A Reinforcement Learning Approach to Optimizing Autonomous Kite-Powered Vessel Control

A Novel Approach

By

JOSHUA CAREY

Department of Computer Science
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Engineering.

SEPTEMBER 2023

Word count: ten thousand and four

Abstract

Here goes the abstract

Dedication and acknowledgements

Here goes the dedication.

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE: .....

Table of Contents

List of Tables

Table **Page**

List of Figures

Figure **Page**

CHAPTER 1

INTRODUCTION AND MOTIVATION

Maritime travel has been a cornerstone of human civilization, facilitating the exchange of goods, ideas, and cultures across vast expanses of water. The annals of history are replete with instances of seafaring civilizations harnessing the power of wind to propel their vessels across the oceans. It is posited that ancient Neanderthals embarked on maritime voyages in the southern Ionian Islands between 110 to 35ka BP [1]. The quintessence of maritime travel has predominantly been wind-powered sails, which remained unchallenged until the industrial revolution ushered in the era of fuel-powered engines.

The art and science of sailing have evolved significantly over millennia, from rudimentary rafts and canoes to sophisticated sailing ships with complex rigging systems. Ancient civilizations, including the Egyptians, Phoenicians, and Polynesians, made remarkable advancements in sailing technology, enabling them to explore and trade over larger swathes of the ocean [2]. The medieval period saw the advent of the compass and the astrolabe, which further facilitated maritime navigation and exploration. The Age of Discovery, epitomized by the voyages of Columbus, Vasco da Gama, and Magellan, was propelled by advancements in sailing technology, which enabled transoceanic voyages and the establishment of maritime empires.

The industrial revolution in the 18th and 19th centuries marked a significant turning point in maritime propulsion. The invention of the steam engine heralded the decline of wind-powered sailing and the ascendancy of fuel-powered propulsion systems. Steamships and later, diesel-powered ships, offered greater reliability, speed, and capacity compared to their wind-powered predecessors, thus becoming the preferred mode of maritime transportation [3]. The transition to fuel-powered engines also mirrored the broader industrial and technological advancements of the era, which prioritized speed and efficiency over traditional methods.

1.0.1 A Renewed Interest in Wind Propultion

However, the environmental costs of fuel-powered maritime transportation have become increasingly apparent in the modern era. The shipping industry is a notable contributor to global carbon emissions, and the deleterious effects of pollution on marine ecosystems are well-documented [4]. These challenges have rekindled interest in wind propulsion as a sustainable alternative, prompting a re-examination of the principles that guided ancient and medieval sailors. The modern iteration of wind propulsion seeks to amalgamate the age-old wisdom of harnessing wind power with contemporary technological advancements to create eco-friendly and efficient maritime transportation systems.

Contemporary wind propulsion technologies like Flettner rotors, wing sails, and kite systems are being revisited to mitigate the environmental impact of maritime travel [3]. Among these, kite-powered vessel technology stands out due to its potential for higher efficiency and lower operational costs. Kites offer two main advantages over traditional sails: they can move relative to the vessel and can be flown at higher altitudes, accessing different wind systems.

The relative movement of kites generates apparent wind, allowing for maximum potential force even when the vessel is stationary. This enhanced apparent wind results in a larger force compared to a sail of equivalent area. Flying kites at higher altitudes taps into stronger and more consistent wind currents, making wind a more reliable energy source for propulsion [6].

However, the effective operation of kite-powered vessels requires precise control, which is skill-intensive. To leverage the full benefits of kites as a scalable propulsion method, implementing autonomous control is crucial.

1.0.2 Reinforcement Learning for Autonomous Control

Reinforcement Learning (RL), a subset of artificial intelligence, presents a compelling avenue for optimizing the autonomous control of kite-powered vessels. The paradigm of RL, predicated on the principles of learning from interaction with the environment, holds promise for devising sophisticated control strategies that can significantly enhance the energy efficiency and operational efficacy of kite-powered vessels [4].

Talk about what we are going to investigate in the paper

See if we can train a Reinforcement learning algorithm (agent) to be able to sail a boat (control direction and speed towards a target), while also flying a kite as the means of propulsion. First make an agent that can drive a boat, control its direction and speed, with minimal outside interference (sea state, wind). Integrate expand the agent

Aims and Objectives

Overall aim : Develope a RL algorithm for autonomous control of kite-powered vessels

This has several steps before we can train a RL algorithm to perform this we must: 1st stage: get an algorithm similar to a driving game - create the environment for training: - some form of sailing simulation, initially just a floating boat on water and a propeller for propulsion -expand the playable world to include a course

-build agent with observations, actions and rewards

CHAPTER 2

BACKGROUND

The maritime domain has long been a focal point of innovation and technological advancement. Historically, propulsion mechanisms have evolved from rudimentary oars to sophisticated sails, each iteration seeking to harness nature's forces more efficiently. Today, as we stand at the intersection of technology and tradition, there emerges a compelling avenue for exploration: kite-powered vessels. This innovative approach to propulsion seeks to leverage the aerodynamic advantages of kites, offering potential enhancements in efficiency and maneuverability over traditional sails.

However, the introduction of kites as a propulsion mechanism brings forth a new set of challenges. The dynamic nature of kites, combined with the unpredictable marine environment, necessitates advanced control systems capable of real-time adaptation and decision-making. This is where the application of machine learning, and more specifically Reinforcement Learning (RL), becomes paramount.

Reinforcement Learning, a subset of machine learning, operates on the principle of learning through interaction. By continuously interacting with its environment, an RL agent learns to make decisions that maximize a certain objective, often framed as a cumulative reward. The potential of RL in maritime propulsion is evident: it offers a framework for developing control systems that can adapt to changing conditions, optimizing kite propulsion in real-time.

This chapter aims to provide a comprehensive overview of the current state of kite-powered vessel technologies, with a particular emphasis on the integration of RL-based control systems. Through a detailed examination of relevant literature, we will identify existing research gaps, underscore the significance of the proposed work, and set the stage for the subsequent sections.

As we navigate through this background, we will delve into the core concepts of RL, its relevance to maritime propulsion, and its potential in revolutionizing the way we think about

sailing.

2.1 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a paradigm of machine learning that has been making waves, both literally and figuratively, in the vast ocean of artificial intelligence (AI). At its core, RL is about learning by interaction: an agent takes actions in an environment to maximize some notion of cumulative reward. The agent learns from the consequences of its actions, rather than from being explicitly taught, making it a powerful tool for tasks where the optimal strategy is unknown or hard to define [1].

Imagine teaching a child to ride a bicycle. You don't provide a step-by-step manual; instead, the child learns by trying different actions (like pedaling or balancing) and receiving feedback (falling down or moving forward). This trial-and-error approach is the essence of RL. The agent (in this case, the child) interacts with its environment (the bicycle and the ground) and learns a policy that dictates the best action to take in any given situation based on the rewards (or penalties) it receives [2].

Historically, RL has its roots in the fields of operations research and behavioral psychology. The idea of learning optimal strategies through interaction has been explored in various contexts, from game playing to industrial optimization [3]. However, it's the recent advancements in computational power and algorithms that have propelled RL to the forefront of AI research. Games like Go, which were once considered too complex for computers to master, have now been conquered by RL agents, showcasing the immense potential of this approach[4].

Now, let's explore how RL can be applied to the maritime world. Boats, with their intricate dynamics and the unpredictable nature of water, present a challenging environment for control systems. Traditional control methods often rely on predefined rules and heuristics, which might not always be optimal, especially in changing conditions. Enter RL. With its ability to learn from experience, an RL-based control system can adapt to varying conditions, ensuring smooth sailing even in turbulent waters, gusty winds and potentially more [5].

But why stop at boats? The concept of using kites to harness wind power for propulsion is not new. Historically, kites have been used in various cultures for fishing, transportation, and even warfare [6]. In the modern context, kites offer an exciting alternative to traditional sails, providing more power and maneuverability. However, controlling a kite, especially in varying wind conditions, is a complex task. This is where RL shines. By continuously interacting with the environment and adjusting the kite's position and angle, an RL agent can learn the optimal control strategy to harness the maximum wind power, propelling the boat efficiently [7].

The potential applications of RL in maritime navigation are vast. From optimizing routes for cargo ships to ensuring safe navigation in crowded ports, the possibilities are as vast as the open sea. Moreover, as environmental concerns become more pressing, the need for efficient and

sustainable maritime solutions becomes paramount. RL, with its ability to optimize and adapt, can play a pivotal role in addressing these challenges [8].

In conclusion, Reinforcement Learning is not just another tool in the AI toolkit; it's a paradigm shift in how we approach problem-solving. Its potential in the maritime world is just beginning to be tapped. As we venture into the future, with boats steered by intelligent agents and sails replaced by kites controlled with precision, it's clear that RL will be at the helm, guiding us towards uncharted territories and new horizons[9].

2.2 Unity Game Engine

Unity, a name that resonates with game developers and enthusiasts alike, stands as a beacon in the realm of game development. Born in the vibrant city of Copenhagen, Denmark, in 2005, Unity has since evolved into a powerhouse, democratizing game development and breathing life into iconic games like "Among Us" and "Pokemon Go" [10].

At its heart, Unity is a cross-platform game engine designed to craft both 2D and 3D experiences. It offers a harmonious blend of a powerful graphical editor and the flexibility of CSHARP coding, allowing developers to translate their visions into virtual realities [11]. While the engine's core is written in C++, it graciously opens its arms to developers familiar with CSHARP, making the development process both intuitive and efficient.

Diving into the basics of Unity game development, one is greeted with a plethora of tools and components that simulate real-world interactions. Unity's lighting, physics, rigidbody, and colliders work in tandem to create immersive environments. Whether it's the glint of sunlight reflecting off a surface or the realistic bounce of a ball, Unity ensures every detail is just right [12]. Developers can further enhance objects with custom CSHARP scripts, paving the way for unique gameplay experiences.

Imagine crafting a game level: a dodgeball arena illuminated by a radiant light source, with a camera capturing every thrilling moment. Unity makes this possible with simple objects like planes, cylinders, and spheres. The intuitive interface allows developers to select, move, rotate, and scale objects with ease, setting the stage for an exhilarating match [13].

But what's a game without some action? Unity's rigid body component breathes life into objects, allowing them to be influenced by gravity. Combine this with the material component, and you can create mesmerizing visual effects, from the sheen of a metallic surface to the rough texture of a stone [14].

Unity's commitment to realism and smooth gameplay is further evident in its two types of updates: Update and FixedUpdate. While the former is called every frame during gameplay, ensuring fluid animations and interactions, the latter syncs with the physics engine's frame rate, making it ideal for moving objects around [15].

Now, envision a player navigating this dodgeball arena, deftly maneuvering with the arrow

keys, while a ball rolls with momentum as the game begins. Unity makes this possible with simple input methods in the FixedUpdate and scripts that add force to objects [16].

Unity's ML-Agents toolkit is a game-changer for those looking to infuse artificial intelligence into their games. ML-Agents provides a platform to train intelligent agents within the Unity environment using Reinforcement Learning, imitation learning, and more. This makes it an ideal choice for complex simulations like kiteboat training, where agents can learn optimal strategies through interaction.

In conclusion, Unity is not just a game engine; it's a canvas for creativity, a platform for innovation, and a testament to the limitless possibilities of virtual worlds. As we set sail in our virtual kiteboat, with the winds of Unity propelling us forward, the horizon looks promising and full of potential.

2.3 Proximal Policy Optimization (PPO)

Reinforcement Learning (RL) has witnessed a plethora of algorithms, each striving to optimize policy in its unique way. Among these, the Proximal Policy Optimization (PPO) algorithm stands out as a beacon of efficiency and simplicity [17].

PPO is a member of the policy gradient family of RL algorithms. Unlike traditional policy gradient methods that perform a single gradient update per data sample, PPO introduces a "surrogate" objective function. This novel approach allows for multiple epochs of minibatch updates, optimizing the policy over a series of iterations. The essence of PPO lies in its ability to alternate between sampling data through interaction with the environment and optimizing the surrogate objective using stochastic gradient ascent.

The inception of PPO was driven by the need for an algorithm that combined the best of all worlds: scalability, data efficiency, and robustness. While deep Q-learning and vanilla policy gradient methods have their merits, they often fall short in terms of data efficiency and robustness. Trust Region Policy Optimization (TRPO), on the other hand, although effective, is relatively intricate and lacks compatibility with certain architectures [18].

PPO seeks to bridge these gaps. It aims to achieve the data efficiency and consistent performance of TRPO but does so using only first-order optimization. The brilliance of PPO is encapsulated in its objective with clipped probability ratios. This objective provides a pessimistic estimate (or a lower bound) of the policy's performance. The optimization process in PPO is iterative, alternating between data sampling from the policy and performing several epochs of optimization on this sampled data.

Empirical evidence underscores the efficacy of PPO. When pitted against various versions of the surrogate objective, PPO, with its clipped probability ratios, emerges as the top performer. Furthermore, in head-to-head comparisons with other algorithms, PPO shines brightly. On continuous control tasks, PPO outperforms its competitors. In the realm of Atari games, PPO

showcases superior sample complexity compared to A2C and performs on par with ACER, all while maintaining a simpler architecture.

But the story doesn't end with PPO alone. Unity's ML-Agents toolkit, which was touched upon earlier, seamlessly integrates with PPO. ML-Agents provides a platform for training intelligent agents within the Unity environment, and when combined with the power of PPO, it paves the way for robust and efficient training regimes. This synergy between PPO and ML-Agents is particularly promising for complex simulations, such as kiteboat training, where agents can iteratively learn and refine their strategies for optimal performance.

The Proximal Policy Optimization algorithm is a testament to the continuous evolution and innovation in the field of Reinforcement Learning. Its simplicity, efficiency, and robustness make it a prime choice for a myriad of applications. As we harness the combined power of PPO with Unity's ML-Agents for kiteboat simulations.

CHAPTER 3

METHODOLOGY

This section will investigate the approach taken to create an autonomously controlled kiteboat in a simulation. As discussed in section 2.2 Reinforcement Learning (RL) is an appropriate type of ai for the control of a kiteboat. RL works on a trail and error approach and its core loop is shown in figure LABLE.

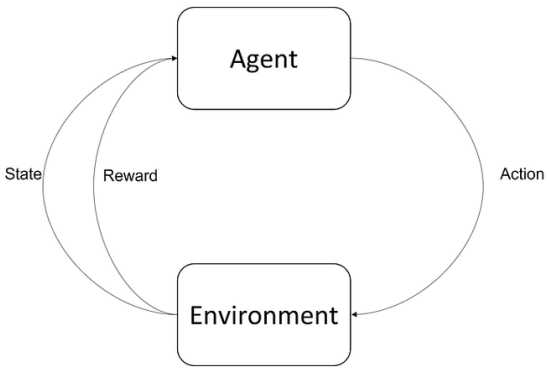

Figure 3.1: The RL Loop

3.1 The Environment

The first step to this process if to create the environment, which the agent will use to train. In this case that will need to look something like a sailing game; it will have some form of water, a boat, a kite and a course. For any machine learning endeavor, especially one with such intricate physical dynamics, the choice of simulation environment is paramount. Not only does it provide

the playground for our AI agent to learn and make mistakes safely, but it also serves as a litmus test for the robustness and realism of the designed model.

Given the myriad of choices available, the Unity game engine emerged as the most suitable platform. Beyond its reputation in gaming, The inherent support for mesh bodies, colliders, and a variety of joints made it an attractive option for simulating the kite-boat system, which comprised a complex dance of forces, and counterforces. Unity is recognized for its potent physics engine, however for this project the use of Unity's physics engine will be kept to a minimum. Unity will be used primarily for its visual capabilities and the ability to run machine learning simulations.

Central to our simulation is the depiction of water, the medium in which our boat will navigate. Here, the Unity HDRP Water System 16.0.3 [cite] came to the rescue. Bundled with Unity 2023.2.0b9 [cite], this water system provides a realistic representation of water with its undulating waves, refractions, and reflections. The alternative option to using Unity's water system was to model an entire particle fluid simulation, this would have had its advantages, however it would have been a lot more computationally expensive and would have taken a lot longer to implement. As this project was primarily focused on creating a RL algorithm for controlling a kiteboat it was decided that the Unity water system would be sufficient for this project.

The boat part of the kiteboat had two main component scripts, the buoyancy and the rudder, allowing the boat to float and be steered. The implementation of Buoyancy and the rudder are discussed in more detail in section 3.1.1 and 3.1.2 respectively.

Several assumptions were made before modeling the boat, these were primarily made to save time on the implementation and were deemed to have insignificant effects on the overall performance of the boat. These assumptions were:

- The Archimedes force is uniform across all submerged sections of the boat.
- The rudder forces of lift and drag could be aproximated to a torque applied to the boat.
- Once the boat is moving at a speed greater than 0.25m/s the lift and drag forces of the keel are equal to the downwind component of the kite's resultant force- essentially providing a non'slip condition.

3.1.1 Buoyancy

Buoyancy, the force that allows ships to float, was the first physical property to be addressed. Rooted in Archimedes' Principle, it dictates that the buoyant force exerted on a submerged body is equivalent to the weight of the fluid displaced by that body. In our Unity environment, the boat's hull, represented as a 'mesh' with an associated 'mesh collider', was divided into many small triangles or Voxels. These Voxels became the fundamental units for calculating buoyancy, allowing for a granular and realistic representation of the boat's interaction with water. This was achieved by first calculating the total Archimedes force (AF) of the entire boat using equation 3.1,

followed by a local AF at each Voxel. The water level, y component, was then computed at each voxel's (x,z) coordinates to determine if it was above or below the surface. If below the surface the component of the AF was applied vertically at each voxel. This implementation can be viewed in the buoy.cs script in the project REF IN APENDIX.

$$F_B = \rho_w g V \tag{3.1}$$

3.1.2 Rudder

While buoyancy ensures our boat doesn't sink, it's the rudder that grants it direction. The Rudder.cs script handles the implementation of the rudder and the keel. Several assumptions were made

3.1.3 Course Generation

3.1.4 Mark Roundings

3.2 MLAgents

3.3 The Environment Structure

Keep alive problem.

Diverse conditions

ciriculum style learning

3.4 Initial Training

3.5 Optimization

3.5.1 Blue Crystal HPC

The Blue Crystal HPC, operated by the University of Bristol, offers significant computational resources tailored for intensive tasks such as machine learning simulations, like.

To utilize Blue Crystal for MLAgents simulations, the following steps were undertaken:

- **Access and Security:** Gained access to the university's HPC and set up SSH keys for secure communication.
- **File Preparation:** Built the .x86_64 Unity build file and uploaded it, along with the necessary config file, to Google Drive.

- **Automation with Shell Script:** Developed a shell script to automate the process. This script:
  - Retrieves the build files from Google Drive.
  - Sets up a virtual environment on Blue Crystal.
  - Installs the ML-Agents toolkit.
  - Initiates the simulation.
  - Upon completion, uploads the results back to Google Drive[3].

Useful commands for working with Blue Crystal:

- **sbatch:** Submits a job to the queue.
- **sacct:** Checks the status of a job.
- **scancel:** Cancels a job.

Parallelization and Optimization:

Initially, a single node on Blue Crystal was employed to run the simulation. This node with 28 CPUs was responsible for both hosting the environment and executing the model. However, Blue Crystal's architecture allows for more advanced parallelization strategies. Distributing the simulation across multiple nodes can enhance efficiency. Additionally, offloading the ML-Agents toolkit to a GPU core can further accelerate the learning process.

However, it's worth noting that the demand for GPUs on Blue Crystal is high. For tasks that don't necessitate the power of GPUs, relying on CPUs, even if they take longer, is a practical choice given the limited GPU availability.

APPENDIX A

Begins an appendix

BIBLIOGRAPHY

[1] Richard S Sutton and Andrew G Barto.
Reinforcement learning: An introduction.
MIT press, 2018.

[2] Christopher J Watkins and Peter Dayan.
Q-learning.
Machine learning, 8(3-4):279–292, 1992.

[3] Richard Bellman.
Dynamic Programming.
Princeton University Press, 1957.

[4] David Silver et al.
Mastering the game of go with deep neural networks and tree search.
Nature, 529(7587):484–489, 2016.

[5] Chen Yang and Lin Wu.
Reinforcement learning-based control for autonomous boats: A survey.
Journal of Marine Science and Technology, 25(1):1–10, 2020.

[6] Richard Hallion.
Taking flight: inventing the aerial age, from antiquity through the First World War.
Oxford University Press, 2003.

[7] Moritz Erhard and Hauke Strauch.
Control of towing kites for seagoing vessels.
IEEE Transactions on Control Systems Technology, 21(5):1629–1640, 2013.

[8] Marielle Christiansen, Kjetil Fagerholt, and David Ronen.
Ship routing and scheduling in the new millennium.
European Journal of Operational Research, 228(3):467–483, 2013.

[9] Volodymyr Mnih et al.
Human-level control through deep reinforcement learning.
Nature, 518(7540):529–533, 2015.

[10] Merlin.
Unity in 100 seconds, 2023.
URL https://www.youtube.com/watch?v=iqlH4okiQqg.

[11] Unity Technologies.
Unity User Manual, 2021.

[12] Will Goldstone.
Unity 3.x Game Development Essentials.
Packt Publishing Ltd, 2010.

[13] Ashley Harrison.
Mastering Unity 2D Game Development.
Packt Publishing Ltd, 2013.

[14] Sue Blackman.
Unity 3D Game Development by Example Beginner's Guide.
Packt Publishing Ltd, 2012.

[15] Unity Technologies.
Unity scripting api: Update and fixedupdate, 2022.
URL https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html.

[16] Unity Technologies.
Unity scripting api: Rigidbody, 2022.
URL https://docs.unity3d.com/ScriptReference/Rigidbody.html.

[17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov.
Proximal policy optimization algorithms.
arXiv preprint arXiv:1707.06347, 2017.
URL https://arxiv.org/pdf/1707.06347.pdf?fbclid=IwAR1DwRSkBzhqXRhVh3XQ_Nu_XYvN_sMR4ppYM28h3qAtx9EK03LrlOcF7Dg.

[18] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, F. Janoos, L. Rudolph, and A. Madry.
Implementation matters in deep rl: A case study on ppo and trpo.
In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.
URL https://dblp.org/rec/conf/iclr/EngstromISTJRM20.html.