University of Oxford

DEPARTMENT OF
STATISTICS

The Application of Genetic Reinforcement Learning Techniques for the Control of Microscopic Robots

by

Benjamin Archer

Wolfson College

Under the supervision of

Valentin De Bortoli, Arnaud Doucet, Clément Moreau

A dissertation submitted in partial fulfilment of the degree of Master of Science in Applied Statistics.

Department of Statistics, 24–29 St Giles,
Oxford, OX1 3LB

September 2021

This is my own work (except where otherwise indicated)

Candidate: Benjamin David Archer

Signed: Benjamin Archer

Date: 27/09/2021

Abstract

Bacteria, plankton, and sperm are examples of natural self-propelled microorganisms. These often-unicellular organisms are able to deform and change their shape in order to move through their microscopic environment. Drawing inspiration from these natural swimmers is crucial for the development of artificial robots that could be used in medicine for microsurgery, diagnosis or targeted drug delivery.

In this dissertation, we focus upon a model of these organisms; a synthetic, artificial microswimmer. This swimmer is represented through a chain of N inflexible links able to deform and move in the 2D plane. Our aim is to design locomotory gaits for this N-link swimmer where due to the microscopic scale, the constraints on self-propulsion differ [1]. We propose to diverge from defining swimming strokes a priori and discover these gaits through Reinforcement Learning, specifically using the genetic algorithm NeuroEvolution of Augmenting Topologies (NEAT).

We model each microswimmer's interactions with its environment through Resistive Force Theory (RFT) and define a system of Ordinary Differential Equations (ODEs) that fully describe its equations of motion. We then produce an explicit formula for a generalised N-link Purcell stroke with which we can compare our results. To improve on this, we introduce the concepts of reinforcement learning and NEAT, explicitly stating the methods involved. We will apply these techniques to our scenario and assess our developed locomotory gaits, comparing them to strokes found through optimal control theory [2]. We aim to show that not only is the optimal stroke found using this technique but also strokes more complex than could be found using optimal control theory.

A large part of this dissertation is the code used to model, train and compare these microswimmers, all of which is available, fully annotated, on GitHub:

https://github.com/benarcher444/Oxford-Dissertation

Acknowledgements

I would like to thank my research supervisors, Arnaud Doucet, Valentin De Bortoli and Clément Moreau. Their help has been invaluable throughout this project and it has been an honour to work with them and share their passion.

Contents

1 Introduction **3**
- 1.1 An Introduction to the Microscopic World . . . 3
- 1.2 Reynolds Numbers . . . 3
- 1.3 N-link Swimmers . . . 4
- 1.4 Structure and Contributions . . . 5

2 The N-link Swimmer Model **7**
- 2.1 Equations of Motion . . . 8
- 2.2 Computational Methods for Calculating the N-matrix . . . 14
- 2.3 Numerical Errors . . . 17
  - 2.3.1 The Euler Method . . . 18
  - 2.3.2 The Classic Runge-Kutta Method . . . 18
  - 2.3.3 Reciprocal Motions for Testing Numerical Errors . . . 18
- 2.4 Reproducing the Purcell Stroke . . . 20
  - 2.4.1 Analysis of the Influence of L and δ . . . 21
  - 2.4.2 The N-link Purcell Swimmer . . . 24

3 Reinforcement Learning **26**
- 3.1 Introduction to Reinforcement Learning . . . 26
- 3.2 NeuroEvolution . . . 29
  - 3.2.1 Neural Network Topology . . . 29
- 3.3 NeuroEvolution of Augmenting Topologies . . . 30
  - 3.3.1 Representation, Mutation and Crossover of Different Topologies . . . 30
  - 3.3.2 Protecting Innovation through Speciation . . . 32
  - 3.3.3 The NEAT Method . . . 34

4 Applying NEAT to the N-link Model Swimmers **35**
- 4.1 Constructing our Reinforcement Learning Scenario . . . 35
  - 4.1.1 Defining the State . . . 35
  - 4.1.2 Defining the Actions . . . 36
  - 4.1.3 The Reward Function . . . 37
- 4.2 Configuration of NEAT . . . 37

5 Analysis of Results **40**
- 5.1 The Trained 3-link Stroke . . . 40
- 5.2 Larger N Trained N-link Strokes . . . 43

6 Conclusion **51**

Chapter 1

Introduction

1.1 An Introduction to the Microscopic World

The macroscopic world - the world we see around us, tends to act how we would expect given the physics we are used to. We have learnt, through experience, to expect certain responses from certain causes. For example, the slightest tap to a marble on a smooth flat surface would cause it to roll a significant distance. Its inertia carries it. However, at a microscopic level, the rules now change. Viscous forces dominate and inertia is all but irrelevant. The marble would stop the second it left your finger.

At first, this may seem odd, however, for the vast majority of life on Earth this is very normal. All microscopic cells, from sperm cells to bacteria, live within these microscopic rules. For this reason, there is currently a growing interest in understanding how microscopic swimmers move through their inertia-less environment [3] [4] [5]. It is an essential and crucial process in biology. Reproduction within mammals is just one example that fundamentally relies upon it [6]. Through research, it has been observed that microorganisms evolving within these constraints have developed diverse methods of locomotion. They manage to exploit viscous resistance to motion, waving and rotating flagellum with molecular motors to propel themselves forward [7]. Taking inspiration from them, synthetic microswimmers, able to navigate complex biological environments, would have applications throughout medicine and micro and nano technology. Microswimmers could be used in the blood, performing targeted drug delivery, removing plaque (rotational atherectomy) or even destroying blood clots (thrombolysis) [8]. These are just a few of the important and potentially lifesaving areas fully mobile microswimmers could excel in.

1.2 Reynolds Numbers

Firstly, we will focus on the forces on particles at microscopic levels. Figure (1.1) shows an object with velocity v and dimension a of any shape, travelling through a mixture with viscosity η and density ρ. Our focus is upon its Reynolds number, $\mathcal{R}$, a dimensionless measure of the ratio of the inertial forces to the viscous forces upon it.

$$\mathcal{R} = \frac{av\rho}{\eta} \approx \frac{\text{Inertial Forces}}{\text{Viscous Forces}} \tag{1.1}$$

We will define $\nu = \frac{\rho}{\eta}$ as the kinematic viscosity of a substance. This gives us the equation

$$\mathcal{R} = \frac{av}{\nu}. \tag{1.2}$$

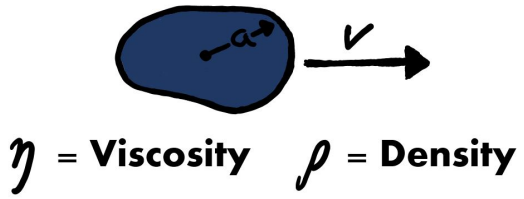

Figure 1.1: An object with dimension a moving through a fluid with velocity v.

For water, we have $\nu \approx 10^{-2}\text{cm}^2 s^{-1}$. Figure (1.2) shows the effect as you become smaller and a decreases, the Reynolds number dropping to 10^{-4} or 10^{-5} for water bound microbes [9]. By definition, at low Reynolds numbers the viscous forces will dominate.

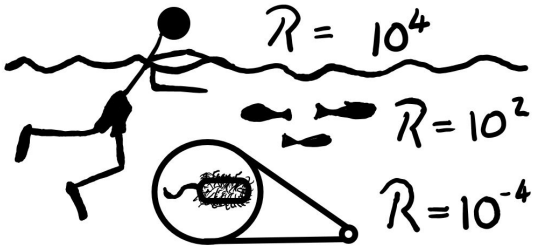

Figure 1.2: The Reynolds numbers for different objects of varying size travelling through water.

The irrelevance of inertia at microscopic scale leads to some interesting observations. Time seemingly doesn't matter. The pattern of motion is the same, whether you move fast or slow [9]. Imagine you are riding a fixed pedal bike in an inertia free world, such that when you stop pedalling the bike stops moving. As each rotation of the pedals turns the wheels a certain amount, and therefore moves you forward that same amount, if you turned the pedals fast or slow, as long as they turned the same amount of times you would always finish at the same place.

This further leads to what is often called the Scallop Theorem [9] which states that no reciprocal motion can produce an overall net displacement. This theorem gets its name from the scallop, an animal which moves by opening its shell slowly then closing it fast. This method of motion would not work at low Reynolds numbers.

1.3 N-link Swimmers

Now, as reciprocal motions cannot cause a net motion, we must now delve further to develop swimming gaits that can. Purcell hypothesised that an animal with as little as just two hinges could swim, using what is now called the Purcell Stroke [9]. This hypothetical organism is visualised in Figure (1.3) with a simple structure of three straight rods with hinges marked in

red. This Purcell stroke is not reciprocal, but it does cycle.

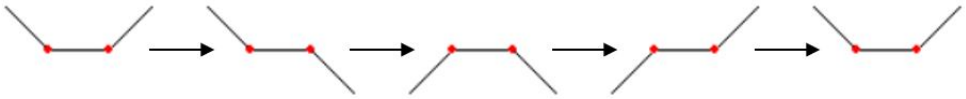

Figure 1.3: The four stages of the Purcell stroke. The fifth diagram shows the final step, back into the original shape, so that the cycle can begin again.

We will take inspiration from this idea and construct swimmers like shown in Figure (1.3). Our swimmers will be modelled from a set of inflexible rods joined together by hinges. We will define these models as N-link swimmers, where N is the total number of rods in the swimmer. For example, Figure (1.3) shows a 3-link Swimmer and Figure(1.4) shows a 6-link Swimmer. In essence, we are taking inspiration from biological swimmers and building a discrete model of a flexible tail shaped flagellum. As N grows, the approximation becomes more and more precise. Using this discretization satisfyingly simplifies the kinematics [10] and allows our swimmers to move by controlling the hinges throughout their structure.

Figure 1.4: A 6-link Swimmer.

1.4 Structure and Contributions

Effective swimming strokes for these synthetic microswimmers can be difficult to define. How does a synthetic swimmer swim efficiently and how can this be programmed? In this dissertation, we aim to apply reinforcement learning to this problem. We will allow a simple model of a synthetic microswimmer to discover and learn efficient swimming methods by themselves, without any prior assumptions. In section 2, we will mathematically define our N-link model then link this swimmer to its environment, modelling the viscous forces to produce equations of motion. We will further discuss numerical issues throughout the process then apply the Purcell swimming stroke, experimenting with its constraints. In section 3, we will introduce the core concepts of Reinforcement Learning along with NEAT, the main algorithm we will use throughout the project. We will fully define and describe the methods within this algorithm, then in section 4 apply it to the microscopic scenario. Finally, in section 5 we will analyse our results and assess the efficiency of our swimmers and the learning techniques.

Another goal of this project has been to produce code allowing these swimmers to be efficiently simulated and visualised. This code, with the trained models, can be found through the link in the abstract. With this code you can visualise each type of swimming stroke moving through its viscous environment in real time.

Through this dissertation, our contributions are as follows:

1. In Chapter 2, we introduce a N-link generalisation of the 3-link Purcell swimmer. We assess the relationship between the number of links and the displacement travelled.

2. We analyse the influence of both the size of a swimmer and the maximum angle excursion in Chapter 2 and Chapter 5. We vary each, and assess the effect on the efficiency of both 3-link Purcell swimmers and 3-link taught swimmers.

3. We provide a methodological contribution, applying the NEAT algorithm to the 3-Link micro-swimmer scenario in Chapter 3 and Chapter 4. We then compare these results to the optimal stroke [2] in Chapter 5 proving this technique's ability to converge to optimality.

4. In Chapter 5, we expand this methodology to N-links, producing strokes for 4-link, 5-link, 6-link and 10-link swimmers. We then compare these strokes to their Purcell alternatives.

5. We provide open-source code, which allows the training, comparing and visualising of NEAT swimmers. This can be found on GitHub through the link in the abstract.

Chapter 2

The N-link Swimmer Model

In this section, we aim to mathematically define our model of a swimmer. We will then proceed to model its interactions with its environment and assess any numerical errors. Finally, we introduce a generalisation of the Purcell swimmer and analyse the influence of the constraints upon it.

An N-link swimmer will consist of N links with $N-1$ hinges. The swimmer will exist in the 2D plane, defined by $(\mathbf{e}_x, \mathbf{e}_y)$. We set $\mathbf{e}_z := \mathbf{e}_x \times \mathbf{e}_y$. The ith link in our swimmer stretches from start point $\mathbf{x}_i$ to end point $\mathbf{x}_{i+1}$, where $\mathbf{x}_i$ represents the two-dimensional coordinates $(x_i, y_i) \in \mathbf{R}^2$. Each link has length L_i and we define θ_i as the angle it makes with the horizontal x-axis. Using trigonometry, we can now express $\mathbf{x}_i$, for $i \in \{2, \ldots, N\}$, as a function of $\mathbf{x}_1$, θ_k and L_k for $k \in \{1, \ldots, i-1\}$

$$\begin{aligned} x_i &= x_1 + \sum_{k=1}^{i-1} L_k \cos(\theta_k), \\ y_i &= y_1 + \sum_{k=1}^{i-1} L_k \sin(\theta_k). \end{aligned} \tag{2.1}$$

Furthermore, we will reparameterize the θ_i to allow a more meaningful representation of the angles along the swimmer. We define

$$\alpha_i = \theta_i - \theta_{i-1}, \qquad \text{for } i \in \{2, \ldots, N\}, \tag{2.2}$$

giving us the relative bend from one link to the next. These α_i are shown in Figure (2.1).

As the length of each link is kept constant, we can now fully describe the shape of the N-link swimmer with just $N+2$ variables: x_1, y_1, θ_1 and α_i for $i \in \{2, \ldots, N\}$. The first three define the starting point and orientation of the swimmer's first link, the rest then allow us to calculate the swimmer's full position. This is more efficient than recording every $\mathbf{x}$ coordinate, which would require instead $2N$ variables.

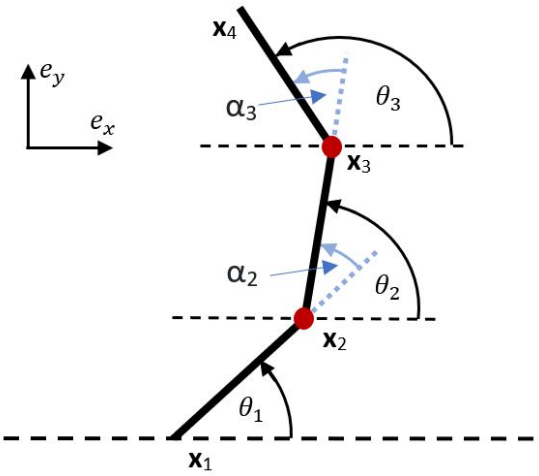

Figure 2.1: A 3-link swimmer is shown, with the joints marked in red. The angles θ_i for $i \in \{1, 2, 3\}$ and α for $i \in \{2, 3\}$ are marked on, along with $\mathbf{x}_i$ for $i \in \{1, \ldots, 4\}$.

2.1 Equations of Motion

We now consider the kinematics of the N-link model, examining the relationship between the swimmer's movement and the viscous liquid around it. Our goal is to understand how the model maneuvers as it changes shape in its microscopic environment. With this understood, we will then aim to reproduce and simulate the Purcell swimming stroke.

As discussed before, to calculate the full orientation of a N-link swimmer, we need $N + 2$ variables: x_1, y_1, θ_1 and α_i for $i \in \{2, \ldots, N\}$. Consequently, to plot the swimmers motion as it swims, we need to calculate and record the values of these $N + 2$ variables over time. We will do this by calculating their time derivatives. During this dissertation, we will define $\dot{\omega}$ as the time derivative of ω. Our swimmers move by varying the values of each α_i and therefore by controlling the values of $\dot{\alpha}_i$ for $i \in \{2, \ldots, N\}$. This leaves us to calculate $\dot{x}$, $\dot{y}$ and $\dot{\theta}_i$ given $\dot{\alpha}_i$. In essence, we aim to construct a function, f, such that

$$f\left(\dot{\alpha_2}, \dot{\alpha_3}, \ldots, \dot{\alpha_N}\right) = \left(\dot{x_1}, \dot{y_1}, \dot{\theta_1}\right) \tag{2.3}$$

Firstly, we know that the dynamics of the swimmer must follow Newton's laws. This gives us a set of 3 ordinary differential equations (ODEs) forming a system which completely governs the dynamics of the swimmer [10]. As we are working with microscopic swimmers at low Reynold numbers, we can neglect inertia entirely, giving us the equations

$$\begin{cases} \mathbf{F} = \mathbf{0}, \\ \mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z = 0, \end{cases} \tag{2.4}$$

which can also be written as,

$$\begin{cases} F_x = 0, \\ F_y = 0, \\ \mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z = 0, \end{cases} \tag{2.5}$$

where $\mathbf{F}$ is the total net force applied to the swimmer and F_x and F_y are the net forces in the directions $\mathbf{e_x}$ and $\mathbf{e_y}$ respectively. $\mathbf{T}_{\mathbf{x}_1}$ represents the total torque acting on the point $\mathbf{x_1}$. These are our 3 ODEs to solve.

Due to the microscopic scale, the non-local hydrodynamic forces exerted on the swimmer can be easily approximated with Resistive Force Theory (RFT), in which the drag forces depend linearly on the velocity of each point [11]. This method has been proven to be relatively accurate with the N-link model [12] and allows us to directly couple the fluid and the swimmer.

It is now useful to introduce two unit vectors,

$$\mathbf{e}_i = \begin{pmatrix} \cos(\theta_i) \\ \sin(\theta_i) \end{pmatrix}, \qquad \mathbf{e}_i^{\perp} = \begin{pmatrix} -\sin(\theta_i) \\ \cos(\theta_i) \end{pmatrix}, \tag{2.6}$$

which are parallel and perpendicular respectively to the ith link as shown in Figure (2.2).

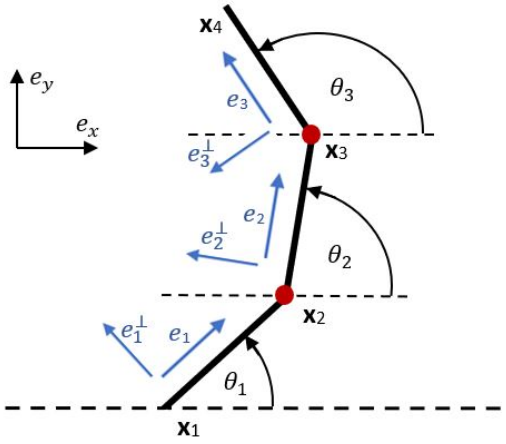

Figure 2.2: A 3-link swimmer is shown, with the joints marked in red. The angles θ_i for $i \in \{1, 2, 3\}$ and $\mathbf{x}_i$ for $i \in \{1, \ldots, 4\}$ are marked on, along with both e_i and $e_i^{\perp}$ for $i \in \{1, \ldots, 3\}$ which are marked as blue arrows.

We also introduce the variable s, $(0 \leq s \leq L_i)$, as the distance of a point from $\mathbf{x}_i$ along the ith link. Using these, we can define the set of points on the ith link using,

$$\mathbf{x}_i(s) = \mathbf{x}_i + s\mathbf{e}_i. \tag{2.7}$$

Differentiating this also gives us the velocity at each point

$$\mathbf{v}_i(s) = \dot{\mathbf{x}}_i + s\dot{\theta_i}\mathbf{e}_i^{\perp}. \tag{2.8}$$

where $\dot{\mathbf{x}}_i = (\dot{x}_i, \dot{y}_i)$. The second term is calculated using the chain rule

$$\frac{\mathrm{d}\mathbf{e}_i}{\mathrm{d}t} = \frac{\mathrm{d}\mathbf{e}_i}{\mathrm{d}\theta_i}\frac{\mathrm{d}\theta_i}{\mathrm{d}t}. \tag{2.9}$$

Using RFT, the density of the force acting on the ith segment, $\mathbf{f}_i$ is approximated by drag forces depending linearly on the velocity. We can therefore define it as

$$\mathbf{f}_i(s) := -\xi(\mathbf{v}_i(s)\cdot\mathbf{e}_i)\mathbf{e}_i - \eta(\mathbf{v}_i(s)\cdot\mathbf{e}_i^{\perp})\mathbf{e}_i^{\perp}, \tag{2.10}$$

where ξ and η are the drag coefficients parallel and perpendicular to the ith link respectively. We will set these constants as $\xi = 0.38\times10^{-3}$ Nsm $^{-2}$ and $\eta = 0.7182\times10^{-3}$ Nsm $^{-2}$ as per [10] giving a ratio of $\frac{\eta}{\xi} = 1.89$. Note that within the RFT framework, only the ratio of the friction coefficients play a role in determining swimming velocities [12].

By summing the forces along each link in our model, we obtain

$$\begin{cases} \mathbf{F} = \displaystyle\sum_{i=0}^{N}\int_0^{L_i}\mathbf{f}_i(s)\,ds, & (2.11)\\[2ex] \mathbf{T}_{\mathbf{x}_1}\cdot\mathbf{e}_z = \mathbf{e}_z\cdot\displaystyle\sum_{i=0}^{N}\int_0^{L_i}(\mathbf{x}_i(s)-\mathbf{x}_1)\times\mathbf{f}_i(s)\,ds. & (2.12)\end{cases}$$

We will now work to simplify and solve this system. First, we will begin by looking at the expression for the force, defined in equation (2.10). We can express this in terms of $\dot{\mathbf{x}}_i$ and $\dot{\theta}_i$ by plugging (2.8) into (2.10) to get the equation

$$\mathbf{f}_i(s) = -\xi\Big((\dot{\mathbf{x}}_i + s\dot{\theta}_i\mathbf{e}_i^{\perp})\cdot\mathbf{e}_i\Big)\mathbf{e}_i - \eta\Big((\dot{\mathbf{x}}_i + s\dot{\theta}_i\mathbf{e}_i^{\perp})\cdot\mathbf{e}_i^{\perp}\Big)\mathbf{e}_i^{\perp}. \tag{2.13}$$

As $\mathbf{e}_i$ and $\mathbf{e}_i^{\perp}$ are perpendicular unit vectors, we know that

$$\mathbf{e}_i\cdot\mathbf{e}_i = 1, \qquad \mathbf{e}_i^{\perp}\cdot\mathbf{e}_i^{\perp} = 1, \qquad \mathbf{e}_i\cdot\mathbf{e}_i^{\perp} = 0. \tag{2.14}$$

Therefore

$$\mathbf{f}_i(s) = -\xi(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i)\mathbf{e}_i - \eta(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i^{\perp})\mathbf{e}_i^{\perp} - \eta s\,\dot{\theta}_i\mathbf{e}_i^{\perp}. \tag{2.15}$$

Using this, we can now write (2.11) as

$$\begin{aligned}\mathbf{F} &= \sum_{i=0}^{N}\int_0^{L_i} -\xi(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i)\mathbf{e}_i - \eta(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i^{\perp})\mathbf{e}_i^{\perp} - \eta s\,\dot{\theta}_i\mathbf{e}_i^{\perp}ds, & (2.16)\\ &= \sum_{i=0}^{N}\left[-\xi s(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i)\mathbf{e}_i - \eta s(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i^{\perp})\mathbf{e}_i^{\perp} - \eta\frac{s^2}{2}\,\dot{\theta}_i\mathbf{e}_i^{\perp}\right]_0^{L_i}, & (2.17)\\ &= \sum_{i=0}^{N} -\xi L_i(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i)\mathbf{e}_i - \eta L_i(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i^{\perp})\mathbf{e}_i^{\perp} - \eta\frac{L_i^2}{2}\,\dot{\theta}_i\mathbf{e}_i^{\perp}. & (2.18)\end{aligned}$$

Or, using the definition of $\mathbf{e}_i$ and $\mathbf{e}_i^{\perp}$ as expressed in (2.6), we can say

$$F_x = \sum_{i=0}^{N} -\xi L_i(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i)\cos(\theta_i) + \eta L_i(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i^{\perp})\sin(\theta_i) + \eta\frac{L_i^2}{2}\,\dot{\theta}_i\sin(\theta_i), \tag{2.19}$$

$$F_y = \sum_{i=0}^{N} -\xi L_i(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i)\sin(\theta_i) - \eta L_i(\dot{\mathbf{x}}_i\cdot\mathbf{e}_i^{\perp})\cos(\theta_i) - \eta\frac{L_i^2}{2}\,\dot{\theta}_i\cos(\theta_i). \tag{2.20}$$

Next, we focus on to (2.12), the torque in the system. Firstly, we use equation (2.15) to get

$$\mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z = \mathbf{e}_z \cdot \sum_{i=0}^{N} \int_0^{L_i} (\mathbf{x}_i(s) - \mathbf{x}_1) \times \Big(-\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\mathbf{e}_i - \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\mathbf{e}_i^{\perp} - \eta s\, \dot{\theta}_i \mathbf{e}_i^{\perp} \Big)\, ds. \tag{2.21}$$

We can also move $\mathbf{e}_z = (0,0,1)^T$ to the inside of the sum and the integral. This gives us

$$\mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z = \sum_{i=0}^{N} \int_0^{L_i} A\ ds \tag{2.22}$$

where

$$\begin{aligned} A &= \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \cdot \left(\begin{pmatrix} x_i(s) - x_1 \\ y_i(s) - y_1 \\ 0 \end{pmatrix} \times \begin{pmatrix} -\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\cos(\theta_i) + \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\sin(\theta_i) + \eta s\, \dot{\theta}_i \sin(\theta_i) \\ -\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\sin(\theta_i) - \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\cos(\theta_i) - \eta s\, \dot{\theta}_i \cos(\theta_i) \\ 0 \end{pmatrix} \right), \\ &= -(x_i(s) - x_1)(\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\sin(\theta_i) + \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\cos(\theta_i) + \eta s\, \dot{\theta}_i \cos(\theta_i)) \\ &\quad - (y_i(s) - y_1)(-\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\cos(\theta_i) + \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\sin(\theta_i) + \eta s\, \dot{\theta}_i \sin(\theta_i)). \end{aligned}$$

Now, using (2.7) we can write

$$x_i(s) = x_i + s\cos(\theta_i), \qquad y_i(s) = y_i + s\sin(\theta_i), \tag{2.23}$$

which allows us to replace $x_i(s)$ and $y_i(s)$ in our expression for $\mathbf{A}$. We then expand the brackets, giving a polynomial in terms of s

$$\begin{aligned} A = &-(x_i - x_1)(\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\sin(\theta_i) + \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\cos(\theta_i) + \eta s\, \dot{\theta}_i \cos(\theta_i)) \\ &- (\xi s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\sin(\theta_i)\cos(\theta_i) + \eta s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\cos^2(\theta_i) + \eta s^2 \dot{\theta}_i \cos^2(\theta_i)) \\ &- (y_i - y_1)(-\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\cos(\theta_i) + \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\sin(\theta_i) + \eta s\, \dot{\theta}_i \sin(\theta_i)) \\ &- (-\xi s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\sin(\theta_i)\cos(\theta_i) + \eta s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\sin^2(\theta_i) + \eta s^2\, \dot{\theta}_i \sin^2(\theta_i)). \end{aligned}$$

Here, the second and fourth lines of this addition will cancel down. The first term in each will cancel out and the second and third simplify using the identity

$$\sin^2(\theta) + \cos^2(\theta) \equiv 1. \tag{2.24}$$

Finally, placing $\mathbf{A}$ back into (2.22), we are left with

$$\begin{aligned} \mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z = \sum_{i=0}^{N} \int_0^{L_i} \Big(&- (x_i - x_1)(\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\sin(\theta_i) + \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\cos(\theta_i) + \eta s\, \dot{\theta}_i \cos(\theta_i)) \\ &- (y_i - y_1)(-\xi(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\cos(\theta_i) + \eta(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\sin(\theta_i) + \eta s\, \dot{\theta}_i \sin(\theta_i)) \\ &- \eta s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp}) - \eta s^2 \dot{\theta}_i \Big) ds, \end{aligned}$$

$$\begin{aligned} = \sum_{i=0}^{N} \Big[&- (x_i - x_1)(\xi s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\sin(\theta_i) + \eta s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\cos(\theta_i) + \eta \frac{s^2}{2}\, \dot{\theta}_i \cos(\theta_i)) \\ &- (y_i - y_1)(-\xi s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\cos(\theta_i) + \eta s(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\sin(\theta_i) + \eta \frac{s^2}{2}\, \dot{\theta}_i \sin(\theta_i)) \\ &- \eta \frac{s^2}{2}(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp}) - \eta \frac{s^3}{3} \dot{\theta}_i \Big]_0^{L_i}, \end{aligned}$$

$$
\begin{aligned}
= \sum_{i=0}^{N} & -(x_i - x_1)(\xi L_i(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\sin(\theta_i) + \eta L_i(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\cos(\theta_i) + \eta\frac{L_i^2}{2}\,\dot{\theta}_i\cos(\theta_i)) \\
& -(y_i - y_1)(-\xi L_i(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i)\cos(\theta_i) + \eta L_i(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp})\sin(\theta_i) + \eta\frac{L_i^2}{2}\,\dot{\theta}_i\sin(\theta_i)) \\
& -\eta\frac{L_i^2}{2}(\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp}) - \eta\frac{L_i^3}{3}\dot{\theta}_i.
\end{aligned}
\tag{2.25}
$$

Whilst this way of expressing (2.4) may seem long, when we add that

$$
\begin{aligned}
\dot{\mathbf{x}}_i \cdot \mathbf{e}_i &= \dot{x}_i\cos(\theta_i) + \dot{y}\sin(\theta_i), \\
\dot{\mathbf{x}}_i \cdot \mathbf{e}_i^{\perp} &= -\dot{x}_i\sin(\theta_i) + \dot{y}\cos(\theta_i),
\end{aligned}
\tag{2.26}
$$

we can easily see the linearity of the system with respect to $\dot{x}_i$, $\dot{y}_i$ and $\dot{\theta}_i$ for $i \in \{1, \ldots, N\}$. Furthermore, differentiating (2.1) we get the equations

$$
\begin{aligned}
\dot{x}_i &= \dot{x}_1 - \sum_{k=1}^{i-1} L_k\dot{\theta_k}\sin(\theta_k), \\
\dot{y}_i &= \dot{y}_1 + \sum_{k=1}^{i-1} L_k\dot{\theta_k}\cos(\theta_k).
\end{aligned}
\tag{2.27}
$$

These allow us to express any $\dot{x}_i$ or $\dot{y}_i$ for $i \in \{1, ..., N\}$ as a completely linear function of just $\dot{x}_1$ and $\dot{y}_1$. Consequently, this entails that the whole system of (2.11) and (2.12) is entirely linear in $\dot{x}_1$, $\dot{y}_1$ and $\dot{\theta}_i$ for $i \in \{1, \ldots, N\}$. We can therefore express system (2.5) in the form

$$
\begin{pmatrix} F_x \\ F_y \\ \mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z \end{pmatrix} = \mathbf{M}(\mathbf{x}_1, \ldots, \mathbf{x}_{N+1}, \theta_1, \ldots, \theta_N) \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}
\tag{2.28}
$$

where M is a $3 \times N+2$ matrix, requiring only the current values of x_i and y_i for $i \in \{1, \ldots, N+1\}$ and θ_i for $i \in \{1, \ldots, N\}$. We will define the exact form of this further on. However, as discussed, to control the swimming motion of a swimmer, we will alter $\dot{\alpha}_i$ instead of $\dot{\theta}_i$ for $i \in \{2, \ldots, N\}$. Consequently, we desire a reparameterized system of the form

$$
\begin{pmatrix} F_x \\ F_y \\ \mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z \end{pmatrix} = \mathbf{N}(\mathbf{x}_1, \ldots, \mathbf{x}_{N+1}, \theta_1, \ldots, \theta_N) \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \dot{\alpha}_2 \\ \vdots \\ \dot{\alpha}_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},
\tag{2.29}
$$

where $\mathbf{N}$ is again a $3 \times N + 2$ matrix. In order to achieve this, we can differentiate (2.2) to see

$$
\dot{\alpha}_i = \dot{\theta}_i - \dot{\theta}_{i-1}, \qquad \text{for } i \in \{2, ..., N\}.
\tag{2.30}
$$

Using this relationship, we can control $\dot{\alpha}_i$ and then efficiently calculate the $\dot{\theta}_i$ for $i \in \{2, \dots, N\}$ using the matrix $\mathbf{C}$ where

$$\begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \dot{\alpha}_2 \\ \vdots \\ \dot{\alpha}_N \end{pmatrix} = \mathbf{C} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \\ \vdots \\ \dot{\theta}_N \end{pmatrix}, \qquad \text{with } \mathbf{C} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 1 & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & -1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & 0 & -1 & 1 \end{pmatrix} \tag{2.31}$$

Furthermore, we can easily show that $\mathbf{N} = \mathbf{M}\mathbf{C}^{-\mathbf{1}}$, as

$$\mathbf{M} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_N \end{pmatrix} = \mathbf{M}\mathbf{C}^{-\mathbf{1}}\mathbf{C} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \\ \vdots \\ \dot{\theta}_N \end{pmatrix} = \mathbf{M}\mathbf{C}^{-\mathbf{1}} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \dot{\alpha}_2 \\ \vdots \\ \dot{\alpha}_N \end{pmatrix} = \mathbf{N} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \dot{\alpha}_2 \\ \vdots \\ \dot{\alpha}_N \end{pmatrix}. \tag{2.32}$$

We can now solve this system. We know the values for each $\dot{\alpha}_i$ and are only looking to calculate $\dot{x}, \dot{y}$ and $\dot{\theta}_1$. Therefore, it can be useful to partition $\mathbf{N}$ into two sub matrices: a 3×3 matrix $\mathbf{A}$ and a $3 \times N - 1$ matrix $\mathbf{B}$, according to

$$\mathbf{N} = (\ \mathbf{A} \mid \mathbf{B}\ ). \tag{2.33}$$

Here, $\mathbf{A}$ is called the "Grand Resistance Matrix" of a rigid system evolving at frozen shape (i.e. when all $\dot{\alpha}_i = 0$). It is symmetric and negative definite [13] and therefore always invertible. This partitioning of the $\mathbf{N}$ matrix allows us to write

$$\mathbf{N} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \dot{\alpha}_2 \\ \vdots \\ \dot{\alpha}_N \end{pmatrix} = \mathbf{A} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \end{pmatrix} + \mathbf{B} \begin{pmatrix} \dot{\alpha}_2 \\ \vdots \\ \dot{\alpha}_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \tag{2.34}$$

With a little rearranging, we are now finished. The solution of our system, given $\dot{\alpha}_i$ for $i \in \{2, \dots, N\}$, takes the form

$$\begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \end{pmatrix} = -\mathbf{A}^{-\mathbf{1}}\mathbf{B} \begin{pmatrix} \dot{\alpha}_2 \\ \vdots \\ \dot{\alpha}_N \end{pmatrix}. \tag{2.35}$$

2.2 Computational Methods for Calculating the N-matrix

In this section, we will show steps to compute the **N**-matrix, allowing us to calculate $\dot{x}_1$, $\dot{y}_1$ and $\dot{\theta}_1$ using equation (2.35). We will start by looking at equation (2.18), which we will split into equations (2.19) and (2.20). These two equations make up the first and second $N+2$ long rows of the matrix **N**. Using the equations in (2.26) we can re-factorise (2.19) and (2.20) as

$$\begin{aligned} F_x = \sum_{i=1}^{N} \quad & \dot{x}_i\big(-\xi L_i \cos^2(\theta_i) - \eta L_i \sin^2(\theta_i)\big) \\ & + \dot{y}_i\big(-\xi L_i \sin(\theta_i)\cos(\theta_i) + \eta L_i \sin(\theta_i)\cos(\theta_i)\big) \\ & + \dot{\theta}_i\big(\ \eta \frac{L_i^2}{2}\sin(\theta_i)\big), \end{aligned} \tag{2.36}$$

$$\begin{aligned} F_y = \sum_{i=1}^{N} \quad & \dot{x}_i\big(-\xi L_i \sin(\theta_i)\cos(\theta_i) + \eta L_i \sin(\theta_i)\cos(\theta_i)\big) \\ & + \dot{y}_i\big(-\xi L_i \sin^2(\theta_i) - \eta L_i \cos^2(\theta_i)\big) \\ & + \dot{\theta}_i\big(-\eta \frac{L_i^2}{2}\cos(\theta_i)\big). \end{aligned} \tag{2.37}$$

Therefore, setting

$$a_i = -\xi L_i \cos^2(\theta_i) - \eta L_i \sin^2(\theta_i), \tag{2.38}$$

$$b_i = -\xi L_i \sin(\theta_i)\cos(\theta_i) + \eta L_i \sin(\theta_i)\cos(\theta_i), \tag{2.39}$$

$$c_i = -\xi L_i \sin^2(\theta_i) - \eta L_i \cos^2(\theta_i), \tag{2.40}$$

we can form two $1 \times 3N$ vectors

$$\mathbf{P}_1 = \Big(\ a_1 \quad \cdots \quad a_N \quad | \quad b_1 \quad \cdots \quad b_N \quad | \quad \eta\frac{L_1^2}{2}\sin(\theta_1) \quad \cdots \quad \eta\frac{L_N^2}{2}\sin(\theta_N)\ \Big), \tag{2.41}$$

$$\mathbf{P}_2 = \Big(\ b_1 \quad \cdots \quad b_N \quad | \quad c_1 \quad \cdots \quad c_N \quad | \quad -\eta\frac{L_1^2}{2}\cos(\theta_1) \quad \cdots \quad -\eta\frac{L_N^2}{2}\cos(\theta_N)\ \Big), \tag{2.42}$$

which will allow us to rewrite equation (2.11) as

$$\mathbf{F} = \begin{pmatrix} F_x \\ F_y \end{pmatrix} = \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix} \begin{pmatrix} \dot{x}_1 \\ \vdots \\ \dot{x}_N \\ -- \\ \dot{y}_1 \\ \vdots \\ \dot{y}_N \\ -- \\ \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_N \end{pmatrix}. \tag{2.43}$$

Here, $\mathbf{P}_1$ and $\mathbf{P}_2$ combine to form a $2 \times 3N$ matrix. However, as shown in (2.28) and (2.29), we only want to include the values of $\dot{x}_1$, $\dot{y}_1$ and $\dot{\theta}_i$ for $i \in \{1, \ldots, N\}$. We don't want to include

$\dot{x}_i$ and $\dot{y}_i$ for $i \in \{2, \dots, N\}$, despite their presence in (2.36) and (2.37). To solve this we can employ the linear relationship between these variables, as shown in equations (2.27), to form a $3N \times N + 2$ matrix $\mathbf{Q}$, such that

$$\begin{pmatrix} \dot{x}_1 \\ \vdots \\ \dot{x}_N \\ -- \\ \dot{y}_1 \\ \vdots \\ \dot{y}_N \\ -- \\ \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_N \end{pmatrix} = \mathbf{Q} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_N \end{pmatrix}, \text{ where } \mathbf{Q} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & -L_1\sin(\theta_1) & 0 & 0 & \cdots & 0 \\ 1 & 0 & -L_1\sin(\theta_1) & -L_2\sin(\theta_2) & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ 1 & 0 & -L_1\sin(\theta_1) & -L_2\sin(\theta_2) & \cdots & -L_{N-1}\sin(\theta_{N-1}) & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & L_1\cos(\theta_1) & 0 & 0 & \cdots & 0 \\ 0 & 1 & L_1\cos(\theta_1) & L_2\cos(\theta_2) & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \cdots & 0 \\ 0 & 1 & L_1\cos(\theta_1) & L_2\cos(\theta_2) & \cdots & L_{N-1}\cos(\theta_{N-1}) & 0 \\ 0 & 0 & & & & & \\ \vdots & \vdots & & & \mathbf{Id}_N & & \\ 0 & 0 & & & & & \end{pmatrix}. \tag{2.44}$$

Here, $\mathbf{Id}_N$ represents an $N \times N$ identity matrix. This allows us to write equation (2.43) without $\dot{x}_i$ and $\dot{y}_i$ for $i \in \{2, \dots, N\}$ as

$$\mathbf{F} = \begin{pmatrix} F_x \\ F_y \end{pmatrix} = \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix} \mathbf{Q} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_N \end{pmatrix}. \tag{2.45}$$

We will now undertake the same process for equation (2.25), first re-factorising into the form

$$\begin{aligned} \mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z = \sum_{i=1}^{N} \quad & \dot{x}_i \Big(-(x_i - x_1)(L_i\xi\sin(\theta_i)\cos(\theta_i) - L_i\eta\sin(\theta_i)\cos(\theta_i)) \\ & \quad -(y_i - y_1)(-L_i\xi\cos^2(\theta_i) - L_i\eta\sin^2(\theta_i)) \\ & \quad +\eta\frac{L_i^2}{2}\sin(\theta_i)\Big) \\ & + \dot{y}_i \Big(-(x_i - x_1)(L_i\xi\sin^2(\theta_i) + L_i\eta\cos^2(\theta_i)) \\ & \quad -(y_i - y_1)(-L_i\xi\sin(\theta_i)\cos(\theta_i) + L_i\eta\sin(\theta_i)\cos(\theta_i)) \\ & \quad -\eta\frac{L_i^2}{2}\cos(\theta_i)\Big) \\ & + \dot{\theta}_i \Big(-(x_i - x_1)\eta\frac{L_i^2}{2}\cos(\theta_i) - (y_i - y_1)\eta\frac{L_i^2}{2}\sin(\theta_i) - \eta\frac{L_i^3}{3}\Big). \end{aligned} \tag{2.46}$$

If we then set

$$d_i = -(x_i - x_1)L_i \sin(\theta_i)\cos(\theta_i)(\xi - \eta) - (y_i - y_1)L_i\big(-\xi\cos^2(\theta_i) - \eta\sin^2(\theta_i)\big) + \eta\frac{L_i^2}{2}\sin(\theta_i),$$
$$e_i = -(x_i - x_1)L_i\big(\xi\sin^2(\theta_i) + \eta\cos^2(\theta_i)\big) - (y_i - y_1)L_i\sin(\theta_i)\cos(\theta_i)(-\xi + \eta) - \eta\frac{L_i^2}{2}\cos(\theta_i),$$
$$f_i = -(x_i - x_1)\eta\frac{L_i^2}{2}\cos(\theta_i) - (y_i - y_1)\eta\frac{L_i^2}{2}\sin(\theta_i) - \eta\frac{L_i^3}{3},$$

we can form a final $1 \times 3N$ vector, $\mathbf{P}_3$, where

$$\mathbf{P}_3 = \Big(\ d_1 \quad \cdots \quad d_N \quad | \quad e_1 \quad \cdots \quad e_N \quad | \quad f_1 \quad \cdots \quad f_N \ \Big). \tag{2.47}$$

Using $\mathbf{Q}$ again from (2.44), this will allow us to express equation (2.12) in the form

$$\mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z = \mathbf{P}_3\ \mathbf{Q} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_N \end{pmatrix}. \tag{2.48}$$

Subsequently, we can write the whole system of equations show in (2.4) as

$$\begin{pmatrix} \mathbf{F} \\ \mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z \end{pmatrix} = \begin{pmatrix} F_x \\ F_y \\ \mathbf{T}_{\mathbf{x}_1} \cdot \mathbf{e}_z \end{pmatrix} = \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{pmatrix} \mathbf{Q} \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \tag{2.49}$$

where $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$ form a $3 \times 3N$ matrix. This, finally, gives us the relation

$$\mathbf{M} = \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{pmatrix} \mathbf{Q}, \tag{2.50}$$

which allows us to calculate $\mathbf{N}$, using

$$\mathbf{N} = \mathbf{M}\mathbf{C}^{-\mathbf{1}}. \tag{2.51}$$

2.3 Numerical Errors

Now, with the matrix $\mathbf{N}$ calculated, we have all the tools to compute the derivatives: $\dot{x}_1$, $\dot{y}_1$ and $\dot{\theta}_1$. All we need to know is $\dot{\alpha}_i$ for $i \in \{2, \ldots, N\}$ at each point in time. However, even with these derivatives, solving this system of ODEs for x_1, y_1 and θ_1 can cause numerical errors. We can only estimate these values depending on a time step and approximate them numerically.

This estimation can lead to large numerical errors over time if not accounted for, as shown in Figure (2.3). This is a toy problem where we suppose we can calculate the derivative easily for any value of x, but do not know the real value of y. After only 3 steps our estimate has diverged significantly from the real curve and continues to diverge at an increasing rate. In this section, we briefly present the numerical scheme used to ensure acceptable numerical errors in our simulations before starting the learning process.

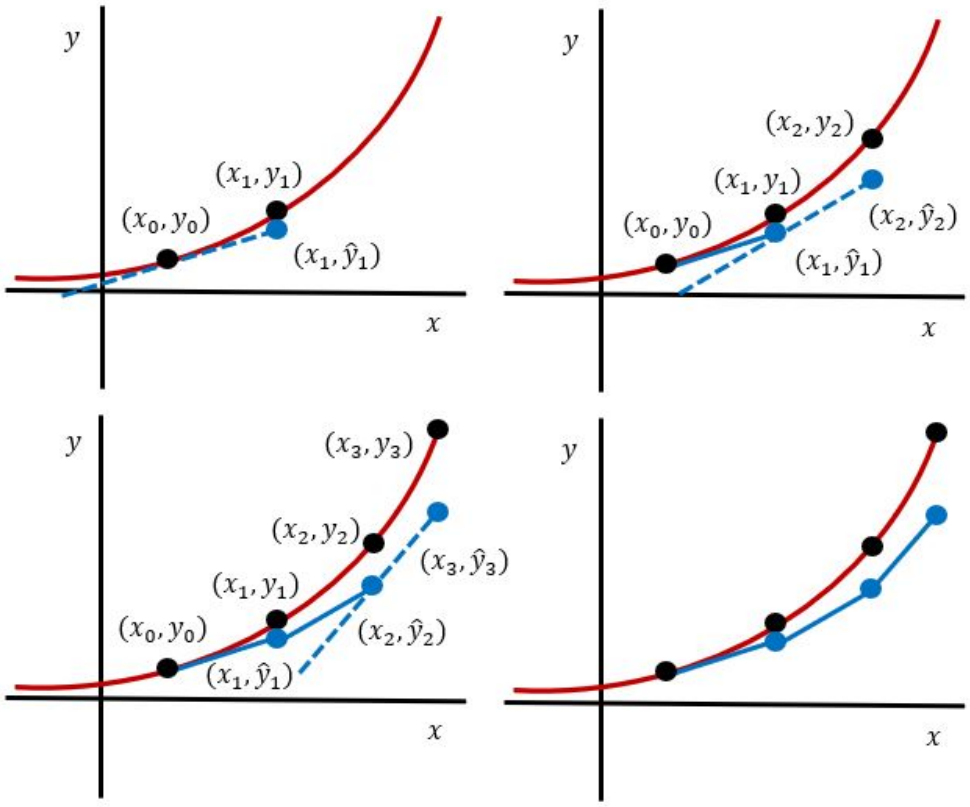

Figure 2.3: Here we show a toy-example of numerical errors building over time due to approximating the curve from the gradient. The first three panels show the first three steps to approximate y. The fourth shows the step-wise approximation (blue) to the curve given (red).

2.3.1 The Euler Method

The technique used in Figure (2.3) is called the Euler Method [14]. We assume that we have a scenario like ours, in which we can calculate the gradient at any point but we only know the initial coordinates (t_0, y_0). Here, y is an unknown vector function of the time t.

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y), \qquad y(t_0) = y_0. \tag{2.52}$$

Using a time-step, h, we can then calculate new values of y using the relationship [15]

$$y(t + h) = y(t) + h \cdot f(t, y) \tag{2.53}$$

In our case, y is a three dimensional vector, consisting of x_1, y_1 and θ_1. Therefore

$$\begin{aligned} x_1(t + h) &= x_1(t) + h \cdot \dot{x}_1(t), \\ y_1(t + h) &= y_1(t) + h \cdot \dot{y}_1(t), \\ \theta_1(t + h) &= \theta_1(t) + h \cdot \dot{\theta}_1(t). \end{aligned} \tag{2.54}$$

The Euler method is arguably the simplest way to integrate the differential equations but usually leads to large numerical errors over time, as shown on the toy problem. The error at each iteration is called the *local truncation error* and for Euler's method this error is of order $\mathcal{O}(h^2)$ [16].

2.3.2 The Classic Runge-Kutta Method

To reduce these numerical errors, we can also use more advanced methods to calculate the derivatives. We will now briefly introduce a particular Runge Kutta method, often called the classic Runge-Kutta or RK4, as a much more effective way of solving ODE's. In fact, Euler's method is itself a Runge Kutta method, however, RK4 has a much lower truncation error of order $\mathcal{O}(h^5)$ [16]. In order to calculate new values of y we use the relationship [15]

$$y(t + h) = y(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{2.55}$$

where

$$k_1 = f(t, y), \tag{2.56}$$
$$k_2 = f(t + 0.5h, y + 0.5 * h * k_1), \tag{2.57}$$
$$k_3 = f(t + 0.5h, y_n + 0.5 * h * k_2), \tag{2.58}$$
$$k_4 = f(t + h, y + h * k_3). \tag{2.59}$$

This is a more complex method but the advantages in performance merit it. We will now introduce some tests we can run in order to assess the accuracy of each technique.

2.3.3 Reciprocal Motions for Testing Numerical Errors

In this section, we will define two different reciprocal patterns of motion that should, according to the Scallop Theorem, cause no net motion for the swimmer. Any drifting motion for these swimmers over time would be a sign of numerical errors building.

Firstly, we will test a 3-link swimmer that stays still, apart from one end which moves back and forth at a constant speed, like a hand waving. This is a reciprocal motion so the swimmer

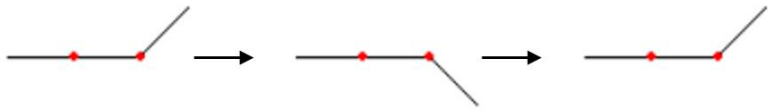

Figure 2.4: The motion of the 'Wave' test, simply moving the last link up and down.

should come back to exactly the same spot after one full cycle. If it doesn't, our simulation must have numerical errors building. This test swimmer's motion is shown in Figure (2.4).

Our second test will be a very simple 3-link model of a scallop itself. The swimmer will bend both its ends together, slowly, then quickly bend them back to straight. At a larger scale with inertia, this would cause net movement, but with no numerical errors it should have no displacement in our microscopic setting. This test swimmer's motion over time is shown in Figure (2.5).

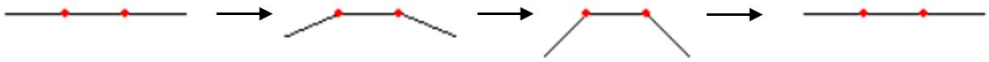

Figure 2.5: The motion of the Scallop swimmer. It is shown in four steps to emphasise the slow folding followed by the twice as fast re-straightening.

With these tests, we can now assess the accuracy of both the Euler method and the Classic Runge Kutta method in our microscopic simulation. If the simulation has no numerical errors, we would expect a completely flat line for the scallop swimmer and a looping curve that returns to $x = 0$ at the end of each iteration for the wave swimmer.

The results of these tests are shown in Figure (2.6). The top and bottom rows show the results using the Euler Method and the Classic Runge Kutta method respectively. From left to right, we simulated with time-steps of $\frac{1}{30}$s, 10^{-2} s and 10^{-3} s. At larger time-steps the error is severe, especially for the Euler Method. At $h = \frac{1}{30}s$, the scallop swimmer systematically drifts backwards and after 30 seconds finishes at a displacement of -4.5 µm. There is also a net displacement for the wave swimmer, although it is less severe. As expected, when we decrease the time-step the errors do decrease. However, even at very small time-steps, while the wave swimmer looks fairly stable, the scallop swimmer has drifted a noticeable distance with a displacement of -0.3 µm. Whilst we could decrease the time-step further, we have to balance the accuracy of the ODE solver with the computational complexity which rises as the time-step gets smaller. Looking at the bottom row, the results using RK4, there is a significant improvement from the Euler method. At each time-step there is significantly less drift. Secondly, the drift decreases drastically as the time-step decreases and at a time-step of 10^{-3} s is almost unnoticeable. At this time-step, after 30 seconds the scallop test swimmer only drifts -0.0809 µm away from 0.

We conclude that the classic Runge Kutta method gives accurate solutions to the equations of motion presented in section (2.1). We now move on to defining further swimming strokes focused on allowing the 3-link swimmers to move.

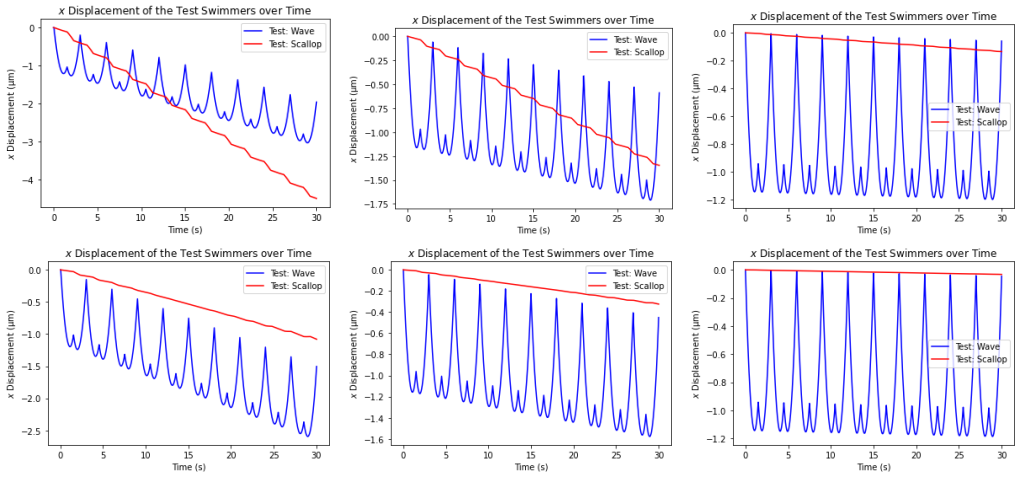

Figure 2.6: The x displacement of both test swimmers over 30 seconds. The top and bottom rows show the results using the Euler Method and the Classic Runge Kutta method respectively. From left to right, we simulated with time-steps of $\frac{1}{30}$s, 10^{-2} s and 10^{-3} s.

2.4 Reproducing the Purcell Stroke

In this section, we focus on simulating the Purcell swimming stroke. We have already come across this swimmer in Figure(1.3), proposed by Purcell in 1977, but now we will mathematically define it. Furthermore, we will introduce a generalisation of this stroke to N-links. For each of our swimmers let us denote the maximal angle excursion to be δ and define therefore that each $\alpha_i \in [-\frac{\delta}{2}, \frac{\delta}{2}]$. We will also denote the maximum angular velocity by v, such that each $\dot{\alpha}_i \in [-v, v]$. Here t will represent the time passed in seconds.

We can now define the Purcell swimmer, for $t \in [0, \frac{4\delta}{v})$, as the periodic stroke

$$(\alpha_2, \alpha_3) = \begin{cases} (-\frac{\delta}{2}, -\frac{\delta}{2} + vt) & \text{if } 0 \leq t < \frac{\delta}{v} \\ (-\frac{\delta}{2} + vt\ ,\ \frac{\delta}{2}) & \text{if } \frac{\delta}{v} \leq t < \frac{2\delta}{v} \\ (\ \frac{\delta}{2}\ ,\ \frac{\delta}{2} - vt) & \text{if } \frac{2\delta}{v} \leq t < \frac{3\delta}{v} \\ (\ \frac{\delta}{2} - vt, -\frac{\delta}{2}) & \text{if } \frac{3\delta}{v} \leq t < \frac{4\delta}{v} \end{cases} \tag{2.60}$$

Differentiating this, we get the values of $\dot{\alpha}$ throughout each stroke, with which we can use equation (2.35) to calculate the swimmer's motion. This periodic cycle repeats every $\frac{4\delta}{v}$ seconds.

$$(\dot{\alpha}_2, \dot{\alpha}_3) = \begin{cases} (0\ ,\ \ v) & \text{if } 0 \leq t < \frac{\delta}{v} \\ (v\ ,\ \ 0) & \text{if } \frac{\delta}{v} \leq t < \frac{2\delta}{v} \\ (0\ , -v) & \text{if } \frac{2\delta}{v} \leq t < \frac{3\delta}{v} \\ (-v,\ 0) & \text{if } \frac{3\delta}{v} \leq t < \frac{4\delta}{v} \end{cases} \tag{2.61}$$

Figure(2.7) shows the α traces for this stroke over time. Here, we have set $\delta = 1.5$ and $v = 1$. This leads to each cycle of the Purcell stroke taking 6 seconds as shown.

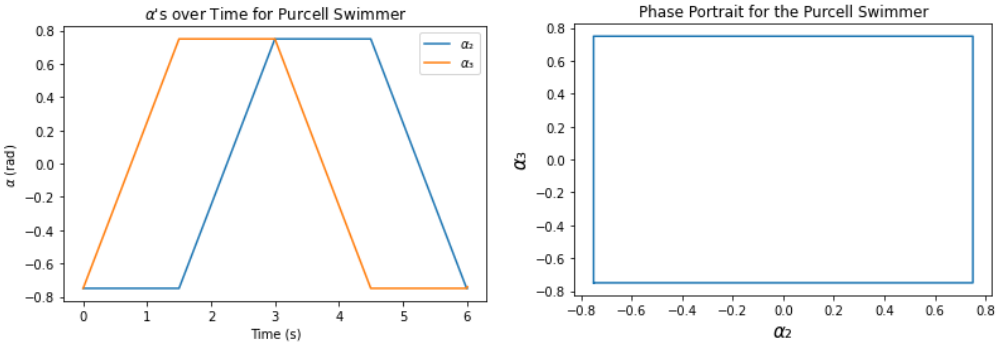

Figure 2.7: Left: The α_i values over time for one six second cycle. Right: The phase portrait for the Purcell stroke, plotting α_2 against α_3.

We now simulate a Purcell swimmer and record its x and y displacement over time. We take the measurements from the middle of the swimmer, halfway along the second link. This point is calculated using

$$x_{\text{middle}} = x_1 + L_1 \cos(\theta_1) + \frac{1}{2} L_2 \cos(\theta_2) \tag{2.62}$$

$$y_{\text{middle}} = y_1 + L_1 \sin(\theta_1) + \frac{1}{2} L_2 \sin(\theta_2) \tag{2.63}$$

For this simulation, we have again fixed $v = 1$ and $\delta = 1.5$. Additionally, we have set each link to a length of 20 µm, such that $L_i = 20$ for $i \in \{1, \ldots, N\}$. Looking at Figure(2.8), we can immediately see the periodic nature of the swimming, forming distinctive curves for each stroke. The displacement is not monotonous - it varies forwards and backwards through each stroke. Also plotted, on the right, we have the y-displacement over time. Whilst the swimmer does move up and down during each stroke, the net y-displacement is zero. Figure(2.9) plots the path of the swimmer in both the x and y dimensions, giving a better picture of how the motion looks in the 2D plane.

2.4.1 Analysis of the Influence of L and δ

We can now find the relationship between the individual link lengths and the overall x-displacement. Such a relationship is crucial to find the most efficient size for the swimmers. Furthermore, we can examine the effect of varying δ, the maximum angle excursion. It is reasonable to foresee that if this is too small, the swimmer will be almost unable to move, but if it is too large moving the links about so much may be inefficient. Both of these factors are crucial to our analysis and their individual effects on the overall displacement is important to understand.

Figure (2.10) shows the effects of sequentially increasing both L_i and δ. In the top pair of graphs we alter the link length and find its relationship with the x-displacement to be almost exactly linear. For every unit of length we add to each link, we get an overall extra displacement of roughly 0.5 units over 30 seconds. This equates to 0.1 units per stroke. Furthermore, as the links get bigger, not only does the net displacement increase, the amplitude of each stroke increases too. This linear relationship could be anticipated from the factor of L_i present when calculating $\dot{x}_i$.

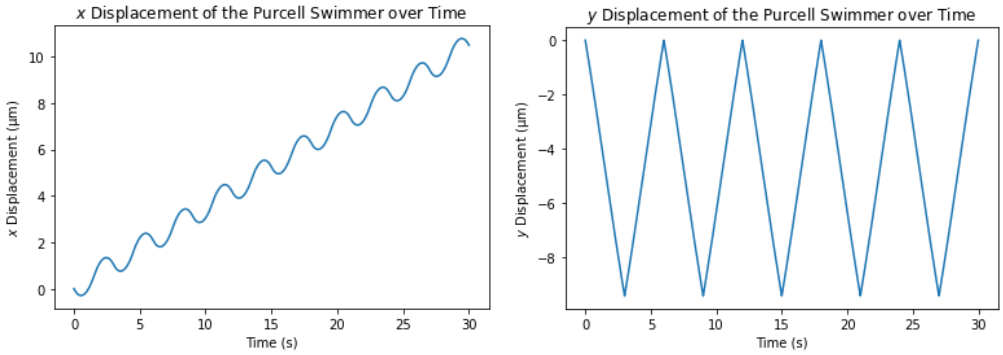

Figure 2.8: Left: The x displacement of the Purcell swimmer over time. Right: The y displacement of the Purcell swimmer over time. In both of these plots $L = 20$, $\delta = 1.5$ and the measurements are taken from the middle of the swimmer.

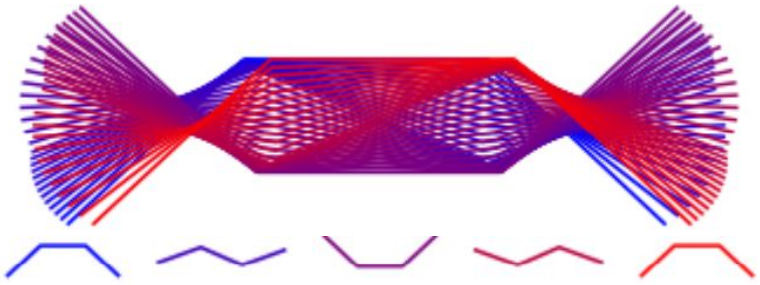

Figure 2.9: Top: A visualisation of one stroke for a Purcell swimmer, taking a snapshot of the position of the swimmer every 0.12 seconds and compiling them together. In blue we see the starting position of the swimmer, gradually turning to red as the stroke progresses. Bottom: 5 snapshots of the swimmer taken at 0, 1.5, 3, 4.5 and 6 seconds respectively. They each show one of the steps in Figure (1.3).

In the bottom pair of graphs, we now vary the maximum angle excursion. This time the relationship is not linear. If δ is very small, the displacement of the swimmer is also very small, and the relationship looks almost linear as we increase δ until $\delta = 1$. Beyond this, the increase in displacement drops drastically, in fact being detrimental beyond around $\delta = 2.5$. This may be due to the increasingly longer periods each stroke takes, where, as δ grows, the small increase in displacement per stroke no longer validates the extra time needed. Furthermore, we have to be wary that with large values of δ assumptions behind the RFT we are using begin to break down. RFT assumes that the only forces on the swimmer are caused directly by the viscosity of the liquid with no other effect but when δ becomes large and the links get closer together this could be compromised. For the remainder of this dissertation we will stick to $\delta = 1.5$, a balance between swimming efficiency and maintaining our model's validity.

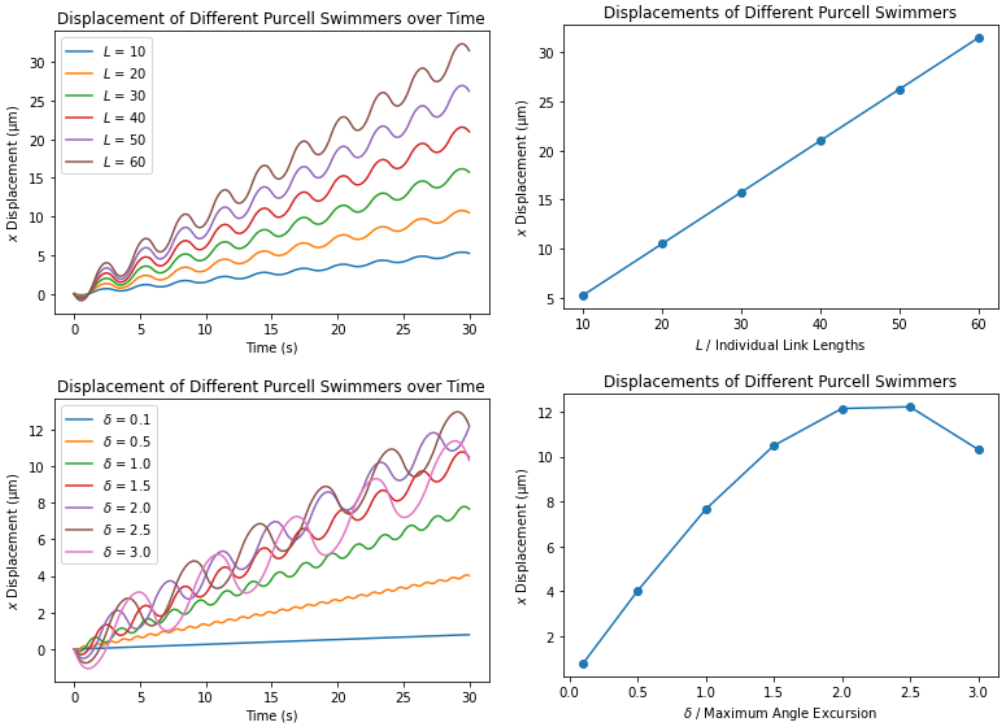

Figure 2.10: Top Left: The x displacements of six different swimmers over 30 seconds, each with different link lengths. Top Right: The relationship between the link length and the x displacement. Bottom Left: The x displacements of six different swimmers over 30 seconds, each with different values for ϵ. Bottom Right: The relationship between the maximum angle excursion, ϵ, and the x displacement.

2.4.2 The N-link Purcell Swimmer

In this section, we introduce a N-link generalisation of the 3-link Purcell swimmer. We define the $\dot{\alpha}_i$ values for this N-link Purcell swimmer as

$$\begin{pmatrix}\dot{\alpha}_2 & \dots & \dot{\alpha}_N\end{pmatrix} = \begin{cases} \begin{pmatrix}0 & \dots & 0 & v\end{pmatrix} & \text{if } 0 \le t < \frac{\delta}{v} \\ \begin{pmatrix}0 & \dots & v & 0\end{pmatrix} & \text{if } \frac{\delta}{v} \le t < \frac{2\delta}{v} \\ \quad\vdots & & \vdots & \\ \begin{pmatrix}v & 0 & \dots & 0\end{pmatrix} & \text{if } \frac{(N-2)\delta}{v} \le t < \frac{(N-1)\delta}{v} \\ \begin{pmatrix}0 & \dots & 0 & -v\end{pmatrix} & \text{if } \frac{(N-1)\delta}{v} \le t < \frac{N\delta}{v} \\ \begin{pmatrix}0 & \dots & -v & 0\end{pmatrix} & \text{if } \frac{N\delta}{v} \le t < \frac{(N+1)\delta}{v} \\ \quad\vdots & & \vdots & \\ \begin{pmatrix}-v & 0 & \dots & 0\end{pmatrix} & \text{if } \frac{(2N-3)\delta}{v} \le t < \frac{(2N-2)\delta}{v} \end{cases} \tag{2.64}$$

We can see that for a N-link swimmer, this extrapolation of the Purcell stroke now has a period of $\frac{(2N-2)\delta}{v}$ seconds. Figure (2.11) shows a trace of a 4-link Purcell swimmer through one full stroke. Here, the stroke takes 9 seconds as $\delta = 1.5$, $v = 1$ and $N = 4$.

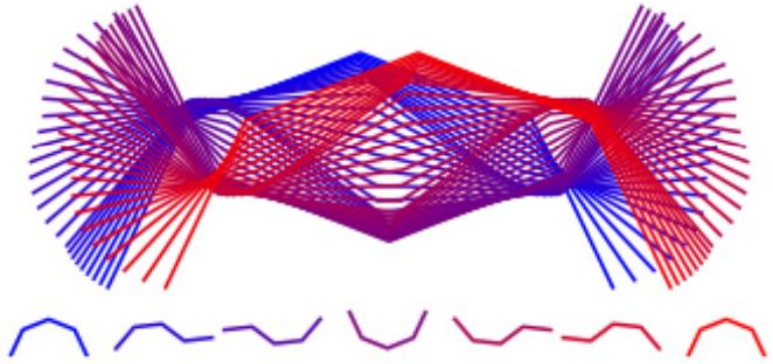

Figure 2.11: Top: A visualisation of one nine second stroke for a 4 - link Purcell swimmer. In blue we see the starting position of the swimmer, gradually turning to red as the stroke progresses. Bottom: 7 snapshots of the swimmer taken every 1.5 seconds.

We can now compare the effect of adding extra links to our Purcell swimmers. Figure (2.12) shows a simulation of multiple Purcell swimmers, each with increasing numbers of links. At first, as the number of links increases, the Purcell stroke becomes more effective, however, the curve seems to flatten. Once N reaches 6, the increase in displacement is minimal with the 7-link Purcell swimmer travelling only a fraction further the 6-link. It is very nearly less effective and this decrease in improvement shows the inefficiency in only moving one link at a time, causing large N swimmers to move slowly.

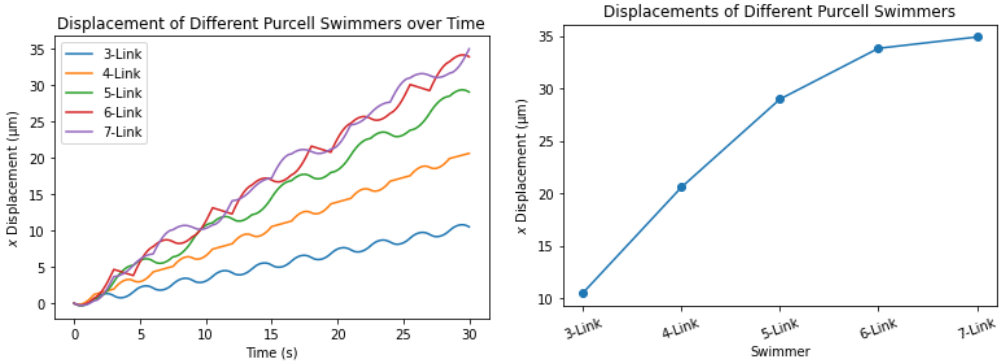

Figure 2.12: Left: The x-displacements of each swimmer over 30 seconds. Right: The final x-displacement of each swimmer after 30 seconds.

This inefficiency is clear in swimmers with more links but also implies that even at 3-links the Purcell stroke is far from optimal. So what strokes are optimal at 3-links and higher? With 3 links we can solve this by picturing this as an optimization problem, as in [2] which calculates the optimal stroke. However, for more than 3 links, the system becomes too complex for optimal control. It is interesting therefore, to see if for models with higher numbers of links we can construct effective strokes instead through reinforcement learning. Can the swimmers, entirely by themselves, find an efficient way to swim? In the next section we focus on reinforcement learning, and look into applying reinforcement learning techniques to our scenario.

Chapter 3

Reinforcement Learning

In this section we establish the concept of reinforcement learning and the techniques and terminology it relies upon. We focus upon its potential problems and techniques that can be taken to avoid them. We then explicitly introduce neuroevolution and neural networks, leading to the full definition of the NeuroEvolution of Augmenting Topologies algorithm (NEAT), the main focus of this chapter. This is the algorithm we will then use to train our swimmers.

3.1 Introduction to Reinforcement Learning

Reinforcement learning focuses on understanding the relationship between a learners actions and the environment around them and excels when using this relationship in order to complete goal oriented tasks [17]. Unlike other forms of learning, such as supervised and unsupervised learning, reinforcement learning does not need a dataset [18]. This makes it a lot more applicable for scenarios like ours where the swimmer must learn with no prior information. Reinforcement techniques have been used in many other similar situations [19] [20] [21]. With reinforcement learning, the swimmer is not told what actions to take or what structure to follow but instead must discover what works best for itself, learning through trial and improvement. Through this, a learner can be taught how to map different situations to different actions, sometimes producing unseen and better strategies than any human player could.

In order to discuss the framework of reinforcement learning we need to introduce notation and some crucial elements for a reinforcement learning system. In any *state*, s, a learner/*agent* can take a set of actions, a. In our case, the state is the position and speed of the swimmer, along with the environment around it. Its actions are to simply move each link up or down. Which actions to take are chosen by the agent's *policy*, which it learns over time. The policy is the core of reinforcement learning; it decides what to do in each situation and is developed with experience. The overall goal is to reach an efficient or optimal policy. Policies can be stochastic, offering a probability distribution for each action [17], however in this dissertation we will stick to deterministic cases.

Furthermore, we introduce the concept of a *reward signal* which summarises how well the task has been completed and therefore numerically evaluates the strength of the learnt policy. Often, the goal would be to maximise this value, i.e. to swim the furthest distance over a certain amount of time, however the goal could also be to minimise an error. In the case of our swimmers, a delayed reward will be used; we will assess the policy not constantly but after an established

amount of time. If a reward is given straight away, only immediately positive actions would be learnt, not long term strategies. For example, with our Purcell swimmer, the first movement of each stroke actually takes the swimmer slightly backwards and therefore would be negatively perceived if a reward was calculated at each time-step. Instead, allowing a delayed reward, say after ten seconds, allows the full stroke to develop, a stroke that we know gives positive motion and should be rewarded. Not giving a reward after every action, but instead only after multiple is called having a *sparse reward setting* [17]. Figure (3.1) shows a typical reinforcement learning scenario, where an agent takes an action and receives a new state and reward. In a sparse reward setting, this reward would be delayed until multiple actions are taken or a certain amount of time has passed.

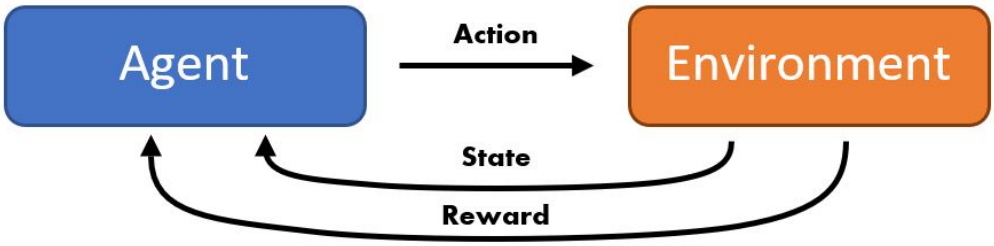

Figure 3.1: The reinforcement learning method. An action is taken then the new state is perceived from the environment and a reward is given.

However, when a reward is delayed it can cause some problems. How does the agent know which actions were actually relevant to the reward it received? This issue is called the *credit assignment problem* [22] and occurs in sparse reward settings. This problem can lead to the learning being very sample inefficient as the agent can learn incorrectly that certain actions are effective, which then take more samples to unlearn.

The traditional approach to solve this is using *reward shaping*; manually defining a reward function that guides an agent to the desired outcome. We could, for example, increase the reward given at the end for every second the swimmer moved. This could add more of a rationale for the agent to swim. There are some significant downsides to this, however. Reward shaping can suffer from alignment problems, where the agent finds unprecedented and undesirable ways to receive a high reward. It can in a way be seen as a counterpart to overfitting in supervised learning. The policy overfits to the exact reward function you have specified and consequently doesn't generalise well to the actual problem. As a result of this, reward shaping must be done with much care.

Moreover, in order for an agent to learn through reinforcement learning, we must allow the agents to try many different actions, and run through the scenarios multiple times. Often, this is not physically attainable or even possible and so we rely on building a model and simulating the conditions instead of using empirical data [23]. The inevitable slight differences between the model produced for our simulation and the real world is called the *reality gap*. This can be quite a large problem, especially if the model is weak; the strategies learnt by the agent in the model, may not work in the real world. In our case, we will use all the theory we have introduced previously to model the swimmer and simulate its movement through its environment. This, however, will always still just be an approximation. Realistically, there will always be a slight

reality gap, but with a good model what is learnt will still be useful.

In reinforcement learning, an agent must also find a balance between exploration, trying new actions, and exploitation, taking advantage of what it already knows to work. For optimal policies to be found, there must be an element of both. An agent must prefer previously positive actions that gave a high reward, but also have a willingness to experiment and find better methods. Emphasising just one side of this balance would be ineffective. Imagine a swimmer in our microscopic setting. With no exploration, it might learn that staying still is optimal, compared to another motion that haphazardly drifts backwards. It would be happy to conclude that this is the best option and not try anything else. On the other hand, with too much exploration it could try lots of swimming styles but never properly take advantage of those that work. A trade-off must be struck. Figure (3.2) shows a toy example of two algorithms trying to find the global maximum of a curve. Neither algorithm can see the full curve but can just choose points to evaluate along it to build an estimate. One only explores, while the other only exploits. One technique is inefficient without the other, the first finding a smaller maximum but not the global, and the second randomly guessing points. As a result neither find the global maximum.

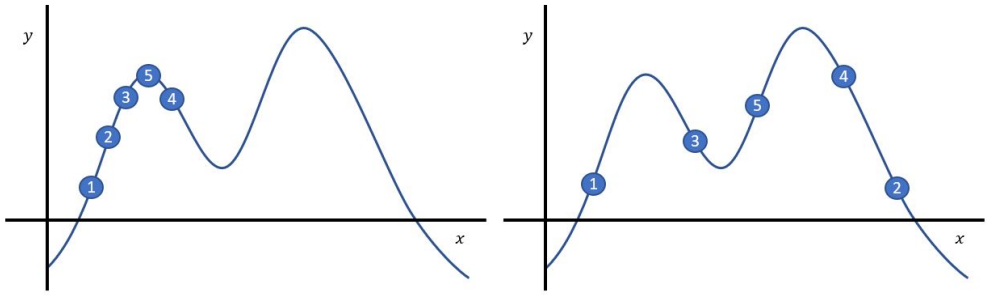

Figure 3.2: Here we see two different toy algorithms attempting to find the global maximum of a function. On the left we have an exploitative approach and on the right just exploration.

For some methods, the policy operates using a value function, which numerically estimates the advantage of taking each different action. If the policy always chooses the action that it currently thinks is the best, this is called a greedy policy. This can lead to the policy being too exploitative. We can introduce more exploration by behaving greedily most of the time, but occasionally, with small probability, ϵ, selecting an action at random, uniformly, independent of the policy. This sort of policy is called ϵ-greedy [17]. However, in this dissertation, we will not be focusing on this type of technique using value functions and instead find this exploitation exploration balance by using a *genetic algorithm*.

Genetic algorithms apply multiple static policies, which are defined before the simulation, each interacting with a separate instance of the environment. The most successful policies, the policies that receive the highest rewards, are then carried over to the next 'generation'. In this next generation, multiple slight variations of each successful policy are created and then applied in the environment again with the most successful being passed on. This process then repeats allowing increasingly successful policies from each generation to be passed to the next much like genes in nature, allowing the policy to iteratively evolve. This analogy gives these methods their name, as much like in biology, with evolution skilled, intelligent organisms are formed even if they do

not learn in their individual lifetimes. We will be focusing on a specific genetic algorithm called the NeuroEvolution of Augmenting Topologies algorithm or NEAT.

3.2 NeuroEvolution

Neuroevolution is the process of evolving artificial neural networks using genetic algorithms, classically while maintaining their architecture or topology [24]. It has been found to be very effective in reinforcement learning tasks and is built upon in [25] which introduces the algorithm 'NeuroEvolution of Augmenting Topologies' or NEAT: the main focus of this chapter. Before we dive into this, however, will we first spend some time introducing artificial neural networks and the role they play in our reinforcement learning scenario.

In neuroevolution, we use a neural network as our policy, taking our agent's state as its input and taking the chosen action as its output. Neural networks can be an effective class of decision making systems to evolve given their capability of representing solutions to many different kinds of problems as well as their ability to serve as universal function approximators [26]. They also allow continuous variables as inputs which is important for us, allowing us to easily represent the multidimensional and continuous states of our agent. Because neuroevolution searches for a behavior instead of a value function, it is effective in problems with continuous and high-dimensional state spaces [27]. Furthermore, in our case, having continuous outputs for the neural network is also useful, improving our ability to precisely control the swimmer.

3.2.1 Neural Network Topology

Figure (3.3) shows a neural network with 3 inputs, 2 hidden layers each with 4 hidden units and 2 outputs. Furthermore, this network is feed-forward; each node only affects nodes further on in the network and they have no way of affecting node values from previous layers [28]. This network is also dense, each layer of nodes is fully connected by weights to the next layer of nodes. This is called the networks architecture or topology, and is actually fairly arbitrary. There is currently little theory behind the structure of neural networks and so often finding an optimal topology is trial and error, despite potentially having a large effect.

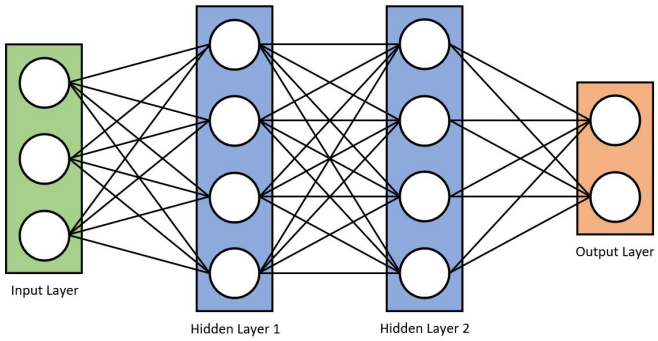

Figure 3.3: Here, we have a simple neural network with 3 inputs, 2 hidden layers each of width 4 and finally 2 outputs. The weights are shown as the black lines connecting each node.

In supervised learning, a network too small may not be able to model the data, but a model too big might take too long to train and you can run the risk of overfitting [29]. In reinforcement learning you will face similar problems, too small of a network may be unable to produce a strong policy, but too large may never find one due the time taken to train. The exact topology of the optimal neural network is hard to define and likely different for any problem.

3.3 NeuroEvolution of Augmenting Topologies

NEAT is an example of a Topology and Weight Evolving Artificial Neural Network (TWEANN) [27]. Further examples include [26] [30]. It not only evolves its weights through neuroevolution but also the network's topology. There is some controversy behind this approach though as evolving topology could potentially over complicate the search [27]. We are not only searching for the optimal weights but also permuting the entire structure. In traditional neuroevolution approaches, one would define an exact network topology before any evolution takes place. Thus, the sole goal of fixed-topology neuroevolution is simply to optimize the connection weights that determine the functionality of a network [27]. The crux of our issue, however, remains: could also evolving the topology be advantageous? It is known that a fully connected network can in principle approximate any continuous function [31], so why bother augmenting the topologies?

Despite this, as discussed before, the topology of a neural network will hugely affect its functionality. In fact, in supervised learning, adjusting the network architecture has been shown to be an effective technique [32]. Looking at it from another angle, in [33] it is argued that evolving structure automatically saves the time spent by humans considering which exact topology to use. It prevents the necessary training of multiple models with different topologies in order to test the assumptions behind the architecture. Interestingly, in [25] it is shown through ablation that evolving the topology is significantly advantageous. Given 1000 generations to find a solution to a pole balancing problem, only 20% of the time the non-topology evolving algorithm could find a solution, compared to 100% with the full NEAT algorithm. Additionally, it took 8.5 times more evaluations on average. This is strong evidence that TWEANNs and NEAT especially are efficient and evolving the topology is a powerful and necessary tool.

3.3.1 Representation, Mutation and Crossover of Different Topologies

In order for NEAT to mutate the structure of networks, it needs an efficient way to record the topologies within the generations. Instead of using a direct encoding scheme, which would specify every connection and node, it uses an indirect encoding [25]. This instead only specifies rules for constructing the neural network, not every detail. This is a much more compact representation as although one can derive every connection and node from the encoding, not every detail has to be explicitly specified. Continuing the genetic metaphor, the indirect coding is called the genome, and the neural network produced from it the phenotype. Figure (3.4) shows an example of a genome that consists of a set of node genes and connection genes. Each node gene specifies the type of node present, whether it is an input, an output or a hidden node. Each connection gene represents a weight within the network and specifies the in-node, the out-node, the weight itself, if the gene is actually expressed and finally an innovation number, which we will introduce further on.

Now, during NEAT, mutation can happen in multiple different ways. It can change both the connection weights and the network topology. Focusing first on mutating existing weights, NEAT works like any other neuroevolution system. Each weight is mutated with a preset probability

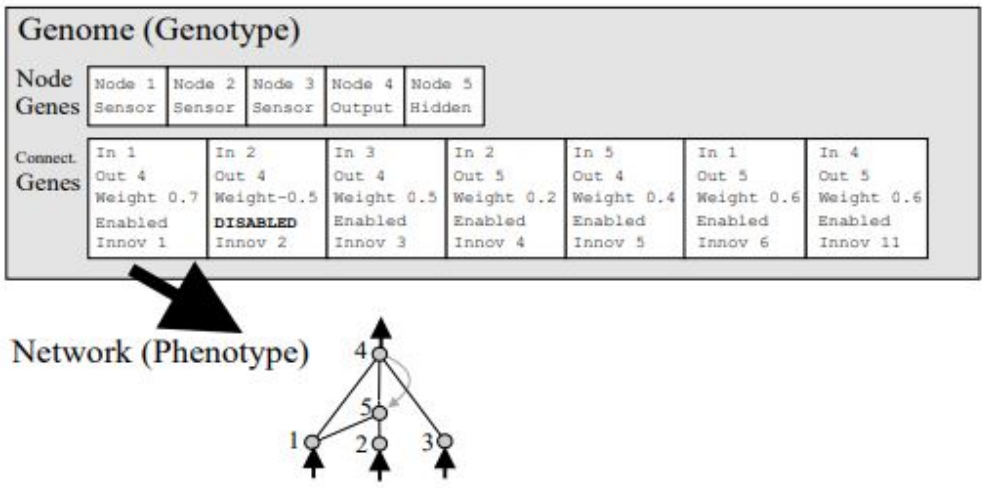

Figure 3.4: This shows an example of a phenotype, or neural network, constructed using the genome or genotype. Here we have 3 input nodes, 1 output node and 1 hidden node between them. We have 7 different connection genes, one of them recurrent, although the second is disabled, meaning it is not expressed in the phenotype. This figure is extracted from [27].

called the mutation rate, p_m. If a weight, w, is mutated, we alter it using

$$w^* = w + z, \quad \text{with } z \in N(0, \sigma_m^2) \tag{3.1}$$

The old weight, w, is then replaced by the new weight, w^*. Here, σ_m is called the mutation power. Alternatively, with another preset probability called the replace rate, p_r, the weight can be completely rejected, replacing it with w^* where

$$w^* = z, \quad \text{with } z \in N(\mu_w, \sigma_w^2). \tag{3.2}$$

Here μ_w and σ_w^2 are the Initialisation Mean and the Initialisation Standard Deviation values respectively. The exact same set of techniques are taken for mutating the existing biases.

The structural mutations can be split into two main sections: those that add or delete weights and those that add or delete nodes. Firstly, with probability, p_c^+, a new connection is added and with probability, p_c^-, an existing connection is removed. When a new connection is added, the two nodes it connects are chosen uniformly at random, and a new weight is initialised from $N(\mu_w, \sigma_w^2)$. When a connection is removed, it is marked as disabled in the genotype, therefore no longer appearing in the network. Secondly, just like when adding and removing connections, adding and removing nodes happens with preset probabilities p_n^+ and p_n^- respectively. If a node gene is added, a random connection is split, making way for the new node in the middle. This helps avoid nodes from being made but unconnected and unused. Both new connections will take the same weight as the old connection, which is now marked as disabled, hopefully keeping the function of the network fairly similar at first. Finally, if a node is deleted, the node gene is removed and all connections to it are disabled. Adding a new connection and node is shown in Figure (3.5).

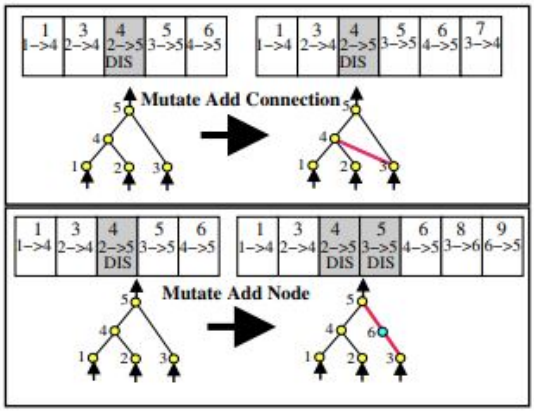

Figure 3.5: The addition of both a connection and a node to an existing network. This figure is extracted from [27].

However, before the mutation process, at the end of each generation a new generation must be produced from the previous. To do this, NEAT uses an efficient technique called crossover. Firstly, it chooses a select number of the individuals from the previous generation with the highest reward, or fitness to continue our evolutionary metaphor, and discards the rest. The proportion chosen is called the survival threshold, τ, and is analogous to survival of the fittest in nature – only the strongest from each generation can produce the next. It then selects pairs of parents at random with each pair producing a child network for the next generation using crossover. This process continues until the population of the next generation is full.

In Figure (3.5), each connection has its own innovation number, unique to that connection. For every mutation a new innovation number is produced and no two innovation numbers are the same. This records the full historical ancestry of each gene. If two parents differ by a gene, this gene is said to be excess if the gene occurs outside the range of the other parents innovation numbers and disjoint if not. When crossover occurs, the matching genes are always given straight to the child but the excess and disjoint genes are taken just from the fitter parent. If the parents have matching fitness, they are taken randomly from each. This method of crossover allows two different networks to be compared easily and efficiently without intricately examining the networks topology. Figure (3.6) shows an example of crossover between two parent networks to produce a child network for a new generation.

3.3.2 Protecting Innovation through Speciation

Often, when first mutating a new node or new connection, it will have a negative effect on the overall function of a network. This addition is random and newly initialised, not fine tuned to the network. This can mean that individuals with newly mutated connections or nodes are ineffective in the initial generation and therefore not selected to produce the next. This can happen even if in the long run the mutation would actually be really constructive; the network is not given enough time to develop.

NEAT solves this problem through speciation. It aims to keep networks with similar topologies and weights together in "species". A top proportion of each species is always saved for the

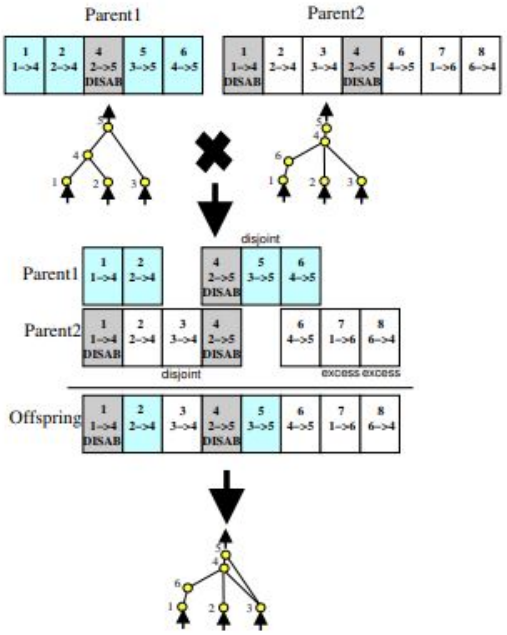

Figure 3.6: The crossover process in action. In this case we assume that both parents are equally fit. This figure is extracted from [27]

next generation and so, even if a new topological innovation is initially inefficient it is allowed to develop protected before having to compete with the rest of the population. The amount preserved and saved from each species is called the elitism, ϕ. This method of speciation is used in other areas to promote and protect diverse approaches, such as optimizing multimodal functions [34]. After a while, however, if a species shows no sign of improvement it is said to be stagnating and can be removed. The number of generations this takes is called the max stagnation, Λ. Normally, it is important to preserve the best species, no matter how long they stagnate, so as not to lose the most effective policy. The number of species we protect is called the species elitism, Φ.

So, how can we define species? Once again, we can take advantage of the innovation numbers to calculate the "genetic distance", Δ, between two individual neural networks. We will count the number of excess, E, and disjoint, D, genes along with the average weight difference of matching genes, $\bar{W}$. We can then calculate

$$\Delta = \frac{c_1 E}{N} + \frac{c_2 E}{N} + c_3 \cdot \bar{W}, \tag{3.3}$$

where, here, N is the number of genes in the larger genome used as normalisation. The coefficients c_1, c_2 and c_3 are parameters we control to weight the importance of each factor. We will generally keep c_1 and c_2 the same. We will also define the compatibility threshold, Π. If the genetic distance, Δ, from to an individual to a member of a species, chosen uniformly at random, is below this threshold, it joins this species. We place each genome in the first species where this is the case, it cannot join two. If all other species are too distant, it creates a new one.

In order to stop one species taking over the entire population, NEAT also uses explicit fitness sharing to decide which species can grow [27]. To calculate this, we take the original fitness of each individual and divide it by the total number of individuals in its species, which we will represent by N_j for species j. We call this the adjusted fitness, where $f_{i,j}$ represents the adjusted fitness for individual i in species j. We can then calculate the size of each species in the next generation using

$$N'_j = \frac{\sum_{i=1}^{N_j} f_{i,j}}{\bar{f}}, \tag{3.4}$$

where N_j represents the number of individuals in species j, N'_j represents the new number of individuals in species j in the next generation and $\bar{f}$ is the mean adjusted fitness.

3.3.3 The NEAT Method

In order to run the NEAT algorithm, a number of hyper parameters need to be set. This includes the number of generations that the algorithm should run for, Ω, and the population size of each generation, ω. We also need to define the initial network structure: the number of inputs, n_i, the number of outputs, n_o, the number of hidden nodes, n_h, and the initial connections between them. It is customary to start with a fully connected network with no hidden nodes. This keeps the initial topology simple, reducing dimensionality, whilst also allowing a quick start to the process with no nodes unconnected and wasted [25]. After this set up, we take the following steps:

1. Create a first generation, size ω, of newly initialised fully connected networks. Each weight, w is sampled from $w \in N(\mu_w, \sigma_w^2)$.
2. Simulate this generation and record each individual's fitness.
3. Sort the individuals into species and calculate the population of each species for the next generation using the adjusted fitness.
4. Select the proportion of individuals with the highest fitness, τ, and discard the rest.
5. Use crossover with these individuals to produce a new generation of networks.
6. Mutate each of these network's structure and weights.
7. Simulate this generation and record each individual's fitness.
8. If the total number of generations has been reached, Ω, return the network with the highest fitness. Else, return to step 3.

In the next section, we will now apply NEAT to our microscopic swimming scenario, training swimmers to move entirely by themselves. We will focus on precisely defining the scenario, including determining what defines our agent's state and the exact reward function used.

Chapter 4

Applying NEAT to the N-link Model Swimmers

In this chapter we will focus on applying the NEAT algorithm directly to our microscopic swimmers. We will discuss the exact methods taken in order to efficiently produce effective swimmers and, moreover, the issues found during the process. Firstly, we will look into the relationship between the neural network and the swimmer and focus on how each policy works. We will then define our reward function and discuss the issues of reward shaping. Finally, we will present the complete application of NEAT for our swimmers including the setting of hyper-parameters for during the algorithm.

4.1 Constructing our Reinforcement Learning Scenario

As discussed previously, we are using a neural network as our agent's policy. As input it will take the state of our swimmer and then output a signal to direct the required action. In order to do so, we must first consider how we define the state of our N-link swimmer/agent. Secondly, we must define the relationship between the actual output of our network and the action taken by the swimmer. As discussed before, we will control the swimmer by changing the values of each $\dot{\alpha}_i$, however, how the outputs of the network directly relate to these needs to be defined. Finally, we will define our reward function, which assesses the fitness of each individual in a generation.

4.1.1 Defining the State

For a N-link swimmer, we will use $3N-2$ variables to fully describe its state: θ_1 and α_i, $\dot{\alpha}_i$, λ_i for $i \in \{2, \dots, N\}$. Here, we define a set of $N-1$ binary variables, $\lambda_i \in \{-1, 1\}$ for $i \in \{2, \dots, N\}$, which record the last side each link hit. Each neural network constructed must therefore have $3N-2$ input nodes. We have chosen these variables to define our state in order to increase the efficiency of the learning process and furthermore avoid introducing unnecessary biases.

In our scenario, the state of our swimmer is its condition and orientation, we don't need any further sensors from the environment. An effective starting place to describe a N-link swimmer is recording the $N+2$ key variables: x_1, y_1, θ_1 and α_i for $i \in \{2, \dots, N\}$. However, the variables x_1 and y_1 shouldn't be included in the swimmers state. Our swimmer should swim the same, no matter its position in the 2D plane, and therefore its 2D coordinates are irrelevant to the policy

and could induce bias. The variable time, t, is omitted as an input for similar reasons. We could use a modulated version of time as input, however, this could lead to a bias when forming the stroke duration and so it is also omitted. The rest of the variables, however, are useful and will be kept.

However, sometimes different actions are needed even if the current position and orientation of the swimmer is the same. To solve this, we also include the value of each $\dot{\alpha}_i$ for $i \in \{2, \ldots, N\}$ to record the current movement. This allows our network to understand the previous motion of the swimmer and act accordingly. Additionally, the swimmers can still often get stuck once they reach the full extension of each link due to the initial networks being fairly simple. This warrants the inclusion of λ_i $i \in \{2, \ldots, N\}$. Each $\alpha_i \in [-\frac{\delta}{2}, \frac{\delta}{2}]$ and so every time α_i hits the boundary of these constraints, we simply multiply λ_i by -1. This gives a distinct change to the network input when the link reaches its maximum or minimum values, inspiring more movement.

4.1.2 Defining the Actions

We will use one output from the network to control each $\dot{\alpha}_i$. We are using the sigmoid function as our activation function and so the range of each output is $[0, 1]$. We will use this continuous output to fully control our swimmers. We first multiply each output by by $2v$ then subtract v, giving us the new range $[-v, v]$, where v is again the maximum angular velocity. We then take each of these transformed outputs as the new value of $\dot{\alpha}_i$ at this point in time. Through this, we can fully control our swimmer with only $N - 1$ outputs, avoiding making the network too complex. Figure (4.1) shows what the starting network would look like for a 3-link Swimmer.

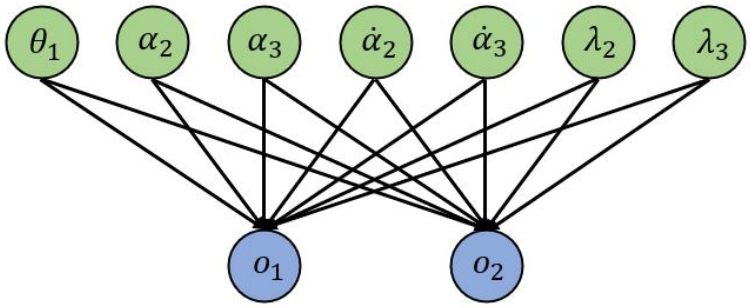

Figure 4.1: A diagram of the neural network policy for a 3-link swimmer to be trained using NEAT. It has 7 inputs, marked in green, and 2 outputs marked in blue. The outputs then control the swimmer using $\alpha_i = 2v \cdot o_{i-1} - v$ for $i \in \{2, 3\}$.

Originally, we experimented with having three different outputs controlling the value of each $\dot{\alpha}_i$. One output would tell the $\dot{\alpha}_i$ to increase at rate v, one to decrease with rate v and the other to stay constant. The output with the highest value from the network would be chosen as the action at each point in time. This concept was fueled by the idea that there would never be any benefit to the link rotating slower than v, so it made sense to use step-wise control for it. In reality, however, this led to over parameterized networks and slow learning.

4.1.3 The Reward Function

In this project we will define our reward function as the x-displacement travelled by each swimmer after 30 seconds. We will measure this distance from the end point of each swimmer, the variable x_{N+1}. Our fitness function is therefore

$$f = x_{N+1}^{(\mathrm{t}\,=\,30)} - x_{N+1}^{(\mathrm{t}\,=\,0)}. \tag{4.1}$$

In theory, we could set a very short period of time instead of 30 seconds, just enough for one stroke to develop. This, however, leads to some problems. The swimmers can almost overfit to this time period, learning to propel themselves as far forward as possible until the time runs out, and not worry about continuing swimming after that. This technique tends to teach the swimmers ineffective techniques in the long run, instead prioritising immediate movement to long term efficiency. Moreover, using a longer period of time allows effective swimming techniques time to flourish and stand out from others.

Originally, the fitness function took measurements from the middle of the swimmer, prioritising the swimmer to move its middle as far forward as possible as this is the point we plot when we visualise the swimmer's displacement. This worked effectively in smaller N-link swimmers, but started to break down as the swimmers became larger. The larger swimmers, with more range of movement, started bending their strokes so that towards the end of the 30 second window, the middle of the swimmer was further forward. Figure (4.2) shows an example of such a stroke. Here a 4-link swimmer was swimming fairly straight at 30 generations, but after 60 developed a rotation in order to move the middle further. Also shown is this same pattern getting more extreme with higher N. This is obviously not desirable. Using instead the end point, x_{N+1}, solves this problem but unfortunately causes swimmers trained this way to have very slightly less effective strokes at moving the middle forward. This can lead to them looking less efficient when we look at their plotted x-displacements. Further experimentation of instead using a weighted midpoint of the middle and end led to similar problems as using the middle, where swimmers bent but to a lesser degree. An interesting possibility for future work would be to penalise excessive y-displacement after 30 seconds or penalise the swimmer if the average θ_i value grows too large indicating turning. This, however, would have to be carefully applied in order to avoid the alignment problem.

4.2 Configuration of NEAT

Before we can run NEAT, we must decide the initial orientation of each swimmer before the simulation starts. For a Purcell swimmer, we already know its stroke and therefore can place it in a starting position that fits this swimming pattern. With our learners, we do not already know what positions they will take throughout their stroke and so we can't initialise them into it. Our solution is to start every swimmer flat with $\theta_i = 0$ for $i \in \{1, \ldots, N\}$. This adds no bias to what stroke the agent will learn. It does, however, complicate the learning process as the swimmer must not only learn a strong stroke, but also learn how to get into that stroke pattern from flat which may not occur during its normal stroke. This period before it converges to a set stroke we will call the initialisation period.

Furthermore, for our training we shall set $\delta = 1.5$, $v = 1$ and $L_i = 20$ for $i \in \{1, \ldots, N\}$ unless otherwise stated. However, the network does not automatically know not to overextend the angles. This can lead to problems where α_i is at its maximum value, but the policy advises

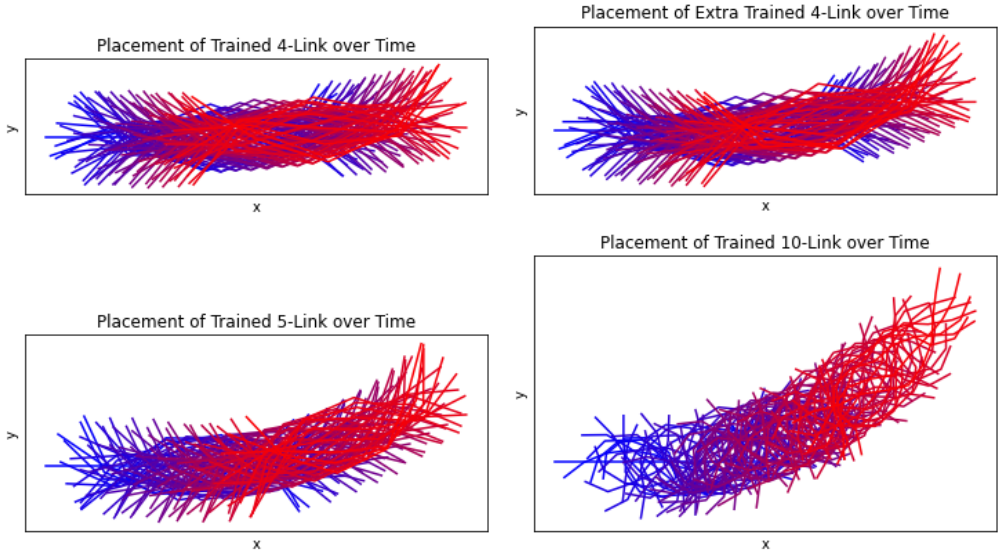

Figure 4.2: Top: A trained 4-link swimmer after 30 then 60 generations. We see it develops more of a bend, moving its middle further forwards. Bottom: A trained 5-link swimmer and a trained 10-link swimmer. We can see that the problem is exacerbated as N gets bigger.

it to keep expanding. We solve this problem by simply ignoring the network output and setting $\dot{\alpha}_i = 0$ if otherwise $|\alpha_i| > \frac{\delta}{2}$. This way we never overextend our angles even if the policy suggests that we should.

Finally, we will consider the preset parameters for the NEAT algorithm. The values for these parameters are shown in the table below. Note that we change the number of inputs and outputs depending on the value of N, the number of links in the swimmer. The rest of the hyperparameters are chosen in order to allow a balance of exploration and exploitation whilst also creating enough speciation to allow innovation to thrive. In the next section, we will analyse the results from this reinforcement learning process.

Variable	Notation	Value
Population Size	ω	100
Number of Generations	Ω	60
Number of Input Nodes	n_i	3N - 2
Number of Output Nodes	n_o,	N - 1
Number of Hidden Nodes	n_h	0
Weight and Bias Initialisation Mean	μ_w	0
Weight and Bias Initialisation Standard Deviation	σ_w	1
Weight and Bias Mutation Power	σ_m	0.5
Weight and Bias Mutation Rate	p_m	0.7
Weight and Bias Replace Rate	p_r	0.1
Add Connection Probability	p_c^+	0.5
Delete Connection Probability	p_c^-	0.5
Add Node Probability	p_n^+	0.2
Delete Node Probability	p_n^-	0.2
Compatibility Excess Coefficient	c_1	0.9
Compatibility Disjoint Coefficient	c_2	0.9
Compatibility Weight Coefficient	c_3	0.7
Compatibility Threshold	Π	3.0
Elitism	ϕ	2
Species Elitism	Φ	1
Survival Threshold	τ	0.2
Max Stagnation	Λ	10

Chapter 5

Analysis of Results

In this section, we will evaluate the results from the NEAT algorithm. These are analysed in two sections. The first section will focus purely on the trained 3-link swimmer, comparing it to the 3-link Purcell swimmer and assessing the influence of δ. Furthermore we will compare this stroke to the optimal stroke calculated through optimal control theory [2]. The second section will concentrate on swimmers with a larger number of links. We will analyse the stroke patterns of each of these swimmers and compare the learnt stroke's efficacy both compared to the N-link Purcell swimmer and each other.

5.1 The Trained 3-link Stroke

In this section, we analyse the 3-link swimmer trained using the NEAT algorithm. Figure (5.1) shows the displacement over time for this swimmer, compared to the 3-link Purcell swimmer. The NEAT swimmer has developed a faster stroke covering 12.8 µm in 30 seconds, whereas comparatively the Purcell swimmer only travels 10.5 µm. Once again, the displacement graph is not monotonous and so the learning is favoured by the delayed reward function. Interestingly, the trained 3-link swimmer adapts again to a repeated stroke, but this time with a faster period as evidenced by the increased number of peaks and troughs through the graph.

Figure (5.2) shows a more detailed analysis of the learnt stroke. The top figure displays the swimmer over the full 30 seconds. Notably, it doesn't start to curve but swims straight forward, keeping the y-displacement negligible. On the middle row, we present different plots of the α values throughout the stroke. On the left, we plot the α_i values over the first 10 seconds, including the initialisation period. The initialisation period is short with the swimmer falling into its regular stroke after only roughly only half a second. The middle plot shows a more focused graph of one stroke, between 11 and 15 seconds, once the stroke has fully initialised. The stroke forms a wave, pausing briefly at each end point, with α_3 roughly copying the values of α_2 about 1 second behind. The period of the stroke is 4 seconds, $\frac{2}{3}$ of the length of the Purcell swimmer stroke. On the right, we display the phase plot of the α values, plotted against each other. The initialisation period stands out, drifting away from the middle, however, for the rest of the 30 seconds there are no other separate lines, implying that the swimmer has found a cyclic stroke. Finally, at the bottom, pictured are eight different visualisations of the swimmer between 11 and 15 seconds; one complete stroke. This helps visualise the stroke pattern.

Thus far, we have set each link to length 20 µm and $\delta = 1.5$. In Section 2.4.1 we experimented

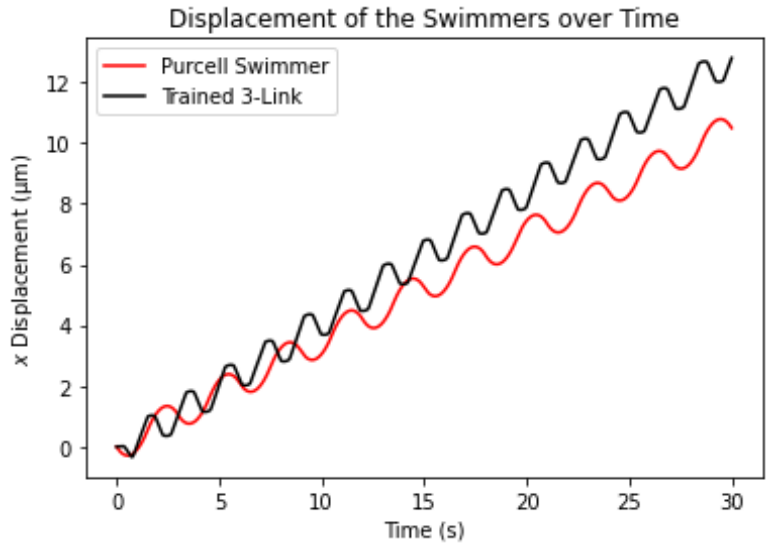

Figure 5.1: The displacements of both the 3-link Purcell swimmer and our taught NEAT swimmer over 30 seconds. Each link in each swimmer is 20 µm with $\delta = 1.5$.

varying these constants. We found that the relationship between L and the net x-displacement was exactly linear. We can extrapolate this for any stroke. However, the relationship between δ and the x-displacement was non-linear. Additionally, this relationship was specific for the Purcell swimmer and in this case should not be generalised to the NEAT swimmers. It is important, therefore, to explore this relationship. In Figure (5.3), we run four different swimmers, each with different values for δ ranging from 0.5 to 2 and record their displacements over 30 seconds. As we increase δ the efficiency of each stroke increases monotonously within this range. This is to be expected as we would hope that even if δ increases past the optimal stroke, the swimmer would learn to stop the α_i values prematurely and limit them to the optimal range. This would, however, cause the λ_i values to be rendered obsolete as the maximum values would no longer be reached. This could cause difficulties for the training process, making it take much longer as a set of important inputs are neglected.

Assessing the 4 different varying δ swimmers we have trained, the increase in x-displacement appears almost linear, and so it could be argued that we should increase δ further. However, as we increase δ further, the assumptions behind RFT can be broken, leading to our previous compromise of $\delta = 1.5$. The strokes become shorter as the δ decreases, as there is a lower range to vary across. This is evidenced by fewer peaks and troughs on the graph, ranging from large defined curves at $\delta = 2$ to an almost straight line at $\delta = 0.5$.

Figure (5.4) shows the α_i plots over time and the corresponding α traces of each different NEAT swimmer. Notably, no matter the value of δ, the swimmer pauses each link for a moment at the maximum and minimum values, causing the flat edges of the α traces. Furthermore, looking at the trace plots, they each look fairly similar, when scaled to match, implying that the ratio of time that each alpha stays motionless is almost a constant proportion for each value of δ. For example, the time spent still at the maximum value for the $\delta = 0.5$ stroke is roughly 0.25s, for a

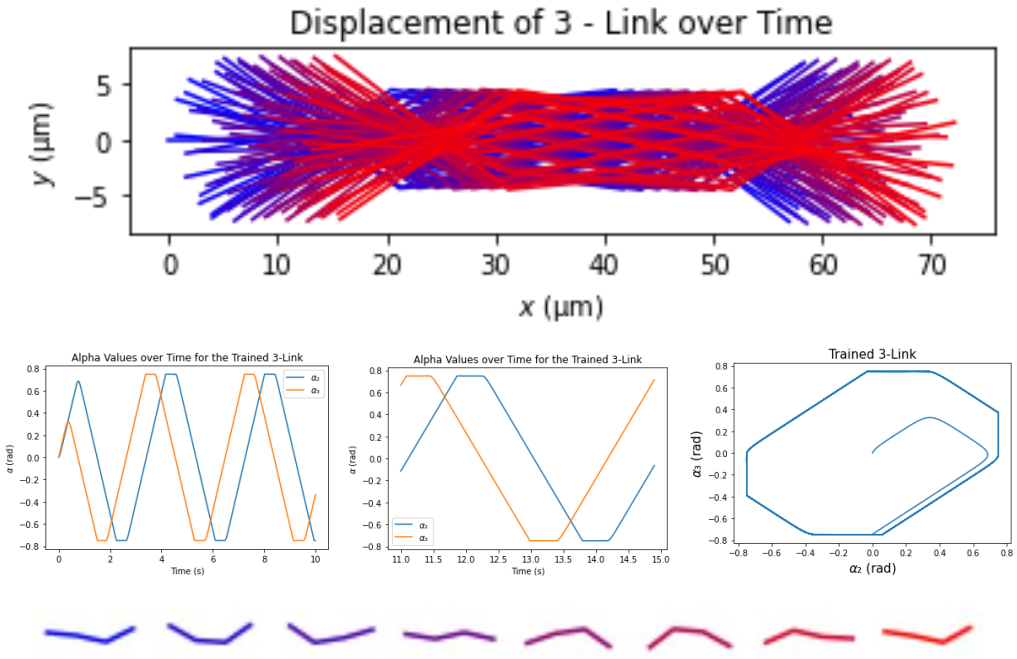

Figure 5.2: Top: The progression of the swimmer over the full 30 seconds coloured from blue to red as time increases. Middle left: The α values over time, including the initialisation period. Middle: One full stroke of the swimmer, taken between 11 and 15 seconds, well after the swimmer has moved past the initialisation period. Middle right: A phase plot of the two alpha values throughout the simulation. Bottom: 8 different shots of the swimmer as it completes one stroke, taken between the 11th and 15th second as per the middle plot.

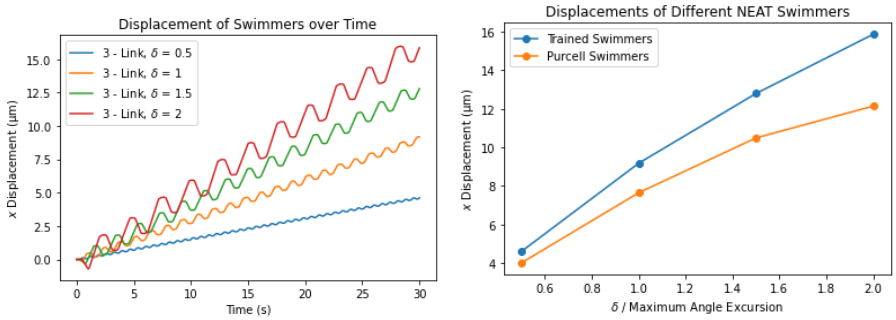

Figure 5.3: Left: A plot of the four NEAT swimmer's x displacements over 30 seconds. The swimmers have δ values of 0.5, 1, 1.5 and 2. Right: A plot of the final displacement of each swimmer after 30 seconds by their δ value. The values for the trained swimmers are shown in blue with their Purcell counterparts in orange.

stroke of length 1.3s. For the $\delta = 2$ stroke these times are 0.9s and 5.1s respectively. These give very similar proportions, 0.15 and 0.18, of the total stroke. With larger δ values, the stroke is not only longer because of the time taken for each link to rotate, but also due to a longer wait time at each extreme.

Our trained swimmers are much faster than their Purcell stroke counterparts. However, are they actually the optimal strokes or could faster strokes be found? NEAT does not give any theoretical guarantees that the optimum will be found no matter how many generations you train for. We ran each NEAT simulation for 60 generations and recorded the population fitness throughout. Figure (5.5) shows these and although we cannot confirm that the exact optimal has been found, the improvement for the best fitness on each of these graphs begins to plateau by 60 generations. Interestingly, [2] calculates and defines the optimal stroke with $\delta \approx 1$. In order to deduce the efficacy of our NEAT training method we can compare these strokes, as shown in Figure (5.6). It is worth noting that they reparameterized α_2 and α_3 such that $-\alpha_2 = \beta_1$ and $-\alpha_3 = \beta_2$ so in this figure we have reflected our data in the y-axis, allowing the strokes to be compared. Whilst the two α phases aren't identical, our NEAT version is an accurate emulation of the optimal stroke. It follows a very similar pattern, of a asymmetrical irregular octagon and only differs slightly in the length of each side. Again, this stroke was only trained for 60 generations, and whilst the fitness of each generation does seem to plateau, with more generations a closer approximation to the optimal stroke could be found.

5.2 Larger N Trained N-link Strokes

We now analyse the results of training longer N-link swimmers. Figure (5.7) shows the x displacement over time for 5 different length trained swimmers ranging from the 3-link swimmer previously analysed, to a much longer 10-link swimmer. As the number of links increases, the swimmers each travel further. It appears that there is an almost linear relationship between the number of links and the distance travelled, with a slight dip at $N = 5$.

However, this analysis provides a strong bias for longer chain swimmers. As we have already explored, increasing the length of the swimmer has a linear effect on the displacement. In Figure (5.7), each swimmer has link length 20 µm. This means that the 3-link swimmer has length 60 µm, whereas the 10-link swimmer is much longer with a total length of 200 µm. This gives this swimmer an advantage and we should avoid comparing them directly. In Figure (5.8), we instead set the total length of each swimmer to 60 µm and plot the same displacement graphs. A very different relationship between the variables is now present, appearing to tend towards an asymptote. This is what we would expect as we tend toward a better approximation of a real flagellum which must have a finite distance it can travel in 30 seconds.

Figure (5.9) compares the x-displacement for each N-link swimmer to its corresponding N-link Purcell swimmer. Each of the trained swimmers outperforms its Purcell counterpart by an increasing amount as N grows. We include at the bottom both the scaled, and unscaled swimmers displacements. The scaled Purcell swimmers become less efficient after $N = 5$, but the trained swimmers only improve.

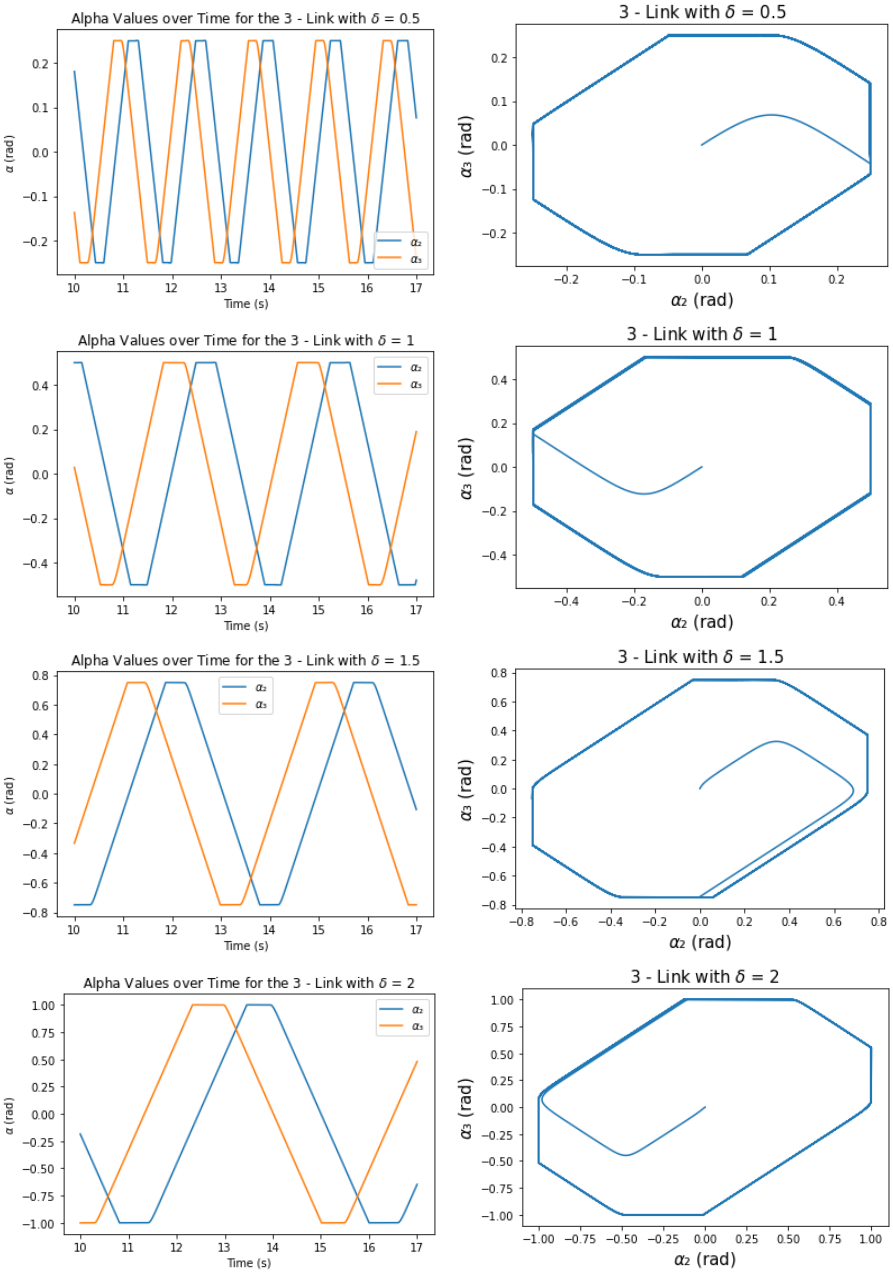

Figure 5.4: Four different trained swimmers each with varying δ values. From top to bottom δ is set to 0.5, 1, 1.5 and 2. On the left we can see a snapshot of each swimmer's α values between 10 and 17 seconds. On the right we have the α trace plots for each swimmer.

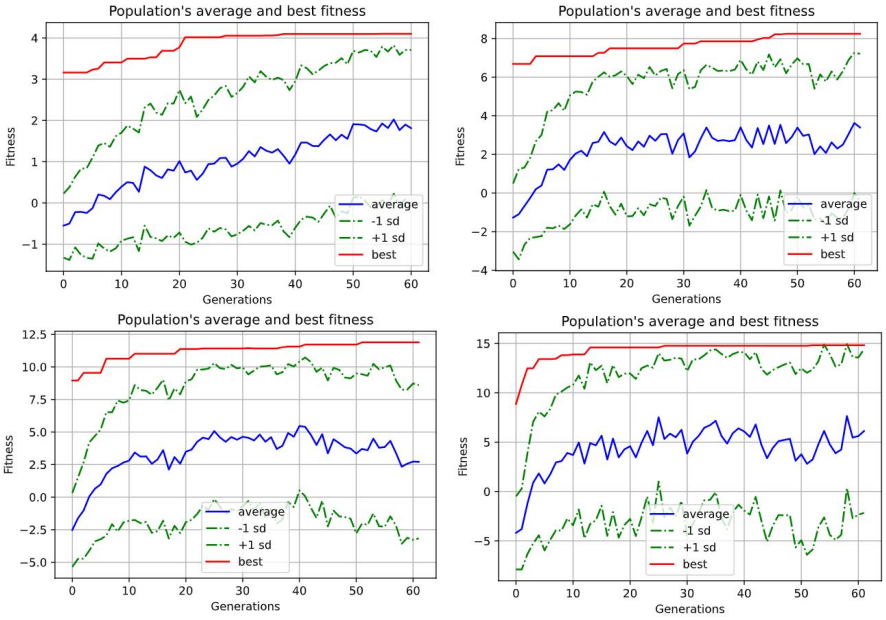

Figure 5.5: Plots of the population fitness over the 60 generations for each swimmer. Top Left: $\delta = 0.5$. Top Right: $\delta = 1$. Bottom Left: $\delta = 1.5$. Bottom Right: $\delta = 2$.

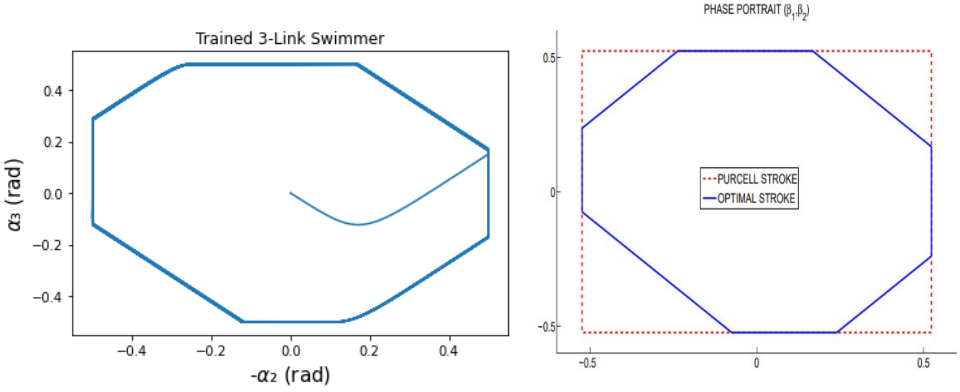

Figure 5.6: Left: The phase plot of our NEAT 3-link swimmer, where $\delta = \frac{\pi}{3}$. Right: A phase plot of the optimal 3-link swimmer. Figure extracted from [2].

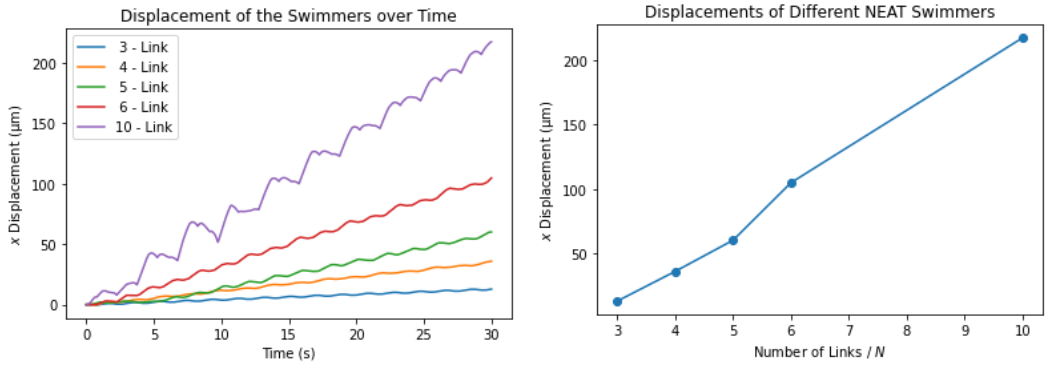

Figure 5.7: Left: The x-displacements over different length trained micro swimmers over 30 seconds. Right: The final x-displacement for each swimmer after 30 seconds, by the number of links within it. Each swimmer has $\delta = 1.5$ and each link length $20\mu m$.

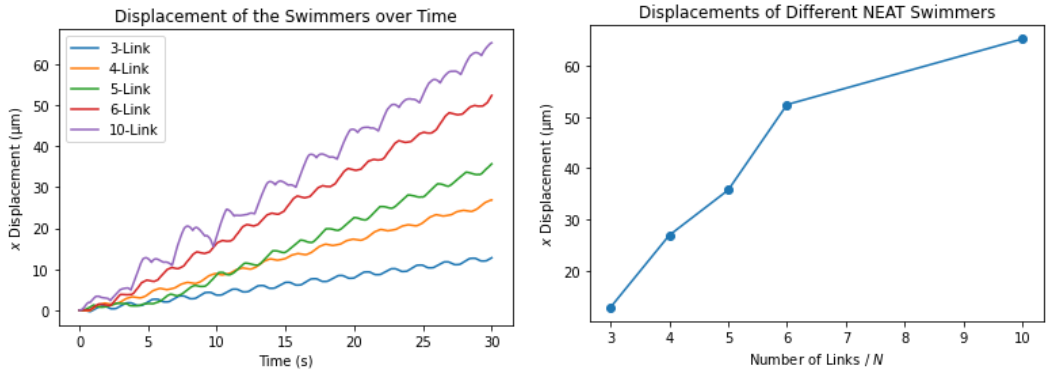

Figure 5.8: Left: The x-displacements over different length trained micro swimmers over 30 seconds. Right: The final x-displacement for each swimmer after 30 seconds, by the number of links within it. Each swimmer has $\delta = 1.5$ but now the same total link length of $60\mu m$.

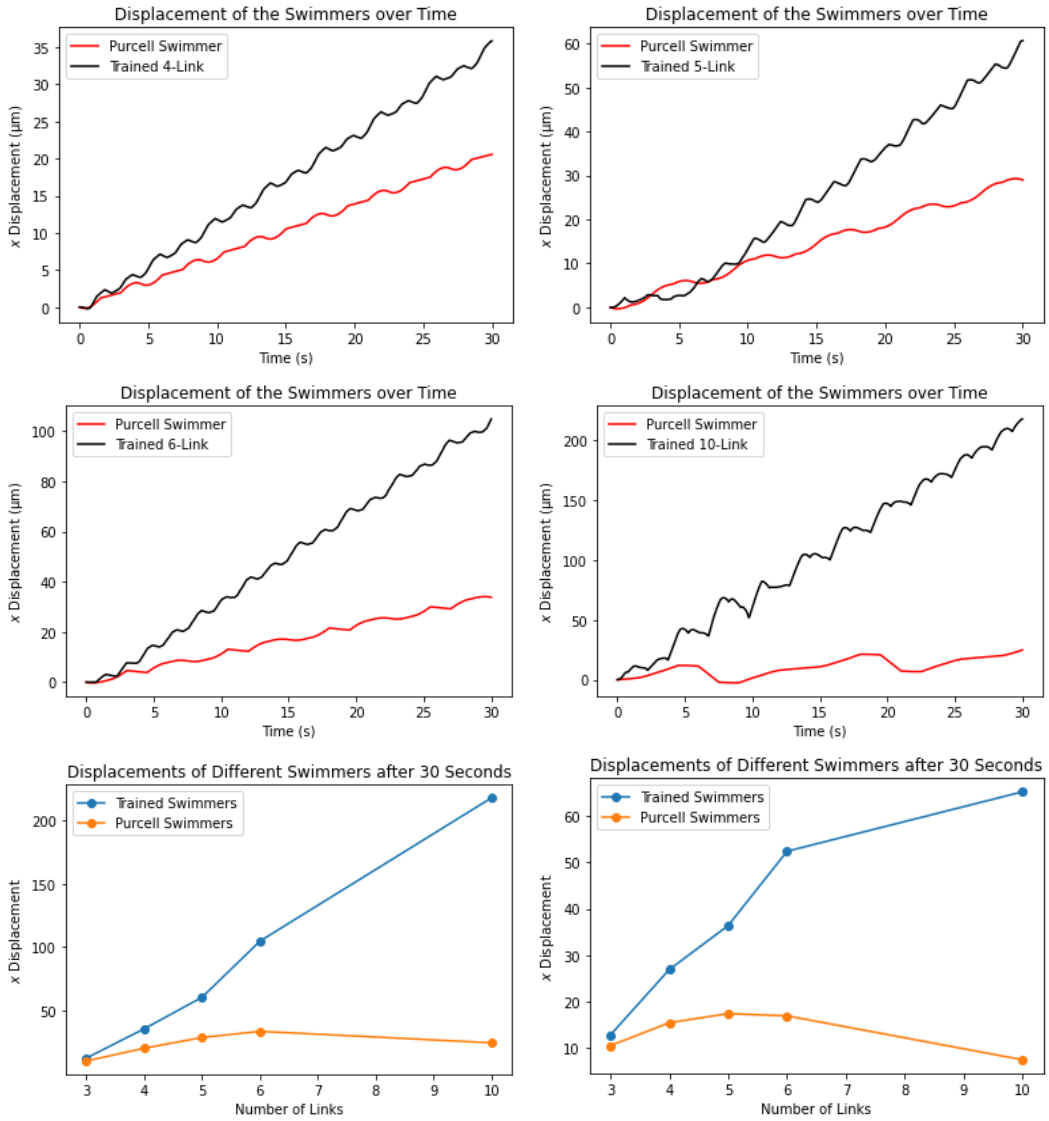

Figure 5.9: The top two rows show the displacement over time for the 4-link, 5-link, 6-link and 10-link trained swimmers, plotted against their respective N-link Purcell Swimmer. At the bottom we have a plot of their final displacements. On the left, the swimmers each have link length 20 µm. On the right, they are scaled so that their total length is 60 µm.

We will now evaluate the trained 4-link swimmer in more detail. An analysis of the stroke is shown in Figure (5.10). Exactly like the 3-link swimmer, a travelling wave forms, with each α_i copying the previous half a second behind. Interestingly, we again find a 4 second stroke formed, roughly the same length as the 3-link swimmer. We use a 3D plot to visualise the relationship between the three α_i values, with the trace appearing to form an almost regular octagon existing within a 2D-plane. The initialisation period is short and the swimmer finds a constant stroke.

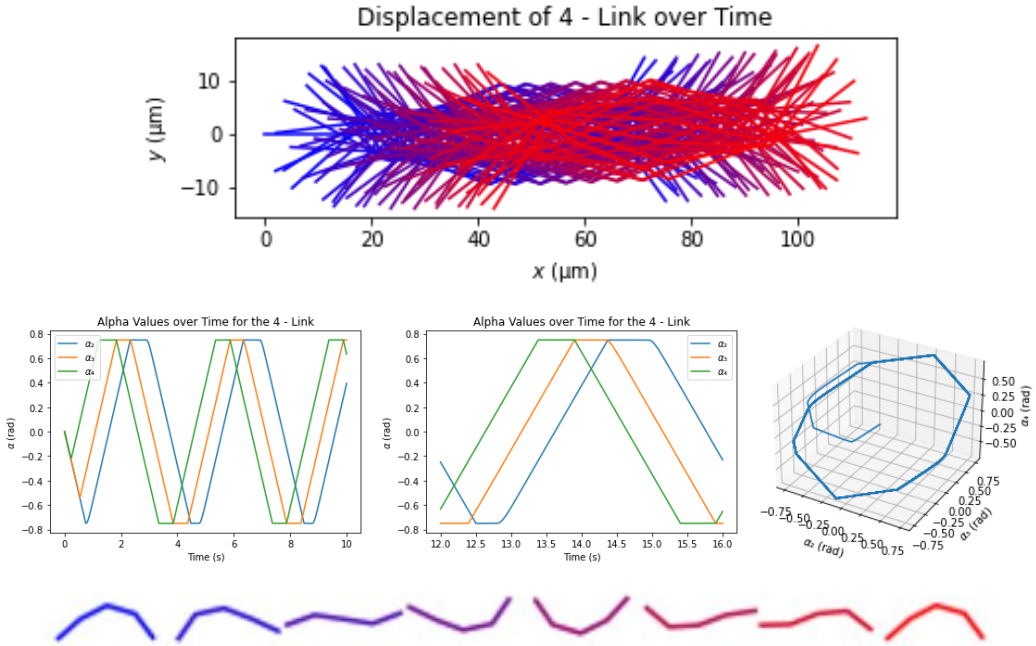

Figure 5.10: Top: The progression of the swimmer over the full 30 seconds coloured from blue to red as time increases. Middle Left: The α values over time, including the initialisation period. Middle: One full stroke of the swimmer, taken between 10 and 16 seconds, well after the swimmer has moved past the initialisation period. Middle Right: A phase plot of the three α values throughout the simulation shown as a 3d plot. Bottom: 8 different shots of the swimmer as it completes one stroke, taken between the 10th and 18th second as per the middle plot.

Figure (5.11) shows the α_i over time for both the 5-link swimmer and the 6-link swimmer. A similar trend is visible for these swimmers, forming a travelling wave, with each α_i following a similar path. The stroke lengths are once again both roughly 4 seconds long. This leads to the α_i traces becoming closer together and increasingly overlapped. Also included is a visualisation of each swimmer between 10 and 14 seconds showing the travelling wave over one full stroke.

Finally, we now focus on the trained 10-link swimmer. Figure (5.12) shows a detailed analysis of this swimmer. Firstly, the swimmer has not found a cyclic stroke that it can repeat throughout the 30 seconds, noticeable from the lack of a repeated curve on the displacement plot. Furthermore, this is evident from the chaotic alpha graph and the variations within the 2D and 3D plot. This alludes to the fact that this swimmer has not reached an optimal stroke pattern and requires further training. Responsible for this is the increasing size of the policy network. The

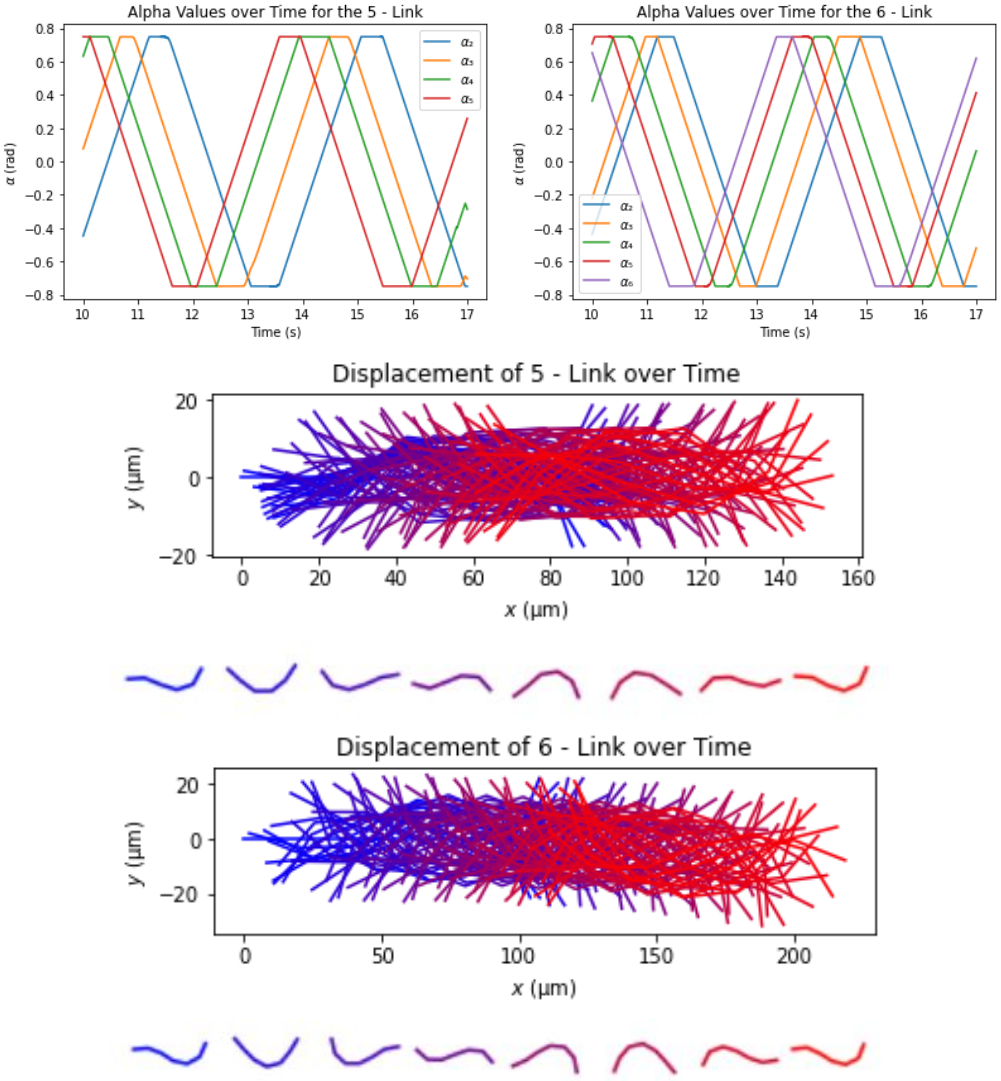

Figure 5.11: Left: The α values over time for the 5-link Swimmer. Right: The α values over time for the 6-link Swimmer. Below we have the displacements of the 5-link Swimmer and the 6-link swimmer over the full 30 seconds. Under each of these is a visualisation of the swimmer between 10 and 14 seconds. The travelling wave shape adopted by the swimmer is visible.

network we are trying to train has 28 inputs, with 9 outputs, giving a total of 531 initial weights and biases to optimize. This will require substantially more training and computation than the simpler 3-link model. However, despite this drawback, as N gets larger, the individual α_i values become less important, with the priority becoming the overall shape and movement of the swimmer. Despite not forming a perfect stroke pattern, looking at the bottom of Figure (5.12) we can see that the stroke forms a travelling wave emulating the movement of a flagellum in nature.

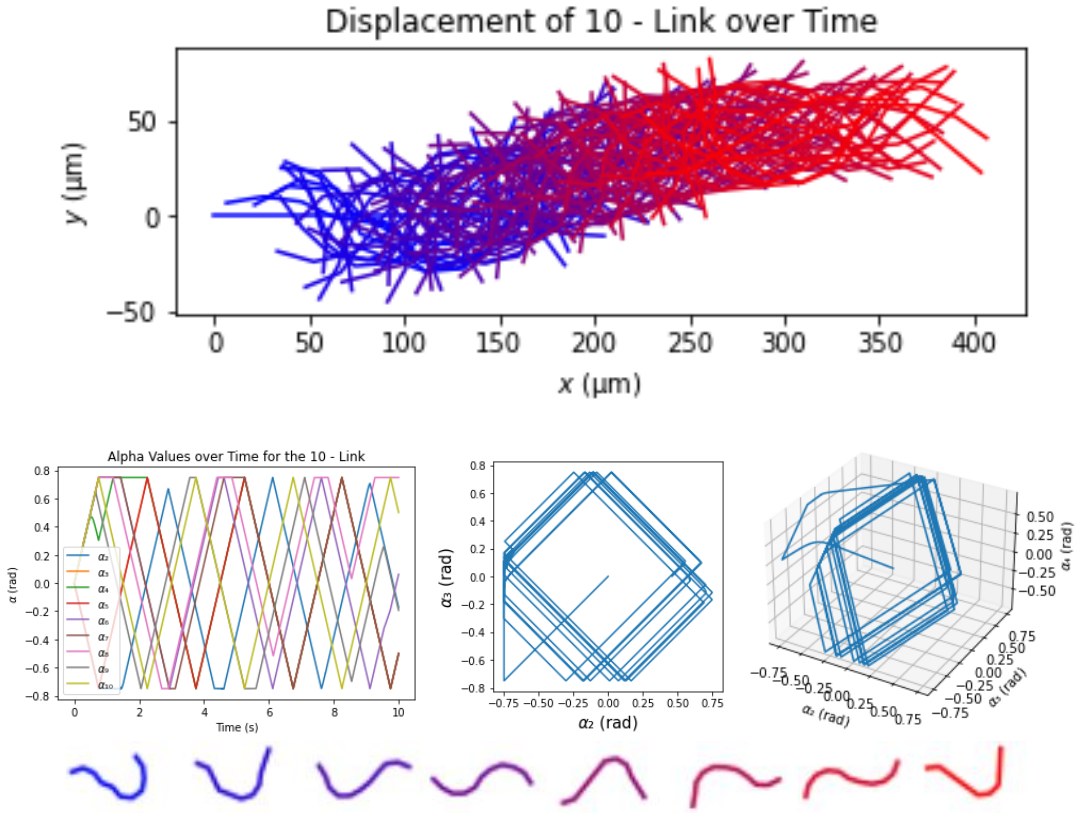

Figure 5.12: Top: We can see the progression of the swimmer over the full 30 seconds coloured from blue to red as time increases. Middle Left: The α values over time, including the initialisation period. Middle: A 2D phase plot of the first two α_i values throughout the simulation. Middle Right: A 3D phase plot of the first three α_i values throughout the simulation. Bottom: 8 different shots of the swimmer as it completes one stroke, taken between the 1st and 3rd second.

Chapter 6

Conclusion

We started this dissertation introducing the concept of low Reynold number environments and the changes these bring to the physics we are used to. We defined the N-Link model, a simple microscopic swimmer from which we then built a set of equations describing its motion. Using RFT these took the form of ODEs which fully described its interactions with the environment. We then briefly looked at the numerical aspects of solving these before introducing and analysing the inefficiencies of the N-link Purcell stroke, especially as our model becomes larger. To solve this, we focused on the crux of this paper: using reinforcement learning to produce faster locomotive gaits. We contemplated the main sub-elements of this set of techniques then focused on genetic algorithms, specifically NEAT. We introduced a methodology, applying the NEAT algorithm to our microswimmers, developing efficient and fast swimming strokes.

We are now left to conclude on the overall efficiency of this method. Firstly, when training 3-link swimmers, the trained stroke accurately emulated the optimal stroke, calculated through optimal control theory. This stroke pattern seemed mirrored in our results for the 4-link, 5-link and 6-link swimmers and although we cannot compare them to a defined optimal stroke, there is an evident travelling wave shape along each of them. Whilst we cannot prove these are optimal, their propulsion method is greatly improved on the Purcell stroke.

On the other hand, our 10 link swimmer did not reach as satisfying of a conclusion as it failed to find a cyclic stroke. Furthermore, its α plots are chaotic and an overall pattern is hard to find. Despite this, as the number of links grows, the importance of each alpha individually is diminished. More important now is the general shape of the swimmer and how the α_i values affect each other. Again, even though our 10-link swimmer does not converge exactly to optimal, there is a distinct travelling wave pattern throughout its stroke. It forms a very natural looking approximation to a biological flagellum with an almost elastic nature. It may not be exactly optimal, but its method of motion is a great approximation to what we were aiming for and a large improvement on the 10-link Purcell swimmer.

This lack of convergence in the 10-link swimmer could be for a number of reasons. Firstly, as mentioned prior, the network size grows as you increase the number of links, so we would expect a much longer time taken to converge. Additionally, the numerical errors will be larger with 10-links. Furthermore, we are using RFT, an approximation of the drag forces on the N-link model. This assumes that the only forces on the model are directly from the encompassing viscous liquid, not from other parts of the swimmer. As N gets larger, it is easier for different

parts of the swimmer to get closer to each other potentially breaking this assumption. It could be worth trying to solve this by reducing δ as N grows, limiting the bend in the swimmer. Furthermore, RFT is always just a simple approximation. To try fix this further it could be worth employing more advanced methods to measure these drag forces. With this we might discover more effective strategies with larger of values of δ.

NEAT is a very effective algorithm when working with simple neural networks. In fact, [7] used NEAT to solve a very similar scenario, training microswimmers to emulate chemo-taxis. Many of the developments from NEAT, such as hyper-NEAT [35], are focused on deep reinforcement learning which would be excessive for our problem. However, at 10-links these networks are no longer simple. Despite its success with the smaller networks, NEAT may simply not be an effective tool as N continues to grow.

For future research, it would be interesting to expand both our model of the swimmer and the model of its reactions with its environment. We could include elasticity within the swimmer [36] as well as improving on RFT with other techniques. This would help reduce the reality gap of our reinforcement learning. Furthermore, in this dissertation, we have focused on the strokes learnt, not necessarily the intelligence or adaptability of the swimmers. It would be an interesting extension to train adaptive swimmers, where the policy does not learn just one stroke, but multiple depending on the environment conditions. These conditions could be varying viscosity or even nutrient gradients.

Bibliography

[1] A. Shapere and F. Wilczek, "Geometry of self-propulsion at low reynolds number," *Journal of Fluid Mechanics*, vol. 198, pp. 557–585, 1989.

[2] L. Giraldi, P. Martinon, and M. Zoppello, "Controllability and optimal strokes for n-link microswimmer," in *52nd IEEE Conference on Decision and Control*, pp. 3870–3875, IEEE, 2013.

[3] L. E. Becker, S. A. Koehler, and H. A. Stone, "On self-propulsion of micro-machines at low reynolds number: Purcell's three-link swimmer," *Journal of fluid mechanics*, vol. 490, pp. 15–35, 2003.

[4] R. Dreyfus, J. Baudry, M. L. Roper, M. Fermigier, H. A. Stone, and J. Bibette, "Microscopic artificial swimmers," *Nature*, vol. 437, no. 7060, pp. 862–865, 2005.

[5] E. Lauga and T. R. Powers, "The hydrodynamics of swimming microorganisms," *Reports on Progress in Physics*, vol. 72, no. 9, p. 096601, 2009.

[6] E. A. Gaffney, H. Gadêlha, D. Smith, J. Blake, and J. C. Kirkman-Brown, "Mammalian sperm motility: observation and theory," *Annual Review of Fluid Mechanics*, vol. 43, pp. 501–528, 2011.

[7] B. Hartl, M. Hübl, G. Kahl, and A. Zöttl, "Microswimmers learning chemotaxis with genetic algorithms," *Proceedings of the National Academy of Sciences*, vol. 118, no. 19, 2021.

[8] B. J. Nelson, I. K. Kaliakatsos, and J. J. Abbott, "Microrobots for minimally invasive medicine," *Annual review of biomedical engineering*, vol. 12, pp. 55–85, 2010.

[9] E. M. Purcell, "Life at low reynolds number," *American journal of physics*, vol. 45, no. 1, pp. 3–11, 1977.

[10] F. Alouges, A. DeSimone, L. Giraldi, and M. Zoppello, "Self-propulsion of slender micro-swimmers by curvature control: N-link swimmers," *International Journal of Non-Linear Mechanics*, vol. 56, pp. 132–141, 2013.

[11] J. Gray and G. Hancock, "The propulsion of sea-urchin spermatozoa," *Journal of Experimental Biology*, vol. 32, no. 4, pp. 802–814, 1955.

[12] B. M. Friedrich, I. H. Riedel-Kruse, J. Howard, and F. Jülicher, "High-precision tracking of sperm swimming fine structure provides strong test of resistive force theory," *Journal of Experimental Biology*, vol. 213, no. 8, pp. 1226–1234, 2010.

[13] J. Happel and H. Brenner, *Low Reynolds number hydrodynamics: with special applications to particulate media*, vol. 1. Springer Science & Business Media, 2012.

[14] Z. Lei and J. Hongzhou, "Variable step euler method for real-time simulation," in *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology*, pp. 2006–2010, IEEE, 2012.

[15] J. C. Butcher, *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[16] A. Iserles, *A first course in the numerical analysis of differential equations*. No. 44, Cambridge university press, 2009.

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[18] A. Nandy and M. Biswas, *Reinforcement Learning: With Open AI, TensorFlow and Keras Using Python*. Apress, 2017.

[19] A. C. H. Tsang, P. W. Tong, S. Nallan, and O. S. Pak, "Self-learning how to swim at low reynolds number," *Physical Review Fluids*, vol. 5, no. 7, p. 074101, 2020.

[20] S. Colabrese, K. Gustavsson, A. Celani, and L. Biferale, "Flow navigation by smart microswimmers via reinforcement learning," *Physical review letters*, vol. 118, no. 15, p. 158004, 2017.

[21] S. Muiños-Landin, A. Fischer, V. Holubec, and F. Cichos, "Reinforcement learning with artificial microswimmers," *Science Robotics*, vol. 6, no. 52, 2021.

[22] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[23] F. Cichos, K. Gustavsson, B. Mehlig, and G. Volpe, "Machine learning for active matter," *Nature Machine Intelligence*, vol. 2, no. 2, pp. 94–103, 2020.

[24] E. Ronald and M. Schoenauer, "Genetic lander: An experiment in accurate neuro-genetic control," in *International Conference on Parallel Problem Solving from Nature*, pp. 452–461, Springer, 1994.

[25] K. O. Stanley and R. Miikkulainen, "Efficient evolution of neural network topologies," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, vol. 2, pp. 1757–1762, IEEE, 2002.

[26] A. Behjat, S. Chidambaran, and S. Chowdhury, "Adaptive genomic evolution of neural network topologies (agent) for state-to-action mapping in autonomous agents," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 9638–9644, IEEE, 2019.

[27] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[28] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms.," in *IJCAI*, vol. 89, pp. 762–767, 1989.

[29] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[30] V. Stanovov, S. Akhmedova, and E. Semenkin, "Difference-based mutation operation for neuroevolution of augmented topologies," *Algorithms*, vol. 14, no. 5, p. 127, 2021.

[31] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[32] D. Chen, C. L. Giles, G.-Z. Sun, H. Chen, Y.-C. Lee, and M. W. Goudreau, "Constructive learning of recurrent neural networks," in *IEEE International Conference on Neural Networks*, pp. 1196–1201, IEEE, 1993.

[33] F. Gruau, D. Whitley, and L. Pyeatt, "A comparison between cellular encoding and direct encoding for genetic neural networks," in *Proceedings of the 1st annual conference on genetic programming*, pp. 81–89, 1996.

[34] S. W. Mahfoud, *Niching methods for genetic algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[35] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial life*, vol. 15, no. 2, pp. 185–212, 2009.

[36] C. Moreau, L. Giraldi, and H. Gadêlha, "The asymptotic coarse-graining formulation of slender-rods, bio-filaments and flagella," *Journal of the Royal Society Interface*, vol. 15, no. 144, p. 20180235, 2018.