episode to force the deep guidance system to generalize across a range of initial states and not simply master a single trajectory. The initial states were randomized around the nominal ones according to a normal distribution with a standard deviation
Figure 2.6: Visualization of chaser trajectories at various episodes during the training process.
of 0.3 m for position and $\frac{\pi}{2}$ rad for attitude. Figure 2.7 shows the learning curve and associated loss function during training. The learning curve, shown in Fig. 2.7a, shows that the agent successfully learned a more general guidance strategy in the presence of randomized initial conditions. The learning curve appears noisier than the learning curve shown in Fig. 2.5a because each episode may have slightly more or less rewards available depending on the initial conditions. Sample trajectories once training was complete are shown in Fig. 2.8.

2.6.2 Docking with a Spinning Target

This subsection presents the second scenario that the deep guidance system was trained on. That is, a spacecraft pose tracking and docking task in the presence of a spinning target. All learning parameters are identical to those presented in Sec. 2.5.3 demonstrating the generality of the proposed deep guidance approach. The target spacecraft is given a constant counter-clockwise angular velocity of $\omega = \frac{\pi}{45}$ rad/s. The velocity bounds on the chaser are increased to $\pm 0.1$ m/s. The simulation is run for 180 seconds with a 0.2 second timestep. For the first 90 seconds, the chaser is incentivized to track the moving hold position, and afterwards it is rewarded for tracking the moving docking point. The initial conditions have a mean of [3 m, 1 m, 0 rad] and
Figure 2.8: Examples of learned chaser trajectories with randomized initial conditions.
[1.85 m, 1.2 m, 0 rad] for the chaser and target, respectively, and a standard deviation of 0.3 m for position and $\pi/2$ rad for attitude. Since the target is rotating, the hold and docking points are inertially moving with time. The learning curve in Fig. 2.9a shows that a deep guidance policy was successfully learned. Sample trajectories, shown in Fig. 2.10, show example motion of the chaser. The target performs two complete rotations during the episode so its initial and final orientation are as shown.

2.6.3 Docking While Avoiding an Obstacle

This section presents a numerical simulation that is identical to Sec. 2.6.2 except for the addition of a stationary obstacle that must be avoided. The input to the policy is modified to include the distance from the chaser to the obstacle, as follows:

$$o_n = \left[ e_n^T, \mathbf{x}_o^T - \mathbf{x}_n^T \right]^T$$

(2.22)

where $\mathbf{x}_o$ is the position of the obstacle and $\mathbf{x}_n$ is the chaser position from Eq. (2.10). For this scenario, the deep guidance velocity is calculated through

$$\mathbf{x}^d_n = \pi_\theta(o_n)$$

(2.23)
Figure 2.10: Examples of learned chaser trajectories with a rotating target.
The $r_{\text{collide}}$ penalty is also applied when the chaser collides with the obstacle. Collision occurs when the centre of mass of the chaser is within 0.3 m of the obstacle. The obstacle’s position is $[1.2, 1.2]$ m.

The learning curve in Fig. 2.11a shows that a deep guidance policy was successfully learned. Sample trajectories, shown in Fig. 2.12, show the motion of the chaser. A policy that causes the chaser to track the target while avoiding collision with the obstacle was successfully learned.

The deep guidance system presented in this paper was successfully trained on simulated spacecraft pose tracking and docking tasks. It allows the designer to easily specify a reward function to convey the desired behaviour to the agent instead of hand-crafting a guidance trajectory. Although the guidance trajectories learned in this paper by the deep guidance system would be trivial to hand-craft, the purpose of this paper is to introduce the deep guidance technique. It is expected that the deep guidance technique will unlock the ability for more difficult learned guidance strategies in future work. All code used has been open-sourced and can be accessed at www.github.com/Kirkados/JSR2020_D4PG.

The trained guidance policies are tested in experiment in the following section.
Figure 2.12: Examples of learned chaser trajectories while avoiding an obstacle.
2.7 Experimental Validation

To validate the numerical simulations, and to test if the proposed deep guidance strategy can overcome the simulation-to-reality gap, experiments are performed in a laboratory environment at Carleton University. A planar gravity-offset testbed is used, where two spacecraft platforms are positioned on a flat granite surface. Air bearings are used to provide a near friction-free planar environment. The experimental facility is discussed, followed by the experimental setup and results.

2.7.1 Experiment Facility

Experiments were conducted at Carleton University’s Spacecraft Robotics and Control Laboratory, using the Spacecraft Proximity Operations Testbed (SPOT). Specifically, SPOT consists of two air-bearing spacecraft platforms operating in close proximity on a 2.4 m × 3.7 m granite surface. The use of air bearings on the platforms reduces the friction to a negligible level. Due to surface slope angles of 0.0026° and 0.0031° along both directions, residual gravitational accelerations of 0.439 and 0.525 mm/s² perturb the dynamics of the floating platforms along the X and Y directions, respectively. Both platforms have dimensions of 0.3 × 0.3 × 0.3 m and are actuated by expelling compressed air at 550 kPa (80 psi) through eight miniature air nozzles distributed around each platform, thereby providing full planar control authority. Each thruster generates approximately 0.25 N of thrust and is controlled at a frequency of 500 Hz by a pulse-width modulation scheme using solenoid valves. Pressurized air for the thrusters and the air bearing flotation system is stored onboard in a single air cylinder at 31 MPa (4,500 psi). The structure consists of an aluminum frame with four corner rods on which three modular decks are stacked. To protect the internal components, the structure is covered with semi-transparent acrylic panels. Figure 2.13a shows the SPOT laboratory facility and Fig. 2.13b shows two SPOT platforms in a proximity operations configuration.

The motion of both platforms is measured in real-time through four active LEDs on each platform which are tracked by an 10-camera PhaseSpace© motion capture system. This provides sub-millimetre accurate ground-truth position and attitude data. All motion capture cameras are connected to a PhaseSpace© server, which is
Figure 2.13: The Spacecraft Proximity Operations Testbed (SPOT).
connected to a ground station computer. The ground station computer wirelessly communicates ground truth navigation information to the platforms’ onboard computers, which consist of Raspberry Pi-3s running the Raspbian Linux operating system. Based on the position and attitude data the platforms wirelessly receive, they can perform feedback control, by calculating the required thrust to maneuver autonomously and actuating the appropriate solenoid valves to realize this motion. The ground station computer also receives real-time telemetry data (i.e., any signals of interest, as specified by the user) from all on-board computers, for post-experiment analysis purposes.

A MATLAB/Simulink® numerical simulator that recreates the dynamics and emulates the different on-board sensors and actuators is first used to design and test the upcoming experiment. Once the performance in simulations is satisfactory, the control software is converted into C/C++ using Embedded Coder®, compiled, and then executed on the platforms’ Raspberry Pi-3 computers.

An Nvidia Jetson TX2 Module is used to run the trained deep guidance policy neural network in real-time. It repeatedly executes Eq. (2.12), returning a guidance velocity signal to the Raspberry Pi-3 that executes a control law to track the commanded velocity.

### 2.7.2 Setup

The simulations presented in Sec. 2.6 were chosen such that they are replicable experimentally. In other words, all three pose tracking and docking tasks with randomized initial conditions are attempted in experiment. The final parameters, $\theta$, of the trained deep guidance policies are exported for use on the chaser SPOT platform. The value network is only used during training and is not exported along with the policy network. Initial conditions are similar to those used previously in simulation.

The platforms remain in contact with the table until a strong lock has been acquired on the LEDs by the motion capture system. Following this, the platforms begin to float and maneuver to the desired initial conditions. Then, the target remains stationary or begins rotating while the chaser platform uses the deep guidance policy trained in simulation to guide itself towards the hold point and finally the
docking point on the target.

It should be noted that a significant number of discrepancies exist between the simulated environment the policy was trained within and the experimental facility. The simulated environment did not account for the discrete thrusters and their limitations, the control thrust allocation strategy, signal noise, system delays, friction, air resistance, centre of mass offsets, thruster plume interaction, and table slope. In addition, the spacecraft mass used to evaluate the training was 10 kg whereas the experimental spacecraft platforms have a mass of 16.9 kg. Significant discrepancies exist between the simulated training environment and the experimental environment, which is therefore an excellent test for the simulation-to-reality capabilities of the proposed deep guidance technique. Due to facility size limitations, the hold point was reduced from 1.0 m to 0.9 m offset from the front-face of the target.

2.7.3 Results

A trajectory of the experiment with a stationary target is shown in Fig. 2.14a. It shows the the deep guidance successfully outputs velocity commands that brings the chaser to the hold point and then additional velocity commands to move towards the target docking port. A trajectory of the experiment with a rotating target is shown in Fig. 2.14b, and a trajectory of the experiment with an obstacle and a rotating target is shown in Fig. 2.14c. The learned deep guidance technique successfully commands an appropriate velocity signal for the chaser to complete the tasks.

The deep guidance technique was successfully trained exclusively in simulation and deployed to an experimental facility and achieved similar performance to that during training. Combining the neural-network guidance with conventional control allowed the trained system to handle unmodeled effects present in the experiment. The proposed technique successfully demonstrates the deep guidance technique as a viable solution to the simulation-to-reality problem present in deep reinforcement learning. A video of the simulated and experimental results can be found online at https://youtu.be/n7K6aC5v0aY.
Figure 2.14: Experimental trajectories of deep guidance in Carleton’s Spacecraft Proximity Operations Testbed.
2.8 Conclusion

This paper introduced deep reinforcement learning to the guidance problem for spacecraft robotics. Through training a guidance policy to accomplish a goal, complex behaviours can be learned rather than hand-crafted. The simulation-to-reality gap dictates that policies trained in simulation often do not transfer well to reality. To avoid this, and to avoid additional training once deployed to a physical robot, the authors restrict the policy to output a guidance signal, which a conventional controller is tasked with tracking. Conventional control can handle the modelling errors that typically plague reinforcement learning. This paper tests this learned guidance technique, which the authors call deep guidance, on a simple problem. That is, spacecraft pose tracking and docking. Numerical simulations show that proximity operation tasks can be successfully learned using the deep guidance technique. The trained policies are then deployed to three experiments, with comparable results to those in simulation even though the simulated environment did not model all effects present in the experimental facility. Future work will further explore the generality of the technique and its use on more difficult problems.

Acknowledgments

This research was financially supported in part by the Natural Sciences and Engineering Research Council of Canada under the Postgraduate Scholarship-Doctoral PGSD3-503919-2017 award.

Continuity to Next Chapter

This Chapter used deep reinforcement learning to issue velocity guidance signals. The following Chapter investigates whether acceleration signals are better than velocity signals. In addition, it applies the technique to a different experimental scenario (a quadrotor facility at École Nationale de l’Aviation Civile) to probe the suitability of the technique to other domains.
Chapter 3

Acceleration-based Quadrotor Guidance Under Time Delays Using Deep Reinforcement Learning

This chapter has been peer-reviewed and was presented in the AIAA Guidance, Navigation, and Control Conference. It should be cited as:


The paper is included in this thesis with minor formatting changes and variable renaming (for consistency between chapters). The DRL theory section was removed to avoid repetition, with the reader being referred to Sec. 2.3. This paper was co-authored by the thesis author, Kirk Hovell, his thesis supervisor, Prof. Steve Ulrich, and a colleague, Prof. Murat Bronz. Mr. Hovell set-up, developed, wrote, and evaluated the relevant code; obtained and analyzed simulated results; prepared software for experimental demonstrations; analyzed experimental results; and wrote the manuscript. Theory development and manuscript editing were conducted jointly. Prof. Bronz was responsible for obtaining experimental results.

Paper Context

This second paper builds off the deep reinforcement learning-based guidance technique developed in Chap. 2. It compares the velocity-based approach, developed in the previous chapter, to an acceleration-based guidance approach. In addition, the guidance technique is applied to a quadrotor proximity operations task to explore the feasibility of the deep reinforcement learning-based technique to different domains.
3.1 Abstract

This paper investigates the use of deep reinforcement learning to act as closed-loop guidance for quadrotors and the ability for such a system to be trained entirely in simulation before being transferred for use on a real quadrotor. It improves upon the previous work where velocity-based deep reinforcement learning was used to guide the motion of spacecraft. Here, an acceleration-based closed-loop deep reinforcement learning guidance system is developed and compared to the previous work in Chapter 2. In addition, state augmentation is included due to the presence of dynamic delays. Simulated results show acceleration-based deep reinforcement learning closed-loop guidance has significant performance benefits compared to velocity-based guidance in the previous work, namely: a simpler reward function, lesser overshoot, and better steady-state error. To evaluate the use of this system on a real quadrotor, the trained system is deployed to the Paparazzi aircraft simulation software, and is implemented on real flight hardware at École Nationale de l’Aviation Civile for an experimental comparison. Experimental results confirm the simulated results—that acceleration-based deep guidance outperforms velocity-based deep guidance and should therefore be used in future work.

3.2 Introduction

Quadrotors have become useful in recent decades due to advances in microcontroller and battery technology, with common applications of: research, search and rescue, surveillance, photography, package delivery, and sport racing. Many quadrotor control theories have been presented [93–95]. Some have applied backstepping control [96], sliding mode control [97], model predictive control [98], dynamic inversion techniques [99–102], or used cameras for feedback [103, 104]. A number of guidance approaches have also been proposed to guide the motion of a quadrotor [105,106], including trajectory calculations for quadrotor swarms [107] and dynamically feasible trajectory calculations [108]. The traditional guidance and control techniques presented are hand-crafted to solve a particular problem and their design may require significant engineering effort. As quadrotors are applied to more difficult tasks,
the effort required to design these guidance algorithms and controllers may become infeasible. For example, an autonomous search-and-rescue task where an unknown building is explored and GPS is unavailable is a difficult problem to hand-craft a response to. Motivated by difficult guidance tasks, this paper uses deep reinforcement learning to learn, rather than hand-craft, a guidance strategy for quadrotors.

Deep reinforcement learning consists of an agent that chooses actions to explore an environment. The environment returns rewards to the agent corresponding to the quality of the action taken from a given observation. Through trial and error, the agent attempts to learn a policy that creates an optimal mapping from observations to actions in order to maximize the rewards received. In using this approach, complex behaviour can emerge from a relatively simple reward structure. For example, deep reinforcement learning has learned how to play video games at a superhuman level, using only the game score as the reward signal; the policy input observation is the screen pixels and the action output is the button presses on the controller [26]. Neural networks have become a popular choice for representing the policy, as they have been shown to be universal function approximators [70]. Deep reinforcement learning has seen many successes in recent years, from the successful completion of many Atari 2600 games [26] to mastering the game of Go [72].

Training deep reinforcement learning policies on physical robots is time consuming, expensive, and leads to significant wear-and-tear on the robot because, even with state-of-the-art learning algorithms, the task may have to be attempted hundreds or thousands of times before learning succeeds. Alternatively, training can be completed in simulation and the resulting policy can be transferred to a real robot. However, this approach often encounters problems due to the simulation-to-reality gap—policies trained entirely in simulation tend to overfit the simulated dynamics, which can never perfectly model reality [31, 46, 49]. Algorithms that focus on learning speed are being developed such that learning can be performed entirely on a robot in a short period of time [44, 45]. However, some high-risk robotic domains cannot afford any on-board learning due to the possibility of damaging the robot. For this reason, the development of algorithms that can be trained entirely in simulation and deployed to a robot is an active research area. Domain randomization is one solution to this problem.
where environmental parameters are randomized for each training simulation to force the policy to become robust to a variety of environments. Other efforts combine domain randomization with neural representations of intermediate states [113], while others train in simulation and fine-tune the policy once deployed to experiment [47, 112, 113].

Reinforcement learning has been used in quadrotor applications. In 2005, model-based reinforcement learning was used to generate a dynamics model of a quadrotor that was used to develop an optimal controller [34]. Reinforcement learning has also been used as an inner-loop controller for a simulated quadrotor, where it was shown that reinforcement learning can outperform classical control techniques [36]. Others used quadrotors to carry payloads using reinforcement learning [114]. A quadrotor stabilization task [35] summed the policy output with a conventional PD controller to guide the learning process and help with the simulation-to-reality transfer. A simulated fleet of wildfire surveillance aircraft used deep reinforcement learning to command the flight-path of the aircraft [37]. Domain randomization has enabled real quadrotor flight after being trained entirely in simulation [51], and domain randomization has since also been used for drone racing [52].

In terms of quadrotor guidance, reinforcement learning was used to guide a quadrotor in a grid-world [115] and, along with model predictive control, it was used to guide a quadrotor through an indoor maze [116]. The approach uses a fixed grid of nodes, each with an associated cost. When each node is visited, the cost is updated. Instead of generating a grid that uses reinforcement learning to evaluate the desirability of each location, this paper, in contrast, uses deep reinforcement learning as closed-loop guidance to generate desired acceleration signals in real-time that are fed to a controller. Restricting reinforcement learning to handle guidance-only was suggested by Harris et al. [59]. They argued that reinforcement learning should not be tasked with learning guidance and control since control theory already has great success. Hovell and Ulrich’s previous work used deep reinforcement learning for closed-loop guidance and used a conventional controller to track the guidance signal [1]. The closed-loop
guidance was trained entirely in simulation and transferred to a real spacecraft platform experiment. The controller was able to handle discrepancies between the simulated training environment and the experimental facility. The technique was named deep guidance, and was shown to be a possible bridge from simulation to reality for reinforcement learning.

This paper continues the study of the deep guidance technique [1]. It improves the technique, by proposing a novel implementation where acceleration guidance signals are used, and compares it to the previous implementation where velocity guidance signals were used. In addition, this paper applies the deep guidance technique to a new quadrotor domain, where nonlinear dynamics are present and system delays must be accounted for. In light of the above, the novel contributions of this work are:

1. The application of deep reinforcement learning to real quadrotors using a modified closed-loop deep guidance technique.

2. Improving the deep guidance technique itself by introducing closed-loop acceleration signals and comparing it to previous work.

This paper is organized as follows: Sec. 3.3 presents background on deep reinforcement learning and the specific learning algorithm used in this paper, Sec. 3.4 describes the quadrotor scenario considered and how the improved deep guidance technique is applied to it, Sec. 3.5 presents numerical simulations demonstrating the effectiveness of the technique, Sec. 3.6 presents experimental results, and Sec. 3.7 concludes this paper.

3.3 Markov Decision Processes and Deep Reinforcement Learning

To avoid repetition, this section has been removed; please see Sec. 2.3 for a description of deep reinforcement learning and the D4PG algorithm used in this work.

3.4 Problem Statement

This section discusses the quadrotor pose tracking environment within which the DRL-based guidance system (which the authors call deep guidance) will be trained.
Though the deep guidance technique may allow for complex behaviours to be learned, a relatively simple task is attempted here in order to determine whether a velocity- or acceleration-based implementation of the deep guidance technique is most appropriate. Conclusions from this work will be used to inform attempts at solving difficult guidance problems in Chaps. 4 and 5.

The task presented here consists of two quadrotors: a target and a follower. The quadrotors start at some initial conditions, and the follower must move itself to a location three metres offset from the target. The target and follower positions and velocities are

\[
\begin{align*}
x_t^n &= \begin{bmatrix} x_{t_n} & y_{t_n} & \psi_{t_n}^t \end{bmatrix} \\
v_t^n &= \begin{bmatrix} v_{x_{t_n}} & v_{y_{t_n}} & \omega_{t_n}^t \end{bmatrix} \\
x_f^n &= \begin{bmatrix} x_{f_n} & y_{f_n} \end{bmatrix} \\
v_f^n &= \begin{bmatrix} v_{x_{f_n}} & v_{y_{f_n}} \end{bmatrix}
\end{align*}
\]

where \(x_t^n, v_t^n, x_f^n, v_f^n\) are the positions and velocities of the target and follower, respectively, \(x\) and \(y\) represent the positions in Cartesian space, \(\psi\) represents the yaw about the \(z\) axis, \(\omega\) represents the yaw rate, subscript \(n\) refers to the time step number, superscript \(t\) corresponds to the target, superscript \(f\) corresponds to the follower, and \(v\) is the velocity.

### 3.4.1 Deep Guidance

The deep guidance technique, introduced by Hovell and Ulrich [1], allows for reinforcement learning to be used on real robot platforms, despite being trained entirely in simulation, by limiting deep reinforcement learning to only learn the guidance portion of the guidance, navigation, and control process. The learned closed-loop guidance system passes signals, calculated in real-time, to a conventional controller to track regardless of modelling errors, and is presented as a possible solution to the simulation-to-reality problem. Figure 3.1 shows the previous work’s closed-loop deep guidance approach for velocity-based guidance. The observation is \(o_n\), the state is \(x_n\), the desired velocity guidance signal is \(\mathbf{d}_n^{fd}\), the control effort is \(u_n\), and the next
In this work, a different strategy is proposed: instead of the deep guidance block issuing desired velocities, it issues desired accelerations, as shown in Fig. 3.2, where $\ddot{x}^d_n$ is the guided acceleration. This modification to the deep guidance technique is expected to improve performance while simplifying the reward function, since training will now occur within a domain of the same order to the dynamics where the policy will be used once training is complete. To ensure the deep guidance policy does not overfit a particular controller during training, an ideal controller is assumed. This allows for the controller and dynamics blocks to be combined into a single kinematics block, as shown in Fig. 3.3. Any controller may be used along with the deep guidance system to control a real robot, so long as it accurately tracks the desired accelerations. As will be shown in this paper, the controller and dynamics used to evaluate the deep guidance performance in simulation can be significantly different than the controller.
and dynamics used in an experiment. The velocity-based and acceleration-based deep guidance strategies are compared in this paper.

### 3.4.2 Kinematics Model

While the deep guidance policy is being trained, a kinematic model approximates the dynamics model and the ideal controller, as shown in Fig. 3.3. When the velocity-based deep guidance model is used, the policy input observation is:

\[
o_n = \begin{bmatrix} x_n^t & x_n^f \end{bmatrix}^T \tag{3.5}
\]

When the acceleration-based deep guidance model is used, the policy input observation is:

\[
o_n = \begin{bmatrix} x_n^t & x_n^f & v_n^t & v_n^f \end{bmatrix}^T \tag{3.6}
\]

During training, with an ideal controller assumed, the guided velocity signal \( \hat{x}_n^{f,d} \) is directly integrated once to obtain the next state. The acceleration signal \( \hat{\ddot{x}}_n^{f,d} \) is integrated twice to obtain the next state. All integration is performed using the Scipy Adams/Backward differentiation formula methods in Python. When delays are included, as discussed in the following subsection, the guided velocity or acceleration is stored and an action \( D \) time steps old is used instead.

### 3.4.3 Time Delay Consideration

This work includes the possibility for the closed-loop deep guidance model to encounter system delays, which may occur through actuation delays, signal delays, or measurement delays. When the action taken is not immediately realized on the quadrotor, the input observation to the system, Eqs. (3.5) or (3.6), no longer contain all the information necessary for the policy to decide on an appropriate action, i.e., the Markov property is violated. One response to this problem is to augment the observation with past actions equal to the length of the delay \([117]\). This allows the agent to become aware of previous actions it has taken when making decisions on what action to take next. When augmentation is used, the observation for velocity-based
guidance becomes
\[ o_n = \begin{bmatrix} x_n^f & x_n^f & \ddot{x}_{n-1}^{f,d} & \ddot{x}_{n-2}^{f,d} & \ldots & \ddot{x}_{n-D}^{f,d} \end{bmatrix}^T \] (3.7)
for a delay of \( D \) time steps. When the acceleration-based deep guidance model is used, the augmented observation is:
\[ o_n = \begin{bmatrix} x_n^f & x_n^f & v_n^f & v_n^f & \ddot{x}_{n-1}^{f,d} & \ddot{x}_{n-2}^{f,d} & \ldots & \ddot{x}_{n-D}^{f,d} \end{bmatrix}^T \] (3.8)

Although state augmentation allows for optimal policies to be found despite system delays, if the delay is too long the observation may grow too large for learning to be tractable.

### 3.4.4 Dynamics Model

The policy is trained within a kinematics environment to remove any overfitting to simulated dynamics or a specific controller. To measure the learning performance, the trained policy is periodically evaluated on an environment with full dynamics and a controller, as shown in Figs. 3.1 and 3.2. In other words, the trained deep guidance policy is “deployed” to another simulation for evaluation in much the same way that it is deployed to an experiment in Sec. 3.6. When a velocity-based guidance signal is issued, it is tracked using a proportional controller of the form
\[ u_n = K_p \left( \dot{x}_n^{f,d} - v_n^f \right) \] (3.9)
where \( K_p = \text{diag}\{0.1, 0.1\} \). The \( K_p \) values were chosen by trial-and-error until satisfactory performance was achieved. When the acceleration-based deep guidance configuration is used, an integral controller is used of the form
\[ u_n = u_{n-1} + K_I \left( \ddot{x}_n^{f,d} - \dot{v}_n^f \right) \] (3.10)
with \( K_I = \text{diag}\{0.5, 0.5\} \), also chosen by trial-and-error. Note that only planar translational motion is commanded due to coupling between the \( x, y, \) and \( z \) axes, as discussed in Sec. 3.6.3.
Regardless of which guidance method is used, the associated controller outputs a control effort that is executed on the same dynamics. A planar double-integrator dynamics model is used to simulate the follower motion. The simplicity of this model compared to the actual quadrotor dynamics will further show how this technique can handle dynamic differences between simulation and reality. The accelerations due to the control forces are

\[ \ddot{x} = \frac{F_x}{m} \]  \quad (3.11)  

\[ \ddot{y} = \frac{F_y}{m} \]  \quad (3.12)

where \( F_x \) and \( F_y \) are the forces applied in the \( X \) and \( Y \) directions, respectively, \( m \) is the follower mass, \( \ddot{x} \) and \( \ddot{y} \) are the accelerations in \( X \) and \( Y \), respectively. The accelerations are numerically integrated twice to obtain the position and velocity at the following time step.

The following subsection discusses the reward function used to incentivize the deep guidance policy to learn the desired behaviour.

### 3.4.5 Reward Function

At each time step, the follower receives rewards according to the state and the action taken. It is the designer’s role to craft this reward function to encourage the desired behaviour. Note that the reward function may be based off of the state even though the policy only receives an observation of the state and not the underlying state itself. In this work, the follower quadrotor is tasked with moving to a location three metres offset from the target quadrotor. The reward function has three components:

1. Rewards are given according to the follower position. If the follower moves in the direction of the desired location, it receives a positive reward.

2. Penalties are given for the follower colliding with the target.

3. For the velocity-based deep guidance only: penalties are given for high velocities near the desired location to discourage overshooting.
To calculate the position reward, a reward field $f$ is generated.

$$f(x_n) = -|x'_n + \begin{bmatrix} 3 \cos(\psi^i_n) & 3 \sin(\psi^i_n) \end{bmatrix} - x^f_n|$$ (3.13)

The first two terms represent the desired location. When the follower is at the desired location, the reward field is zero. It decreases linearly away from the desired state. The difference in the reward field between the current and previous time step is used to calculate the reward given to the agent. A positive reward is therefore given if the action chosen brings the follower closer to the desired location, and a negative reward otherwise.

$$r_n = \|K(f(x_n) - f(x_{n-1}))\|$$ (3.14)

the states are weighted with $K = \text{diag}\{125, 125\}$, determined by trial-and-error. A penalty, $r_{\text{collide}} = 15$, is given if the follower and target collide to encourage the reward-seeking follower to move to the desired location safely.

For the velocity-based guidance approach, the follower often overshoots the desired location. To reduce this overshoot, a penalty is given proportional to the guided velocity signal. This value is divided by the distance to the desired location such that high velocities far from the desired location are not severely penalized.

To summarize, the reward function for the velocity-based deep guidance strategy is

$$r_n = \begin{cases} \|K(f(x_n) - f(x_{n-1}))\| - c_1 \frac{\|z^d_n\|}{\|f(x_n)\| + \eta} - r_{\text{collide}} & \text{for } \|x'_n - x^f_n\| \leq 0.3 \\ \|K(f(x_n) - f(x_{n-1}))\| - c_1 \frac{\|z^d_n\|}{\|f(x_n)\| + \eta} & \text{otherwise} \end{cases}$$ (3.15)

with a small constant $\eta = 0.01$ to avoid dividing by zero and a velocity penalty weight $c_1 = 0.5$ such that it does not dominate the reward function. Collisions are said to occur when the quadrotors come within 30 cm of each other, i.e., $\|x'_n - x^f_n\| \leq 0.3$. The reward function for the acceleration-based deep guidance strategy does not include
the velocity-penalizing term and is therefore

\[ r_n = \begin{cases} 
    \|K(f(x_n) - f(x_{n-1}))\| - r_{\text{collide}} & \text{for } \|x_n - x_{n-1}\| \leq 0.3 \\
    \|K(f(x_n) - f(x_{n-1}))\| & \text{otherwise}
\end{cases} \]  \hspace{1cm} (3.16)

As expected, the reward function for the acceleration-based deep guidance approach
in Eq. (3.16) is simpler than the velocity-based reward function in Eq. (3.15). A sim-
pler reward function is possible since overshoots will be experienced during training
(with acceleration-based guidance) and therefore the policy will learn how to prevent
them, as opposed to needing a user-designed term in the reward function to prevent
overshoots.

The learning algorithm details are presented in the following subsection.

3.4.6 Learning Algorithm Implementation Details

The policy and value neural networks are equipped with 400 and 300 neurons in
their first and second hidden layers, respectively—Fig. 2.1 is not to scale. The neural
network architecture was chosen according to the original DDPG paper [90]. The
network size may need to be adjusted based on the complexity of the problem, but
adjustments were not needed in this work. The action input to the value network skips
the first hidden layer, as this was empirically shown to yield better results [83, 90].
Rectified linear units are used as the nonlinear activation functions within each neuron
in both hidden layers, shown below

\[ g(y) = \begin{cases} 
    0 & \text{for } y < 0 \\
    y & \text{for } y \geq 0
\end{cases} \]  \hspace{1cm} (3.17)

A \( g(y) = \tanh(y) \) nonlinear activation function is used in the output layer of the
policy network to ensure the guided velocity or acceleration is bounded; it is then
scaled to the action range. The output layer of the value network uses a softmax
function to ensure the output is indeed a valid probability distribution

\[ g(y_i) = \frac{e^{y_i}}{\sum_{k=1}^{B} e^{y_k}} \quad \forall \ i = 1, \ldots, B \] (3.18)

for each element \( y_i \) and for \( B \) bins in the value distribution. Fifty-one evenly-spaced bins are used, \( B = 51 \), inspired by the original value distribution paper [84]. Increasing \( B \) yields more granular reward predictions but increases the learning burden. The value bounds, within which the bins are evenly divided, were empirically found to be \([-5000, 0]\) for the velocity-based guidance and \([-200, 300]\) for the acceleration-based guidance. The stochastic gradient-descent optimization routine named Adam [91] is used to train the policy and value networks. Learning rates of \( \alpha = \beta = 0.0001 \) are used. The observations and action inputs are normalized before passing through the networks to avoid the vanishing gradients problem [118]. The replay buffer \( R \) holds \( 10^6 \) samples, and during training, mini-batches of size \( M = 256 \) are used. The smoothed network parameters are updated each training iteration with \( \epsilon = 0.001 \).

The standard deviation of the noise applied to the actions during training to force exploration is \( \sigma = \frac{1}{3} \left[ \max(a) - \min(a) \right] (0.99998)^E \), where \( E \) is the episode number. Selecting this standard deviation empirically leads to good exploration of the action space, and decaying the exploration as episodes continue refines the search space. Ten actors \( K = 10 \), a discount factor of \( \gamma = 0.99 \), a dynamics delay of length \( D = 3 \), all selected by trial-and-error. A time step of 0.2 seconds were chosen. N-step return lengths (discussed in Sec. 2.3) of \( N = 1 \) and \( N = 2 \) were used for the velocity- and acceleration-based guidance strategies, respectively. The Tensorflow [92] machine learning framework was used to generate, train, and evaluate the neural networks.

Every five training episodes, the current policy is “deployed” and run in a full dynamics environment with a controller, as described in Sec. 3.4.4. During deployment, \( \sigma = 0 \) in Eq. (2.9) such that no exploration noise is applied to the deep guidance velocity or acceleration signals.
3.5 Simulation Results

To determine which deep guidance strategy is most effective, both the velocity-based deep guidance and acceleration-based deep guidance policies are trained in simulation. Both are trained on the same task—that is, to learn how to guide a follower quadrotor from a randomized initial position to a position three metres offset from the front-face of the target quadrotor. While this task is easily accomplished by conventional guidance and control techniques, it serves to determine the suitability to use deep reinforcement learning for quadrotor guidance and to compare the velocity- and acceleration-based deep guidance approaches. The initial conditions of the target and follower are

\[
\begin{align*}
\mathbf{x}_0^t &= \begin{bmatrix} 0 \text{ m} & 0 \text{ m} & 0 \text{ rad} \end{bmatrix} + \mathcal{N}(0, 1) \quad (3.19) \\
\mathbf{v}_0^t &= \begin{bmatrix} 0 \text{ m/s} & 0 \text{ m/s} & 0 \text{ rad/s} \end{bmatrix} \quad (3.20) \\
\mathbf{x}_0^f &= \begin{bmatrix} 0 \text{ m} & 2 \text{ m} \end{bmatrix} + \mathcal{N}(0, 1) \quad (3.21) \\
\mathbf{v}_0^f &= \begin{bmatrix} 0 \text{ m/s} & 0 \text{ m/s} \end{bmatrix} \quad (3.22)
\end{align*}
\]

In other words, the target and follower initial positions are randomized around their nominal locations on each episode. This forces the policy to become robust to a variety of initial conditions.

3.5.1 Velocity-based Guidance Results

The velocity-based deep guidance system was trained on the quadrotor pose tracking task. A learning curve is shown in Fig. 3.4a. The learning curve shows the total rewards received per episode as a function of how many training episodes were completed. Deep reinforcement learning aims to increase the average rewards per episode as training progresses. The learning curve increases as training progresses, indicating that the task is being learned. Similarly, the loss function decreases in Fig. 3.4b. Sample follower trajectories are shown in Fig. 3.5. The shaded quadrotor represents the initial follower position, the solid one represents its final location, and the dashed line shows its trajectory. The shaded rotors on the target represent its front—the
chaser is tasked with moving to three metres in front of the target. Even once the learning curve and loss function reached their plateaus, indicating that training was complete, significant overshoots and steady-state error are observed when individual velocity-based guidance trajectories are plotted.

### 3.5.2 Acceleration-based Guidance Results

The acceleration-based deep guidance system was also trained on the identical quadrotor task. A learning curve is shown in Fig. 3.6a, and the associated loss function is shown in Fig. 3.6b. The learning curve shows the rewards received as a function of episodes (simulated attempts) and reaches a plateau indicating that peak performance has been reached. The loss function describes how well the critic performs versus the number of back-propagation training iterations, and reaches a near steady-state value when training was halted. The learning curve increases as expected. Sample trajectories are shown in Fig. 3.7, and show that the acceleration-based deep guidance strategy effectively learned to solve the quadrotor pose tracking task. Figure 3.8 shows a time-series comparison of the scalar position error between the follower and the desired location—3 m away from the target. Both the velocity- and acceleration-based deep guidance strategies are shown for comparison. As expected, the acceleration strategy
Figure 3.5: Visualization of velocity-based follower trajectories at various episodes once training is complete.
Figure 3.6: Acceleration-based deep guidance training progress.

exhibits less overshoot and steady-state error than the velocity strategy. In addition, the acceleration-based reward function was simpler as it did not require additional terms to try and artificially dampen the overshoots, as shown in Eqs. (3.15) and (3.16). It is likely that the overshoot-dampening term of the velocity-based reward function contributed to the steady-state offset seen in Fig. 3.8. It can be concluded that for this application, and likely all second-order systems, it is more appropriate to use acceleration-based guidance as opposed to velocity-based guidance. The root cause is hypothesized to be: a deep guidance system that issues desired velocity signals is trained in a first-order kinematics environment and therefore does not encounter key factors that second-order systems possess, like momentum. Therefore, since the velocity-based deep guidance system has been trained in an environment when it can realize any velocity immediately, it is inappropriate to apply such a technique to a second-order system—large overshoots will inevitably occur even with additional reward function crafting. With acceleration-based deep guidance, the system learns that it must apply a negative acceleration before reaching the desired location to avoid overshoots because it sees this during training. In addition, desired accelerations can be realized much more quickly than desired velocities. Having a system be able to track the deep guidance signals accurately is crucial for the high-level problem-solving abilities of deep reinforcement learning for robotics to be realized and useful.
Figure 3.7: Visualization of acceleration-based follower trajectories at various episodes once training is complete.
The simulations are validated experimentally in the following section.

3.6 Experimental Validation

To explore whether the proposed deep guidance method detailed above will allow for the policies trained entirely in simulation to be directly transferred to real robot platforms, the guidance policies trained in simulation are executed on real quadrotors in an experimental facility at École Nationale de l’Aviation Civile (ENAC). In addition, an experiment is performed on a hexacopter (without retraining the guidance system) to demonstrate transferability of the technique to other vehicles. Details of the indoor quadrotor experimental facility are presented, followed by the experimental setup and results.

3.6.1 Experiment Facility

Quadrotors are flown in an indoor facility at ENAC. The quadrotors are named Explorer 1 and 2, and are shown in Fig. 3.9a. Their mass is 535 g (including the battery), and their maximum thrust is 40 N. They are powered by a 3-cell battery at 11.1 V and 2300 mAh, which provides 15 minutes of flight time. The flight arena is 10 m × 10 m × 9 m with a mesh exterior, as shown in Fig. 3.9b. A 16-camera Optitrack system is used to track the motion of the quadrotors at sub-millimetre resolution in real-time and relays this information wirelessly to the quadrotors. The
Figure 3.9: ENAC facility and flight hardware.
Explorer vehicles used are equipped with the *Paparazzi Autopilot System* [119], an open-sourced software package for unmanned aerial systems. *Paparazzi* consists of a ground segment, running on a personal computer, an airborne segment, running on-board the quadrotor, and a communication link between them. The on-board *Tawaki* is shown in Fig. 3.9c has the characteristics listed in Table 3.1.

**Table 3.1:** General characteristics for the Tawaki v1.0 autopilot board

<table>
<thead>
<tr>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU</td>
<td>STM32F7</td>
</tr>
<tr>
<td>IMU</td>
<td>ICM20600 (accel, gyro) + LIS3MDL (mag)</td>
</tr>
<tr>
<td>Baro</td>
<td>BMP3</td>
</tr>
<tr>
<td>Serial</td>
<td>3 UARTS, I2C (5V + 3.3V), SPI</td>
</tr>
<tr>
<td>Servo</td>
<td>8 PWM/DShot output (+ ESC telemetry)</td>
</tr>
<tr>
<td>RC</td>
<td>2 inputs: PPM, SBUS, Spektrum</td>
</tr>
<tr>
<td>AUX</td>
<td>8 multi purpose auxiliary pins</td>
</tr>
<tr>
<td></td>
<td>(ADC, timers, UART, flow control, GPIO, ...)</td>
</tr>
<tr>
<td>Logger</td>
<td>SD card slot</td>
</tr>
<tr>
<td>USB</td>
<td>DFU flash, mass storage, serial over USB</td>
</tr>
<tr>
<td>Power</td>
<td>6V to 17V input (2-4S LiPo)</td>
</tr>
<tr>
<td></td>
<td>3.3V and 5V, 4A output</td>
</tr>
<tr>
<td>Weight</td>
<td>12 grams</td>
</tr>
</tbody>
</table>

### 3.6.2 Experimental Setup

The simulations designed and presented in Sec. 3.5 are replicated experimentally. The same quadrotor motion task is presented, and both the velocity-guidance and the acceleration-guidance are compared. The final parameters $\theta$ of the policy network trained in simulation are exported for use in experiment; no further training is performed.

The quadrotors take off under their regular autopilot software and move to their initial conditions. Then, once at zero velocity, the follower quadrotor is switched into deep guidance mode where it listens for guided desired velocities or accelerations from the policy network, and then uses feedback control to track that velocity or acceleration signal. The follower quadrotor is tasked to move from its initial location to three
metres offset from the front-face of the target quadrotor. There are many discrepancies between the simulated environment within which the policy is trained and the experimental facility where its performance is evaluated. The simulated environment did not model the vehicles as quadrotors—they were modelled as double-integrator point masses. Rotor dynamics, air disturbances, and sensor inaccuracies were unmodelled. In addition, mass, size, and the on-board controllers that track the velocity and acceleration signals were different than the ones used during evaluation of the policy performance in simulation. Incremental nonlinear dynamic inversion control [101] is used as the on-board controller to track the commanded velocities or accelerations. Dramatic discrepancies exist between the simulated and experimental environments. Therefore, this is an excellent test of the simulation-to-reality capabilities of the deep guidance technique, and is an appropriate facility to compare the two proposed deep guidance solutions.

3.6.3 Experimental Results

The experimental results are shown in Fig. 3.10. Both the velocity- and acceleration-based deep guidance approaches have successfully transferred from simulation to experiment, as the trajectories in Fig. 3.10a and 3.10b show. The acceleration-based model was expected to outperform the velocity-based one in terms of overshoot, as discussed in Sec. 3.5.2. The steady-state error the velocity-based model suffers from is likely due to the velocity-based reward function in Eq. (3.15), where a term was included to penalize high velocities near the desired location, weighted by $c_1$, in order to reduce overshoots. However, it appears that the term also discourages the final steady-state error to be low, since the velocity needed to move to the desired location may result in more penalties than rewards. In trying to prevent overshoot, this term caused steady-state offset, though perhaps with additional tuning of the $c_1$ parameter this could be prevented. The result is further evidence that a simpler reward function is best, and the acceleration-based deep guidance approach allows for a simpler reward function with better performance.

A time-series view of the results is shown in Fig. 3.10c, where the acceleration-based experiment outperforms the velocity-based one both in terms of overshoot and
Figure 3.10: Experimental results.
steady-state offset. To further test the capabilities of the learned deep guidance system, the target is moved twice in experiment—a task that was not encountered during training. Both deep guidance systems continue to track the desired location even when the target is moved. To further test the acceleration deep guidance model’s ability to effectively transfer from simulation to reality, the same acceleration model used on the quadrotor is also is executed on a hexacopter (chosen because it was readily-available). Results, shown in Fig. 3.11, show that the same guidance logic is applicable to an entirely different vehicle—a fact that would not be possible if deep reinforcement learning was used to directly calculate rotor torques from observations (i.e., if deep reinforcement learning was tasked with guidance and control).

Both the velocity- and acceleration-based deep guidance models performed worse in experiment than in simulation. Many differences between the simulator the policy was trained within and the experimental facility exist, such as: the quadrotors have significantly more complex dynamics than the double-integrator model that was used in training; the experimental quadrotor mass, thrust, perturbations, and controller were different than the policy was evaluated on in simulation; significant accelerometer noise was encountered; and dynamic coupling between the axes was experienced in experiment. Although the experimental results were slightly worse than the simulated ones, it demonstrates the ability of the deep guidance system to transfer from
simulation to reality even when significant dynamic differences are present. Altitude commands from the deep guidance system were attempted but ultimately led to poor performance due to the dynamic coupling that exists between the axes. Future work should examine how deep guidance can be applied to systems where coupling exists (this is considered in Chap. 5).

Videos from the two experiments can be found at: https://youtu.be/36S_s0fTc-0. All the code used in this work is available at https://github.com/Kirkados/AIAA_GNC_2021.

3.7 Conclusion

This paper improved on previous work where deep reinforcement learning was applied to the guidance problem for robotics. It used deep reinforcement learning to perform closed-loop guidance, named deep guidance, and compared whether it is more appropriate for velocity- or acceleration-based guidance signals to be issued. A simulated quadrotor task was designed where one quadrotor starts at a randomized initial state and must move itself in front of another quadrotor. Results conclusively showed that acceleration-based deep guidance is more effective at guiding the motion and it allowed for a simpler reward function. Training the system using kinematics that are of the same order as the dynamics within which the policy will be implemented appears to be important. To demonstrate the ability of the deep guidance system to transfer to reality, the trained policy is transferred from simulation to a real quadrotor facility at École Nationale de l’Aviation Civile in Toulouse, France. The two quadrotors performed flights in an indoor facility such that the velocity- and acceleration-based guidance systems could be compared. The experimental results confirmed that the acceleration-based deep guidance approach is more appropriate, and it confirmed the simulation-to-reality transfer abilities of the deep guidance approach. Future work should explore the use of the acceleration-based deep guidance technique on more difficult problems in robotics (this is considered in Chaps. 4 and 5), and investigate the use of the deep guidance technique in scenarios where dynamic coupling exists (this is considered in Chap. 5).
Acknowledgments

This research was financially supported in part by the Natural Sciences and Engineering Research Council of Canada under the Postgraduate Scholarship-Doctoral PGSD3-503919-2017 award and the Ontario Graduate Scholarship.

Continuity to Next Chapter

This paper determined that a deep reinforcement learning-based guidance strategy performs better when desired accelerations are issued instead of desired velocities. It also determined how to apply the technique to quadrotors that experience time delays. The next paper builds the technique further by applying it to a cooperative multi-agent quadrotor runway exploration task.
Chapter 4

Learned Multi-agent Real-time Guidance with Applications to Quadrotor Runway Inspection

This chapter is under peer-review at the journal Field Robotics. The following paper citation is preferred, and this thesis should only be cited if the appropriate citation for the included article cannot be found:


The paper is included in this thesis with minor formatting changes and variable renaming (for consistency between chapters). The DRL theory section was removed to avoid repetition, with the reader being referred to Sec. 2.3. This paper was co-authored by the thesis author, Kirk Hovell, his thesis supervisor, Prof. Steve Ulrich, and a colleague, Prof. Murat Bronz. Mr. Hovell set-up, developed, wrote, and evaluated the relevant code; obtained and analyzed simulated results; prepared software for experimental demonstrations; analyzed experimental results; and wrote the manuscript. Theory development and manuscript editing were conducted jointly. Prof. Bronz was responsible for obtaining experimental results.

Paper Context

This third paper extends the deep reinforcement learning-based guidance technique (using desired accelerations, as the previous Chapter concluded was best) to a multi-agent cooperative quadrotor runway exploration scenario. This represents the first application of the technique to a difficult and unsolved scenario—where the benefits of bringing deep reinforcement learning to aerospace robotics begin to materialize. A
A video summary of the paper can be found at https://youtu.be/Pu5rWnLgyZs.

4.1 Abstract

Aircraft runways are periodically inspected for debris and damage. Instead of having pilots coordinate the motion of the quadrotors manually, or hand-crafting the desired quadrotor behaviour into a guidance law, this paper proposes the use of deep reinforcement learning to learn a closed-loop multi-agent real-time guidance strategy for quadrotors to autonomously perform such inspections. This yields a significant reduction in engineering effort while enabling highly-flexible real-time performance. The runway is discretized into a number of rectangular tiles, which must all be visited for the runway to be considered inspected. The proposed guidance system calculates a desired acceleration in real-time for the quadrotor(s) to track in order to complete the task. This paper first develops the guidance technique, trains it in simulation, and evaluates it experimentally using an indoor quadrotor laboratory at École Nationale de l’Aviation Civile. This process is then repeated for an outdoor setting on a real runway, where the proposed guidance strategy is compared to a hand-crafted strategy. The guidance technique is then applied to a multi-quadrotor scenario in the outdoor environment, where real-time adaptation to the failure of one quadrotor is successfully demonstrated. Additional simulation results demonstrate the technique is viable in a swarm of up to 12 quadrotors and on a variety of exploration area shapes. This work shows how modern learning-based techniques can: 1) reduce the engineering effort required to design complex guidance systems; 2) outperform classical techniques while being more robust; and 3) be implemented on real hardware in a representative outdoor environment.

4.2 Introduction

Airport runways are inspected multiple times per day to ensure no debris or damage exists that may pose a safety hazard to aircraft. Typically, takeoffs and landings are paused to allow for a ground vehicle to safely drive down the runway to perform a
visual inspection. To minimize delays to customers, these runway shutdown durations should be minimized. Recently, quadrotors have been flown by pilots for this task [120, 121]. Quadrotors have become useful in recent years due to the increased availability and reduced costs of microcontrollers, batteries, and 3D printing. Common quadrotor applications are: search and rescue, surveillance, package delivery, defence, sport racing, and research. This work proposes using multiple quadrotors to quickly and autonomously inspect a runway. The quadrotors could capture images of the runway which could be merged into a composite image of the entire runway and analyzed for debris, damage, and, over time, longer-term runway problems. The autonomous acquisition of such images must rely on an intelligent trajectory guidance and control system. A variety of quadrotor guidance and control theories have been developed for such purposes. Backstepping control [96], sliding mode control [97], and dynamic inversion techniques [99–102] have all been applied to quadrotor control. Guidance, also known as path planning, has been demonstrated for individual quadrotors [105, 106, 108] and swarms of quadrotors [107, 122]. While the quadrotor flight is autonomous using these advanced guidance and control techniques, the techniques themselves are hand-crafted by researchers and may require significant engineering effort. In addition, many hand-crafted guidance strategies are complex and their trajectories must be generated offline for open-loop use. This work, in contrast, allows for autonomous multi-agent flight to be learned, rather than designed. Once the offline computationally-intensive learning process is complete, the resulting guidance model can easily generate trajectories in real-time.

Deep reinforcement learning is a branch of artificial intelligence that, through trial-and-error, discovers an appropriate behaviour to take in a given circumstance and generalizes this behaviour well to new experiences. The trial-and-error behaviours are given the notion of good or bad through a human-designed reward function, which may be very simple. For example, giving a reward of +1 for winning a chess game, a reward of -1 for losing a chess game, and a reward of 0 for every move during the game has led to the most powerful chess engine in existence [123]. The notion of delegating the learning to the reinforcement learning algorithm, rather than hand-crafting the desired behaviours, can lead to entities that outperform humans in a variety of tasks.
and games [26, 28, 72, 123]. In principle, it removes the burden of the designer from: 1) knowing how to solve the problem; and 2) determining how to encode that behaviour.

Training reinforcement learning algorithms on real robots is difficult due to the trial-and-error nature of the learning process. Even with state-of-the-art fast-learning algorithms [44, 45], a robot may require hundreds or thousands of attempts before the learning succeeds, which is expensive and may lead to significant wear-and-tear on the robot. An alternate method is to train the reinforcement learning algorithm in simulation and to deploy the trained model to a real robot. However, problems arise with this approach since the simulated environment cannot perfectly match the real environment. The learned model overfits the simulated dynamics and performs poorly in the real world; this is known as the simulation-to-reality problem. Efforts to get around this problem involve randomizing the environmental parameters during training, known as domain randomization [51–53, 109–112, 124], or fine-tuning once deployed to experiment [47, 112, 113].

Multi-agent deep reinforcement learning has also been explored in previous work [28, 125, 126]. Typically, each agent present learns its own behaviour model. However, this causes the environment to become “nonstationary,” i.e., for a given agent, the environment, which includes the other agents’ behaviour, changes with time and slows learning. Inspired by recent work [127–129], this paper shares a single behaviour model among all agents. As discussed in Sec. 4.4.7, this allows agents to implicitly share behaviour knowledge while increasing the data collection rate and removing the nonstationarity from the environment, as proven by [127].

Due to the inherent advantages discussed above, reinforcement learning has been used in quadrotor trajectory guidance and control applications. It has been used to learn an inner-loop controller that outperformed classical control techniques [36, 130], a stabilization control task that included a PD controller [35], and, along with domain randomization, has been used to enable real quadrotor flight [51, 52]. Other learning techniques have been used to guide quadrotors through a maze with a discrete set of nodes [115, 116]. Reinforcement learning was used to train a basic vision-based guidance system in experiment [131, 132]. Lastly, a multi-agent stochastic wildfire surveillance guidance technique was developed with simulated results [37].
Unless otherwise stated, the above applications used deep reinforcement learning to learn a combined guidance and control strategy. In other words, control efforts for the robot to execute were calculated directly from the system state—the intermediate guidance and control aspects were combined into a single end-to-end calculation. However, it is possible that the guidance portion (which calculates the trajectory) of the learned system is appropriate for solving the task but the control portion (which commands the actuators to track the trajectory) has overfit the simulated dynamics and reduces performance. In this context, and as suggested by [59], it may therefore be beneficial to restrict reinforcement learning to learn a guidance behaviour only, with the control aspect being left to the well-developed control theory community.

The authors’ past work consisted of restricting the reinforcement learning algorithm to exclusively learn a guidance policy, and letting an on-board controller handle any errors encountered while crossing the simulation-to-reality gap. It calculated closed-loop real-time velocity-based guidance signals for spacecraft proximity operations (in Chap. 2) [1], and was improved for use on quadrotor proximity operations where acceleration-based guidance signals were calculated (in Chap. 3) [133]. This previous work considered single-agent scenarios only. The authors named the deep reinforcement learning-based guidance technique deep guidance.

In this work, the deep guidance technique is extended to a multi-agent application and is used to generate real-time acceleration commands for multiple quadrotors to guide their inspection of a runway. Such a system is trained in simulation and tested in two experimental facilities at École Nationale de l’Aviation Civile in Toulouse, France. An indoor facility is first used for proof-of-concept experiments to develop the runway inspection technique. Then, outdoor GPS-driven real-world quadrotor runway inspection experiments are presented. The proposed guidance technique is also compared to a hand-crafted technique. In addition, the fault-tolerance of a two-quadrotor model is explored, the use of quadrotor swarms is studied, and the applicability of the technique to non-rectangular shapes also is presented.

The original contributions of this paper are:

2. The application and analysis of the technique to both indoor and outdoor quadrotor experimental facilities.

This paper is organized as follows: Sec. 4.3 presents background on deep reinforcement learning and the specific learning algorithm used in this paper, Sec. 4.4 discusses the multi-quadrotor runway inspection scenario considered and how the deep guidance technique is applied to it, Sec. 4.5 presents numerical simulations showing the training of the system and simulated performance, Sec. 4.6 presents experimental results in both the indoor and outdoor facilities, and Sec. 4.7 concludes this paper.

4.3 Markov Decision Processes and Deep Reinforcement Learning

To avoid repetition, this section has been removed; please see Sec. 2.3 for a description of deep reinforcement learning and the D4PG algorithm used in this work.

4.4 Problem statement

This section discusses the multi-quadrotor runway inspection environment within which the deep guidance system will be trained. It then presents the deep reinforcement learning implementation for the environment. In contrast to the authors’ previous work, where the deep guidance technique was developed using simple tasks with shaped reward fields [1, 133], the deep guidance technique is herein used to learn a task that is significantly more difficult and has a more abstract, sparse, reward function.

The task is presented as follows: an arbitrary number of quadrotors, \( Q \), must inspect a runway. The runway is divided into a number of tiles, as shown in Fig. 4.1. Each quadrotor has full knowledge of its and the other quadrotors’ positions and velocities. Each time a previously-undiscovered runway tile is inspected, a reward of +1 is given to all quadrotors. A small penalty is given to quadrotors who become too close to each other to discourage collisions. Through attempting to maximize the rewards received, the learning algorithm will, in turn, learn an approach to cooperatively inspect the runway.
The position and velocity of the \(i\)th quadrotor at time step \(n\) are

\[
x^i_n = \begin{bmatrix} x^i_n \\ y^i_n \end{bmatrix} \quad (4.1)
\]
\[
v^i_n = \begin{bmatrix} v^i_{xn} \\ v^i_{yn} \end{bmatrix} \quad (4.2)
\]

where \(x\) and \(y\) represent the positions in Cartesian space and \(v\) is the velocity. Only planar motion is considered. The matrix containing the state of the runway, where each element corresponds to a runway tile in Fig. 4.1, has the form:

\[
S_n = \begin{bmatrix}
s_{11} & s_{12} & \cdots & s_{1S_W} \\
s_{21} & s_{22} & \cdots & s_{2S_W} \\
\vdots & \vdots & \ddots & \vdots \\
s_{SL1} & s_{SL2} & \cdots & s_{SLS_W}
\end{bmatrix} \quad (4.3)
\]

where \(s_{jk} = \{0, 1\}\) represent the state of the \(j\)th row and \(k\)th column tile of the runway, and \(S_L\) and \(S_W\) represent the number of tiles in length and width, respectively. A value of 0 represents the tile being uninspected and a value of 1 indicates the tile has been inspected.
4.4.1 Deep guidance

The deep guidance technique allows for reinforcement learning to be used on real robot platforms, despite being trained entirely in simulation, by limiting deep reinforcement learning to only learn the guidance portion of the guidance, navigation, and control process \([1, 133]\). This constitutes the authors’ solution to the simulation-to-reality gap. The learned closed-loop guidance system calculates an acceleration signal at each time step, and passes that acceleration to a conventional controller to track. The controller’s ability to track the acceleration signal, regardless of modelling errors, is the motivation to restrict the reinforcement learning algorithm to learn guidance only.

Figure 4.2a shows a block-scheme diagram of the approach. The desired acceleration signal for quadrotor \(i\) is \(\ddot{x}_n^{i,d}\) and the control effort is \(u_n^i\). To prevent the deep guidance policy from overfitting a particular controller during training, an ideal controller is assumed. The ideal controller and dynamics blocks can be combined into a single kinematics block, as shown in Fig. 4.2b. Any controller may be used along with the deep guidance system to control a real robot, so long as it accurately tracks the desired accelerations.

4.4.2 Kinematics model

The deep guidance policy input observation for the \(i^{th}\) quadrotor is:

\[
o_n^i = \begin{bmatrix} x_n^i & v_n^i & x_{n+1}^i & v_{n+1}^i & \ldots & x_{n+\text{mod}(i+Q-1, i)} & v_{n+\text{mod}(i+Q-1, i)} & e^T(I \otimes S_n)^T \end{bmatrix}^T
\]  

(4.4)
for $Q$ quadrotors, where $I$ is an identity matrix of size $S_W \times S_W$ and $e$ is a column matrix of length $S^2_W$ filled with zeros except for ones in rows $(j - 1)S_W + j \ \forall \ j = 1, 2, \ldots, S_W$ and $\otimes$ is the Kronecker product; the final term reshapes the runway state matrix in Eq. (4.3) into a row vector. When this observation is passed through the deep guidance policy in Eq. (2.1), an action, which in this application is an acceleration signal, $\ddot{x}_{i;d}^n$, is returned. During the training it is assumed that the ideal controller perfectly achieves the desired acceleration signal. Therefore, the guided acceleration signal is integrated twice to obtain the next state. All integration is performed using the Scipy [89] Adams/Backward differentiation formula methods in Python. When delays are included, as discussed in the following subsection, the guided acceleration is stored and the acceleration $D$ time steps old is used instead.

4.4.3 Time delay consideration

Delays may arise from actuation delays, signal delays, or measurement delays. In the event that the desired acceleration is not immediately realized by the quadrotor, the observation no longer contains sufficient information for an appropriate action to be calculated—the Markov assumption has been violated. If the observation is “augmented” with past actions equal to the number of time steps of system delay present, the problem can be alleviated [117]. With a delay of $D$ time steps, the
system’s augmented observation becomes

\[
O_n^i = \begin{bmatrix}
    x_n^i \\
    v_n^i \\
    x_{n+1}^i \\
    v_{n+1}^i \\
    \vdots \\
    x_{n \mod (i+Q-1, i)}^i \\
    v_{n \mod (i+Q-1, i)}^i \\
    \ddot{x}_{n-1}^i \\
    \ddot{x}_{n-2}^i \\
    \vdots \\
    \ddot{x}_{n-D}^i \\
    (I \otimes S_n)e
\end{bmatrix}
\] (4.5)

Although state augmentation allows for optimal policies to be found despite system delays, if the delay is too long or too many quadrotors are used, the observation may grow too large for learning to be tractable, as explored in Sec. 4.5.4.

### 4.4.4 Dynamics model

To prevent the deep guidance policy from overfitting any simulated dynamics or controller, it is trained within the kinematics environment presented in Sec. 4.4.2. Occasionally, however, its performance must be evaluated in a dynamics environment with a controller in much the same way that it will be evaluated once deployed to a real quadrotor experiment. Figure 4.2a shows the type of simulation the policy is tested within. A planar double-integrator dynamic model is used

\[
\ddot{x}^i = \frac{u_i^x}{m_i} \quad (4.6)
\]

\[
\ddot{y}^i = \frac{u_i^y}{m_i} \quad (4.7)
\]
where \( u^i_x \) and \( u^i_y \) are the forces applied to the \( i^{th} \) quadrotor along the \( X \), and \( Y \) axes, \( m^i \) is its mass, and \( \ddot{x}^i \) and \( \ddot{y}^i \) are its accelerations in \( X \) and \( Y \), respectively. The accelerations are numerically integrated twice to obtain the position at the following time step. Note that only planar \( X \) and \( Y \) translational motion is considered since altitude changes are not needed for runway inspection.

An integral controller of the form

\[
\begin{align*}
  \mathbf{u}^i_n &= \mathbf{u}^i_{n-1} + K_I \left( \ddot{x}^i_{n,d} - \dot{v}^i_n \right) \\
  &\quad (4.8)
\end{align*}
\]

is used in simulation, with \( K_I \) as an integral gain matrix, chosen by trial-and-error, and \( \mathbf{u} = [u_x \ u_y]^T \). Each quadrotor’s controller outputs a control effort that is executed on its simulated dynamics. The kinematics model used for training and the dynamics model used for evaluation are vastly simpler than the true dynamics of a quadrotor. This large discrepancy highlights the ability of the deep guidance technique to handle differences between simulation and reality.

### 4.4.5 Reward function

The quadrotors are tasked with autonomously and cooperatively learning how to inspect an aircraft runway. The following logic is used to convey the desired behaviour through a scalar reward returned to the quadrotors at each time step:

1. The runway is divided into a number of rectangular tiles in a grid that is \( S_L \) tiles long and \( S_W \) tiles wide, as shown in Fig. 4.1. The grid size was arbitrarily chosen; sensitivity to the grid size will be investigated in future work.

2. If any quadrotor visits a new grid tile that was previously-uninspected, a reward of +1 is given to all quadrotors.

3. If a quadrotor visits a previously-inspected tile, a reward of 0 is given.

4. A penalty is given to individual quadrotors who become too close to another quadrotor in order to encourage independent inspection and discourage collisions. If there are many quadrotors, the proximity to only the nearest quadrotor is used.
The runway state, \( S_n \), is initialized as a grid of zeros. When a tile becomes inspected, it is replaced with a one. The above logic is written as a scalar reward function for the \( i^{th} \) quadrotor at time step \( n \) through:

\[
r_i^n = \text{Tr}\{S_n^T S_n\} - \text{Tr}\{S_{n-1}^T S_{n-1}\} - c_1 e^{-\min_j \|x_{i,n}^j - x_{i}^n\| / c_2} \forall \ j \in Q \land j \neq i \tag{4.9}
\]

for some positive scaling constants \( c_1 \) and \( c_2 \), and where \( \text{Tr}\{\cdot\} \) represents the trace operation. The first two terms calculate the change in the runway’s state while the third term penalizes quadrotor proximity.

### 4.4.6 Learning algorithm implementation details

The policy and value neural networks have the same shape: 400 neurons in their first hidden layer and 300 neurons in their second—Fig. 2.1 is not to scale. For the value network, the action input skips the first hidden layer, as empirically this was shown to yield better results \([83,90]\). Neurons in all hidden layers use the rectified linear unit (ReLU) as their nonlinear activation function, shown below

\[
g(y) = \begin{cases} 
0 & \text{for } y < 0 \\
y & \text{for } y \geq 0 
\end{cases}
\tag{4.10}
\]

The output layer of the policy network uses a \( g(y) = \tanh(y) \) activation function to ensure the output is bounded, and is then scaled to the action range, and the output layer of the value network uses a softmax function to ensure it outputs a valid probability distribution

\[
g(y_j) = \frac{e^{y_j}}{\sum_{k=1}^B e^{y_k}} \quad \forall \ j = 1, \ldots, B \tag{4.11}
\]

for each neuron \( y_j \) and for the number of bins in the distribution, \( B \). As per the original value distribution paper, \( B = 51 \) \([84]\). The bins are divided evenly across the expected range of total rewards received during an episode: \([0, S_L S_W]\). The Adam \([91]\) stochastic gradient descent algorithm is used to train the neural networks, with learning rates of \( \alpha = \beta = 0.0001 \). To avoid the vanishing gradients problem
[118], the inputs to the neural networks are normalized. Mini-batches of $M = 256$ data points are randomly drawn from the replay buffer $R$, of size $10^6$ samples, to train the networks. A discount factor of $\gamma = 0.99$ and an N-step return of $N = 5$ are used. The parameters for the smoothed policy and value networks are updated on each training iteration with $\epsilon = 0.001$. All parameters were tuned using trial-and-error with the goal of speeding up learning. To force the $K = 10$ actors to inspect their environment, noise is applied to their chosen actions with a standard deviation of $\sigma = \frac{1}{3} \left[ \max(a) - \min(a) \right] (0.99998)^E$, where $E$ is the episode number. This empirically leads to good inspection of the action space, and the decaying noise refines the search space as learning progresses. A dynamics delay of length $D = 3$ is used, as this was found to be the delay present in both the indoor and outdoor experimental facility when the time step was 0.2 seconds. The Tensorflow\textsuperscript{1} machine learning framework is used to generate, train, and evaluate the neural networks.

After a designated agent performs five training episodes, the most up-to-date policy is "deployed" and run in a full dynamics environment with a controller, as described in Sec. 4.4.4. During deployment, $\sigma = 0$ in Eq. (2.9) such that no exploration noise is applied and the performance can be readily evaluated.

4.4.7 Multi-agent considerations

Coordinating the behaviour of multiple quadrotors requires careful consideration due to the nonstationarity of the environment. A nonstationary environment changes with time, which makes learning more difficult since the policy’s knowledge of the environment becomes invalid over time. If each agent present learns a separate policy, nonstationarity emerges since each agent attempts to learn the best actions according to the environment it experiences—the other agents are therefore viewed as part of the environment. As an example, Agent 1 will learn a behaviour based on its environment, which includes Agent 2’s behaviour. Then, Agent 2 will learn a behaviour based on its environment, which includes Agent 1’s new behaviour. Even though the environment is not changing, the changing agents cause information to “ring” between agents and leads to slow learning [127].

\textsuperscript{1}Software available from https://www.tensorflow.org
Instead of learning a separate policy for each agent, methods for sharing one policy between agents exist [128, 129]. The options are: a) just the policy can be shared; b) just the value network can be shared; or c) both the policy and value networks can be shared. Option c) is used in this paper, where a single policy and value network are shared among all agents. Through each agent using the same policy and value network, an understanding for the other agents’ behaviours is implicitly communicated. This removes the nonstationarity of the environment and increases the learning speed as proven in [127].

The observation of the system in Eq. (4.5) contains the positions and velocities of all quadrotors. When calculating the acceleration for quadrotor $i$, its own position $x^i_n$ and velocity $v^i_n$ are the first two entries in the observation, followed by the other quadrotors’ positions and velocities, followed by the $i$th quadrotor’s past actions (discussed in Sec. 4.4.3), followed by the runway state $S_n$ flattened into a column vector. By positioning a quadrotor’s own information in the observation first, the policy is hypothesized to learn patterns about how to coordinate the motion of that quadrotor given the knowledge of the others. The same policy can therefore be used for all quadrotors—the observation is simply tailored to each quadrotor. A downside to including all quadrotors’ state information in the observation is that the problem does not scale well. As more quadrotors are added, the observation grows and will eventually become intractable; this is explored in Sec. 4.5.4.

Through sharing the policy and value networks among all agents, the total number of neural networks needed is reduced and the data collection rate is increased. At each time step, all $Q$ quadrotors take actions and experience the environment. Therefore, each quadrotor’s experience can be separately assembled into an observation and logged in the replay buffer $R$ along with its action, reward, and next observation. Therefore, each of the $K$ parallel simulations generate $Q$ data points at each time step.

### 4.4.8 Fault-tolerance considerations

To further increase the flexibility and reliability of the multi-agent deep guidance approach, fault-tolerance was built into the deep guidance model. In other words,
a single model should complete the runway inspection task with two quadrotors. However, if one quadrotor fails, the same model should automatically command the remaining quadrotor to complete the runway inspection. Since all trajectories are generated in real-time (through the acceleration signals), the surviving quadrotor’s trajectory can be automatically modified upon a quadrotor failure.

To build in this fault-tolerance, it must be experienced during training. During the training of the dual-quadrotor model, one quadrotor is programmed to fail 50% of the time within the first 30 s of the episode. When a quadrotor fails, its position and velocity are fixed at a certain value and its collected data are no longer used for training purposes. As training progresses, the still-operational quadrotor will learn to recognize when the other quadrotor has failed. The result is a dual-quadrotor deep guidance model that can tolerate quadrotor failures automatically. This fault-tolerance is in addition to the other benefits of the deep guidance approach: flexibility in initial condition, real-time trajectory generation, and automatic cooperation between quadrotors—all of which are learned. When the outdoor dual-quadrotor model is evaluated, both in simulation (Sec. 4.5.2) and experiment (Sec. 4.6.5), the ability of the policy to both inspect the runway with and without a quadrotor failure is presented.

4.4.9 Inspecting non-rectangular areas

While this paper was motivated by the runway inspection task, this subsection considers the applicability of the technique to other non-rectangular inspection tasks. For example, agricultural inspection or search and rescue may require non-rectangular areas to be inspected. Two shapes are considered: a C-shaped area and an L-shaped area, shown in Fig. 4.3.

The non-rectangular areas use an identical logic: all tiles within the inspection area boundary yield a reward of +1. Therefore, the only modification for non-rectangular shapes is to change the structure of $S_n$ in Eq. (4.3). All other training parameters are identical.
4.4.10 Hand-crafted zig-zag approach

The proposed deep guidance runway inspection strategy is compared to a hand-crafted approach. An intuitive zig-zag strategy to inspect the runway is shown in Fig. 4.4. This hand-crafted set of waypoints readily guides the quadrotor to inspect all tiles. However, this approach is inflexible to changing initial conditions—the quadrotor will need to waste time flying to the starting point from its initial condition. In addition, it is unclear how to make this approach work for two cooperative quadrotors that have random initial conditions. Does each quadrotor become responsible for 1/2 the runway? What if the quadrotors start on the opposite side of the runway than they were assigned? Additionally, it is unclear how to proceed should one of those quadrotors fail. While more engineering effort could be applied to solve these problems, a similar amount of effort to the design of the deep guidance reward function (a reward of +1 for inspecting a new runway tile; penalties for quadrotor proximity) was used. This hand-crafted zig-zag approach is compared to the single-quadrotor deep guidance approach in the outdoor flight experiments in Sec. 4.6.5.

Figure 4.3: Non-rectangular areas to inspect.

Figure 4.4: The hand-crafted zig-zag approach.
4.5 Simulation results

To determine whether the real-time closed-loop deep guidance technique is effective at solving the multi-quadrotor runway inspection problem, the system is trained in simulation. The simulated environment is chosen to reflect the available hardware present at École Nationale de l’Aviation Civile where the trained policies will ultimately be evaluated in experiment. An indoor flight arena and an outdoor runway are available; each facility has two quadrotors available for use, detailed in Sec. 4.6. With the hardware available in mind, the following simulations are designed:

1. An indoor runway inspected by one quadrotor.

2. An indoor runway inspected by two quadrotors.

3. An outdoor runway with one quadrotor inspecting it, used to compare against the hand-crafted zig-zag guidance strategy presented in Sec. 4.4.10.

4. An outdoor runway with two quadrotors inspecting it, used to test the fault-tolerant ability of the system to complete the task when one quadrotor fails, as discussed in Sec. 4.4.8.

5. The ability to inspect non-rectangular areas with two quadrotors.

6. An outdoor runway with a quadrotor swarm (3, 5, 8, and 12 quadrotors) inspecting it. Using 3+ quadrotors is not replicable experimentally but is used to test the limits of the proposed technique.

4.5.1 Indoor simulated training results

The indoor runway is chosen to be 4 x 4 m, which is divided into a $S_W = 4$ tiles along its width and $S_L = 8$ tiles along its length, as shown in Fig. 4.1a. The indoor and outdoor runways are divided into the same number of tiles for consistency. The acceleration bounds are set at $\pm 2 \text{ m/s}^2$ and the maximum velocity is limited to $\pm 4 \text{ m/s}$. For each episode the quadrotor initial positions are uniformly randomized across the runway to force the policy to learn the behaviour from any starting configuration.
Figure 4.5: Indoor single-quadrotor runway deep guidance training progress.

Three hundred time steps (60 s) are the maximum allotted for the agent to fully inspect the runway. For the dual-quadrotor task, proximity penalty constants of $c_1 = 1$ and $c_2 = 0.43$ are used such that a penalty of 0.01 per time step is applied when the quadrotors are 2 m apart. Training occurs in the kinematics environment described in Sec. 4.4.2 and the policy is occasionally deployed to the dynamics environment for evaluation. The dynamics environment, discussed in Sec. 4.4.4, uses a mass of 0.5 kg and its controller has $K_I = \text{diag}\{0.5, 0.5\}$, chosen by trial and error, to yield adequate tracking of the guided acceleration signal. All learning plots presented have been exponentially smoothed with a factor of 0.9.

Learning curves are shown in Fig. 4.5a and solve time curves are shown in Fig. 4.5b. Training results for the single- and dual-quadrotor cases are included on the plots to allow for comparison. The learning curve shows the total rewards received by the agent when its performance is evaluated in a dynamics environment as a function of the number of training episodes completed. The performance quickly increases to its maximum of 32, indicating that runway inspection task has been successfully learned in both in the single- and dual-quadrotor indoor environments. Once the ability to fully inspect the runway was learned, further training reduced the amount of time needed to complete the task, as shown in Fig. 4.5b. The discount factor in the reinforcement learning algorithm causes future rewards to be weighted lower than current rewards, which drives the agent to collect rewards as fast as possible.
and inspect the runway quickly. The average time required to solve once training was complete was 14 seconds and 10.8 seconds for the single- and dual-quadrotor scenarios, respectively, indicating that, as expected, two quadrotors can inspect the runway faster than one quadrotor. A single-quadrotor sample trajectory over time is shown in Fig. 4.6a and a dual-quadrotor sample trajectory over time is shown in Fig. 4.6b. The runway tiles are shaded as they become inspected.

With runway inspection behaviour successfully learned for the single- and dual-quadrotor scenarios indoors, outdoor training simulations are performed next.

### 4.5.2 Outdoor single- and dual-quadrotor simulated training results

The École Nationale de l’Aviation Civile in Toulouse, France, has an access to a private radio-controlled (RC) model runway that is 124 m × 12.5 m, which is divided into grid tiles similar as shown in Fig. 4.1b. For these outdoor simulations, the acceleration and velocity bounds are increased to ±2.5 m/s² and ±5 m/s, respectively. The proximity penalty constants are changed to $c_1 = 1$ and $c_2 = 4.3$ such that...
a penalty of 0.01 per time step is applied when the quadrotors are 20 m apart to encourage independent inspection. The maximum number of time steps allotted is increased to 500 (100 s). When two quadrotors are used, a 50% chance of one quadrotor failing is included. When a quadrotor fails, its velocity is forced to 0, its position is forced to one corner of the runway, and the data that quadrotor generates is no longer included in the replay buffer for training purposes as discussed in Sec. 4.4.8. Figure 4.7 shows the learning curve and solve time plot for the outdoor runway inspection with both one and two quadrotors. The outdoor simulated performance is similar to the previous indoor performance. With random initial conditions, the two quadrotors can successfully calculate guidance signals to cooperatively inspect the runway environment without being explicitly programmed. In addition, the runway can reliably be fully inspected when one quadrotor has failed. A single-quadrotor sample trajectory is shown in Fig. 4.8a, a dual-quadrotor sample trajectory without a failure is shown in Fig. 4.8b, and a dual-quadrotor sample trajectory with a failure is shown in Fig. 4.8c.

4.5.3 Non-rectangular inspection results

Two quadrotors are used to inspect the C- and L-shaped areas discussed in Sec. 4.4.9. The learning progression is shown in Fig. 4.9. The learning curve, shown in Fig. 4.9a,
increases to its maximum. It should be noted that the maximum reward is different for the non-rectangular shapes, since the total reward available is reduced by the number of tiles omitted from $S_n$. The time needed to solve the inspection task, shown in Fig. 4.9b, decreases, indicating that the non-rectangular inspection task is successfully learned.

Sample trajectories of the two non-rectangular inspection tasks are shown in Fig. 4.10. The quadrotors successfully coordinate their behaviour to explore the non-rectangular areas quickly and safely. The following simulation is used to test the limits of the deep guidance algorithm by increasing the number of quadrotors.

### 4.5.4 Quadrotor swarm simulated training results

Although only two quadrotors are available for experiments, the proposed guidance algorithm is trained with additional quadrotors in this section to examine how well the system scales to many quadrotors. The same outdoor runway is used and tested with 3, 5, 8, and 12 quadrotors. Quadrotor failures are not simulated, and the proximity penalty coefficient $c_2$ in Eq. (4.9) is reduced to 2.15 for the 3-quadrotor scenario and
Figure 4.9: Learning to explore non-rectangular shapes.

Figure 4.10: Visualization of sample trajectories from the outdoor simulated non-rectangular area training scenarios. Quadrotors are enlarged to show their current position.
Figure 4.11: Training results with higher numbers of quadrotors. The exponential smoothing factor is increased to 0.99 for clarity.

to 1.08 for the remaining scenarios to allow for more quadrotors to inspect the runway without being as penalized for proximity. The learning plots are shown in Fig. 4.11.

The learning curves show that the runway inspection task can be successfully learned with up to 12 quadrotors despite the observation in Eq. (4.5) growing significantly in size. The observation has 50 elements when two quadrotors are used and 110 elements when 12 quadrotors are used. Additional quadrotors needed less time to fully inspect the runway than the dual-quadrotor approach. However, the learned behaviour with higher numbers of quadrotors is not as creative—all quadrotors fly a similar trajectory left and right on the runway. Since so many quadrotors are present, and randomly distributed along the runway, this simple behaviour in large numbers is sufficient to inspect the runway. These simulated results represent a scenario where the learning-based approach exploited the reward function. Technically, the left-right behaviour seen by all agents is effective at inspecting the entire runway, but it is not the coordinated behaviour that was intended. Future work should explore this quadrotor swarm domain further. The single- and dual-quadrotor indoor and outdoor simulations are experimentally validated in the following section.
4.6 Experimental validation

To explore if the proposed deep guidance technique detailed and simulated above will allow for policies trained entirely in simulation to be transferred to real robots, the quadrotor experimental facilities at École Nationale de l’Aviation Civile (ENAC) are used. The indoor and outdoor quadrotor experimental facilities are presented, followed by the experimental setup and results.

4.6.1 Indoor experiment facility

The indoor experimental facility at ENAC consists of a 10 m × 10 m × 9 m flight volume with a mesh exterior, as shown in Fig. 4.12a. A 16-camera Optitrack system is used to track the motion of the quadrotors with sub-millimetre resolution in real time—a feature that is replaced by considerably less accurate GPS when flying outdoors. The quadrotors are called Explorer 1 and Explorer 2, and are shown in Fig. 4.12b. Including the battery, their mass is 535 g and their maximum thrust is 40 N. A 3-cell battery operates at 11.1 V and contains 2,300 mAh, which provides roughly 15 minutes of flight time.

The indoor quadrotors use the *Paparazzi Autopilot System* [119], an open-sourced software package for unmanned aerial systems. *Paparazzi* consists of a ground segment, running on a personal computer, an airborne segment, running on-board the quadrotor, and an XBee communication link between them. The on-board computer is the *Tawaki*, shown in Fig. 4.12c, with its characteristics listed in Table 4.1.

4.6.2 Outdoor experimental facility

Outdoor runway experiments are tested in a local RC airfield runway located in Muret, France, that is 124 m × 12.5 m in size, as shown in Fig. 4.1b. ENAC has privileged access to this airfield, and can do tests in a volume of 500 m radius and up to 150 m height (which can be increased to 450 m in certain cases). Different quadrotors are designed and manufactured to be used outdoors. They consist of the same avionics and propulsion system used on the indoor quadrotors, but are additionally equipped with Ublox-M8 GPS receiver which supplies 5 Hz position...
Figure 4.12: ENAC indoor facility and flight hardware.
Table 4.1: General characteristics for the Tawaki autopilot board.

<table>
<thead>
<tr>
<th>Item</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU</td>
<td>STM32F7</td>
</tr>
<tr>
<td>IMU</td>
<td>ICM20600 (accel, gyro) + LIS3MDL (mag)</td>
</tr>
<tr>
<td>Baro</td>
<td>BMP3</td>
</tr>
<tr>
<td>Serial</td>
<td>3 UARTS, I2C (5V + 3.3V), SPI</td>
</tr>
<tr>
<td>Servo</td>
<td>8 PWM/DShot output (+ ESC telemetry)</td>
</tr>
<tr>
<td>RC</td>
<td>2 inputs: PPM, SBUS, Spektrum</td>
</tr>
<tr>
<td>AUX</td>
<td>8 multi purpose auxiliary pins (ADC, timers, UART, flow control, GPIO, ...)</td>
</tr>
<tr>
<td>Logger</td>
<td>SD card slot</td>
</tr>
<tr>
<td>USB</td>
<td>DFU flash, mass storage, serial over USB</td>
</tr>
<tr>
<td>Power</td>
<td>6V to 17V input (2-4S LiPo)</td>
</tr>
<tr>
<td></td>
<td>3.3V and 5V, 4A output</td>
</tr>
<tr>
<td>Weight</td>
<td>12 grams</td>
</tr>
</tbody>
</table>

information with an accuracy of approximately 1.5 m. The quadrotors that are used for the outdoor flights are shown in Fig. 4.13b alongside with the ground control station (ordinary laptop), safety-pilot transmitters, and an XBee radio-modem that is used for telemetry and down-link communication. Figure 4.14 presents, at a high level, how the signals flow between the various components.

4.6.3 Experimental setup

The indoor simulations presented in Secs. 4.5.1 and 4.5.2 were designed to be replicated in the indoor and outdoor experimental facilities at ENAC, respectively. The parameters $\theta$ from the policies trained in simulation are directly exported for use in experiment. The policy input observation in Eq. (4.5) is assembled using experimental data. Position and velocity information are obtained from the Optitrack ground truth system for indoor flights, and from GPS measurements for outdoor flights.

The quadrotors take off and manoeuvre to their randomly-assigned holding positions using the autopilot software. Then, the quadrotors are switched into deep guidance mode, where they begin listening for real-time deep guidance acceleration signals from the policy network. The on-board controller uses incremental nonlinear dynamic inversion control [101] to track that acceleration signal. Once the runway is
Figure 4.13: ENAC outdoor facility and flight hardware.

Figure 4.14: Communication between the various segments in the outdoor experiments. Dashed arrows between the quadrotor and ground computer represent wireless communication using XBee. Indoor experiments obtained $\mathbf{x}$ and $\mathbf{v}$ from a ground truth system instead of an on-board GPS module.
fully inspected, the quadrotors return to their holding positions.

There are many discrepancies between the simulated environments within which the policies were trained and the experimental facilities where their performance is evaluated. The simulated environments did not model the vehicles as quadrotors—they were modelled as double-integrator point masses. Rotor dynamics, air disturbances, actuator limitations, wind, GPS, IMU, ground truth, and barometer inaccuracies, and coupling between altitude control and translational control were unmodelled. In addition, the mass and the on-board controllers that track the guided acceleration signals (incremental nonlinear dynamic inversion controller) were different than the ones used during evaluation of the policy performance in simulation (integral controller). Dramatic discrepancies exist between the simulated and experimental environments. Therefore, this is an excellent test of the simulation-to-reality capabilities of the deep guidance technique.

### 4.6.4 Indoor experimental results

The initial conditions for each experiment were randomized. The indoor single-quadrotor runway inspection experiments completed their task in 21.1 s on average, with a standard deviation of 3.6 s over 9 trials. Nine out of the ten attempts were deemed successful, with one attempt requiring more than the maximum amount of time allotted to finish. A successful single-quadrotor sample trajectory is shown in Fig. 4.15a. The indoor dual-quadrotor runway experiments completed the task in 15.7 s on average, with a standard deviation of 6.9 s over 10 trials. All dual-quadrotor experiments were successful; a sample dual-quadrotor trajectory is shown in Fig. 4.15b. The experimental results confirmed that the average time needed to fully inspect the runway is reduced when two quadrotors are used, as expected. The experimental results also confirm that the deep guidance technique is viable for bridging the simulation-to-reality gap, and successfully shows the application of learning-based guidance techniques to real quadrotors. The indoor experimental campaign overall took roughly 50% more time to complete the runway inspection than in simulation during training, which is attributed to the significant differences between the simulated and experimental environments listed in Sec. 4.6.3. In addition, communication
issues resulting in packet loss, evident by the straight trajectory segments in Fig. 4.15, likely led to some tiles being missed, requiring additional flight time to inspect them.

4.6.5 Outdoor experimental results

The outdoor single-quadrotor experiment solved the runway inspection task in 76.6 s on average, with a standard deviation of 10.6 s over 5 trials with randomized initial conditions. Compared to the simulated environment, the experiments took 20% more time to complete the inspection. A representative trajectory is shown in Fig. 4.16a. The zig-zag manoeuvre, presented in Sec. 4.4.10, was also tested using randomized initial conditions, and is used to compare a hand-crafted approach to the learned approach. The zig-zag trajectory took 88.8 s on average, with a standard deviation of 5.7 s over 5 trials. A sample trajectory is shown in Fig. 4.16c.

The single-quadrotor outdoor experimental result shown in Fig. 4.16a has resemblance to the hand-crafted zig-zag trajectory shown in Fig. 4.16c in that it oscillates left and right as it travels along the runway. Any part of each runway tile needs to
be flown over for it to qualify as being inspected—notice how the trajectory barely
touches all tiles in the third row from the top of the runway. In comparison, the zig-
zag approach was inflexible to changes in initial conditions—it simply followed the
precribed set of waypoints. A waypoint is considered “reached” when the quadrotor
is within 2.5 m of it. Still, the quadrotor loses significant speed at each waypoint,
through making sharp turns, which causes the zig-zag approach to complete the in-
spection slower than the deep guidance approach. The single-quadrotor deep guidance
approach effectively learned a modified zig-zag pattern that is able to maintain its
speed throughout the flight, while automatically adapting its flight path to every
initial condition encountered.

The dual-quadrotor outdoor experiments were performed with 4 different initial
conditions. They fully inspected the runway in 56.5 s on average, with a standard
deviation of 11.8 s, taking 25% more time than in simulation. The increase in solve
time during the outdoor experiments, compared to the training simulations, was lower
than the indoor experiments’ 50% increase. Both the indoor and outdoor experiments
used an XBee communications system, but the outdoor flights suffered from less
packet loss than the indoor flights, leading to one less barrier between simulation
and reality for the deep guidance system to overcome, and is the likely source of the
improvement. A sample trajectory is shown in Fig. 4.16b. Following this, 4 more
trials were flown with a simulated failure of one quadrotor occurring at 20 s into the
inspection. That is, one quadrotor fails while the other one must continue to inspect
the runway. In all cases, the remaining quadrotor finished the inspection task on
its own, without any human intervention, demonstrating the flexibility of the deep
guidance approach. The average solve time with a failure rose to 70.0 s on average,
with a standard deviation of 13.4 s. A sample trajectory is shown in Fig. 4.16d.

As illustrated in Fig. 4.16b the red quadrotor opted to cover the top portion of
the runway while the black quadrotor chose to inspect the bottom portion. This is
despite the red quadrotor starting near the centre of the runway, and therefore could
have inspected in either direction. The cooperative nature of the system is demon-
strated—the quadrotors choose their behaviour based on the anticipated behaviour
of the other, the state of each quadrotor, and the state of the runway. In the case presented in Fig. 4.16d, notice how the initial conditions were swapped. Here, the black quadrotor chose to inspect the top portion of the runway while the red quadrotor inspected the bottom portion. The same trained policy was used in both experiments that generated Figs. 4.16d and 4.16b, demonstrating the wide range of behaviours that are generated from a single policy. In addition, one quadrotor suffered a failure in Fig. 4.16d. Upon sensing the failure, the still-operational red quadrotor completed the inspection of its portion of the runway before inspecting the final tile on the upper portion of the runway. This clearly demonstrates the flexibility of the dual-quadrotor deep guidance system to varying initial conditions and quadrotor failures.

The complexity of the learned behaviours has emerged from a simple reward function: +1 for inspecting a new tile and penalties for quadrotor proximity. Though it was not attempted, it would likely require significant engineering effort to hand-craft a dual-quadrotor guidance algorithm that can similarly: a) handle any initial condition efficiently; and b) gracefully adjust when one quadrotor fails. These results demonstrate the power of learning-based approaches to solve tasks specified at a high level—"inspect the runway" in this paper—and the deep guidance technique, which permits only guidance to be learned, allows for these behaviours to be realized not only in simulation but also in a representative, outdoor, GPS-driven facility, providing a possible solution to the simulation-to-reality gap.

Learning-based methods have their limitations: learning can be difficult to achieve, performance cannot be guaranteed, and significant computing power (during the training phase) is required. A video summarizing the simulations and experiments can be found at: https://youtu.be/Pu5rWnLgyZs. All code used can be found at https://github.com/Carleton-SRCL/Field_Robotics_2021.

4.7 Conclusions

This paper used deep reinforcement learning to act as closed-loop real-time guidance for quadrotors to autonomously and cooperatively learn to perform a runway inspection. A simple reward scheme was used that encouraged cooperative runway inspection guidance to be learned without any human intervention or design, significantly
Figure 4.16: Outdoor experimental results.
reducing the engineering effort required to complete the task. The multi-quadrotor runway inspection system was trained entirely in simulation and deployed to two real-world experimental facilities at École Nationale de l’Aviation Civile, where both indoor and outdoor GPS-driven experiments were performed. The indoor and outdoor experiments were successful, though they required 50% and 25% more time, respectively, to complete the inspection than during training in simulation. There were significant discrepancies between the simulated environment within which the deep guidance policy was trained and the experimental environment where it was evaluated. This highlights the simulation-to-reality capability of the deep guidance approach: using deep reinforcement learning for guidance is a potential avenue for allowing policies trained exclusively in simulation to be executed on real hardware.

The deep guidance approach was shown to complete the task faster than a handcrafted zig-zag approach, create successful trajectories regardless of its initial conditions, and, in the dual-quadrotor scenario, be tolerant to the failure of one quadrotor. The dual-quadrotor model learned to divide the runway into portions with each quadrotor assuming responsibility for the nearest area, which varied from experiment to experiment since the initial conditions were randomized. The deep guidance system was shown to be trivial to retrain for use on non-rectangular inspection areas by removing rewards from irrelevant tiles. Both C- and L-shaped runways were used to show that this approach can be used as a more general distributed inspection system—not only for rectangular runways.

While reinforcement learning is able to produce impressive multi-agent results with little engineering effort (i.e., only specifying the reward function), effort is required to tune all the relevant learning parameters, $\alpha$, $\beta$, $\gamma$, $\epsilon$, $\sigma$, $M$, and $N$, among others. An incorrectly-chosen learning parameter may cause the learning algorithm to fail. Careful attention to the reward function is also important. The learning algorithm has no sense of the desired behaviour, it is only driven to maximize the rewards received; the desired behaviour is simply a byproduct of this reward-seeking behaviour. The reward function must be designed to promote the desired behaviour while keeping in mind the algorithm may learn unintended behaviour in its pursuit of rewards (as was the case in the quadrotor swarm scenario). The quadrotor swarm scenario
learned an exploration strategy that blindly moved all agents left and right over
the runway, using their large numbers to explore the entire runway instead of truly
learning a coordinated exploration strategy. Lastly, during preliminary experiments,
the acceleration signals were not being properly tracked, which put the quadrotors
into states not often encountered during training (far outside the runway). Since
the learning algorithm had never encountered quadrotors being very far from the
runway during training, it did not learn a suitable behaviour to recover the flight.
Although learning algorithms can generalize the examples seen during training to
new unseen scenarios, those new scenarios must still be within the range of scenarios
encountered during training. It is therefore equally important for the controller used
in experiment, which may be different than the one used in simulation, to track the
guided accelerations as closely as possible.

Future work should capture images at each tile and merge them into a composite
runway image for processing, modify the observation such that it doesn’t grow with in-
creased quadrotors, investigate performance guarantees of learning-based approaches,
or develop an algorithm that can inspect an arbitrary shape without retraining.

Acknowledgments

This research was financially supported in part by the Natural Sciences and Engi-
neering Research Council of Canada under the Postgraduate Scholarship-Doctoral
PGSD3-503919-2017 and the Ontario Graduate Scholarship awards. The authors
would like to thank Xavier Paris for his support during both the indoor and outdoor
flight experiments. This research was enabled in part by support provided by Re-
search Computing Services\textsuperscript{2} at Carleton University; specific thanks to Ryan Taylor
for his help.

\textsuperscript{2}https://carleton.ca/rcs
Continuity to Next Chapter

The next, and final, paper returns to solve the problem that motivated this research—manipulator-based spacecraft capture—and applies the now-developed deep reinforcement learning-based guidance technique to the problem.
Chapter 5

Spacecraft Robotic Capture and Simultaneous Stabilization Using Deep Reinforcement Learning-based Guidance

*This chapter is under peer-review at the Journal of Guidance, Control, and Dynamics. The following paper citation is preferred, and this thesis should only be cited if the appropriate citation for the included article cannot be found:*


The paper is included in this thesis with three minor modifications: 1) Eqs. (5.18) and (5.19) are included for completeness along with their elements defined in App. A; 2) minor formatting changes; and 3) the DRL theory section was removed to avoid repetition, with the reader being referred to Sec. 2.3. This paper was co-authored by the thesis author, Kirk Hovell, and his thesis supervisor, Prof. Steve Ulrich. Mr. Hovell set-up, developed, wrote, and evaluated the relevant code; obtained and analyzed simulated and experimental results; and wrote the manuscript. Theory development and manuscript editing were conducted jointly. Details of the author’s contributions to the Spacecraft Proximity Operations Testbed, in order to realize the experiments presented in this chapter, are discussed in Appendix B.

**Paper Context**

In this final paper, the author returns to the central motivator of this research: manipulator-based capture of a spinning spacecraft (for on-orbit servicing or space
debris removal), where the DRL-based guidance technique developed over the preceding chapters is finally applied to the problem. The technique outperforms currently-available hand-crafted spacecraft capture techniques, demonstrating the benefits of learning-based techniques for aerospace robotics applications.

5.1 Abstract

This paper considers a planar three-link manipulator-equipped spacecraft capturing and simultaneously stabilizing an uncooperative spinning target. Deep reinforcement learning is used to autonomously learn a high-level closed-loop real-time capture and simultaneous stabilization guidance strategy. The learned guidance strategy is trained entirely in simulation with a sparse reward signal—the simple reward signal drives the desired behaviour to be learned instead of the designer hand-crafting the behaviour. The chaser spacecraft successfully learns to capture the spinning target despite position, velocity, and acceleration constraints, and randomized initial positions and target angular velocities, while ensuring near-zero post-capture angular velocity. The learned guidance policy is analyzed over a subset of the state space. The trained guidance policy can be run in real-time, and is transferred to a spacecraft platform in the Spacecraft Proximity Operations Testbed, an experimental facility at Carleton University. A spinning target platform is captured and stabilized successfully in experiment, despite being subjected to initial conditions and perturbations unseen in training. A video of the simulated and experimental results can be found online at https://youtu.be/\_oWpEH_dalo.

5.2 Introduction

Since the first demonstrations of on-orbit servicing using a robotic manipulator with ETS-VII in 1997 [134] and Orbital Express in 2007 [135], robotic manipulator research has accelerated in hopes of enabling autonomous on-orbit servicing and space debris removal. For a chaser spacecraft to service a target it must perform two tasks: 1) capture the target with its end-effector; and then 2) stabilize the spinning target post-capture. The first task, target capture, is typically accomplished using one of
two approaches: i) the chaser spacecraft is brought to rest near the target, after which the manipulator alone executes the capture, typically in a way that minimizes the disturbances on the unactuated spacecraft base [9–12]; or ii) the chaser spacecraft approaches the target and captures it in one motion (i.e., both the spacecraft base and manipulator are actuated). The latter approach is more flexible than the former, since it can handle time-varying keep-out zones, and has therefore been extensively studied with optimization methods [13–17], model predictive control methods [18], and sample-based methods [19]. The second task, stabilizing the spinning target post-capture, has been separately studied through minimum-time formulations [13,20] and various advanced control approaches [21–24].

Recently, Virgili-Llop and Romano [25] presented a method that performs both the capture and stabilization simultaneously. It uses an optimization-based approach to generate a real-time guidance trajectory that satisfies a number of terminal constraints such that the capture and stabilization are considered simultaneously in the optimization. This technique leads the chaser spacecraft to capture the target with some momentum of its own while ensuring the end-effector gently captures the target. The chaser’s momentum effectively cancels the target’s angular momentum upon capture, negating the spin of the combined system once the manipulator joints are brought to rest. The authors experimentally validated their work while demonstrating the approach can be executed on low-powered hardware. While this optimization-based work has shown great promise, it is accompanied by prescribed capture times and orientations, constant chaser initial conditions, and hard-coded manipulator motion during the final seconds before capture—all required for the optimization to be tractable. In addition, the approach requires significant engineering effort to design and was less effective at capturing moderately-tumbling targets greater than seven deg/s (92.7% success rate at seven deg/s; 64.8% success rate at eight deg/s) [25]. Compared to this previous capture and simultaneous stabilization result, the objective of this paper is to: a) relax a number of assumptions; while b) reducing the engineering effort associated with the approach by developing a machine learning-based guidance strategy for the problem; and c) improving performance by enabling the capture of high spin-rate targets.
Specifically, this paper uses a learning-based approach to learn a guidance strategy for simultaneous capture and stabilization, rather than designing one by hand. This reduces the engineering effort and removes a large part of human bias from the eventual guidance strategy. In addition, once trained, learning-based approaches can be very computationally-efficient and generate closed-loop guidance trajectories in real-time. Deep reinforcement learning is the chosen branch of artificial intelligence for this task.

Deep reinforcement learning uses an agent that, through trial and error, discovers the best way to solve a given problem. It does so by calculating an action to take from a given observation of the state. The agent’s notion of success comes from a human-designed scalar reward function. Through analyzing each time step of data containing: observations encountered, actions taken, next observations encountered, and rewards received over a large number of time steps, the deep reinforcement learning agent learns the best actions to take from a given observation to maximize the rewards received. Through maximizing the rewards received, the agent indirectly learns to solve the task. This is the motivation to use reinforcement learning for space robotics—highly-complex behaviour may emerge from simple reward functions [123], unlocking new capabilities for spacecraft.

Reinforcement learning has been cited as a major influence in the future of aerospace engineering [30]. To date, it has been used with spacecraft orbit determination [32], uncrewed aerial vehicle wildfire detection [37], planetary landing [38, 39], and spacecraft proximity operations [33, 40, 41] with experimental results [1]. A recurrent neural network was used to estimate a dynamics model, which was then used in model-predictive control [136]. Neural networks have been trained to mimic optimal control for pinpoint planetary landing [79]. Reinforcement learning has recently been used for a dual-arm stationary spacecraft to move its two end-effectors near a target [42]. Similar dual-arm work, with a free-floating spacecraft base and additional dynamical constraints has been presented in [43] with preliminary results (capture is considered successful when both end-effectors are within 0.2 m of the docking ports). Unless otherwise reported, all previous work has been studied in simulation only.

In this paper, deep reinforcement learning is used to guide a manipulator-equipped
chaser spacecraft to capture and simultaneously stabilize a spinning target from arbitrary initial conditions despite position, velocity, and acceleration constraints. The learned real-time closed-loop guidance technique is computationally-efficient and is evaluated in hardware experiments at Carleton University’s planar Spacecraft Proximity Operations Testbed. The novel contributions of this work are:

1. A guidance technique that is learned, rather than designed, for the planar capture and simultaneous stabilization of a spinning target spacecraft. Significant assumptions from previous work are removed, discussed in Sec. 5.4.1, along with improved performance at high target angular rates.

2. The first demonstration, to the best of the authors’ knowledge, of using artificial intelligence techniques to perform manipulator-based capture of a spinning target in an experiment.

This paper is organized as follows: Sec. 5.3 presents background on deep reinforcement learning and the specific learning algorithm used in this paper, Sec. 5.4 discusses the spacecraft capture task, presents the equations of motion, and describes how the deep reinforcement learning-based guidance technique is applied, Sec. 5.5 presents the training of the guidance algorithm and analyzes the learned behaviour, Sec. 5.6 presents and discusses experimental results, and Sec. 5.7 concludes this paper.

5.3 Markov Decision Processes and Deep Reinforcement Learning

To avoid repetition, this section has been removed; please see Sec. 2.3 for a description of deep reinforcement learning and the D4PG algorithm used in this work.

5.4 Problem Statement

This section describes the planar robotic capture and stabilization task and how the learning algorithm presented in Sec. 2.3 is structured to solve the task. Two spacecraft exist in a planar environment: a chaser spacecraft is actively controlled and has a three-link robotic manipulator; a target spacecraft is passive, spinning, and has a docking port. The goal of this work is to: 1) have the chaser spacecraft learn how
to capture the spinning target with its robotic manipulator; while 2) simultaneously bring the target to rest upon being captured. To capture the target and simultaneously bring it to rest, the chaser must capture it with equal and opposite angular momentum about the centre of mass of the post-capture system. The task to be learned is illustrated in Fig. 5.1.

5.4.1 Assumptions and Improvements

To allow for comparison, this work matches some core assumptions present in the previous work [25]: 1) a target-centred reference frame is used (such that the chaser’s velocity is reported relative to the target); 2) the full state is assumed to be known; 3) the chaser, its manipulator, and the target are rigid bodies that exist in planar space; 4) orbital effects and perturbations are neglected due to time scale and proximity (as is common in proximity operations research [25, 62, 85, 88]); 5) the target has a docking port which the chaser engages immediately when its end-effector contacts it; and 6) the chaser’s mass remains constant throughout the capture.

Compared to previous optimization-based work accomplishing the same task [25], this deep reinforcement learning-based approach allows for the following improvements to be made: 1) the chaser and manipulator initial positions are randomized; 2) the chaser and manipulator configuration at capture are not prescribed; 3) manipulator motion is not hard-coded before capture; and 4) no prescribed manoeuvre time is set (i.e., the chaser is free to capture the target at any time).
5.4.2 Deep Reinforcement Learning-based Guidance

To apply deep reinforcement learning to the capture scenario considered in this paper, the deep reinforcement learning-based guidance technique developed by the authors [1, 133] is herein used. Deep reinforcement learning is used to learn a guidance strategy (i.e., to issue a desired acceleration signal), while control (to track that desired acceleration signal) is performed using traditional approaches. The motivation for this architecture is twofold: 1) it allows the high-level problem solving abilities of deep reinforcement learning to be harnessed for robotics applications; while 2) preventing the policy from overfitting a controller to the simulated dynamics (which are not identical to reality). Any controller, specific to the robot, may then be used to track the desired signals from the learned guidance system. A block-scheme diagram of the guidance and control system is shown in Fig. 5.2, where $\dot{\mathbf{x}}_{c,n}^d$ is the desired chaser acceleration signal and $\mathbf{u}_n$ is the control effort.

The chaser is defined by the state

$$\mathbf{x}_c = \begin{bmatrix} x_c & y_c & q_1 & q_2 & q_3 \end{bmatrix}^T$$

(5.1)

where all variables are defined in Fig. 5.3. The angles $q_1$, $q_2$, and $q_3$ are defined in the manipulator joint reference frames while all other variables are defined in the inertial reference frame.

The target state, as defined in Fig. 5.3, is

$$\mathbf{x}_t = \begin{bmatrix} x_t & y_t & q_t \end{bmatrix}^T$$

(5.2)
5.4.3 Kinematics

The planar kinematics of the manipulator-equipped chaser and the target are presented in this section. The manipulator joint centre of mass positions, \( p_1, p_2, p_3 \), respectively, can be calculated in the inertial frame through

\[
\begin{align*}
\mathbf{p}_1 &= \begin{bmatrix} x_c + b_c \cos(\phi_c + q_c) + a_1 \cos(\pi/2 + q_c + q_1) \\ y_c + b_c \sin(\phi_c + q_c) + a_1 \sin(\pi/2 + q_c + q_1) \\ 0 \end{bmatrix} \\
\mathbf{p}_2 &= \mathbf{p}_1 + \begin{bmatrix} b_1 \cos(\pi/2 + q_c + q_1) + a_2 \cos(\pi/2 + q_c + q_1 + q_2) \\ b_1 \sin(\pi/2 + q_c + q_1) + a_2 \sin(\pi/2 + q_c + q_1 + q_2) \\ 0 \end{bmatrix} \\
\mathbf{p}_3 &= \mathbf{p}_2 + \begin{bmatrix} b_2 \cos(\pi/2 + q_c + q_1 + q_2) + a_3 \cos(\pi/2 + q_c + q_1 + q_2 + q_3) \\ b_2 \sin(\pi/2 + q_c + q_1 + q_2) + a_3 \sin(\pi/2 + q_c + q_1 + q_2 + q_3) \\ 0 \end{bmatrix}
\end{align*}
\]

The end-effector position, \( \mathbf{p}_e \), is
\[ p_e = p_3 + \begin{bmatrix} b_3 \cos(\pi/2 + q_c + q_1 + q_2 + q_3) \\ b_3 \sin(\pi/2 + q_c + q_1 + q_2 + q_3) \\ 0 \end{bmatrix} \] (5.6)

where all variables are defined in Fig. 5.3. The docking port position, \( d_t \), in the inertial frame is

\[ d_t = \begin{bmatrix} x_t + b_t \cos(\phi_t + q_t) \\ y_t + b_t \sin(\phi_t + q_t) \\ 0 \end{bmatrix} \] (5.7)

The Jacobian matrix is a convenient way to formulate the velocity kinematics, such that the velocities of each element are related to the generalized state. For velocity

\[ \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}^T = J_v(x_c) \dot{x}_c \] (5.8)

and for the angular velocity

\[ \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}^T = J_\omega(x_c) \dot{x}_c \] (5.9)

where \( J_v(x_c), \ J_\omega(x_c) \in \mathbb{R}^{3 \times 6} \). When the planar spacecraft manipulator scenario is considered, the Jacobian for the spacecraft body is

\[ \begin{bmatrix} J_{v_c}(x_c) \\ J_{\omega_c}(x_c) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \] (5.10)

The Jacobian for the first joint is
\[
\begin{bmatrix}
J_{v_1}(x_c) \\
J_{\omega_1}(x_c)
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & -b_c S_c - a_1 S_1 & -a_1 S_1 & 0 & 0 \\
0 & 1 & b_c C_c + a_1 C_1 & a_1 C_1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 \\
\end{bmatrix}
\] (5.11)

and the second joint

\[
\begin{bmatrix}
J_{v_2}(x_c) \\
J_{\omega_2}(x_c)
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & -b_c S_c - L_1 S_1 - a_2 S_2 & -L_1 S_1 - a_2 S_2 & -a_2 S_2 & 0 \\
0 & 1 & b_c C_c + L_1 C_1 + a_2 C_2 & L_1 C_1 + a_2 C_2 & a_2 C_2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 \\
\end{bmatrix}
\] (5.12)

Finally, the Jacobian for the third joint is

\[
\begin{bmatrix}
J_{v_3}(x_c) \\
J_{\omega_3}(x_c)
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & b_c C_c + L_1 C_1 + L_2 C_2 + a_3 C_3 & L_1 C_1 + L_2 C_2 + a_3 C_3 & 0 & 0 \\
- L_1 S_1 - L_2 S_2 - a_3 S_3 & L_1 C_1 + L_2 C_2 + a_3 C_3 & 0 & 0 & 0 & 1 \\
- L_2 S_2 - a_3 S_3 & L_2 C_2 + a_3 C_3 & 0 & 0 & 0 & 1 \\
-a_3 S_3 & a_3 C_3 & 0 & 0 & 0 & 1 \\
\end{bmatrix}^T
\] (5.13)

where

\[
\begin{align*}
S_c &= \sin(\phi_c + q_c) \\
S_1 &= \sin(\pi/2 + q_c + q_1) \\
S_2 &= \sin(\pi/2 + q_c + q_1 + q_2) \\
S_3 &= \sin(\pi/2 + q_c + q_1 + q_2 + q_3) \\
C_c &= \cos(\phi_c + q_c) \\
C_1 &= \cos(\pi/2 + q_c + q_1) \\
\end{align*}
\]
\[ C_2 = \cos(\pi/2 + q_c + q_1 + q_2) \]
\[ C_3 = \cos(\pi/2 + q_c + q_1 + q_2 + q_3) \]
\[ L_1 = a_1 + b_1 \]
\[ L_2 = a_2 + b_2 \]

### 5.4.4 Dynamics

The non-linear, planar, dynamics model used for the chaser spacecraft with its rigid three-link manipulator is given by

\[
M(x_c)\ddot{x}_c + C(x_c, \dot{x}_c)\dot{x}_c = u_c
\]  

(5.14)

where \( M(x_c) \) is the inertia matrix, \( C(x_c) \) is the Coriolis matrix, and \( u_c \) is the vector of generalized forces and torques, defined by

\[
u_c = \begin{bmatrix} u_x & u_y & u_c & u_1 & u_2 & u_3 \end{bmatrix}^T
\]  

(5.15)

where \( u_x \) and \( u_y \) are the linear control forces applied to the chaser in the inertial frame, and \( u_c, u_1, u_2, \) and \( u_3 \) are the control torques applied to the chaser and its three manipulator joints, respectively. To determine the value of each matrix element in \( M \) and \( C \), the Euler-Lagrange formulation is used [137]. The total translational kinetic energy, \( T_v \in \mathbb{R}^{6 \times 6} \), is

\[
T_v = \frac{1}{2} \dot{x}_c^T(m_c J_{vc}^T J_{vc} + m_1 J_{v1}^T J_{v1} + m_2 J_{v2}^T J_{v2} + m_3 J_{v3}^T J_{v3}) \dot{x}_c
\]  

(5.16)

where \( (x_c) \) is omitted for clarity, and \( m_c, m_1, m_2, \) and \( m_3 \) are the masses of the chaser and its three manipulator links, respectively. The total rotational kinetic energy, \( T_\omega \in \mathbb{R}^{6 \times 6} \), is

\[
T_\omega = \frac{1}{2} \dot{x}_c^T(I_c J_{\omega c}^T J_{\omega c} + I_1 J_{\omega 1}^T J_{\omega 1} + I_2 J_{\omega 2}^T J_{\omega 2} + I_3 J_{\omega 3}^T J_{\omega 3}) \dot{x}_c
\]  

(5.17)

where \( I_c, I_1, I_2, \) and \( I_3 \) are the moments of inertias of the chaser and its three manipulator links, respectively. The inertia matrix can therefore be obtained from
the sum of $T_v$ and $T_\omega$

$$M(x_c) = \begin{bmatrix}
M_{11} & M_{12} & M_{13} & M_{14} & M_{15} & M_{16} \\
M_{21} & M_{22} & M_{23} & M_{24} & M_{25} & M_{26} \\
M_{31} & M_{32} & M_{33} & M_{34} & M_{35} & M_{36} \\
M_{41} & M_{42} & M_{43} & M_{44} & M_{45} & M_{46} \\
M_{51} & M_{52} & M_{53} & M_{54} & M_{55} & M_{56} \\
M_{61} & M_{62} & M_{63} & M_{64} & M_{65} & M_{66}
\end{bmatrix} \tag{5.18}$$

where $M_{ij} \forall i,j = 1, \ldots, 6$ are listed in Appendix A. The Christoffel symbols are derived and the Coriolis matrix is obtained

$$C(x_c, \dot{x}_c) = \begin{bmatrix}
C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\
C_{21} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\
C_{31} & C_{32} & C_{33} & C_{34} & C_{35} & C_{36} \\
C_{41} & C_{42} & C_{43} & C_{44} & C_{45} & C_{46} \\
C_{51} & C_{52} & C_{53} & C_{54} & C_{55} & C_{56} \\
C_{61} & C_{62} & C_{63} & C_{64} & C_{65} & C_{66}
\end{bmatrix} \tag{5.19}$$

where $C_{ij} \forall i,j = 1, \ldots, 6$ are also defined in Appendix A.

The target spacecraft is assumed to be uncooperative. That is, it is simply a planar uncontrolled rotating platform. It has a position $(x_t, y_t)$ and an attitude $q_t$.

Orbital effects are not considered due to the short separation distance and short timescale of the capture scenario compared to the period of the orbit, rendering them negligible [62]. In addition, orbital effects are not present in the experimental facility at Carleton University where this work is experimentally validated in Sec. 5.6.

### 5.4.5 Post-capture Angular Momentum

The chaser base becomes unactuated the moment it captures the target. Therefore, the angular momentum at the moment of capture will remain constant, ignoring perturbations, in the post-capture phase. The chaser is tasked with capturing the target in such a way that the combined system is stabilized. The post-capture dynamics are not considered in simulation, but the total angular momentum at the moment
of capture can be calculated, and is used to measure the effectiveness of the capture manoeuvre at stabilizing the target.

To calculate the total angular momentum of the chaser-manipulator-target system upon capture, first the centre of mass of the chaser-manipulator system, \( \mathbf{p}_{cm} \), is found, through

\[
\mathbf{p}_{cm} = \frac{m_c \mathbf{p}_c + m_1 \mathbf{p}_1 + m_2 \mathbf{p}_2 + m_3 \mathbf{p}_3}{m_c + m_1 + m_2 + m_3} \quad (5.20)
\]

Next, the velocity of the centre of mass of each manipulator link is found using the Jacobian matrices defined in Eqs. (5.11)-(5.13)

\[
\mathbf{v}_1 = J_{v_1}(\mathbf{x}_c) \dot{\mathbf{x}}_c 
\]
\[
\mathbf{v}_2 = J_{v_2}(\mathbf{x}_c) \dot{\mathbf{x}}_c 
\]
\[
\mathbf{v}_3 = J_{v_3}(\mathbf{x}_c) \dot{\mathbf{x}}_c 
\]

The total linear momentum of the chaser-manipulator system can now be calculated

\[
\mathbf{l}_{cm} = m_c \mathbf{v}_c + m_1 \mathbf{v}_1 + m_2 \mathbf{v}_2 + m_3 \mathbf{v}_3 \quad (5.24)
\]

Similarly, the total angular momentum of the chaser-manipulator system about its centre of mass is

\[
h_c = I_c \dot{\mathbf{q}}_c + m_c(\mathbf{p}_c - \mathbf{p}_{cm}) \times \mathbf{v}_c \quad (5.25)
\]
\[
h_1 = I_1(\dot{\mathbf{q}}_c + \dot{\mathbf{q}}_1) + m_1(\mathbf{p}_1 - \mathbf{p}_{cm}) \times \mathbf{v}_1 \quad (5.26)
\]
\[
h_2 = I_2(\dot{\mathbf{q}}_c + \dot{\mathbf{q}}_1 + \dot{\mathbf{q}}_2) + m_2(\mathbf{p}_2 - \mathbf{p}_{cm}) \times \mathbf{v}_2 \quad (5.27)
\]
\[
h_3 = I_3(\dot{\mathbf{q}}_c + \dot{\mathbf{q}}_1 + \dot{\mathbf{q}}_2 + \dot{\mathbf{q}}_3) + m_3(\mathbf{p}_3 - \mathbf{p}_{cm}) \times \mathbf{v}_3 \quad (5.28)
\]
\[
h_{cm} = h_c + h_1 + h_2 + h_3 \quad (5.29)
\]

where the \((\cdot) \times\) operator represents the z-component of the cross-product matrix (the
only relevant entry for the planar scenario considered)

\[ \mathbf{p}^x = \begin{bmatrix} -p_y & p_x & 0 \end{bmatrix} \]  

(5.30)  

The target’s linear and angular momentum are obtained through  

\[ l_t = m_t v_t \]  

(5.31)  

\[ h_t = I_t \dot{\theta}_t \]  

(5.32)  

The chaser-manipulator-target combined system centre of mass is calculated using  

\[ \mathbf{p}_{cmt} = \frac{(m_c + m_1 + m_2 + m_3) \mathbf{p}_{cm} + m_t \mathbf{p}_t}{m_c + m_1 + m_2 + m_3 + m_t} \]  

(5.33)  

which can then be used to obtain the combined chaser-manipulator-target angular momentum about the combined centre of mass  

\[ h_{cmt} = h_{cm} + (\mathbf{p}_{cm} - \mathbf{p}_{cmt})^x l_{cm} + h_t + (\mathbf{p}_t - \mathbf{p}_{cmt})^x l_t \]  

(5.34)  

For the simultaneous stabilization portion of the task to be successful, the angular momentum at the moment of capture in Eq. (5.34) must be near zero.

5.4.6 Reward Function

The reward function is an abstracted and powerful way for the designer to communicate the desired behaviour to the reinforcement learning agent. It typically yields positive rewards when the task is completed and negative rewards, or no rewards, otherwise. Rewards can be shaped, meaning that a reward is given at each time step to encourage a certain behaviour, or they can be sparse, when a reward is given only at task completion. Shaped rewards typically make learning easier, at the cost of unintentionally biasing the learned behaviour towards the designer’s preferred solution. Sparse rewards, however, allow the agent more freedom in discovering a solution—one that is often better than a human could design. In the context of the capture and stabilization task, a sparse reward scheme is used as follows: a reward of 0 is given...
at all time steps, except:

- When the manipulator successfully captures the target, which occurs when the end-effector is within 4 cm of the centre of the target’s docking port, a reward of 100 is given. At this time step only, three penalties are also given to encourage proper capture:

  1. A penalty proportional to the end-effector angle, to encourage capture from the proper direction. A sine function is used to deliver zero penalty when zero angle between the end-effector and docking port is achieved, and maximum penalty when the angle is $\pm \pi/2$.

  2. A penalty proportional to the linear and angular relative velocity of the end-effector with respect to the docking port, to encourage low impact forces upon capture.

  3. A penalty proportional to the post-capture combined angular momentum, calculated in Eq. (5.34), to encourage simultaneous stabilization upon capture.

- A one-time reward of 25 for bringing the end-effector within 10 cm of the docking port.

- A penalty of 5 for each time step where one of the manipulator joints is at its limit, to discourage keeping the arm in a locked position.

- A penalty of 100 is applied, and the episode is terminated early, if the chaser exceeds position boundaries (driven by experimental constraints, discussed in Sec. 5.6.1).
More concretely, the reward function is

\[
    r_n = \begin{cases}
        -5, & \text{if } |q_1, q_2, q_3| \geq \pi/2 \\
        -100, & \text{if } x_c \notin [0, 3.5] \lor y_c \notin [0, 2.4] \\
        25 \text{ (awarded once)}, & \text{if } \|d_t - p_e\| \leq 0.1 \text{ m} \\
        100 - 50|\sin(q_e + q_1 + q_2 + q_3 - q_t - \pi/2)| - 50\|\dot{p}_e - \dot{d}_e\| - 50|\dot{q}_e + \dot{q}_1 + \dot{q}_2 + \dot{q}_3 - \dot{q}_t| - 25|h_{cmt}|, & \text{if } \|d_t - p_e\| \leq 0.04 \text{ m} \\
        0, & \text{otherwise} \\
    \end{cases}
\]

(5.35)

The relative weighting of these parameters is proportional to their severity and are selected using trial-and-error until the desired behaviour emerges.

In addition, the episode is terminated early, with a reward of zero, if: 1) the chaser collides with the target in any fashion; or 2) the maximum time limit is reached. In seeking to maximize rewards, the deep reinforcement learning-based guidance system indirectly learns to master the target capture and simultaneous stabilization task.

### 5.4.7 Learning Algorithm Implementation Details

The total system state \( x \) passed to the learning algorithm must fully describe the system in order for the Markovian assumption to be satisfied. The chaser state, manipulator state, their derivatives, the relative target position, and the target angular rate fully defines the system

\[
    x = \begin{bmatrix}
        x_c & \dot{x}_c & \dot{q}_t & x_t - x_c & y_t - y_c & q_t - q_c
    \end{bmatrix}^T
\]

(5.36)

Passing the relative position of the target, along with its absolute angular velocity, was found to help the learning process.

Both the policy and the critic neural networks use 400 and 300 neurons in their first and second hidden layers, respectively—Fig. 2.1 is not to scale. Due to empirical studies showing better performance, the action input skips the first layer of the critic [83,90]. The rectified linear unit activation function is used for all neurons except: 1) the output layer of the policy uses a tanh function, scaled to the action range, to
ensure bounded actions; and 2) the output layer of the value network uses a softmax function to ensure its output is indeed a valid probability distribution. Drawing from the original value distribution paper, $B = 51 \ [84]$, evenly spaced across the range of expected rewards: $[-100, 125]$. Learning rates of $\alpha = \beta = 0.0001$ are used alongside the Adam [91] stochastic gradient-descent algorithm. All neural network inputs are normalized to avoid the vanishing-gradients problem [118]. An N-step return of $N = 5$ is used, along with an exponential smoothing factor of $\epsilon = 0.001$ for the target networks. The discount factor was chosen to be $\gamma = 0.99$. Ten actors ($K = 10$) run episodes simultaneously to sample data from the simulated environment, with a time step of 0.2 seconds, of which the most recent $R = 10^6$ time steps are stored in the replay buffer. Batches of $M = 256$ samples are randomly drawn for each training iteration. To force exploration, Gaussian noise with a standard deviation equal to one third the action range, i.e. $\sigma = (\text{max}(a) - \text{min}(a))/3$, is applied to the calculated actions in Eq. (2.9). The Tensorflow\textsuperscript{1} machine learning framework is used to generate, train, and use the neural networks. All hyperparameters presented in this paragraph were hand-selected in the pursuit of faster learning.

After a designated agent performs five training episodes, the most up-to-date policy is run without any exploration noise applied, $\sigma = 0$ in Eq. (2.9), which allows for the training performance to be readily evaluated. The following section trains and evaluates the deep reinforcement learning-based guidance algorithm on the capture and stabilization task.

### 5.5 Training in Simulation

The capture task is trained entirely in simulation before being transferred to the spacecraft experimental laboratory at Carleton University; the simulations are herein designed to be replicated experimentally. Dynamic parameters listed in Table 5.1 are representative of the experimental facility and are used in the numerical simulations. The position, velocity, and acceleration constraints are listed in Table 5.2. The chaser, manipulator, and target initial conditions are uniformly randomized at the beginning of each episode according to Table 5.3. All objects have zero initial velocity other

\textsuperscript{1}Software available at www.tensorflow.org.
Table 5.1: Dynamics parameters

<table>
<thead>
<tr>
<th>Body</th>
<th>Body Identifier, i</th>
<th>$\phi_i$ (deg)</th>
<th>$a_i$ (m)</th>
<th>$b_i$ (m)</th>
<th>$m_i$ (kg)</th>
<th>$I_i$ (kg·m$^2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chaser</td>
<td>c</td>
<td>74.31</td>
<td>—</td>
<td>0.2515</td>
<td>11.211</td>
<td>$2.022 \times 10^{-1}$</td>
</tr>
<tr>
<td>Link #1</td>
<td>1</td>
<td>—</td>
<td>0.1968</td>
<td>0.1077</td>
<td>0.3450</td>
<td>$3.704 \times 10^{-3}$</td>
</tr>
<tr>
<td>Link #2</td>
<td>2</td>
<td>—</td>
<td>0.1982</td>
<td>0.1063</td>
<td>0.3350</td>
<td>$3.506 \times 10^{-3}$</td>
</tr>
<tr>
<td>Link #3</td>
<td>3</td>
<td>—</td>
<td>0.0621</td>
<td>0.0252</td>
<td>0.1110</td>
<td>$1.060 \times 10^{-4}$</td>
</tr>
<tr>
<td>Target</td>
<td>t</td>
<td>72.00</td>
<td>—</td>
<td>0.2406</td>
<td>12.039</td>
<td>$2.257 \times 10^{-1}$</td>
</tr>
</tbody>
</table>

Table 5.2: Position, velocity, and acceleration limits

<table>
<thead>
<tr>
<th></th>
<th>$x_c$</th>
<th>$y_c$</th>
<th>$q_c$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position, m or rad</td>
<td>[0, 3.5]</td>
<td>[0, 2.4]</td>
<td>$\infty$</td>
<td>$\pm \pi/2$</td>
<td>$\pm \pi/2$</td>
<td>$\pm \pi/2$</td>
</tr>
<tr>
<td>Velocity, m/s or rad/s</td>
<td>$\pm 0.1$</td>
<td>$\pm 0.1$</td>
<td>$\pm \pi/12$</td>
<td>$\pm \pi/6$</td>
<td>$\pm \pi/6$</td>
<td>$\pm \pi/6$</td>
</tr>
<tr>
<td>Acceleration, m/s$^2$ or rad/s$^2$</td>
<td>$\pm 0.02$</td>
<td>$\pm 0.02$</td>
<td>$\pm 0.05$</td>
<td>$\pm 0.1$</td>
<td>$\pm 0.1$</td>
<td>$\pm 0.1$</td>
</tr>
</tbody>
</table>

than the target’s uniformly randomized initial angular velocity across the range $\dot{q}_t = \pm 10$ deg/s.

5.5.1 Control

The chaser spacecraft and its manipulator are actively controlled. The controller is tasked with tracking the desired acceleration signal calculated by the policy, $\ddot{x}_c^d$. In simulation, a feedforward open-loop controller is used

$$u_c = M(x_c^d)\ddot{x}_c^d + C(x_c^d, \dot{x}_c^d)\dot{x}_c^d \quad (5.37)$$

Table 5.3: Initial conditions and their uniform randomization.

<table>
<thead>
<tr>
<th></th>
<th>Initial Position</th>
<th>Randomization</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_c$ (m)</td>
<td>1.2</td>
<td>$\pm 0.05$</td>
</tr>
<tr>
<td>$y_c$ (m)</td>
<td>1.2</td>
<td>$\pm 0.05$</td>
</tr>
<tr>
<td>$q_c$ (rad)</td>
<td>0</td>
<td>$\pm \pi$</td>
</tr>
<tr>
<td>$q_1$ (rad)</td>
<td>0</td>
<td>$\pm \pi/2$</td>
</tr>
<tr>
<td>$q_2$ (rad)</td>
<td>0</td>
<td>$\pm \pi/2$</td>
</tr>
<tr>
<td>$q_3$ (rad)</td>
<td>0</td>
<td>$\pm \pi/2$</td>
</tr>
<tr>
<td>$x_t$ (m)</td>
<td>2.3</td>
<td>$\pm 0.05$</td>
</tr>
<tr>
<td>$y_t$ (m)</td>
<td>1.2</td>
<td>$\pm 0.05$</td>
</tr>
<tr>
<td>$q_t$ (rad)</td>
<td>0</td>
<td>$\pm \pi$</td>
</tr>
</tbody>
</table>
Ensuring that deep reinforcement learning only learns guidance allows for any controller to be used locally—the same controller does not need to be used in both simulated training and once deployed to an experiment.

5.5.2 Training Results

The training is performed on Carleton University’s Research Computing and Development Cloud (32-Intel Xenon x86 CPUs) over a period of 15 days. Plots describing the learning progression are shown in Fig. 5.4. The learning curve in Fig. 5.4a shows

\[ \text{(a) Learning curve} \]

\[ \text{(b) Episode time} \]

\[ \text{(c) Post-capture angular velocity} \]

\[ \text{Figure 5.4: Training progression in simulation.} \]
the total rewards accumulated over each episode as a function of the number of training episodes. The performance is drawn from the occasional test episodes, where an agent attempts the capture task without any exploration noise introduced to its actions (i.e., $\sigma = 0$). The learning curve, which is exponentially-smoothed for clarity with a factor of 0.95, increases as training progresses. The policy is initialized with random parameters $\theta$, leading to very poor initial performance; penalties for exceeding the table boundaries are often encountered, along with manipulator joint limit penalties. The corresponding plot showing the average time per episode in Fig. 5.4b shows that $\sim 10$ seconds were used per episode initially. However, better performance, as judged by the rewards received, is quickly learned within the first $\sim 10^5$ episodes. During the rapid-learning phase, the time required to complete the episode increases dramatically—this corresponds to slow, yet successful, capture. As training continues, the maximum average reward received after $4.5 \times 10^5$ episodes is 90, which is near the theoretical maximum rewards of 125 obtainable for this task. In addition, the corresponding episode time decreases as the capture behaviour becomes mastered. Starting from purely random behaviour, the deep reinforcement learning-based guidance algorithm has successfully learned to manoeuvre the chaser, through calculating desired acceleration signals, to capture the target spacecraft. In addition, it learned to do so despite position, velocity, and acceleration constraints, as well as with randomized initial conditions and target angular velocities.

The exponentially-smoothed post-capture angular velocity of the combined chaser-manipulator-target system, along with one standard deviation, is shown in Fig. 5.4c. The post-capture angular velocity is only reported when successful capture occurs, and is why the curve does not start from episode 0. As learning progresses, the average post-capture angular velocity of the combined system approaches zero and its variance decreases (final values are $-0.17^\circ/s$ average residual angular velocity with a standard deviation of $2.56^\circ/s$). This demonstrates that as learning progresses, a strategy for stabilizing the target upon capture is learned (analyzed further in Sec. 5.5.4). The complex stabilization behaviour is encouraged by the trivial component of the reward function that penalizes post-capture angular momentum. The following subsection analyzes a simulation episode in detail.
5.5.3 Simulation Example

Snapshots of a simulated capture episode, once training is complete, are shown in Fig. 5.5. The target has initial conditions of $x_t = 2.33$ m, $y_t = 1.15$ m, $q_t = 171.1^\circ$, and $\dot{q}_t = 6.8^\circ$/s. The chaser has initial conditions of: $x_c = 1.13$ m, $y_c = 1.19$ m, $q_c = 306.8^\circ$, $q_1 = 39.7^\circ$, $q_2 = -32.0^\circ$, $q_3 = 14.2^\circ$, and all their derivatives are zero. After capture, the combined system’s angular rate is $0.24^\circ$/s. The chaser, starting from a random initial condition, manoeuvres and captures the target by placing its end-effector on the docking port. The counter-clockwise rotation of the target led the chaser to approach the docking port in a clockwise fashion, such that the post-capture system has little residual angular motion. Figure 5.6a shows the accelerations calculated by the policy over the duration of the episode and Fig. 5.6b shows the corresponding angular velocity.

5.5.4 Policy Analysis

To further analyze the learned behaviour, the guidance policy’s performance on the final 10,000 training evaluation episodes is logged and shown in Fig. 5.7a. The capture success rate (a ‘success’ being measured as greater than 25 rewards received—only possible if capture occurred) as a function of the initial target angular rate (binned
Figure 5.6: Acceleration signals and angular rates corresponding to the motion shown in Fig. 5.5.

into $2^\circ/s$ increments) is reported. The trained policy has a near-consistent success rate independent of the target’s angular rate, in contrast to previous work where the success rate declined with increased target angular rate [25]. Although this work reports lower success rates at low target angular velocities than previous work, its performance at higher angular velocities has improved and a number of assumptions have been lifted (such that the guidance policy calculates desired acceleration signals for the entirety of the task without intervention or hard-coded motion).

To understand the capture behaviour learned by the policy, driven by the reward function, the guided desired acceleration signals calculated by the policy are sampled at all points and plotted as a guidance field. In other words, for a given target position and angular velocity, the chaser position is swept and the desired linear accelerations at each chaser position are plotted. Since the policy input, in Eq. (5.36), also includes chaser attitude, velocity, arm angles, and arm angular rates, some assumptions are made. First, it assumes that the chaser velocity and arm angular rates are zero at each chaser position considered. The chaser attitude and arm angles are initialized
Figure 5.7: Understanding the learned policy. Red lines show example trajectories generated from following the guidance field.
at zero and allowed to change based off the guided acceleration signal; for clarity the final chaser and arm attitude are not plotted.

Figure 5.7b shows the linear guidance field of the policy when encountering a target with an angular velocity of $\dot{\theta}_t = 5^\circ/s$. The arrows correspond to the desired linear accelerations in $\dot{x}_c^d$ and $\dot{y}_c^d$ for the chaser to track, effectively showing the desired direction of travel for the chaser. From this plot, the general behaviour chaser linear motion can be examined. From nearly all starting positions, the chaser eventually translates to a position above the target where it can await capture; this is the behaviour realized in the simulation example in Fig. 5.5. In addition, the chaser typically moves to this location in a clockwise fashion around the target—an effective approach trajectory to neutralize the counter-clockwise spin of the target upon capture. An example trajectory following the guidance field is shown in red.

Figure 5.7c shows a guidance field when the target has an angular velocity of $\dot{\theta}_t = -5^\circ/s$. From nearly all chaser starting positions, the arrows guide the chaser to a position below the target and near the docking port. In addition, the chaser moves counter-clockwise around the target, neutralizing the clockwise angular rate of the target upon capture.

The training phase of this work is complete: the chaser successfully completes the capture task in simulation, independent of target angular rate, and two main capture behaviours have emerged according to the target spin direction. The trained deep reinforcement learning-based guidance system is exported to an experimental facility where its real-world performance is evaluated.

5.6 Experimental Validation

The proposed deep reinforcement learning-based guidance spacecraft capture technique, simulated, trained, and evaluated in the previous section, is exported for use in experiments at Carleton University’s Spacecraft Proximity Operations Testbed. This section first discusses the experimental facility and the experimental setup, and then presents and analyzes results.
Experiment Facility

Experiments are conducted at Carleton University’s Spacecraft Robotics and Control Laboratory, using the Spacecraft Proximity Operations Testbed (SPOT). Specifically, SPOT consists of two air-bearing spacecraft platforms operating in close proximity on a 2.4 m × 3.5 m granite surface. The use of air bearings on the platforms reduces the friction to a negligible level. Due to surface slope angles of 0.0026° and 0.0031° along both directions, residual gravitational accelerations of 0.439 and 0.525 mm/s² perturb the dynamics of the floating platforms along the X and Y directions, respectively.

Both platforms have dimensions of 0.3 × 0.3 × 0.3 m and are actuated by expelling compressed air at 550 kPa (80 psi) through eight miniature air nozzles distributed around each platform, thereby providing full planar control authority. Each thruster generates approximately 0.22 N of thrust and is controlled at a frequency of 10 Hz by a pulse-width modulation scheme using solenoid valves. Pressurized air for the thrusters and the air bearing flotation system is stored on-board in a refillable air cylinder at 31 MPa (4,500 psi). The structure consists of an aluminium frame with four corner rods on which three modular decks are stacked. To protect the internal components, the structure is covered with semi-transparent acrylic panels. Figure 5.8a shows the SPOT laboratory facility and Fig. 5.8b shows two SPOT platforms, one equipped with a robotic manipulator (the chaser) and the other with a magnetic docking port (the target).
The motion of both platforms is measured in real-time through four active light-emitting-diodes (LEDs) on each platform which are tracked by a 10-camera PhaseSpace motion capture system capable of recording up to 960 Hz. The on-board Raspberry Pi-3 computers run the Raspbian Linux operating system. The Raspberry Pi-3s wirelessly communicate with the PhaseSpace server to obtain their resolved states in the inertial reference frame. The chaser then passes the total state to the trained guidance policy, which returns a desired acceleration signal. The chaser then uses a controller to calculate the required thrust and actuates the appropriate solenoid valves to realize the desired accelerations. In experiment, a feedforward controller, of the same form as Eq. (5.37) is used, albeit with $M_{31}, M_{32} = 0$ to account for joint friction. The commanded forces are then passed to a control allocator which calculates duty cycles for each of the eight thrusters. The control allocation strategy accounts for the moving centre of mass of the chaser due to manipulator motion.

The ground station computer receives data from the on-board computers for post-experiment analysis purposes. A three-link robotic manipulator is attached to the chaser platform as shown in Fig. 5.8b. It consists of three Dynamixel MX-64 actuators, each with a maximum torque of 6.0 N/m of torque at 12 V. The robotic manipulator has 0.088°-resolution encoders within each joint to measure the speeds and orientations. The experimental feedforward controller does not command torques to the manipulator motors—the motors have built-in proportional-integral (PI) speed controllers. Therefore, the desired angular acceleration of each joint is integrated and the resulting angular velocity is passed to each joint motor to be tracked. The target platform has an electromagnetic docking port, shown in Fig. 5.8b, that is capable of rigidly engaging the end-effector when it comes within a short distance.

A MATLAB/Simulink numerical simulator that recreates the dynamics and emulates the different on-board sensors and actuators is first used to design and test an on-board acceleration controller for the upcoming experiment. The Simulink diagram contains driver blocks for interfacing with the real platform hardware. Once the controller performance in simulation is satisfactory, the software is converted into C++ using the Embedded Coder toolbox, compiled, and then executed on the platforms’ Raspberry Pi-3 computers. The Raspberry Pi-3 computer interfaces, over ethernet,
with an Nvidia Jetson TX2 that runs the pre-trained deep reinforcement learning-based guidance policy neural network. The Jetson TX2 accepts the chaser, target, and manipulator state information from the Pi-3, calculates the desired acceleration to complete the capture task using the deep reinforcement learning-based guidance policy, and returns the acceleration signal to the Pi-3 which uses the on-board controller to command the actuators in order to realize this desired acceleration signal.

The platforms are capable of performing a variety of different experiments, such as tethered capture of spinning space debris [138] and optimal trajectory planning for manipulator-equipped spacecraft [12].

5.6.2 Setup

The spacecraft capture task presented in Sec. 5.4 was chosen such that it is replicable experimentally. The final version of the trained-in-simulation guidance policy is exported for use in the SPOT facility. No additional training or fine-tuning is performed during the experiment. The critic is not exported for the experiment because it serves only to train the policy.

The chaser and target platforms are initially in contact with the granite table until a strong lock on the LEDs has been acquired by the motion capture system. Next, both platforms float and translate to their desired initial conditions and come to rest. Then, the chaser enables the trained guidance policy and uses its on-board controller to track the guided acceleration signal it receives from the policy. The policy guides the chaser to capture with and simultaneously stabilize the spinning target.

A significant number of discrepancies exist between the experimental facility and the simulated environment within which the policy was trained. Namely, the simulated environment dynamics did not model friction (between the air bearings and the table, nor at the manipulator joints), air resistance, signal noise, system delays, centre of mass offsets, discrete thrusters and the thrust actuation strategy, thruster plume interaction, or the table slope.
Figure 5.9: Snapshots of three experimental trajectories.
Figure 5.10: Angular rate over time for three sets of initial conditions.
5.6.3 Experimental Results

Three initial conditions are presented. Initial condition 1 has a nominal chaser starting position \((x_c = 1.2 \text{ m}; y_c = 1.2 \text{ m}; q_c, q_1, q_2, q_3 = 0^\circ)\) with target initial conditions of \((x_t = 2.3 \text{ m}; y_t = 1.2 \text{ m}; q_t = 0^\circ)\). All bodies start from rest except the target’s initial angular rate of \(\dot{q}_t = 5^\circ/\text{s}\). The experiment is repeated five times; eight snapshots in time of one trial are shown in Fig. 5.9a. The angular velocities over time, averaged over all five trials, are shown in Fig. 5.10a. After capture, the manipulator slows to a stop while the magnetic docking port fully engages. After the capture is complete, the residual angular velocity of the combined chaser-manipulator-target system is \(-1.5^\circ/\text{s}\) on average.

Initial condition 2 has a chaser starting coordinate \(y_c\) not seen during training \((x_c = 1.2 \text{ m}; y_c = 0.6 \text{ m}; q_c = 180^\circ; q_1 = 60^\circ; q_2 = -45^\circ; q_3 = 80^\circ)\). The target initial condition is identical except its angular rate has been increased to \(\dot{q}_t = 10^\circ/\text{s}\), the highest level encountered in training. Snapshots of the motion of a single trial are shown in Fig. 5.9b, and the angular velocities averaged across all trials are shown in Fig. 5.10b. After capture, the average residual angular velocity of the system is \(0.5^\circ/\text{s}\).

Initial condition 3 has chaser starting coordinates \(x_c\) and \(y_c\) not encountered during training \((x_c = 0.5 \text{ m}; y_c = 1.6 \text{ m}; q_c = 45^\circ; q_1 = 90^\circ; q_2 = 90^\circ; q_3 = 0^\circ)\), along with its arm in the stowed configuration. The target has an initial angular rate of \(\dot{q}_t = -5^\circ/\text{s}\). Snapshots of the motion are found in Fig. 5.9c and angular rates are found in Fig. 5.10c. The average post-capture angular rate of the system is \(1.0^\circ/\text{s}\). The time between capture and stabilization, i.e., the time after the ‘Mean capture time’ and before the steady-state angular rate in Fig. 5.10, varies from experiment to experiment, due to the docking port electromagnet taking time to fully engage the end-effector depending on its contact angle.

These experimental results largely follow the behaviour expected when analyzing the policy in Sec. 5.5.4: for positive target angular rates (initial conditions 1 and 2) the chaser typically captures the target at the top of the table, often waiting for the target docking port to come into view; for negative target angular rates (initial condition 3), the chaser typically moves downward through the middle of the table.
to capture the target with motion that counteracts the target’s angular rate. Both
behaviours were indeed observed here.

Initial conditions had to be carefully selected for these experiments due to ex-
perimental limitations that were not accounted for in simulation. Namely, control
allocation problems and thruster plume problems. The control allocation strategy is
a function that turns desired forces and torque into thruster firings for the eight mini-
ture air thrusters on the SPOT platforms—these thruster firings must be accurate in
order for the guided accelerations to be properly tracked. It was discovered that the
thruster air pressure decreases as a function of the number of thrusters firing simulta-
neously, leading to control allocation problems. As a result, in certain situations the
control allocator would not properly track the guided acceleration signal calculated
by the policy. Initial conditions that led to these situations were excluded, as this
experiment-related control problem was deemed outside the scope of this work. Also,
the thruster plumes from the chaser often affected the motion of the target, leading to
the chaser dramatically affecting the motion of the target immediately before capture.
This often caused the chaser to not properly capture the target, so initial conditions
were chosen by trial-and-error that minimized the thruster plume interactions. How-
ever, plume interactions were still present in the successful experiment, as observed
in Fig. 5.10c around 25 s. Here, the target angular rate slows over 5 seconds just
before capture, due to the chaser’s thrusters interacting with the target’s structure.
The target is actively controlled to maintain its desired angular rate, so it returns
to the desired angular rate once the chaser no longer overpowers it. Thruster plume
interactions must be a consideration with any future spacecraft proximity operations
experiments.

These three sets of experimental results show the successful capture and simul-
taneous stabilization of a spinning uncooperative target, using a real-time deep re-
inforcement learning-based guidance strategy that was learned entirely in simulation
according to a simple reward function. In addition, capture and stabilization was
still successful despite initial conditions that were not seen during training. This
represents the first, to the best of the authors’ knowledge, experiment demonstrating
manipulator-based capture of an uncooperative spinning spacecraft using artificial
intelligence-based techniques.

5.6.4 Perturbations

This section investigates the ability of the chaser to capture the target despite external perturbations. Two additional experiments are performed, where: 1) the chaser is pushed off course; and 2) the target is stopped from spinning mid-way through the experiment. Both perturbations are applied by manually interfering with the SPOT platforms mid-experiment. Experimental results of these two perturbations are shown in Fig. 5.11.

In Fig. 5.11a, the chaser is abruptly moved to a new location and orientation mid-way through the experiment. Since the guidance policy is calculating desired acceleration signals in real-time based off the current state of the chaser, manipulator, and target, it immediately adapts to the perturbation and captures the target as intended. Similarly, in Fig. 5.11b the target is perturbed to a new attitude and its angular rate is brought to zero mid-experiment. The guidance policy, again, immediately adapts in real-time to the new scenario and captures the target regardless. These two perturbation experiments highlight the real-time nature of the deep reinforcement learning-based guidance technique and its ability to handle perturbations not seen during training.

5.6.5 Analysis

Using learning-based techniques enabled this work to learn a capture and simultaneous stabilization guidance strategy that had less assumptions and no prescribed motion compared to previous hand-crafted work. Through the simulated results presented in Sec. 5.5, it was shown that this technique can successfully reduce the post-capture angular momentum of the chaser-manipulator-target combination from a range of initial conditions. The experimental results demonstrate that this technique can be successfully transferred to an experimental spacecraft platform for real-time use. The development of the technique was straightforward: once the learning algorithm was properly implemented and a representative simulated environment was
Figure 5.11: Snapshots of two perturbation experiments.
developed, only a reward function had to be specified, from which the complex behaviour emerged. A number of lessons were learned from the reward function specification. It is important to carefully craft the reward function to only reward the desired behaviour, and to carefully balance when it is best to assign penalties for unwanted behaviours (manipulator joints reaching their limits) or when to simply terminate the episode for other behaviours (colliding with the target).

While deep reinforcement learning enabled capture not possible with previous techniques, these results are not guaranteed in the traditional sense—neither capture success nor stability is ensured. In addition, if states well outside those seen during training are encountered, the resulting behaviour may not be as intended. Neural network safety is an active research area [139], though stability guarantees for learning-based robotics applications will be difficult; a debate within the guidance and control community should emerge: do the new capabilities unlocked by learning-based approaches outweigh the risks associated with the lack of performance guarantees?

A video summarizing the simulations and experiments can be found online at: https://youtu.be/_oWpEH_dalo. All code used can be found at https://github.com/Carleton-SRCL/JGCD_2021.

5.7 Conclusion

Using a robotic manipulator to capture and stabilize an uncooperative target is a promising solution to the growing space debris problem and to enable satellite servicing. The motion for a manipulator-equipped spacecraft to successfully capture and simultaneously stabilize a spinning target is complex, leading traditional approaches to require significant engineering effort to solve. Deep reinforcement learning, on the other hand, allows for complex behaviours to be learned, rather than designed, according to a simple reward function. In an effort to reduce the engineering burden of the manipulator-based spacecraft capture scenario, improve performance, and reduce assumptions from previous work, deep reinforcement learning was applied to the problem. The guidance system was trained in simulation and its learned behaviour was examined through a simulation example and an in-depth analysis of the policy. It was found that the policy learned to capture the target despite position, velocity,
and acceleration constraints, randomized initial conditions, and across target spin rates from $-10^\circ/s$ to $10^\circ/s$, at a uniform $80\%$ success rate. In addition, the chaser learned to capture the target such that the combined system had $-0.17^\circ/s$ average residual angular velocity (with a standard deviation of $2.56^\circ/s$). The trained guidance policy was then transferred to the Spacecraft Proximity Operations Testbed facility at Carleton University where the learned behaviour was successfully executed on a real spacecraft platform that was able to capture and simultaneously stabilize an uncooperative spinning target despite being subjected to initial conditions and perturbations not seen during training. This represents the first, to the best of the authors’ knowledge, use of artificial intelligence techniques to enable the capture of a spinning spacecraft in experiment. Future work should investigate performance guarantees of learning-based approaches.

**Acknowledgements**

This research was financially supported in part by the Natural Sciences and Engineering Research Council of Canada under the Postgraduate Scholarship-Doctoral PGSD3-503919-2017 and the Ontario Graduate Scholarship awards. This research was enabled in part by support provided by Research Computing Services\(^2\) at Carleton University; specific thanks to Ryan Taylor for his help. Computations were also made on the supercomputer Béluga from Université du Québec, managed by Calcul Québec and Compute Canada. The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), Ministère de l’Économie et de l’Innovation du Québec (MEI) and le Fonds de recherche du Québec - Nature et technologies (FRQ-NT).

\(^2\)https://carleton.ca/rcs
Chapter 6

Conclusion

In this chapter, the objective of this thesis is recalled and the contributions of the four papers, submitted as the chapters to this integrated thesis, are summarized. A list of publications that has resulted from this work is presented, followed by suggested paths for future work.

6.1 Thesis Summary

Active debris removal is essential to prevent the runaway growth of the space debris population. On-orbit servicing promises to transform space infrastructure and the space economy. Both of these ideas demand the technical ability of a chaser spacecraft to approach and autonomously dock with an uncooperative spinning target. A robotic manipulator is among the most viable methods for performing both of these tasks. This thesis improved upon the state-of-the-art in manipulator-based spacecraft capture through applying deep reinforcement learning to the guidance of such spacecraft. Deep reinforcement learning has had recent success in achieving super-human performance in a number of decision-making domains. This thesis applied deep reinforcement learning to aerospace robotics, while being sensitive to aerospace-specific requirements, and enabled state-of-the-art autonomous uncooperative spacecraft capture.

In Chapter 2, this work first determined aerospace-specific requirements that must be met in order to successfully use deep reinforcement learning. The dominant requirement was: no algorithm training can be performed on the robot. All training must be done in simulation, and the learned policy must be subsequently transferred to the robot. However, the simulation-to-reality gap was a barrier from simply transferring learned policies to a hardware experiment. To combat the simulation-to-reality
gap, the deep reinforcement learning algorithm was restricted to be used as guidance-only. Guidance (path planning) is the decision-making aspect a robot’s guidance, navigation, and control suite, and was therefore the most desirable component to be replaced with deep reinforcement learning (since DRL excels at decision-making tasks). A guidance-only deep reinforcement learning algorithm was developed that issued velocity commands to a controller in order to learn a spacecraft proximity operations task. A chaser spacecraft was tasked with safely approaching a spinning target while avoiding an obstacle, which it learned successfully in simulation. The DRL-based guidance model was then exported to the Spacecraft Robotics and Control Laboratory at Carleton University where successful proximity operations behaviour was also obtained in experiment.

Next, Chapter 3 compared two versions of the deep reinforcement learning-based guidance technique: 1) the velocity-based one developed in Chap. 2; and 2) a new acceleration-based one. The acceleration-based guidance technique issued desired acceleration signals to the on-board controller to track. This acceleration-based approach was compared to the velocity-based approach through quadrotor proximity operations simulations. Results showed that the acceleration-based guidance system is better suited for dynamical systems. The trained model was then exported for use on real quadrotors in an indoor experimental facility at École Nationale de l’Aviation Civile in Toulouse, France, where the same conclusion was made: acceleration-based guidance signals should be used going forward.

In Chapter 4, the acceleration-based DRL guidance algorithm was used to solve a more difficult and representative task (the tasks considered in Chaps. 2 and 3 were proof-of-concept tasks whose solutions could easily be hand-crafted). Airport runways need to be inspected multiple times per day, which is usually performed by an operator driving a vehicle down the runway. The task considered was to perform this runway exploration autonomously using quadrotors. This necessitated cooperative exploration between the quadrotors. In addition, fault-tolerance was important: in the event that one quadrotor fails, the remaining quadrotor should seamlessly complete the task. A simple reward function was used: rewards were given for exploring
new runway tiles and penalties were given for quadrotor proximity. The desired behaviour was successfully learned in simulation and exported to a real outdoor runway facility, complete with two GPS-driven quadrotors. Their cooperative exploration and fault-tolerance behaviour was successfully demonstrated. A video summarizing the research in Chap. 4 can be found at \url{https://youtu.be/Pu5rWnLgyZs}.

Lastly, in Chapter 5, armed with a suitable deep reinforcement learning-based guidance technique for aerospace vehicles, developed over the preceding three papers, the author returned to the central motivation of this research: autonomous manipulator-based capture of a spinning spacecraft. The deep reinforcement learning-based guidance technique was applied to the capture scenario, where rewards were given for proper docking and penalties were given for collisions (among a few other rewards). From this reward function, the desired behaviour was learned: the successful capture of a spinning target with a robotic manipulator. In addition, a number of assumptions were removed compared to previous work. The trained guidance model was then exported for use in Carleton University’s Spacecraft Robotics and Control Laboratory, where experimental validation was successfully demonstrated, along with investigations into the model’s ability to handle perturbations and initial conditions not seen during training. A video summary of Chap. 5 can be found at \url{https://youtu.be/_oWpEH_dalo}, and details of the author’s contributions to the laboratory can be found in App. B.

This research designed, trained, and analyzed a technique that allows the aerospace community to take advantage of the recent successes of the deep reinforcement learning community: expert decision making can be obtained with learning-based approaches. At each step of the development process, the author made sure to experimentally validate the work on a range of aerospace applications.

6.2 Major Contributions

The major contributions of this work to the field of aerospace engineering are:

- A deep reinforcement learning-based guidance algorithm for aerospace vehicles that enables state-of-the-art behaviour and transfers well from simulation to
reality. The algorithm was developed over four papers.

- Experimental demonstrations of deep reinforcement learning techniques being applied to quadrotors in a cooperative and fault-tolerant way.

- The first, to the best of the author’s knowledge, experimental demonstration of applying artificial intelligence techniques to a spacecraft platform in experiment for both proximity operations and manipulator-based capture.

- Improvements to the Spacecraft Proximity Operations Testbed laboratory at Carleton University, consisting of: 1) a complete mass property determination; and 2) an improved control allocation strategy that accounts for both the manipulator’s real-time orientation and thruster pressure drop. Details of this work can be found in App. B.

The significance of this work is evidenced by the publications and presentations in guidance, dynamics, and control-related journals and conferences, listed below:

**Journal Papers**


Conference Proceedings


6.3 Future Work

Several areas of future work remain:

**Stability**

Traditional guidance, navigation, and control approaches often demand stability proofs before they are to be trusted; however, stability cannot be readily proven with neural networks (though the neural network can be shown to work in many scenarios numerically). A question the aerospace guidance, navigation, and control field will have to answer is: does the improved performance of learning-based techniques come at too high a cost due to the lack of stability guarantees? Personally, the author believes that the improved performance and reduction in engineering effort associated with learning-based techniques is worth pursuing.

**Six Degree of Freedom**

This thesis considered planar motion due to the gravity-offset spacecraft facility at Carleton University being constrained to planar motion. However, these techniques must be extended to six-degree of freedom to be useful for on-orbit demonstrations.
It is hypothesized that the extension to three-dimensional motion will be trivial but may require more training time due to the larger state- and action-spaces.

**More Difficult Applications**

Now that the deep reinforcement learning-based guidance technique has been developed, and its capabilities have been demonstrated in a number of scenarios, it is time to probe its limits. What class of problems are too difficult for it to solve? What other domains is it applicable to? Some initial ideas: 1) multi-agent scenarios without perfect communication (i.e., each agent only has state information about its nearest neighbour); or 2) partially-observable Markov decision process problems (i.e., not having perfect state information available, but instead a measurement, i.e. a camera image, from which the state must be inferred in real-time).

**6.3.1 Thruster Minimization and Plume Interactions**

All spacecraft proximity operations research must consider the problem of thruster plumes from the chaser affecting the motion of the target. Future work could have the reward function decentivize chaser behaviour that causes thruster plume interactions. Additionally, since desired accelerations are directly related to thruster use, penalties could be included for any acceleration signals, encouraging a thrust-minimized policy to emerge.

**6.3.2 Measurement Noise**

Many real-time measurements in Chaps. 2 and 5 were assumed to be known accurately, which may not be the case during a real mission. For example, a sub-millimetre accurate ground-truth system will not be available during spacecraft proximity operations—real navigation with cameras and other sensors will be used. Studying how real measurement noise affects this work is another important research direction to build system trust.
References


doi:10.1109/ICRA.2017.7989250,  
arXiv:1609.07088

doi:10.15607/RSS.2017.XIII.034

doi:10.1109/TRO.2019.2942989

arXiv:1808.00177

arXiv:1703.00472

doi:arXiv:1804.10332v2,  
arXiv:1804.10332

doi:10.1109/IROS.2017.8202133,  
arXiv:1703.06907

arXiv:1710.06425


doi:10.2514/6.2020-0458

doi:10.2514/6.2020-1910

doi:10.1109/JAS.2014.7004667


doi:10.2514/1.G002357

doi:10.2514/1.G001491


doi:10.2514/1.G001309

arXiv:1804.08617
arXiv:1707.06887

doi:10.1109/TCST.2018.2866963

doi:10.1016/j.actaastro.2014.01.002

doi:10.1016/j.ast.2018.04.009

doi:10.1016/j.actaastro.2014.10.027

doi:10.1109/MCSE.2007.58

arXiv:1509.02971

arXiv:1412.6980

arXiv:1603.04467

doi:10.3929/ethz-a-010085499

doi:10.2514/6.2007-6461

doi:10.1109/IROS.2007.4399042

doi:10.1109/IROS.2006.282433

doi:10.1109/CDC.2006.377588


doi:10.1049/iet-cta:20080002

doi:10.1109/IROS.2016.7759827

doi:10.2514/1.G001490

doi:10.1016/j.conengprac.2018.01.003

doi:10.1109/ROBOT.2002.1013341

doi:10.1109/robot.2003.1242264


doi:10.1109/ICRA.2011.5980409

doi:10.1109/TRO.2018.2853613

doi:10.1177/0278364911434236

arXiv:1910.07113

arXiv:1901.07517


doi:10.1109/TAC.2003.809799

doi:10.1142/S0218488598000094

doi:10.4233/uuid:b38fadb7-e6bd-44d0-93be-f7dd1457be60

doi:10.14311/MAD.2015.16.04


doi:10.1126/scirobots.abc3000

arXiv:1712.01815

doi:10.1177/0278364919887447


Appendix A

Elements of the Inertia and Coriolis Matrices

These equations are used in Eqs. (5.18) and (5.19) in Chap. 5. The elements $M_{ij} \forall i = 1, \ldots, 6, j = 1, \ldots, 6$ are defined below:

\[
M_{11} = m_c + m_1 + m_2 + m_3
\]
\[
M_{12} = 0
\]
\[
M_{13} = (-a_1 m_1 - m_2 (a_1 + b_1) - m_3 (a_1 + b_1)) \cos(q_c + q_1) + (-a_2 m_2 - m_3 (a_2 + b_2)) \cos(q_c + q_1 + q_2)
\]
\[
\quad + (-a_3 m_3) \cos(q_c + q_1 + q_2 + q_3) + (-b_c m_1 - b_c m_2 - b_c m_3) \sin(\phi_c + q_c)
\]
\[
M_{14} = (-a_1 m_1 - m_2 (a_1 + b_1) - m_3 (a_1 + b_1)) \cos(q_c + q_1) + (-a_2 m_2 - m_3 (a_2 + b_2)) \cos(q_c + q_1 + q_2)
\]
\[
\quad + (-a_3 m_3) \cos(q_c + q_1 + q_2 + q_3)
\]
\[
M_{15} = (-a_2 m_2 - a_2 m_3 - b_2 m_3) \cos(q_c + q_1 + q_2) + (-a_3 m_3) \cos(q_c + q_1 + q_2 + q_3)
\]
\[
M_{16} = (-a_3 m_3) \cos(q_c + q_1 + q_2 + q_3)
\]
\[
M_{21} = 0
\]
\[
M_{22} = m_c + m_1 + m_2 + m_3
\]
\[
M_{23} = (b_c m_1 + b_c m_2 + b_c m_3) \cos(\phi_c + q_c) + (-a_1 m_1 - m_2 (a_1 + b_1) - m_3 (a_1 + b_1)) \sin(q_c + q_1)
\]
\[
\quad + (-a_2 m_2 - m_3 (a_2 + b_2)) \sin(q_c + q_1 + q_2) + (-a_3 m_3) \sin(q_c + q_1 + q_2 + q_3)
\]
\[
M_{24} = (-a_1 m_1 - m_2 (a_1 + b_1) - m_3 (a_1 + b_1)) \sin(q_c + q_1) + (-a_2 m_2 - m_3 (a_2 + b_2)) \sin(q_c + q_1 + q_2)
\]
\[
\quad + (-a_3 m_3) \sin(q_c + q_1 + q_2 + q_3)
\]
\[
M_{25} = (-a_2 m_2 - a_2 m_3 - b_2 m_3) \sin(q_c + q_1 + q_2) + (-a_3 m_3) \sin(q_c + q_1 + q_2 + q_3)
\]
\[
M_{26} = (-a_3 m_3) \sin(q_c + q_1 + q_2 + q_3)
\]
\[
M_{31} = M_{13}
\]
\[
M_{32} = M_{23}
\]
\[ M_{33} = (2a_1a_2m_2 + 2a_1a_2m_3 + 2a_2b_1m_2 + 2a_1b_2m_3 + 2a_2b_1m_3 + 2b_1b_2m_3) \cos(q_2) \\
+ (2a_2a_3m_3 + 2a_3b_2m_3) \cos(q_3) + (2a_1a_3m_3 + 2a_3b_1m_3) \cos(q_2 + q_3) \\
+ (2a_1b_1m_1 + 2a_1b_2m_2 + 2a_1b_2m_3 + 2b_1b_1m_2 + 2b_2b_1m_3) \sin(\phi_c - q_1) \\
+ (-2a_2b_c m_2 - 2a_2b_c m_3 - 2b_c b_2m_3) \sin(q_1 - \phi_c + q_2) + (-2a_3b_c m_3) \sin(q_1 - \phi_c + q_2 + q_3) \\
+ I_c + I_1 + I_2 + I_3 + a_1^2 m_1 + a_1^2 m_2 + a_1^2 m_3 + a_2^2 m_2 + a_2^2 m_3 + a_3^2 m_3 + b_c^2 m_1 + b_c^2 m_2 \\
+ b_c^2 m_3 + b_1^2 m_2 + b_1^2 m_3 + b_2^2 m_3 + 2a_1b_1m_2 + 2a_1b_1m_3 + 2a_2b_2m_3 \\
M_{34} = (2a_1a_2m_2 + 2a_1a_2m_3 + 2a_2b_1m_2 + 2a_1b_2m_3 + 2a_2b_1m_3 + 2b_1b_2m_3) \cos(q_2) \\
+ (2a_2a_3m_3 + 2a_3b_2m_3) \cos(q_3) + (2a_1a_3m_3 + 2a_3b_1m_3) \cos(q_2 + q_3) \\
+ (a_1b_cm_1 + a_1b_cm_2 + a_1b_cm_3 + b_c b_1m_2 + b_c b_1m_3) \sin(\phi_c - q_1) \\
+ (-a_2b_c m_2 - a_2b_c m_3 - b_c b_2m_3) \sin(q_1 - \phi_c + q_2) + (-a_3b_c m_3) \sin(q_1 - \phi_c + q_2 + q_3) \\
+ I_1 + I_2 + I_3 + a_1^2 m_1 + a_1^2 m_2 + a_1^2 m_3 + a_2^2 m_2 + a_2^2 m_3 + a_3^2 m_3 + b_1^2 m_2 + b_1^2 m_3 + b_2^2 m_3 \\
+ 2a_1b_1m_2 + 2a_1b_1m_3 + 2a_2b_2m_3 \\
M_{35} = (a_1a_2m_2 + a_1a_2m_3 + a_2b_1m_2 + a_1b_2m_3 + a_3b_1m_3 + b_1b_2m_3) \cos(q_2) \\
+ (2a_2a_3m_3 + 2a_3b_2m_3) \cos(q_3) + (a_1a_3m_3 + a_3b_1m_3) \cos(q_2 + q_3) \\
+ (-a_2b_c m_2 - a_2b_c m_3 - b_c b_2m_3) \sin(q_1 - \phi_c + q_2) + (-a_3b_c m_3) \sin(q_1 - \phi_c + q_2 + q_3) \\
+ I_2 + I_3 + a_2^2 m_2 + a_2^2 m_3 + a_3^2 m_3 + b_2^2 m_3 + 2a_2b_2m_3 \\
M_{36} = (a_2a_3m_3 + a_3b_2m_3) \cos(q_3) + (a_1a_3m_3 + a_3b_1m_3) \cos(q_2 + q_3) \\
+ (-a_3b_c m_3) \sin(q_1 - \phi_c + q_2 + q_3) + m_3 a_3^2 + I_3 \\
M_{41} = M_{14} \\
M_{42} = M_{24} \\
M_{43} = M_{34} \\
M_{44} = (2a_1a_2m_2 + 2a_1a_2m_3 + 2a_2b_1m_2 + 2a_1b_2m_3 + 2a_2b_1m_3 + 2b_1b_2m_3) \cos(q_2) \\
+ (2a_2a_3m_3 + 2a_3b_2m_3) \cos(q_3) + (2a_1a_3m_3 + 2a_3b_1m_3) \cos(q_2 + q_3) + \ldots \]
\[I_1 + I_2 + I_3 + a_1^2 m_1 + a_1^2 m_2 + a_1^2 m_3 + a_2^2 m_2 + a_2^2 m_3 + a_3^2 m_3 + b_1^2 m_2 + b_1^2 m_3 + b_2^2 m_3 + 2 a_1 b_1 m_2 + 2 a_1 b_1 m_3 + 2 a_2 b_2 m_3\]

\[M_{45} = (a_1 a_2 m_2 + a_1 a_2 m_3 + a_2 b_1 m_2 + a_1 b_1 m_3 + a_2 b_1 m_3 + b_1 b_2 m_3) \cos(q_2) + (2 a_2 a_3 m_3 + 2 a_3 b_2 m_3) \cos(q_3) + (a_1 a_3 m_3 + a_3 b_1 m_3) \cos(q_2 + q_3) + I_2 + I_3 + a_2^2 m_2 + a_2^2 m_3 + a_3^2 m_3 + b_2^2 m_3 + 2 a_2 b_2 m_3\]

\[M_{46} = (a_2 a_3 m_3 + a_3 b_2 m_3) \cos(q_3) + (a_1 a_3 m_3 + a_3 b_1 m_3) \cos(q_2 + q_3) + m_3 a_3^2 + I_3\]

\[M_{51} = M_{15}\]
\[M_{52} = M_{25}\]
\[M_{53} = M_{35}\]
\[M_{54} = M_{45}\]
\[M_{55} = (2 a_2 a_3 m_3 + 2 a_3 b_2 m_3) \cos(q_3) + I_2 + I_3 + a_2^2 m_2 + a_2^2 m_3 + a_3^2 m_3 + b_2^2 m_3 + 2 a_2 b_2 m_3\]

\[M_{56} = (a_2 a_3 m_3 + a_3 b_2 m_3) \cos(q_3) + m_3 a_3^2 + I_3\]

\[M_{61} = M_{16}\]
\[M_{62} = M_{26}\]
\[M_{63} = M_{36}\]
\[M_{64} = M_{46}\]
\[M_{65} = M_{56}\]
\[M_{66} = m_3 a_3^2 + I_3\]
The components $C_{ij} \forall i = 1...6, j = 1...6$ are derived and presented below:

\[ C_{11} = 0 \]
\[ C_{12} = 0 \]
\[ C_{13} = (-\dot{\phi}_0 (b_c m_1 + b_c m_2 + b_c m_3)) \cos(\phi_c + q_c) \]
\[ + (-\dot{\phi}_0 (a_1 m_1 + m_2 (a_1 + b_1) + m_3 (a_1 + b_1)) - \dot{q}_1 (a_1 m_1 + m_2 (a_1 + b_1) + m_3 (a_1 + b_1))) \cos(q_c + q_1 + \frac{\pi}{2}) \]
\[ + (-\dot{\phi}_0 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{q}_1 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{q}_2 (a_2 m_2 + m_3 (a_2 + b_2))) \cos(q_c + q_1 + q_2 + \frac{\pi}{2}) \]
\[ + (-a_3 m_3 \dot{q}_0 - a_3 m_3 \dot{q}_1 - a_3 m_3 \dot{q}_2 - a_3 m_3 \dot{q}_3) \cos(q_c + q_1 + q_2 + q_3 + \frac{\pi}{2}) \]
\[ C_{14} = (-\dot{\phi}_0 (a_1 m_1 + m_2 (a_1 + b_1) + m_3 (a_1 + b_1)) - \dot{q}_1 (a_1 m_1 + m_2 (a_1 + b_1) + m_3 (a_1 + b_1))) \cos(q_c + q_1 + q_2 + q_3) \]
\[ + (-\dot{\phi}_0 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{q}_1 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{q}_2 (a_2 m_2 + m_3 (a_2 + b_2))) \cos(q_c + q_1 + q_2 + q_3) \]
\[ + (-a_3 m_3 \dot{q}_0 - a_3 m_3 \dot{q}_1 - a_3 m_3 \dot{q}_2 - a_3 m_3 \dot{q}_3) \cos(q_c + q_1 + q_2 + q_3 + \frac{\pi}{2}) \]
\[ C_{15} = (a_3 m_3 (\dot{q}_0 + \dot{q}_1 + \dot{q}_2 + \dot{q}_3)) \sin(q_c + q_1 + q_2 + q_3) \]
\[ C_{16} = 0 \]
\[ C_{21} = 0 \]
\[ C_{22} = 0 \]
\[ C_{23} = (-\dot{\phi}_0 (b_c m_1 + b_c m_2 + b_c m_3)) \sin(\phi_c + q_c) \]
\[ + (-\dot{\phi}_0 (a_1 m_1 + m_2 (a_1 + b_1) + m_3 (a_1 + b_1)) - \dot{q}_1 (a_1 m_1 + m_2 (a_1 + b_1) + m_3 (a_1 + b_1))) \sin(q_c + q_1 + \frac{\pi}{2}) \]
\[ + (-\dot{\phi}_0 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{q}_1 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{q}_2 (a_2 m_2 + m_3 (a_2 + b_2))) \sin(q_c + q_1 + q_2 + \frac{\pi}{2}) \]
\[ + (-a_3 m_3 \dot{q}_0 - a_3 m_3 \dot{q}_1 - a_3 m_3 \dot{q}_2 - a_3 m_3 \dot{q}_3) \sin(q_c + q_1 + q_2 + q_3 + \frac{\pi}{2}) \]
\[ C_{24} = (-\dot{\phi}_0 (a_1 m_1 + m_2 (a_1 + b_1) + m_3 (a_1 + b_1)) - \dot{q}_1 (a_1 m_1 + m_2 (a_1 + b_1) + m_3 (a_1 + b_1))) \sin(q_c + q_1 + \frac{\pi}{2}) \]
\[ + (-\dot{\phi}_0 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{q}_1 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{q}_2 (a_2 m_2 + m_3 (a_2 + b_2))) \sin(q_c + q_1 + q_2 + \frac{\pi}{2}) \]
\[ C_{25} = (-\dot{\theta}_0 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{\phi}_1 (a_2 m_2 + m_3 (a_2 + b_2)) - \dot{\phi}_2 (a_2 m_2 + m_3 (a_2 + b_2))) \sin(q_c + q_1 + q_2 + q_3 + \frac{\pi}{2}) \\
C_{26} = (-a_3 m_3 (\dot{\theta}_0 + \dot{\phi}_1 + \dot{\phi}_2 + \dot{\phi}_3)) \cos(q_c + q_1 + q_2 + q_3) \\
C_{31} = 0 \\
C_{32} = 0 \\
C_{33} = (-a_1 b_c m_1 \dot{\theta}_1 - a_1 b_c m_2 \dot{\theta}_1 - a_1 b_c m_3 \dot{\theta}_1 - b_c b_1 m_2 \dot{\phi}_1 - b_c b_1 m_3 \dot{\phi}_1) \cos(\phi_c - q_1) \\
+ (-a_2 b_c m_2 \dot{\phi}_1 - a_2 b_c m_3 \dot{\phi}_1 - a_2 b_c m_3 \dot{\phi}_1 - b_c b_2 m_3 \dot{\phi}_1) \cos(q_1 - \phi_c + q_2 + q_3) \\
+ (-a_2 b_c m_2 \dot{\phi}_1 - a_2 b_c m_3 \dot{\phi}_1 - a_2 b_c m_3 \dot{\phi}_1 - b_c b_2 m_3 \dot{\phi}_1) \cos(q_1 - \phi_c + q_2 + q_3) \\
a_1 a_2 m_3 \dot{\phi}_2) \sin(q_2) \\
+ (-a_3 b_c m_3 \dot{\phi}_3 - a_2 a_3 m_3 \dot{\phi}_3) \sin(q_3) \\
+ (-a_3 b_c m_3 \dot{\phi}_3 - a_2 a_3 m_3 \dot{\phi}_3) \sin(q_3) \\
C_{34} = (-a_1 b_c m_1 \dot{\theta}_0 - a_1 b_c m_1 \dot{\theta}_1 - a_1 b_c m_2 \dot{\theta}_0 - a_1 b_c m_2 \dot{\theta}_1 - a_1 b_c m_3 \dot{\theta}_0 - a_1 b_c m_3 \dot{\theta}_1 \\
- b_c b_1 m_2 \dot{\phi}_0 - b_c b_1 m_2 \dot{\phi}_1 - b_c b_1 m_3 \dot{\phi}_0 - b_c b_1 m_3 \dot{\phi}_1) \cos(\phi_c - q_1) \\
+ (-a_2 b_c m_2 \dot{\phi}_0 - a_2 b_c m_2 \dot{\phi}_1 \\
- a_2 b_c m_3 \dot{\phi}_0 - a_2 b_c m_2 \dot{\phi}_2 - a_2 b_c m_3 \dot{\phi}_1 - a_2 b_c m_2 \dot{\phi}_2 - b_c b_2 m_3 \dot{\phi}_0 - b_c b_2 m_3 \dot{\phi}_1 \\
b_c b_2 m_3 \dot{\phi}_2) \cos(q_1 - \phi_c + q_2) + (-a_3 b_c m_3 \dot{\phi}_0 - a_3 b_c m_3 \dot{\phi}_1 - a_3 b_c m_3 \dot{\phi}_2 - a_3 b_c m_3 \dot{\phi}_3) \cos(q_1 - \phi_c + q_2 + q_3) \\
+ (-a_2 b_1 m_2 \dot{\phi}_2 - a_2 b_1 m_3 \dot{\phi}_2 - a_2 b_1 m_3 \dot{\phi}_2 - b_1 b_2 m_3 \dot{\phi}_2 - a_1 a_2 m_3 \dot{\phi}_2) \sin(q_2) \\
+ (-a_3 b_2 m_3 \dot{\phi}_3 - a_2 a_3 m_3 \dot{\phi}_3) \sin(q_3) + (-a_3 b_1 m_3 \dot{\phi}_2 - a_3 b_1 m_3 \dot{\phi}_3 - a_1 a_3 m_3 \dot{\phi}_2 - a_1 a_3 m_3 \dot{\phi}_3) \sin(q_2 + q_3) \\
C_{35} = (-a_2 b_c m_2 \dot{\theta}_0 - a_2 b_c m_2 \dot{\theta}_1 - a_2 b_c m_2 \dot{\theta}_1 - a_2 b_c m_3 \dot{\theta}_0 - a_2 b_c m_3 \dot{\theta}_2 - a_2 b_c m_3 \dot{\theta}_2 \\
- b_c b_2 m_3 \dot{\phi}_0 - b_c b_2 m_3 \dot{\phi}_1 - b_c b_2 m_3 \dot{\phi}_2) \cos(q_1 - \phi_c + q_2) + (-a_3 b_c m_3 \dot{\phi}_0 - a_3 b_c m_3 \dot{\phi}_1 \\
a_3 b_c m_3 \dot{\phi}_1 \\
a_3 b_c m_3 \dot{\phi}_2 - a_3 b_c m_3 \dot{\phi}_3) \cos(q_1 - \phi_c + q_2 + q_3) + (-a_2 b_1 m_2 \dot{\theta}_0 - a_1 b_2 m_3 \dot{\theta}_0) \]
\[-a_2 b_1 m_2 \dot{q}_1 - a_2 b_1 m_3 \dot{q}_0 - a_1 b_2 m_3 \dot{q}_1 - a_2 b_1 m_2 \dot{q}_2 - a_2 b_1 m_3 \dot{q}_1 - a_1 b_2 m_3 \dot{q}_0 - a_2 b_1 m_3 \dot{q}_0 \sin(q_2) + (-b_1 b_2 m_3 \dot{q}_1 - b_1 b_2 m_3 \dot{q}_2 - a_1 b_2 m_2 \dot{q}_0 - a_1 a_2 m_3 \dot{q}_0 - a_1 a_2 m_3 \dot{q}_1 - a_1 a_2 m_2 \dot{q}_2 - a_1 a_2 m_3 \dot{q}_1) \sin(q_2) + (-a_3 b_3 m_3 \dot{q}_3 - a_2 a_3 m_3 \dot{q}_3) \sin(q_3) + (-a_3 b_3 m_3 \dot{q}_0 - a_3 b_3 m_3 \dot{q}_1 - a_3 b_3 m_3 \dot{q}_2 - a_3 b_3 m_3 \dot{q}_3 - a_1 a_3 m_3 \dot{q}_0 - a_1 a_3 m_3 \dot{q}_1 - a_1 a_3 m_3 \dot{q}_2 - a_1 a_3 m_3 \dot{q}_3) \sin(q_2) \]

\[C_{36} = (-a_3 b_3 m_3 (\dot{q}_0 + \dot{q}_1 + \dot{q}_2 + \dot{q}_3)) \cos(q_1 - \phi_c + q_2 + q_3) + (-a_3 m_3 (a_2 + b_2) (\dot{q}_0 + \dot{q}_1 + \dot{q}_2 + \dot{q}_3)) \sin(q_3) \]

\[+ (-a_3 m_3 (a_1 + b_1) (\dot{q}_0 + \dot{q}_1 + \dot{q}_2 + \dot{q}_3)) \sin(q_2 + q_3) \]

\[C_{41} = 0 \]

\[C_{42} = 0 \]

\[C_{43} = (a_1 b_1 m_1 \dot{q}_0 + a_1 b_1 m_2 \dot{q}_0 + a_1 b_1 m_3 \dot{q}_0 + b_c b_1 m_2 \dot{q}_0 + b_c b_1 m_3 \dot{q}_0) \cos(\phi_c - q_1) + (a_2 b_2 m_2 \dot{q}_0 + a_3 b_2 m_3 \dot{q}_0 + b_c b_2 m_3 \dot{q}_0) \cos(q_1 - \phi_c + q_2) + (a_3 b_3 m_3 \dot{q}_0) \cos(q_1 - \phi_c + q_2 + q_3) + (-a_3 b_1 m_2 \dot{q}_1 - a_4 b_1 m_3 \dot{q}_2 - a_2 b_1 m_3 \dot{q}_2 - b_1 b_2 m_3 \dot{q}_2 - a_4 b_1 m_3 \dot{q}_3 - a_3 b_1 m_3 \dot{q}_3 - a_3 b_1 m_3 \dot{q}_3 - a_1 a_3 m_3 \dot{q}_3) \sin(q_2 + q_3) \]

\[C_{44} = (-a_2 b_1 m_2 \dot{q}_2 - a_1 b_2 m_3 \dot{q}_2 - a_2 b_1 m_3 \dot{q}_2 - b_1 b_2 m_3 \dot{q}_2 - a_1 b_2 m_3 \dot{q}_2 - a_1 a_2 m_3 \dot{q}_2) \sin(q_2) + (-a_3 b_2 m_3 \dot{q}_3 - a_2 a_3 m_3 \dot{q}_3) \sin(q_3) + (-a_3 b_1 m_3 \dot{q}_2 - a_4 b_1 m_3 \dot{q}_3) \sin(q_2 + q_3) \]

\[C_{45} = (-a_2 b_1 m_2 \dot{q}_0 - a_1 b_2 m_3 \dot{q}_0 - a_2 b_1 m_2 \dot{q}_1 - a_2 b_1 m_3 \dot{q}_0 - a_1 b_2 m_3 \dot{q}_1 - a_2 b_1 m_2 \dot{q}_2 - a_1 b_2 m_3 \dot{q}_2 - a_2 b_1 m_3 \dot{q}_2 - b_1 b_2 m_3 \dot{q}_0) \sin(q_2) + (-b_1 b_2 m_3 \dot{q}_1 - b_1 b_2 m_3 \dot{q}_2 - a_1 a_2 m_3 \dot{q}_1 - a_1 a_2 m_3 \dot{q}_0 - a_1 a_2 m_3 \dot{q}_0 - a_1 a_2 m_3 \dot{q}_0 - a_1 a_2 m_3 \dot{q}_0) \sin(q_2) + (-a_3 b_2 m_3 \dot{q}_3 - a_2 a_3 m_3 \dot{q}_3) \sin(q_3) + (-a_3 b_1 m_3 \dot{q}_0 - a_3 b_1 m_3 \dot{q}_1 - a_3 b_1 m_3 \dot{q}_2 - a_3 b_1 m_3 \dot{q}_3 - a_1 a_3 m_3 \dot{q}_0 - a_1 a_3 m_3 \dot{q}_1 - a_1 a_3 m_3 \dot{q}_2 - a_1 a_3 m_3 \dot{q}_3) \sin(q_2 + q_3) \]

\[C_{46} = (-a_3 m_3 (a_2 + b_2) (\dot{q}_0 + \dot{q}_1 + \dot{q}_2 + \dot{q}_3)) \sin(q_3) + (-a_3 m_3 (a_1 + b_1) (\dot{q}_0 + \dot{q}_1 + \dot{q}_2 + \dot{q}_3)) \sin(q_2 + q_3) \]
$C_{51} = 0$

$C_{52} = 0$

$C_{53} = (a_2 b_c m_2 \dot{q}_0 + a_2 b_c m_3 \dot{q}_0 + b_c b_2 m_3 \dot{q}_0) \cos(q_1 - \phi_c + q_2) + (a_3 b_c m_3 \dot{q}_0) \cos(q_1 - \phi_c + q_2 + q_3) + (a_2 b_1 m_2 \dot{q}_0 + a_1 b_2 m_3 \dot{q}_0 + a_2 b_1 m_3 \dot{q}_0 + a_1 b_2 m_3 \dot{q}_0 + a_2 b_1 m_3 \dot{q}_1 + a_2 b_1 m_3 \dot{q}_1 + b_1 b_2 m_3 \dot{q}_0 + b_1 b_2 m_3 \dot{q}_0 + a_1 a_2 m_2 \dot{q}_0 + a_1 a_2 m_2 \dot{q}_1 + a_1 a_2 m_3 \dot{q}_0 + a_1 a_2 m_3 \dot{q}_1) \sin(q_2)

+ (-a_3 b_2 m_3 \dot{q}_3 - a_2 a_3 m_3 \dot{q}_3) \sin(q_3) + (a_3 b_1 m_3 \dot{q}_0 + a_3 b_1 m_3 \dot{q}_3 + a_1 a_3 m_3 \dot{q}_0 + a_1 a_3 m_3 \dot{q}_1) \sin(q_2 + q_3)

C_{54} = (a_2 b_1 m_2 \dot{q}_0 + a_1 b_2 m_3 \dot{q}_0 + a_2 b_1 m_3 \dot{q}_0 + a_1 b_2 m_3 \dot{q}_0 + a_2 b_1 m_3 \dot{q}_1 + a_2 b_1 m_3 \dot{q}_1 + b_1 b_2 m_3 \dot{q}_0 + b_1 b_2 m_3 \dot{q}_1 + a_1 a_2 m_2 \dot{q}_0 + a_1 a_2 m_2 \dot{q}_1 + a_1 a_2 m_3 \dot{q}_0 + a_1 a_2 m_3 \dot{q}_1) \sin(q_2)

+ (-a_3 b_2 m_3 \dot{q}_3 - a_2 a_3 m_3 \dot{q}_3) \sin(q_3) + (a_3 b_1 m_3 \dot{q}_0 + a_3 b_1 m_3 \dot{q}_3 + a_1 a_3 m_3 \dot{q}_0 + a_1 a_3 m_3 \dot{q}_1) \sin(q_2 + q_3)

C_{55} = (-a_3 m_3 \dot{q}_3 (a_2 + b_2)) \sin(q_3)

C_{56} = (-a_3 m_3 (a_2 + b_2) (\dot{q}_0 + \dot{q}_1 + \dot{q}_2 + \dot{q}_3)) \sin(q_3)

C_{61} = 0

C_{62} = 0

C_{63} = (a_3 b_c m_3 \dot{q}_0) \cos(q_1 - \phi_c + q_2 + q_3) + (a_3 m_3 (a_2 \dot{q}_0 + a_2 \dot{q}_1 + a_2 \dot{q}_2 + b_2 \dot{q}_0 + b_2 \dot{q}_1 + b_2 \dot{q}_2)) \sin(q_3) + (a_3 m_3 (a_1 \dot{q}_0 + a_1 \dot{q}_1 + b_1 \dot{q}_0 + b_1 \dot{q}_1)) \sin(q_2 + q_3)

C_{64} = (a_3 m_3 (a_2 \dot{q}_0 + a_2 \dot{q}_1 + a_2 \dot{q}_2 + b_2 \dot{q}_0 + b_2 \dot{q}_1 + b_2 \dot{q}_2)) \sin(q_3)

+ (a_3 m_3 (a_1 \dot{q}_0 + a_1 \dot{q}_1 + b_1 \dot{q}_0 + b_1 \dot{q}_1)) \sin(q_2 + q_3)

C_{65} = (a_3 m_3 (a_2 + b_2) (\dot{q}_0 + \dot{q}_1 + \dot{q}_2)) \sin(q_3)

C_{66} = 0
Appendix B

SPOT Laboratory Contributions

This section details the contributions that were made to the Spacecraft Proximity Operations Testbed, part of Carleton University’s Spacecraft Robotics and Control Laboratory, during the course of the author’s PhD.

B.1 Mass Property Determination

The feedforward controller presented in Sec. 5.5.1 requires an accurate dynamics model of the chaser spacecraft and manipulator. A measurement campaign was undertaken to measure all relevant mass properties as accurately as possible.

B.1.1 Mass Measurements

The chaser spacecraft base mass was measured, without the robotic manipulator, using three kitchen food scales with a 15 kg limit and 1 g resolution, as shown in Fig. B.1a. A full air tank was installed to provide a realistic measurement. The sum of the three measurements is taken as the total mass of the platform. The centre of mass location was also obtained as the ratio of the different measurements, as described in [141].

The manipulator was disassembled and each link was measured separately. One manipulator link being measured is shown in Fig. B.1b. The centre of mass location of each link is obtained using the ratio of the two readings, as outlined in [141].

B.1.2 Moment of Inertia Measurements

A bifilar pendulum was constructed to measure the inertia of each spacecraft component. Only the moment of inertia about the Z axis is desired, since that’s the only axis the spacecraft platform can rotate about during its planar experiments. The
(a) Base spacecraft mass and centre of mass measurement

(b) Manipulator mass and centre of mass measurement

**Figure B.1:** Mass property measurements with digital scales.
Table B.1: Chaser and target mass and inertia properties under different configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Mass (kg)</th>
<th>Inertia (kg-m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target (yes panels)</td>
<td>12.039</td>
<td>0.2257</td>
</tr>
<tr>
<td>Chaser (no panels; yes shoulder motor)</td>
<td>11.211</td>
<td>0.2022</td>
</tr>
<tr>
<td>Chaser (no panels; no shoulder motor)</td>
<td>10.876</td>
<td>0.1843</td>
</tr>
<tr>
<td>Chaser (yes panels; no shoulder motor)</td>
<td>12.734</td>
<td>0.2209</td>
</tr>
<tr>
<td>Chaser (yes panels; yes shoulder motor)</td>
<td>13.072</td>
<td>0.2405</td>
</tr>
<tr>
<td>Bicep</td>
<td>0.345</td>
<td>0.003704</td>
</tr>
<tr>
<td>Forearm</td>
<td>0.335</td>
<td>0.003506</td>
</tr>
<tr>
<td>End-effector</td>
<td>0.0621</td>
<td>0.000106</td>
</tr>
</tbody>
</table>

spacecraft base and manipulator link on the bifilar pendulum are shown in Fig. B.2. The object was given a small angular displacement and the average period of oscillation, \( \tau \), was measured over five cycles. In addition, the experiment was repeated thrice to limit measurement bias. The distance between the bifilars was measured to be: \( d = 0.28 \text{ m} \), and the length of the bifilars was \( L = 2.529 \text{ m} \) (though this changes with each measurement). It is important to ensure the object being measured is level, such that the rotation is indeed about the desired axis (use a ‘taut line hitch’ knot to achieve this). Once all measurements are taken, the moment of inertia about the specified axis can be calculated using

\[
I = \frac{\tau^2 mgd^2}{16\pi^2 L}
\]  

(M.1)

Mass property values for various spacecraft configurations are shown in Table B.1. All values include a filled air tank at the time of measurement, and no reaction wheels present on either spacecraft.

B.2 Control Mixer

In order for the spacecraft to meet the desired forces and torques from the controller at each time step, software must be used to convert the desired forces into durations each of the eight air thrusters should be opened for. This process is performed using a control mixer (also known as a control allocation strategy). Since the air thrusters are controlled by solenoid valves that are either: a) fully open; or b) fully closed, a
Figure B.2: Moment of inertia measurements with a bifilar pendulum.
pulse-width modulation scheme is used to control the percentage of time each valve is open (also called the duty cycle). A frequency of 10 Hz is used, i.e., the duty cycle is recalculated ten times per second. Within each period, the duty cycle dictates the percentage of time a specified thruster is open for. For example, a duty cycle of 0.2 at 10 Hz yields a thruster that is open from 0 to 0.02 s then closed from 0.02 to 0.1 s. The Raspberry Pi-3 computer has a minimum pin-on time of 100 µs, and therefore, a PWM resolution of 0.1% at 10 Hz. The hardware solenoid valve has a switching time of 0.007 s, leading to a minimum duty cycle of 7%.

The resulting forces on the spacecraft, in its body frame, by the eight air thrusters is given by

$$\begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix} = Hu$$

(B.2)

with

$$H = \begin{bmatrix} F_{1x} & F_{2x} & F_{3x} & F_{4x} & F_{5x} & F_{6x} & F_{7x} & F_{8x} \\ F_{1y} & F_{2y} & F_{3y} & F_{4y} & F_{5y} & F_{6y} & F_{7y} & F_{8y} \\ \tau_{1z} & \tau_{2z} & \tau_{3z} & \tau_{4z} & \tau_{5z} & \tau_{6z} & \tau_{7z} & \tau_{8z} \end{bmatrix}$$

(B.3)

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \end{bmatrix}^T$$

(B.4)

where $u_i \, \forall \, i = 1, \ldots, 8$ are the duty cycle values for each of the eight thrusters, $F_{ix}$ and $F_{iy} \, \forall \, i = 1, \ldots, 8$ are the x- and y-components of thrust generated by each thruster (in Newtons), respectively, and $\tau_{iz} \, \forall \, i = 1, \ldots, 8$ is the torque generated by each thruster about the z-axis, in Newton-metres. Consider the inverse calculation to Eq. (B.2): for a given (desired) $F_x, F_y, \tau_z$, what are the required duty cycles for each thruster? Solving for $u$

$$u = H^+ \begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix}$$

(B.5)

where $H^+$ is the Moore-Penrose inverse, needed because $H$ is non-square.
\[ H^+ = H^T(HH^T)^{-1} \] (B.6)

Roughly half of the duty cycles calculated in Eq. (B.6) contain negative numbers. These are not realistic and are therefore discarded. To compensate for this rejection, non-discarded values are doubled, with the assumption that complementary thrusters (i.e., thrusters 1 & 5 or 3 & 7 on Fig. B.3) yield identical torque. This assumption is not valid, but the results are still reasonably close to correct. Future work should explore other control mixing strategies to improve this. Sigma-Delta modulation is a possible avenue [142], or an optimization routine to replace the pseudo-inverse that is subject to non-negative constraints.

To determine the values in the \( H \) matrix (i.e., each thruster’s force and moment), I ran a number of experiments where a desired force was specified. The resulting acceleration of the chaser is measured, and the applied force can be inferred according to \( F = ma \). The value of each thruster in the \( H \) matrix is tuned until the commanded force is reasonably close to the achieved force. The resulting nominal force in the \( x \) direction is 0.2196 N and the nominal force in the \( y \) direction is 0.2825 N. Since the \( x \)-axis thrusters had one additional 90-degree bend in the plumbing, they resulted in a lower thrust than the \( y \)-axis thrusters.

The torques delivered by each thruster, i.e., the \( \tau_{iz} \) values in Eq. (B.4), are calculated from the inferred thruster force and the measured moment arm between the thruster and the spacecraft’s centre of mass. The centre of mass of the spacecraft, found according to Sec. B.1.1, is used to find the moment arm for each thruster. The thruster-to-centre-of-mass calculation performed is:
Figure B.3: Thruster numbering, centre of mass offset, and variable definition

\[ D = \begin{cases} 
0.15 - d_1 - \Delta y & \text{for thruster 1} \\
-0.15 + d_2 - \Delta y & \text{for thruster 2} \\
0.15 - d_3 - \Delta x & \text{for thruster 3} \\
-0.15 + d_4 - \Delta x & \text{for thruster 4} \\
0.15 - d_5 + \Delta y & \text{for thruster 5} \\
-0.15 + d_6 + \Delta y & \text{for thruster 6} \\
0.15 - d_7 + \Delta x & \text{for thruster 7} \\
-0.15 + d_8 + \Delta x & \text{for thruster 8} 
\end{cases} \]  

(B.7)

where all values are defined in Fig. B.3. The resulting moment created by each thruster is

\[ \tau = FD \]  

(B.8)

where \( F \) is the force supplied by the thruster and \( D \) is its corresponding moment arm.

All the measured nominal values of the chaser spacecraft used in this research have an \( H \) of
which is valid for the base spacecraft without a robotic manipulator. With a robotic manipulator, as used in Chap. 5, the manipulator orientation changes the centre of mass location of the spacecraft, which changes the thruster moment arms and invalidates Eq. (B.9).

### B.2.1 Manipulator Effect on Centre of Mass Location

The centre of mass is of the chaser is affected by the state of its manipulator. Each link’s centre of mass with respect to the centre of mass of the chaser, $p_1, p_2, p_3$, can be calculated according to

\[
H = \begin{bmatrix}
-0.2196 & -0.2196 & 0 & 0 & 0.2196 & 0 & 0 \\
0 & 0 & 0.2825 & 0.2825 & 0 & 0 & -0.2825 & -0.2825 \\
0.0152 & -0.0124 & 0.0141 & -0.0230 & 0.0134 & -0.0150 & 0.0223 & -0.0147
\end{bmatrix}
\] (B.9)

where all variables have been defined in Fig. 5.3. The resulting shift in the centre of mass of the chaser due to its manipulator is
\[ \Delta C O M_{\text{arm}} = \begin{bmatrix} \Delta x_{\text{arm}} \\ \Delta y_{\text{arm}} \\ \Delta z_{\text{arm}} \end{bmatrix} \frac{ m_1 p_1 + m_2 p_2 + m_3 p_3 }{ m_0 + m_1 + m_2 + m_3 } \]  

(B.13)

The moment arm for each of the eight thrusters can be re-calculated using:

\[
D = \begin{cases} 
0.15 - d_1 - \Delta y - \Delta y_{\text{arm}} & \text{for thruster 1} \\
-0.15 + d_2 - \Delta y - \Delta y_{\text{arm}} & \text{for thruster 2} \\
0.15 - d_3 - \Delta x - \Delta x_{\text{arm}} & \text{for thruster 3} \\
-0.15 + d_4 - \Delta x - \Delta x_{\text{arm}} & \text{for thruster 4} \\
0.15 - d_5 + \Delta y + \Delta y_{\text{arm}} & \text{for thruster 5} \\
-0.15 + d_6 + \Delta y + \Delta y_{\text{arm}} & \text{for thruster 6} \\
0.15 - d_7 + \Delta x + \Delta x_{\text{arm}} & \text{for thruster 7} \\
-0.15 + d_8 + \Delta x + \Delta x_{\text{arm}} & \text{for thruster 8} 
\end{cases}
\]  

(B.14)

which must be evaluated at each time step to account for the arm motion. In other words, the manipulator motion affects the centre of mass location, which affects each thruster’s moment arm, which affects each thruster’s torque value listed in Eq. (B.4). Updating Eq. (B.4) at each time step leads to significantly improved performance when the robotic manipulator is present.

Despite these additions, excellent acceleration tracking (necessary to get the spacecraft to follow the desired acceleration signals by the deep guidance algorithm) was still not obtained due to the thrust decay.

### B.2.2 Thrust Decay Quantification

All eight air thrusters are supplied by a common air supply. The air flows from an on-board air tank, through a regulator, and to the air thrusters. Figure B.4 shows how air flows through the system. When one thruster is firing, it reduces the pressure available to all other thrusters. Therefore, with each additional thruster that is open, the thrust force produced by all others is reduced, which affects \( H \) and therefore renders the calculated duty cycles \( u \) invalid. This section performs a rough
Accurately estimating the decay of thrust as a function of other thruster use is a complex problem: the exact pressure drop needs to be calculated and the relationship between pressure and thrust needs to be known. Since high sample-rate pressure sensors are not installed on SPOT, another approach had to be taken. The average duty cycle represents how long each thruster is opened, on average. In theory, the pressure drop in the lines should be related to the amount of time the thrusters are opened for. The following procedure was used to measure this effect:

1. Command a desired thrust
2. Log the resulting average duty cycle
3. Observe the resulting acceleration
4. Calculate the true thrust achieved and compare it to the desired thrust
5. Repeat for different average duty cycles

with results shown in Fig. B.5. An equation for the decay is
Figure B.5: Experimentally-determined thrust decay

\[ f = \begin{cases} 
1 & \text{for } \text{mean}(u) < 0.3 \\
1.6 - 2\text{mean}(u) & \text{otherwise}
\end{cases} \quad (B.15) \]

where \( f \) represents the thrust factor. A factor of 1 means no decay and a factor of 0.3 means 30% of thrust remains. Therefore, the \( H \) matrix is modified to include the thrust decay factor

\[
H = f \begin{bmatrix}
F_{1x} & F_{2x} & F_{3x} & F_{4x} & F_{5x} & F_{6x} & F_{7x} & F_{8x} \\
F_{1y} & F_{2y} & F_{3y} & F_{4y} & F_{5y} & F_{6y} & F_{7y} & F_{8y} \\
\tau_{1z} & \tau_{2z} & \tau_{3z} & \tau_{4z} & \tau_{5z} & \tau_{6z} & \tau_{7z} & \tau_{8z}
\end{bmatrix} \quad (B.16)
\]

The use of the control mixer, with the improvements discussed in this previous subsections, is summarized next.

Control Mixer Summary

To use the improved control mixer, at each time step:

1. Evaluate the arm’s contribution (if the arm is used) to the centre of mass location using Eq. (B.13).

2. Calculate the moment arm for each thruster according to Eq. (B.14).
3. Assemble the preliminary control mixing matrix $H$ using Eq. (B.4). Nominal forces of $F_{ix} = 0.2196$ N and $F_{iy} = 0.2825$ N are used, and the torques are calculated according to Eq. (B.8).

4. For a desired thrust and torque, calculate the corresponding duty cycle using Eq. (B.5). Remove the negatives and double the positives in $u$.

5. Now, perform one iteration to improve the duty cycle calculation by accounting for the thrust drop. First, calculate the thrust factor $f$ using Eq. (B.15).


7. Recalculate the thruster duty cycles according to Eq. (B.5). Remove the negatives and double the positives in $u$.

This procedure calculates, with one iteration, the duty cycles for each of the eight thrusters to realize the desired forces and torques, while accounting for the manipulator motion and (roughly) accounting for the thrust drop according to the number of thrusters firing. This is an acceptable first pass at the problem, but it could certainly be improved. Suggested future work should improve Eq. (B.5) to avoid the assumption that complementary thrusters are identical (not valid because each thruster’s moment arm is different). Possible avenues are: an optimization approach to replace the pseudo-inverse with a non-negative number constraint; or through a new control mixing strategy detailed in [142].

More accurate measurements of the thrust should be performed—instead of inferring them from measured accelerations. Lastly, the thrust decay due to pressure drop should either be: 1) quantified much more accurately; or 2) eliminated through the construction of a physical plenum chamber (such that one thruster’s action does not affect the pressure delivered to the others).

B.3 Nvidia Jetson Interface

An Nvidia Jetson TX2 computer was used on-board the chaser platform to perform inference on the trained guidance policy neural network. The Jetson interfaces with
the Raspberry Pi-3 via ethernet. A custom device driver block was made in Simulink such that the communication can be seamless. A number of scripts have to be run to allow the Jetson to communicate properly with the Pi, but these details are omitted from this Appendix since they are covered in [143].
Appendix C

Neural Networks

This appendix details the structures of different neural networks and how they are trained. Neural networks are nonlinear function approximators that are often used in deep reinforcement learning.

C.1 Supervised Machine Learning

Machine learning, a class of artificial intelligence, is a collection of techniques that has recently taken off due to the improved computational abilities and the availability of large amounts of data. At a high level, machine learning is divided into two major classes: supervised learning and unsupervised learning. Supervised learning attempts to generate a nonlinear function that maps inputs to known outputs. When trained on a large number of input/output pairs, the learned nonlinear function can generalize to notice patterns in the input/output data. Once trained, the neural network can be presented with new input data that it has not seen during training and it can accurately predict the output. Unsupervised learning is used to decipher input data without any defined outputs. It is a less-developed field than supervised learning, but it can effectively find patterns in input data without any knowledge of what the data represents.

C.1.1 Feedforward Neural Networks

Feedforward Neural Networks (what is typically referred to with the term ‘neural network’) learn how to map a number of inputs to a number of outputs, through the training of many parameters [144]. A schematic of a feedforward neural network is shown in Fig. C.1. Inputs travel through the network, along the paths represented by arrows, and exit the network with an output $u$. Figure C.1 shows how a number of inputs, $x_1$, $x_2$, and $x_3$, are passed into the network. Each input is multiplied by a
parameter associated with each arrow the input takes. In each neuron, represented by circles, all the incoming values (and one additional learned value) are added together and then a nonlinear “activation function” is applied to the result. This value is then passed along to the next layer of neurons, which is multiplied by all the weights associated with the arrows, summed, and passed through another activation function, and so on. The resulting output, $u$, is the prediction of the neural network.

The two layers in the middle of the network, each with 3 neurons, as drawn in Fig. C.1, are called “hidden layers” because their outputs are not seen in the output of the entire network. Typically, more neurons and more layers will allow the network to resolve more complex input/output mappings. The more hidden layers a neural network has, the “deeper” it is said to be. However, deeper layers lead to many more parameters and therefore require more data and computational power to train and run.

The number of hidden layers and number of neurons in each layer are chosen by the user. Any parameters that are chosen by the network designer are called “hyperparameters,” as compared to the learned “parameters” (the numbers associated with each of the arrows that the signal gets multiplied by). Training the network is the act of presenting the network with many input/output pairs that allow the
network to search for near-optimal parameters that minimize the error between its predictions and the desired outputs from the training data. The *backpropagation* algorithm is used to compute how each parameters should change [145]. To train the network: 1) the input data are forward-propagated through the network to obtain a set of predictions; 2) the errors between the predictions and the desired outputs are calculated; 3) the errors are backpropagated through the network in order to calculate the gradients of the error with respect to the parameters; and 4) the parameters are modified slightly in the direction dictated by the gradient in order to adjust the parameters such that they will reduce the error. The “learning rate” is a hyperparameter multiplied by the gradients to determine how much each parameter should change with each update. Through repeating this process many times for many input/output pairs, the parameters of the network will eventually converge such that the errors between the predicted outputs and the desired outputs are minimized. An example of training a simple neural network is presented in Sec. C.2.

### C.1.2 Convolutional Neural Networks

If an image were fed into a feedforward neural network, where each pixel was one input to the network, the neural network could have millions of inputs. This would lead to billions of parameters that would have to be updated simultaneously, which is infeasible for modern computers (at the time of writing). This is the motivation for convolutional neural networks (CNNs), which are predominantly used to process images with many-fewer parameters [146]. A convolutional layer convolves many smaller images (that have parameters of their own) with the input image. The result of this convolution is another image, which is then processed by another convolutional layer, and so on. Each convolutional layer shrinks the dimensions of the image such that eventually, the image becomes small enough to be fed to a FFNN which eventually leads to an output, $u$, as described in Sec. C.1.1. Convolutional networks greatly reduce the number of parameters needed to process an image compared to simply passing the entire image through a FFNN immediately. CNNs have been able to achieve superhuman image recognition abilities [147].
Table C.1: AND gate truth table.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure C.2: One-neuron example neural network used to model an AND gate.

C.1.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are like FFNNs, but with feedback among neurons. In contrast to a FFNN, this causes the output, $u$, to depend on the current input and past inputs of the RNN [147]. Recurrent neural networks excel at processing sequential data, such as: speech-to-text, language translation, and real-time signal processing.

C.2 Neural Network Training Example

This section presents an example of the backpropagation algorithm for demonstration purposes. Suppose we are trying to train a one-neuron feedforward neural network to model a logical AND gate. The truth table for the AND gate is shown in Table C.1. The inputs are $x_1$ and $x_2$, and the output $Y$ is the desired output. Using supervised learning, we will train a simple neural network to mimic an AND gate as closely as possible. The neural network is shown in Fig. C.2.

The two inputs, $x_1$ and $x_2$, are each multiplied by a weight associated with each
Figure C.3: Computation graph for neural network used in the AND gate example.

Table C.2: Outputs of initialized neural network.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
<th>Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$^\hat{Y}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.62</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.71</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.48</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.57</td>
</tr>
</tbody>
</table>

arrow. The results then flow into the neuron (circle) where the incoming signals are summed. A bias is added to the result, and then the result is passed through a nonlinear “activation” function. In this case, the activation function was chosen to be the sigmoid function:

$$^\hat{Y}(Z) = \frac{1}{1 + e^{-Z}} \quad (C.1)$$

The sigmoid function was chosen because it forces the output to be between 0 and 1, which is the range of the AND gate.

An alternative to presenting a neural network as done in Fig. C.2 is to use a computation graph. Computation graphs show how the signal flows through the neural network, and is useful for computing how the network should be trained, as explained in Sec. C.2.1. A computation graph of the neural network is shown in Fig. C.3. Here, $X \in \mathbb{R}^2$ is the input, $W \in \mathbb{R}^2$ are the weights, and $b \in \mathbb{R}$ is the bias. The intermediate variable, $Z$, represents the output of the multiplication and addition operation. The prediction of the network, $^\hat{Y}$ is obtained by passing $Z$ through the sigmoid function.

Randomly initializing the parameters to $W = [0.4, 0.6]^T$ and $b = 0.5$, we obtain the neural network predictions shown in Table C.2. We immediately see that the output values are not close to the truth values at all. To improve the accuracy, we must tune
the weights $W$ and bias $b$. We do not guess at how to change the weights. Instead, we calculate how the weights should be changed using backpropagation, explained in the following section.

C.2.1 Backpropagation

The backpropagation algorithm is a powerful technique that is at the root of all neural network training [145]. To improve the quality of the neural network, a measurement of the quality is needed. We call this measurement a loss function. For the current example, the selected loss function is:

$$\text{Loss} = -Y \ln(\hat{Y}) - (1 - Y) \ln(1 - \hat{Y})$$  \hspace{1cm} (C.2)

We immediately see that if the “truth” value is $Y = 0$, the loss function simplifies to:

$$\text{Loss} = -\ln(1 - \hat{Y})$$  \hspace{1cm} (C.3)

which is positive when $\hat{Y} \neq Y$. Similarly, when $Y = 1$, the $\text{Loss}$ function will increase the farther away $\hat{Y}$ is from $Y$—therefore, the loss function is minimized when the neural network outputs the correct answers. We can expand the computation graph to include the loss function, shown in Fig. C.4. Using Eq. (C.2), the loss for each data point can be calculated, shown in Table C.3. The goal is to reduce the loss by systematically adjusting the parameters $W$ and $b$. To do so, we must calculate the gradient of the loss with respect to the parameters. Specifically, we need:

$$\frac{\partial \text{Loss}}{\partial W} \text{ and } \frac{\partial \text{Loss}}{\partial b}$$  \hspace{1cm} (C.4)
Table C.3: Calculating loss on initialized neural network.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
<th>Truth</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$Y$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$0$</td>
<td>$1$</td>
<td>$0.71$</td>
<td>$0$</td>
</tr>
<tr>
<td>$1$</td>
<td>$0$</td>
<td>$0.48$</td>
<td>$0$</td>
</tr>
<tr>
<td>$1$</td>
<td>$1$</td>
<td>$0.57$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

Average Loss: $0.85$

To do so, we use the computation graph along with the chain rule. For $W$,

$$
\frac{\partial \text{Loss}}{\partial W} = \frac{\partial \text{Loss}}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial Z} \cdot \frac{\partial Z}{\partial W}
$$

(C.5)

of which each term can be computed. For $b$,

$$
\frac{\partial \text{Loss}}{\partial b} = \frac{\partial \text{Loss}}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial Z} \cdot \frac{\partial Z}{\partial b}
$$

(C.6)

For this specific neural network, each partial derivative is evaluated below:

$$
\frac{\partial \text{Loss}}{\partial \hat{Y}} = \frac{\partial}{\partial \hat{Y}} \left[ -Y \ln(\hat{Y}) - (1 - Y) \ln(1 - \hat{Y}) \right] = -\frac{Y}{\hat{Y}} + \frac{1 - Y}{1 - \hat{Y}}
$$

(C.7)

$$
\frac{\partial \hat{Y}}{\partial Z} = \frac{\partial}{\partial Z} \left[ \text{sigmoid}(Z) \right] = \frac{\partial}{\partial Z} \left[ \frac{1}{1 + e^{-Z}} \right] = \hat{Y}(1 - \hat{Y})
$$

(C.8)

$$
\frac{\partial Z}{\partial W} = \frac{\partial}{\partial W} \left[ W^T X + b \right] = X
$$

(C.9)

$$
\frac{\partial Z}{\partial b} = \frac{\partial}{\partial b} \left[ W^T X + b \right] = 1
$$

(C.10)

which leads to the following parameter gradients:

$$
\frac{\partial \text{Loss}}{\partial W} = \frac{\partial \text{Loss}}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial Z} \cdot \frac{\partial Z}{\partial W} = (\hat{Y} - Y)X
$$

(C.11)

$$
\frac{\partial \text{Loss}}{\partial b} = \frac{\partial \text{Loss}}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial Z} \cdot \frac{\partial Z}{\partial b} = \hat{Y} - Y
$$

(C.12)
Calculating the gradients using Eqs. (C.11) and (C.12), we obtain:

\[
\frac{\partial \text{Loss}}{\partial W} = (\hat{Y} - Y)X = \begin{pmatrix} 0.62 \\ 0.71 \\ 0.48 \\ 0.57 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.29 \\ 0.05 \end{pmatrix} \tag{C.13}
\]

\[
\frac{\partial \text{Loss}}{\partial b} = \hat{Y} - Y = 0.35 \tag{C.14}
\]

The gradients calculated above show how much a unit change in the parameter increases the loss. Therefore, in the pursuit of decreasing the loss, we change the weights in the opposite direction.

\[
W = W - \frac{\partial \text{Loss}}{\partial W} = \begin{pmatrix} 0.4 \\ -0.6 \end{pmatrix} - \begin{pmatrix} 0.29 \\ 0.05 \end{pmatrix} = \begin{pmatrix} 0.11 \\ -0.65 \end{pmatrix} \tag{C.15}
\]

\[
b = b - \frac{\partial \text{Loss}}{\partial b} = 0.5 - 0.35 = 0.15 \tag{C.16}
\]

Note that we subtracted the full gradient from the parameters. In practice, a “learning rate” is multiplied by the gradient, so that the parameters change much slower. Typically, the learning rate is on the order of 0.001; here, we have used a learning rate of 1. Now, the updated weights can be used in the neural network to make new predictions based on the data. The predictions of the neural network are shown in Table C.4.

The network still does not perform well, but the average loss has decreased. The learning process is now repeated. Namely:
Table C.5: Calculating predictions and loss after 100 training iterations.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
<th>Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$Y$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.003</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.12</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.12</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.6: Calculating predictions and loss after 1,000 training iterations.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
<th>Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$Y$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.000003</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Calculate network predictions.

2. Calculate parameter gradients.

3. Update parameters in the direction of decreasing the loss.

It is not necessary to calculate the loss after each iteration, but it is useful for visualizing the learning progress. After 100 iterations, the network has the performance shown in Table C.5. After 1,000 iterations, the performance is as shown in Table C.6. Here, training is deemed to be complete because the outputs $\hat{Y}$ closely match the desired outputs $Y$ and the average loss is low. The final parameters are:

$$W = \begin{bmatrix} 8.35 \\ 8.35 \end{bmatrix} \quad \text{and} \quad b = -12.59 \quad (C.17)$$

Therefore, the final, trained, neural network has the structure shown in Fig. C.5, and can be used to make predictions similar to a real AND gate. Since the output doesn’t perfectly match the AND gate, the output may need to be rounded.

To summarize the procedure of using neural networks to make predictions for a
Figure C.5: Trained neural network, representing an AND logic gate.

given problem:

1. Collect data that the neural network should be trained on. Once trained, the
   neural network can then make predictions based on new data.

2. Choose the structure of neural network and the activation functions. See
   Sec. C.3 for details.

3. Choose a loss function to quantify how well the network is performing.

4. Differentiate the loss function with respect to the trainable parameters, using
   the chain rule through the computation graph.

5. Randomly initialize parameters.

6. Train network until you are satisfied.

7. Use the trained network to make predictions.

This example was meant to show the basic operation of a neural network. There
are many techniques to improve the learning that are not discussed in this Appendix.

One final note: if neural networks are trained too long, they can “over-fit” the
training data, such that the loss function is evaluated to be very small, but the
neural network has not learned any meaning from the data. When an over-fitted
network is shown any new data, the network will make very poor predictions. It is
important for the network to “generalize” to the data. This means that the network
understands how the input data is mapped to the outputs, which will enable it to make good predictions on data unseen during training. Therefore, it is important to stop training the network before it begins to become over-fitted to the training data. Techniques to prevent over-fitting are called “regularization” techniques.

Neural networks contain many parameters, or weights, that are trained. There are many other parameters that affect how a neural network performs, though. These parameters are called “hyperparameters”, and are discussed in the following section.

C.3 Neural Network Hyperparameter Selection

Examples of hyper-parameters in neural networks include: number of layers, number of neurons in each layer, non-linear activation function in each neuron, loss function, learning rate, etc. They are named “hyperparameters” because their manual selection ultimately affects the trainable network parameters.

Choosing hyperparameters often requires intuition and experience - there is no set strategy on selecting them. Often, researchers will try many different combinations of hyperparameters and will see what works. Some rules of thumb:

- Larger networks can learn more complex mappings.
- The ReLU activation function appears to lead to the fastest learning, though this has not been proven.
- Larger neural networks require more training data before good performance is achieved. Larger networks are more susceptible to over-fitting.
- A small learning rate is typically more stable, but leads to slower learning.

For these reasons, it is often a good idea to train many networks with different hyperparameters and find what works best.

C.4 Neural Network Limitations

Neural networks, in certain cases, are excellent at making predictions based on data it is given. At times, a neural network can outperform human experts at certain tasks.
What neural networks don’t do, thus far, is provide insight for researchers about how the neural network is making its decisions. There are simply too many parameters in a large neural network for a human to be able to understand the decision making process of a neural network. This leads to problems regarding guaranteeing performance, because if we do not know how the neural network is making its decisions, it is difficult to know how robust their decision-making process is. A great deal of current research is attempting to gain more insights to the inner workings of trained neural networks [148, 149].

C.5 Machine Learning Frameworks

Calculating the derivatives required for backpropagation by hand is tedious, especially when the neural network is large. Several machine learning frameworks exist in a variety of programming languages, and many automate the backpropagation step. The researcher then only needs to create the computation graph and specify the loss function (among other hyperparameters), and the machine learning library will perform the backpropagation for you. This is possible because the framework can evaluate the partial derivative for each step in the computation graph, and therefore automatically evaluate the gradients. The author uses the open-sourced TensorFlow machine learning framework [92], written for Python, in their research.
Appendix D

Reinforcement Learning

Reinforcement Learning is a very general form of machine learning, that consists of an agent interacting with an environment. At each time step, the agent performs an action (the selection of which may be deterministic or stochastic) depending on its observation of the environment. The agent then executes the action in the environment which leads it to arrive at a new state, one time step later, based on the environment dynamics (which may be deterministic or stochastic). The environment also returns a scalar reward to the agent that reflects how favourable or unfavourable the current state and action are. The agent then attempts to learn what actions it should take in order to maximize the rewards received. The designer selects the reward function, and is how the designer communicates the desired behaviour to the reinforcement learning agent. The agent learns a “policy”, $\pi$, that determines what action to take based on the current state or observation. It is important to note that the agent does not have any knowledge or awareness of the task it is trying to learn. It simply attempts, through trial and error, to learn what actions to take that maximize its expected rewards. With recent advances in computing power, this trial and error process can be repeated many times until an expert policy has been discovered.

Different classes of reward functions are possible. Sparse reward functions are typically used in competitive games, where a reward of 1 is given if the agent wins the game, a reward of $-1$ is given if the agent looses the game, and a reward of 0 is given for each move. With sparse rewards, the agent has to play an entire game before it receives any reward, which leads to the credit assignment problem \cite{150}. The credit assignment problem is the difficulty in knowing which actions in the past contributed to the win and therefore should have their probability increased. This problem is typically overcome by letting agents play millions of games. Dense rewards signals, on the other hand, result in a non-zero reward being given to the agent at each time
step. An example is an agent learning to control a robot to run down a track, where the forward motion is rewarded at each time step [151]. However, dense reward signals are typically more difficult to design [152], and can introduce more human bias into the agent’s learned behaviour. The environment to which reinforcement learning is applied is typically formulated as a Markov Decision Process.

D.1 Markov Decision Processes and Deep Reinforcement Learning

A Markov Decision Process (MDP) describes a discrete-time sequence of events to which reinforcement learning can be applied. For a state \( x_n \in X \) at time step \( n \) upon which a chosen action \( a_n \in A \) is applied, a new state \( x_{n+1} \) is returned according to a state transition function \( P_n(x_n, x_{n+1}) \), which may be stochastic, along with a corresponding scalar reward \( r_n(x_n, a_n) \). The MDP must satisfy the Markov property, which requires the process be memoryless. In other words, the state \( x \) must contain all information required to make an optimal decision. In many cases, the true state is unknown and must be inferred from an observation of the state, \( o \in O \), in which case the process is named a Partially Observable Markov Decision Process (POMDP). In the context of robotics, the state of the dynamic system is \( x \), the equations of motion represent the state transition function (which may be deterministic), the chosen action is the control effort executed on the dynamics, and the reward is a designer-chosen function that corresponds to task completion and drives the learned behaviour. If camera images are used as inputs, they would represent an observation of the state from which the true state must be inferred.

There are three dominant areas of research for accomplishing the reinforcement learning goal of training a policy to maximize the expected rewards. Namely: model-based methods, model-free value function methods, and model-free policy gradient methods.

D.2 Model-based Methods

Model-based reinforcement learning methods work on the following principle: they attempt to learn a model of the environment, and then use that learned model to plan
which action to take [153–156]. A neural network is often used to approximate the environment dynamics and is trained with supervised learning, described in Sec. C.1. Similar to value function methods (discussed in Sec D.3.1), an explicit policy is not used in model-based reinforcement learning. Instead, once a reasonable model of the environment is learned, the model can be used to plan appropriate actions to take at each time step. Several planning methods are available, such as Monte Carlo Tree Search [27,157] or Model-Predictive Control [158]. Model-based reinforcement learning is typically more sample-efficient than policy gradient or value-based methods, meaning that less data (i.e., less episodes) are needed before reasonable performance is achieved [159,160]. However, this does not mean that model-based reinforcement learning always converges faster in terms of wall time—planning actions using the learned model can be very computationally expensive. Model-based reinforcement learning is applicable to real robot systems where one would like to minimize the amount of episodes that are required before reasonable performance is achieved.

D.3 Model-free Methods

In contrast to model-based methods, model-free methods do not learn an explicit model of the environment. Instead, they learn other mappings, either an accurate measure of the value of each state-action pair (called ‘value function methods’ and useful for discrete action spaces) or a reward-seeking policy (called ‘policy gradient’ methods, useful for continuous action spaces).

D.3.1 Value Function Methods

Value function methods attempt to learn a relationship between the current state, $x$, the action taken, $u$, and the total rewards received for the remainder of the episode, called the “Q-value”, $Q$. One episode refers to one simulation or one game played. In small environments with discrete states and actions, a lookup table can be used to log the Q-values of taking each action in each state. Then, for a given state, the action that has the corresponding highest Q-value is selected, according to:

$$\pi(x) = \arg\max_u (Q(x,u))$$  \hspace{1cm} (D.1)
A Q-value table becomes very large if either the states or the actions are highly dimensional, as is typical in robotics. When continuous states are used, a lookup table becomes infinitely large. To overcome this problem, a neural network that maps the state-action pair to the Q-value is learned, and this technique is named “Deep Q-learning”. The reader is referred to Appendix C if some background on neural networks is desired. The success of the Deep Q-learning algorithm lies in its ability to train the neural network that approximates the Q-value for all states and actions. Deep Q-Learning has had a number of recent successes [26,161,162]. A value function example is presented below.

Example: Action-value Algorithm

Imagine a lottery machine-style game, called the n-armed bandit. It consists of n levers which can be pulled, each which returns a reward. One lever can be pulled on each turn, and the reward given is sampled from the probability distribution associated with each lever. The statistics of each lever are unknown to the player; the player wants to maximize the rewards received. How should the player accomplish this? Pull each lever once and then continue to pull the most-profitable lever indefinitely afterwards? A reinforcement learning strategy for this game is called the action-value algorithm.

Imagine the true, but unknown, statistics for each of the 5 levers (in this example, \( n = 5 \)), are shown in Table D.1. Clearly, the best lever to pull, on average, is Lever 3, but the agent does not know this. Start by pulling a random lever: number 4. It returns a reward of 2.1. This information can be recorded in an action-value table, as shown in Table D.2. Now, try pulling each remaining lever once. This yields the rewards shown in Table D.3.

At this point, it appears that the most profitable lever is Lever 4. If we were to
Table D.2: Action-value table after 1 lever pull

<table>
<thead>
<tr>
<th></th>
<th>Lever 1</th>
<th>Lever 2</th>
<th>Lever 3</th>
<th>Lever 4</th>
<th>Lever 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Reward</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2.1</td>
<td>—</td>
</tr>
<tr>
<td>Number of pulls ($k_a$)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table D.3: Action-value table after 5 lever pulls

<table>
<thead>
<tr>
<th></th>
<th>Lever 1</th>
<th>Lever 2</th>
<th>Lever 3</th>
<th>Lever 4</th>
<th>Lever 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Reward</td>
<td>0.5</td>
<td>1</td>
<td>-1.5</td>
<td>2.1</td>
<td>2</td>
</tr>
<tr>
<td>Number of pulls ($k_a$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

begin to exploit this and pull Lever 4 indefinitely, the total rewards would not be maximized. Therefore, some tradeoff between exploration and exploitation is needed. A simple procedure is to use an $\epsilon$-greedy approach: for $\epsilon < 1$, a random lever is pulled with probability $\epsilon$ (representing the exploration) and the greedy action is taken with probability $1 - \epsilon$ (representing the exploitation).

Each time a lever is pulled, the corresponding entry in the action-value table is updated according to

$$Q(a) = \frac{r_1 + r_2 + r_3 + \cdots + r_{k_a}}{k_a}$$  \hspace{1cm} (D.2)

where $Q$ represents the action-value, $a$ represents the chosen lever, $r_1, r_2, r_3, \ldots, r_{k_a}$ represents the rewards received for each of the $k_a$ times lever $a$ was pulled. Eq. (D.2) calculates the average return from each lever as more lever pulls are performed.

Setting $\epsilon = 0.1$ and performing 100 lever pulls, the state-action table shown in Table D.4 is obtained. Notice how Lever 4 is converging on its true value (after being pulled so many times). The best lever to pull, Lever 3, has not been explored any more and therefore it appears to be a poor choice. As the strategy is continued for 10,000 pulls, the state-action table shown in Table D.5 is obtained.

Table D.4: Action-value table after 100 lever pulls

<table>
<thead>
<tr>
<th></th>
<th>Lever 1</th>
<th>Lever 2</th>
<th>Lever 3</th>
<th>Lever 4</th>
<th>Lever 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Reward</td>
<td>0.051</td>
<td>1.32</td>
<td>-1.5</td>
<td>2.92</td>
<td>-0.85</td>
</tr>
<tr>
<td>Number of pulls ($k_a$)</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>89</td>
<td>5</td>
</tr>
</tbody>
</table>
Table D.5: Action-value table after 10,000 lever pulls

<table>
<thead>
<tr>
<th></th>
<th>Lever 1</th>
<th>Lever 2</th>
<th>Lever 3</th>
<th>Lever 4</th>
<th>Lever 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Reward</td>
<td>-0.037</td>
<td>1.02</td>
<td>4.99</td>
<td>3.03</td>
<td>-2.79</td>
</tr>
<tr>
<td>Number of pulls ($k_a$)</td>
<td>220</td>
<td>194</td>
<td>8,846</td>
<td>520</td>
<td>220</td>
</tr>
</tbody>
</table>

At this point, the optimal action, Lever 3, is clear. This represents a very simple trial-and-error method for learning the best actions to take when playing an unknown game. Though these results seem trivial, the implications of a general learning strategy, like this action-value algorithm, are tremendous. The techniques presented here represent the basis for all of reinforcement learning—upon which more sophisticated algorithms are built.

Value function methods have traditionally been restricted to discrete action spaces, as it was thought that selecting a continuous action using Eq. (D.1) would require an optimization task at each timestep [31]. In 2016, however, Deep Q-Learning has been extended to continuous action spaces through policy gradient methods [163].

D.3.2 Policy Gradient Methods

Value function and model-based methods optimize their neural networks for accuracy of the Q-value and dynamics model, respectively, but do not directly train a policy that selects actions to yield rewards. Instead, a policy is inferred from the Q-value function or dynamics model. In contrast, policy gradient methods directly optimize a neural network policy that maximizes the rewards received.

The policy input is the state or observation of the system and the output is the action. Policy gradient methods initially randomize their policy parameters, which leads to random initial behaviour. Then, the policy is used in an environment where the state, action, next state, and reward data are collected. Using this data, it is possible to calculate how the policy should be changed in order to increase the total reward received. Policy gradient methods directly update the policy by estimating a gradient for the policy with respect to the rewards. The first policy gradient algorithm was published in 1992 and was named REINFORCE [164].

Typically, the policy, $\pi(u|x)$, dictates the probability of taking action $u$ given
the current state, $x$, and is therefore stochastic. Stochastic reinforcement learning policies are trained in environments where the average reward received over a number of episodes is measured and expected to increase during training. However, due to the stochasticity of the policy, the performance of the next episode is not predictable i.e., the agent selects actions according to a learned probability distribution at each timestep. The transition from a stochastic to a deterministic policy gradient algorithm is highly desirable for real-world applications, especially for spacecraft and robotic operations. A deterministic policy would allow for repeatable performance once the policy is trained. In 2014, a deterministic policy gradient algorithm was discovered [165].

Policy gradient methods are a good fit for robotics, in the sense that they directly optimize rewards, work in continuous environments, and can be deterministic. For these reasons, policy gradient methods are used in this work. Figure D.1 shows a block diagram of a policy gradient algorithm in the context of robotics.

The learned policy, represented by a neural network, accepts the state as the input and outputs a control effort which travels to the dynamics block. The reward function receives the state and control effort used, and returns a reward to the policy depending on how favourable the state-action pair is. The policy is then updated in the direction of increased rewards.
In order to compare deep reinforcement learning algorithm performance, a set of standardized simulation environments are required. A few are currently available, discussed in the following section.

D.4 Benchmarking

With many researchers across the world working on improving the learning speed and abilities of different reinforcement learning approaches, standardized simulation environments to allow for meaningful comparison of their algorithms were needed. In 2013, the Arcade Learning Environment [71] was released, which was a collection of hundreds of Atari 2600 games in a format that can easily interface with learning agents. When testing an algorithm, one can easily run the same algorithm on many different games to test for generality of the learning algorithm. In 2016, OpenAI released Gym [166], an open-sourced implementation of many environments for researchers to test their reinforcement learning algorithms on. It included Atari games, discrete state and action control tasks, and continuous state and action control tasks. In 2018, DeepMind released their Control Suite as open-source, which contains exclusively continuous control tasks [167]. Both OpenAI Gym and DeepMind Control Suite use the MuJoCo physics engine to efficiently run their dynamics simulations [168].

D.5 Other Learning Methods

A number of other learning methods exist, briefly summarized here.

D.5.1 Imitation Learning

Imitation learning, also known as behaviour cloning, is a technique used to mimic an expert in performing a task. An expert performs a task in which all of the states and actions are logged. Then, a neural network is trained to map the states to the actions using supervised learning. In contrast to reinforcement learning, where the correct actions are not available and the agent must discover the correct actions for itself, imitation learning is provided the correct actions from data collected from an expert. Once trained, the network can then be used to generate appropriate actions, for a
given state, that rival the expert in performance. However, since the neural network will have small errors in its output as compared to the expert, the states encountered when imitation learning is used will eventually drift from the states encountered by the expert. Once the state space encountered is no longer representative of the state space area used for training, the neural network will output poor actions and the agent no longer performs well.

The DAgger (Data Augmentation) \cite{169} algorithm can be used to address this problem of state drift due to the nature of a neural network outputting small errors in the action space. The DAgger algorithm works as follows:

1. Perform imitation learning.

2. Run the agent that is performing imitation learning and log all of the states encountered.

3. Have an expert label all states encountered by the imitating agent, with the actions that the expert would have taken.

4. Augment the original dataset with this new dataset.

5. Repeat items 1 to 4 until the imitating agent performs sufficiently well.

In essence, the DAgger algorithm allows the imitating agent to perform its task and fail due to drift in its states. It then asks an expert to tell the agent which actions should have been taken in its drifted state. The agent is then retrained with this additional information until the agent sufficiently matches the expert’s abilities.

In certain tasks, asking an expert to label the appropriate actions to take in each state is very expensive. In other tasks, an expert is not available. Therefore, it is desirable for an agent to learn on its own instead of attempting to mimic an expert. This is another motivation for reinforcement learning.

D.5.2 Iterative Learning Control

Iterative Learning Control (ILC) is a feedforward control strategy that is most applicable when robotic systems need to repeat an identical task over a number of trials.
ILC works in a feedforward manner along with a conventional feedback controller. At each successive trial, the control effort is logged for use in the next trial. Then, in the next trial, the previous trial’s control effort is fed forward. The feedback controller therefore only has to correct for non-repeating disturbances or other small tracking errors [170].
Appendix E

Example: Reinforcement Learning on Two-link Manipulator

This chapter presents theory and preliminary results of using the Deep Deterministic Policy Gradient (DDPG) algorithm [90], a deep reinforcement learning algorithm, to guide and control a simulated robotic manipulator to place its end-effector on a desired location. This example was performed as a course project, and forms the basis upon which the main body of the thesis was built. Results show that the DDPG algorithm can learn this task. Additionally, preliminary results show that the algorithm requires less episodes to learn a guidance-only strategy rather than a guidance and control strategy.

E.1 Deep Reinforcement Learning Background

In reinforcement learning, an agent attempts to find the best strategy to solve a problem through selecting actions based on its observation, \( o \), of the system. In a partially-observed system, the observation may be camera or sensor inputs that may or may not be directly related to the state of the system. In a fully-observed system, the observation is equal to the true state, \( x \), of the system. Reinforcement learning requires the Markov property be satisfied [171]. That is, for a given environment, the next state \( x' \) only depends on the current state \( x \) and current action \( u \), along with the environment’s transition dynamics \( D \). If any knowledge of past states or actions is needed, the system is not Markovian. The notion of a good strategy is communicated to the agent through a scalar reward signal that is returned to the learning agent at each timestep. Using this reward signal, and through trial and error, the agent can slowly progress towards a near-optimal strategy that maximizes the rewards received. Formally, assuming a fully-observed system, the agent encounters state \( x \in \mathcal{X} \), in environment \( \mathcal{D} \), and performs an action, \( u \in \mathcal{U} \), where \( \mathcal{X} \) and \( \mathcal{U} \) are the state and action spaces, respectively, which may be discrete or continuous, and
$D$ are the dynamics of the system, which may be probabilistic or deterministic. The 
agent selects an action $u$ in state $x$, and after the environment $D$ is stepped forward 
one timestep, the agent finds itself in a new state, $x'$ and receives a reward, $r$.

The agent attempts to learn a policy, $\pi$, that selects an action based on the 
state or observation, that maximizes the expected rewards it will receive throughout 
the episode. Therefore, through the reward structure, certain behaviours can be 
incentivized. This is the central idea to reinforcement learning: the agent learns how 
to behave based solely on the rewards it receives and the environment in which it is 
acting. This enables reinforcement learning agents to learn interesting and unique 
strategies on their own, without significant human effort or bias. Reinforcement 
learning appears to be the only way, at present, for a designer to “ask” a robot to learn 
a task, and this communication is achieved through the reward signal. Furthermore, 
agents can be trained indefinitely and on many computers, which can lead to agents 
learning to perform at a superhuman level [26, 27, 72].

If the policy is deterministic, $u = \pi(x)$, and if the policy is stochastic, the action 
chosen is distributed according to the policy: $u \sim \pi(x, u) = P(u|x)$.

The goal of reinforcement learning is to maximize the expected rewards received 
over time. If the agent is operating in finite-length episodes, then the goal is to 
maximize the expected rewards, $J$, over each episode.

$$J = \mathbb{E}\left\{\sum_{t=0}^{T} r_t\right\}$$  \hspace{1cm} (E.1)

where $t$ is the timestep number, $T$ is the total number of timesteps, $r_t$ is the reward 
received at each timestep, and $\mathbb{E}$ denotes the expectation. If the agent is operating 
indefinitely, the goal is to maximize the expected rewards with a discount factor, 
$0 \leq \gamma < 1$, applied to future rewards.

$$J = \mathbb{E}\left\{\sum_{t=0}^{\infty} \gamma^t r_t\right\}$$  \hspace{1cm} (E.2)

Policies trained with a small $\gamma$ tend to be greedy towards short term rewards, but 
may perform poorly in the long term. When $\gamma$ nears 1, the policy nears an average 
reward criterion:
\[ J = \lim_{T \to \infty} \mathbb{E} \left\{ \frac{1}{T} \sum_{t=0}^{T} r_t \right\} \] (E.3)

in which case rewards at all points in time are equally weighted.

The agent encounters states but it is not told which action it should take. Therefore, the agent must explore the state space, action space, and reward structure to determine for itself what constitutes a good strategy. Therefore, the agent must occasionally ignore its current strategy, which will return a known reward, and instead explore in hope of discovering a better strategy that may return more rewards. This is known as the exploration-exploitation trade-off. Typically, the agent’s exploration rate is decreased as training progresses. When training is complete, the agent no longer explores and instead performs greedily. When the state and/or action space are highly dimensional, exploring all the possible state and action combinations is prohibitively expensive. This challenge is known as the curse of dimensionality [172].

Typically, the agent operates during a training phase, where the agent executes the current policy and collects state, action, next state, and reward data. These data are used to improve the policy until satisfied. Then, the policy is deployed, where it is simply used with no further training. On-policy methods require that the training data be collected from the current version of the policy, whereas off-policy methods allow for using training data collected from old policies as well.

Considering the above, there are certain important considerations when applying deep reinforcement learning to a robotic scenario. A policy gradient algorithm that is deterministic, continuous, and off-policy appears to be the best fit. Policy gradient algorithms directly optimize for increased rewards, driving the policy to be as effective as possible at achieving the goal specified by the reward function. A deterministic policy creates a reliable policy, which, in the case of robotics, is desirable. A continuous policy is important as robotic applications typically have actuators which may have near-continuous control abilities. An off-policy algorithm can use simulated data collected from previous versions of the policy to use in training the current version of the policy. An off-policy algorithm is said to be more “sample-efficient”, meaning that it requires less data to be collected, and therefore less episodes need to be run.
for an equal amount of learning [173].

Reinforcement learning becomes “deep” reinforcement learning when neural networks are involved. Neural networks have been proven to be universal function approximators [70], and therefore can be useful for representing complex policies. If desired, the reader is referred to Appendix C to learn more about neural networks. After a neural network is trained, it can be deployed and behaves like a function, with inputs and outputs.

The paper titled “Deterministic Policy Gradient Algorithms” [165], released in 2014, shows the first policy gradient algorithm that is continuous, deterministic, and off-policy. This work was extended in 2016 to use neural networks to represent the policy, and is known as the “Deep Deterministic Policy Gradient” (DDPG) algorithm [90]. The DDPG algorithm is very relevant to the field of robotics, and is therefore the algorithm used to obtain the preliminary results of this proposal. The DDPG algorithm is detailed below.

### E.2 DDPG Algorithm

Deep Deterministic Policy Gradient (DDPG) algorithm was developed by Lillicrap et al. in 2016 [90]. It is an actor-critic algorithm which means that two neural networks work together to achieve the desired learning. The actor is the policy neural network, $\pi_\theta(x)$, with parameters $\theta$, and the critic, $Q_\phi(x, u)$, is a second neural network, with parameters $\phi$, that evaluates the decisions of the policy. The critic is a neural network whose inputs are the state and control effort, and whose output is the predicted future reward, similar to the value function methods discussed in Sec. D.3.1. In other words, for a given state and control effort, the critic predicts the total amount of reward expected for the remainder of the simulation, i.e., the Q-value. A block scheme diagram of the DDPG algorithm, in the context of control theory, is shown in Fig. E.1. The critic network can be trained using supervised learning to minimize the loss function shown in Eq. (E.4)

$$L = \frac{1}{N} \sum_{i=1}^{N} (Q_\phi(x_i, u_i) - \dot{Q}_i)^2$$ (E.4)
for a batch of \( N \) data points. The target values, \( \hat{Q}_i \), are:

\[
\hat{Q}_i = r(x_i, u_i) + \gamma Q_\phi(x'_i, \pi_\theta(x'_i))
\]  

(E.5)

where \( \gamma \) is a factor that discounts future rewards. The closer \( \gamma \) is to 1, the longer into the future the critic considers. A small \( \gamma \) leads to greedy actions that may not yield the most reward in the long run. Equation (E.5) calculates the expected rewards for the remainder of the episode to be the reward received at the current timestep, \( r_t \), plus the expected rewards for the remainder of the episode received from the next state \( x'_i \).

Since neural networks are differentiable, the gradient of the critic with respect to the control effort, \( \nabla_u Q_\phi(x, u) \), can be calculated. The policy network, \( \pi_\theta \), is also differentiable. Taking the gradient of the policy with respect to the network weights, \( \nabla_\theta \pi_\theta(x) \), we can now obtain the relationship:

\[
\nabla_\theta J = \nabla_u Q_\phi(x, u) \nabla_\theta \pi_\theta(x)
\]  

(E.6)
where
\[
J = \mathbb{E} \left\{ \sum_{t=0}^{T} r(x_t, u_t) \right\}
\]  \hspace{1cm} (E.7)
and \( \nabla_{\theta} J \) is the gradient of the expected rewards with respect to the policy weights. These gradients can be used to step the policy weights in the direction of increasing expected reward. Over time, the policy will improve in the sense that it will choose actions that lead to higher expected rewards, i.e., the total rewards received as a function of training episode should increase.

Since the critic’s target values, calculated in Eq. (E.5), depend on the current critic and the current policy, they are susceptible to variance in the critic and policy neural networks during training. As an empirical trick, a copy of the critic and policy networks are used in Eq. (E.5) whose parameters are a moving average of the primary policy and critic networks. This trick has been shown to stabilize, but slow, the learning [90].

The policy and critic networks are trained from data generated from episodes where the policy is used as a combined guidance and control law, i.e., its input is the current state and its output is the control effort. Supervised learning assumes that the training data are independent and identically distributed. However, training the neural network with sequential data generated from the dynamics episode is not independent, which will lead to poor policy network generalization. To address this issue, a “replay buffer” is used. The replay buffer, successfully implemented in previous work [26,90], stores data from each simulated timestep in the episode. The data stored are: \( x_t, u_t, x'_t, \) and \( r_t \) at each timestep. Then, during training, a randomly sampled batch of data is taken from the replay buffer. This leads to the training data being uncorrelated which improves the training process.

The DDPG algorithm is summarized in Algorithm 2.

Then, once training is complete, the policy \( u = \pi_{\theta}(x) \) can be used as a standalone combined guidance and control law. Occasionally during training, the policy performance is evaluated. When this occurs, training is paused, and an episode is run without the exploration noise added to the control effort (i.e., \( N_t = 0 \)).

To become familiar with the DDPG algorithm above, it was implemented for a
Randomly initialize policy network, $\pi_\theta(x)$, and critic network, $Q_\phi(x, u)$

Initialize the target networks, $\pi'_\theta(x) \leftarrow \pi_\theta(x)$ and $Q'_\phi(x, u) \leftarrow Q_\phi(x, u)$

Initialize replay buffer to a chosen size

for episode $1 : M$ do
  Generate initial conditions for the dynamics $D$
  for timestep $t = 1 : T$ do
    Select control effort from the policy and add noise to encourage exploration
    $u_t = \pi_\theta(x_t) + \mathcal{N}_t$
    Execute control effort and step environment dynamics one timestep
    Store $(x_t, u_t, x'_t, r_t)$ in the replay buffer
    Randomly sample a batch of $N$ data points from the replay buffer
    Obtain critic target values: $\hat{Q}_i = r_i + \gamma Q'_\phi(x'_i, \pi'_\theta(x'_i))$.
    Update critic by applying one pass of backpropagation to minimize the loss:
    $L = \frac{1}{N} \sum_{i=1}^{N} (Q_\phi(x_i, \pi_\theta(x_i) - \hat{Q}_i)^2$ on the $N$ sampled data points with learning rate $\Delta_c$
    Update the policy network using the sampled policy gradient
    $\nabla_\theta J \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_u Q_\phi(x_i, u_i) \nabla_\theta \pi_\theta(x_i)$ with learning rate $\Delta_p$.
    Update the copied policy and critic networks: $\pi'_\theta \leftarrow \Delta_t \pi_\theta + (1 - \Delta_t) \pi'_\theta$ and $Q'_\phi \leftarrow \Delta_t Q_\phi + (1 - \Delta_t) Q'_\phi$ where $\Delta_t \ll 1$
  end for
end for

Algorithm 2: DDPG algorithm
planar, two-link robotic manipulator. The manipulator dynamics are discussed in the following section.

E.3 Dynamics

The dynamic system considered for this application example is a planar two-link robotic manipulator, shown in Fig. E.2, where \( q_1 \) and \( q_2 \) are the angles of the links and \( \tau_1 \) and \( \tau_2 \) are the torques applied to the joints. A Lagrange formulation is presented to obtain the nonlinear system dynamics in the form:

\[
\dot{x} = f(x, u). \tag{E.8}
\]

Assuming each link has identical mass \( m \) and length \( l \), the total kinetic energy of the system, \( T \), is:

\[
T = \frac{2ml^2q_1^2}{3} + \frac{ml^2(q_1 + \dot{q}_2)^2}{6} + \frac{ml^2\dot{q}_1 \cos(q_2)(\dot{q}_1 + \dot{q}_2)}{2}. \tag{E.9}
\]

The potential energy, \( V = 0 \) due to the absence of gravity and the assumption of rigid links and joints. Therefore, the equation of motion for the first generalized coordinate, \( q_1 \), is:

\[
\frac{d}{dt} \left( \frac{\partial T}{\partial \dot{q}_1} \right) - \frac{\partial T}{\partial q_1} + \frac{\partial V}{\partial q_1} = F_q \tag{E.10}
\]

\[
\left[ \frac{5ml^2}{3} + ml^2 \cos(q_2) \right] \ddot{q}_1 + \left[ \frac{ml^2}{3} + \frac{ml^2 \cos(q_2)}{2} \right] \ddot{q}_2 - \frac{ml^2 \sin(q_2)\dot{q}_2(2\dot{q}_1 + \dot{q}_2)}{2} = \tau_1 \tag{E.11}
\]
and, for the second generalized coordinate, $q_2$:

\[
\frac{d}{dt} \left( \frac{\partial T}{\partial \dot{q}_2} \right) - \frac{\partial T}{\partial q_2} + \frac{\partial V}{\partial q_2} = F_{q_2} \tag{E.12}
\]

\[
\begin{bmatrix}
\frac{ml^2}{3} + \frac{ml^2 \cos(q_2)}{2} \\
\frac{ml^2}{3} + \frac{ml^2 \cos(q_2)}{2}
\end{bmatrix}
\begin{bmatrix}
\ddot{q}_1 \\
\ddot{q}_2
\end{bmatrix}
+ 
\begin{bmatrix}
\frac{ml^2}{3} \\
\frac{ml^2}{3}
\end{bmatrix}
\begin{bmatrix}
\dot{q}_1 \\
\dot{q}_2
\end{bmatrix}
+ 
\frac{ml^2 \sin(q_2)}{2} \ddot{q}_1^2 = \tau_2. \tag{E.13}
\]

Assembling the nonlinear equations of motion into the standard form:

\[
M(q) \ddot{q} + C(q, \dot{q}) \dot{q} + K(q) = \tau \tag{E.14}
\]

yields

\[
\begin{bmatrix}
\frac{5ml^2}{3} + ml^2 \cos(q_2) & \frac{ml^2}{3} + \frac{ml^2 \cos(q_2)}{2} \\
\frac{ml^2}{3} + \frac{ml^2 \cos(q_2)}{2} & -\frac{ml^2 \sin(q_2) q_2}{2} - \frac{ml^2 \sin(q_2)(\dot{q}_1 + \dot{q}_2)}{2} \\
\frac{ml^2 \sin(q_2)}{2} & \frac{ml^2 \sin(q_2)}{2} \dot{q}_1
\end{bmatrix}
\begin{bmatrix}
\dot{q}_1 \\
\dot{q}_2
\end{bmatrix}
= 
\begin{bmatrix}
\tau_1 \\
\tau_2
\end{bmatrix} \tag{E.15}
\]

where $K(q) = 0$. Obtaining first order equations of motion, the state vector is selected to be:

\[
x = 
\begin{bmatrix}
q_1 \\
q_2 \\
\dot{q}_1 \\
\dot{q}_2
\end{bmatrix}
= 
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{bmatrix} \tag{E.16}
\]

\[
\dot{x} = 
\begin{bmatrix}
x_3 \\
x_4
\end{bmatrix}
= 
M(x)^{-1} \left[ \pi_\theta(x) - C(x, \dot{x}) \right] \tag{E.17}
\]

where the policy calculates the torques applied to the joints:

\[
\begin{bmatrix}
\tau_1 \\
\tau_2
\end{bmatrix} = \pi_\theta(x) \tag{E.18}
\]

With the dynamics resolved, the reward function must now be defined.
E.4 Reward Function

In order for the controller to learn the desired behaviour, a scalar reward function has to be returned at each timestep corresponding to how well the agent is performing. This section discusses the desired behaviour and the associated reward function.

The goal of this preliminary work is to learn a neural network guidance and control strategy that can place the end-effector of a robotic manipulator on a desired point in the manipulator’s workspace. Therefore, to incentivize the desired behaviour, a reward of 1 unit per second is returned to the agent at each timestep that the end-effector is perfectly at the target location. The reward is exponentially decayed as the end-effector moves away from the desired location, through:

\[ r(x, u) = e^{-\|p(x) - p_d\|^2} \]  \hspace{1cm} (E.19)

where \( p \) is the current location of the end-effector, which is a function of the state through:

\[ p(x) = \begin{bmatrix} l \cos(q_1) + l \cos(q_1 + q_2) \\ l \sin(q_1) + l \sin(q_1 + q_2) \end{bmatrix} \]  \hspace{1cm} (E.20)

and the desired location, \( p_d \), is set manually, as described in Section E.5. Equation (E.19) is an example of reward shaping, where a nonzero reward is given at all timesteps. This reward structure guides the agent towards the desired state. Sparse rewards, on the other hand, are when a reward of 0 is given at all timesteps except when the agent achieves the goal—when a reward of 1 is given. Sparse rewards lead to considerably longer training times before a good policy is achieved, as with this reward structure the agent must randomly stumble across the state that produces nonzero rewards [152]. However, sparse rewards are easier for the designer to implement.

The following section repeatedly simulates the dynamics presented in this section while the DDPG algorithm presented in Sec. E.2 attempts to learn an appropriate guidance and control strategy.
Table E.1: Manipulator physical parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$, kg</td>
<td>10</td>
</tr>
<tr>
<td>$l$, m</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_{\text{max}}$, Nm</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table E.2: Reinforcement learning hyperparameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta_p$</td>
<td>0.001</td>
</tr>
<tr>
<td>$\Delta_c$</td>
<td>0.0001</td>
</tr>
<tr>
<td>$\Delta_t$</td>
<td>0.001</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.99</td>
</tr>
<tr>
<td>$n_{\text{layers}}$</td>
<td>2</td>
</tr>
<tr>
<td>$n_{\text{neurons}}$</td>
<td>300</td>
</tr>
</tbody>
</table>

E.5 Simulations

The nonlinear equations of motion are simulated in Python with physical parameters as shown in Table E.1. The learning rate parameters for the DDPG algorithm were tuned by performing a grid search and selecting the parameters that led to the best results, which are shown in Table E.2. The number of layers in the actor and critic neural networks is $n_{\text{layers}}$, and the number of neurons in each layer is $n_{\text{neurons}}$. This leads to the actor and critic each having 92,400 trainable parameters that the DDPG algorithm seeks to adjust. Similarly to [90], an Ornstein-Uhlenbeck process [174] with $\sigma = 0.04$ and $\theta = 0.15$ was used to generate noise, $\mathcal{N}_t$, that is added to the policy actions to force exploration of the state space during training. When the policy is tested to see how well it performs, this noise is no longer added. The maximum torque is capped at $\tau_{\text{max}}$. The replay buffer is initialized to hold 800,000 timesteps worth of data before old data is discarded.

All neural network weights, for both the actor and critic, are initialized randomly and then many episodes are run while the DDPG agent learns a policy to satisfy the objective, dictated by the reward function presented in Sec. E.4. Each individual episode is simulated for 40 seconds with a timestep of 0.1 seconds using an Adams
Table E.3: Stationary target initial conditions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$, rad</td>
<td>$\frac{\pi}{2}$</td>
</tr>
<tr>
<td>$q_2$, rad</td>
<td>$-\frac{\pi}{2}$</td>
</tr>
<tr>
<td>$\dot{q}_1$, rad/s</td>
<td>0</td>
</tr>
<tr>
<td>$\dot{q}_2$, rad/s</td>
<td>0</td>
</tr>
<tr>
<td>$p_d$, m</td>
<td>$[-1, 0]$</td>
</tr>
</tbody>
</table>

Figure E.3: Agent performance during training.

integration method. At the beginning of each episode, the state is reset to the initial conditions shown in Table E.3, which corresponds to the state of the manipulator as drawn in Fig. E.2. The desired position is appended to the state before the state is fed to the policy and the critic, such that the agent is aware of the desired location.

The ability of the policy neural network to learn a suitable controller can be quantified through the cumulative rewards received throughout one episode. Since the reward function yields 1 reward per second that the end-effector remains on the target location, the maximum possible reward obtained over the 40 second simulation is 40. In practice, the achieved reward will be less than 40 because it will take time for the end-effector to move to the desired location.

A plot of the learning is shown in Fig. E.3, that took 28 hours of training time to
obtain. As the system initially behaves randomly, the system stumbles into desirable states by chance. The DDPG algorithm then learns which actions led to favourable states and incorporates that information into the policy neural network. The learning curve shows how the performance of the policy sharply increases during the first 1000 episodes until it reaches a plateau. After which, it is difficult to make any further improvements as the policy is already near-optimal. A short video of the learned policy can be viewed online at https://youtu.be/VY2sGj1Uiuo. A policy that brings the manipulator from the initial conditions to the desired location in Table E.3 was successfully learned. However, the learned policy is ineffective at bringing the end-effector to any other location. This is because the agent has only been trained on bringing the end-effector to one specific point. The policy has not been forced to generalize to moving the end-effector to any point. The following scenario attempts to learn a more general policy that can bring the end-effector to any desired location.

A second scenario was run where the initial conditions of the manipulator and the target location were randomized on each trial. This forces the agent to learn a more general policy that can bring the end-effector from any state to any other state within its operating envelope. Unfortunately, with the computing power available to the author, a good policy was not achieved in 75 hours of training. Possibly having more computation power or time would lead to a better policy. A learning plot is shown in Fig. E.4, where the “Random” line represents the scenario under consideration and the “Static” line represents the first scenario where the initial conditions and desired location were constant for each episode, and is included for comparison purposes. The randomized initial conditions learning curve does not reach the performance of the static initial conditions curve presented in the first scenario, even when more training episodes are used.

Preliminary tests of using deep reinforcement learning as a guidance-only strategy were performed next. A policy of the form:

\[
\begin{bmatrix}
\dot{q}_1 \\
\dot{q}_2
\end{bmatrix} = \pi_g(x)
\]  

(E.21)

was used. This policy outputs the desired angular rate of the manipulator joints,
instead of the torque, as a function of the state. This represents using the policy in a guidance-only approach. In the future, an adaptive controller could be used to apply the torques to each joint such that the angular rate of each link tracks the angular rate dictated by the policy. A perfect controller is assumed to exist, in this preliminary work, such that the dynamics simplify to:

\[
\dot{x} = \begin{bmatrix}
\pi_\theta(x) \\
0 \\
0
\end{bmatrix}
\]  \hfill (E.22)

The initial conditions of the manipulator and the desired end-effector location were randomized for each simulation. A maximum angular rate of 0.1 rad/s is imposed. Using the DDPG learning algorithm with these simplified environment dynamics in Eq. (E.22) allowed the DDPG algorithm to learn a reasonable guidance strategy for placing the end-effector at any location in the manipulator workspace.

A reward vs episode plot during the training is shown in Fig. E.5, and is the result of 11 hours of training time. The “Simplified” line represents the guidance-only strategy presented in this scenario. The “Static” line represents the first scenario.
Figure E.5: Training progress with simplified manipulator dynamics.

that was presented, where the initial conditions and target location were fixed on each training episode, and is included for comparison purposes. It is evident that the learning curve for the guidance-only strategy is less smooth than the static scenario. This is due to the randomized initial conditions of each simulation, which leads some initial conditions to be significantly easier than others (i.e., their starting location was closer to the desired location). However, the average reward appears to converge to near 30—a similar value to the first scenario where learning was successful. As confirmed by watching the time domain responses, available online at https://youtu.be/tQUOCybWOHg, a policy for guidance was successfully learned.

The guidance-only policy with randomized initial conditions was successfully learned, whereas the guidance-and-control policy attempting to learn the same task failed. This result indicates that a good deal of training effort is used to learn the low-level control aspect of the robotic arm. Therefore, if modern control theory, such as direct adaptive control, could be used to control the robot, along with using reinforcement learning as the guidance strategy, a very effective novel guidance and control strategy may be possible. This is the central idea of the author’s thesis. That is, proposing a novel guidance and control strategy that uses reinforcement learning
and adaptive control to guide and control robots. In addition, once the agent is trained in simulation, it is hypothesized that the use of adaptive control will mitigate some of the problems associated with deploying the trained system on a real robot.

The author acknowledges that this idea strays from a core idea of deep reinforcement learning. That is, to create a general learner that can map directly from inputs to controls, for any system, using only the reward signal as a guide. However, in the pursuit of being able to realistically apply deep reinforcement learning to real robots, the modifications presented in this proposal may be a necessary trade-off.

Another idea research idea is to improve the robustness of reinforcement learning to parameter uncertainty. Preliminary results of this work are presented below.

### E.6 Robustness

Robustness of reinforcement learning policies is an active research area. One idea for doing so is to randomly adjust the dynamics parameters in simulation for each episode, such that the policy needs to generalize across a range of plant parameters. This is a similar idea to when the initial conditions and target location were randomized for each episode in order to force the policy to generalize across initial conditions and target locations. It is hoped that this will help the policy perform better in a real-world scenario where the dynamics experienced in the real world may not have been seen in simulation. This section presents very preliminary results on this idea.

To test out one idea of creating a parameter-robust reinforcement learning policy, two policies are trained and compared:

1. One policy is trained with constant dynamics parameters for each episode.
2. One policy is trained while the dynamics parameters are varied for each episode.

Each trained network is then deployed in a scenario where the dynamics parameters are varied, and their performance is compared.

The same robotic manipulator and DDPG algorithm simulated in Sec. E.5 was used. The initial conditions and target location were constant for each simulation, as shown in Table E.3. Two policies were trained: one where the link length during each episode of training was constant at 1 m, and a separate policy that was trained where
the link length was varied randomly and uniformly across the range $[1, 1.4]$ m. All other parameters were constant and identical to the simulations presented in Sec. E.5. Both policies were trained for 4000 episodes.

Results are shown in Fig. E.6. Here, as expected, the “non-robust” controller, who was trained exclusively with a link length of 1 m, performs slightly better than the “robust” policy, who was trained with varying link lengths, when both policies are evaluated on a system with a link length of 1 m. This is expected because the non-robust policy has seen the 1 m long links repeatedly in training, where the robust policy has not. As the link length is increased past 1 m, the non-robust policy degrades in performance faster than the robust policy. Therefore, if deployed on a real robot, the robust policy would likely perform better than the non-robust one if the true link length was greater than 1 metre. Surprisingly, the non-robust policy performed better than the robust policy at link lengths less than 1 metre. Further analysis is needed to understand this phenomenon.

These preliminary results give some merit to the idea of training policies with varying plant parameters, to force them to generalize across a range of dynamic situations they may encounter. It is acknowledged, though, that a policy that is
robust to parameter uncertainties may not perform as well as a policy that was trained exclusively on the true link length. However, this tradeoff may be necessary for deploying policies in the real world when dynamics parameters may not be accurately known.

E.7 Conclusion

This Appendix presented the theory of deep reinforcement learning, and detailed the specifics of the Deep Deterministic Policy Gradient (DDPG) algorithm, a policy gradient algorithm that is deterministic, continuous, and off-policy, which is highly suitable for robotic applications.

Preliminary simulations of the DDPG algorithm are presented for a two-link robotic manipulator, which showed how the algorithm can learn to control a manipulator without any plant knowledge, and only using a reward signal indicating how well it is performing. The first step of the idea of using reinforcement learning as a guidance strategy was tested. A guidance-only reinforcement learning strategy was able to learn a suitable guidance law in fewer episodes than the guidance-and-control reinforcement learning agent, when trained on the same task. In the future, direct adaptive control could be combined with the deep reinforcement learning guidance strategy.

Finally, the idea of increasing the robustness of the algorithm by using domain randomization (i.e., changing dynamics parameters during each episode of training) was tested, and results showed that the idea may indeed be feasible.
Biographical Sketch

KIRK HOVELL is a space and artificial intelligence researcher who loves cats and canoe trips. He received his B.Eng and M.A.Sc. degrees in Aerospace Engineering from Carleton University in 2015 and 2017, respectively, where he received Senate Medals for Outstanding Academic Achievement for both of these degrees. He has received funding from the Natural Sciences and Engineering Research Council of Canada (NSERC) at the Undergraduate, Master, and Doctorate levels, in addition to several other scholarships. Kirk designed and built a significant portion of the Spacecraft Proximity Operations Testbed at Carleton University, a world-class spacecraft gravity-offset facility, from 2015 to 2017. He used this facility to publish nine papers before defending his PhD thesis (with two additional papers under review); his papers have already been cited over 100 times. He won the Best Presentation in Session Award at the 2017 AIAA Guidance, Navigation, and Control Conference, as well as being a Graduate Student Paper Competition Finalist at the same conference in 2020. He has been twice-nominated as an Outstanding Teaching Assistant by his students.