Neuro-Evolution Using Recombinational Algorithms and Embryogenesis for Robotic Control

Thesis by

Anthony M. Roy

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

2010
(Defended December 11, 2009)

© 2010
Anthony M. Roy
All Rights Reserved

To Kellie, my future wife

Acknowledgments

A great many individuals helped in the creation of the research presented here, and it would be near impossible to list them all. However, I'd like to begin by thanking my advisors-three, Dr. Antonsson, Dr. Shapiro, and Dr. Burdick. Dr. Erik Antonsson was an integral part of the initial envisioning and often used his considerable expertise to refine the presentation of this work. Dr. Andrew Shapiro helped as a sounding board to bounce ideas off of frequently, and was the prime motivator for studying the inner workings of NEURAE . Dr. Joel Burdick is served as a valuable resource of robotic information as well as administrative advice.

I'd also like to acknowledge the contributions of Or Yogev, Fabien Nicase, and Tomonori Honda, other ESSL members whose frequent exchange of technical information was a fountain of fresh ideas.

Furthermore, I'd like to thank Dr. Swaminathan Krishnan for allowing the use of his Garuda computing cluster. Without it, the algorithms contained within would still be running for another decade or so.

I'd also like to thank the following Caltech students, whose brilliance I occasionally borrowed when needed:

Michael Shearn, Anna Beck, Valerie Scott, Andrew Downard, Jason Keith, Virgil Griffith, Justus Brevik, Julia Braman, Jeremy Ma, David Pekarek, Kakani Young, Mary Dunlop, Matthew Eichenfield, Pablo Abad-Manterola, Angela Capece, Christopher Kovalchick, Philipp Boettcher, Ronnie Bryan, Roseanna Zia, Derek Riendikirk, Geoffrey Lovely, Leonard Lucas, Emily McDowell, Sameer Walavalkar, and Timothy Chung.

Last, but certainly not least, I'd like to thank Anthony Roy, Arnetress Roy, and Yolanda Ware. My family, whose support has been a constant long before, and I'm sure long after the this dissertation, is the bedrock upon much of my success has been built.

Abstract

Control tasks involving dramatic nonlinearities, such as decision making, can be challenging for classical design methods. However, autonomous, stochastic design methods such as evolutionary computation have proved effective. In particular, genetic algorithms that create designs via the application of recombinational rules are robust and highly scalable. Neuro-Evolution Using Recombinational Algorithms and Embryogenesis (NEURAE) is a genetic algorithm that creates C^{++} programs that in turn create neural networks which can function as logic gates. The neural networks created are scalable and robust enough to feature redundancies that allow the network to function despite internal failures. An analysis of NEURAE evinces how biologically inspired phenomena apply to simulated evolution. This allows for an optimization of NEURAE that enables it to create controllers for a simulated swarm of Khepera-inspired robots.

Contents

Acknowledgments **iv**

Abstract **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Outline 6

2 Methodology **8**

2.1 Background 8

2.1.1 Neural Networks 8

2.1.2 Genetic Algorithms 9

2.2 The NEURAE Genotype 11

2.2.1 Overview 11

2.2.2 Biological Analog 12

2.2.3 *If* Structure Nucleotide 12

2.2.4 Condition Nucleotides 14

2.2.5 Action Nucleotides 16

2.2.6 C^{++} Programs (Proteins) 16

2.3 Evaluation, Mutation, and Selection 20

3 Logic-Gate Evolution **24**

3.1 Overview 24

3.2 Robust XOR Gate 25

3.2.1 Evaluation Parameters 25

3.2.2 Evolution Results 26

3.3 Large Parity Gate 28
3.3.1 Evaluation Parameters 28
3.3.2 Evolution Results 29

4 Sensitivity Analysis 33
4.1 Mutation Rates 33
4.2 Qualities of Productive Evolution 38
4.3 Variation of Nucleotides within the NEURAE Codon 41

5 Derivation of Simulation Environment 43
5.1 Nomenclature 43
5.2 Two-Wheeled Robot Movement 46
5.3 Collision Detection 52
5.4 Sensor and World Interaction 53

6 Robotic-Controller Evolution 58
6.1 Overview 58
6.2 Line-Following Robot 58
6.2.1 Evaluation Parameters 58
6.2.2 Evolution Results 59
6.3 Obstacle-Avoiding Robot 61
6.3.1 Evaluation Parameters 61
6.3.2 Evolution Results 64
6.4 Goal-Finding Swarm Robots 65
6.4.1 Evaluation Parameters 65
6.4.2 Evolution Results 68

7 Conclusion 73

Bibliography 84

Appendix 85

List of Figures

2.1 McCulloch-Pitts neuron model. 9

2.2 Steps of a standard genetic algorithm. 10

2.3 Sample genome and biological analog 12

2.4 Nucleotides of each codon 12

2.5 If structure codon and protein transcription 13

2.6 Sample genome and protein pseudocode 17

2.7 Protein pseudocode and sample NAND gate 18

2.8 Flowchart of protein pseudocode 18

2.9 Steps showing the embryogenesis of NAND gate 19

2.10 Point mutation example. The underlined nucleotides are switched 21

2.11 Single-point crossover mutation example. Parts of the genome which have been swapped are underlined 21

2.12 Two-point crossover mutation example. Parts of the genome which have been swapped are underlined 21

2.13 Conjugation mutation example. Parts of the genome which have been inserted are underlined 22

2.14 Gene duplication example. The nucleotides copied more than once are underlined 22

2.15 Gene deletion example. The nucleotides deleted are underlined 23

2.16 Translocation example. The underlined nucleotides are moved to another gene locus 23

3.1 Best fitness throughout the evolution of a robust exclusive-OR logic gate . . 26

3.2 First generated XOR gate 27

3.3 Network functionality 27

3.4 Best generated XOR gate 27

3.5 Network functionality 27

3.6 Code for creating a robust XOR gate 28

3.7 Larger XOR gate 28

3.8 Fitness of best-performing individual throughout the evolution of a scalable parity gate 30

3.9 Scalable parity gate with two inputs 30

3.10 Scalable parity gate with four inputs 31

3.11 Scalable parity gate with 13 inputs 31

3.12 Code for creating parity gates of arbitrary size 32

4.1 Log-log plot of α generation vs. $log\left(\frac{1}{1-F}\right)$ for a point mutation rate of $\mu = 0.4$. 34

4.2 Probability density function and histogram of α generation for mutation rate of $\mu = 0.4$. 34

4.3 Gaussian distribution of best fitness at the end of evolutionary runs with a point mutation rate of $\mu = 0.4$. 35

4.4 The prism is representative of the mutation rate landscape as bounded by the above constraints. 37

4.5 Genes used by the top 10% within a successful evolution 39

4.6 Genes used by the top 10% within an unsuccessful evolution 39

4.7 Structure of genes used by the top 10% of each generation during a successful evolution. 40

4.8 Structure of genes used by the top 10% of each generation during an unsuccessful evolution. 40

5.1 Diagram of variables for two-wheeled motion derivation 46

5.2 Verification of rotational accuracy with and without approximation. 49

5.3 Verification of rotational and translational accuracy used the respective left and right wheel speeds of $\nu_1 = 0$ m/s and $\nu_2 = 1$ m/s. The maximum orientation, x-position, and y-position errors are 0.017 rad, 0.0079 m, and 0.0083 m, respectively. 50

5.4 Verification of rotational and translational accuracy used the respective left and right wheel speeds of $\nu_1 = 0.5$ m/s and $\nu_2 = 1$ m/s. The maximum orientation, x-position, and y-position errors are 0.037 rad, 0.055 m, and 0.056 m, respectively. . . . 51
5.5 Diagram of variables for obstacle collision check. . . . 52
5.6 Collision detection was verified by placing the robot within a small obstacle and having it move around. As shown above, the center of the robot is never closer than 0.5 m (the radius) to the obstacle wall. . . . 54
5.7 Model of robot sensor configuration for path following simulations. . . . 54
5.8 Diagram of variables used for path detection calculations. . . . 54
5.9 Model of robot sensor configuration for full 2-D navigation. . . . 56
5.10 Diagram of variables used for full 2-D navigation. . . . 56
5.11 Graphical verification of accurate laser/object interaction. A blue line indicates the corresponding ANN input is inactivate while red line indicates the corresponding ANN input has been activated. The concentric circles are indicative of the desired goal . . . 57

6.1 Preference function for position error in line following evaluation . . . 59
6.2 Robot path compared with desired path . . . 60
6.3 ANN controller for a line following robot . . . 60
6.4 Code used to make line following controller . . . 61
6.5 Goal sensor configuration for the obstacle avoiding robots. Detection is separated into left, center, and right. . . . 62
6.6 Environment for tier 4 evaluation . . . 63
6.7 Environment for tier 5 evaluation . . . 63
6.8 ANN controller for obstacle avoidance . . . 64
6.9 Obstacle avoidance robot in a densely obstructed environment . . . 65
6.10 Obstacle avoidance robot in a environment with concave obstacle . . . 65
6.11 Code used to make obstacle avoidance controller . . . 66
6.12 Goal sensor configuration for swarming robots where the goal is obstructed from the entire swarm. . . . 67

6.13 Goal sensor configuration for swarming robots where a member of the swarm can detect the goal. 67

6.14 A single swarming robot in an environment with a convex obstacle 67

6.15 A single swarming robot in an environment with a star obstacle 67

6.16 Two swarming robots in an environment with a star obstacle 68

6.17 Two swarming robots in a large environment with various obstacles 68

6.18 ANN controller for each swarming robot 69

6.19 Steps showing the movement of an evolved swarm 71

6.20 A single swarming robot in an environment with concave obstacle 72

6.21 Three swarming robots in a large environment with various obstacles 72

List of Tables

2.1 Universal tiers for adjusting fitness exponent (x) 20

3.1 Desired output pattern for XOR logic-gate 25
3.2 Tiers for adjusting fitness exponent (x) in robust XOR evolution 25
3.3 Tiers for adjusting fitness exponent (x) in scalable parity evolution 29

4.1 The statistical results for varying mutation rates while only using point mutations 35
4.2 Mutation rates for 3-dimensional sensitivity analysis with variables in **bold** are indicative of the chosen points on Figure 4.4 38
4.3 The statistical results for varying mutation rates across the mutation rate landscape given in Figure 4.4 38
4.4 Actions in executed genes 41

6.1 Tier for adjusting fitness exponent (x) in line following evaluation 59
6.2 Dominant logic for line following robots 60
6.3 Tiers for adjusting fitness exponent (x) in obstacle avoidance evaluation . . 62
6.4 Logic test goal finding robots are required to pass before simulation. For this test, all LIDAR inputs are inactive 63
6.5 Tiers for adjusting fitness exponent (x) 68

Chapter 1

Introduction

1.1 Motivation

Artificial neural networks (ANNs) are able to solve mathematically ill-defined problems with a network of computationally simple elements. Inspired by the architecture of the human brain, McCulloch and Pitts (1943) modeled biological neurons as simple mathematical units capable of comprising large networks. Turing (1950) described the plausibility of a complex computing machine being constructed from simple computational units. Hornik et al. (1989) proved that with the proper architecture, an ANN composed of McCulloch-Pitts neurons can approximate any regular function within a finite space to an arbitrary degree of accuracy.

The potential of ANNs has inspired their application in a wide range of fields. The primary use of neural networks has been for classification purposes. Wu et al. (1993) and Odewahn et al. (1992) showed how ANNs can be used to classify malignant tumors in mammograms and star types in telescopic images, respectively. Waibel (1989) found use of temporal ANNs in the realm of speech recognition. Atiya (2001) detailed how neural networks can be capable tools for analyzing credit risk.

Neural networks have also been used for robotic control. Naito et al. (1997) argued the nonlinearity and distributed information storage of ANNs make them attractive candidates for control. Biewald (1996) used a neural network controller for obstacle avoidance by partitioning the problem into separate path planning and local navigation regions. Cui and Shin (1993) controlled multiple manipulators by using neural networks to approximate the Jacobian at various points of the robots' range of motion. Beer et al. (1992) and Lewis et al. (1994) employed recurrent neural networks to control the gait of a hexapod robot. Hornby

et al. (2001) used ANNs as controllers that are able to evolve alongside the morphology of the controlled robots. Yue and Rind (2006) used a neural network for object recognition in an obstacle avoiding robot.

However, there are limits to what current ANN learning algorithms can accomplish. Convergence of the widely used back propagation algorithm is dependent on network architecture and learning rates (Hecht-Nielsen 1992). The setting of these parameters require significant expertise and *a priori* knowledge of the problem to be solved. Otherwise, the network is likely to converge to a non-optimal solution or be unduly influenced by the sequence of learning examples that are given (Sutton 1986). Furthermore, training session require large amounts of historical data and are computationally demanding.

Hebb (1949) posited a theory that biological neural networks adapt by repeated firing. As the activation of one neuron coincides with the activation of another several times, the connection between the two strengthens in such a way that it becomes easier for the first neuron to excite the second. Perhaps the most well-known application of Hebbian learning in an ANN is a Hopfield network. Hopfield (1982) proved that an ANN can use Hebbian learning to converge to a local minimum, thus making the network stable. However, stability requires the network be symmetrical, with nodes being connected to each other with identical weights. Even if this constraint is not enforced, Hebbian learning is a capable method for getting ANNs to classify data (Sanger 1989; Oja 1992; Daucé et al. 1998). However, these methods often converge to local minima and are not suited to finding an global optimum.

Real-time reinforcement is yet another scheme for adapting network connections. Onat et al. (1998) showed how positive reinforcement can be used to strengthen connections between neurons when the network is performing as desired. Chialvo and Bak (1999) showed how similar learning occur with negative reinforcement. Bosman et al. (2003) gave a more generalized approach which combined Hebbian and reinforcement learning. However, as evident in the work of Sutton and Barto (1999), there are several learning parameters of the reward function which must be tuned, and these values require expertise or trial and error to set correctly.

Because training ANNs is inherently a trial-and-error process, it was a natural extension to use a genetic algorithm (GA) to train them. Genetic algorithms, also known as evolutionary algorithms, use simulated evolution to design solutions. As conceived by Holland

(1975), GAs are a machine learning paradigm in which the parameters of a possible design solution are varied over time to eventually find a viable solution. Furthermore, many solutions are designed in parallel, and the parameters of one solution may be used, partly or completely, in the parameters of another. As a result, the design solutions within a GA improve over time in a manner similar to biological evolution. Like ANNs, GAs have found applications in a wide range of fields such as circuit design in electrical engineering (Miller et al. 1997), ligand bonding in chemistry (Morris et al. 1998), and granular composites in material science (Fraternali et al. 2009).

Most ANNs designed by evolutionary algorithms involved optimizing the weight of a set network architecture (Montana and Davis 1989; Eberhart and Kennedy 1995). Further work focused on evolving the parameter of various different learning algorithms (Roy et al. 1999; Chen et al. 1999).

Eventually there was an emergence of GAs in which network architecture and connection weights are coevolved in a process known as **neuro-evolution**. Reed (1999) gives a good overview of many GAs which evolve network architectures through decomposition, where a large, fully connected network has connections and nodes removed. The shortcomings of such schemes were addressed by Angeline et al. (1999) who offered GNARL as an alternative. According to Angeline, decomposition methods often become trapped at local network minima, which causes them to suffer the same non-optimum finding deficiencies GAs were designed to overcome.

More current neuro-evolution efforts include NEAT by Stanley and Miikkulainen (2002), and AGE by Duerr et al. (2006). Both methods utilize genomes that represent the nodes and connections of ANNs. The genomes of NEAT explicitly contain the connection weights. The three tiers of NEAT, gene tracking, speciation, and complexifying, have become so well studied and efficient that Stanley et al. (2005) managed to evolve networks in real time. In AGE, the genome includes a section for each node that, when combined with the similar section of another node, determines the weight of connections. Both NEAT and AGE are able to use evolution to construct networks capable of performing complex control tasks. However, the practical size of evolved networks is limited by the requirement that each node of the network is directly represented in the genome.

There are applications where a large network is necessary, such as the Gammon project (Tesauro 1992). The Gammon project was an attempt to make a neural network a successful

backgammon player. Gammon looks at the current state of the board and possible moves for a given roll of the dice. It then uses the neural net to calculate which possible move for the given dice roll would lead to the highest probability of winning, and moves accordingly. With 198 input units and 40 hidden neurons, it plays on a level even with the best backgammon players in the world. If one were to design such a network with a genetic algorithm, the GA would have to be scalable.

One of the first examples of a scalable GA was introduced by Kitano (1990). In his seminal paper, he used matricies to represent ANN connection weights. He achieved scalability by using single bits to represent small connectivity graphs and allowing recursion of such bits. As a result, a neural network could be represented more compactly with reasonable modularity. Tufte and Haddow (2000) used a similar genome shorthand to evolve large digital circuits.

Theraulaz and Bonabeau (1995) have shown that the reuse of a small set of rules to create a phenotype is an effective alternative to storing and manipulating the large amount of data that describes each individual directly. Bentley and Kumar (1999) have shown that indirect encodings produce solutions to design problems faster and better than their directly encoded counterparts. Federici and Downing (2006) have shown that rule-based encoded designs are more robust as well. Grajdeanu (2007) evolved rules capable of making virtual 2-D organisms with interesting properties such as cell differentiation and repair. Yogev and Antonsson (2007) created 3-dimensional structures by evolving a set a rules which directs how a single cell should grow through a process called **embryogenesis**.

Embryogenesis is best described as genetic programming (GP) applied to the evolution of instructions which in turn determines how an artificial embryo should grow (Garis 1992). A genetic program is a genetic algorithm where the evolution is performed on a computer program. In its inception, Fogel et al. (1966) devised a way to use the evolutionary process that allowed the **recombination** of a computer program into various configurations. Later, LISP programs were evolved by Koza (1989) to create programs which could discover recursive expressions for numerical sequences and pattern recognition. O'Neill and Ryan (2001) went on to make grammatical evolution (GE), which was a scheme for how to do genetic programming in any arbitrary language. However, in GP the program is the end result of evolution. It is when these programs are used to grow something else when true embryogenesis occurs.

Embryogenesis was applied to ANN evolution when Gruau (1992) created cellular encoding (CE), which dictates how a network grows from a single cell. CE was able to create a network of arbitrary size that is capable of detecting logical parity. However, as noted by Luke and Spector (1996), Gruau achieves much of his modularity by using a recursion rule that results in generating nodes with identical inputs and outputs. While his networks are able to perform well for tasks requiring symmetry, his method performs poorly for networks that require asymmetric weights.

Kitano (1995) used his compact representation to encode instructions for the growth of virtual axions and dendrites in graphical ANN. His scheme also implemented cell differentiation. However, this application was geared more towards simulating the growth of a biological neural network instead of creating ANNs for engineering purposes.

Astor and Adami (2000) expanded on the idea of growing neural networks by creating NORGEV, a simulated wet chemistry set. Within their evolutionary algorithm, a network is grown from a single neuron by using cell chemistry and protein diffusion models. One key distinction of their work is that the evolved proteins not only provide growth instructions for the network, but also halt growth. While this method is able to make large neural networks, it can take excessive evolution time as much of the processing power is devoted to simulating chemical diffusion.

Since GAs have been applied successfully in control problems (Yakovenko et al. 2004; Vigraham et al. 2005; Dupuis and Parizeau 2008; Zhang et al. 2008) it may come as now surprise that the synergy of GAs, ANN, and control is a current area of research. Naito et al. (1997) evolved ANN controllers for simulated Khepera (Harlan et al. 2001) robots. Lipson and Pollack (2000); Pollack et al. (2003) have had much success in evolving the morphology and control of robots. Floreano et al. (2007) evolved a swarm of robots which learn complex communication behaviors. Yet, all of these methods use direct representations, and if one were to evolve an ANN complex enough to control an autonomous vehicle(s) (Cremean et al. 2006; Murray 2007), one would need a large ANN and a scalable GA to create it. While Calabretta et al. (1998) and Stanley et al. (2009) have implemented GA with some scalability, their designs scale by using predetermined modules and symmetries, which are not generally known *a priori*.

1.2 Outline

This thesis will detail the methodology, analysis, and implementation of a new genetic algorithm for neuro-evolution. Designs in the GA are grown via a set of variable-length rules that are decoded to create a C^{++} program. The C^{++} programs used to create the ANNs have an *If-CONDITION-Then-ACTION* structure. Each program has multiple sections that cycle through all pairings of nodes with tests and actions of the form:

> *If Node α and/or Node β meet certain CONDITION(S), Then perform ACTION(S).*

The expected result is to create an encoding scheme that, like CE, can take advantage of modularity to create large networks. However, it will also use the innovations of NORGEV to evolve a more controlled growth as well. Having the growth directed by C^{++} programs comprising various recombinations of *If-Then* statements instead of solutions of complex diffusion equations will lead to shorter evolution times. While Neuro-Evolution Using Recombinational Algorithms and Embryogenesis, or NEURAE, may seem akin to the GE of Tsoulos et al. (2005), the work presented here is only superficially similar. Limiting the evolution to only *If-Then* commands constrains the search while remaining flexible enough to explore highly productive regions of the solution space. Furthermore, the programs generated by NEURAE are the rules for embryogenesis, which provide scalability and produce modularity. Conversely, the programs created by conventional GAs are direct representations of an ANN, and do not exhibit such scalability or modularity.

This thesis will show that NEURAE is a unifying GA capable of accomplishing a wide range of neuro-evolutionary goals. Chapter 2 will introduce the methodology of NEURAE after a brief background of artificial neural networks and genetic algorithms. Chapter 3 will show that NEURAE is capable of evolving two types of parity evaluators. The first is a 2-input XOR gate with many network redundancies. The second is a parity gate of an arbitrary size. The first task has definitive exploration versus exploitation regions, which simplifies the analysis of the evolved rules. Furthermore, it will be shown that modularity can be produced in a randomly changing environment, in opposition to Kashtan and Alon (2005). The second task can be directly compared to existing literature, particularly that of Gruau (1994), and will show how NEURAE can scale well to create large ANNs.

Chapter 4 will analyze how and why NEURAE works in an effort to make the evolutionary process more efficient. Like evolutionary algorithms themselves, many of the mutations used in NEURAE where inspired by natural mutations. Experiments were conducted to verify if and how the artificial mutations actually enhance evolution as well as their biological counterparts are theorized to do. Next is an analysis of the individual created in a good and failed evolution to see what differences lie on a genomic level. Finally, an investigation was conducted to see how different conditions and actions are used, and how their removal affects the GA.

Chapter 5 will give the derivations of the formulas used to create the robotic simulations in Chapter 6. Chapter 6 will show how NEURAE is able to evolve robotic controllers in deceptive design domains. NEURAE will easily make controllers for a line following robot, and obstacle avoiding robot, and a coordinated swarm without any changes to its core functionality. Chapter 7 will provide a conclusion and the possible future of NEURAE.

Chapter 2

Methodology

2.1 Background

2.1.1 Neural Networks

An artificial neural network (ANN) is a computing paradigm which is a gestalt of simple computational units called neurons or nodes. All ANNs in NEURAE are composed of McCulloch and Pitts (1943) modeled neurons. The input to each neuron is multiplied by some scalar, or weight, w_n. Next, the weighted inputs are summed and are in turn used as the input, u, for a (usually) nonlinear activation function $O(\cdot)$, as shown in Equation 2.1. In the original McCulloch-Pitts model, the nonlinearity could be any bounded function. Due to the desire to make learning algorithms easier to prove and implement, the activation function usually forces the output of the neuron to be within [-1, 1]. This, however, is not a requirement and an activation function that bounds the output between 0 and 1 can be used. Furthermore, digital networks usually use a discontinuous activation function while $O(\cdot)$ in an analog network would likely be continuous (Kartalopoulos 1996). Finally, neurons usually feature a constant, or bias, which is also summed to the inputs and serves to shift the activation function along the dependent axis.

$$u = \sum_{i=1}^{n} w_i \tag{2.1}$$

The neurons in NEURAE use the activation function shown in Equation 2.2. The activation function, $O(\cdot)$, is a Heaviside function with a bias which acts as a threshold and separates the on/off regions at the constant, t. Thus, each neuron in NEURAE is either completely off or on. Even though the bounded output of each neuron may be

weighted before it is used as an input to another node, $O(u)$ for an output neuron is always unweighted, resulting in a binary output for the entire ANN. The model of neurons used in NEURAE is shown in Figure 2.1.

$$O(u) = \begin{cases} 1 & \text{if } u > t, \\ 0 & \text{if } u \leq t. \end{cases} \tag{2.2}$$

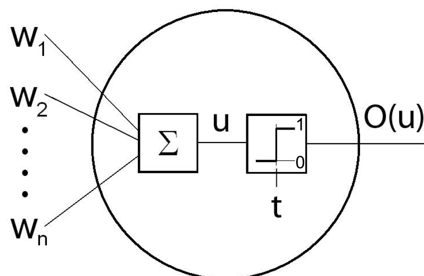


Figure 2.1: McCulloch-Pitts neuron model.

2.1.2 Genetic Algorithms

Genetic Algorithms (GAs) are a class of evolutionary computation, and repeatedly reiterate randomly created designs to find a desired solution. The design solutions are commonly referred to as individuals, and the goal is to eventually create individuals that are capable of solving the design problem. Figure 2.2 is a simplified flowchart of the various steps contained within a standard GA. GAs begin with an initial population of individuals with randomly created genomes. For all GAs there is a difference between the genotype and phenotype. The genotype dictates the design parameters of the individual, and it is the altering of the genotype that ultimately alters the design parameters of the solution. The phenotype, however, is the realization of the individual, and it is the phenotype which is evaluated. Thus, the individuals' fitnesses are based upon how well their phenotypes complete the design challenge.

However, the randomly created initial population is made up of poorly performing individuals. The best performing of these individuals are selected from the population. These selected individuals are slightly modified to create a new population. This process of eval-

uation, selection, and mutation is repeated until either a prescribed time limit has passed or a good design is found.

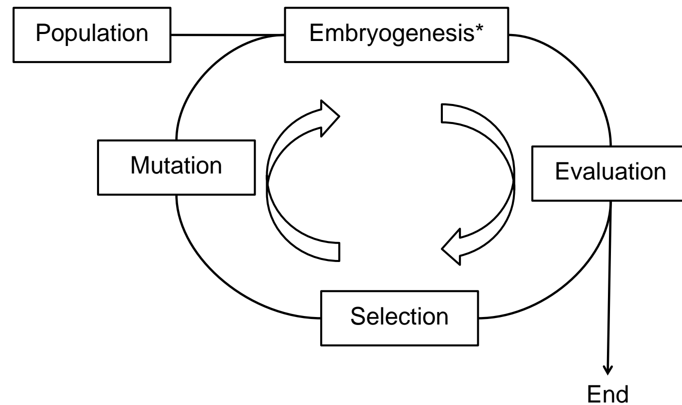


Figure 2.2: Steps of a standard genetic algorithm.
(*Denotes an optional step).

The way individuals are represented, or encoded, within a GA is of paramount importance to how they are evolved. As the encoding becomes more complex, the genotype to phenotype mapping becomes a more involved process known as embryogenesis in which the phenotype starts as a small embryo, then grows according to its genome before or even during evaluation. Stanley and Miikkulainen (2003) offer classifications for the different types of genomic encoding within present-day GAs.

- **Direct -** The design parameters of the phenotype are represented *directly* within the genotype. The approach works well for optimizing a design parameter, but the one-to-one, genotype to phenotype relationship makes scalability a significant problem. Also, the lack of inherent modularity and symmetry makes it a poor candidate for design synthesis.

- **Developmental -** The genotype is a compacted representation of the phenotype, and makes the phenotype by using a prescribed set of rules. This can scale well and takes advantage of known modularity and symmetry. However, evolution is unable to discover and exploit unknown symmetries. Furthermore, the way modularity and symmetry are used to compact genomic representation can unduly bias or even limit the solutions acquired.

- **Implicit -** The genotype is the rules that, when executed, create a phenotype from

an embryo. This approach offers the widest range of possible answers, and thus is the best method for generating completely novel designs. However, optimization is hampered by the strongly non-injective mapping between the genotype and phenotype. Evolution times can also be slowed by extended periods of embryogenesis.

For NEURAE, an implicit encoding scheme was decided to place as little restriction as possible on the type of ANNs created. Thus, many of the examples of NEURAE exemplify the creation of novel network architectures rather than the optimization of well-known ANN problems.

2.2 The NEURAE Genotype

2.2.1 Overview

Each individual in NEURAE is a digital, feed-forward neural network. However, the implicit encoding scheme of NEURAE means each ANN is created by the execution of the rules encoded in its genome. When the genomes are decoded, the result is a C^{++} program. When the program is compiled and executed, the ANN is created.

The neural networks begin as a few neurons, but are grown according to the instructions encoded within their genomes. All ANNs start as the desired number of input neurons with a threshold of 0. Each input node is able to create up to seven addition neurons. These subsequent neurons can exist within either the hidden or output layers, and can each make up to seven addition hidden or output neurons. However, once the desired number of output nodes are created, the entire ANN is unable to create any additional neurons.

Each neuron can also make connections, and can continue to do so even after no more neurons can be created. To ensure the ANNs are feed-forward, nodes are only able to make connections to neurons created after themselves. Furthermore, connections to any input node are prohibited. While nodes within the same hidden layer are unable to connect to each other in most ANN applications, no such constraint is imposed here. Neurons within the hidden layer are able to connect to any other node within the hidden layer so long as the receiving node was created after the transmitting node. Finally, each neuron can have a maximum of 99 inputs and 99 outputs.

2.2.2 Biological Analog

A biological analogy was the inspiration for the encoding scheme used here. The genome of each individual is a variable-length array of integers which is decoded to create a C^{++} program. Every digit is analogous to a *nucleotide* whose value is inclusively between 1 and 100. A collection of six nucleotides forms a complete *If-CONDITION-Then-ACTION* statement, and are analogous to a *codon*. These tests in the *If-Then* statements are not independent, and the sequence of codons will greatly influence how the individual will grow. In particular, the *If-Then* structure can be arranged such that multiple conditions are tested before an action can be executed. The closure of all *If-Then* statements, condition tests, and actions form a block analogous to a *gene*. The resulting (closed) *If-Then* statements in the C^{++} programs are similar to *proteins*. These concepts are shown in Figure 2.3.

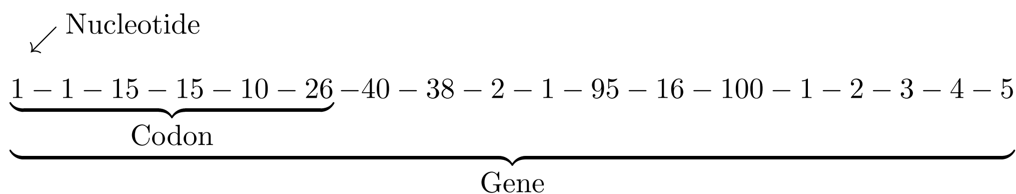


Figure 2.3: Sample genome and biological analog

2.2.3 *If* Structure Nucleotide

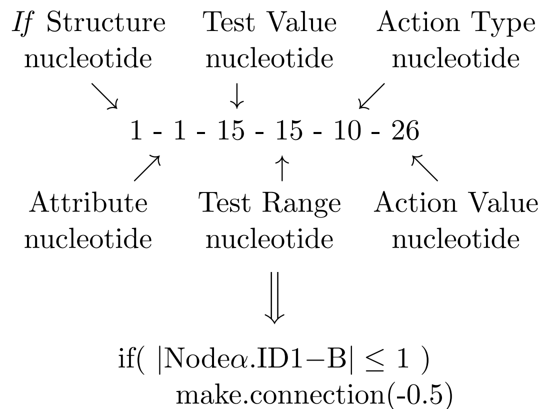


Figure 2.4: Nucleotides of each codon

The first nucleotide of each codon dictates the overall logic of the corresponding C^{++} program. As shown in Figure 2.5, a simple change in the order or nesting of the *If-*

CONDITION-Then-ACTION tests can have a large effect on the computational process. This flexibility allows the GA to build complex algorithms from simple building blocks.

The logic corresponding to the numerical value of the first nucleotide is listed below.

- *If* - Opens an *If-Then* statement. Adds action to the action stack. *Nucleotides* $[1-25]$
- *End-If* - Writes in and removes last action placed into the action stack. Closes an *If-Then* statement. Opens another *If-Then* statement. Adds action to the action stack. *Nucleotides* $[26-40]$
- *End-End-If* - Writes in and removes last action placed into the action stack. Closes an *If-Then* statement. Executes and removes last action placed into the action stack stack. Closes another *If-Then* statement. Opens an *If-Then* statement. Adds action to the action stack. *Nucleotides* $[41-55]$
- *End* - Writes in and removes last action placed into the action stack. Closes an *If-Then* statement. *Nucleotides* $[56-75]$
- *End-End* - Writes in and removes last action placed into the action stack. Closes an *If-Then* statement. Executes and removes new last action placed into the action stack stack. Closes an *If-Then* statement. *Nucleotides* $[76-90]$
- *End-All* - Writes in and removes last action placed into the action stack. Closes an *If-Then* statement. Repeats until all *If-Then* statements are closed. *Nucleotides* $[91-100]$

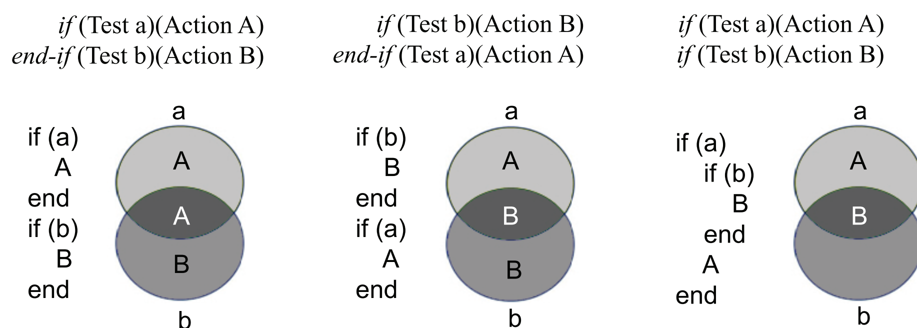


Figure 2.5: If structure codon and protein transcription

2.2.4 Condition Nucleotides

The next three nucleotides determine which of the ANN states that can cause actions to occur will be tested. The second nucleotide in each codon dictates which attribute will be tested. The attributes are current states of Node α and/or Node β. Many of these attributes affect the functionality of the neural network, such as the threshold of the neuron or the number of connections it has. However, each node also has a three-part identification number that aids in evolution without affecting the functionality of the neuron. The first part of the identification number (ID1) is denoted by a letter between A and H. Input nodes all have an ID1 of A and output nodes all have an ID1 of H. Hidden nodes can have an ID1 of B through G, which is determined explicitly by the action which creates it. A node's second ID number (ID2) is determined by the parent node which created it. If this is the first node the parent node has made, the new node will have an ID2 of 1. If it is the third node the parent node has made, the new node will have an ID2 of 3. ID2 values can range between 1 and 8 since any node can make, at most, 8 other nodes. ID3 values denote how many nodes within the entire network have the same ID1 *and* ID2 values. Thus the first node with an ID1 value of B and an ID2 value of 5 will have an ID3 value of 1, while the second node with the same ID1 and ID2 values will have an ID3 value of 2. These values can range from 1 to 100. The result of the three different ID types is that each node will have a unique identification number.

The following list presents all possible node states which can be used by the attribute nucleotide. In addition to using the explicit values of Node α and/or Node β, relative differences between the two nodes can be considered as well. For values where a state of Node α relative to Node β or *Rel* $\alpha\beta$ are considered, the attribute of Node β is subtracted from the value of the same attribute of Node α.

Similarly, there are options to consider the attributes of Node β relative to Node α, or *Rel* $\beta\alpha$. This can apply to all of the attributes listed above except for the connection weight. The value used for connection weight is the value of the weight from Node α to β or vice versa. The nucleotide ranges are for [*Node* α] [*Node* β] [*Rel* $\alpha\beta$] [*Rel* $\beta\alpha$]. Equation 2.3 is used to get discrete values between ± 1, excluding 0, where z is the nucleotide and v is the value written into the C^{++} program.

The test attributes corresponding to the numerical value of the second nucleotide are

listed below.

- *ID1* - Takes the ID1 value of a node, which can be between A and H. *Nucleotides* $[1-5][27-31][53-55][77-79]$
- *ID2* - Takes the ID2 value of a node, which can be between 1 and 8. *Nucleotides* $[6-10][32-36][56-58][80-82]$
- *ID3* - Takes the ID3 value of a node, which can be between 1 and 100. *Nucleotides* $[11-14][37-40][59-61][83-85]$
- *Threshold* - Takes the threshold of a neuron. Due to Equation 2.3, this can be a number in the range [-1 – 1]/0 in 0.02 increments. *Nucleotides* $[15-17][41-43][62-64][86-88]$
- *Number of Nodes Made* - The number of subsequent nodes a node has made. Can be between 1 and 8. *Nucleotides* $[18-20][44-46][65-67][89-91]$
- *Number of inputs* - Number of inputs into a node. Can be between 0 and 99. *Nucleotides* $[21-23][47-49][68-70][92-94]$
- *Number of outputs* - Number of outputs from a node. Can be between 0 and 99. *Nucleotides* $[24-26][50-52][71-73][95-97]$
- *Connection weight* - Takes the weight of a connection between two nodes. Due to Equation 2.3, this can be a number in the range [-1 – 1]/0 in 0.02 increments. *Nucleotides* $[74-76][98-100]$

$$v(z) = \begin{cases} \frac{z-50}{50} & \text{if} \quad z \geq 51, \\ \frac{z-51}{50} & \text{if} \quad z < 51. \end{cases} \tag{2.3}$$

The third nucleotide writes the appropriate value into the test. In order for a condition test to return textit/true, the attribute (second) nucleotide must be within a certain range of this test value nucleotide. The values written into the program depend on the attribute being tested. If the possible range is [0, 99], the number written into the program is the test value nucleotide minus 1. However, attributes that have only 8 possible values require equation 2.4 to convert the test value nucleotide into values suitable for the comparison.

For threshold and connection values, Equation 2.3 is used if the attribute is a connection or the threshold of a neuron. However, if the attribute is the relative threshold of a neuron, Equation 2.5, which gives a range of [0, 1.98], is used instead.

$$v = \left\lfloor \frac{z-1}{12.5} \right\rfloor, \tag{2.4}$$

$$v(z) = \frac{z-1}{50}. \tag{2.5}$$

The fourth nucleotide determines the range over which the attribute can vary from the test value and still have the condition return true. Similar to the test value nucleotide, the test range the nucleotide writes into the code depends on the attribute being tested. For cases where letters are compared, this is the lexicographical range between the letters where two sequential letters have a lexicographical difference of 1.

2.2.5 Action Nucleotides

The final two nucleotides determine which actions are performed if the condition test is true. The fifth nucleotide determines which type of action will be placed into the action stack. As mentioned above, the last in the "stack" of actions is written into the program whenever an *If-Then* statement is closed. Some nucleotides will result in the creation of a new node. Others will create a connection between Node α and Node β. In both these cases, the action value nucleotide dictates the threshold of the new node or weight of the connection, respectively. The nucleotide-to-program transcription options are given by Equation 2.3. However, there are also *No Action* and *End Turn* action type nucleotides which will not insert any new action commands and end the pairing permutation, respectively. In these cases, the action value nucleotide is not used for anything. Figure 2.6 shows the genetic string used to create a C^{++} program.

2.2.6 C^{++} Programs (Proteins)

Each C^{++} program is a collection of proteins that build the phenotype. While the genome creates the bulk of the algorithm, there are a few rules hard-coded into the C^{++} program of every individual. These hard-coded rules are implemented to impose the minimum constraints any viable feed-forward ANN must have, while leaving enough flexibility to create

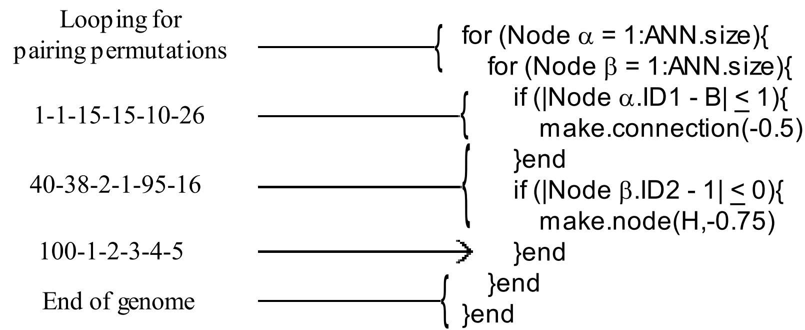


Figure 2.6: Sample genome and protein pseudocode

a variety of architectures. First, the test statements described in the previous section are always placed within two *for* loops which cycle through all the different pairs of the ANN. Also, all of the inputs nodes have a ID2 value of 1. As there is no option to create another input, each ANN will have the same number of input nodes.

However, there are also other mandatory conditions that must be met before an action is executed, even if the *CONDITION* within the genome is true. For actions that make a connection, the first test is to make sure the two nodes are not already connected. Next, the process ensures that the neuron being connected to is not an input to the entire ANN, and that the neuron being connected from is not the output for the entire ANN. Finally, there is a check that the neuron being connected to was made before the neuron which spawned the connection to ensure the ANN is feed-forward.

To keep ANN size reasonable, ANNs have a limited amount of energy available for growth. The act of creating a node or connection consumes one of the predetermined energy units for the entire ANN. Once a pairing executes an action that uses an energy unit, that pairing is over. The individual is considered to be completely developed once the individual uses all 200 energy units or the programs cycles through all pairing permutations without performing any actions. Figure 2.7 shows the development of a NAND gate using the pseudo-code from Figure 2.6. It is important to note that an infinite number of different genomes could have created an identical ANN.

```
for (Node α = 1:ANN.size ) {
   for (Node β = 1:ANN.size ) {
      if ( |Node α.ID1 - B| ≤ 1 ){
         make.connection(-0.5)
      }end
      if ( |Node β.ID3 - 1| ≤ 0 ){
         make.node(H,-0.75)
      } end
   } end
} end
```

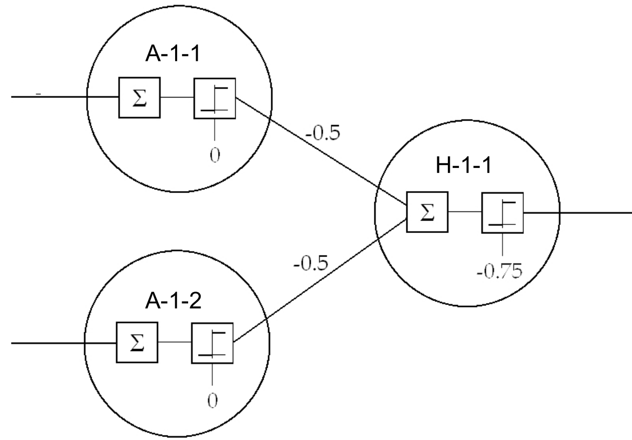


Figure 2.7: Protein pseudocode and sample NAND gate

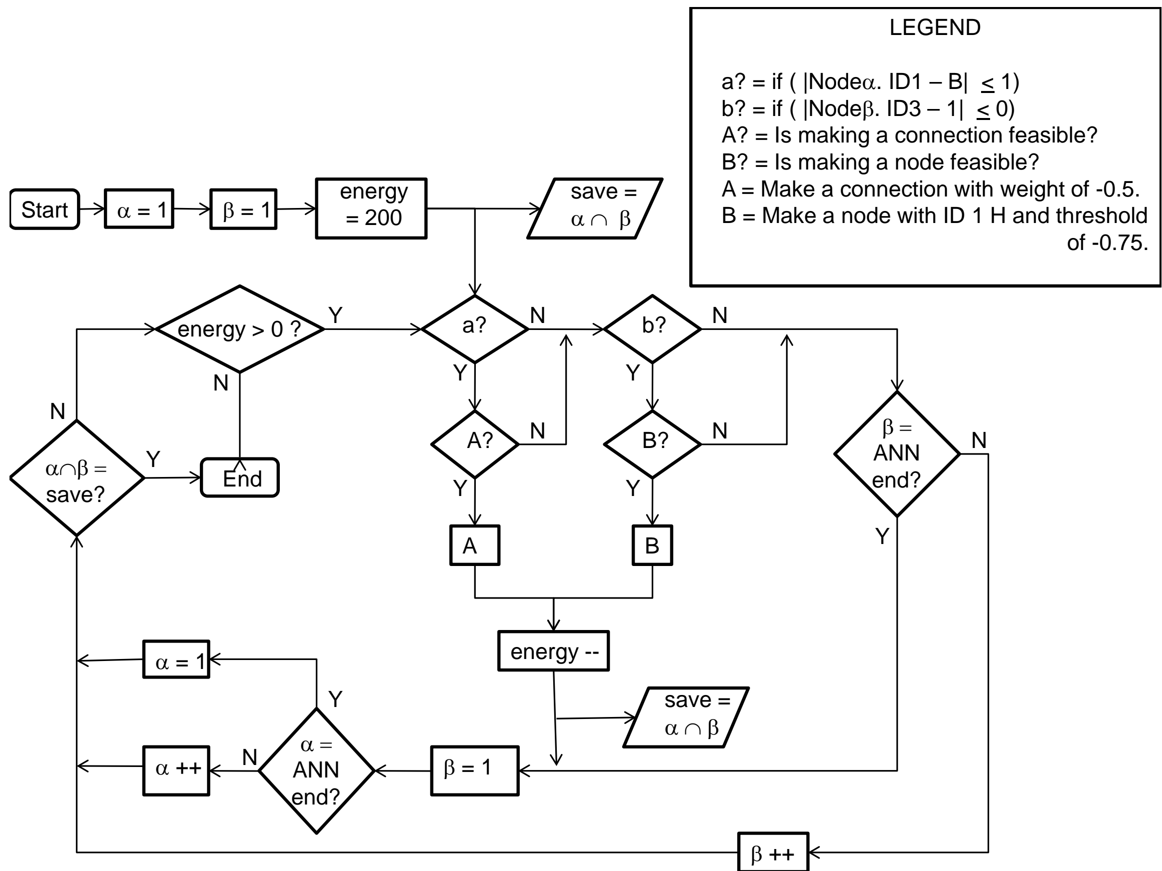
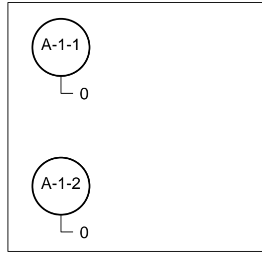
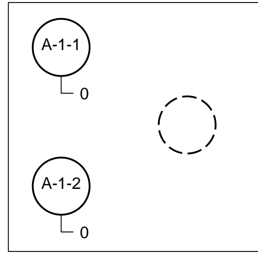


Figure 2.8: Flowchart of protein pseudocode

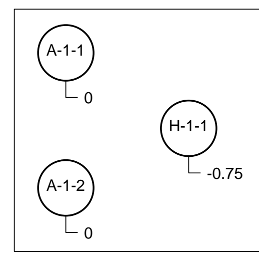
EMBRYOGENESIS START



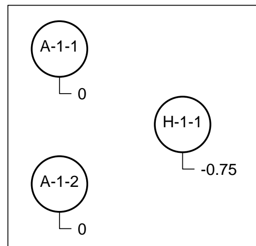
Step 1:
Neuron α = A-1-1
Neuron β = A-1-1
Action: Make Node



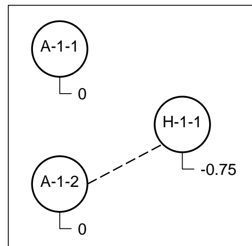
Step 2:
Neuron α = A-1-2
Neuron β = A-1-1
Action: None



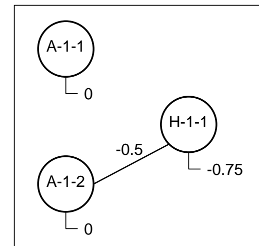
Step 3:
Neuron α = A-1-1
Neuron β = A-1-2
Action: None



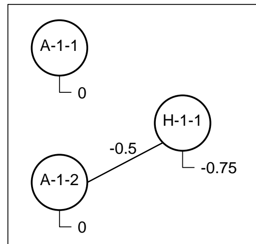
Step 4:
Neuron α = A-1-2
Neuron β = H-1-1
Action: Make Connection



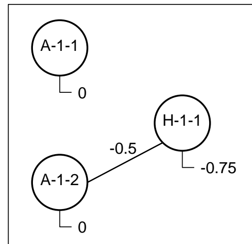
Step 5:
Neuron α = H-1-1
Neuron β = A-1-1
Action: None



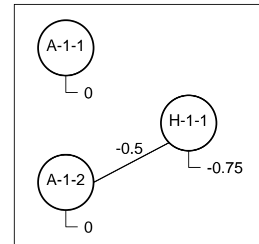
Step 6:
Neuron α = H-1-1
Neuron β = A-1-2
Action: None



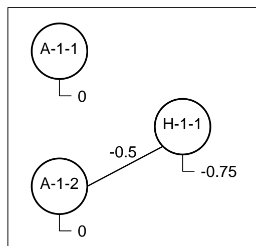
Step 7:
Neuron α = H-1-1
Neuron β = H-1-1
Action: None



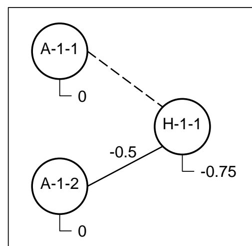
Step 8:
Neuron α = A-1-1
Neuron β = A-1-1
Action: None



Step 9:
Neuron α = A-1-1
Neuron β = A-1-2
Action: None



Step 10:
Neuron α = A-1-1
Neuron β = H-1-1
Action: Make Connection



EMBRYOGENESIS FINISHED

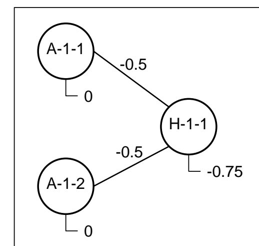


Figure 2.9: Steps showing the embryogenesis of NAND gate

2.3 Evaluation, Mutation, and Selection

Each ANN is evaluated after the embryogenesis of each individual, as described by the method above. Evaluations in NEURAE are performed in tiers to ensure network feasibility and to promote evolution of complex behaviors (Graham et al. 2009).

The first tier ensures the individual grows the correct number of output nodes. If the correct number of outputs are made, the individual advances to the second tier, where the exponent is increased for each output node with a connection. These two requirements, listed in Table 2.1, are the minimum for any possibly viable ANN circuit, and once met, will yield an exponent value of $x - 1 = 1$. The remaining tiers vary depending on the design problem, and are listed alongside the design problem to which they pertain.

Table 2.1: Universal tiers for adjusting fitness exponent (x)

Tier	Test	Change in Exponent
1	Are there enough output nodes?	fraction of desired output nodes
2	Are there a connections to each output node?	+ fraction of output nodes with connections

Another commonality all evaluations share is the fitness function shown in Equation 2.6. While x is a linear comparison of two individuals, the exponential nature of Equation 2.6 magnifies any improvements and greatly improves convergence in NEURAE. Furthermore, the floor function ensures individuals which are unable to pass the first tier have zero fitness, virtually nullifying their odds of survival.

$$\text{Fitness} = \left\lfloor 2^{x-1} \right\rfloor . \tag{2.6}$$

A roulette style of selection determines which individuals are used for creating the next generation. The population size in each generation is conserved. The probability of selecting an individual is determined using Equation 2.7; where P_i, f_i, and N are the probability of selecting the ith individual, the fitness of the ith individual, and the population size, respectively. A quarter of the population of the current generation survives to the next generation. The remainder of the population is created by using the operations of point

mutation, conjugation, translocation, genome replication, and genome deletion.

$$P_i = \frac{f_i}{\sum_{j=1}^{N} f_j}. \tag{2.7}$$

As described by Holland (1992), classical GAs change the genotype of future populations through point mutation and crossover of current individuals. Figure 2.10 shows an example of a point mutation in a binary genome where a random bit is flipped. Point mutations are also used in NEURAE, but instead of a binary bit flip, a random nucleotide is replaced with a randomly chosen integer inclusively between 1 and 100.

$$11\underline{1}0001110\underline{0}0 \quad \Rightarrow \quad 11\underline{0}0001110\underline{1}0$$

Figure 2.10: Point mutation example. The underlined nucleotides are switched

Crossover mutations require two individuals to make two more individuals and are usually either single-point or two-point crossover. With single-point crossover, two individuals make two new individuals by having their genomes broken and swapped at a random location on the genetic string. In two-point crossover, only a section of the genomes are swapped. Figures 2.11 and 2.12 give an example of both types. For GAs in which all genomes must be the same size, the sections to be swapped must be of identical length. Furthermore, the sections are usually at the same genome locus such that the information being exchanged at that locus has some correlation to its purpose in the phenotype. In NEURAE, however, there is little correlation between the functions of the same section of genome between two different individuals. Furthermore, while crossover may produce one improved individual, they seldom create two. Thus, genetic material is shared during mutations in NEURAE through a process inspired by, and named after, biological conjugation.

$$\begin{matrix} 111000\underline{111000} \\ \mathbf{101010}\underline{\mathbf{101010}} \end{matrix} \Rightarrow \begin{matrix} 111000\underline{\mathbf{101010}} \\ \mathbf{101010}\underline{111000} \end{matrix}$$

Figure 2.11: Single-point crossover mutation example. Parts of the genome which have been swapped are underlined

$$\begin{matrix} 1110\underline{0011}1000 \\ \mathbf{1010}\underline{\mathbf{1010}}\mathbf{1010} \end{matrix} \Rightarrow \begin{matrix} 1110\underline{\mathbf{1010}}1000 \\ \mathbf{1010}\underline{0011}\mathbf{1010} \end{matrix}$$

Figure 2.12: Two-point crossover mutation example. Parts of the genome which have been swapped are underlined

In biology, conjugation is a process used by many species of bacteria where one bacterium gives part of its DNA to another. Martin and Russell (2002) showed how this type of genomic

exchange may have been key in the evolutionary jump from prokaryotes to eukaryotes and Jain et al. (1999) and Ochman et al. (2000) offer conjugation as a reason for the high adaptability of present-day bacteria. NEURAE uses conjugation in the manner shown in Figure 2.13, where a section of one genome is inserted into the genome into another. Thus, new rules can be exchanged between individuals and, hopefully, the benefits of biological conjugation can also be used by NEURAE.

111000111000
10101010101010 ⇒ 111000**1010**111000

Figure 2.13: Conjugation mutation example. Parts of the genome which have been inserted are underlined

Ohno (1970) introduced the concept of genome duplication as another key component of biological evolution. During replication, portions of the genome are at times copied more than once, resulting in an offspring that has two genes which make the same protein. Ohno theorized this redundancy made the individual more robust to future mutations, because if one gene became non-functional, there is another copy to do the same job. This redundancy was also noted by Britten (2005), who observed that many sections of the human genome have sequences that are too similar to have arisen independently. NEURAE uses a genome duplication process as shown in Figure 2.14, where a section of a genome is copied more than once when it is being replicated.

111000111000 ⇒ 111000111111000

Figure 2.14: Gene duplication example. The nucleotides copied more than once are underlined

The final two mutation types are gene deletion and translocation. In gene deletion a section of the genome is removed during replication. While gene deletion is an observable phenomenon in biology, its effects are usually damaging (Lewis 2005). However, it was added as a mutation here to counter the concatenating effects of conjugation and gene duplication. Translocation, where a section of the genome is moved to another locus, is yet another observed biological mutation. Regardless of its implications to biological evolution, Figure 2.5 shows that the order of rules are very important in the embryogenesis of an individual, so an operation which varies this order was included. Figures 2.15 and 2.16

show examples of these two processes in NEURAE.

111000111000 ⇒ 111000000

Figure 2.15: Gene deletion example. The nucleotides deleted are underlined

111000111000 ⇒ 111111000000

Figure 2.16: Translocation example. The underlined nucleotides are moved to another gene locus

Finally, it was necessary to prevent frame-shift mutations. A frame-shift mutation adds or deletes only part of a codon. The result is a shift in nucleotides that causes all following codons after the mutation to be different. Ohno (1970) detailed how such mutations are almost always deleterious in biology and care is taken to avoid them here.

Chapter 3

Logic-Gate Evolution

3.1 Overview

This chapter will describe how NEURAE creates logic gates. Each evolutionary run begins with the random creation of 200 individuals for 1000 generations. These values were found to give good results in run times around 4 hours on a cluster of 25 dual quad-core, 2.33 GHz computers. Furthermore, each individual started with a genome 300 nucleotides (50 codons) long. During evolution, a genome is allowed to double in size before being trimmed to the default length. Genome length was constrained to prevent the well-documented problem of bloat in genetic programming (Koza 1992; Langdon 2000). While this arbitrary setting of genome length may bias evolution, Szathmáry and Smith (1995) have evidence showing that overall genome length of a biological organism has little to do with the complexity of the phenotype.

The first goal is to evolve an ANN that can serve as an XOR logic gate (Table 3.1), even if the ANN suffers multiple failures. This circuit was chosen because its nonlinearity requires the creation of a hidden layer and is a common benchmark in the evolution of ANN logic circuits (Koehn 1996; Ashlock 2006). The next logic gate to be evolved is a parity gate. A parity gate is a standard logic circuit used in simple error detection. An even parity logic circuit will always have an even number of inputs and output active. This design challenge exemplifies NEURAE's capability to make a scalable ANN.

Table 3.1: Desired output pattern for XOR logic-gate

		Input 2	
		0	1
Input 1	0	0	1
	1	1	0

3.2 Robust XOR Gate

3.2.1 Evaluation Parameters

Table 3.2 shows the tiers used in evaluating the evolved XOR gates, the exponent gets an additional point for each correct answer. If an individual is able to get to the third tier, the exponent in Equation 2.6 has a value of $x - 1 = 1$. At this point, the network's truth table is compared with that of the desired circuit in tier 3. If the individual passes tier 3 and is a functional XOR gate, $x = 6$ and the individual will have an overall fitness of 32. In tier 4, a node is randomly removed, and the ANN is compared to the target XOR logic again. Nodes are continually removed until the circuit no longer produces the target logic. This test for robustness is performed for each generation the individual is alive. Because the order in which the nodes are removed changes with each generation, the fitness of an individual is not constant, and the overall robustness will increase.

Table 3.2: Tiers for adjusting fitness exponent (x) in robust XOR evolution

Tier	Test	Change in Exponent
1	Are there enough output nodes?	fraction of desired output nodes
2	Are there a connections to each output node?	+ fraction of output nodes with connections
3	Compare to the desired truth table	+ # of correct answers in each table entry
4	Break nodes until failure	+ fraction of nodes broken

3.2.2 Evolution Results

Figure 3.1 shows the fitness of the best individual of each generation. Figure 3.2 shows the first XOR gate synthesized by evolution in generation 823, and Figure 3.3 shows how it functions. In these figures, a node is filled-in (black) when it is activated. A solid connection indicates a positive weight while a dashed connection is indicative of a negative weight. As shown in Figure 3.3, the activation of either input will activate only the output. Once both nodes are on, three of the four hidden nodes are activated, and their inhibitory connections to the output are enough to deactivate it. However, this ANN is not robust, as all three hidden nodes are needed to counter the activation of both inputs, and the removal of any one will break the entire ANN.

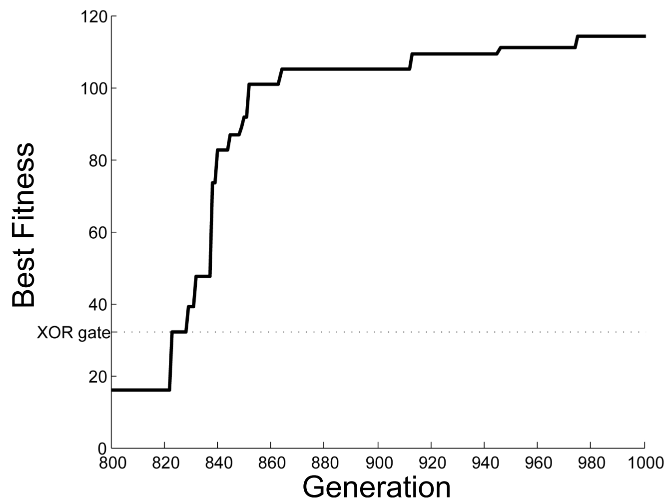


Figure 3.1: Best fitness throughout the evolution of a robust exclusive-OR logic gate

By the end of the evolutionary run, a much larger ANN was created and is shown in Figure 3.4. This ANN comprises 49 nodes and 140 connections. The algorithm created this ANN by taking the smallest possible XOR gate (shown in Figure 3.5) and making duplicate copies of it. The resulting ANN can have all but one hidden node removed, and is as robust to node removal as possible. Furthermore, the ANN used 189 out of the 200 possible energy units, making it close to the maximum size this evolution would allow.

Nevertheless, this is not the largest, fully redundant ANN this genetic algorithm could have made. Figure 3.6 shows a refined version of the individual's code, which shows only the proteins used in making the ANN. The last protein in the code is responsible for making

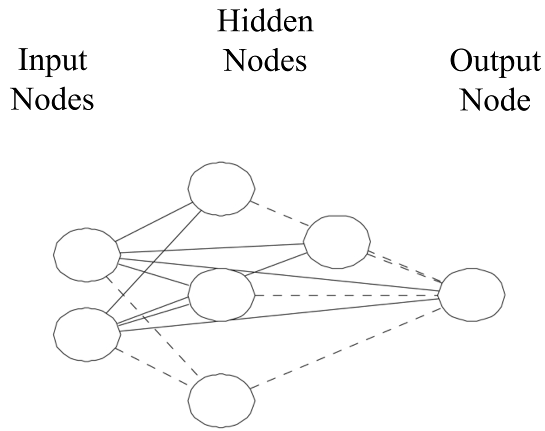


Figure 3.2: First generated XOR gate

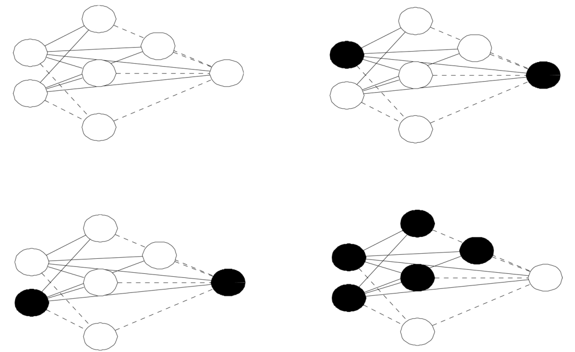


Figure 3.3: Network functionality

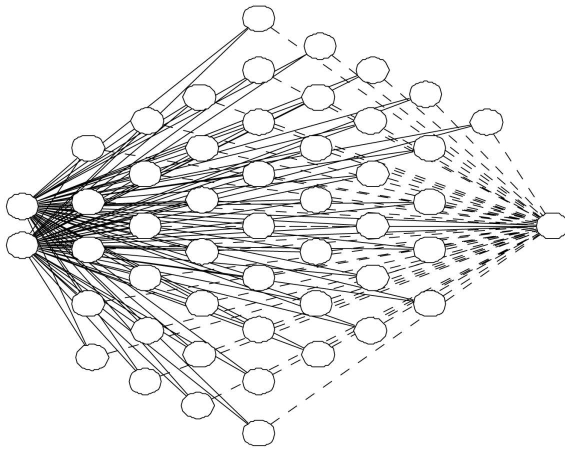


Figure 3.4: Best generated XOR gate

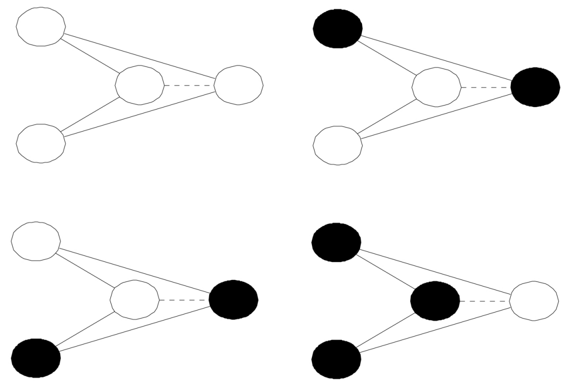


Figure 3.5: Network functionality

the output node, which in turn halts all further neuron growth. If the test value is increased from 3 to 5, and the maximum number of energy units available for growth is not limited, then the 195 node network shown in Figure 3.7 is produced.

```
for (Node α = 1:ANN.size ){
   for(Node β = 1:ANN.size ){
      if |Relβα.ID1 - 6| ≤ 6{
         if |Relβα.threshold - 1.46| ≤ 0.82 {
            make.connection(0.90)
         }
         make.node(D,0.92)
      }
      if |Relβα.ID1 - 6| ≤ 4{
         make.connection(-0.94)
      }
      if |Relαβ.ID2 - 3| ≤ 0{
         make.output(H,0.86)
      }
   }
}
```

Figure 3.6: Code for creating a robust XOR gate

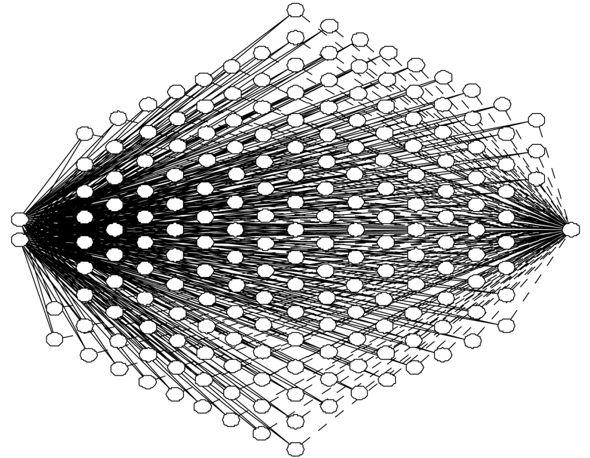


Figure 3.7: Larger XOR gate

The results of this experiment show that NEURAE is able to create large and complex network structures. Not only is this GA able to solve the standard benchmark in logic neuroevolution, it was able to expand on it by finding the core module and replicating it. The ability of NEURAE to construct large networks with such regular structure will be key for future applications.

3.3 Large Parity Gate

3.3.1 Evaluation Parameters

Table 3.3 shows that for the creation of a variable-size parity gate, the exponent is increased by the fraction of entries in the truth table that are correct. Here, a 2-input parity gate will have an exponent of $x - 1 = 2$ and a fitness of 4. Once 2-input even parity is developed, the ANN is rebuilt using the same genome, but starts with three inputs. The individual goes through the three tiers again, with the exponent increasing by one for each test. Therefore, a successful three-input parity gate will have an exponent of $x - 1 = 5$ and a fitness of 32. These three tiers are repeated for up to 21 inputs.

Table 3.3: Tiers for adjusting fitness exponent (x) in scalable parity evolution

Tier	Test	Change in Exponent
1	Are there enough output nodes?	fraction of desired output nodes
2	Are there a connections to each output node?	+ fraction of output nodes with connections
3	Compare to the desired truth table	+ fraction of correct answers in each table entry

3.3.2 Evolution Results

The genetic algorithm was also able to create a parity gate for an arbitrary number of inputs. Figure 3.8 shows the fitness of the best performing individual throughout evolution. The particular evolutionary run shown here produced a 2-input parity (i.e., XOR) gate much more quickly than the run shown in the previous section. This large variability is a by-product of the stochastic nature of GAs. At the 621st generation, NEURAE finally generated a fully scalable individual. However, the discovery of this individual resulted in the halting of the GA due to the excessive time required to evaluate $\sum_{n=2}^{21} 2^n$ input configurations. While a more elegant evaluation method could have circumvented this issue (Gruau 1994), the fact still remains that NEURAE was able to solve the problem at hand.

As shown in Figure 3.9, the 2-input parity gate works by having hidden nodes which inhibit the output once both input nodes are activated. The hidden nodes, however, also inhibit the activation of other hidden nodes that were made afterwards. This cascading effect can also be seen in the 4-input parity gate shown in Figure 3.10. The internal cascading structure of the 2-input network is able to scale accordingly to the 4-input network by having the number of hidden nodes equal the number of output nodes. Having two inputs active in the 4-input gate is identical to having two inputs active in the 2-input gate. Activating a third input is able to turn on the output node without activating another hidden node. However, the activation of a fourth input activates another hidden node, which in turn is sufficient to inhibit the excitation of all four inputs. Figure 3.11 shows this cascading effect scales with the number of inputs in an ANN with 13 inputs.

As shown in the code in Figure 3.12 the magnitude of a negative connection is *exactly*

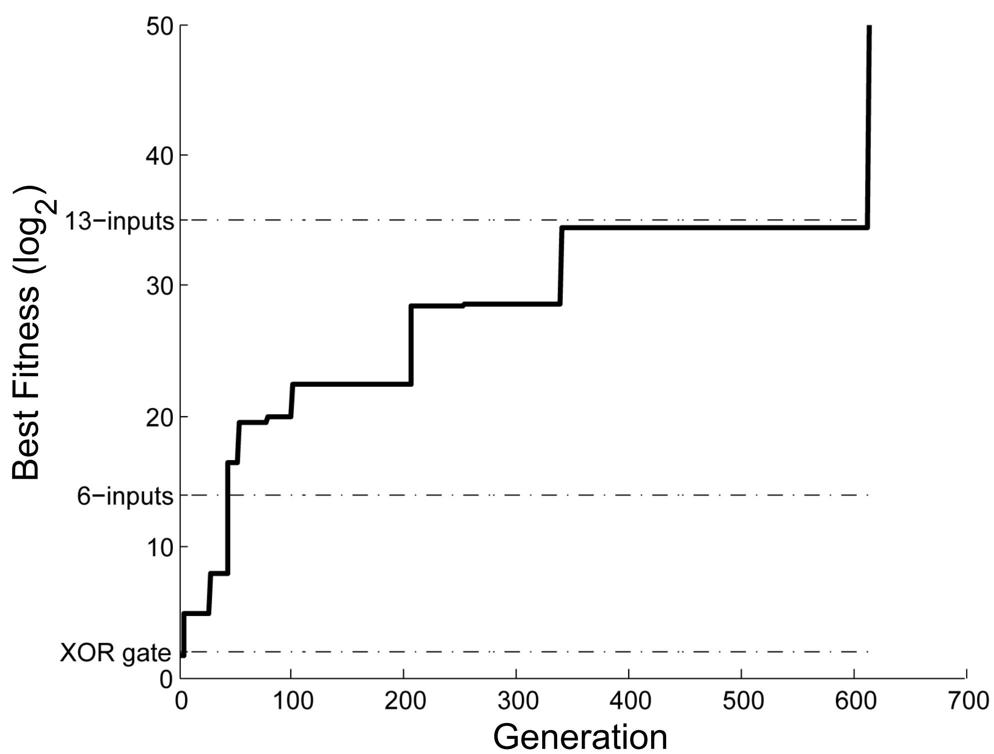


Figure 3.8: Fitness of best-performing individual throughout the evolution of a scalable parity gate

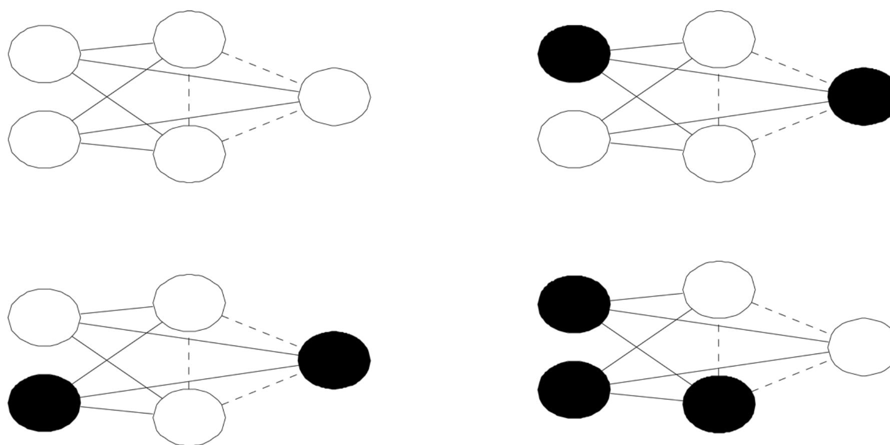


Figure 3.9: Scalable parity gate with two inputs

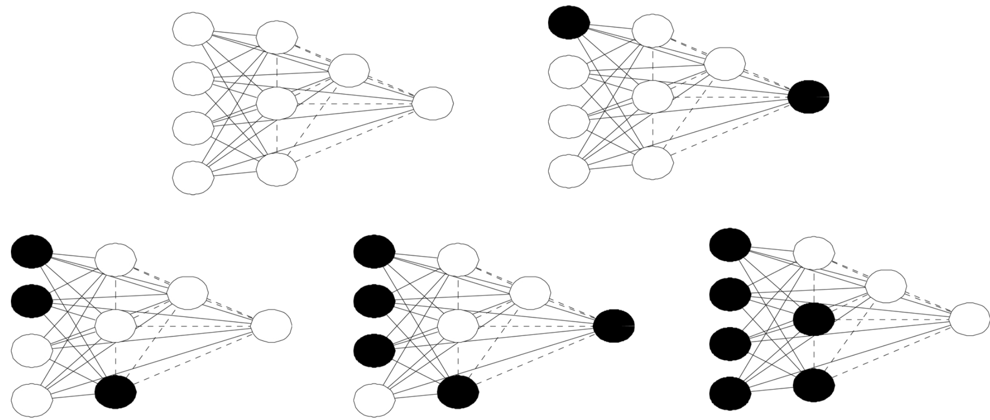


Figure 3.10: Scalable parity gate with four inputs

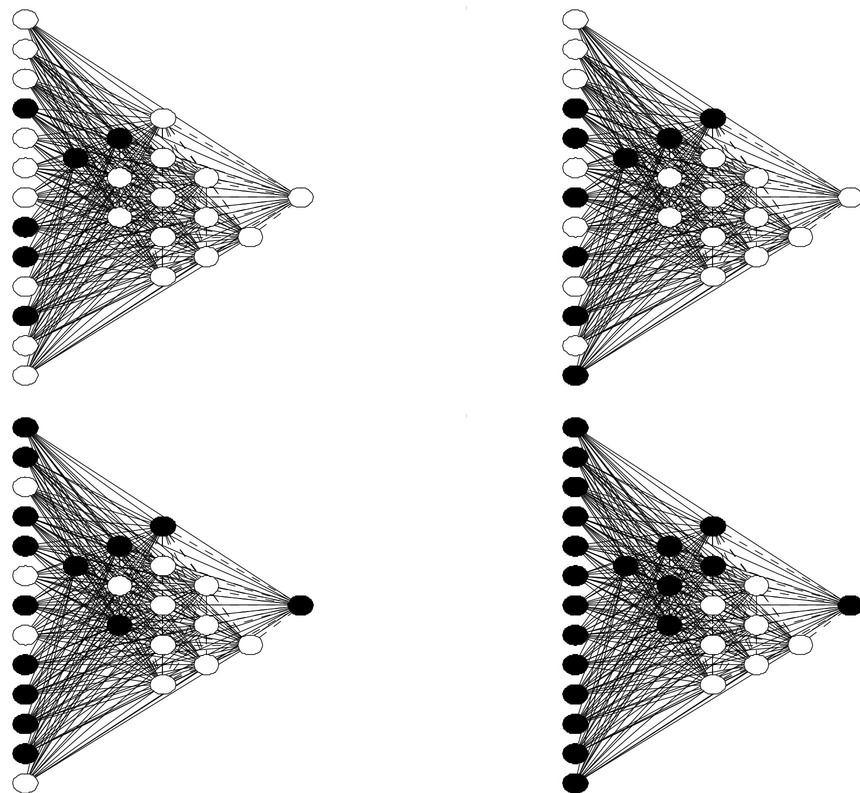


Figure 3.11: Scalable parity gate with 13 inputs

twice the magnitude of a positive connection. Thus the excitation of two input nodes is canceled out by the excitation of one hidden node. Furthermore, as the network begins with more inputs, the number of hidden nodes made during embryogenesis increase as well, providing scalability.

Once again, certain hard limits prevent parity gates of any arbitrarily large size to be created. First, a limit of 200 energy units prevents this network from growing a parity gate with more than 13 inputs. Also, the 99 connection limit placed on the maximum number of inputs and outputs caps the parity gate size at 66 inputs. Fortunately, both these limits were established only to help the evolution process and can be increased as necessary to allow the code in Figure 3.12 to create parity logic for an arbitrary number of inputs.

```
for (Node α = 1:ANN.size() ){
   for(Node β = 1:ANN.size() ){
      if |Relβα.nodes_made - 0| ≤ 0{
         make.connection(-0.96)
      }
      if |Nodeα.nodes_made - 1| ≤ 0{
         make.node(H,0.02)
      }
      if |Nodeβ.ID1 - E| ≤ 3{
         make.connection(0.48)
      }
      if |Relβα.ID2 - 3| ≤ 3{
         make.node(E,0.40)
      }
   }
}
```

Figure 3.12: Code for creating parity gates of arbitrary size

Chapter 4

Sensitivity Analysis

4.1 Mutation Rates

Many of the values used for the genetic algorithm were heuristic. Fortunately, NEURAE is able to solve the robust XOR problem with a wide range of values. Still, as the design challenges for NEURAE become more difficult, it is important to not disadvantage NEURAE by using suboptimal evolutionary parameters. Some parameters, such as population size and number of generations per evolution, are dependent on the computer resources available. However, the mutation rates were arbitrarily chosen, and are likely not the optimum. Furthermore, these mutation values can be adjusted independently of the hardware used and, hopefully, independently of the problem being solved.

NEURAE has a two-step process in determining mutations. After an individual is selected to produce offspring, its genome is scanned using the overall mutation rate, $\mu \in [0, 1]$. Each codon has a probability μ of undergoing some type of mutation. Based on this random selection, when a mutation will occur, NEURAE then randomly selects from the secondary mutation options the type of mutation the codon will undergo. The possible mutations of point, conjugation, duplication (recopy), deletion, and translocation have the respective rates of $\mu_P, \mu_C, \mu_R, \mu_D$, and μ_T.

In order to determine the appropriate balance of the various mutation rates, a series of experiments were conducted. Each series was composed of ten evolutionary runs. Because the creation of an XOR gate is feasible by using only point mutations, a series of tests were run to determine the optimal point mutation rate. These tests set the μ_P rate to 1.0, and varied the μ rate from 0.05 to 1.0. The metrics by which the different tests were judged were the number of generations it took to make an XOR gate and the fitness of the

highest-scoring individual at the end of evolution.

Statistical data for the first generation in which an XOR gate was made, or α generation, was fitted to a two-parameter Weibull distribution (Weibull 1951). A Weibull distribution has the cumulative distribution function (CDF) and probability distribution function (PDF) given in Equations 4.1 and 4.2, respectively. In these equations, k is the shape parameter and λ is the scale parameter. These parameters were found by performing a least-squares line-fit on the data shown in Figure 4.1, where the slope of the line is k, and the x-intercept is λ. Once these values are found the integral of the PDF (Equation 4.2) is used to determine the likelihood of an XOR gate will being created within 1000 generations.

$$F(x) = 1 - e^{-(x/\lambda)^k}, \tag{4.1}$$

$$P(x) = \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k}. \tag{4.2}$$

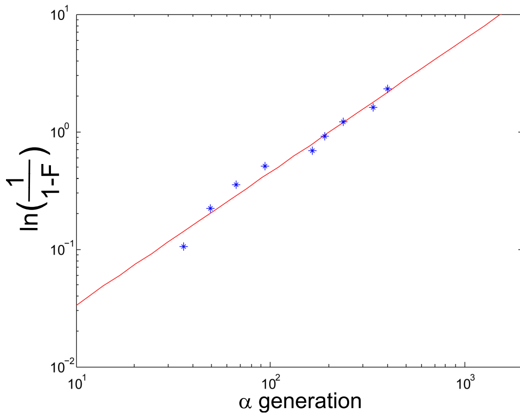


Figure 4.1: Log-log plot of α generation vs. $log\left(\frac{1}{1-F}\right)$ for a point mutation rate of $\mu = 0.4$.

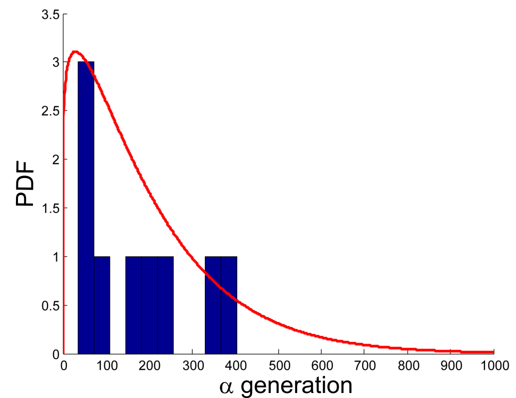


Figure 4.2: Probability density function and histogram of α generation for mutation rate of $\mu = 0.4$.

The Ω fitness is the fitness of the best performing individual at the end of the evolutionary run. Because cases where an XOR is never found are capped at 16, those runs are excluded to focus on the exploitative effects of the mutation rates. This statistical data was found to be best fit to a Gaussian distribution, as shown in Figure 4.3.

Table 4.1 illustrates that evolutions using mutation rates at the extremes are both less likely to make an XOR gate and are worse at optimizing a gate if it does. This is congruent

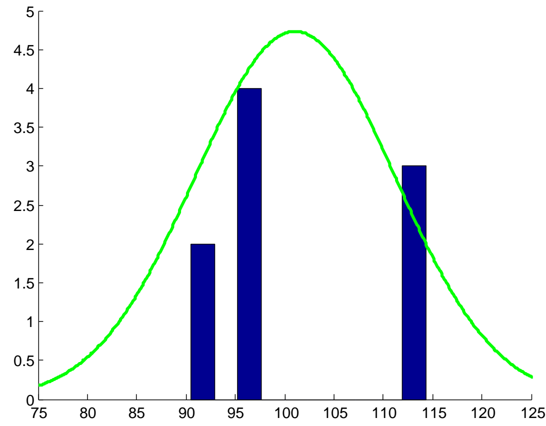


Figure 4.3: Gaussian distribution of best fitness at the end of evolutionary runs with a point mutation rate of $\mu = 0.4$.

Table 4.1: The statistical results for varying mutation rates while only using point mutations

Case	μ	Probability α gen ≤ 1000	Ω fit mean	Ω fit st. dev.
1	0.05	73.1%	99.67	10.36
2	0.1	90.2%	108.8	7.26
3	0.2	92.9%	106.0	8.40
4	0.4	97.3%	101.1	10.12
5	0.6	99.5%	97.39	23.64
6	0.8	99.1%	94.35	18.47
7	1.0	88.1%	86.70	17.76

with other literature which shows that extremely high and low mutation rates are often deleterious to GAs (Mühlenbein 1992; Bäck and Schutz 1996).

However, mutation rates between 0.1 and 0.8 offer a trade-off between the likelihood of finding an XOR gate and optimizing an ANN. As shown in Table 4.1, a higher mutation rate makes finding an XOR gate more likely. However, lower mutation rates are generally more capable of exploiting a functional XOR design and making it robust. Thus, a user can either decide whether the problem being solved is more explorative or exploitative in nature, and choose μ_P accordingly, or use variable mutation rates, such as those shown by McGinley et al. (2008).

It may be possible to improve both the explorative and exploitative capabilities of NEURAE without using a variable mutation rate which comes with its own biases and problems (Bäck 1992). It was hoped that other mutations found in nature would be beneficial to include in NEURAE as well. As mentioned in Chapter 2, NEURAE is capable of altering newly created genomes using mutations besides simple point mutations. A sensitivity analysis was conducted to determine the appropriate rates of the rest of the mutation types. However, the mutation rates are interdependent, so the sensitivity analysis was administered in a manner detailed by Montgomery (2004) for studying the effects of dependent variables. Overall, there are 6 variables. However, there are a few constraints that reduce the degrees of freedom.

The first constraint, Equation 4.3, requires the probability of a point mutation to be held at 0.4. The value of 0.4 was chosen because it is in the middle of the plateau of mutation rates that perform well. Furthermore, the previous experiments prove that the overall mutation rate can be increased without adversely affecting NEURAE.

$$\mu \cdot \mu_P = 0.4. \tag{4.3}$$

Next, the secondary mutation rates must sum to 1, as shown in Equation 4.4. This is to ensure that a mutation happens as the overall mutation rate, μ, dictates. The constraint shown in Equation 4.5 was added because the operations of crossover and gene duplication lengthens the genome while deletion shortens it. Having the mutation rates of these operations balanced makes sure the genomes' lengths are not unduly biased. This constraint, when combined with the constraint that all mutation rates must sum to 1.0, leads to the

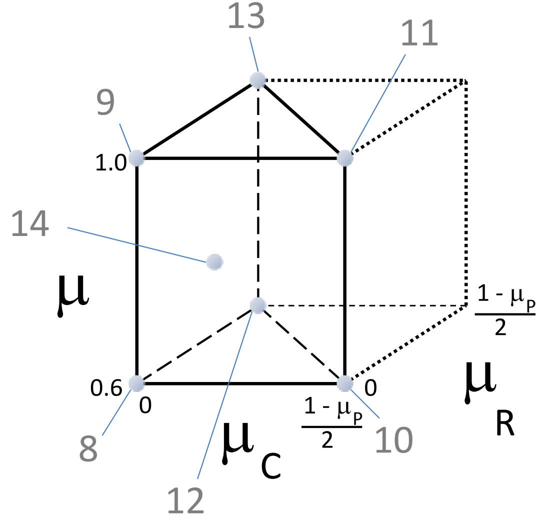


Figure 4.4: The prism is representative of the mutation rate landscape as bounded by the above constraints.

inequality in Equation 4.6.

$$\mu_P + \mu_C + \mu_R + \mu_D + \mu_T = 1.0, \tag{4.4}$$

$$\mu_C + \mu_R = \mu_D, \tag{4.5}$$

$$\mu_C + \mu_R \leq \frac{1 - \mu_P}{2}. \tag{4.6}$$

These constraints can be used to create the mutation rate landscape shown in Figure 4.4 and a 3-dimensional sensitivity analysis can be performed by varying μ, μ_C, and μ_R with data taken at the corners and centroid of the prism to maximize the exploration of the mutation rate landscape. Table 4.2 shows the values used for exploring the mutation rate landscape, which are at the corners and centroid of the prism shown in Figure 4.4.

Table 4.3 offers the results of the mutation rate sensitivity analysis. In general, the excessively high mutation rates ($\mu = 1.0$) were once again the poorest performing. Furthermore, cases that use only point mutations and genome size changing mutations (i.e., conjugation, duplication, and deletion) perform worse than using point mutations alone. However, using only point and translocation mutation with a moderate overall mutation

Table 4.2: Mutation rates for 3-dimensional sensitivity analysis with variables in **bold** are indicative of the chosen points on Figure 4.4

Case	$\boldsymbol{\mu}$	μ_P	$\boldsymbol{\mu_C}$	$\boldsymbol{\mu_R}$	μ_D	μ_T
8	**0.6**	0.66	0.0	0.0	0.0	0.34
9	**1.0**	0.40	**0.0**	**0.0**	0.0	0.60
10	**0.6**	0.66	**0.17**	**0.0**	0.17	0.0
11	**1.0**	0.40	**0.30**	**0.0**	0.30	0.0
12	**0.6**	0.66	**0.0**	**0.17**	0.17	0.0
13	**1.0**	0.40	**0.0**	**0.30**	0.30	0.0
14	**0.8**	0.5	**0.075**	**0.075**	0.15	0.20

rate, as was done in case 8, achieved good results. Still, there is a delicate balance between these values since case 9, which also only used point and translocation mutations, was by far the worst performing test case. This case only had two of the 10 runs produce an XOR gate. Nevertheless, the best combination of mutations rates is case 14, which uses all of the mutation types. These runs have a high probability of discovering an XOR gate (99.95%) coupled with good optimization. As a result, this became the balance of mutation rates used for future design problems.

Table 4.3: The statistical results for varying mutation rates across the mutation rate landscape given in Figure 4.4

Case	Probability α gen ≤ 1000	Ω fit mean	Ω fit St. Dev.
8	97.3%	106.6	8.48
9	19.1%	75.7	40.6
10	86.1%	92.99	18.84
11	56.6%	100.8	8.95
12	71.8%	98.34	10.52
13	56.5%	92.95	5.48
14	99.95%	102.8	10.60

4.2 Qualities of Productive Evolution

While it is important to see which mutation values optimizes NEURAE, an analysis of *why* could help make improvements as well. Thus, a look at two different runs from an earlier version of NEURAE (Roy et al. 2008) were analyzed. Both evolutions were performed

using only point mutations, but one case had a moderate mutation rate ($\mu = 0.2, \mu_P = 1.0$) which often produced XOR gates. The second group had a higher mutation rate ($\mu = 0.8, \mu_P = 1.0$) which seldom produced an XOR gate. Characteristics of successful, XOR producing runs were compared to those of non-XOR producing, unsuccessful runs. While the quantitative results differ between the two groups, the qualitative results for each group are similar.

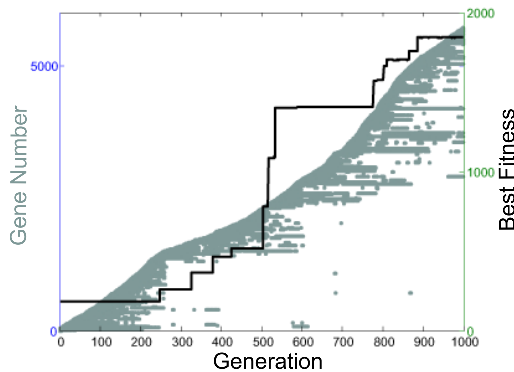


Figure 4.5: Genes used by the top 10% within a successful evolution

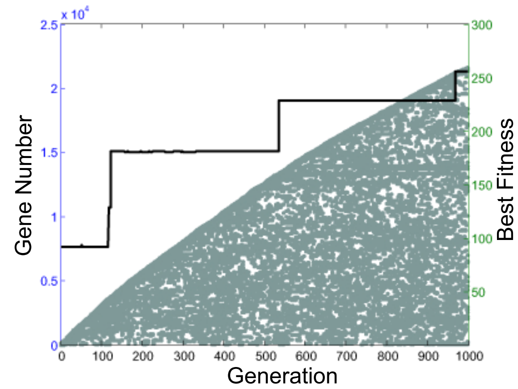


Figure 4.6: Genes used by the top 10% within an unsuccessful evolution

Figures 4.5 and 4.6 show which genes were used by the best individuals (top 10%) throughout evolution. Each time a gene is used, a dot is placed that shows in which generation it was used. Furthermore, the figure is overlaid with a plot of the fitness of the best performing individual of each generation.

In Figure 4.5 there are sudden shifts in the genome of the population elite, known as punctuated equilibria (PEs). Eldredge and Gould (1972) describe PEs as sudden shifts in the phenotype of a population that results in speciation happening quickly as opposed to gradually. While this theory was applied to observations of phenotypes within paleological records, Figure 4.5 shows PEs happen on a genomic level in the simulated evolution near generations 270 and 610. The first PE happens shortly after the first jump in fitness of the best individual. The second PE happens after a relatively small change (~1%) increase in the best fitness. Finally, the majority of fitness improvements do not result in a large shift of the genomes in the population.

The analysis was repeated for poorly performing evolutions with the elevated mutation rate. Figure 4.6 reveals what happens within the genome of the best performing 10% during

an unsuccessful evolution. Due to the elevated mutation rate, more genes are generated. However, the lack of any PEs show that none of the genes are ever eliminated within the elite population. Thus, there is a correlation between PE and evolutionary progress.

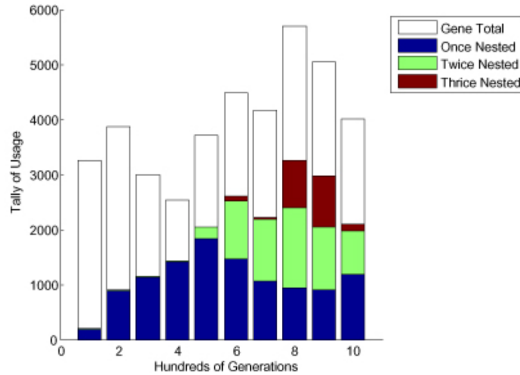


Figure 4.7: Structure of genes used by the top 10% of each generation during a successful evolution.
(Once nested is at the bottom).

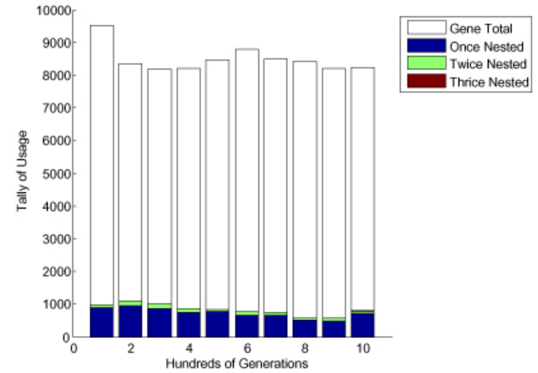


Figure 4.8: Structure of genes used by the top 10% of each generation during an unsuccessful evolution.
(Once nested is at the bottom).

Figure 4.7 shows how the rules become more complex throughout evolution. The height of the overall bar diagram shows how many different genes were used throughout evolution, grouped for every hundred generations. The number of nestings indicate the number of additional conditions that must test true in order for an action to be executed. Thus, a thrice-nested rule must have four IF statements prove true for its action to execute. Over time, a higher percentage of the rules used have additional nestings. Furthermore, the number of genes used by the best individuals changes as well. As Adami et al. (2000) argues, a more complex gene contains more information about its environment, and genes that require more specified conditions to execute an action contain more information about the required state of the network. The results of Figure 4.7 are contrasted with the unsuccessful results shown in Figure 4.8. The illustration reconfirms that many more genes were generated during the unsuccessful evolution. However, there is little variation throughout evolution. Furthermore, the rules used do not become more complex.

Finally, statistics looking at the structure of the rules are examined. The actions of every codon within each gene that is executed are tallied for each run. It is important to note that the sum of these tallies will be higher than the total number of genes used because nested genes contain multiple codons, and thus, multiple actions. Furthermore, while the

actual numbers are given, it is the relative ratios that remain consistent among similar runs. Table 4.4 reveals that making a connection was the most common action. However, the second most common action was the end turn action, which prevents the growing network from performing tasks. This suggests that the control of growth is nearly as important as growth itself. In other words, *evolving rules prohibiting actions may be as important as involving rules that promote actions.*

Table 4.4: Actions in executed genes

	Make Connection	Make Node	Do Nothing	End Turn
Successful Run	4297	1628	1566	3981
Unsuccessful Run	12750	4346	612	8054

4.3 Variation of Nucleotides within the NEURAE Codon

It was argued earlier that having more complex genomes meant using more information from the environment. Furthermore, the previous section showed that as individuals became more fit, the rules often required a growing ANN to meet more conditions before an action is executed. However, this just means the use of more environmental information is correlated to more successful evolutions, but not necessarily the cause of them. Thus, the following experiment was devised to disable the genome from using any information from the environment for embryogenesis. Every test range (4th) nucleotide was set to write a large number (250) into the C^{++} program. This test range is large enough to encompass all possible ANN states and results in every condition test to be true. With this configuration, the programs in NEURAE run similar to the programs in Gruau's CE, where the order in which actions are executed are completely determined by the sequence of actions in the program.

This change seems to completely break NEURAE, as none of the evolutionary runs produced an XOR gate. While it can be argued that implementing more action options or not resetting the program for each pairing permutation could have produced an XOR gate, it's clear that NEURAE benefits in having information from the environment to correctly apply embryogenesis.

The second experiment tested the effect of growth controls. For this experiment, *End Turn* action (5th) nucleotides were replaced with *Do Nothing* nucleotides. This results in a set of rules in which actions cannot be actively halted.

Even though this experiment used the same mutation rates as in case 14, the removal of *End Turn* nucleotides results in the probability of an XOR gate being created dropping to 88.8%. However, if a desirable circuit was created, the runs were able to optimize it as effectively as the evolutions in case 14, with an Ω fitness average at 102.6 and Ω fitness standard deviation of 10.3. However, one curious side effect is that evolutions without *End Turn* nucleotides took more than twice the computational time. While computation time was not an explicit evaluation parameter for evolution, clearly using more time to get worse results is undesirable. Thus, its clear that including *End Turn* action codons is beneficial for the practical application of NEURAE.

Chapter 5

Derivation of Simulation Environment

5.1 Nomenclature

A = Amplitude of path sinusoid

$\vec{a}$ = Shortest vector from robot center to obstacle wall

a_x = x-coordinate of $\vec{a}$

a_y = y-coordinate of $\vec{a}$

$\vec{b}$ = Vector coincident with obstacle wall

b_x = x-coordinate of $\vec{b}$

b_y = y-coordinate of $\vec{b}$

C = Slope of path sinusoid

c_1 = Chord length of left wheel movement approximation

c_2 = Chord length of right wheel movement approximation

d = Diameter of robot

f = Frequency for path sinusoid

$g(\cdot)$ = Function which is centerline of path

h = Distance from left wheel to point of rigid body rotation

$\vec{l}$ = Unit vector coincident with LIDAR sensor

l_x = x-coordinate of $\vec{l}$

l_y = y-coordinate of $\vec{l}$

$\vec{l}_\perp$ = Unit vector perpendicular to LIDAR sensor.

m = Slope of line connecting photovoltaic sensor and closest point to path

$\vec{p}_1$ = Global position vector to first obstacle vertex

p_{1x} = x-coordinate of $\vec{p}_1$

p_{1y} = y-coordinate of $\vec{p}_1$

$\vec{p}_2$ = Global position vector to second obstacle vertex

p_{2x} = x-coordinate of $\vec{p}_2$

p_{2y} = y-coordinate of $\vec{p}_2$

$\vec{q}_1$ = Vector from robot center to first obstacle vertex

q_{1x} = x-coordinate of $\vec{q}_1$

q_{1y} = y-coordinate of $\vec{q}_1$

$\vec{q}_2$ = Vector from robot center to first obstacle vertex

q_{2x} = x-coordinate of $\vec{q}_2$

q_{2y} = y-coordinate of $\vec{q}_2$

r = Radius of robot

s_1 = Arc traversed by left wheel

s_2 = Arc traversed by right wheel

t = Time

$\vec{v}_1$ = Left wheel movement approximation vector

$\vec{v}_2$ = Right wheel movement approximation vector

$\vec{v}_{cg}$ = Robot center movement approximation vector

w = Width of the path

x_1 = x-coordinate of photovoltaic sensor

x_2 = x-coordinate of path closest to photovoltaic sensor

$\vec{x}_i$ = Vector to initial robot global position

$\vec{x}_f$ = Vector to final robot global position

$\vec{x}_t$ = Vector to test robot global position

x_{tx} = x-coordinate of $\vec{x}_t$

x_{ty} = y-coordinate of $\vec{x}_t$

y_1 = y-coordinate of photovoltaic sensor

y_2 = y-coordinate of path closest to photovoltaic sensor

α = Angle of rigid body rotation

β = Angle between $\vec{v}_2$ and vector pointing from the left wheel to the right wheel

η = Distance from laser origin to wall

γ = Angle perpendicular to initial robot orientation

θ = Angle laser makes with global x-axis.

κ = Scalar used to find an arbitrary location along obstacle wall

ν_1 = Left wheel translational speed

ν_2 = Right wheel translational speed

σ = Angle between $\vec{v_1}$ and global x-axis

τ = Discrete time between simulation steps

ϕ_i = Initial robot orientation

ϕ_f = Final robot orientation

ϕ_t = Test robot orientation

5.2 Two-Wheeled Robot Movement

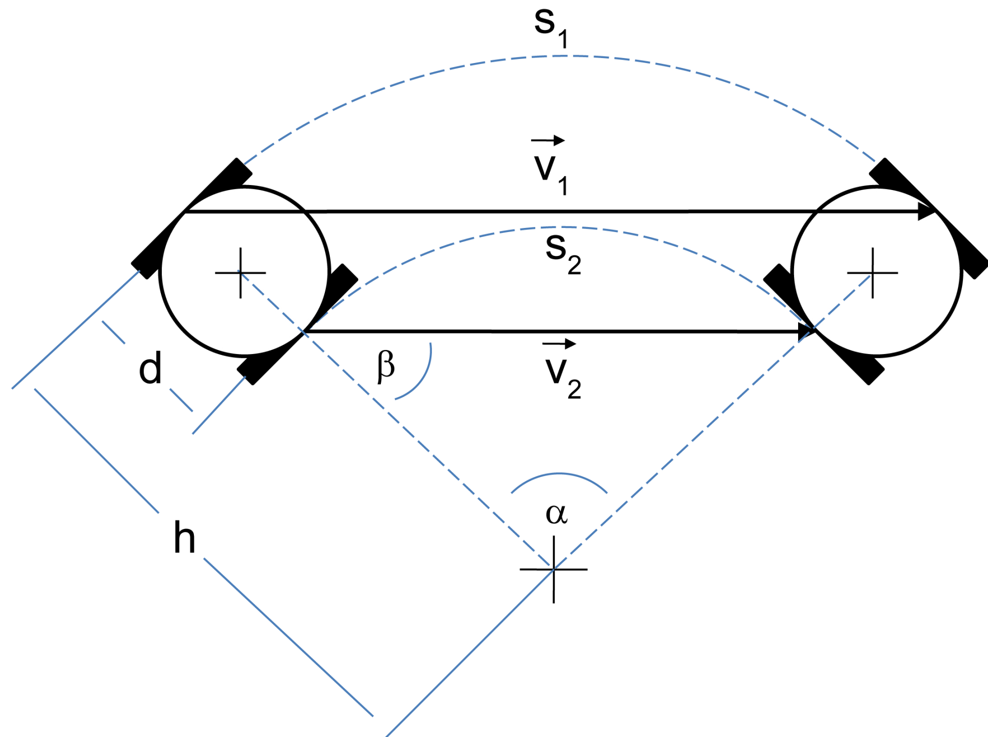


Figure 5.1: Diagram of variables for two-wheeled motion derivation

While the following robots may have varying sensor setups, they all have the same basic movement model. All robots herein have the two-wheeled model shown in Figure 5.1. The assumption that the wheels never slip enables robot movement to be modeled as rotation of a rigid body rotating about some point in the 2-D plane.

As the left wheel travels, it moves along the arc,

$$s_1 = h\alpha. \tag{5.1}$$

Figure 5.1 illustrates $\vec{v}_1$ and $\vec{v}_2$ are respective chords for the arcs s_1 and s_2. Using the

Law of Cosines, the magnitude of the chord, c_1, squared is

$$c_1^2 = 2h^2 - 2h^2\cos(\alpha) = 2h^2(1 - \cos(\alpha)). \tag{5.2}$$

However the Taylor series expansion of $cos(\alpha)$ about $\alpha = 0$ is

$$\cos(\alpha)|_{\alpha=0} = 1 - \frac{\alpha^2}{2!} + \frac{\alpha^4}{4!} - H.O.T. \tag{5.3}$$

Plugging the truncation of the Taylor series expansion into Equation 5.2 gives

$$c_1^2 \approx 2h^2 \left(1 - (1 - \frac{\alpha^2}{2} + \frac{\alpha^4}{24})\right), \tag{5.4}$$

$$c_1 \approx h\alpha - \frac{\alpha^2}{2\sqrt{3}}. \tag{5.5}$$

The error between the arc length in Equation 5.1 and the chord length in Equation 5.5 has a maximum error of $\frac{\alpha^2}{2\sqrt{3}}$. If α is small, using the chord to approximate wheel movement in Equation 5.1 is acceptable. Thus, the simulation time steps are kept small and the wheels are assumed to move along the chords instead of the arcs.

Equation 5.2 can be rewritten to make

$$\cos(\alpha) = \frac{2h^2 - c_1^2}{2h^2} = 1 - \frac{c_1^2}{2h^2}. \tag{5.6}$$

Using similar triangles,

$$\frac{c_1}{h} = \frac{c_2}{h - d}, \tag{5.7}$$

$$h = \frac{c_1 d}{c_1 - c_2}. \tag{5.8}$$

Substituting Equation 5.8 into Equation 5.6 gives

$$\cos(\alpha) = 1 - \frac{(c_1 - c_2)^2}{2d^2}. \tag{5.9}$$

Now, Equation 5.9 can be solved for α in terms of known qualities,

$$\alpha = \cos^{-1}\left(1 - \frac{(c_1 - c_2)^2}{2d^2}\right). \tag{5.10}$$

It is necessary to verify that the assumption made in Equation 5.5 is accurate enough. Having the wheels rotate in opposite directions and at equal magnitudes will result in the the robot spinning in place and have the largest possible estimation error of the orientation. If the wheels are assumed to move along the arc, the orientation will change according to Equation 5.11,

$$\alpha(t) = \frac{s_1 t}{h}. \tag{5.11}$$

The exact movement represented by Equation 5.11 and the approximate movement represented by Equation 5.10 are compared. For the verification of rotational accuracy the following values were given: $\nu_1 = 1$ m/s, $\nu_2 = -1$ m/s, $d = 1$ m, $\tau = 0.02$ s. This leads to the following values of $s_1 = c_1 = \tau\nu_1 = 0.02$ m, $c_2 = \tau\nu_2 = -0.02$ m, and $h = \frac{d}{2} = 0.5$ m during each simulation step. The exact and approximated results are shown in Figure 5.2 to be nearly identical with a maximum error of 0.003 rad.

Once it is known how much the robot has changed its orientation during the time step, it is necessary to determine the displacement of its center. Due to the fact that the angles of the isosceles triangle in Figure 5.1 must add up to π, $\beta = \frac{\pi-\alpha}{2}$. However, there is a need to account for clockwise or counterclockwise rotations for determining the global orientation of the two displacement vectors, $\vec{v_1}$ and $\vec{v_2}$.

$$\phi_t = \begin{cases} \gamma + \beta & \text{if } c_1 > c_2, \\ \gamma - \beta + \pi & \text{if } c_1 \leq c_2. \end{cases} \tag{5.12}$$

By knowing the orientation and magnitude of the displacement of each wheel, $\vec{v_1}$ and $\vec{v_2}$ can be found by Equations 5.13 and 5.14.

$$\vec{v_1} = \begin{bmatrix} \cos(\phi_t) \\ \sin(\phi_t) \end{bmatrix} c_1, \tag{5.13}$$

$$\vec{v_2} = \begin{bmatrix} \cos(\phi_t) \\ \sin(\phi_t) \end{bmatrix} c_2. \tag{5.14}$$

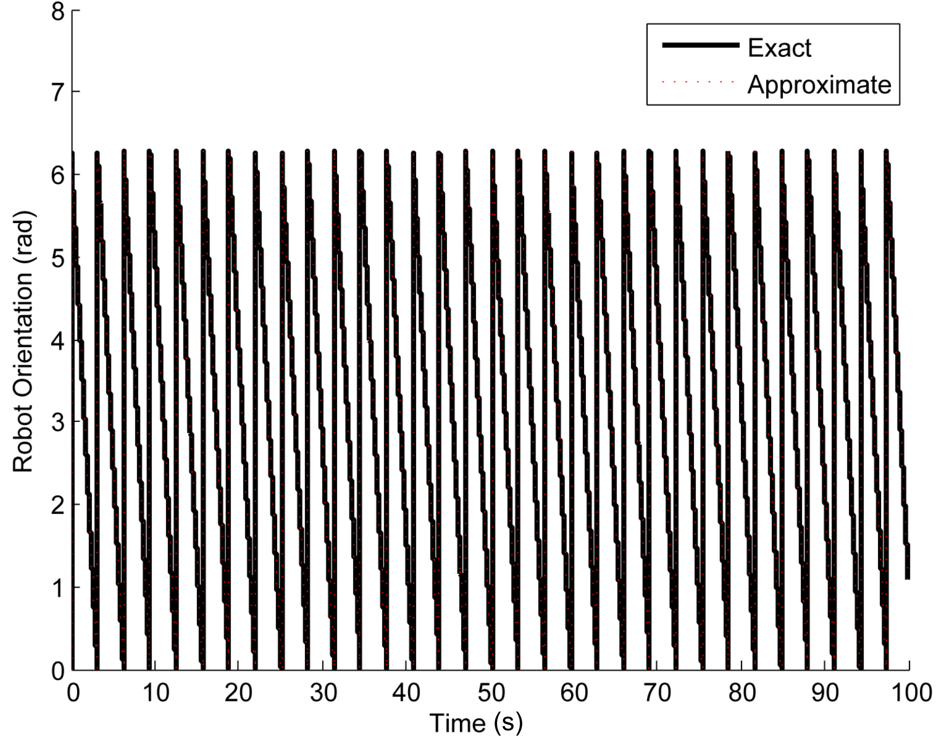


Figure 5.2: Verification of rotational accuracy with and without approximation.

The displacement of the center of the robot is the average of the displacement of the two wheels, so $\vec{v}_{cg} = \frac{\vec{v}_1+\vec{v}_2}{2}$. Finally, the overall change of the robot position is shown in Equations 5.15 and 5.16.

$$\phi_t = \phi_i + \alpha, \tag{5.15}$$

$$\vec{x}_t = \vec{x}_i + \vec{v}_{cg}. \tag{5.16}$$

To verify that the approximations are accurate, two more simulations were run: one with a stationary wheel, and another with the wheels at two different, but constant, speeds. The exact movement results from Equations 5.17 - 5.22 are compared to the approximation results in Equations 5.15 and 5.16.

$$\alpha(t) = \frac{c_1 t}{2r}. \tag{5.17}$$

$$x(t) = r\,\sin\left(\frac{c_1 t}{2r}\right). \tag{5.18}$$

$$y(t) = r\left(1 - \cos\left(\frac{c_1 t}{2r}\right)\right). \tag{5.19}$$

$$\alpha(t) = \frac{c_1 t}{4r}. \tag{5.20}$$

$$x(t) = 3r\,\sin\left(\frac{c_1 t}{4r}\right). \tag{5.21}$$

$$y(t) = 3r\left(1 - \cos\left(\frac{c_1 t}{4r}\right)\right). \tag{5.22}$$

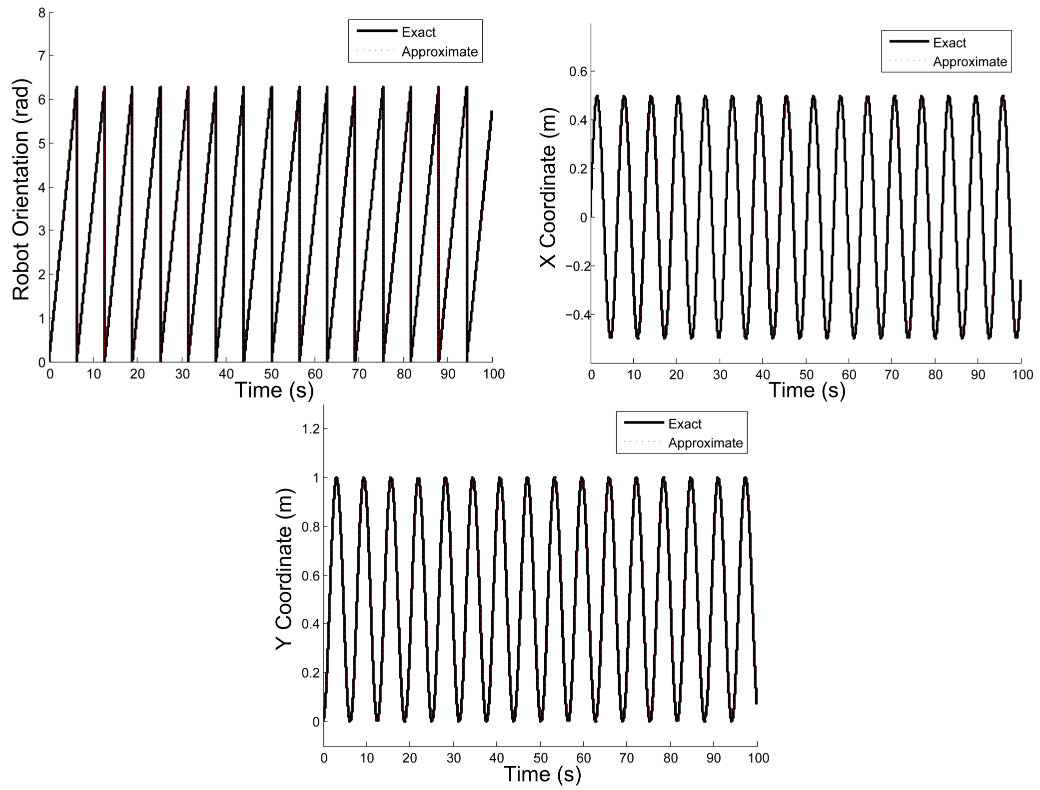


Figure 5.3: Verification of rotational and translational accuracy used the respective left and right wheel speeds of $\nu_1 = 0$ m/s and $\nu_2 = 1$ m/s. The maximum orientation, x-position, and y-position errors are 0.017 rad, 0.0079 m, and 0.0083 m, respectively.

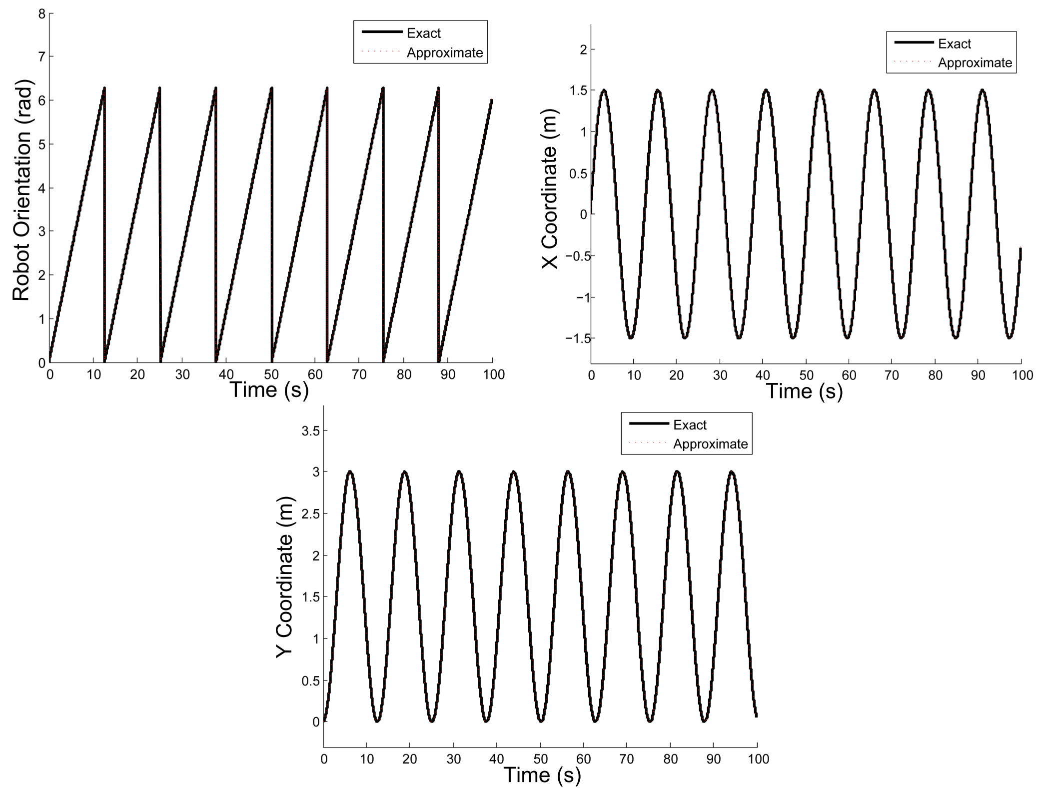


Figure 5.4: Verification of rotational and translational accuracy used the respective left and right wheel speeds of $\nu_1 = 0.5$ m/s and $\nu_2 = 1$ m/s. The maximum orientation, x-position, and y-position errors are 0.037 rad, 0.055 m, and 0.056 m, respectively.

5.3 Collision Detection

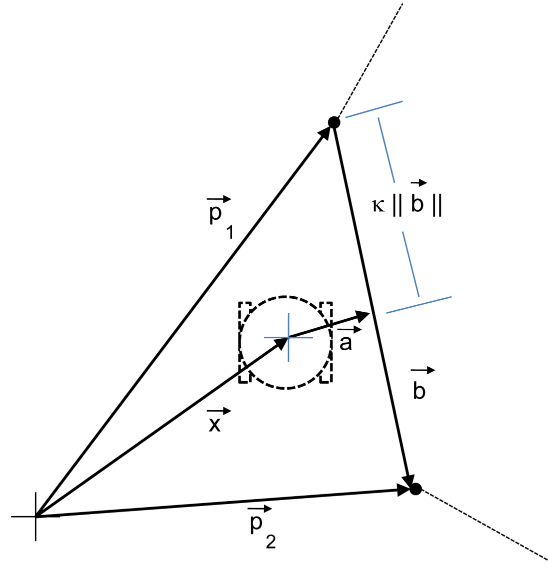


Figure 5.5: Diagram of variables for obstacle collision check.

The next thing to account for is interactions between the robot and obstacles. All obstacles in the simulation world are polygons. Before the robot moves to the new position determined by Equation 5.16, there is first a check to make sure it does not pass the boundaries of an obstacle, i.e., collide with an obstacle. In Figure 5.5, the point where $\vec{a}$ intersects $\vec{b}$ is shown in Equations 5.23 and 5.24.

$$\vec{x}_t + \vec{a} = \vec{p}_1 + \kappa\vec{b}, \tag{5.23}$$

$$\vec{a} = \vec{p}_1 + \kappa\vec{b} - \vec{x}_t. \tag{5.24}$$

However, $\vec{a} \perp \vec{b}$, so there dot product is zero, as shown in Equation 5.25.

$$\vec{a} \cdot \vec{b} = (\vec{p}_1 + \kappa\vec{b} - \vec{x}_t) \cdot \vec{b} = 0. \tag{5.25}$$

Solving Equation 5.25 for κ yields the result shown in Equation 5.26.

$$\kappa = \frac{(b_x x_{tx} + b_y x_{ty}) - (b_x p_{1x} + b_y p_{1y})}{b_x^2 + b_y^2}. \tag{5.26}$$

If $0 < \kappa < 1$, then $\vec{a}$ coincides with the line $\vec{b}$ within the line segment of the wall. Equation 5.27 is used to check if the shortest distance from the center of the robots to the wall is greater than the radius of the robot.

$$\|a\| = \|\vec{p}_1 + \kappa\vec{b} - \vec{x}\| > r. \tag{5.27}$$

If κ is not within the range (0,1), Equation 5.28 is used to ensure the robot clears the vertices of the obstacle.

$$\|\vec{p}_1 - \vec{x}\| > r \cap \|\vec{p}_2 - \vec{x}\| > r. \tag{5.28}$$

If the inequalities in either Equation 5.27 or Equation 5.28 are not satisfied, then the robot will cross a boundary within the next simulation step. To prohibit this, the robot keeps the same position it previously had. However, the robot is free to rotate as it normally would. For verification, the robot is placed in a box and moves and rotates in increments.

5.4 Sensor and World Interaction

After the robot moves to the new orientation, the sensors are updated. For the line following robot, photovoltaic sensors are configured to be on if the sensor is positioned above the black line, and off otherwise. The centerline of the line to be followed is a sinusoid with a slope and is governed by Equations 5.29 and 5.30.

$$g(x) = A\cos(fx) - A - Cx, \tag{5.29}$$

$$y_2 = A\cos(fx_2) - A - Cx_2. \tag{5.30}$$

The line connecting the photovoltaic sensor and the point on the centerline closest to it, as shown in Figure 5.8, is represented by Equation 5.31.

$$y_2 = y_1 + m(x_2 - x_1). \tag{5.31}$$

However, the slope of the centerline of the path at x_2 can be found by Equation 5.32.

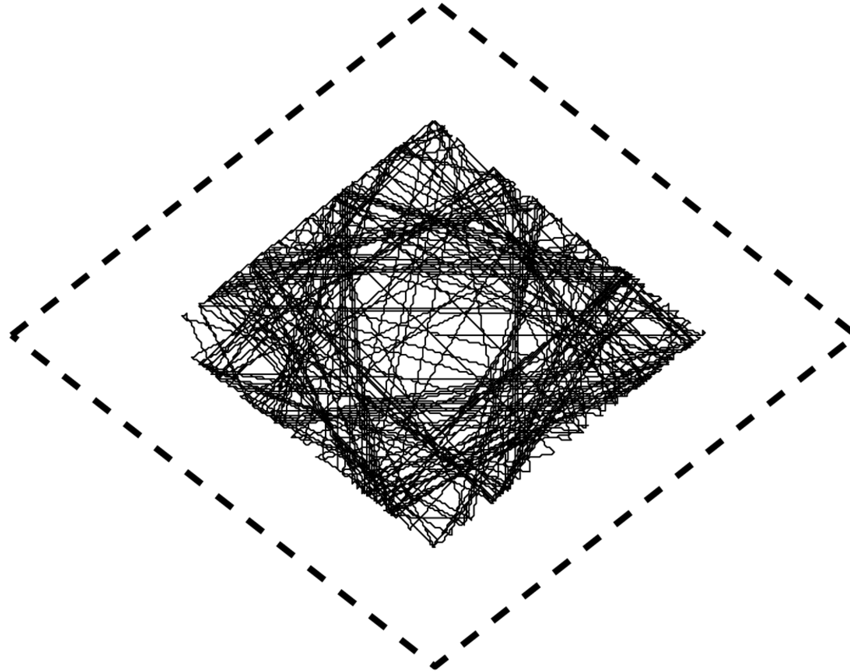


Figure 5.6: Collision detection was verified by placing the robot within a small obstacle and having it move around. As shown above, the center of the robot is never closer than 0.5 m (the radius) to the obstacle wall.

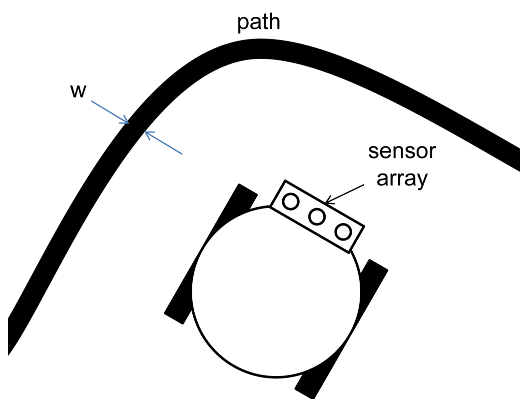


Figure 5.7: Model of robot sensor configuration for path following simulations.

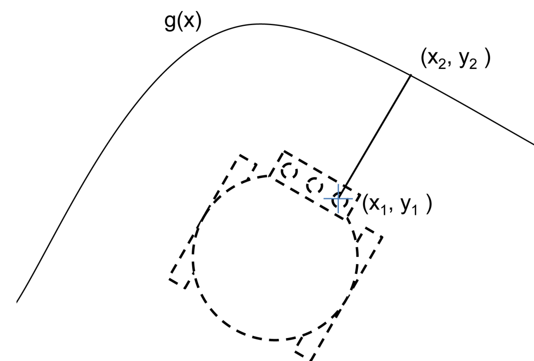


Figure 5.8: Diagram of variables used for path detection calculations.

$$g'(x_2) = C - Af\text{sin}(fx). \tag{5.32}$$

This slope, however, is perpendicular to the connecting line shown in Figure 5.8. Thus, the slope, m, of the connecting line must be the negative inverse of the slope of the centerline as shown in Equation 5.33.

$$m = \frac{-1}{(g'(x_2)} = \frac{1}{Af\text{sin}(fx) - C}. \tag{5.33}$$

Substituting Equations 5.30 and 5.33 into Equation 5.31 yields Equation 5.34.

$$A\text{cos}(fx_2) - A + Cx_2 = y_1 + \frac{x_1 - x_2}{C - Af\text{sin}(fx_2)}. \tag{5.34}$$

However, Equation 5.34 will have problems when the slope of the sinusoid is 0. Thus it is converted to the following equation:

$$x_1 - x_2 + (C - Af\text{sin}(fx_2))(y_1 + A - A\text{cos}(fx_2) - Cx_2) = 0. \tag{5.35}$$

Equation 5.35 is used to solve for x_2 within the range of $x_1 - \frac{w}{2}$ and $x_1 + \frac{w}{2}$ numerically via the secant method. If a zero for x_2 is not within these bounds, the sensor must be further than $\frac{w}{2}$ away from the centerline and off the path. However, the search region must be broken in two sections to account for multiple roots. Thus, for regions $[x_1 - \frac{w}{2}, x_1]$ and $[x_1, x_1 + \frac{w}{2}]$ are searched separately. If a zero is found within these bounds, the secant method finds the root quickly. Once x_2 is calculated, y_2 can be found with Equation 5.30. If $\left((x_1 - x_2)^2 + (y_1 - y_2)^2\right) \leq \frac{w^2}{4}$ then the sensor is over the line and is consequently activated. Otherwise, the sensor is off.

The fully 2-D robot navigates by using simulated LIDAR sensors which can detect the distance to an obstacle in front of it. $\vec{l}$ is a unit vector collinear with the LIDAR. $\vec{l}_\perp$ is used to check if the LIDAR unit intersects $\vec{b}$ within the wall segment with the following inequality,

$$(\vec{l}_\perp \cdot \vec{q}_1) \cdot (\vec{l}_\perp \cdot \vec{q}_2) \leq 0. \tag{5.36}$$

If Inequality 5.36 is true, it is necessary to first check if the laser is collinear with the

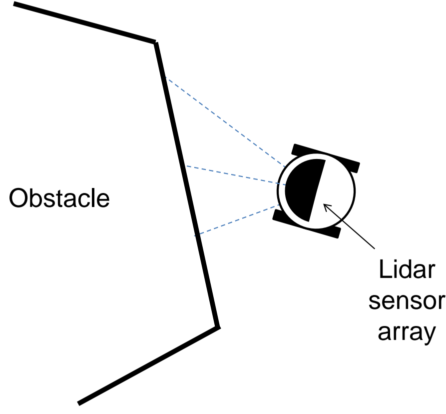


Figure 5.9: Model of robot sensor configuration for full 2-D navigation.

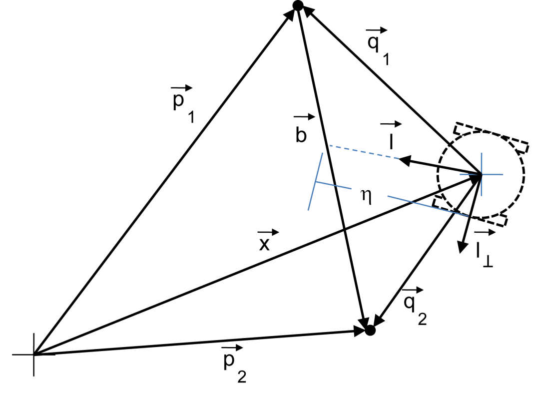


Figure 5.10: Diagram of variables used for full 2-D navigation.

wall by evaluating Equation 5.37.

$$(\vec{l}_\perp \cdot \vec{q}_1) \cdot (\vec{l}_\perp \cdot \vec{q}_2) = 0. \tag{5.37}$$

If Equation 5.37 is true, it is necessary to check $(\vec{l}_\perp \cdot \vec{q}_1)$ and $(\vec{l}_\perp \cdot \vec{q}_2)$ separately. If both equal 0, $\eta = \min(\|\vec{q}_1\|, \|\vec{q}_2\|)$. Otherwise, $\eta = \|\vec{q}_i\|$ for which $(\vec{l}_\perp \cdot \vec{q}_i) = 0$.

However, if the product of dot products in Equation 5.36 is less than 0, then $\vec{l}_\perp$ intersects $\vec{b}$. To find the distance Equation 5.38 is used.

$$\vec{q}_1 + \kappa\vec{b} = \eta\vec{l}, \tag{5.38}$$

which becomes the linear equation shown in Equation 5.39.

$$\begin{bmatrix} l_x b_x \\ l_y b_y \end{bmatrix} \begin{bmatrix} \eta \\ -\kappa \end{bmatrix} = \begin{bmatrix} q_{1x} \\ q_{1y} \end{bmatrix}. \tag{5.39}$$

Then, the distance η becomes

$$\eta = \frac{q_{1x} b_y - q_{1y} b_x}{l_x b_y - l_y b_x}. \tag{5.40}$$

If $\eta \geq 0$, the LIDAR sensor will hit the wall and return a distance η. If $\eta < 0$, the wall is behind the sensor so there is no reading. This process is repeated for each wall, and the smallest distance is the value that the sensor returns. A value less than the diameter of the robot will cause the corresponding ANN input to active. Figure 5.4 shows the simulated

robot with laser/obstacle interaction.

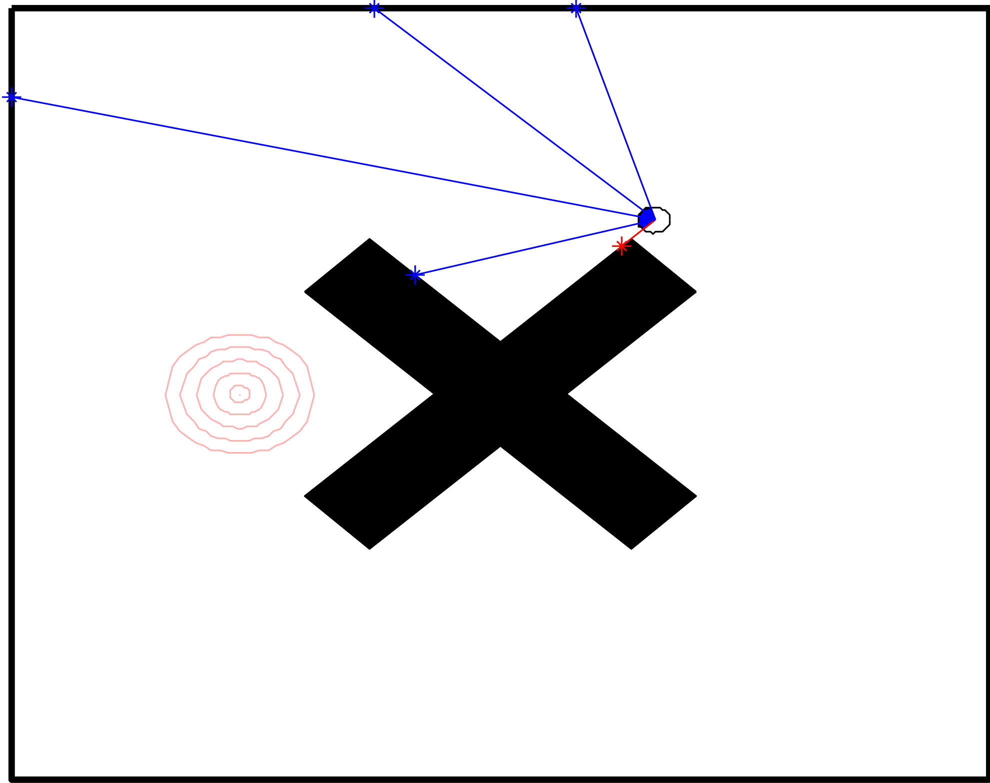


Figure 5.11: Graphical verification of accurate laser/object interaction. A blue line indicates the corresponding ANN input is inactivate while red line indicates the corresponding ANN input has been activated. The concentric circles are indicative of the desired goal

Chapter 6

Robotic-Controller Evolution

6.1 Overview

This chapter will describe the evolution of digital controllers for the simulated robots detailed in Chapter 5. All evolutionary runs have a population of 200 individuals with a starting genome length of 150 nucleotides. The mutation rates are set in accordance with the best performing case runs found in Chapter 4, $\mu = 0.80, \mu_P = 0.50, \mu_C = 0.075, \mu_R = 0.075, \mu_D = 0.15$, and $\mu_T = 0.20$. The design problems to be solved are creating a controller for a line-following robot, creating an obstacle avoiding robot, and creating controllers for a swarm of goal finding robots. As a result, the exponential fitness has the form in Equation 6.1 to further magnify slight improvements in the later tiers.

$$Fitness = \left\lfloor 2^{2(x-1)} \right\rfloor . \tag{6.1}$$

6.2 Line-Following Robot

6.2.1 Evaluation Parameters

Each ANN begins as three input neurons, one for each photovoltaic sensor. Table 6.1 shows the tiers used for the exponent in this simulation. Once again, an individual that passes the second tier has a fitness exponent of $x-1=1$. However, these individuals need to grow and connect two outputs instead of the one in the previous logic evolutions.

Once an individual grows and connects to two outputs, it gets to tier 3 and its line following ability is tested. The path to be followed is a line with a width w, and a centerline that satisfies Equation 5.29. The robot starts at the origin facing in the direction of the

Tier	Test	Change in Exponent
1	Are there enough output nodes?	% of desired output nodes
2	Are there a connections to each output node?	+ % of output nodes with connections
3	Simulate robot for 20 seconds	+ % of path followed correctly

Table 6.1: Tier for adjusting fitness exponent (x) in line following evaluation

positive x-axis. The constants are chosen to ensure the line intersects the origin with the center sensor over the line. Furthermore, the curvature of the line is always less than the turning radius of the robot, r. An individual is allotted 20 seconds of simulated time. At each time step, it is evaluated by Equation 6.2 where ϵ is the error between the robot's center and the centerline of the path. These values are summed and divided by the sum of Equation 6.2 if ϵ were 0 for all time steps. This fraction is then added to the exponent in Equation 6.1.

$$f(x) = 1 - \frac{1}{1 + e^{-4\frac{\epsilon - r}{w}}}. \tag{6.2}$$

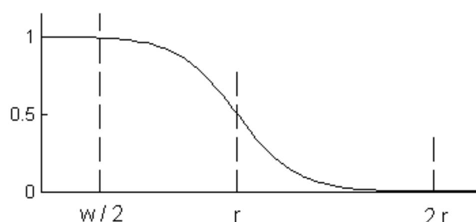


Figure 6.1: Preference function for position error in line following evaluation

6.2.2 Evolution Results

Figure 6.2 shows that the center of the robot traveled along the path and is a capable line follower. The ANN controller of the robot is shown in Figure 6.3. While there was no explicit penalty for building extra neurons, an ANN with a hidden layer could cause a lag in response time which would cause a larger error while following the path.

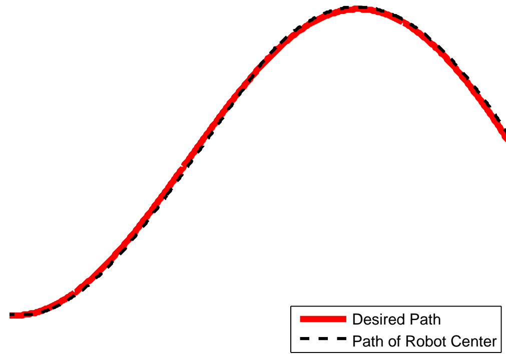


Figure 6.2: Robot path compared with desired path

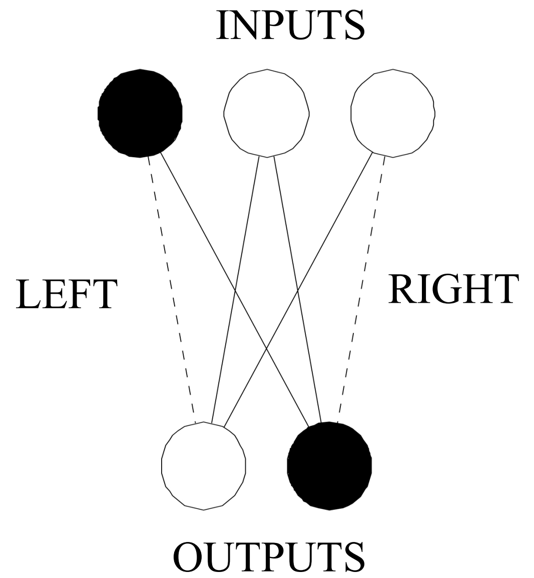


Figure 6.3: ANN controller for a line following robot

Left Sensor	Center Sensor	Right Sensor	Left Wheel	Right Wheel
0	0	0	1	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	0 or 1
1	0	0	0	1
1	1	0	0 or 1	1

Table 6.2: Dominant logic for line following robots

The resulting line-following logic is show in Table 6.2. While some of the entries are self-evident, such as turn right when only the right sensor is active, it was not clear what the right action should be when the line is not sensed. However, it was found through evolution that the best course of action if the line is not detected is to go forward. Given the limited sensing abilities of the robot, this is the best general-purpose line search the robot could perform.

```
for (Node α = 1:ANN.size ){
   for(Node β = 1:ANN.size ){
      if |Relαβ.inputs - 0| ≤ 1
         make.output(H,-0.22)
      }
      if |Relαβ.inputs - 1| ≤ 1{
         make.connection(-0.50)
      }
      if |Nodeα.ID3 - 1| ≤ 0{
         make.connection(0.24)
      }
      if |Relβα.threshold - (-0.26)| ≤ 0.14{
         end.turn()
      }
      if |Nodeα.outputs - 1| ≤ 0{
         make.connection(0.10)
      }
   }
}
```

Figure 6.4: Code used to make line following controller

6.3 Obstacle-Avoiding Robot

6.3.1 Evaluation Parameters

For this problem an ANN was evolved that could function as a controller for a robot that could find a goal within a closed 2-D room. The inputs to the ANN are the goal sensors and LIDAR sensors of the robot. The three goal sensors are configured to be on in accordance to Figure 6.5 with the center sensor having a 45° arc. These sensors give directional data, but not ranging information. Furthermore, the goal sensors are able to detect the goal regardless of distance or if there is an obstacle between the robot and the goal. As shown in Figure 5.4, there are five LIDAR sensors which are set at equivalent angles in the 120° arc in front of the robot. Goal sensor inputs for the ANN have an ID1 of A and LIDAR inputs have an ID1 of B. Because an ID1 of B is being used for an input, neurons grown during embryogenesis cannot have an ID1 of B.

Originally, the third tier was a simulation tier of having the robot find the goal in an enclosed room without internal obstacles. However, several individuals were able to find the

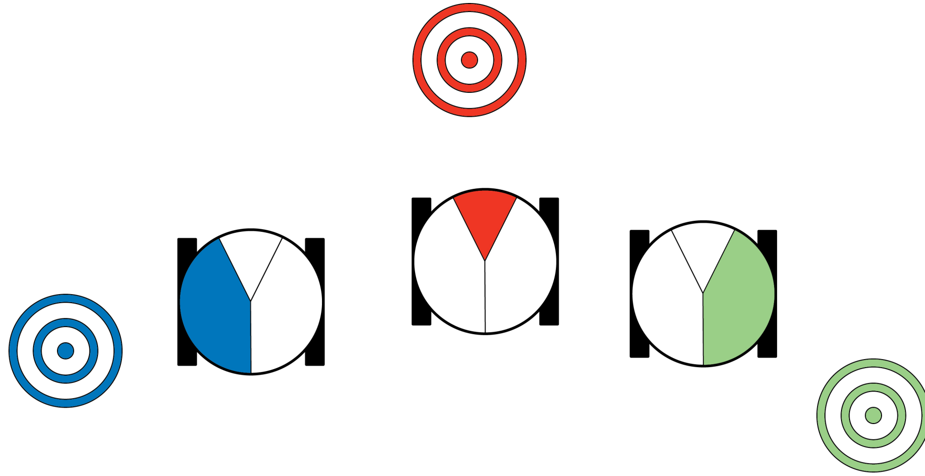


Figure 6.5: Goal sensor configuration for the obstacle avoiding robots. Detection is separated into left, center, and right.

Tier	Test	Change in Exponent
1	Are there enough output nodes?	% of desired output nodes
2	Are there a connections to each output node?	+ % of output nodes with connections
3	Logic test	+ % correct answers
4	Simulate with convex obstacle	+ summed distance to goal
5	Simulate with star obstacle	+ summed distance to goal

Table 6.3: Tiers for adjusting fitness exponent (x) in obstacle avoidance evaluation

obstacle without evolving the ability to turn both left and right! Usually, individuals would only be able to sense if the goal was to one side or another, and then use LIDAR detection of the border to make enough turns to compensate. Thus, the third tier was replaced with the logic test shown in Table 6.4. For these tests, it is assumed all the LIDAR inputs are off. This ensured the controller exhibited efficient logic in finding the goal in the absence of obstacles.

Once an ANN controller evinces the logic in Table 6.4, it moves to tier 4. Here, the robot is tested to see if it can find a goal with an obstacle between the starting point and goal. The environment shown in Figure 6.6 starts the robot in a random position and

Left Goal Sensor	Center Goal Sensor	Right Goal Sensor	Left Wheel	Right Wheel
0	0	1	1	0
0	1	0	1	1
1	0	0	0	1

Table 6.4: Logic test goal finding robots are required to pass before simulation. For this test, all LIDAR inputs are inactive

orientation in the upper-right corner, and its movement is simulated for 20 seconds. At each time step, the distance of the robot is evaluated by Equation 6.3, with ϵ being the distance between the robot and the goal. This distance is doubled if the robot is in contact with an obstacle, providing further evolutionary pressure for obstacle avoidance. As with line following evaluation, this value is summed and divided by the sum of Equation 6.3 if ϵ is equal to 0 for all time steps. This fraction is then added to the exponent in Equation 6.1. If at the end of the simulation, the robot is within one diameter of the goal, it is allowed to move on to tier 5.

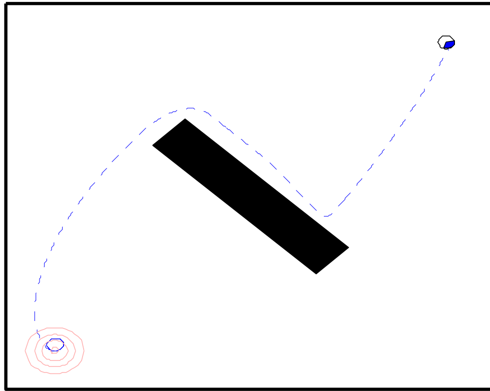


Figure 6.6: Environment for tier 4 evaluation

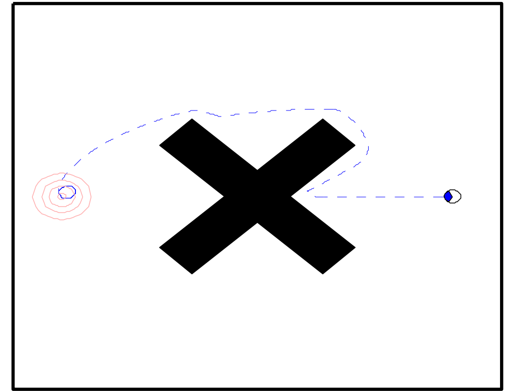


Figure 6.7: Environment for tier 5 evaluation

$$f(x) = \begin{cases} \frac{1}{1+e^{\frac{2\epsilon-10}{3}}} & \text{if there is a collision,} \\ \frac{1}{1+e^{\frac{\epsilon-10}{3}}} & \text{otherwise.} \end{cases} \tag{6.3}$$

Tier 5 is almost identical to tier 4, except now the environment includes the star obstacle shown in Figure 6.7. Usually, robots which performed well in tier 4 also performed well here, but this tier did help refine the controllers. Figure 6.7 shows the path taken by a successful individual.

6.3.2 Evolution Results

The synthesized controller shown in Figure 6.8 was able to navigate around convex and star obstacles. In the figure, the left three inputs correspond to the goal sensors, and the right five are the inputs from the LIDAR unit. The activation pattern shown in Figure 6.8 is the result of the goal being in front of the robot, but a wall is in close proximity of the two leftmost LIDAR sensors. The corresponding output is to have the left wheel on and the right wheel off, which will cause the robot to turn right, as desired.

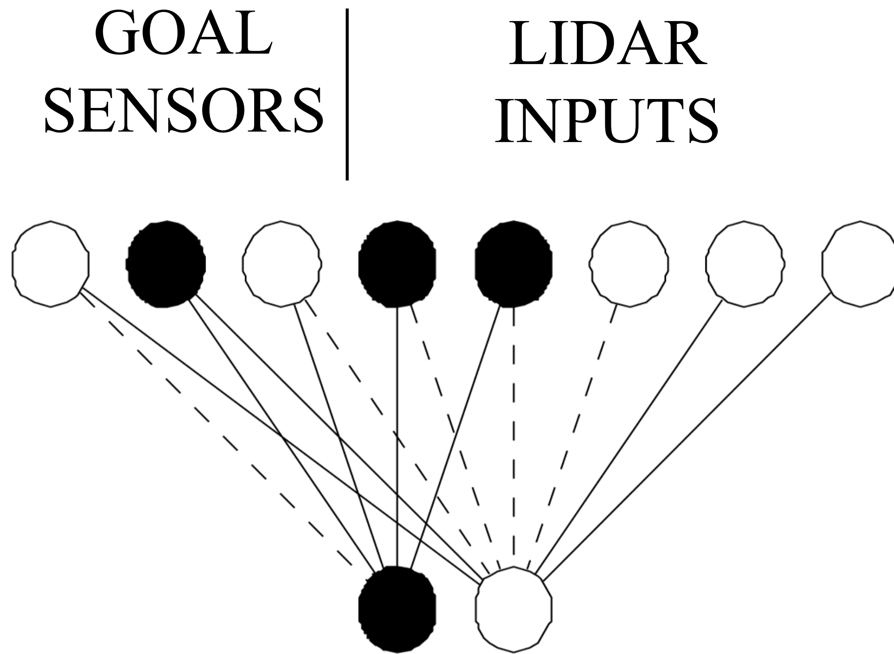


Figure 6.8: ANN controller for obstacle avoidance

In order to demonstrate the general capabilities of this controller, the individual was placed in two more simulation environments after evolution was completed. The first is an environment that is densely obstructed. As Figure 6.9 shows, the robot is still able to avoid the obstacles and reach the goal. The next task shown in Figure 6.10 could not be accomplished by the individual. In order to surmount this challenge, the robot had to be able to encounter the obstacle, then *move away* from the goal as it moved along the contour of the wall. Figure 6.10 shows that the robot was able to trace the wall and is able to follow the wall until the robot is facing the away from the goal. However, the goal sensors indicate the goal is on the right side of the robot, although nearly behind it. As a result, the

robot continues to turn right, looping toward the goal and away from the obstacle. Once it encounters the obstacle again, the cycle is restarted. While there may be a fine-tuned solution to create a feed-forward network for this problem, it is likely that this solution will be brittle. This problem may require a recursive neural network so that the controller can store and use gathered information about the environment.

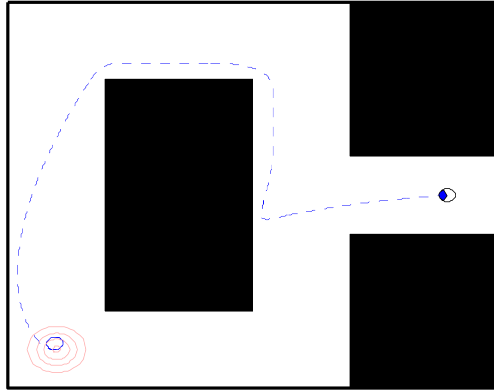


Figure 6.9: Obstacle avoidance robot in a densely obstructed environment

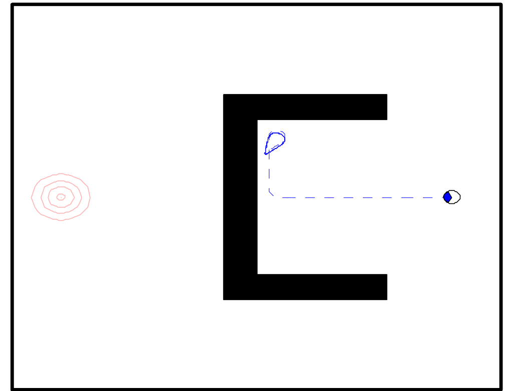


Figure 6.10: Obstacle avoidance robot in a environment with concave obstacle

6.4 Goal-Finding Swarm Robots

6.4.1 Evaluation Parameters

A network capable of controlling swarm behavior was the final goal. For this challenge, individuals had the same types of inputs as they did in the previous obstacle avoidance section, but the number of LIDAR input were increased to eight to provide higher fidelity. Furthermore, the goal sensors were reconfigured to not be able to detect the goal if an obstacle is blocking it, as shown in Figure 6.12. Thus, the individual had to evolve logic which enables it to search for the goal, then converge once found.

While, the robots here were unable to detect the goal if there is an obstacle between the two, Figure 6.13 shows that once a robot is able to see the goal, it sends out a signal at its own location, which other robots are able to detect. If the second robot is unable to see the goal, its goal sensors will indicate in what direction the first robot is. However, once a robot is able to detect the goal on its own, the goal sensors will ignore the signal from other goal-detecting robots, and give the direction of the goal.

```
for (Node α = 1:ANN.size ) {
   for(Node β = 1:ANN.size ) {
      if |Nodeβ.inputs - 0| ≤ 0 {
         make.output(H,0.26)
      }
      if |Nodeβ.inputs - 2| ≤ 2 {
         if |Relαβ.inputs - 0| ≤ 0 {
            make.connection(-0.10)
         }
         make.connection(0.44)
      }
      if |Relβα.inputs - 6| ≤ 0 {
         make.connection(-0.08)
      }
      if |Relαβ.ID3 - 0| ≤ 1 {
         make.connection(-0.96)
      }
   }
}
```

Figure 6.11: Code used to make obstacle avoidance controller

Table 6.5 shows the tiers used for evaluating swarm behavior. Rather than forcing a viable ANN to conform to an imposed logic table, the robot was simulated in the convex and star obstacle environments displayed in Figures 6.14 and 6.15.

The fifth tier is the first time swarming behavior is tested. For this challenge, one robot is placed near the goal. A second robot is placed on the other side of a star obstacle. The challenge for the individual is to create a controller where one robot can go toward a global signal without colliding with an obstacle. Figure 6.16 shows that NEURAE produced an individual capable of completing this task.

The sixth and final tier places the swarm in a larger room shown in Figure 6.17. For this tier, both robots are placed outside of detection range of the goal. Eventually, one of the robots finds the goal and the other is able to find it as well. Due to the increased number of tiers present in this evolution, most populations were still improving at the

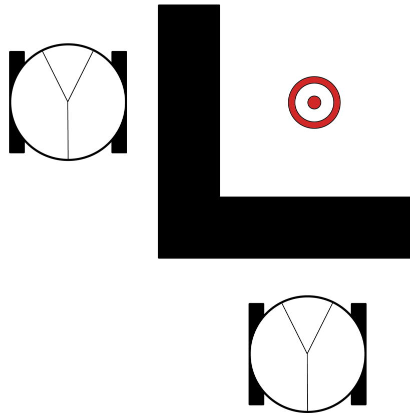


Figure 6.12: Goal sensor configuration for swarming robots where the goal is obstructed from the entire swarm.

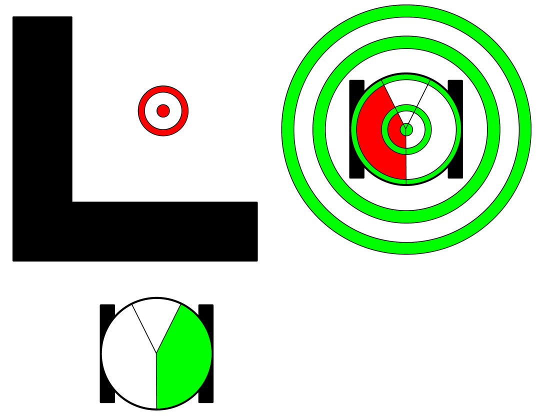


Figure 6.13: Goal sensor configuration for swarming robots where a member of the swarm can detect the goal.

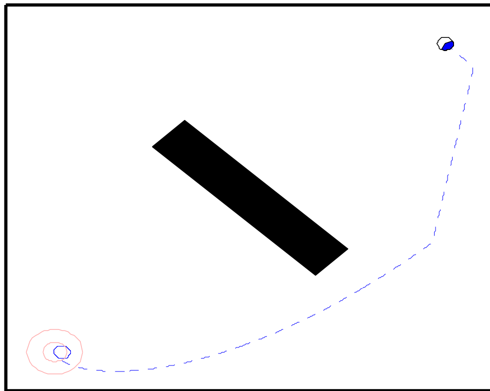


Figure 6.14: A single swarming robot in an environment with a convex obstacle

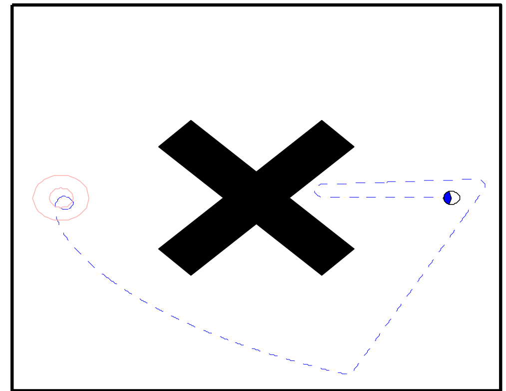


Figure 6.15: A single swarming robot in an environment with a star obstacle

Tier	Test	Change in Exponent
1	Are there enough output nodes?	% of desired output nodes
2	Are there a connections to each output node?	+ % of output nodes with connections
3	Simulate single robot with convex obstacle	+ summed distance to goal
4	Simulated single robot with star obstacle	+ % summed distance to goal
5	Simulate swarm with star obstacle	+ average summed distance to goal
6	Simulate swarm in large room	+ average summed distance to goal

Table 6.5: Tiers for adjusting fitness exponent (x)

1000th generation. As a result, evolutionary runs evolving swarming behavior were allowed to run for 1500 generations.

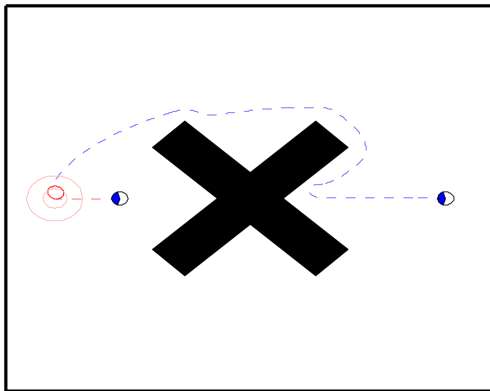


Figure 6.16: Two swarming robots in an environment with a star obstacle

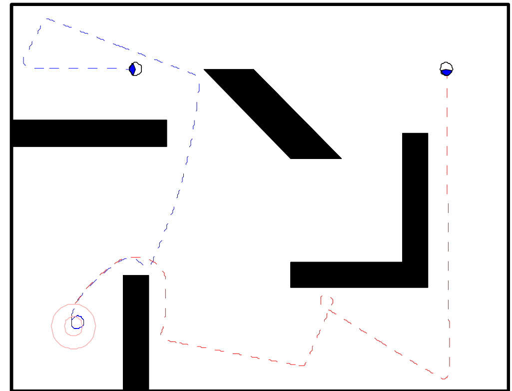


Figure 6.17: Two swarming robots in a large environment with various obstacles

6.4.2 Evolution Results

The individual that could control a swarm of robots as shown above, produced the controller shown in Figure 6.18. This particular ANN eventually evolved the logic to turn right whenever any of the LIDAR sensors detected a wall. As in the previous section, individuals

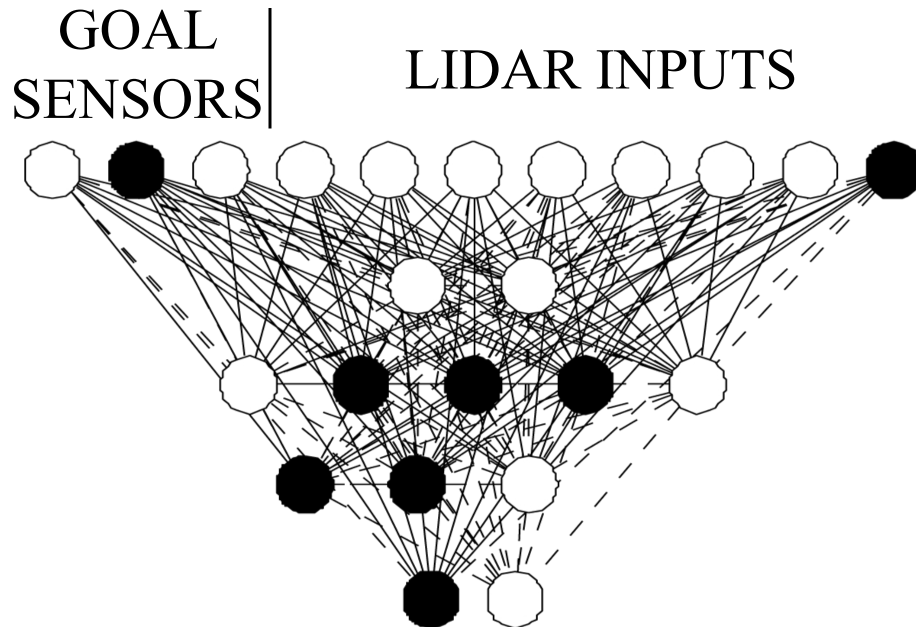


Figure 6.18: ANN controller for each swarming robot

that had a single robot capable of passing tier 3 seldom had trouble with tier 4. However, evolving the ability to avoid objects while tracking the signal of a robot in tier 5 was an equivalent challenge to the obstacle avoidance in section 6.3. Tier 6 proved to be an effective trial in which the swarm controllers were further refined.

Figure 6.19 shows the progression of the two robots at various times during the simulation of a successful individual in tier 6. The goal is in the lower left-corner, and the two robots begin in the upper-left and upper-right corners of the environment. For discussion, robot 1 begins in the upper left and robot 2 starts in the upper right. The robots roam about the room avoiding obstacles until, eventually, robot 1 is within direct line of sight of the goal, as shown at time = 31.00 s. The causes robot 1 to emit a signal, shown in Figure 6.19 by the concentric circles, that allows the goal sensors of robot 2 to detect the position of robot 1. Robot 2 begins to move toward robot 1, but the L-shaped obstacle prevents it from taking a direct path. Furthermore, at time = 45.00 s, robot 1 loses sight of the goal and both robots reenter their goal searching behavior. Nevertheless, robot 1 quickly reacquires the goal by time = 50.00 s, and moves toward it. Robot 2 once again moves toward robot 1, and begins to maneuver around the vertical obstacle. At time 75.00 s, robot 2 can also detect the goal and by time 80.00 s, both robots circle around the goal,

while avoiding contact with each other.

Once again, the evolved individuals were verified by being presented situations in which they were not explicitly evolved. The first is a revisit to the single robot seeking the goal with a concave obstacle. This time, however, the goal sensors do not cause the robot to loop within the obstacle because the robot is not within line of sight of the goal. As a result, a single robot is able to navigate around the environment to find the goal, as shown in Figure 6.20. Next, a swarm of three robots was placed in the environment shown in Figure 6.21. The entire swarm is once again able to converge at the goal. However, the robots are not able to avoid each other in such close proximity, and end up colliding.

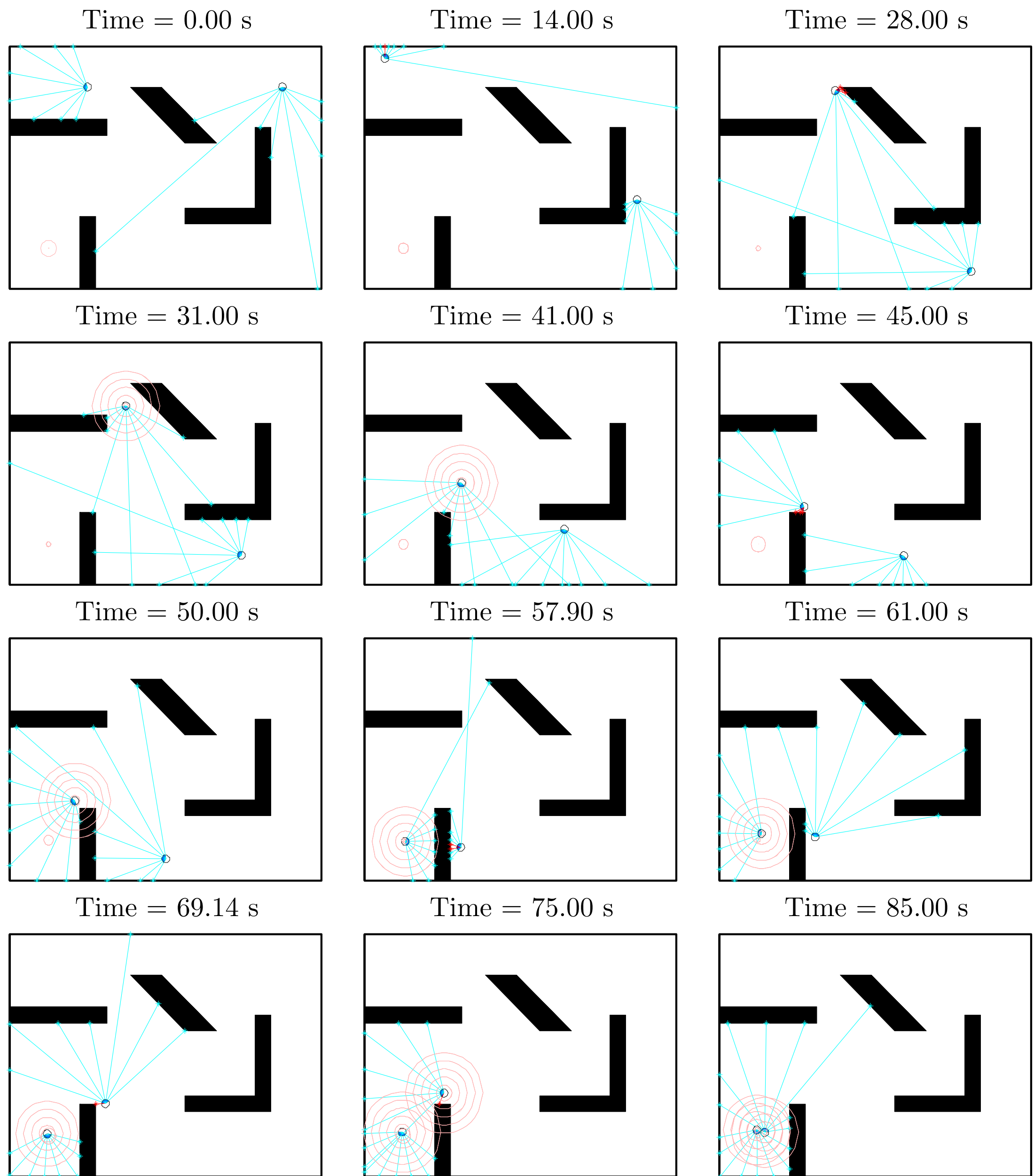


Figure 6.19: Steps showing the movement of an evolved swarm

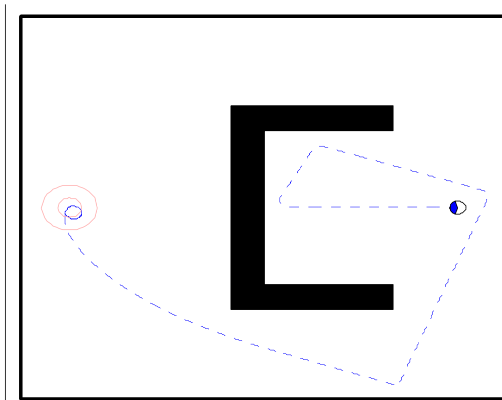


Figure 6.20: A single swarming robot in an environment with concave obstacle

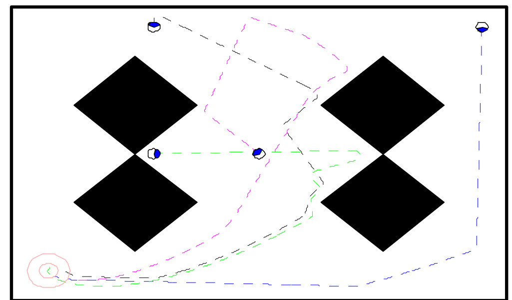


Figure 6.21: Three swarming robots in a large environment with various obstacles

Chapter 7

Conclusion

This dissertation has presented NEURAE, a genetic algorithm capable of generating artificial neural networks via the application of interchangeable rules. Furthermore, these networks have shown to be modular, scalable, and suitable for robotic control. The *If-CONDITION-Then-ACTION* structure of programs produced by NEURAE allows rules to be easily rearranged and create unanticipated, yet desirable results. In fact, the development of complex rules from simple building blocks may be a key element to the modularity expressed in the phenotypes. The design of the robust XOR gate demonstrates the ability of NEURAE to find and use the inherent modularity within a problem. Having a GA which can discover and use modules on its own is particularly advantageous when these modules are not known beforehand. Furthermore, modules predetermined by a human designer may unintentionally exclude desirable designs. Embryogenesis also provides the scalability required to create parity networks of arbitrary size. NEURAE was able to evolve a genome which could create an even parity logic gate for 2 or 200 inputs. The fact that both ANNs could be made from the same four codons demonstrates that NEURAE can evolve large neural networks in a manner most neuro-evolutionary GAs cannot.

While this was an accomplishment in its own right, NEURAE was honed further through a sensitivity analysis of the mutation rates and types. Experiments were conducted to properly balance the point mutation, conjugation, gene duplication, gene deletion, and translocation mutation rates. As a result, the explorative and exploitative capabilities of NEURAE were optimized. It also shows that biologically inspired mutations, such as gene duplication and conjugation, are important to virtual evolutions as well. These experiments provided further evidence that as evolution used more information from the environment, the designs produced became more complex.

This refined version of NEURAE was used to make robotic controllers. The neural networks for these cases were able to find the correct controller logic by simulating the robot, not by fitting an explicit logic table. As a result, a controller can be designed without having to know the controller's precise functionality but instead by rewarding the higher level behavior.

These goals were achieved even with several constraints placed on NEURAE that are not necessary for future applications. For example, NEURAE is inherently able to make recurrent networks, but that ability was specifically removed in the examples provided here to simplify the evaluation of ANN logic. A version of NEURAE with recursion enabled could exhibit many of desirable properties networks mentioned in the introduction have, but with the modularity and scalability embryogenesis provides. NEURAE is also capable of generated networks that are not purely digital. Most ANN applications use a continuous activation function within each neuron to produce a range of values between -1 and 1. A particular benefit to using analog networks would be the ability to use Hebbian type learning, for control applications in particular. Nolfi et al. (1994), Stanley et al. (2003), and Soltoggio et al. (2007) have all successfully used reinforcement learning for the real time training of an ANN controller. However, these methods have been used for directly encoded genetic algorithms and are thus impractical for large networks. NEURAE, however, could find the core module necessary for such real-time learning ANNs and replicate it to make large networks.

Future iterations of NEURAE could benefit from other advancements in the field of evolutionary computation. One of the key components of NEAT (Stanley and Miikkulainen 2002) was an evaluation which rewarded robotic controllers for novelty. Instead of dictating a single evolutionary path with evaluation in tiers, rewarding novelty promotes several evolutionary paths at once. Another improvement might be the use of other selection methods. Rather than using the roulette method shown in Equation 2.7, selection can be done via tournaments (Miller and Goldberg 1995) or Pareto optimization (Horn et al. 1994).

These improvements would likely further optimize NEURAE for use in other applications. Many of the classification methods mentioned in the introduction train a large ANN with a set architecture. These training sessions are sensitive to the initial weights and the training sequence. NEURAE has shown it can make large, robust ANNs, and such ANNs would be less sensitive to varying initial weights and training sequences. As a result, better

classifiers could be made, which would have applications in computer vision for robotics, or many of the other fields mentioned in the introduction.

Most promising, the results obtained here may have implications beyond robotics and neuro-evolution. While the importance of point and crossover mutations have been well studied in classical GAs, the effects of gene duplication, gene deletion, and translocation have not. It would be interesting to study how these mutations affect other implicit GAs, and in particular, see if similar results are yielded. Likewise, Davidson (2006) has shown how controlling growth is an important feature of biological regulatory systems, and more work is needed to test the effect of regulatory systems in other GAs which use embryogenesis. Finally, the occurrence and correlation of punctuated equilibrium in an artificial evolution with embryogenesis is not well studied and is likely not unique to NEURAE.

Bibliography

Adami, C., Ofria, C., and Collier, T. C. (2000). Evolution of biological complexity. *Proceeding of the National Academy of Sciences*, 97(9):4463–4468.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1999). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54 – 65.

Ashlock, D. (2006). *Optimization and Modeling with Evolutionary Computation*. Springer-Verlag.

Astor, J. C. and Adami, C. (2000). A developmental model for the evolution of artificial neural networks. *Artificial Life*, 6(3):189–218.

Atiya, A. F. (2001). Bankruptcy prediction for credit risk using neural networks: A survey and new results. *IEEE Transactions on Neural Networks*, 12:929–935.

Bäck, T. (1992). Self-adaptation in genetic algorithm. In *Proceedings of the 1st European Conference on Artificial Life*, pages 263–271.

Bäck, T. and Schutz, M. (1996). Intellegent mutation rate control in canonical gas. In *Proceeding of Foundation of Intellegent Systems 9th International Symposium*, volume 2, pages 158–167.

Beer, R. D., Chiel, H. J., Quinn, R. D., Espenschied, K. S., and Larsson, P. (1992). A distributed neural network architecture for hexapod robot locomotion. *Neural Computation*, 4(3):356–365.

Bentley, P. and Kumar, S. (1999). Three ways to grow designs: A comparison of embryogenesis for an evolutionary design problem. In *Genetic and Evolutionary Computation Conference*, pages 35–43, New York, NY, USA. ACM.

Biewald, R. (1996). A neural network controller for the navigation and obstacle avoidance of a mobile robot. In Zalzala, A. M. S. and Morris, A. S., editors, *Neural network for robotic control*, pages 162–191. Ellis Horwood, Upper Saddle River, NJ, USA.

Bosman, R. J. C., van Leeuwen, W. A., and Wemmenhove, B. (2003). Combining hebbian and reinforcement learning in a minibrain model. *Neural Networks*, 17:29–36.

Britten, R. J. (2005). The majority of human genes have regions repeated in other human genes. *Proceeding of the National Academy of Sciences*, 102(15):5466–5470.

Calabretta, R., Nolfi, S., Parisi, D., and Wagner, G. P. (1998). Emergence of functional modularity in robots. In *Proceedings of the fifth international conference on simulation of adaptive behavior on From animals to animats*, pages 497–504.

Chen, S., Wu, Y., and Luk, B. L. (1999). Combined genetic algorithm optimization and regularized orthogonal least squares learning for radial basis function networks. *IEEE Transactions on Neural Networks*, 10(5):1239–1243.

Chialvo, D. R. and Bak, P. (1999). Learning from mistakes. *Neuroscience*, 90(4):1137–1167.

Cremean, L. B., Foote, T. B., Gillula, J. H., Hines, G. H., Kogan, D., Kriechbaum, K. L., Lamb, J. C., Leibs, J., Lindzey, L., Rasmussen, C. E., Stewart, A. D., Burdick, J. W., and Murray, R. M. (2006). Alice: An information-rich autonomous vehicle for high-speed desert navigation. *Journal of Field Robotics*, 23(9):777–810.

Cui, X. and Shin, K. G. (1993). Direct control and coordination using neural networks. *IEEE Transactions on Systems, Man and Cybernetics*, 23:686–697.

Daucé, E., Quoy, M., Cessac, B., and Samuelides, M. (1998). Self-organization and pattern-induced reduction of dynamics in recurrent networks. *Neural Networks*, 11:521–533.

Davidson, E. H. (2006). *The Regulatory Genome: Gene Regulatory Networks in Development and Evolution*. Elsevier, London, UK.

Duerr, P., Mattiussi, C., and Floreano, D. (2006). Neuroevolution with analog genetic encoding. In *Parallel Problem Solving from Nature*, volume 9, pages 671 – 680.

Dupuis, J.-F. and Parizeau, M. (2008). Evolving a vision-based line-following robot controller. In *Proceedings of the Third Canadian Conference on Computer and Robot Vision*, pages 75–81.

Eberhart, R. and Kennedy, J. (1995). A new optimizer using particles swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Piscataway, NJ. IEEE Press.

Eldredge, N. and Gould, S. J. (1972). Punctuated equilibria: An alternative to phyletic gradualism. In Schopf, T. J., editor, *Models in Paleobiology*, chapter 5, pages 82–115. Freeman, Cooper and Company, San Francisco, U.S.A.

Federici, D. and Downing, K. (2006). Evolution and development of a multicellular organism: Scalability, resilience, and neutral complexification. *Artificial Life*, 12(3):381–409.

Floreano, D., Mitri, S., and Magnenat, S. (2007). Evolutionary conditions for the emergence of communication in robots. *Crruent Biology*, 17(6):514–519.

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artifical intiellegence through simulation evolution*. Wiley, New York, NY, USA.

Fraternali, F., Porter, M. A., and Daraio, C. (2009). Optimal design of composite granular protectors. *Mechanics of Advanced Materials and Structures*. In Press.

Garis, H. D. (1992). Artificial embryology - the genetic programming of an artificial embryo. In Soucek, B. and IRIS, editors, *Dynamic, Genetic, and Chaotic Programming*, chapter 14, pages 373–393. Wiley, New York, NY, USA.

Graham, L., Cattral, R., and Oppacher, F. (2009). Beneficial preadaptation in the evolution of a 2d agent control system with genetic programming. In *Proceedings of the 12th European Conference on Genetic Programming*, pages 303 – 314.

Grajdeanu, A. (2007). Methods for open-box analysis in artificial development. In *Genetic and Evolutionary Computation Conference*, pages 1005–1012, New York, NY, USA. ACM.

Gruau, F. (1992). Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In *International Workshop on Combinations of Genetic Algorithms and Neural Networks, 1992 (COGANN-92)*, pages 55–74.

Gruau, F. (1994). *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France.

Harlan, R. M., Levine, D. B., and McClarigan, S. (2001). Evolving neural networks. *ACM SIGCSE Bulletin*, 33(1):105–109.

Hebb, D. (1949). *The Organization of Behavior*. Wiley, New York, NY, USA.

Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. *Neural Network for Perception*, 2:65–93.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA.

Holland, J. (1992). Genetic algorithms. *Scientific American*, 267(1):66–72.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceeding of the National Academy of Sciences*, 79(8):2554–2558.

Horn, J., Nafpliotis, N., and Goldberg, D. E. (1994). A niched pareto genetic algorithm for multiobjective optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 82–87.

Hornby, G. S., Lipson, H., and Pollack, J. B. (2001). Evolution of generative design systems for modular physical robots. *IEEE International Conference on Robotics and Automation*, pages 4146 – 4151.

Hornik, K., Stinchcombe, M. B., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359 – 366.

Jain, R., Rivera, M. C., and Lake, J. A. (1999). Horizontal gene transfer among genomes: The complexity hypothesis. *Proceeding of the National Academy of Sciences*, 96(7):3801–3806.

Kartalopoulos, S. V. (1996). *Understanding Neural Networks and Fuzzy Logic*. IEEE Press, Piscataway, NJ.

Kashtan, N. and Alon, U. (2005). Spontaneous evolution of modularity and network motifs. *Proceeding of the National Academy of Sciences*, 102(39):13773–13778.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–476.

Kitano, H. (1995). A simple model of neurogenesis and cell differentialtion based on evolutionary large-scale choas. *Artificial Life*, 2(1):79–97.

Koehn, P. (1996). Genetic encoding strategies for neural networks. In *Proceedings of Information Processing and Management of Uncertainty in Knowledge-Based Systems*.

Koza, J. R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 768–774, San Mateo, CA, USA. Morgan Kaufman.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Langdon, W. B. (2000). Quadratic bloat in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 451–458.

Lewis, M. A., Fagg, A. H., and Bekey, G. A. (1994). Genetic algorithms for gait synthesis in a hexapod robot. In Zheng, Y. F., editor, *Recent Trends in Mobile Robots*. World Scientific, New Jersey, USA.

Lewis, R. (2005). *Human Genetics: Concepts and Applications*. McGraw-Hill.

Lipson, H. and Pollack, J. B. (2000). Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–977.

Luke, S. and Spector, L. (1996). Evolving graphs and networks with edge encoding: Preliminary report. In *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 117–124.

Martin, W. and Russell, M. J. (2002). On the origins of cells: a hypothesis for the evolutionary transition from abiotic geochemistry to chemoautotropic prokaryotes, and from prokaryotes to nucleated cells. *Philosophical Transactions of the Royal Society*, 358:59–85.

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

McGinley, B., Morgan, F., and O'Riordan, C. (2008). Maintaining diversity through adaptive selection, crossover and mutation. In *Genetic and Evolutionary Computation Conference*, pages 1127–1128, New York, NY, USA. ACM.

Miller, B. L. and Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3):193–212.

Miller, J. F., P.Thomson, and Fogarty, T. (1997). Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In Quagliarella, D., Periaux, J., Poloni, C., and Winter, G., editors, *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications.* Wiley, New York, NY, USA.

Montana, D. J. and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767, San Mateo, CA,USA. Morgan Kaufmann.

Montgomery, D. C. (2004). *Design and Analysis of Experiments.* Wiley, New York, NY, USA, 6 edition.

Morris, G. M., Goodsell, D. S., Halliday, R. S., Huey, R., Hart, W. E., Belew, R. K., and Olson, A. J. (1998). Automated docking using a lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry*, 19(14):1639–1662.

Mühlenbein, H. (1992). How genetic algorithms really work I: Mutation and hillclimbing. In *Parallel Problem Solving from Nature*, volume 2, pages 15–25.

Murray, R. M. (2007). Recent research in cooperative controls of mulit-vehicle sytems. *ASME Journal of Dynamic Systems Measurement and control*, 129(5):571–582.

Naito, T., Odagiri, R., Matsunaga, Y., Tanifuji, M., and Murase, K. (1997). Genetic evolution of a logic circuit which controls an autonomous mobile robot. *Lecture Notes in Computer Science*, 1259:210–219.

Nolfi, S., Miglino, O., and Parisi, D. (1994). Phenotypic plasticity in evolving neural networks. In *Proceedings of the First Conference Frome Perception to Action*, pages 146–157. IEEE Computer Society Press.

Ochman, H., Lawrence, J. G., and Groisman, E. A. (2000). Lateral gene transfer and the nature of bacterial innovation. *Nature*, 405:299–304.

Odewahn, S. C., Pennington, E. B. S. R. L., Humphreys, R. M., and Zumach, W. A. (1992). Automated star / galaxy discrimination with neural networks. *Astronomical Journal*, 103(1):318 – 331.

Ohno, S. (1970). *Evolution by gene duplication.* Springer-Verlag.

Oja, E. (1992). Princle components, minor components, and linear neural networks. *Neural Networks*, 5:927–935.

Onat, A., Kita, H., and Nishikawa, Y. (1998). Recurrent neural networks for reinforcement learning: architecture, learning algorithms and internal representation. In *IEEE International Joint Conference on Neural Networks*, pages 2010–2015.

O'Neill, M. and Ryan, C. (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5:349–358.

Pollack, J. B., Hornby, G. S., Lipson, H., and Funes, P. (2003). Computer creativity in the automatic design of robots. *Leonardo*, 36(2):115–121.

Reed, R. (1999). Pruning algorithms - a survey. *IEEE Transactions on Neural Networks*, 4(5):740–744.

Roy, A., Govil, S., and Miranda, R. (1999). A neural-network learning theory and a polynomial time rbf algorithm. *IEEE Transactions on Neural Networks*, 8(6):1301–1306.

Roy, A. M., Antonsson, E. K., and Shapiro, A. A. (2008). An investigation into the structure of genomes within an evolution that uses empryogenesis. In *Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2137–2142, New York, NY, USA. ACM.

Sanger, T. D. (1989). Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks*, 2:459–473.

Soltoggio, A., Peter Dü, r., Mattiussi, C., and Floreano, D. (2007). Evolving neuromodularity topologies for reinforcement learning-like problems. In *IEEE International Joint Conference on Neural Networks*, pages 2010–2015.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2003). Evolving adaptive neural networks with and without adaptive synapses. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2003*, pages 2557–2564.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005). Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–693.

Stanley, K. O., D'Ambrosio, D., and Gauci, J. (2009). A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–223.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Stanley, K. O. and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93 – 130.

Sutton, R. S. (1986). Two problems with backpropogation and other steepest-descent learning procedures for networks. In *Proceeding of the Eighth Annual Conference of the Cognitive Science Society*, pages 823–831.

Sutton, R. S. and Barto, A. G. (1999). Reinforcement learning. *Journal of cognitive neuroscience*, 11:126–134.

Szathmáry, E. and Smith, J. M. (1995). The major evolutionary transitions. *Nature*, 374:227–232.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257–277.

Theraulaz, G. and Bonabeau, E. (1995). Coordination in distributed building. *Science*, 269(5224):686–688.

Tsoulos, I. G., Gavrili, D., and Glavas, E. (2005). Neural network construction using grammatical evolution. In *IEEE International Symposium on Signal Processing and Information Technology*, pages 827–831.

Tufte, G. and Haddow, P. C. (2000). An evolvable hardware fpga for adaptive hardware. In *Proceedings of the 2000 Conference on Evolutionary Computation*, pages 553–560.

Turing, A. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.

Vigraham, S. A., Gallagher, J. C., and Boddhu, S. K. (2005). Evolving analog controllers for correcting thermoacoustic instability in real hardware. In *Genetic and Evolutionary Computation Conference*, pages 933–940, New York, NY, USA. ACM.

Waibel, A. (1989). Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1:39 – 46.

Weibull, W. (1951). A statistical distribution function of wide applicability. *Journal of applied mechanics*, 18:292–297.

Wu, Y., Giger, M. L., Doi, K., and Vyborny, K. (1993). Artificial neural networks in mammography: Application to decision making in the diagnosis of breast cancer. *Radiology*, 187(1):81.

Yakovenko, S., Gritsenko, V., and Prochazka, A. (2004). Contribution of strech reflexes to locomotor control: a modeling study. *Biological Cybernetics*, 90:146–155.

Yogev, O. and Antonsson, E. K. (2007). Growth and development of continuous structures. In *Genetic and Evolutionary Computation Conference*, pages 1064–1065, New York, NY, USA. ACM.

Yue, S. and Rind, F. C. (2006). Collision detection in complex dynamic scenes using an lgmd-based visual neural network with feature enhancement. *IEEE TRANSACTIONS ON NEURAL NETWORKS*, 17(3):705–716.

Zhang, Y., Antonsson, E. K., and Martinoli, A. (2008). Evolutionary engineering design synthesis of on-board traffic monitoring sensors. *Research in engineering design*, 19:113–125.

Appendix

Included in the appendix is the source code used to make NEURAE work. Because many different version of NEURAE were developed in the process of this thesis, the codes have several sections which were obsolete or never finished. Furthermore, the first three programs listed were for evolving the robust XOR logic gate, while the final library was used for the evolution of swarming robot controllers.

- *Evovle.cpp* is the executable program, and contains the various libraries listed afterwards. This is the program which conducted the genetic algorithm. Pages 86 - 106
- *node_lib.h* is a libraries which defined the node and neural network object class, as well as useful functions for both. Pages 107 - 115
- *evo_lib.h* contains many useful functions used throughout evolution, such as those used for evaluation and mutation. It also contains the definition for an individual as well. Furthermore, the *make_protien* function contained within *evo_lib.h* was responsible for transcibing the integers of an individual's genome into a compilable C^{++} program. Pages 116 - 163
- *robot_lib.h* contains the definition and functions needed for robot simulations. Pages 164 - 180

```
1 //This is the main script that will control evolution
2
3 #include <iostream>
4 #include <fstream>
5 #include <vector>
6 #include <string>
7 #include <sstream>
8 #include <ctime>
9 #include <math.h>
10 #include "chimera_lib.h"
11 #include "node_lib_omega4.h"
12 #include "evo_lib_omega4.h"
13 #include <mpi.h>
14
15 using namespace std;
16
17
18 int main(int argc, char *argv[]){
19     //----- These values are set by arguments during ↙
       program calls ----------------
20     //Template for new runs:
21     //    Evolve.exe N last_generation default genome ↙
       length
22     //Template for continuing runs
23     //    Evolve.exe -c N last_generation ↙
       restarting_generation
24     char restart; //This determines whether evolution will ↙
        start from scratch or a member of Ark.txt
25     int N; //The number of individuals per generation
26     int default_genome_length;
27     int last_generation; //Max number of generations
28     int start_gen,counter; //Used for regenesis
29     vector<int> restart_individuals; //Used for regenesis
30     //----------------------------------------------
31     //--- Number of inputs and outputs for each ANN ------ ↙
       --
32     const int no_of_inputs = 2;
33     const int max_no_of_outputs = 1;
34     const int max_connections = 99;
35     //------ Mutation Rates & Values -----------
36     const float mu = 1.0;   //Chance of mutation at each ↙
       reading frame
37     vector < vector<float> > mutation_ratios;  //Ratio for ↙
        each case.  The # of cases determines the number of ↙
       children each individual can have
38     vector<float> mu_ratio (5, 1.0); //Mutation rate of ↙
       each mutator.  Make sure they add up to 1.0
39     mu_ratio[0] = 0.40;mu_ratio[1] = 0.30;mu_ratio[2] = 0. ↙
       00;mu_ratio[3] = 0.30;mu_ratio[4] = 0.00;
40     mutation_ratios.push_back(mu_ratio);
41     mu_ratio[0] = 0.40;mu_ratio[1] = 0.30;mu_ratio[2] = 0. ↙
       00;mu_ratio[3] = 0.30;mu_ratio[4] = 0.00;
```

```
42      mutation_ratios.push_back(mu_ratio);
43      mu_ratio[0] = 0.40;mu_ratio[1] = 0.30;mu_ratio[2] = 0.
        00;mu_ratio[3] = 0.30;mu_ratio[4] = 0.00;
44      mutation_ratios.push_back(mu_ratio);
45      float mu_point_mutation;
46      float mu_conjugation;
47      float mu_recopy;
48      float mu_deletion;
49      float mu_translocation;
50      //----------------------------
51      //----- Random Seeding   -----------
52      time_t start,end,seed;
53      int dif_t;
54      time (&start);  //Sets the start time for this
        evolution run
55      time (&seed);
56      //seed = 1244495693;
57      srand(seed);  //Seeds the randomizer
58      //--------------------------
59      //------Used for organizing individuals throughout
        evolution -----------------------
60      int generation = 0;//The current generation
61      int newly_made,vets,reduced; //Keeps track of the
        number of individuals made each generation
62      int Ark_no, my_Ark_no, Ark_no2;  //Used to keep track
        of Ark numbers within the hub and satellite computers
63      int genome_size;
64      vector<int> recalled_genome; //Used for regenesis
65      vector<individual> Ark; //Holds all the individuals
66      vector<individual> my_Ark; //Array for satellite
        computers that has its individuals
67      vector<int> my_Ark_conversion; //Ark_conversion[i] on
        satellite comp == Ark_no on comp 0
68      int Ark_search;  //Used to find the right Ark_no on
        satellites
69      //----------------------------------------------------
        ---------
70      //----- Used for selection in survival and proceation
        ---------------
71      vector<int> procreation; //For selecting whose genes
        will be passed on
72      vector<int> unmade; //Individuals whose ANN's haven't
        been made
73      vector<int> alive; // All individuals alive this
        generation
74      vector<int> still_alive; // Individuals that were
        alive this generation and will live onto the next
75      vector<int> stay_alive; //Individuals that have been
        selected during death and procreation loops
76      float all_fitness,max_fitness;
77      int lucky_one,newbies,mutation_method;
78      vector< vector<int> > mutation_info; //The individuals
```

```
    ' subject number and the method of its mutation
79      vector< vector<int> > mutation_Ark;  //The individuals ↙
    ' genomes
80      float low_fit, high_fit, range_fit, num_fit;
81      int selection_q, low_int, high_int, range_int, num_int ↙
    ;
82      //----------------------------------------------------- ↙
    -
83      //  -------- Declared here and used to temporarily     ↙
    hold info --------------
84      int junk_int;
85      char junk_char;
86      float junk_float;
87      vector<int> junk_ints;
88      //-----------------------------
89      //MPI Variables
90      MPI::Init(argc, argv);
91      int dest, noProcesses, processID, tag, src;
92      int hub = 0;
93      vector<int> mutationID, embryogenesisID;
94      tag = 0;
95      MPI_Status status;
96      noProcesses = MPI::COMM_WORLD.Get_size();
97      processID = MPI::COMM_WORLD.Get_rank();
98      int data_pack[2]; //Used to hold info when sending    ↙
    info to other comps
99      int data_pack2[3]; //Also used to hold info when      ↙
    sending info to other comps in mutation loop
100     int back_size; //Tells hub how much info is being sent ↙
     back
101     vector<int> temp_genome; //Used as a temp place holder ↙
     for sending genomes to other comps
102     vector<int> temp_genome2;
103     vector<int> sub_back; //Used to send subject numbers   ↙
    back to computer hub
104     vector<float> fit_back; //Used to send fitnesses back  ↙
    to computer hub
105     vector<int> ruleset_length_back; //Used to prep       ↙
    computer hub for the number of rules coming back
106     vector<int> ruleset_back;  //Rules creating in making  ↙
    each ANN.
107     int ruleset_back_size;  //Same as ruleset_back.size(), ↙
     but shorter way can't be used alone
108     //----------//
109
110     if((processID%8) == 0){ //This is a fix for GARUDA so  ↙
    each computer does this only once
111         //Cleans up scratch if anything is there
112         //Copies necessary libraries to /scratch           ↙
    directories of each comp
113         //-------------------CHANGE THIS FOR EACH VERSION- ↙
    ----------------------
```

```
114         system("scp /home/roy/chimera_lib.h /scratch/
    chimera_lib.h");
115         system("scp /home/roy/Evolution/Version_omega4/
    node_lib_omega4.h /scratch/node_lib_omega4.h");
116         //-----------------------------------------------
    -------------------------------
117     }
118     else{
119         system("sleep 5");
120     }
121     //------------------------------------------------
122     //This if/else loop determines if we are continuing
    from a past evo run or starting a new one, then sets
    the variables accordingly
123     if(argc == 5){
124         N = atoi(argv[2]);
125         last_generation = atoi(argv[3]);
126         start_gen = atoi(argv[4]);
127         counter = 0; //Tracks how many indivuals have been
     restarted
128         Ark_no = 0;
129         if (processID == hub){
130             //First, we read the Chronograph to see which
    individuals were alive at the given gen
131             ifstream infile1("Chronograph.txt");
132             for(int i=0;i<start_gen;i++){ //Skips down the
     right gen
133                 infile1>>junk_int; //Gets the gen
134                 for(int j=0;j<N;j++){
135                     infile1>>junk_int; //Gets the subject
    number
136                     infile1>>junk_float; //Get the fitness
137                 }
138             }
139             infile1>>junk_int; //Gets the gen again
140             for(int i=0;i<N;i++){  //Gets and saves the
    subject numbers
141                 infile1>>junk_int; //Gets the subject
    number
142                 restart_individuals.push_back(junk_int);
143                 infile1>>junk_float; //Get the fitness
144             }
145             if(restart_individuals.size()!= N){ //Check
146                 cout<<"There was a problem with
    determining which individuals were alive at generation
     "<<start_gen<<endl;
147                 return 0;
148             }
149             sort_vector(restart_individuals);
150             cout<<"Individuals to be restarted from
    generation "<<start_gen<<":"<<endl;
151             print_vector(restart_individuals);
```

```
152             system("cp ./Ark.txt ./Arktmp.txt"); //Creates ↙
      a temp file to read from
153             system("rm Ark.txt");
154             system("rm Chronograph.txt");
155             ofstream datafile_temp("Ark.txt");
156             datafile_temp<<seed<<endl; //Gets the seed
157             ifstream infile2("Arktmp.txt");
158             //Now we read through the Ark and compares the ↙
      subject number of the Ark with the individuals marked ↙
      for restart
159             infile2>>junk_int; //Gets the old seed
160             while(counter<N){
161                 if(any(restart_individuals,Ark_no)){//Save ↙
      the individual
162                     infile2>>junk_int;
163                     infile2>>junk_int;
164                     infile2>>junk_int;
165                     infile2>>junk_char;
166                     infile2>>genome_size;
167                     for(int j=0;j<genome_size;j++){
168                         infile2>>junk_int;
169                         recalled_genome.push_back         ↙
     (junk_int);
170                     }
171                     generate_designed(Ark,recalled_genome, ↙
     generation);
172                     recalled_genome.clear();
173                     Ark_Load(Ark[counter]);
174                     unmade.push_back(counter);
175                     alive.push_back(counter);
176                     cout<<"Individual "<<Ark_no<<" was      ↙
     reborn as "<<Ark[counter].get_fcall()<<endl;
177                     counter++;
178                 }
179                 else{//Discards it
180                     infile2>>junk_int;
181                     infile2>>junk_int;
182                     infile2>>junk_int;
183                     infile2>>junk_char;
184                     infile2>>genome_size;
185                     for(int j=0;j<genome_size;j++){
186                         infile2>>junk_int;
187                     }
188                 }
189                 Ark_no++;// Moves onto next individual in  ↙
     the Ark
190             }
191             system("rm Arktmp.txt");
192             newly_made = unmade.size();
193             vets = 0;
194             reduced = 0;
195         }
```

```
196     }
197     else{
198         N = atoi(argv[1]);
199         last_generation = atoi(argv[2]);
200         default_genome_length = atoi(argv[3]);
201         if (processID == hub){
202             system("rm Ark.txt");
203             system("rm Chronograph.txt");
204             ofstream datafile_temp("Ark.txt");
205             datafile_temp<<seed<<endl;
206             for(int i=0;i<N;i++){ //This will generate N ↙
        random individuals
207                 generate_random(Ark,default_genome_length, ↙
        generation);
208                 Ark_Load(Ark[i]);
209                 unmade.push_back(i);
210                 alive.push_back(i);
211             }
212             newly_made = unmade.size();
213             vets = 0;
214             reduced = 0;
215         }
216     }
217     //-----------------------------------------
218     // --- This partions the satellites into evaluators ↙
        and mutators -----------
219     if((int(N/24)+2)<noProcesses){
220         for(int i=0;i<N/24;i++){
221             mutationID.push_back(i+1);
222         }
223         if(mutationID.size()==0){ //A fix for small runs ↙
        where there would be no mutation processor
224             mutationID.push_back(1);
225         }
226         for(int i=(mutationID.size()+1);i<noProcesses;i++) ↙
        {
227             embryogenesisID.push_back(i);
228         }
229     }
230     else{
231         cout<<"Use more processors or this will be VERY ↙
        slow"<<endl;
232         mutationID.push_back(1);
233         for(int i=1;i<noProcesses;i++){
234             embryogenesisID.push_back(i);
235         }
236     }
237     // --------------------------------------------------- ↙
        ------
238     MPI::COMM_WORLD.Bcast(&newly_made,1,MPI::INT,hub);
239     MPI::COMM_WORLD.Bcast(&vets,1,MPI::INT,hub);
240     MPI::COMM_WORLD.Bcast(&reduced,1,MPI::INT,hub);
```

```
241     //  ------------------------------------------------------- ↙
        ---------------------
242     //  -------------- BEGINNING OF EVOLUTION LOOP --------- ↙
        ---------------------
243     //  ------------------------------------------------------- ↙
        ---------------------
244     for(generation;generation<last_generation;generation+ ↙
        +){
245         if(processID == hub){
246             cout<<"Generation: "<<generation<<endl;
247             assert((still_alive.size()+unmade.size())==N);
248             assert(alive.size()==N);
249         }
250 
251         // --- Re-evaluates the individuals that lived ↙
        from last generation ---
252         if(generation != 0){
253             for(int i=0;i<vets;i++){
254                 if(processID == hub){
255                     Ark_no = still_alive[i];
256                 }
257                 MPI::COMM_WORLD.Bcast(&Ark_no,1,MPI::INT, ↙
        hub);
258                 dest = embryogenesisID[Ark_no% ↙
        (embryogenesisID.size())]; //See page 20 Vol. 2 for ↙
        logic
259                 if(processID == hub){
260                     Ark_no = still_alive[i];
261                     MPI::COMM_WORLD.Send(&Ark_no,1,MPI:: ↙
        INT,dest,tag);
262                 }
263                 if(processID == dest){
264                     MPI::COMM_WORLD.Recv(&Ark_no,1,MPI:: ↙
        INT,hub,tag);
265                     Ark_search = -1;
266                     int my_Ark_counter = 0;
267                     while(Ark_search < Ark_no){
268                         Ark_search = my_Ark_conversion ↙
        [my_Ark_counter];
269                         my_Ark_counter++;
270                     }
271                     my_Ark_no = my_Ark_counter-1;
272                     //cout<<"Process "<<processID<<" is re ↙
        -evaluating "<<my_Ark[my_Ark_no].get_fcall()<<endl;
273                     if(my_Ark[my_Ark_no].get_fitness() >= ↙
        pow(2.0,(2*max_no_of_outputs - 1))){ //Repeats if a ↙
        good ANN is made ↙

274                         my_Ark[my_Ark_no].eval_robustness ↙
        ();
275                     }
276                     sub_back.push_back(Ark_no);
```

```
277                         fit_back.push_back(my_Ark[my_Ark_no].  ↙
      get_fitness());
278                     }
279                 }
280                 if(any(embryogenesisID,processID)){  //Sends  ↙
      results to process the hub
281                     back_size = sub_back.size();
282                     MPI::COMM_WORLD.Send(&back_size,1,MPI::INT ↙
      ,hub,tag);
283                     MPI::COMM_WORLD.Send(&sub_back[0],         ↙
      back_size,MPI::INT,hub,tag);
284                     MPI::COMM_WORLD.Send(&fit_back[0],         ↙
      back_size,MPI::FLOAT,hub,tag);
285                     //cout<<"Process ID "<<processID<<" sent   ↙
      back (from re-evaluation):"<<endl;
286                     //print_vector(sub_back);
287                     sub_back.clear();
288                     fit_back.clear();
289                 }
290                 if(processID == hub){  //The hub collects      ↙
      results
291                     for(int i=0; i<embryogenesisID.size(); i+  ↙
      +){
292                         src = embryogenesisID[i];
293                         MPI::COMM_WORLD.Recv(&back_size,1,MPI: ↙
      :INT,src,tag);
294                         sub_back.resize(back_size);
295                         fit_back.resize(back_size);
296                         MPI::COMM_WORLD.Recv(&sub_back[0],     ↙
      back_size,MPI::INT,src,tag);
297                         MPI::COMM_WORLD.Recv(&fit_back[0],     ↙
      back_size,MPI::FLOAT,src,tag);
298                         //cout<<"Hub received ";
299                         //print_vector(sub_back);
300                         //cout<<" from process "<<src<<endl;
301                         for(int j=0;j<back_size;j++){
302                             Ark[sub_back[j]].make_fitness      ↙
      (fit_back[j]);//Gives the individual sub_back[i] the   ↙
      fitness fit_back[i]
303                         }
304                         sub_back.clear();
305                         fit_back.clear();
306                     }
307                 }
308             }
309             // ------------- End of re-evaluating survivors -- ↙
      ----------------
310
311             // ---------------------------------------------- ↙
      ----------------
312             // -------  Sends out indivuals for embryogenesis  ↙
      and evaluation-----
```

```
313         for(int i=0;i<(newly_made+reduced);i++){
314             if(processID == hub){
315                 Ark_no = unmade[i];
316             }
317             MPI::COMM_WORLD.Bcast(&Ark_no,1,MPI::INT,hub);
318             dest = embryogenesisID[Ark_no%(embryogenesisID ↙
        .size())]; //See page 20 Vol. 2 for logic
319             // --- Hub loop -------
320             if(processID == hub){
321                 Ark_no = unmade[i];
322                 genome_size = Ark[Ark_no].                    ↙
        get_genome_length();
323                 data_pack[0] = Ark_no;
324                 data_pack[1] = genome_size;
325                 //cout<<Ark[Ark_no].get_fcall()<<" was        ↙
        sent to process "<<dest<<" for evaluation."<<endl;     ↙

326                 for(int j=0;j<genome_size;j++)
327                     temp_genome.push_back(Ark[Ark_no].        ↙
        get_genome(j));
328                 MPI::COMM_WORLD.Send(&data_pack,2,MPI::INT ↙
        ,dest,tag);
329                 MPI::COMM_WORLD.Send(&temp_genome[0],         ↙
        genome_size,MPI::INT,dest,tag);
330                 temp_genome.clear(); //Empties for next       ↙
        time
331             }
332             // -------------------------------------------- ↙
        -----------------
333             // -------------- Satellite Loop ------------- ↙
        ---------
334             if(processID == dest){
335                 MPI::COMM_WORLD.Recv(&data_pack[0],2,MPI:: ↙
        INT,hub,tag);
336                 Ark_no = data_pack[0];
337                 my_Ark_conversion.push_back(Ark_no);
338                 my_Ark_no = my_Ark.size();
339                 genome_size = data_pack[1];
340                 temp_genome.resize(genome_size);
341                 MPI::COMM_WORLD.Recv(&temp_genome[0],         ↙
        genome_size,MPI::INT,hub,tag);
342                 generate_satellite(my_Ark,temp_genome,        ↙
        generation,Ark_no);
343                 make_protein(my_Ark[my_Ark_no],              ↙
        no_of_inputs,max_no_of_outputs,max_connections,       ↙
        processID);
344                 my_Ark[my_Ark_no].make_ANN(processID);
345                 my_Ark[my_Ark_no].eval_XOR_logic();
346                 my_Ark[my_Ark_no].eval_robustness();
347                 //cout<<"Process ID = "<<processID<<"         ↙
        Ark_no = "<<Ark_no<<" my_Ark_no = "<<my_Ark_no<<endl;  ↙
```

```
348                     sub_back.push_back(Ark_no);
349                     fit_back.push_back(my_Ark[my_Ark_no].
        get_fitness());
350                     ruleset_length_back.push_back(my_Ark
        [my_Ark_no].get_rules_length());
351                     for(int j=0;j<my_Ark[my_Ark_no].
        get_rules_length();j++){
352                         ruleset_back.push_back(my_Ark
        [my_Ark_no].get_rule(j));
353                     }
354                 }
355                 //-------------------------------
356             }
357             //-------------------------------
358             //  ------------ Collects the info at the hub ----
        ------------------------
359             //  ------------  Satellite loop -----------------
        ---------------
360             if(any(embryogenesisID,processID)){  //Sends
        results to process 0
361                 back_size = sub_back.size();
362                 MPI::COMM_WORLD.Send(&back_size,1,MPI::INT,hub
        ,tag);
363                 MPI::COMM_WORLD.Send(&sub_back[0],back_size,
        MPI::INT,hub,tag);
364                 MPI::COMM_WORLD.Send(&fit_back[0],back_size,
        MPI::FLOAT,hub,tag);
365                 MPI::COMM_WORLD.Send(&ruleset_length_back[0],
        back_size,MPI::INT,0,tag);
366                 //cout<<"Process ID "<<processID<<" sent back:
        "<<endl;
367                 //print_vector(sub_back);
368                 ruleset_back_size = 0;
369                 for(int j=0;j<back_size;j++){
370                     ruleset_back_size = ruleset_back_size +
        ruleset_length_back[j];
371                 }
372                 MPI::COMM_WORLD.Send(&ruleset_back[0],
        ruleset_back_size,MPI::INT,0,tag);
373                 sub_back.clear();
374                 fit_back.clear();
375                 ruleset_length_back.resize(0);
376                 ruleset_back.resize(0);
377             }
378             // ------------------------------------------
379             // ----------- Hub loop -------------------------
380             if(processID == hub){  //If rank is 0, collect
        results and preps for next gen
381                 for(int i=0; i<embryogenesisID.size(); i++){

382                     src = embryogenesisID[i];
383                     MPI::COMM_WORLD.Recv(&back_size,1,MPI::INT
```

```
        ,src,tag);
384                     sub_back.resize(back_size);
385                     fit_back.resize(back_size);
386                     ruleset_length_back.resize(back_size);
387                     MPI::COMM_WORLD.Recv(&sub_back[0],                ↙
        back_size,MPI::INT,src,tag);
388                     MPI::COMM_WORLD.Recv(&fit_back[0],                ↙
        back_size,MPI::FLOAT,src,tag);
389                     MPI::COMM_WORLD.Recv(&ruleset_length_back         ↙
        [0],back_size,MPI::INT,src,tag);
390                     ruleset_back_size = 0;
391                     for(int j=0;j<back_size;j++){
392                         ruleset_back_size = ruleset_back_size         ↙
        + ruleset_length_back[j];
393                     }
394                     ruleset_back.resize(ruleset_back_size);
395                     MPI::COMM_WORLD.Recv(&ruleset_back[0],            ↙
        ruleset_back_size,MPI::INT,src,tag);
396                     int rule_pointer = 0;
397                     for(int j=0;j<back_size;j++){
398                         Ark[sub_back[j]].make_fitness(fit_back        ↙
        [j]);//Gives the individual sub_pack[i] the fitness               ↙
        fit_pack[i]
399                         for(int k=0;k<ruleset_length_back[j];k        ↙
        ++){
400                             Ark[sub_back[j]].save_rule                ↙
        (ruleset_back[rule_pointer+k]);
401                         }
402                         rule_pointer = rule_pointer +                 ↙
        ruleset_length_back[j];
403                     }
404                     sub_back.clear();
405                     fit_back.clear();
406                     ruleset_length_back.resize(0);
407                     ruleset_back.resize(0);
408                 }
409                 Record_Gen(Ark,still_alive,unmade,generation);        ↙
         //Saves final state
410                 newly_made = 0;
411                 reduced = 0;
412                 unmade.clear(); //Empties unmade...
413             }
414             // ---------------------------------
415             // --- The hub selectes the survivors and parents         ↙
        for the next generation ----------
416             if(processID == hub){   //If rank is 0, collect           ↙
        results and preps for next gen
417                 //All that are alive have a chance to                 ↙
        procreate
418                 procreation.clear();
419                 for(int i=0;i<alive.size();i++){
420                     Ark_no = alive[i];
```

```
421                     procreation.push_back(Ark_no);
422                 }
423                 /*
424                 //------------Fitness check-------------------
425
426                 for(int i=0;i<alive.size();i++){
427                     cout<<Ark[alive[i]].get_fcall()<<" has a
        fitness of "<<Ark[alive[i]].get_fitness()<<endl;
428                 }
429                 */
430
431                 //-----------------------------DEATH LOOP----
        ---------------------------
432                 while(stay_alive.size()<N/4){
433                     all_fitness = 0;
434                     max_fitness = 0;
435                     for(int i=0;i<alive.size();i++){
436                         Ark_no = alive[i];
437                         if(!any(stay_alive,Ark_no)){
438                             all_fitness += Ark[Ark_no].
        get_fitness();
439                         }
440                         if(Ark[Ark_no].get_fitness()==-1){
441                             cout<<Ark[Ark_no].get_fcall()<<"
        wasn't evaluated.  Ending program."<<endl;
442                             return 0;
443                         }
444                         // -----------The fittest one last
        made is always pardoned!---------------
445                         if((Ark[Ark_no].get_fitness()>=
        max_fitness)&&(stay_alive.size()==0)){
446                             max_fitness = Ark[Ark_no].
        get_fitness();
447                             lucky_one = Ark_no;
448                         }
449                     }
450                     all_fitness -= max_fitness;
451                     //Max fitness is always 0 if something has
         been pardoned
452                     //This does the actually pardoning of the
        fittest one last made
453                     if(stay_alive.size() == 0){
454                         stay_alive.push_back(lucky_one);
455                         cout<<Ark[lucky_one].get_fcall()<<"
        has stayed alive (ELITE) with fitness: "<<Ark
        [lucky_one].get_fitness()<<endl;
456                     }
457                     //----------End of elite selection--------
        --------
458                     //cout<<"End of elite selection"<<endl;
```

```
459                 if(all_fitness!=0){
460                     num_fit = random_float(.000001,
    all_fitness);
461                     selection_q = 0;
462                     while(num_fit>0){
463                         Ark_no = alive[selection_q];
464                         if(!any(stay_alive,Ark_no)){
465                             num_fit -= Ark[Ark_no].
    get_fitness();
466                         }
467                         selection_q++;
468                     }
469                     stay_alive.push_back(Ark_no);//
    PARDONED!!
470                     cout<<Ark[Ark_no].get_fcall()<<" has
    stayed alive with fitness: "<<Ark[Ark_no].get_fitness
    ()<<endl;
471                 }
472                 else{
473                     //cout<<"Zero fitness"<<endl;
474                     low_int = 0;
475                     high_int = alive.size()-1;
476                     vector<int> exclude;
477                     for(int i=0;i<alive.size();i++){
478                         if(any(stay_alive,alive[i])){
479                             exclude.push_back(i);
480                         }
481                     }
482                     num_int = random_int(low_int,high_int,
    exclude);
483                     Ark_no = alive[num_int];
484                     stay_alive.push_back(Ark_no); //
    PARDONED (Zero Fitness)!!
485                     cout<<Ark[Ark_no].get_fcall()<<" has
    randomly stayed alive with "<<Ark[Ark_no].get_fitness
    ()<<" (zero) fitness."<<endl;
486                 }
487             }
488 
489             for(int i=0;i<alive.size();i++){
490                 if(!any(stay_alive,alive[i])){
491                     Ark[alive[i]].kill(generation); //COLD
    -BLOODED!!
492                     //cout<<Ark[alive[i]].get_fcall()<<"
    did not make it across the river."<<endl;
493                 }
494             }
495 
496             alive.clear(); //Empties alive...
497             still_alive.clear();//...and empties
    still_alive...
498             for(int i=0;i<stay_alive.size();i++){//...then
```

```
        refills them with stay alive
499                     alive.push_back(stay_alive[i]);
500                     //cout<<Ark[stay_alive[i]].get_fcall()<<" ↙
        is alive."<<endl;
501                     still_alive.push_back(stay_alive[i]);
502                     //cout<<Ark[stay_alive[i]].get_fcall()<<" ↙
        is still alive."<<endl;
503                 }
504                 stay_alive.clear();
505                 vets = still_alive.size();
506                 //--------------------------------------------- ↙
        ------------------------------
507             }
508             MPI::COMM_WORLD.Bcast(&vets,1,MPI::INT,hub);
509
510             //-----------------Procreation Selection Loop(s)--- ↙
        -------------------------
511             //This (these) loops will select a primary and     ↙
        secondary parent for each loop
512             //The number of loops is determined by the number  ↙
        of mutation ratio sets
513             if(processID == hub){
514                 while((newly_made+reduced+still_alive.size() < ↙
         N)){
515                     for(int i=0;i<mutation_ratios.size();i++){ ↙

516                         vector<int> primary_parents;
517                         vector<int> secondary_parents;
518                         while((secondary_parents.size()<((N-   ↙
        (reduced+still_alive.size()))/mutation_ratios.size     ↙
        ()))){
519                             //The primary parent is selected   ↙
        first
520                             all_fitness = 0;
521                             for(int j=0;j<procreation.size();j ↙
        ++){
522                                 Ark_no = procreation[j];
523                                 if(!any(primary_parents,       ↙
        Ark_no)){
524                                     all_fitness += Ark[Ark_no] ↙
        .get_fitness();
525                                 }
526                             }
527                             if(all_fitness!=0){
528                                 num_fit = random_float(.000001 ↙
        ,all_fitness);
529                                 selection_q = 0;
530                                 while(num_fit>0){
531                                     Ark_no = procreation       ↙
        [selection_q];
532                                     if(!any(primary_parents,   ↙
        Ark_no)){
```

```
533                                     num_fit -= Ark[Ark_no] ↙
        .get_fitness();
534                                 }
535                                 selection_q++;
536                             }
537                             primary_parents.push_back       ↙
        (Ark_no);
538                             //cout<<Ark[Ark_no].get_fcall   ↙
        ()<<" with fitness "<<Ark[Ark_no].get_fitness()<<" was ↙
         selected as primary parent for selection loop "<<i<   ↙
        <endl;
539                         }
540                         else{
541                             //cout<<"Zero fitness loop for ↙
         primary parent in selection loop "<<i<<endl;
542                             low_int = 0;
543                             high_int = procreation.size()- ↙
        1;
544                             vector<int> exclude;
545                             for(int i=0;i<procreation.size ↙
        ();i++){
546                                 if(any(primary_parents,     ↙
        procreation[i])){
547                                     exclude.push_back(i);
548                                 }
549                             }
550                             num_int = random_int(low_int,   ↙
        high_int,exclude);
551                             Ark_no = procreation[num_int];
552                             primary_parents.push_back       ↙
        (Ark_no);
553                             //cout<<Ark[Ark_no].get_fcall   ↙
        ()<<" has been randomly selection for primary parent   ↙
        with "<<Ark[Ark_no].get_fitness()<<" (zero) fitness."< ↙
        <endl;
554                         }
555                         //Repeat for secondary parents
556                         all_fitness = 0;
557                         for(int j=0;j<procreation.size();j ↙
        ++){
558                             Ark_no = procreation[j];
559                             if((!any(secondary_parents,     ↙
        Ark_no))&&Ark_no!=primary_parents.back()){ //Skips     ↙
        already chosen secodanry parents and the primary       ↙
        parent that was last chosen
560                                 all_fitness += Ark[Ark_no] ↙
        .get_fitness();
561                             }
562                         }
563                         if(all_fitness!=0){
564                             num_fit = random_float(.000001 ↙
        ,all_fitness);
```

```
565                         selection_q = 0;
566                         while(num_fit>0){
567                             Ark_no = procreation
    [selection_q];
568                             if((!any(secondary_parents
    ,Ark_no))&&Ark_no!=primary_parents.back()){ //Skips
    already chosen secodanry parents and the primary
    parent that was last chosen
569                                 num_fit -= Ark[Ark_no]
    .get_fitness();
570                             }
571                             selection_q++;
572                         }
573                         secondary_parents.push_back
    (Ark_no);
574                         //cout<<Ark[Ark_no].get_fcall
    ()<<" with fitness "<<Ark[Ark_no].get_fitness()<<" was
     selected as secondary parent for selection loop "<<i<
    <endl;
575                     }
576                     else{
577                         //cout<<"Zero fitness loop for
     secondary parent in selection loop "<<i<<endl;
578                         low_int = 0;
579                         high_int = procreation.size()-
    1;
580                         vector<int> exclude;
581                         //I want to exclude the
    primary parent that was just chosen
582                         exclude.push_back(num_int);
583                         for(int i=0;i<procreation.size
    ();i++){
584                             if(any(secondary_parents,
    procreation[i])){
585                                 exclude.push_back(i);
586                             }
587                         }
588                         num_int = random_int(low_int,
    high_int,exclude);
589                         Ark_no = procreation[num_int];
590                         secondary_parents.push_back
    (Ark_no);
591                         //cout<<Ark[Ark_no].get_fcall
    ()<<" has been randomly selection for secondary parent
     with "<<Ark[Ark_no].get_fitness()<<" (zero) fitness."
    <<endl;
592                     }

593                 }
594                 //Check to make sure an equal number
    of primary and seconday parents were chosen
595                 if(primary_parents.size()!=
```

```
        secondary_parents.size()){
596                         cout<<"An equal number of primary ↙
        and seconday parents were chosen"<<endl;
597                         return 0;
598                     }
599                     //Place primary parent, secondary ↙
        parent, and mutation method into mutation info
600                     for(int j=0;j<secondary_parents.size() ↙
        ;j++){
601                         junk_ints.push_back ↙
        (primary_parents[j]);
602                         junk_ints.push_back ↙
        (secondary_parents[j]);
603                         junk_ints.push_back(i);
604                         mutation_info.push_back(junk_ints) ↙
        ;
605                         newly_made++;
606                         junk_ints.clear();
607                     }
608                     primary_parents.clear();
609                     secondary_parents.clear();
610                 }
611             }
612         }
613         //------------------------------------------------------ ↙
        -----------------------
614
615         //------------------------------------------------------ ↙
        ---------------------------------
616
617         MPI::COMM_WORLD.Bcast(&newly_made,1,MPI::INT,hub);
618
619         //----------------------Sends genomes to be ↙
        mutated---------------------------------
620
621         for(int i=0;i<newly_made;i++){
622             dest = mutationID[i%(mutationID.size())]; // ↙
        See page 20 Vol. 2 for logic
623             if(processID == hub){
624                 Ark_no = mutation_info[i][0]; ↙

625                 genome_size = Ark[Ark_no]. ↙
        get_genome_length();
626                 data_pack2[0] = Ark_no;
627                 data_pack2[1] = mutation_info[i][2];
628                 data_pack2[2] = genome_size;
629                 for(int j=0;j<genome_size;j++){
630                     temp_genome.push_back(Ark[Ark_no]. ↙
        get_genome(j));
631                 }
632                 MPI::COMM_WORLD.Send(&data_pack2,3,MPI:: ↙
        INT,dest,tag);
```

```
633                     MPI::COMM_WORLD.Send(&temp_genome[0],        ↙
        genome_size,MPI::INT,dest,tag);
634                     temp_genome.clear(); //Empties for next      ↙
        time
635                     //This sends another genome selected for a ↙
         mutation
636                     Ark_no = mutation_info[i][1];
637                     genome_size = Ark[Ark_no].                   ↙
        get_genome_length();
638                     data_pack[0] = Ark_no;
639                     data_pack[1] = genome_size;
640                     for(int j=0;j<genome_size;j++){
641                         temp_genome.push_back(Ark[Ark_no].       ↙
        get_genome(j));
642                     }
643                     MPI::COMM_WORLD.Send(&data_pack,2,MPI::INT ↙
        ,dest,tag);
644                     MPI::COMM_WORLD.Send(&temp_genome[0],        ↙
        genome_size,MPI::INT,dest,tag);
645                     temp_genome.clear(); //Empties for next      ↙
        time
646                 }
647                 if(processID == dest){
648                     MPI::COMM_WORLD.Recv(&data_pack2[0],3,MPI: ↙
        :INT,hub,tag);
649                     Ark_no = data_pack2[0];
650                     mutation_method = data_pack2[1];
651                     genome_size = data_pack2[2];
652                     temp_genome.resize(genome_size);
653                     MPI::COMM_WORLD.Recv(&temp_genome[0],        ↙
        genome_size,MPI::INT,hub,tag);
654                     MPI::COMM_WORLD.Recv(&data_pack[0],2,MPI:: ↙
        INT,hub,tag);
655                     Ark_no2 = data_pack[0];
656                     genome_size = data_pack[1];
657                     temp_genome2.resize(genome_size);
658                     MPI::COMM_WORLD.Recv(&temp_genome2[0],       ↙
        genome_size,MPI::INT,hub,tag);
659                     mu_point_mutation = mu_ratio[0];
660                     mu_conjugation = mu_ratio[1];
661                     mu_recopy = mu_ratio[2];
662                     mu_deletion = mu_ratio[3];
663                     mu_translocation = mu_ratio[4];
664                     mutator(temp_genome,temp_genome2,mu,         ↙
        mu_point_mutation,mu_conjugation,mu_recopy,mu_deletion ↙
        ,mu_translocation);
665                     junk_ints.push_back(Ark_no);
666                     junk_ints.push_back(Ark_no2);
667                     junk_ints.push_back(mutation_method);
668                     mutation_info.push_back(junk_ints);          ↙

669                     mutation_Ark.push_back(temp_genome);
```

```
670                     junk_ints.clear();
671                     temp_genome.clear();
672                     temp_genome2.clear();
673                 }
674             }
675 
676             //Satallites send the new genomes back to the hub
677             if(any(mutationID,processID)){
678                 back_size = mutation_Ark.size();
679                 MPI::COMM_WORLD.Send(&back_size,1,MPI::INT,hub ↙
        ,tag);
680                 for(int j=0;j<back_size;j++){
681                     data_pack2[0] = mutation_info[j][0];
682                     data_pack2[1] = mutation_info[j][1];
683                     data_pack2[2] = mutation_info[j][2];
684                     MPI::COMM_WORLD.Send(&data_pack2,3,MPI::   ↙
        INT,hub,tag);
685                     genome_size = mutation_Ark[j].size();
686                     MPI::COMM_WORLD.Send(&genome_size,1,MPI::  ↙
        INT,hub,tag);
687                     MPI::COMM_WORLD.Send(&mutation_Ark[j][0],  ↙
        genome_size,MPI::INT,hub,tag);
688                 }
689                 mutation_info.clear();
690                 mutation_Ark.clear();
691             }
692 
693             if(processID == hub){
694                 //Collects and places new individuals into the ↙
         Ark
695                 mutation_info.clear();
696                 mutation_Ark.clear();
697                 for(int i=0;i<mutationID.size();i++){
698                     src = mutationID[i];
699                     MPI::COMM_WORLD.Recv(&back_size,1,MPI::INT ↙
        ,src,tag);
700                     for(int j=0;j<back_size;j++){
701                         MPI::COMM_WORLD.Recv(&data_pack2[0],3, ↙
        MPI::INT,src,tag);
702                         Ark_no = data_pack2[0];
703                         Ark_no2 = data_pack2[1];
704                         mutation_method = data_pack2[2];
705                         junk_ints.push_back(Ark_no);
706                         junk_ints.push_back(Ark_no2);
707                         junk_ints.push_back(mutation_method);
708                         mutation_info.push_back(junk_ints);    ↙

709                         junk_ints.clear();
710                         MPI::COMM_WORLD.Recv(&genome_size,1,   ↙
        MPI::INT,src,tag);
711                         temp_genome.resize(genome_size);
712                         MPI::COMM_WORLD.Recv(&temp_genome[0],  ↙
```

```
genome_size,MPI::INT,src,tag);
713                         mutation_Ark.push_back(temp_genome);
714                         temp_genome.clear();
715                     }
716                 }
717                 for(int i=0;i<mutation_Ark.size();i++){
718                     Ark_no = mutation_info[i][0];
719                     Ark_no2 = mutation_info[i][1];
720                     generate_offspring(Ark,mutation_Ark[i],    ↙
    Ark_no,Ark_no2,(generation+1));
721                     Ark_Load(Ark[Ark.size()-1]);
722                     unmade.push_back(Ark.size()-1);
723                     //cout<<Ark[Ark.size()-1].get_fcall()<<"   ↙
    is unmade."<<endl;
724                     alive.push_back(Ark.size()-1);
725                     //cout<<Ark[Ark.size()-1].get_fcall()<<"   ↙
    is alive (2)."<<endl;
726                 }
727                 mutation_info.clear();
728                 mutation_Ark.clear();
729             }
730
731     }   //----------End of generation loop---------------- ↙
    ----
732
733     //  -------  This echoes the Final results
734     if(processID==hub){
735         cout<<"-----------FINAL RESULTS-------------"<     ↙
    <endl;
736         time (&end);
737         dif_t = int(difftime(end,start));
738         int hr,min,sec;
739         hr = int(dif_t/3600);
740         min = int((dif_t%3600)/60);
741         sec = dif_t%60;
742         cout<<"Evolution took "<<hr<<" hours, "<<min<<"    ↙
    minutes and "<<sec<<" seconds."<<endl;
743
744         //Echo back certain results for debugging
745         /*
746         for(int i=0;i<Ark.size();i++)
747         cout<<Ark[i].get_fcall()<<" "<<Ark[i].get_fitness  ↙
    ()<<endl;
748         for(int i=0;i<unmade.size();i++)
749         cout<<unmade[i]<<" ";
750         cout<<endl;
751         for(int i=0;i<alive.size();i++)
752         cout<<alive[i]<<" ";
753         cout<<endl;
754         */
755
756         Record_Gen(Ark,still_alive,unmade,generation); //  ↙
```

```
    Saves final state
757     }
758     MPI::Finalize();
759     return 0;
760 }
761
```

```
1  using namespace std;
2
3  //------------------Classes for Neural Nets---------------------------------------
   ----
4  class connection
5  {private:
6  float weight;
7  int node_from;
8  int node_to;
9  float Heb_rate;
10 float random_rate;
11 public:
12     connection()
13     {}
14     void operator = (const connection& right){
15         if (this != &right){
16             weight = right.weight;
17             node_from = right.node_from;
18             node_to = right.node_to;
19             Heb_rate = right.Heb_rate;
20             random_rate = right.random_rate;
21         }
22     }
23
24     void make_connection_private(int n_from,int n_to,float
        w,float h, float r)//Used with make_connection
       function
25     {
26     weight = w;
27     node_from = n_from;
28     node_to = n_to;
29     Heb_rate = h;
30     random_rate = r;
31     }
32     float get_weight(){
33         return(weight);
34     }
35     void set_weight_private(float x){
36         weight = x;
37     }
38     int get_node_from(){
39         return(node_from);
40     }
41     int get_node_to(){
42         return(node_to);
43     }
44     float get_Hebbian_rate(){
45         return(Heb_rate);
46     }
47     float get_random_rate(){
48         return(random_rate);
49     }
```

```
50 };
51
52 class node
53 {private:
54 float bias;
55 float slope;
56 char layer; //Denote whether a node is an input (I),
       hidden (H), or an output(O)  Don't confuse with type 1
57 int type1; //Denotes the type of node.  Integer corralates
        to A - H
58 int type2; //Also denotes numerical order of the node
59 int type3;
60 int nodes_made; //Records the number of new nodes a node
       has made
61 float activation; //Tells us the activation level of a
       node
62 public:
63     node()
64     { }
65  //It works, but I get an warning evrytime it's compiled
66     void operator = (const node& right){
67         if (this != &right){
68             bias = right.bias;
69             slope = right.slope;
70             layer = right.layer;
71             type1 = right.type1;
72             type2 = right.type2;
73             type3 = right.type3;
74             nodes_made = right.nodes_made;
75             activation = right.activation;
76         }
77     }
78
79     void make_node_private(char l,int t1,int t2,int t3,
       float s,float b){//Used with make_node function
80         layer = l;
81         type1 = t1;
82         bias = b;
83         slope = s;
84         type2 = t2;
85         type3 = t3;
86         nodes_made = 0;
87         activation = 0.0;
88     }
89     char get_layer(){
90         return(layer);
91     }
92     int get_nodes_made(){
93         return(nodes_made);
94     }
95     void inc_nodes_made(){
96         nodes_made++;
```

```
97      }
98      float get_bias(){
99          return(bias);
100     }
101     float get_slope(){
102         return(slope);
103     }
104     int get_type1(){
105         return(type1);
106     }
107     int get_type2(){
108         return(type2);
109     }
110     int get_type3(){
111         return(type3);
112     }
113     float get_activation_private(){
114         return(activation);
115     }
116     void set_activation_private(float x){
117         activation = x;
118     }
119 };
120
121 class neural_net
122 {private:
123 vector<connection> connections;
124 vector<node> nodes;
125 float reinforcement;
126 public:
127     neural_net()
128     { }
129     //It works, but I get an warning everytime it's
        compiled
130     void operator= (const neural_net& right){
131         if (this != &right){
132             connections = right.connections;
133             nodes = right.nodes;
134         }
135     }
136     void clear_ANN(){
137         connections.clear();
138         nodes.clear();
139     }
140     int get_ANN_size(){
141         return(nodes.size());
142     }
143     node get_node(int n){
144         return(nodes[n]);
145     }
146     float get_activation(int n){
147         return(nodes[n].get_activation_private());
```

```
148     }
149     void set_activation(int n, float x){
150         nodes[n].set_activation_private(x);
151     }
152     void make_node(int p_node,char l,int t1,float s,float ↙
        b){
153         int t2,t3;
154         int counter = 0;
155         node new_node;
156         t2 = nodes[p_node].get_nodes_made();
157         for(int i=0;i<nodes.size();i++){
158             if((nodes[i].get_type1()== t1)&&(nodes[i].   ↙
        get_type2()== t2)){
159                 counter++;
160             }
161         }
162         t3 = counter%100;
163         new_node.make_node_private(l,t1,t2,t3,s,b);
164         nodes.push_back(new_node);
165         nodes[p_node].inc_nodes_made();
166     }
167     void make_input(int t1){
168         int t2,t3;
169         int counter = 0;
170         node new_node;
171         t2 = 0;
172         for(int i=0;i<nodes.size();i++){
173             if(nodes[i].get_type1()== t1 ){
174                 counter++;
175             }
176         }
177         t3 = counter%100;
178         new_node.make_node_private('I',t1,t2,t3,0,0);   ↙

179         nodes.push_back(new_node);
180     }
181     void make_output(int p_node,int t1,float s,float b){
182         int t2,t3;
183         int counter = 0;
184         node new_node;
185         t2 = nodes[p_node].get_nodes_made();
186         for(int i=0;i<nodes.size();i++){
187             if((nodes[i].get_type1()== t1)&&(nodes[i].   ↙
        get_type2()== t2)){
188                 counter++;
189             }
190         }
191         t3 = counter%100;
192         new_node.make_node_private('O',t1,t2,t3,s,b);   ↙

193         nodes.push_back(new_node);
194         nodes[p_node].inc_nodes_made();
```

```
195     }
196     int get_total_connections(){
197         return(connections.size());
198     }
199     connection get_connection(int n){
200         return(connections[n]);
201     }
202     void set_weight(int n,float x){
203         connections[n].set_weight_private(x);
204     }
205     void make_connection(int n_from,int n_to,float w,float ↙
         h,float r){
206         connection new_connection;
207         new_connection.make_connection_private(n_from,n_to ↙
        ,w,h,r);
208         connections.push_back(new_connection);
209     }
210     int get_total_inputs(){
211         int count = 0;
212         node temp_node;
213         for(int i=0;i<nodes.size();i++){
214             temp_node = nodes[i];
215             if(temp_node.get_layer()=='I'){
216                 count++;
217             }
218         }
219         return(count);
220     }
221     int get_total_outputs(){
222         int count = 0;
223         node temp_node;
224         for(int i=0;i<nodes.size();i++){
225             temp_node = nodes[i];
226             if(temp_node.get_layer()=='O'){
227                 count++;
228             }
229         }
230         return(count);
231     }
232     float get_reinforcement(){
233         return(reinforcement);
234     }
235     void set_reinforcement(float x){
236         reinforcement = x;
237     }
238     int get_inputs_to(int n){
239         int ins = 0;
240         for(int i=0;i<connections.size();i++){
241             if(connections[i].get_node_to()==n){
242                 ins++;
243             }
244         }
```

```
245         return(ins);
246     }
247     int get_outputs_from(int n){
248         int outs = 0;
249         for(int i=0;i<connections.size();i++){
250             if(connections[i].get_node_from()==n){
251                 outs++;
252             }
253         }
254         return(outs);
255     }
256     float sum_inputs_to(int n){
257         float ins = 0;
258         for(int i=0;i<connections.size();i++){
259             if(connections[i].get_node_to()==n){
260                 ins = ins + connections[i].get_weight();
261             }
262         }
263         return(ins);
264     }
265     float sum_outputs_from(int n){
266         float outs = 0;
267         for(int i=0;i<connections.size();i++){
268             if(connections[i].get_node_from()==n){
269                 outs = outs + connections[i].get_weight();
270             }
271         }
272         return(outs);
273     }
274     float get_connection_weight(int i,int j){
275         //float w = 0;
276         float w = -100; //Changed to this so it will         ↙
    return a non-working answer if there is no connection
277         for(int i=0;i<connections.size();i++){
278             if((connections[i].get_node_from()==i)&&         ↙
    (connections[i].get_node_to()==j)){
279                 w = connections[i].get_weight();
280             }
281         }
282         return(w);
283     }
284     void print_net(){
285         cout<<"Node:\tLayer\tType:\tBias:\tSlope:\n";
286         for(int i=0;i<nodes.size();i++){
287             cout<<i<<"\t"<<nodes[i].get_layer()<<"\t"<     ↙
    <nodes[i].get_type1()<<nodes[i].get_type2();
288             cout<<"\t"<<nodes[i].get_bias()<<"\t"<<nodes   ↙
    [i].get_slope()<<endl;
289         }
290         cout<<"Conn:\tFrom\tTo:\tWeight:\tHeb:\tRand:\n";
291         for(int i=0;i<connections.size();i++){
292             cout<<i<<"\t"<<connections[i].get_node_from()< ↙
```

```
<"\t"<<connections[i].get_node_to();
293             cout<<"\t"<<connections[i].get_weight()<<"\t"<
<connections[i].get_Hebbian_rate();
294             cout<<"\t"<<connections[i].get_random_rate()<
<endl;
295         }
296     }
297     void write_net(string& filename){
298         ofstream ANNfile(&filename[0]);
299         ANNfile<<nodes.size()<<endl;
300         for(int i=0;i<nodes.size();i++){
301             ANNfile<<i<<" "<<nodes[i].get_layer()<<" "<
<nodes[i].get_type1()<<" "<<nodes[i].get_type2()<<" "<
<nodes[i].get_type3();
302             ANNfile<<" "<<nodes[i].get_bias()<<" "<<nodes
[i].get_slope()<<" "<<nodes[i].get_nodes_made()<<endl;
303         }
304         ANNfile<<connections.size()<<endl;
305         for(int i=0;i<connections.size();i++){
306             ANNfile<<i<<"\t"<<connections[i].get_node_from
()<<"\t"<<connections[i].get_node_to();
307             ANNfile<<"\t"<<connections[i].get_weight()<<"\
t"<<connections[i].get_Hebbian_rate();
308             ANNfile<<"\t"<<connections[i].get_random_rate
()<<endl;
309         }
310     }
311     void read_net(string& filename){
312         ifstream ANNfile(&filename[0]);
313         int number_of_nodes,nodes_made,
number_of_connections;
314         int junk_int,type1,type2,type3,node_from,node_to;
315         char layer;
316         float bias,slope,weight,Heb,rand;
317         node temp_node;
318         connection temp_conn;
319         ANNfile>>number_of_nodes;
320         for(int i=0;i<number_of_nodes;i++){
321             ANNfile>>junk_int;
322             ANNfile>>layer;
323             ANNfile>>type1;
324             ANNfile>>type2;
325             ANNfile>>type3;
326             ANNfile>>bias;
327             ANNfile>>slope;
328             ANNfile>>nodes_made;
329             temp_node.make_node_private(layer,type1,type2,
type3,slope,bias);
330             nodes.push_back(temp_node);
331             for(int j=0;j<nodes_made;j++){
332                 nodes[i].inc_nodes_made();
333             }
```

```
334             }
335             ANNfile>>number_of_connections;
336             for(int i=0;i<number_of_connections;i++){
337                 ANNfile>>junk_int;
338                 ANNfile>>node_from;
339                 ANNfile>>node_to;
340                 ANNfile>>weight;
341                 ANNfile>>Heb;
342                 ANNfile>>rand;
343                 temp_conn.make_connection_private(node_from,    ↙
        node_to,weight,Heb,rand);
344                 connections.push_back(temp_conn);
345             }
346         }
347 };
348 //---------------------------End of Neural Net Classes---- ↙
        ----------------------------
349
350 //--------------Functions for making and using ANN          ↙
        Matricies------------------
351
352 bool make_node_check(neural_net ANN,int n,int max_outs){
353     node temp_node = ANN.get_node(n);
354     int outs = ANN.get_total_outputs();
355     bool verdict = false;
356     if((temp_node.get_nodes_made()<7) &&(outs < max_outs)) ↙
        {
357         verdict = true;
358     }
359     return(verdict);
360 }
361
362 bool make_connection_check(neural_net ANN,int n_from,int   ↙
        n_to,int max_conns){
363     bool verdict = true;
364     connection temp_conn;
365     node from_node = ANN.get_node(n_from);
366     node to_node = ANN.get_node(n_to);
367     int from_counter = 0;
368     int to_counter = 0;
369     for(int i=0;i<ANN.get_total_connections();i++){
370         temp_conn = ANN.get_connection(i);
371         if(temp_conn.get_node_from() == n_from){
372             from_counter++;
373         }
374         if(temp_conn.get_node_to() == n_to){
375             to_counter++;
376         }
377         if((temp_conn.get_node_from() == n_from)&&             ↙
        (temp_conn.get_node_to() == n_to)){
378             verdict = false;
379         }
```

```
380     }
381     if(n_to <= n_from){
382         verdict = false;
383     }
384     if(from_node.get_layer() == 'O'){
385         verdict = false;
386     }
387     if(to_node.get_layer() == 'I'){
388         verdict = false;
389     }
390     if((from_counter>=max_conns)|(to_counter>=max_conns)){
391         verdict = false;
392     }
393     return(verdict);
394 }
395
396 //---------------------------------------------------------- ↙
        ----------------------
397
```

```
1 // This is the library that contains functions necessary
      for manipulating individuals thoughout evolution
2 // It should follow chimera_lib.h and node_lib.h when
      being called
3
4
5 using namespace std;
6
7 //--------------------------------Individual Class-------
      -----------------
8 class individual
9 {private:
10 int genome_length;
11 vector<int> genome; //The actual genetic string
12 vector<int> ruleset; //Rules within the genome that make
      the ANN
13 string fcall; //Records the name of the .cpp file that
      has the subject's protiens
14 float fitness; //The fitness of an individual. Can become
       an array
15 int genesis[3]; //An array to tell ["gen made" "Parent 1"
       "Parent 2"]
16 char method; /*Tells how the individual was created
17              P - Point Mutation
18              D - Duplication/Deletion of codon(s)
19              C - Crossover
20              R - Randomly Generated
21              I - Intellegently Designed
22              S - Say Again */
23 int death; //Tells the last generation in which an
      individual appeared, thus a -1 means it is still
      alive
24 neural_net ANN; //The individual's neural net
25 vector< vector<float> > ANN_weights; //The individual's
      neural net weight in matrix form
26 vector<float> ANN_biases; //The individual's neural net
      biases in vector form
27 vector<float> ANN_slopes; //The individual's neural net
      slopes in vector form
28 public:
29     individual(){} //Default Constructor
30
31     individual(int sega[3],vector<int> genes){ //
      Constructor - given creation info and genome
32         genome_length = genes.size();
33         genome = genes;
34         fcall = "Subject-1.cpp";
35         fitness = -1;
36         genesis[0] = sega[0];
37         genesis[1] = sega[1];
38         genesis[2] = sega[2];
39         method = 'I';
```

```
40          death = -1;
41      }
42      void operator= (const individual& right){
43          if (this != &right){
44              genome = right.genome;
45              ruleset = right.ruleset;
46              fcall = right.fcall;
47              fitness = right.fitness;
48              genesis[0] = right.genesis[0];
49              genesis[1] = right.genesis[1];
50              genesis[2] = right.genesis[2];
51              method = right.method;
52              death = right.death;
53              ANN = right.ANN;
54              ANN_weights.resize(0); ANN_weights.assign      ↙
        (right.ANN_weights.begin(),right.ANN_weights.end());
55              ANN_biases.resize(0); ANN_biases.assign(right ↙
        .ANN_biases.begin(),right.ANN_biases.end());
56              ANN_slopes.resize(0); ANN_slopes.assign(right ↙
        .ANN_slopes.begin(),right.ANN_slopes.end());         ↙

57          }
58      }
59
60      int get_genome(int n){
61          return(genome[n]);
62      }
63      int get_genome_length(){
64          return(genome_length);
65      }
66      int get_nucleotide(int n){
67          return(genome[n]);
68      }
69      void save_rule(int rule){
70          ruleset.push_back(rule);
71      }
72      int get_rule(int n){
73          return(ruleset[n]);
74      }
75      int get_rules_length(){
76          return(ruleset.size());
77      }
78      string get_fcall(){
79          return(fcall);
80      }
81      float get_fitness(){
82          return(fitness);
83      }
84      void make_fitness(float x){
85          fitness = x;
86      }
87      void inc_fitness(float x){
```

```
88          fitness = fitness + x;
89      }
90      void mult_fitness(float x){
91          fitness = fitness*x;
92      }
93      void dec_fitness(float x){
94          fitness = fitness - x;
95      }
96      int get_genesis(int n){
97          return(genesis[n]);
98      }
99      char get_method(){
100         return(method);
101     }
102     int get_death(){
103         return(death);
104     }
105     void kill(int gen){
106         death = gen;
107     }
108     bool alive(){
109         if(death == -1)
110             return(true);
111         else
112             return(false);
113     }
114     void generate_random_private(int l,int gen,int sub){  ↙
         //Will generate a random genome of length l
115         int lowest=1, highest=100;
116         int range=(highest-lowest)+1;
117         int temp;
118         for(int i=0; i<l; i++){
119             temp = lowest+int(range*(rand()/(RAND_MAX + 1 ↙
        .0)));
120             genome.push_back(temp);
121         }
122         genome_length = l;
123         string num = int2string(sub);
124         fcall = "Subject" + num + ".cpp";
125         fitness = -1;
126         genesis[0] = gen;
127         genesis[1] = 0;
128         genesis[2] = 0;
129         method = 'R';
130         death = -1;
131     }
132     void generate_designed_private(int arr[],int gen,int  ↙
        sub){  //Will generate an individual with the given  ↙
        genome
133         int find_array_length(int[]);
134         int l = find_array_length(arr);
135         for(int i=0; i<l; i++){
```

```
136             genome.push_back(arr[i]);
137         }
138         genome_length = l;
139         string num = int2string(sub);
140         fcall = "Subject" + num + ".cpp";
141         fitness = -1;
142         genesis[0] = gen;
143         genesis[1] = 0;
144         genesis[2] = 0;
145         method = 'I';
146         death = -1;
147     }
148     void generate_designed_private(vector<int> arr,int ↙
        gen,int sub){  //Will generate an individual with the ↙
         given genome
149         int l = arr.size();
150         for(int i=0; i<l; i++){
151             genome.push_back(arr[i]);
152         }
153         genome_length = l;
154         string num = int2string(sub);
155         fcall = "Subject" + num + ".cpp";
156         fitness = -1;
157         genesis[0] = gen;
158         genesis[1] = 0;
159         genesis[2] = 0;
160         method = 'I';
161         death = -1;
162     }
163 
164     void generate_reduced_private(vector<int> arr,int gen ↙
        ,int sub,int parent){  //Will generate an individual  ↙
        with the given genome
165         int l = arr.size();
166         for(int i=0; i<l; i++){
167             if((arr[i]>=1)&&(arr[i]<=100)){
168                 genome.push_back(arr[i]);
169             }
170             else{
171                 int temp_int;
172                 temp_int = random_int(1,100);
173                 genome.push_back(temp_int);
174                 cout<<"The invalid nucleotide "<<arr[i]<  ↙
        <" was replaced with "<<temp_int<<endl;
175             }
176         }
177         genome_length = l;
178         string num = int2string(sub);
179         fcall = "Subject" + num + ".cpp";
180         fitness = -1;
181         genesis[0] = gen;
182         genesis[1] = parent;
```

```
183         genesis[2] = parent;
184         method = 'S';
185         death = -1;
186     }
187 
188     void generate_offspring_private(vector<int> arr,int 
        gen,int sub,int indy1,int indy2){   //Will generate an 
         individual with the given genome
189         int l = arr.size();
190         for(int i=0; i<l; i++){
191             genome.push_back(arr[i]);
192         }
193         genome_length = l;
194         string num = int2string(sub);
195         fcall = "Subject" + num + ".cpp";
196         fitness = -1;
197         genesis[0] = gen;
198         genesis[1] = indy1;
199         genesis[2] = indy2;
200         method = 'O';
201         death = -1;
202     }
203     void Say_Again_private(int sega[],char meth,vector
        <int> arr){
204         int l = arr.size();
205         for(int i=0; i<l; i++)
206             genome.push_back(arr[i]);
207         genome_length = genome.size();
208         fcall = "Subject-1.cpp";
209         fitness = -1;
210         genesis[0] = sega[0];
211         genesis[1] = sega[1];
212         genesis[2] = sega[2];
213         method = meth;
214         death = -1;
215     }
216 
217     void show_genome(){ //The following prints out the 
        genomes
218         for(int i=0; i<genome.size(); i++)
219             cout << genome[i] << "  ";
220         cout << endl;
221     }
222 
223     void show_rules(){ //The following prints frames as 
        they are used
224         for(int i=0; i<ruleset.size(); i++){
225             if(ruleset[i]!=-1){
226                 cout<<ruleset[i]<<'  ';
227                 for(int j=0;j<6;j++)
228                     cout<<genome[ruleset[i]+j]<<'  ';
229                 cout<<endl;
```

```
230                 }
231                 else
232                     cout <<endl;
233             }
234         }
235 
236         void reduce_rules(vector< vector <int> >&
        reduced_protein_table){
237             //This will show which frame numbers made the
        individual
238             reduced_protein_table.clear();
239             if(ruleset.size()==0){
240                 return;
241             }
242             vector< vector<int> >used_proteins_table;
243             vector< vector<int> >sorted_used_proteins_table;
244             vector <int> test_protein;
245             int lowest_rule = 100000;
246             int lowest_rule_size = 100000;
247             int lowest_rule_index = -1;
248             vector <int> used_indexes;
249             for(int i=0;i<ruleset.size();i++){
250                 if(ruleset[i]!= -1){
251                     test_protein.push_back(ruleset[i]);
252                 }
253                 else{
254                     if(!any(test_protein,
        used_proteins_table)){
255                         used_proteins_table.push_back
        (test_protein);
256                     }
257                     test_protein.clear();
258                 }
259             }
260             //print_matrix(used_proteins_table);
261             while(sorted_used_proteins_table.size() <
        used_proteins_table.size()){
262                 for(int i=0;i<used_proteins_table.size();i++)
        {
263                     if((used_proteins_table[i][0]
        <lowest_rule)&&(!any(i,used_indexes))){
264                         lowest_rule = used_proteins_table[i]
        [0];
265                         lowest_rule_size =
        used_proteins_table[i].size();
266                         lowest_rule_index = i;
267                     }
268                     else if((used_proteins_table[i][0]==
        lowest_rule)&&(used_proteins_table[i].size()
        <lowest_rule_size)&&(!any(i,used_indexes))){
269                         lowest_rule = used_proteins_table[i]
        [0];
```

```
270                         lowest_rule_size = ↙
        used_proteins_table[i].size();
271                         lowest_rule_index = i;
272                     }
273                     else if((used_proteins_table[i][0]== ↙
        lowest_rule)&&(used_proteins_table[i].size()== ↙
        lowest_rule_size)&&(!any(i,used_indexes))){
274                         for(int j=i;j<used_proteins_table[i]. ↙
        size();j++){
275                             if(used_proteins_table[i][j] ↙
        <used_proteins_table[lowest_rule_index][j]){
276                                 lowest_rule = ↙
        used_proteins_table[i][0];
277                                 lowest_rule_size = ↙
        used_proteins_table[i].size();
278                                 lowest_rule_index = i;
279                             }
280                         }
281                     }
282                 }
283                 sorted_used_proteins_table.push_back ↙
        (used_proteins_table[lowest_rule_index]);
284                 used_indexes.push_back(lowest_rule_index);
285                 lowest_rule = 100000;
286                 lowest_rule_size = 100000;
287             }
288             //print_matrix(sorted_used_proteins_table);
289             for(int i=0;i<(sorted_used_proteins_table.size()- ↙
        1);i++){
290                 if(sorted_used_proteins_table[i].size()== ↙
        sorted_used_proteins_table[i+1].size()){
291                     reduced_protein_table.push_back ↙
        (sorted_used_proteins_table[i]);
292                 }
293                 else{
294                     for(int j=0;j<sorted_used_proteins_table ↙
        [i].size();j++){
295                         if(sorted_used_proteins_table[i][j]!= ↙
        sorted_used_proteins_table[i+1][j]){
296                             reduced_protein_table.push_back ↙
        (sorted_used_proteins_table[i]);
297                             break;
298                         }
299                     }
300                 }
301             }
302             reduced_protein_table.push_back ↙
        (sorted_used_proteins_table[ ↙
        (sorted_used_proteins_table.size()-1)]);
303             //print_matrix(reduced_protein_table);
304         }
305
```

```
306     void reduce_rules(){
307         vector< vector <int> > reduced_protein_table;
308         //This will show which frame numbers made the ↙
        individual
309         if(ruleset.size()==0){
310             return;
311         }
312         vector< vector<int> >used_proteins_table;
313         vector< vector<int> >sorted_used_proteins_table;
314         vector <int> test_protein;
315         int lowest_rule = 100000;
316         int lowest_rule_size = 100000;
317         int lowest_rule_index = -1;
318         vector <int> used_indexes;
319         for(int i=0;i<ruleset.size();i++){
320             if(ruleset[i]!= -1){
321                 test_protein.push_back(ruleset[i]);
322             }
323             else{
324                 if(!any(test_protein, ↙
        used_proteins_table)){
325                     used_proteins_table.push_back ↙
        (test_protein);
326                 }
327                 test_protein.clear();
328             }
329         }
330         //print_matrix(used_proteins_table);
331         while(sorted_used_proteins_table.size() < ↙
        used_proteins_table.size()){
332             for(int i=0;i<used_proteins_table.size();i++) ↙
        {
333                 if((used_proteins_table[i][0] ↙
        <lowest_rule)&&(!any(i,used_indexes))){
334                     lowest_rule = used_proteins_table[i] ↙
        [0];
335                     lowest_rule_size = ↙
        used_proteins_table[i].size();
336                     lowest_rule_index = i;
337                 }
338                 else if((used_proteins_table[i][0]== ↙
        lowest_rule)&&(used_proteins_table[i].size() ↙
        <lowest_rule_size)&&(!any(i,used_indexes))){
339                     lowest_rule = used_proteins_table[i] ↙
        [0];
340                     lowest_rule_size = ↙
        used_proteins_table[i].size();
341                     lowest_rule_index = i;
342                 }
343                 else if((used_proteins_table[i][0]== ↙
        lowest_rule)&&(used_proteins_table[i].size()== ↙
        lowest_rule_size)&&(!any(i,used_indexes))){
```

```
344                     for(int j=i;j<used_proteins_table[i]. ↙
    size();j++){
345                         if(used_proteins_table[i][j]       ↙
    <used_proteins_table[lowest_rule_index][j]){
346                             lowest_rule =                  ↙
    used_proteins_table[i][0];
347                             lowest_rule_size =             ↙
    used_proteins_table[i].size();
348                             lowest_rule_index = i;
349                         }
350                     }
351                 }
352             }
353             sorted_used_proteins_table.push_back           ↙
    (used_proteins_table[lowest_rule_index]);
354             used_indexes.push_back(lowest_rule_index);
355             lowest_rule = 100000;
356             lowest_rule_size = 100000;
357         }
358         //print_matrix(sorted_used_proteins_table);
359         for(int i=0;i<(sorted_used_proteins_table.size()- ↙
    1);i++){
360             if(sorted_used_proteins_table[i].size()==      ↙
    sorted_used_proteins_table[i+1].size()){
361                 reduced_protein_table.push_back            ↙
    (sorted_used_proteins_table[i]);
362             }
363             else{
364                 for(int j=0;j<sorted_used_proteins_table   ↙
    [i].size();j++){
365                     if(sorted_used_proteins_table[i][j]!= ↙
    sorted_used_proteins_table[i+1][j]){
366                         reduced_protein_table.push_back    ↙
    (sorted_used_proteins_table[i]);
367                         break;
368                     }
369                 }
370             }
371         }
372         reduced_protein_table.push_back                    ↙
    (sorted_used_proteins_table[                               ↙
    (sorted_used_proteins_table.size()-1)]);
373         print_matrix(reduced_protein_table);
374     }
375 
376     //----------------------------------------------------- ↙
    -------------------------------------------
377     //----------------------------------------ANN          ↙
    COMMANDS------------------------------------------
378     //----------------------------------------------------- ↙
    -------------------------------------------
379 
```

```
380     neural_net get_neural_net(){
381         return(ANN);
382     }
383 
384     void make_ANN(int rank_no){
385         ANN.clear_ANN();
386         string pcall = "/scratch/subject"+int2string ↙
        (rank_no)+".exe";
387         string pmake = "g++ -o " + pcall + " /scratch/"+ ↙
        fcall;
388         string ANNfilename = "/scratch/ANN"+int2string ↙
        (rank_no)+".dat";
389         string Rulecall = "/scratch/Rules"+int2string ↙
        (rank_no)+".dat";
390         char *syscall;
391         syscall = &pmake[0];
392         system(syscall);
393         syscall = &pcall[0];
394         system(syscall);
395         ANN.read_net(ANNfilename);
396         ANN_weights.resize(0);
397         ANN_biases.resize(0);
398         ANN_slopes.resize(0);
399         ruleset.resize(0);
400         vector<float> w_fill(ANN.get_ANN_size(),0);
401         node temp_node;
402         connection temp_conn;
403         float temp_slopes,temp_biases,temp_w;
404         int node_to,node_from;
405 
406         for(int i=0;i<ANN.get_ANN_size();i++){
407             ANN_weights.push_back(w_fill);
408             temp_node = ANN.get_node(i);
409             ANN_biases.push_back(temp_node.get_bias());
410             ANN_slopes.push_back(temp_node.get_slope());
411         }
412         for(int i=0;i<ANN.get_total_connections();i++){
413             temp_conn = ANN.get_connection(i);
414             node_to = temp_conn.get_node_to();
415             node_from = temp_conn.get_node_from();
416             temp_w = temp_conn.get_weight();
417             ANN_weights[node_from][node_to] = temp_w; ↙

418         }
419         ifstream infile2(&Rulecall[0]);
420         int temprule;
421         while(!infile2.eof()){
422             infile2 >> temprule;
423             ruleset.push_back(temprule);
424         }
425         ruleset.pop_back(); //For some reason, it always ↙
        saves an extra  -1
```

```
426     }
427 
428     void make_ANN_matrix(){
429         ANN_weights.resize(0);
430         ANN_biases.resize(0);
431         ANN_slopes.resize(0);
432         vector<float> w_fill(ANN.get_ANN_size(),0);
433         node temp_node;
434         connection temp_conn;
435         float temp_slopes,temp_biases,temp_w;
436         int node_to,node_from;
437 
438         for(int i=0;i<ANN.get_ANN_size();i++){
439             ANN_weights.push_back(w_fill);
440             temp_node = ANN.get_node(i);
441             ANN_biases.push_back(temp_node.get_bias());
442             ANN_slopes.push_back(temp_node.get_slope());
443         }
444         for(int i=0;i<ANN.get_total_connections();i++){
445             temp_conn = ANN.get_connection(i);
446             node_to = temp_conn.get_node_to();
447             node_from = temp_conn.get_node_from();
448             temp_w = temp_conn.get_weight();
449             ANN_weights[node_from][node_to] = temp_w;      ↙

450         }
451     }
452 
453     void show_ANN_matrix(){
454         int type1;
455         for(int i=0;i<ANN.get_ANN_size();i++){
456             node temp_node = ANN.get_node(i);
457             for(int j=0;j<ANN.get_ANN_size();j++){
458                 cout<<ANN_weights[i][j]<<"  \t";
459             }
460             cout<<"  \t"<<ANN_biases[i];
461             //cout<<"  \t"<<ANN_slopes[i];
462             type1 = temp_node.get_type1();
463             cout<<" \t";
464             if(type1 == 0)
465                 cout<<"A";
466             else if (type1 == 1)
467                 cout<<"B";
468             else if (type1 == 2)
469                 cout<<"C";
470             else if (type1 == 3)
471                 cout<<"D";
472             else if (type1 == 4)
473                 cout<<"E";
474             else if (type1 == 5)
475                 cout<<"F";
476             else if (type1 == 6)
```

```
477                 cout<<"G";
478             else if (type1 == 7)
479                 cout<<"H";
480             cout<<temp_node.get_type2()<<"-"<<temp_node.
    get_type3()<<endl;
481         }
482     }
483     void break_node_off(int n){
484         ANN_biases[n] = 1000;
485     }
486     void break_node_on(int n){
487         ANN_biases[n] = -1000;
488     }
489     void break_connection(int i, int j){
490         ANN_weights[i][j] = 0;
491     }
492     float get_ANN_weight(int i, int j){
493         return(ANN_weights[i][j]);
494     }
495     float get_ANN_bias(int i){
496         return(ANN_biases[i]);
497     }
498     float get_ANN_slope(int i){
499         return(ANN_slopes[i]);
500     }
501     void Matlab_ANN(){
502         //This puts the matrix weights, biases, and
    slopes into a Matlab script
503         //Rearanges the Matrix so inputs are first,
    outputs are last, and hidden nodes are in between
504         float Matlab_weights[ANN.get_ANN_size()][ANN.
    get_ANN_size()];
505         float Matlab_biases[ANN.get_ANN_size()];
506         float Matlab_slopes[ANN.get_ANN_size()];
507         vector< vector <int> > translation;  //Holds the
    old node number [0] and the new one [1] The [0] entry
     is just the index and isn't necessary, but it makes
    it easier to decipher
508         node temp_node;
509         int temp_int;
510         float temp_float;
511         vector< int > temp_vect;
512         for(int i=0;i<ANN.get_ANN_size();i++){
513             temp_node = ANN.get_node(i);
514             if(temp_node.get_layer()== 'I'){
515                 temp_int = translation.size();
516                 temp_vect.push_back(temp_int);
517                 temp_vect.push_back(i);
518                 translation.push_back(temp_vect);
519                 temp_vect.clear();
520             }
521         }
```

```
522         for(int i=0;i<ANN.get_ANN_size();i++){
523             temp_node = ANN.get_node(i);
524             if(temp_node.get_layer()== 'H'){
525                 temp_int = translation.size();
526                 temp_vect.push_back(temp_int);
527                 temp_vect.push_back(i);
528                 translation.push_back(temp_vect);
529                 temp_vect.clear();
530             }
531         }
532         for(int i=0;i<ANN.get_ANN_size();i++){
533             temp_node = ANN.get_node(i);
534             if(temp_node.get_layer()== 'O'){
535                 temp_int = translation.size();
536                 temp_vect.push_back(temp_int);
537                 temp_vect.push_back(i);
538                 translation.push_back(temp_vect);
539                 temp_vect.clear();
540             }
541         }
542         if(translation.size()!= ANN.get_ANN_size()){
543             cout<<"ERROR: The nodes were not recorded ↙
    properly"<<endl;
544         }
545         for(int i=0;i<translation.size();i++){
546             Matlab_biases[i] = ANN_biases[translation[i] ↙
    [1]];
547             Matlab_slopes[i] = ANN_slopes[translation[i] ↙
    [1]];
548             for(int j=0;j<translation.size();j++){
549                 Matlab_weights[i][j] = ANN_weights ↙
    [translation[i][1]][translation[j][1]];
550             }
551         }
552 
553         ofstream ANNfile("ANN.m");
554         ANNfile<<"W=[";
555         for(int i=0;i<translation.size();i++){
556             for(int j=0;j<translation.size();j++){
557                 ANNfile<<Matlab_weights[i][j]<<" ";
558             }
559             ANNfile<<";";
560         }
561         ANNfile<<"]\n";
562         ANNfile<<"B=[";
563         for(int i=0;i<translation.size();i++){
564             ANNfile<<Matlab_biases[i]<<"; ";
565         }
566         ANNfile<<"]\n";
567         ANNfile<<"S=[";
568         for(int i=0;i<translation.size();i++){
569             ANNfile<<Matlab_slopes[i]<<"; ";
```

```
570         }
571         ANNfile<<"]\n";
572     }
573 
574     void Matlab_ANN_growth(){
575         //This records the order and type of rules used ↙
        so the growth of the ANN can be seen
576 
577         vector<float> Matlab_rules;
578         int action_nucleotide,action_value_nucleotide, ↙
        action_type,nodes_made,outputs_made,max_outputs;
579         float action_value;
580         int make_connection[] = {1,20};
581         int do_nothing[] = {21,35};
582         int end_turn[] = {36,50};
583         int make_node[] = {51,100};
584         int make_nodeH[] = {86,100};
585         nodes_made = 0;
586         outputs_made = 0;
587         max_outputs = 0;
588         node temp_node;
589 
590         for(int i=0;i<ANN.get_ANN_size();i++){
591             temp_node = ANN.get_node(i);
592             if(temp_node.get_layer()== 'I'){
593                 nodes_made++;
594             }
595             else if(temp_node.get_layer()== 'O'){
596                 max_outputs++;
597             }
598         }
599 
600         ofstream ANNfile("ANN_growth.m");
601         ANNfile<<"rules = [";
602         for(int i = 0; i<ruleset.size(); i++){
603             if(ruleset[i]!=-1){
604                 if(ruleset[i+1]==-1){
605                     action_nucleotide = genome[ruleset[i] ↙
        +4];
606                     action_value_nucleotide = genome ↙
        [ruleset[i]+5];
607                     if ((make_node[0]<=action_nucleotide) ↙
        &&(make_node[1]>=action_nucleotide)&&(outputs_made ↙
        <max_outputs)){
608                         action_type = 0; ↙

609                         nodes_made++;
610                         if ((make_nodeH[0]<= ↙
        action_nucleotide)&&(make_nodeH[1]>= ↙
        action_nucleotide)){
611                             action_value = 2;
612                             outputs_made++;
```

```
613                         }
614                         else {
615                             action_value = 1;
616                         }
617                     }
618                     else if ((make_connection[0]<=
        action_nucleotide)&&(make_connection[1]>=
        action_nucleotide)){
619                         action_type = 1;
620                         if(action_value_nucleotide >= 51)
        {
621                             action_value = float
        (action_value_nucleotide-50.0)/50.0;
622                         }
623                         else{
624                             action_value = float
        (action_value_nucleotide-51.0)/50.0;
625                         }
626                     }
627                     else if ((do_nothing[0]<=
        action_nucleotide)&&(do_nothing[1]>=
        action_nucleotide)){
628                         //Do nothing
629                     }
630                     else {
631                         action_type = 2;
632                         action_value = random_int(1,
        nodes_made);
633                     }
634                     ANNfile<<action_type<<" "<
        <action_value<<";";
635                 }
636             }
637         }
638         ANNfile<<"];\n";
639     }
640
641     void show_ANN_states(){
642         node temp_node;
643         for(int i=0;i<ANN.get_ANN_size();i++){
644             cout<<ANN.get_activation(i)<<" ";
645         }
646         cout<<endl;
647     }
648     void update_ANN(vector<float> input,bool learning,
        float r_signal){
649         float unbounded_next,bias,slope,h_rate,r_rate,
        old_weight,del_weight;
650         int node_to,node_from,activated_inputs;
651         int ANN_size = ANN.get_ANN_size();
652         int total_inputs = ANN.get_total_inputs();
653         vector<float> node_activation_levels(ANN_size,0.
```

```
        0);
654         vector<float> new_activation_levels(ANN_size,0.0) ↙
        ;
655         node temp_node;
656         connection temp_connection;
657         for(int i=0;i<ANN_size;i++){
658             node_activation_levels[i] = ANN.                ↙
        get_activation(i);
659         }
660         activated_inputs = 0;
661         for(int i=0;i<ANN_size;i++){
662             assert(activated_inputs<=ANN.get_total_inputs ↙
        ());
663             unbounded_next = 0;
664             temp_node = ANN.get_node(i);
665             bias = ANN_biases[i];
666             slope = ANN_slopes[i];
667             for(int j=0;j<ANN_size;j++){
668                 unbounded_next = unbounded_next +           ↙
        ANN_weights[j][i]*node_activation_levels[j];
669             }
670             if(temp_node.get_layer() == 'I'){//An input   ↙
        stays unbounded
671                 new_activation_levels[i] = unbounded_next ↙
         + input[activated_inputs] - bias;
672                 activated_inputs++;
673             }
674             else{
675                 //new_activation_levels[i] = tanh(          ↙
        (unbounded_next-bias)/(2*slope)); //For a range of -1 ↙
         to 1
676                 //For digital nodes ranged 0 - 1
677                 if(unbounded_next>bias){
678                     new_activation_levels[i] = 1;
679                 }
680                 else{
681                     new_activation_levels[i] = 0;
682                 }
683             }
684         }
685         for(int i=0;i<ANN_size;i++){
686             ANN.set_activation(i,new_activation_levels      ↙
        [i]);
687         }
688         if(!learning){
689             return; //Stops here so weights don't change
690         }
691         for(int i=0;i<ANN.get_total_connections();i++){
692             temp_connection = ANN.get_connection(i);
693             node_to = temp_connection.get_node_to();
694             node_from = temp_connection.get_node_from();
695             h_rate = temp_connection.get_Hebbian_rate();
```

```
696             r_rate = temp_connection.get_random_rate();
697             old_weight = temp_connection.get_weight();
698             ANN.set_reinforcement(r_signal);
699             del_weight = 0;
700             //Hebbian Learning
701             del_weight = (1-r_signal)*h_rate*fabs ↙
    (old_weight)*ANN.get_activation(node_from)*ANN. ↙
    get_activation(node_to);
702             //Random Reinforcement
703             del_weight = del_weight + (1-r_signal)*(1- ↙
    r_signal)*r_rate*fabs(old_weight)*random_float(-1,1);
704             ANN.set_weight(i,(old_weight-del_weight));
705         }
706         make_ANN_matrix();
707         //temp_connection = ANN.get_connection(0);
708         //cout<<"To: "<<temp_connection.get_node_to()<<" ↙
    From: "<<temp_connection.get_node_from();
709         //cout<<" Weight: "<<temp_connection.get_weight() ↙
    <<" H Rate: "<<temp_connection.get_Hebbian_rate()< ↙
    <endl;
710     }
711     void eval_XOR_logic(){
712         int no_of_inputs = 2;
713         float desired_no_of_outputs = 1;
714         float exponent = -1;
715         vector<int> connected_outputs;
716         connection test_conn;
717         node test_node;
718         vector<float> test_input(no_of_inputs,0);
719         bool learning = false;
720         float r_signal = 0;
721         int desired_answer;
722         //Tier 1 - check for number of outputs
723         if(ANN.get_total_outputs() == 0){
724             fitness = 0;
725             return;
726         }
727         exponent += ANN.get_total_outputs()/ ↙
    desired_no_of_outputs;
728         if(exponent < 0){
729             fitness = pow(2.0,exponent);
730             return;
731         }
732         //Tier 2 - outputs with connections
733         for(int i=0;i<ANN.get_total_connections();i++){
734             test_conn = ANN.get_connection(i);
735             test_node = ANN.get_node(test_conn. ↙
    get_node_to());
736             if((test_node.get_layer()=='O')&&(!any ↙
    (connected_outputs,test_conn.get_node_to()))){
737                 connected_outputs.push_back(test_conn. ↙
    get_node_to());
```

```
738                 }
739             }
740             exponent += connected_outputs.size()/
        desired_no_of_outputs;
741             if(exponent < 1){
742                 fitness = pow(2.0,exponent);
743                 return;
744             }
745             // Tier 3 - Logic test
746             for(int test_no = 0;test_no<pow(2.0,no_of_inputs)
        ;test_no++){
747                 int2binary(test_no,test_input);
748                 desired_answer = 0;
749                 for(int i=0;i<test_input.size();i++){
750                     if(test_input[i] == 1){
751                         desired_answer++;
752                     }
753                 }
754                 desired_answer = desired_answer%2;
755                 for(float t=0;t<1;t+=0.01){
756                     update_ANN(test_input,learning,r_signal);
757                 }
758                 for(int i=0;i<ANN.get_ANN_size();i++){
759                     test_node = ANN.get_node(i);
760                     if(test_node.get_layer()=='O'){
761                         if(within_range(0.01,ANN.
        get_activation(i),desired_answer)){
762                             exponent++;
763                         }
764                         break;
765                     }
766                 }
767             }
768             fitness = pow(2.0,exponent);
769         }
770
771         void eval_robustness(){
772             //Tier 4 test - remove nodes until logic fails
773             int no_of_inputs = 2;
774             int no_of_outputs = 1;
775             node test_node;
776             float exponent;
777             float tier_4_exponent = 1 + pow(2.0,no_of_inputs)
        ;
778             if(fitness < pow(2.0,tier_4_exponent)){
779                 return;
780             }
781             int node_break;
782             vector<int> broken_nodes;
783             broken_nodes.clear();
784             bool keep_breaking_nodes = true;
785             vector<float> test_input(no_of_inputs,0);
```

```
786         int desired_answer;
787         bool learning = false;
788         float r_signal = 0;
789 
790         for(int i=0;i<ANN.get_ANN_size();i++){
791             node test_node = ANN.get_node(i);
792             if((test_node.get_layer()=='I')|(test_node.    ↙
    get_layer()=='O')){
793                 broken_nodes.push_back(i);
794             }
795         }
796 
797         while((keep_breaking_nodes)&&(broken_nodes.size() ↙
    <ANN.get_ANN_size())){
798             node_break = random_int(0,ANN.get_ANN_size()- ↙
    1,broken_nodes);
799             break_node_off(node_break);
800             broken_nodes.push_back(node_break);
801             //print_vector(broken_nodes);
802             //Logic retested
803             for(int test_no = 0;test_no<pow(2.0,           ↙
    no_of_inputs);test_no++){
804                 int2binary(test_no,test_input);
805                 desired_answer = 0;
806                 for(int i=0;i<test_input.size();i++){
807                     if(test_input[i] == 1){
808                         desired_answer++;
809                     }
810                 }
811                 desired_answer = desired_answer%2;         ↙

812                 for(float t=0;t<1;t+=0.01){
813                     update_ANN(test_input,learning,        ↙
    r_signal);
814                 }
815                 for(int i=0;i<ANN.get_ANN_size();i++){
816                     test_node = ANN.get_node(i);
817                     if(test_node.get_layer()=='O'){
818                         if(!within_range(0.01,ANN.         ↙
    get_activation(i),desired_answer)){
819                             keep_breaking_nodes = false;
820                             broken_nodes.pop_back();
821                         }
822                         break;
823                     }
824                 }
825             }
826         }
827 
828         make_ANN_matrix(); //Rebuilds ANN
829         //print_vector(broken_nodes);
830         //cout<<"ANN size ="<<ANN.get_ANN_size()<<endl;
```

```
831         //indy.show_ANN_matrix();
832         exponent = float(broken_nodes.size()-                    ↙
        (no_of_inputs+no_of_outputs))/ANN.get_ANN_size();
833         //cout<<"exponent = "<<exponent<<endl;
834         fitness = pow(2.0,(tier_4_exponent + 2*exponent)) ↙
        ;
835         if(fitness > pow(2.0,(tier_4_exponent + 2))){ //  ↙
        Sometimes, a bug makes the fitness go to infinity.    ↙
        This is a fix
836             fitness = 0;
837         }
838     }
839     //----------------------------------------------------- ↙
        ------------------------
840 };
841 
842 //-------------------------End of Individual class------- ↙
        ---------------------
843 
844 
845 //-------------------------GENERATE_NEW_INDIVIDUALS------ ↙
        ---------------------
846 void generate_random(vector<individual>& Ark,int l, int   ↙
        gen)
847 {
848     int Ark_size = Ark.size();
849     Ark.push_back(individual());
850     int subject = Ark_size;
851     Ark[Ark_size].generate_random_private(l,gen,subject);
852 }
853 
854 void generate_designed(vector<individual>& Ark,int arr[], ↙
         int gen)
855 {
856     int Ark_size = Ark.size();
857     Ark.push_back(individual());
858     int subject = Ark_size;
859     Ark[Ark_size].generate_designed_private(arr,gen,        ↙
        subject);
860 }
861 
862 void generate_designed(vector<individual>& Ark,vector     ↙
        <int> arr, int gen)
863 {
864     int Ark_size = Ark.size();
865     Ark.push_back(individual());
866     int subject = Ark_size;
867     Ark[Ark_size].generate_designed_private(arr,gen,        ↙
        subject);
868 }
869 
870 void generate_satellite(vector<individual>& Ark,int arr[] ↙
```

```
        ,int gen,int subject)
871 {
872     int Ark_size = Ark.size();
873     Ark.push_back(individual());
874     Ark[Ark_size].generate_designed_private(arr,gen,
        subject);
875 }
876
877 void generate_satellite(vector<individual>& Ark,vector
        <int> arr,int gen,int subject)
878 {
879     int Ark_size = Ark.size();
880     Ark.push_back(individual());
881     Ark[Ark_size].generate_designed_private(arr,gen,
        subject);
882 }
883
884 void generate_reduced(vector<individual>& Ark,vector<int>
         arr,int gen,int parent)
885 {
886     int Ark_size = Ark.size();
887     Ark.push_back(individual());
888     int subject = Ark_size;
889     Ark[Ark_size].generate_reduced_private(arr,gen,
        subject,parent);
890 }
891
892 void mutator(vector<int>& genome, vector<int> genome2,
        float mu, float p_mu, float c_mu, float r_mu, float
        d_mu, float t_mu)
893 {
894     vector<int> proto_genome;
895     vector<int> codon;
896     int skip_to_codon = 0;
897     vector< vector<int> > translocated_codons;
898     vector<int> translocation_codon_numbers;
899     float x,y;
900     int start,stop,temp_int;  //Start and stop FRAME
        numbers
901     float mu_point_mutation, mu_recopy, mu_deletion,
        mu_conjugation, mu_translocation;
902
903     //If there is a mutation within the codon, odds of
        that mutation being of this given type
904     mu_point_mutation = p_mu;  //Make sure
905     mu_conjugation = c_mu;     //these add
906     mu_recopy = r_mu;          //up to 1.0
907     mu_deletion = d_mu;
908     mu_translocation = t_mu;
909     for(int i=0;i<genome.size();i+=6){
910         for(int j=0;j<6;j++){
911             if((i+j)<genome.size()){
```

```
912                 codon.push_back(genome[i+j]);
913             }
914             else{ //Fills genome with dummy nucleotides ↙
        if genome is too short
915                 codon.push_back(100);
916             }
917         }
918         x = rand();
919         y = x/RAND_MAX;
920         if(i<(skip_to_codon*6)){
921             codon.clear();
922         }
923         else if(y > mu){
924             for(int j=0;j<6;j++){
925                 proto_genome.push_back(codon[j]);
926             }
927             codon.clear();
928         }
929         else{  //Perform a mutation
930             y = y/mu; // y is now a random number between ↙
         0 and 1
931             if( y <= mu_point_mutation){
932                 //This will change exactly one nucleotide ↙
         within the reading frame
933                 vector<int> old_nuc;
934                 int change_nuc;
935                 change_nuc = random_int(0,5);
936                 old_nuc.push_back(codon[change_nuc]);
937                 codon[change_nuc] = random_int(1,100,     ↙
        old_nuc);
938                 for(int j=0;j<6;j++){
939                     proto_genome.push_back(codon[j]);
940                 }
941                 codon.clear();
942             }
943             else if(y < (mu_point_mutation+               ↙
        mu_conjugation)){
944                 //This will insert a section from the     ↙
        secondary parent
945                 int lowest, highest;
946                 lowest = 0;
947                 highest = int((genome2.size())/6);
948                 start = random_int(lowest,highest);
949                 stop = random_int(start,highest);
950                 for(int j=(start*6);j<(stop*6);j++){
951                     proto_genome.push_back(genome2[j]);
952                 }
953                 for(int j=0;j<6;j++){
954                     proto_genome.push_back(codon[j]);
955                 }
956                 codon.clear();
957             }
```

```
958             else if(y < (mu_point_mutation+mu_conjugation ↙
        +mu_recopy)){
959                 //This will duplicate a section of the     ↙
        genome
960                 start = i/6;
961                 stop = random_int(start,int(genome.size() ↙
        /6));
962                 for(int j=(start*6);j<(stop*6);j++){
963                     proto_genome.push_back(genome[j]);
964                 }
965                 for(int j=0;j<6;j++){
966                     proto_genome.push_back(codon[j]);
967                 }
968                 codon.clear();
969             }
970             else if(y < (mu_point_mutation+mu_conjugation ↙
        +mu_recopy+mu_deletion)){
971                 //This will delete a section of the        ↙
        genome
972                 start = i/6;
973                 skip_to_codon = random_int(start,int       ↙
        (genome.size()/6));
974                 codon.clear();
975             }
976             else if(y <= (mu_point_mutation+               ↙
        mu_conjugation+mu_recopy+mu_deletion+                  ↙
        mu_translocation)){
977                 //This will delete a section of the        ↙
        genome and save for later insertion
978                 start = i/6;
979                 skip_to_codon = random_int(start,int       ↙
        (genome.size()/6));
980                 for(int j=((start+1)*6);j<(skip_to_codon* ↙
        6);j++){
981                     codon.push_back(genome[j]);
982                 }
983                 translocated_codons.push_back(codon);
984                 codon.clear();
985             }
986         }
987     }
988     genome.clear();
989     int counter = 0;
990     while((counter<translocated_codons.size())&&           ↙
        (translocation_codon_numbers.size()<=int(proto_genome ↙
        .size()/6))){
991         temp_int = random_int(0,int(proto_genome.size()/  ↙
        6),translocation_codon_numbers);
992         translocation_codon_numbers.push_back(temp_int);
993         counter++;
994     }
995     //If the genome is too short, this check will delete  ↙
```

```
     extra translocations
996      if(translocation_codon_numbers.size()                           ↙
     <translocated_codons.size()){
997          translocated_codons.resize                                  ↙
     (translocation_codon_numbers.size());
998      }
999      for(int i=0;i<proto_genome.size();i++){
1000         if((i%6 == 0)&&any(translocation_codon_numbers,             ↙
     int(i/6))){
1001             for(int j=0;j<translocation_codon_numbers.              ↙
     size();j++){
1002                 if((i/6) == translocation_codon_numbers             ↙
     [j]){
1003                     temp_int = j;
1004                 }
1005             }
1006             for(int j=0;j<translocated_codons[temp_int].            ↙
     size();j++){
1007                 genome.push_back(translocated_codons                ↙
     [temp_int][j]);
1008             }
1009         }
1010         genome.push_back(proto_genome[i]);
1011     }
1012     for(int i=0;i<translocation_codon_numbers.size();i++)           ↙
     {
1013         //Inserts translocations
1014         if(translocation_codon_numbers[i] == int                    ↙
     (proto_genome.size()/6)){
1015             for(int j=0;j<translocated_codons[i].size();j           ↙
     ++){
1016                 genome.push_back(translocated_codons[i]             ↙
     [j]);
1017             }
1018         }
1019     }
1020     if(genome.size()<6){
1021         for(int i=genome.size(); i<6; i++){
1022             genome.push_back(100);
1023         }
1024     }
1025     if (genome.size()>600)
1026         genome.resize(300);
1027     /*
1028     for(int i = 0;i<genome.size();i++)
1029     cout<<genome[i]<<" ";
1030     cout<<endl;
1031     */
1032 }
1033
1034 void focused_mutator(vector<int>& genome, vector<int>               ↙
     genome2, float mu, float p_mu, float c_mu, float r_mu              ↙
```

```
     , float d_mu, float t_mu)
1035 {
1036     vector<int> proto_genome;
1037     vector<int> codon;
1038     int skip_to_codon = 0;
1039     vector< vector<int> > translocated_codons;
1040     vector<int> translocation_codon_numbers;
1041     float x,y;
1042     int start,stop,temp_int;  //Start and stop FRAME
     numbers
1043     float mu_point_mutation, mu_recopy, mu_deletion,
     mu_conjugation, mu_translocation;
1044     bool connection_codon;
1045     int make_connection[] = {1,25};
1046     int action_nucleotide;
1047     //If there is a mutation within the codon, odds of
     that mutation being of this given type
1048     mu_point_mutation = p_mu;  //Make sure
1049     mu_conjugation = c_mu;     //these add
1050     mu_recopy = r_mu;          //up to 1.0
1051     mu_deletion = d_mu;
1052     mu_translocation = t_mu;
1053     for(int i=0;i<genome.size();i+=6){
1054         for(int j=0;j<6;j++){
1055             if((i+j)<genome.size()){
1056                 codon.push_back(genome[i+j]);
1057             }
1058             else{ //Fills genome with dummy nucleotides
     if genome is too short
1059                 codon.push_back(100);
1060             }
1061             if(j==5){
1062                 action_nucleotide = genome[i+j];
1063             }
1064         }
1065         connection_codon = false;
1066         if((make_connection[0]<=action_nucleotide)&&
     (make_connection[1]>=action_nucleotide)){
1067             connection_codon = true;
1068         }
1069         x = rand();
1070         y = x/RAND_MAX;
1071         if(i<(skip_to_codon*6)){
1072             codon.clear();
1073         }
1074         else if(y > mu){
1075             for(int j=0;j<6;j++){
1076                 proto_genome.push_back(codon[j]);
1077             }
1078             codon.clear();
1079         }
1080         else{  //Perform a mutation
```

```
1081
1082            //The following will always mutate a ↙
     connection weight
1083            if(connection_codon){
1084                codon[5] = random_int(1,100);
1085            }
1086
1087            y = y/mu; // y is now a random number between ↙
      0 and 1
1088            if( y <= mu_point_mutation){
1089                //This will change exactly one nucleotide ↙
      within the reading frame
1090                vector<int> old_nuc;
1091                int change_nuc;
1092                change_nuc = random_int(0,5);
1093                old_nuc.push_back(codon[change_nuc]);
1094                codon[change_nuc] = random_int(1,100,        ↙
     old_nuc);
1095                for(int j=0;j<6;j++){
1096                    proto_genome.push_back(codon[j]);
1097                }
1098                codon.clear();
1099            }
1100            else if(y < (mu_point_mutation+                  ↙
     mu_conjugation)){
1101                //This will insert a section from the        ↙
     secondary parent
1102                int lowest, highest;
1103                lowest = 0;
1104                highest = int((genome2.size())/6);
1105                start = random_int(lowest,highest);
1106                stop = random_int(start,highest);
1107                for(int j=(start*6);j<(stop*6);j++){
1108                    proto_genome.push_back(genome2[j]);
1109                }
1110                for(int j=0;j<6;j++){
1111                    proto_genome.push_back(codon[j]);
1112                }
1113                codon.clear();
1114            }
1115            else if(y < (mu_point_mutation+mu_conjugation ↙
     +mu_recopy)){
1116                //This will duplicate a section of the      ↙
     genome
1117                start = i/6;
1118                stop = random_int(start,int(genome.size() ↙
     /6));
1119                for(int j=(start*6);j<(stop*6);j++){
1120                    proto_genome.push_back(genome[j]);
1121                }
1122                for(int j=0;j<6;j++){
1123                    proto_genome.push_back(codon[j]);
```

```
1124                }
1125                codon.clear();
1126            }
1127            else if(y < (mu_point_mutation+mu_conjugation ↙
        +mu_recopy+mu_deletion)){
1128                //This will delete a section of the      ↙
        genome
1129                start = i/6;
1130                skip_to_codon = random_int(start,int     ↙
        (genome.size()/6));
1131                codon.clear();
1132            }
1133            else if(y <= (mu_point_mutation+               ↙
        mu_conjugation+mu_recopy+mu_deletion+                  ↙
        mu_translocation)){
1134                //This will delete a section of the      ↙
        genome and save for later insertion
1135                start = i/6;
1136                skip_to_codon = random_int(start,int     ↙
        (genome.size()/6));
1137                for(int j=((start+1)*6);j<(skip_to_codon* ↙
        6);j++){
1138                    codon.push_back(genome[j]);
1139                }
1140                translocated_codons.push_back(codon);
1141                codon.clear();
1142            }
1143        }
1144    }
1145    genome.clear();
1146    int counter = 0;
1147    while((counter<translocated_codons.size())&&             ↙
        (translocation_codon_numbers.size()<=int(proto_genome ↙
        .size()/6))){
1148        temp_int = random_int(0,int(proto_genome.size()/   ↙
        6),translocation_codon_numbers);
1149        translocation_codon_numbers.push_back(temp_int);
1150        counter++;
1151    }
1152    //If the genome is too short, this check will delete  ↙
        extra translocations
1153    if(translocation_codon_numbers.size()                   ↙
        <translocated_codons.size()){
1154        translocated_codons.resize                          ↙
        (translocation_codon_numbers.size());
1155    }
1156    for(int i=0;i<proto_genome.size();i++){
1157        if((i%6 == 0)&&any(translocation_codon_numbers,    ↙
        int(i/6))){
1158            for(int j=0;j<translocation_codon_numbers.     ↙
        size();j++){
1159                if((i/6) == translocation_codon_numbers    ↙
```

```
        [j]){
1160                        temp_int = j;
1161                    }
1162                }
1163                for(int j=0;j<translocated_codons[temp_int].
        size();j++){
1164                    genome.push_back(translocated_codons
        [temp_int][j]);
1165                }
1166            }
1167            genome.push_back(proto_genome[i]);
1168        }
1169        for(int i=0;i<translocation_codon_numbers.size();i++)
        {
1170            //Inserts translocations
1171            if(translocation_codon_numbers[i] == int
        (proto_genome.size()/6)){
1172                for(int j=0;j<translocated_codons[i].size();j
        ++){
1173                    genome.push_back(translocated_codons[i]
        [j]);
1174                }
1175            }
1176        }
1177        if(genome.size()<6){
1178            for(int i=genome.size(); i<6; i++){
1179                genome.push_back(100);
1180            }
1181        }
1182        if (genome.size()>600)
1183            genome.resize(300);
1184        /*
1185        for(int i = 0;i<genome.size();i++)
1186        cout<<genome[i]<<" ";
1187        cout<<endl;
1188        */
1189    }
1190
1191    void reduce_genome(vector<int>& genome,vector< vector
        <int> > rule_table){
1192        //This operation will reduce the genome into the
        rules that actually produced the ANN
1193        vector <int> new_genome;
1194        int reading_frame,temp_int;
1195        for(int i=0;i<rule_table.size();i++){
1196            for(int j=0;j<rule_table[i].size();j++){
1197                reading_frame = rule_table[i][j];
1198                for(int k=0;k<6;k++){
1199                    if(k==0){
1200                        new_genome.push_back(1); //
        Homogenizes IF's
1201                    }
```

```
1202                 else{
1203                     new_genome.push_back(genome
     [reading_frame+k]);
1204                 }
1205             }
1206         }
1207         for(int k=0;k<6;k++){ //Ends Gene
1208             new_genome.push_back(100);
1209         }
1210     }
1211     genome.clear();
1212     genome = new_genome;
1213     if(genome.size()<6){
1214         for(int i=genome.size(); i<6; i++){
1215             genome.push_back(100);
1216         }
1217     }
1218     //print_vector(new_genome);
1219 }
1220
1221 void generate_offspring(vector<individual>& Ark,vector
     <int> arr,int indy1,int indy2,int gen)
1222 {
1223     int Ark_size = Ark.size();
1224     Ark.push_back(individual());
1225     int subject = Ark_size;
1226     Ark[Ark_size].generate_offspring_private(arr,gen,
     subject,indy1,indy2);
1227 }
1228
1229
1230 //--------------------------------------------------------
     -----------------
1231 //------------------------------MAKE_PROTEIN-------------
     -----------------
1232 //--------------------------------------------------------
     -----------------
1233 //The script that transform the genome into proteins/
     programs
1234 void make_protein(individual indy,int no_of_inputs,int
     outputs,int max_conns,int rank_no){
1235     int genome_length = indy.get_genome_length();
1236     int l,openifs,g;
1237     vector<int> genome;
1238     for(int i=0;i<genome_length;i++){
1239         g = indy.get_genome(i);
1240         genome.push_back(g);
1241     }
1242     string filename1= "/scratch/"+indy.get_fcall();
1243     char *filename2;
1244     filename2 = &filename1[0];
1245     ofstream file(filename2);
```

```
1246    //---------------Protien Primer-------------------------↙
        ------------
1247    file << "#include <iostream>\n";
1248    file << "#include <fstream>\n";
1249    file << "#include <vector>\n";
1250    file << "#include <string>\n";
1251    file << "#include <sstream>\n";
1252    file << "#include <ctime>\n";
1253    file << "#include <math.h>\n";
1254    file << "#include \"chimera_lib.h\"\n";
1255    file << "#include \"node_lib_omega4.h\"\n";    //WILL ↙
        HAVE TO CHANGE THIS LINE TO MATCH VERSION
1256    file << "using namespace std;\n";
1257    file << "int main()\n{\n";
1258    file << "neural_net ANN;\n";
1259    file << "string rules;\n";
1260    file << "int no_of_inputs = "<<no_of_inputs<<";\n";
1261    file << "int Max_Outputs = "<<outputs<<";\n";
1262    file << "int Max_Connections = "<<max_conns<<";\n";
1263    file << "int ANN_Size;\n";
1264    file << "float bias,weight;\n";
1265    file << "int NodeA_type1,NodeA_type2,NodeA_type3,      ↙
        NodeA_bias,NodeA_nodes_made,NodeA_inputs,             ↙
        NodeA_outputs;\n";
1266    file << "int NodeB_type1,NodeB_type2,NodeB_type3,      ↙
        NodeB_bias,NodeB_nodes_made,NodeB_inputs,             ↙
        NodeB_outputs;\n";
1267    file << "int relAB_type1,relAB_type2,relAB_type3,      ↙
        relAB_bias,relAB_nodes_made,relAB_inputs,             ↙
        relAB_outputs,relAB_connection;\n";
1268    file << "int relBA_type1,relBA_type2,relBA_type3,      ↙
        relBA_bias,relBA_nodes_made,relBA_inputs,             ↙
        relBA_outputs,relBA_connection;\n";
1269    file << "bool keep_going=true;\n";
1270    file << "bool turn_over=false;\n";
1271    file << "int no_of_outputs = 0;\n";
1272    file << "int energy_units = 200;\n";
1273    // For looped input creation
1274    file << "for(int i=0;i<no_of_inputs;i++) \n" ;
1275    file << "ANN.make_input(0);\n";
1276    file << "while(keep_going && energy_units > 0){\n";
1277    file << "keep_going = false;\n";
1278    file << "ANN_Size = ANN.get_ANN_size();\n";
1279    file << "for(int i=0;i<ANN_Size;i++){\n";
1280    file << "turn_over = false;\n";
1281    file << "node NodeA = ANN.get_node(i);\n";
1282    file << "NodeA_type1 = NodeA.get_type1();\n";
1283    file << "NodeA_type2 = NodeA.get_type2();\n";
1284    file << "NodeA_type3 = NodeA.get_type3();\n";
1285    //Need to change bias into an integer
1286    file << "bias = NodeA.get_bias();\n";
1287    file << "if(bias>0){\n";
```

```
1288     file << "NodeA_bias = int(50*bias+50+0.5);\n}\nelse{\ ↙
         n";
1289     file << "NodeA_bias = int(50*bias+51+0.5);\n}\n";
1290     file << "NodeA_nodes_made = NodeA.get_nodes_made();\n ↙
         ";
1291     file << "NodeA_inputs = ANN.get_inputs_to(i);\n";
1292     file << "NodeA_outputs = ANN.get_outputs_from(i);\n";
1293     file << "for(int j=0;j<ANN_Size;j++){\n";
1294     file << "node NodeB = ANN.get_node(j);\n";
1295     file << "if(turn_over)\n";
1296     file << "break;\n";
1297     file << "NodeB_type1 = NodeB.get_type1();\n";
1298     file << "NodeB_type2 = NodeB.get_type2();\n";
1299     file << "NodeB_type3 = NodeB.get_type3();\n";
1300     //Need to change bias into an integer
1301     file << "bias = NodeB.get_bias();\n";
1302     file << "if(bias>0){\n";
1303     file << "NodeB_bias = int(50*bias+50+0.5);\n}\nelse{\ ↙
         n";
1304     file << "NodeB_bias = int(50*bias+51+0.5);\n}\n";
1305     file << "NodeB_nodes_made = NodeB.get_nodes_made();\n ↙
         ";
1306     file << "NodeB_inputs = ANN.get_inputs_to(j);\n";
1307     file << "NodeB_outputs = ANN.get_outputs_from(j);\n";
1308     file << "relAB_type1 = NodeA_type1 - NodeB_type1;\n";
1309     file << "relAB_type2 = NodeA_type2 - NodeB_type2;\n";
1310     file << "relAB_type3 = NodeA_type3 - NodeB_type3;\n";
1311     file << "relAB_bias = NodeA_bias - NodeB_bias;\n";
1312     file << "relAB_nodes_made = NodeA_nodes_made -         ↙
         NodeB_nodes_made;\n";
1313     file << "relAB_inputs = NodeB_inputs - NodeA_inputs;\ ↙
         n";
1314     file << "relAB_outputs = NodeB_outputs -               ↙
         NodeA_outputs;\n";
1315     file << "weight = ANN.get_connection_weight(i,j);\n";
1316     file << "if(weight>0){\n";
1317     file << "relAB_connection = int(50*weight+50+0.5);\n} ↙
         \nelse{\n";
1318     file << "relAB_connection = int(50*weight+51+0.5);\n} ↙
         \n";
1319     file << "relBA_type1 = NodeB_type1 - NodeA_type1;\n";
1320     file << "relBA_type2 = NodeB_type2 - NodeA_type2;\n";
1321     file << "relBA_type3 = NodeB_type3 - NodeA_type3;\n";
1322     file << "relBA_bias = NodeB_bias - NodeA_bias;\n";
1323     file << "relBA_nodes_made = NodeB_nodes_made -         ↙
         NodeA_nodes_made;\n";
1324     file << "relBA_inputs = NodeA_inputs - NodeB_inputs;\ ↙
         n";
1325     file << "relBA_outputs = NodeA_outputs -               ↙
         NodeB_outputs;\n";
1326     file << "weight = ANN.get_connection_weight(j,i);\n";
1327     file << "if(weight>0){\n";
```

```
1328        file << "relBA_connection = int(50*weight+50+0.5);\n} ↙
            \nelse{\n";
1329        file << "relBA_connection = int(50*weight+51+0.5);\n} ↙
            \n";
1330
1331        //--------------------------------------------------------- ↙
            -----------
1332        openifs = 0;
1333        int if_struct_nucleotide;
1334        int criterion_nucleotide;
1335        int test_value_nucleotide;
1336        int test_range_nucleotide;
1337        int action_nucleotide;
1338        int action_value_nucleotide;
1339        vector<string> action_stack;
1340        vector<int> rule_stack;
1341        l = genome_length - (genome_length%6);
1342        bool action_commented;
1343        for(int i=0;i<l;i+=6){
1344            if_struct_nucleotide = genome[i];
1345            criterion_nucleotide = genome[i+1];
1346            test_value_nucleotide = genome[i+2];
1347            test_range_nucleotide = genome[i+3];
1348            action_nucleotide = genome[i+4];
1349            action_value_nucleotide = genome[i+5];
1350            //----------------IF STRUCTURE ALGORITHM--------- ↙
            ------------------
1351
1352            int make_if[] = {1,38};
1353            int make_end_if[] = {39,54};
1354            int make_end_end_if[] = {55,70};
1355            int make_end[] = {71,80};
1356            int make_end_end[] = {81,90};
1357            int make_end_all[] = {91,100};
1358
1359            if ((make_if[0]<=if_struct_nucleotide)&&(make_if  ↙
            [1]>=if_struct_nucleotide)){
1360                action_commented = false;
1361                file <<"if(";
1362                openifs++;
1363            }
1364            else if ((make_end_if[0]<=if_struct_nucleotide)&& ↙
            (make_end_if[1]>=if_struct_nucleotide)){
1365                action_commented = false;
1366                if(openifs == 0){
1367                    file <<" if(";
1368                    openifs++;
1369                }
1370                else{
1371                    file << action_stack.back();
1372                    file << "if(turn_over){\n";
1373                    file << "rules = rules + \"";
```

```
1374                 for (int j=0;j<rule_stack.size();j++){
1375                     file << int2string(rule_stack[j]) + " ↙
        ";
1376                 }
1377                 file << "-1 \\n\";\n";
1378                 file << "break;\n}\n";
1379                 action_stack.pop_back();
1380                 file<<"/*";
1381                 for(int j=0;j<rule_stack.size();j++)
1382                     file << rule_stack[j]<<"  ";
1383                 file<<"*/";
1384                 rule_stack.pop_back();
1385                 //file << "}//stack is "<<action_stack.   ↙
        size()<<"\n if(";
1386                 file << "}\n if(";
1387             }
1388         }
1389         else if((make_end_end_if[0]<=                    ↙
        if_struct_nucleotide)&&(make_end_end_if[1]>=          ↙
        if_struct_nucleotide)){
1390             action_commented = false;
1391             if(openifs == 0){
1392                 file <<" if(";
1393                 openifs++;
1394             }
1395             else if(openifs == 1){
1396                 file << action_stack.back();
1397                 file << "if(turn_over){\n";
1398                 file << "rules = rules + \"";
1399                 for (int j=0;j<rule_stack.size();j++){
1400                     file << int2string(rule_stack[j]) + " ↙
        ";
1401                 }
1402                 file << "-1 \\n\";\n";
1403                 file << "break;\n}\n";
1404                 action_stack.pop_back();
1405                 file<<"/*";
1406                 for(int j=0;j<rule_stack.size();j++)
1407                     file << rule_stack[j]<<"  ";
1408                 file<<"*/";
1409                 rule_stack.pop_back();
1410                 //file << "}//stack is "<<action_stack.   ↙
        size()<<"\n if(";
1411                 file << "}\n if(";
1412             }
1413             else{
1414                 file << action_stack.back();
1415                 file << "if(turn_over){\n";
1416                 file << "rules = rules + \"";
1417                 for (int j=0;j<rule_stack.size();j++){
1418                     file << int2string(rule_stack[j]) + " ↙
        ";
```

```
1419                    }
1420                    file << "-1 \\n\";\n";
1421                    file << "break;\n}\n";
1422                    action_stack.pop_back();
1423                    file<<"/*";
1424                    for(int j=0;j<rule_stack.size();j++)
1425                        file << rule_stack[j]<<"  ";
1426                    file<<"*/";
1427                    rule_stack.pop_back();
1428                    //file << "}//stack is "<<action_stack. ↙
        size()<<"\n";
1429                    file << "}\n";
1430                    file << action_stack.back();
1431                    file << "if(turn_over){\n";
1432                    file << "rules = rules + \"";
1433                    for (int j=0;j<rule_stack.size();j++){
1434                        file << int2string(rule_stack[j]) + " ↙
         ";
1435                    }
1436                    file << "-1 \\n\";\n";
1437                    file << "break;\n}\n";
1438                    action_stack.pop_back();
1439                    file<<"/*";
1440                    for(int j=0;j<rule_stack.size();j++)
1441                        file << rule_stack[j]<<"  ";
1442                    file<<"*/";
1443                    rule_stack.pop_back();
1444                    //file << "}//stack is "<<action_stack. ↙
        size()<<"\n if(";
1445                    file << "}\n if(";
1446                    openifs--;
1447                }
1448            }
1449            else if((make_end_all[0]<=if_struct_nucleotide)&& ↙
        (make_end_all[1]>=if_struct_nucleotide)){
1450                action_commented = true;
1451                if(openifs == 0)
1452                    file <<"//";
1453                else{
1454                    for(int j=0;j<openifs;j++){
1455                        file << action_stack.back();
1456                        file << "if(turn_over){\n";
1457                        file << "rules = rules + \"";
1458                        for (int k=0;k<rule_stack.size();k++) ↙
        {
1459                            file << int2string(rule_stack[k]) ↙
         + "  ";
1460                        }
1461                        file << "-1 \\n\";\n";
1462                        file << "break;\n}\n";
1463                        action_stack.pop_back();
1464                        file<<"/*";
```

```
1465                      for(int k=0;k<rule_stack.size();k++)
1466                          file << rule_stack[k]<<" ";
1467                      file<<"*/";
1468                      rule_stack.pop_back();
1469                      //file << "}//stack is "<
     <action_stack.size()<<"\n";
1470                      file << "}\n";
1471                  }
1472                  file << "// FORCED END OF GENE stack size
      is "<<action_stack.size()<<".";
1473                  openifs = 0;
1474              }
1475          }
1476
1477          else if((make_end[0]<=if_struct_nucleotide)&&
     (make_end[1]>=if_struct_nucleotide)){
1478              action_commented = true;
1479              if(openifs == 0)
1480                  file <<"//";
1481              else {
1482                  file << action_stack.back();
1483                  file << "if(turn_over){\n";
1484                  file << "rules = rules + \"";
1485                  for (int j=0;j<rule_stack.size();j++){
1486                      file << int2string(rule_stack[j]) + "
      ";
1487                  }
1488                  file << "-1 \\n\";\n";
1489                  file << "break;\n}\n";
1490                  action_stack.pop_back();
1491                  file<<"/*";
1492                  for(int j=0;j<rule_stack.size();j++)
1493                      file << rule_stack[j]<<" ";
1494                  file<<"*/";
1495                  rule_stack.pop_back();
1496                  //file << "}//stack is "<<action_stack.
     size()<<"\n //";
1497                  file << "}\n //";
1498                  openifs = openifs - 1;
1499              }
1500          }
1501          else if((make_end_end[0]<=if_struct_nucleotide)&&
     (make_end_end[1]>=if_struct_nucleotide)){
1502              action_commented = true;
1503              if(openifs == 0)
1504                  file <<"//";
1505              else if(openifs == 1){
1506                  file << action_stack.back();
1507                  file << "if(turn_over){\n";
1508                  file << "rules = rules + \"";
1509                  for (int j=0;j<rule_stack.size();j++){
1510                      file << int2string(rule_stack[j]) + "
```

```
        ";
1511                }
1512                file << "-1 \\n\";\n";
1513                file << "break;\n}\n";
1514                action_stack.pop_back();
1515                file<<"/*";
1516                for(int j=0;j<rule_stack.size();j++)
1517                    file << rule_stack[j]<<" ";
1518                file<<"*/";
1519                rule_stack.pop_back();
1520                //file << "}//stack is "<<action_stack.
        size()<<"\n //";
1521                file << "}\n //";
1522                openifs = 0;
1523            }
1524            else{
1525                action_commented = true;
1526                file << action_stack.back();
1527                file << "if(turn_over){\n";
1528                file << "rules = rules + \"";
1529                for (int j=0;j<rule_stack.size();j++){
1530                    file << int2string(rule_stack[j]) + "
        ";
1531                }
1532                file << "-1 \\n\";\n";
1533                file << "break;\n}\n";
1534                action_stack.pop_back();
1535                file<<"/*";
1536                for(int j=0;j<rule_stack.size();j++)
1537                    file << rule_stack[j]<<" ";
1538                file<<"*/";
1539                rule_stack.pop_back();
1540                //file << "}//stack is "<<action_stack.
        size()<<"\n";
1541                file << "}\n";
1542                file << action_stack.back();
1543                file << "if(turn_over){\n";
1544                file << "rules = rules + \"";
1545                for (int j=0;j<rule_stack.size();j++){
1546                    file << int2string(rule_stack[j]) + "
        ";
1547                }
1548                file << "-1 \\n\";\n";
1549                file << "break;\n}\n";
1550                action_stack.pop_back();
1551                file<<"/*";
1552                for(int j=0;j<rule_stack.size();j++)
1553                    file << rule_stack[j]<<" ";
1554                file<<"*/";
1555                rule_stack.pop_back();
1556                //file << "//stack is "<<action_stack.
        size()<<"\n //";
```

```
1557                    file << "}\n //";
1558                    openifs = openifs - 2;
1559                }
1560            }
1561            else {
1562                cout<<indy.get_fcall()<<"  ";
1563                cout<<"If structure did not use the following ↙
        nucleotide: "<<if_struct_nucleotide<<endl;
1564            }
1565
1566            //cout<<if_struct_nucleotide<<"\t";
1567            //----------------------- TEST PRIMER ----------- ↙
       ----------------
1568            file << "abs(";
1569
1570            //---------CRITERION AND VALUE SET UP ALGORITHM-- ↙
       ------------------
1571
1572            int NodeA_Type1[] = {1,5};
1573            int NodeA_Type2[] = {6,10};
1574            int NodeA_Type3[] = {11,14};
1575            int NodeA_Bias[] = {15,17};
1576            int NodeA_nodes_made[] = {18,20};
1577            int NodeA_inputs[] = {21,23};
1578            int NodeA_outputs[] = {24,26};
1579
1580
1581            int NodeB_Type1[] = {27,31};
1582            int NodeB_Type2[] = {32,36};
1583            int NodeB_Type3[] = {37,40};
1584            int NodeB_Bias[] = {41,43};
1585            int NodeB_nodes_made[] = {44,46};
1586            int NodeB_inputs[] = {47,49};
1587            int NodeB_outputs[] = {50,52};
1588
1589            int RelAB_Type1[] = {53,55};
1590            int RelAB_Type2[] = {56,58};
1591            int RelAB_Type3[] = {59,61};
1592            int RelAB_Bias[] = {62,64};
1593            int RelAB_nodes_made[] = {65,67};
1594            int RelAB_inputs[] = {68,70};
1595            int RelAB_outputs[] = {71,73};
1596            int RelAB_connection[] = {74,76};
1597
1598
1599            int RelBA_Type1[] = {77,79};
1600            int RelBA_Type2[] = {80,82};
1601            int RelBA_Type3[] = {83,85};
1602            int RelBA_Bias[] = {86,88};
1603            int RelBA_nodes_made[] = {89,91};
1604            int RelBA_inputs[] = {92,94};
1605            int RelBA_outputs[] = {95,97};
```

```
1606        int RelBA_connection[] = {98,100};
1607
1608        string value_type;
1609        if((NodeA_Type1[0]<=criterion_nucleotide)&& ↙
    (NodeA_Type1[1]>=criterion_nucleotide)){
1610            file<<"NodeA_type1 ";
1611            value_type = "Type1";
1612        }
1613        else if((NodeA_Type2[0]<=criterion_nucleotide)&& ↙
    (NodeA_Type2[1]>=criterion_nucleotide)){
1614            file<<"NodeA_type2 ";
1615            value_type = "Type2";
1616        }
1617        else if((NodeA_Type3[0]<=criterion_nucleotide)&& ↙
    (NodeA_Type3[1]>=criterion_nucleotide)){
1618            file<<"NodeA_type3 ";
1619            value_type = "Type3";
1620        }
1621        else if((NodeA_Bias[0]<=criterion_nucleotide)&& ↙
    (NodeA_Bias[1]>=criterion_nucleotide)){
1622            file<<"NodeA_bias ";
1623            value_type = "Bias";
1624        }
1625        else if((NodeA_nodes_made[0]<= ↙
    criterion_nucleotide)&&(NodeA_nodes_made[1]>= ↙
    criterion_nucleotide)){
1626            file<<"NodeA_nodes_made ";
1627            value_type = "nodes_made";
1628        }
1629        else if((NodeA_inputs[0]<=criterion_nucleotide)&& ↙
    (NodeA_inputs[1]>=criterion_nucleotide)){
1630            file<<"NodeA_inputs ";
1631            value_type = "connections";
1632        }
1633        else if((NodeA_outputs[0]<=criterion_nucleotide)& ↙
    &(NodeA_outputs[1]>=criterion_nucleotide)){
1634            file<<"NodeA_outputs ";
1635            value_type = "connections";
1636        }
1637        else if((NodeB_Type1[0]<=criterion_nucleotide)&& ↙
    (NodeB_Type1[1]>=criterion_nucleotide)){
1638            file<<"NodeB_type1 ";
1639            value_type = "Type1";
1640        }
1641        else if((NodeB_Type2[0]<=criterion_nucleotide)&& ↙
    (NodeB_Type2[1]>=criterion_nucleotide)){
1642            file<<"NodeB_type2 ";
1643            value_type = "Type2";
1644        }
1645        else if((NodeB_Type3[0]<=criterion_nucleotide)&& ↙
    (NodeB_Type3[1]>=criterion_nucleotide)){
1646            file<<"NodeB_type3 ";
```

```
1647                value_type = "Type3";
1648            }
1649            else if((NodeB_Bias[0]<=criterion_nucleotide)&& ↙
        (NodeB_Bias[1]>=criterion_nucleotide)){
1650                file<<"NodeB_bias ";
1651                value_type = "Bias";
1652            }
1653            else if((NodeB_nodes_made[0]<=                    ↙
        criterion_nucleotide)&&(NodeB_nodes_made[1]>=           ↙
        criterion_nucleotide)){
1654                file<<"NodeB_nodes_made ";
1655                value_type = "nodes_made";
1656            }
1657            else if((NodeB_inputs[0]<=criterion_nucleotide)&& ↙
        (NodeB_inputs[1]>=criterion_nucleotide)){
1658                file<<"NodeB_inputs ";
1659                value_type = "connections";
1660            }
1661            else if((NodeB_outputs[0]<=criterion_nucleotide)& ↙
        &(NodeB_outputs[1]>=criterion_nucleotide)){
1662                file<<"NodeB_outputs ";
1663                value_type = "connections";
1664            }
1665            else if((RelAB_Type1[0]<=criterion_nucleotide)&&  ↙
        (RelAB_Type1[1]>=criterion_nucleotide)){
1666                file<<"relAB_type1 ";
1667                value_type = "Type1";
1668            }
1669            else if((RelAB_Type2[0]<=criterion_nucleotide)&&  ↙
        (RelAB_Type2[1]>=criterion_nucleotide)){
1670                file<<"relAB_type2 ";
1671                value_type = "Type2";
1672            }
1673            else if((RelAB_Type3[0]<=criterion_nucleotide)&&  ↙
        (RelAB_Type3[1]>=criterion_nucleotide)){
1674                file<<"relAB_type3 ";
1675                value_type = "Type3";
1676            }
1677            else if((RelAB_Bias[0]<=criterion_nucleotide)&&   ↙
        (RelAB_Bias[1]>=criterion_nucleotide)){
1678                file<<"relAB_bias ";
1679                value_type = "Bias";
1680            }
1681            else if((RelAB_nodes_made[0]<=                    ↙
        criterion_nucleotide)&&(RelAB_nodes_made[1]>=           ↙
        criterion_nucleotide)){
1682                file<<"relAB_nodes_made ";
1683                value_type = "nodes_made";
1684            }
1685            else if((RelAB_inputs[0]<=criterion_nucleotide)&& ↙
        (RelAB_inputs[1]>=criterion_nucleotide)){
1686                file<<"relAB_inputs ";
```

```
1687                value_type = "connections";
1688            }
1689            else if((RelAB_outputs[0]<=criterion_nucleotide)& ↙
        &(RelAB_outputs[1]>=criterion_nucleotide)){
1690                file<<"relAB_outputs ";
1691                value_type = "connections";
1692            }
1693            else if((RelAB_connection[0]<=                        ↙
        criterion_nucleotide)&&(RelAB_connection[1]>=            ↙
        criterion_nucleotide)){
1694                file<<"relAB_connection ";
1695                value_type = "Bias";
1696            }
1697            else if((RelBA_Type1[0]<=criterion_nucleotide)&&  ↙
        (RelBA_Type1[1]>=criterion_nucleotide)){
1698                file<<"relBA_type1 ";
1699                value_type = "Type1";
1700            }
1701            else if((RelBA_Type2[0]<=criterion_nucleotide)&&  ↙
        (RelBA_Type2[1]>=criterion_nucleotide)){
1702                file<<"relBA_type2 ";
1703                value_type = "Type2";
1704            }
1705            else if((RelBA_Type3[0]<=criterion_nucleotide)&&  ↙
        (RelBA_Type3[1]>=criterion_nucleotide)){
1706                file<<"relBA_type3 ";
1707                value_type = "Type3";
1708            }
1709            else if((RelBA_Bias[0]<=criterion_nucleotide)&&   ↙
        (RelBA_Bias[1]>=criterion_nucleotide)){
1710                file<<"relBA_bias ";
1711                value_type = "Bias";
1712            }
1713            else if((RelBA_nodes_made[0]<=                        ↙
        criterion_nucleotide)&&(RelBA_nodes_made[1]>=            ↙
        criterion_nucleotide)){
1714                file<<"relBA_nodes_made ";
1715                value_type = "nodes_made";
1716            }
1717            else if((RelBA_inputs[0]<=criterion_nucleotide)&& ↙
        (RelBA_inputs[1]>=criterion_nucleotide)){
1718                file<<"relBA_inputs ";
1719                value_type = "connections";
1720            }
1721            else if((RelBA_outputs[0]<=criterion_nucleotide)& ↙
        &(RelBA_outputs[1]>=criterion_nucleotide)){
1722                file<<"relBA_outputs ";
1723                value_type = "connections";
1724            }
1725            else if((RelBA_connection[0]<=                        ↙
        criterion_nucleotide)&&(RelBA_connection[1]>=            ↙
        criterion_nucleotide)){
```

```
1726            file<<"relBA_connection ";
1727            value_type = "Bias";
1728        }
1729        else {
1730            cout<<"Criterion did not use the following ↙
        nucleotide: "<<criterion_nucleotide<<endl;
1731        }
1732        //cout<<criterion_nucleotide<<"\t";
1733        //---------------------VALUE ALGORITHM----------- ↙
        ---------------
1734        if((value_type == "Type1")||(value_type ==        ↙
        "relType1")||(value_type == "Type2")||(value_type == ↙
        "nodes_made")){
1735            int num;
1736            num = int((test_value_nucleotide-1)/12.5);
1737            file<<"- "<<num<<")";
1738        }
1739        else if(value_type == "Bias"){
1740            int num;
1741            num = test_value_nucleotide;
1742            file<<"- "<<num<<")";
1743        }
1744        else if((value_type == "Type3")||(value_type ==   ↙
        "connections")){
1745            int num;
1746            num = test_value_nucleotide-1;
1747            file<<"- "<<num<<")";
1748        }
1749        else{
1750            cout<<indy.get_fcall()<<"  ";
1751            cout<<"Value type ("<<value_type<<") did not  ↙
        use the following nucleotide: "<                     ↙
        <test_value_nucleotide<<endl;
1752        }
1753        //cout<<test_value_nucleotide<<"\t";
1754        //---------------------TEST RANGE ALGORITHM------ ↙
        --------------------
1755        if((value_type == "Type1")||(value_type ==        ↙
        "relType1")||(value_type == "Type2")||(value_type == ↙
        "nodes_made")){
1756            int num;
1757            num = int((test_range_nucleotide-1)/12.5);
1758            file<<" <= "<<num<<"){\n";
1759        }
1760        else if(value_type == "Bias"){
1761            int num;
1762            num = test_range_nucleotide;
1763            file<<" <= "<<num<<"){\n";
1764        }
1765        else if((value_type == "Type3")||(value_type ==   ↙
        "connections")){
1766            int num;
```

```
1767                num = test_range_nucleotide-1;
1768                file<<" <= "<<num<<"){\n";
1769            }
1770            else{
1771                cout<<indy.get_fcall()<<" ";
1772                cout<<"Value type ("<<value_type<<") did not ↙
          use the following nucleotide: "< ↙
          <test_range_nucleotide<<endl;
1773            }
1774            //cout<<test_nucleotide<<"\t";
1775
1776            //----------------------ACTION ALGORITHM--------- ↙
          -------------
1777            int make_connection[] = {1,20};
1778            int do_nothing[] = {21,35};
1779            int end_turn[] = {36,50};
1780            int make_node[] = {51,100};
1781            int make_nodeB[] = {51,55};
1782            int make_nodeC[] = {56,61};
1783            int make_nodeD[] = {62,67};
1784            int make_nodeE[] = {68,73};
1785            int make_nodeF[] = {74,79};
1786            int make_nodeG[] = {80,85};
1787            int make_nodeH[] = {86,100};
1788            if(!action_commented){ //Determined by ↙
          if_structure codon to comment out rule
1789                rule_stack.push_back(i);
1790            }
1791            string temp_stack = " ";
1792            if ((make_connection[0]<=action_nucleotide)&& ↙
          (make_connection[1]>=action_nucleotide)){
1793                temp_stack += "if(make_connection_check(ANN,i ↙
          ,j,Max_Connections)){\n";
1794                temp_stack += "ANN.make_connection(i,j,";
1795                float x,w,h;
1796                if(action_value_nucleotide >= 51){
1797                    w = float(action_value_nucleotide-50.0)/ ↙
          50.0;
1798                }
1799                else{
1800                    w = float(action_value_nucleotide-51.0)/ ↙
          50.0;
1801                }
1802                //w = fabs(w); //Makes evolution of XOR gate ↙
          impossible
1803                x = float(action_nucleotide);
1804                h = x*0.1/64;
1805                temp_stack += float2string(w); //Base weight
1806                temp_stack += ",";
1807                temp_stack += float2string(h); //Hebbian rate
1808                temp_stack += ",";
1809                temp_stack += float2string(0.0); //Random ↙
```

```
     rate
1810         temp_stack += ");\nkeep_going = true;\
     nturn_over = true;\nenergy_units--;\n}\n";
1811     }
1812     else if ((end_turn[0]<=action_nucleotide)&&
     (end_turn[1]>=action_nucleotide)){
1813         temp_stack += "turn_over = true;\n";
1814     }
1815     else if ((make_node[0]<=action_nucleotide)&&
     (make_node[1]>=action_nucleotide)){
1816         if ((make_nodeB[0]<=action_nucleotide)&&
     (make_nodeB[1]>=action_nucleotide)){
1817             temp_stack += "if(make_node_check(ANN,i,
     Max_Outputs)){\n";
1818             temp_stack += "ANN.make_node(i,'H',1,";
1819         }
1820         else if ((make_nodeC[0]<=action_nucleotide)&&
     (make_nodeC[1]>=action_nucleotide)){
1821             temp_stack += "if(make_node_check(ANN,i,
     Max_Outputs)){\n";
1822             temp_stack += "ANN.make_node(i,'H',2,";
1823         }
1824         else if ((make_nodeD[0]<=action_nucleotide)&&
     (make_nodeD[1]>=action_nucleotide)){
1825             temp_stack += "if(make_node_check(ANN,i,
     Max_Outputs)){\n";
1826             temp_stack += "ANN.make_node(i,'H',3,";
1827         }
1828         else if ((make_nodeE[0]<=action_nucleotide)&&
     (make_nodeE[1]>=action_nucleotide)){
1829             temp_stack += "if(make_node_check(ANN,i,
     Max_Outputs)){\n";
1830             temp_stack += "ANN.make_node(i,'H',4,";
1831         }
1832         else if ((make_nodeF[0]<=action_nucleotide)&&
     (make_nodeF[1]>=action_nucleotide)){
1833             temp_stack += "if(make_node_check(ANN,i,
     Max_Outputs)){\n";
1834             temp_stack += "ANN.make_node(i,'H',5,";
1835         }
1836         else if ((make_nodeG[0]<=action_nucleotide)&&
     (make_nodeG[1]>=action_nucleotide)){
1837             temp_stack += "if(make_node_check(ANN,i,
     Max_Outputs)){\n";
1838             temp_stack += "ANN.make_node(i,'H',6,";
1839         }
1840         else if ((make_nodeH[0]<=action_nucleotide)&&
     (make_nodeH[1]>=action_nucleotide)){
1841             temp_stack += "if(make_node_check(ANN,i,
     Max_Outputs)){\n";
1842             temp_stack += "ANN.make_output(i,7,";
1843         }
```

```
1844             float s;
1845             s = pow(10.0,0.0);
1846             temp_stack += float2string(s);
1847             /*
1848             if(action_codon1[2] == 1){
1849             float s;
1850             s = pow(10.0,-2.0);
1851             temp_stack += float2string(s);
1852             }
1853             else if(action_codon1[2] == 2){
1854             float s;
1855             s = pow(10.0,-0.5);
1856             temp_stack += float2string(s);
1857             }
1858             else if(action_codon1[2] == 3){
1859             float s;
1860             s = pow(10.0,0.0);
1861             temp_stack += float2string(s);
1862             }
1863             else if(action_codon1[2] == 4){
1864             float s;
1865             s = pow(10.0,0.5);
1866             temp_stack += float2string(s);
1867             }
1868             else{
1869             cout<<indy.get_fcall()<<" ";
1870             cout<<"did not use the following slope ↙
     nucleotide:"<<action_codon1[2]<<endl;
1871             }
1872             */
1873             temp_stack += ",";
1874             float b;
1875             if(action_value_nucleotide >= 51){
1876                 b = float(action_value_nucleotide-50.0)/ ↙
     50.0;
1877             }
1878             else{
1879                 b = float(action_value_nucleotide-51.0)/ ↙
     50.0;
1880             }
1881             temp_stack += float2string(b);
1882             temp_stack += ");\nkeep_going = true;\ ↙
     nturn_over = true;\nenergy_units--;\n}\n";
1883         }
1884         else{// A 'Do nothing' is added to the stack. It ↙
     does nothing
1885             temp_stack += "//Do nothing \n";
1886         }
1887         if(!action_commented){ //Determined by ↙
     if_structure codon to comment out action
1888             action_stack.push_back(temp_stack);
1889         }
```

```
1890            //cout<<action_nucleotide<<"\t" ;
1891            //cout<<action_value_nucleotide<<"\n" ;
1892        }
1893
1894        //----------------------CLOSER----------------------- ↙
        -------------
1895
1896        if(openifs != 0){
1897            for(int j=0;j<openifs;j++){
1898                for(int k=(action_stack.size()-1);k>=0;k--){
1899                    file << action_stack[k];
1900                    file << "if(turn_over){\n";
1901                    file << "rules = rules + \"";
1902                    for (int I=0;I<rule_stack.size();I++){
1903                        file << int2string(rule_stack[I]) + " ↙
         ";
1904                    }
1905                    file << "-1 \\n\";\n";
1906                    file << "break;\n}\n";
1907                }
1908                action_stack.pop_back();
1909                file<<"/*";
1910                for(int j=0;j<rule_stack.size();j++)
1911                    file << rule_stack[j]<<" ";
1912                file<<"*/";
1913                rule_stack.pop_back();
1914                file << "}\n";
1915            }
1916        }
1917        file<<"}\n}\n}\n";
1918
1919        file<<"string ANNfilename = \"/scratch/ANN"+             ↙
        int2string(rank_no)+".dat\";\n";
1920        file<<"ANN.write_net(ANNfilename);\n";
1921        string rulesfilename = "/scratch/Rules"+int2string    ↙
        (rank_no)+".dat";
1922        file<<"ofstream outfile2(\""<<rulesfilename<<"\");\n" ↙
        ;
1923        file<<"outfile2<<rules;\n";
1924        //file<<"ANN.print_net();\n"; \\Prints ANN to screen
1925        //file<<rules;\n"; \\Prints rules to screen
1926        file<<"return 0;\n}\n";
1927    }
1928    //-------------------------------------------------------- ↙
        -------------------
1929    //----------------------END OF MAKE_PROTIEN-------------- ↙
        -------------------
1930    //-------------------------------------------------------- ↙
        -------------------
1931
1932    //-------------------------------------------------------- ↙
        ----------------------
```

```
1933 //------------------------DATA RECORDING FUNCTIONS-------- ↙
      ----------------------
1934 //------------------------------------------------------- ↙
      ----------------------
1935
1936 //-------------------------RECORD_GEN--------------------- ↙
      ------------
1937 //This function will record the Ark_no and fitness of the ↙
      population at each generation
1938
1939 void Record_Gen(vector<individual> Ark,vector<int> old,   ↙
      vector<int> young, int gen){
1940     ofstream datafile;
1941     datafile.open("Chronograph.txt",ios_base::app);
1942     datafile<<gen<<endl;
1943     for(int i=0;i<old.size();i++){
1944         datafile<<old[i]<<' ';
1945         datafile<<Ark[old[i]].get_fitness()<<' ';
1946         datafile<<endl;
1947     }
1948     for(int i=0;i<young.size();i++){
1949         datafile<<young[i]<<' ';
1950         datafile<<Ark[young[i]].get_fitness()<<' ';
1951         datafile<<endl;
1952     }
1953     datafile.close();
1954 }
1955
1956 //-------------------------READ_LAST_GEN------------------ ↙
      ---------------
1957 //This function will get the state of the last generation ↙
      in Chronograph.txt
1958
1959 void Read_Last_Gen(int N,vector<int>& unmade, vector<int> ↙
      & alive,vector<int>& still_alive, int gen){
1960     ifstream datafile;
1961     int temp_gen,temp_sub;
1962     float temp_fit;
1963     datafile.open("Chronograph.txt");
1964     datafile>>temp_gen;
1965     while((temp_gen <= gen)&&(!datafile.eof())){
1966         if(temp_gen < gen){  //Don't save results
1967             for(int i=0;i<N;i++){
1968                 datafile>>temp_sub;
1969                 datafile>>temp_fit;
1970             }
1971             datafile>>temp_gen;
1972         }
1973         else{  //Save results
1974             for(int i=0;i<N;i++){
1975                 datafile>>temp_sub;
1976                 alive.push_back(temp_sub);
```

```
1977                datafile>>temp_fit;
1978                if(temp_fit == -1)
1979                    unmade.push_back(temp_sub);
1980                else
1981                    still_alive.push_back(temp_sub);
1982            }
1983            temp_gen = gen+1; //This breaks the cycle ↙

1984        }
1985    }
1986    datafile.close();
1987 }
1988
1989 //------------------------ARK_LOAD------------------------ ↙
    -----------------
1990 //This function will place new individuals into Ark.txt
1991
1992 void Ark_Load(individual indy){
1993     ofstream datafile;
1994     datafile.open("Ark.txt",ios_base::app);
1995     datafile<<indy.get_genesis(0)<<' ';
1996     datafile<<indy.get_genesis(1)<<' ';
1997     datafile<<indy.get_genesis(2)<<' ';
1998     datafile<<indy.get_method()<<' ';
1999     datafile<<indy.get_genome_length()<<' ';
2000     for(int j=0;j<indy.get_genome_length();j++)
2001         datafile<<indy.get_genome(j)<<' ';
2002     //datafile<<indy.get_rules_length()<<' ';
2003     //for(int j=0;j<indy.get_rules_length();j++)
2004     //datafile<<indy.get_rule(j)<<' ';
2005     datafile<<endl;
2006     datafile.close();
2007 }
2008
2009 //------------------------DOCK_LOAD----------------------- ↙
    -----------------
2010 //This function will place a genome into Dock.txt, which ↙
    will be read for continued evolution
2011 void Dock_Load(individual indy){
2012     ofstream datafile;
2013     datafile.open("Dock.txt",ios_base::app);
2014     datafile<<indy.get_genesis(0)<<' ';
2015     datafile<<indy.get_genesis(1)<<' ';
2016     datafile<<indy.get_genesis(2)<<' ';
2017     datafile<<indy.get_method()<<' ';
2018     datafile<<indy.get_genome_length()<<' ';
2019     for(int j=0;j<indy.get_genome_length();j++)
2020         datafile<<indy.get_genome(j)<<' ';
2021     //datafile<<indy.get_rules_length()<<' ';
2022     //for(int j=0;j<indy.get_rules_length();j++)
2023     //datafile<<indy.get_rule(j)<<' ';
2024     datafile<<endl;
```

```
2025     datafile.close();
2026 }
2027
```

```
1  using namespace std;
2
3  //------------------- Robot Classes ------------------------↙
       ---------
4  class signal_robot //A bot of 0 size that sends out a       ↙
       signal
5  {private:
6  float position[2];  //The [0] and [1] are x-y location.
7  public:
8      signal_robot(){ //Default Constructor
9          position[0] = 0;
10         position[1] = 0;
11     }
12
13     signal_robot(float x,float y){ //Constructor - given    ↙
       starting position and error
14         position[0] = x;
15         position[1] = y;
16     }
17     float get_x(){
18         return(position[0]);
19     }
20     float get_y(){
21         return(position[1]);
22     }
23     void set_position(float x, float y){
24         position[0] = x;
25         position[1] = y;
26     }
27 };
28
29 class laser_robot
30 {private:
31 float dia; //The diameter of the robot in meters (m)
32 float max_vel; //Maximum magnitude of output velocity (m/   ↙
       s)
33 float position[3]; //The [0] and [1] are x-y location. [2] ↙
        is heading in degrees. 0 is right/east/x-positive
34 float left_wheel; //The output speed of left wheel
35 float right_wheel; //The output speed of right wheel
36 vector <float> goal_sensors; //Activation of goal input     ↙
       nodes
37 vector <float> lasers; //Activation of obstacle input       ↙
       nodes
38 bool goal_line_of_sight;
39 public:
40     laser_robot(){ //Default constructor
41         dia = 1;
42         max_vel = .5;
43         position[0] = 0;
44         position[1] = 0;
45         position[2] = 0;
```

```
46          left_wheel = 0;
47          right_wheel = 0;
48          vector<float> default_lasers(8,0.0);
49          lasers = default_lasers;
50          vector<float> default_goal_sensors(3,0.0);
51          goal_sensors = default_goal_sensors;
52          goal_line_of_sight = false;
53      }
54      laser_robot(float start_x,float start_y,float
        start_ang){ //Constructor - given starting position
55          dia = 1;
56          max_vel = .5;
57          position[0] = start_x;
58          position[1] = start_y;
59          position[2] = start_ang;
60          left_wheel = 0;
61          right_wheel = 0;
62          vector<float> default_lasers(8,0.0);
63          lasers = default_lasers;
64          vector<float> default_goal_sensors(3,0.0);
65          goal_sensors = default_goal_sensors;
66          goal_line_of_sight = false;
67      }
68      laser_robot(int l_s,float start_x,float start_y,float
        start_ang){ //Constructor - laser size given starting
        position
69          dia = 1;
70          max_vel = .5;
71          position[0] = start_x;
72          position[1] = start_y;
73          position[2] = start_ang;
74          left_wheel = 0;
75          right_wheel = 0;
76          vector<float> default_lasers(l_s,0.0);
77          lasers = default_lasers;
78          vector<float> default_goal_sensors(3,0.0);
79          goal_sensors = default_goal_sensors;
80          goal_line_of_sight = false;
81      }
82      laser_robot(float d,float v,int l_s){ //Constructor -
        given diameter, max velocity and laser size
83          dia = d;
84          max_vel = v;
85          position[0] = 0;
86          position[1] = 0;
87          position[2] = 0;
88          left_wheel = 0;
89          right_wheel = 0;
90          vector<float> default_lasers(l_s,0.0);
91          lasers = default_lasers;
92          vector<float> default_goal_sensors(3,0.0);
93          goal_sensors = default_goal_sensors;
```

```
 94             goal_line_of_sight = false;
 95         }
 96         laser_robot(float d,float v,int l_s,float start_x,     ↙
            float start_y,float start_ang){ //Constructor - given  ↙
            diameter, max velocity, sensor info, and starting      ↙
            position
 97             dia = d;
 98             max_vel = v;
 99             position[0] = start_x;
100             position[1] = start_y;
101             position[2] = start_ang;
102             left_wheel = 0;
103             right_wheel = 0;
104             vector<float> default_lasers(l_s,0.0);
105             lasers = default_lasers;
106             vector<float> default_goal_sensors(3,0.0);
107             goal_sensors = default_goal_sensors;
108             goal_line_of_sight = false;
109         }
110         laser_robot(float d,float v,int l_s,float start_x,     ↙
            float start_y,float start_ang,float left_vel,float     ↙
            right_vel){ //Constructor - given diameter, max        ↙
            velocity, sensor info, starting position and velocity
111             dia = d;
112             max_vel = v;
113             position[0] = start_x;
114             position[1] = start_y;
115             position[2] = start_ang;
116             left_wheel = left_vel;
117             right_wheel = right_vel;
118             vector<float> default_lasers(l_s,0.0);
119             lasers = default_lasers;
120             vector<float> default_goal_sensors(3,0.0);
121             goal_sensors = default_goal_sensors;
122             goal_line_of_sight = false;
123         }
124         void operator= (const laser_robot& right){
125             if (this != &right){
126                 dia = right.dia;
127                 max_vel = right.max_vel;
128                 position[0] = right.position[0];
129                 position[1] = right.position[1];
130                 position[2] = right.position[2];
131                 left_wheel = right.left_wheel;
132                 right_wheel = right.right_wheel;
133                 lasers = right.lasers;
134                 goal_sensors = right.goal_sensors;
135                 goal_line_of_sight = right.goal_line_of_sight;
136             }
137         }
138
139         float get_diameter(){
```

```
140         return(dia);
141     }
142     float get_max_velocity(){
143         return(max_vel);
144     }
145     float get_x(){
146         return(position[0]);
147     }
148     float get_y(){
149         return(position[1]);
150     }
151     float get_heading(){
152         return(position[2]);
153     }
154     void set_position(float x, float y,float ang){
155         position[0] = x;
156         position[1] = y;
157         position[2] = ang;
158     }
159     float get_left_wheel(){
160         return(left_wheel);
161     }
162     void set_left_wheel(float x){
163         left_wheel = x;
164     }
165     float get_right_wheel(){
166         return(right_wheel);
167     }
168     void set_right_wheel(float x){
169         right_wheel = x;
170     }
171     void get_goal_sensors(vector<float>& s){
172         s = goal_sensors;
173     }
174     void set_goal_sensors(vector<float> s){
175         goal_sensors = s;
176     }
177     int get_number_of_goal_sensors(){
178         int temp_int = goal_sensors.size();
179         return(temp_int);
180     }
181     void get_lasers(vector<float>& l){
182         l = lasers;
183     }
184     void set_lasers(vector<float> l){
185         lasers = l;
186     }
187     int get_number_of_lasers(){
188         int temp_int = lasers.size();
189         return(temp_int);
190     }
191     void set_goal_visible_on(){
```

```
192         goal_line_of_sight = true;
193     }
194     void set_goal_visible_off(){
195         goal_line_of_sight = false;
196     }
197     bool get_goal_visible(){
198         return(goal_line_of_sight);
199     }
200 };
201
202 //------------------ End of Robot Class ------------------
    -----
203
204 //-------------------------------World Class--------------
    --------------------
205 class simulation_world
206 {private:
207 vector<vector<float> > obstacles;
208 vector<laser_robot> laserbots;
209 vector<signal_robot> sigbots;
210 public:
211     simulation_world(){
212     }
213     simulation_world(vector<vector<float> > obs){ //
    Constructor - given the obstacles
214         for(int i=0;i<obs.size();i++){
215             //Makes sure obstacles have an even number of
    coordinates and there are at least 3 of them
216             assert((obs[i].size()%2)==0);
217             assert(obs[i].size() >= 6);
218         }
219         obstacles = obs;
220         laserbots.clear();
221         sigbots.clear();
222     }
223     simulation_world(vector<float> obs){ //Constructor -
    given one obstacle
224         //Makes sure obstacle has an even number of
    coordinates and there are at least 3 of them
225         assert((obs.size()%2)==0);
226         assert(obs.size() >= 6);
227         obstacles.clear();
228         obstacles.push_back(obs);
229         laserbots.clear();
230         sigbots.clear();
231     }
232     simulation_world(vector<laser_robot> bots){ //
    Constructor - given the laser robots
233         obstacles.clear();
234         laserbots = bots;
235         sigbots.clear();
236     }
```

```
237     simulation_world(vector<signal_robot> bots){ //
        Constructor - given the signal robots
238         obstacles.clear();
239         laserbots.clear();
240         sigbots = bots;
241     }
242     simulation_world(vector<laser_robot> lbots, vector
        <signal_robot> sbots){ //Constructor - given the laser
         and signal robots
243         obstacles.clear();
244         laserbots = lbots;
245         sigbots = sbots;
246     }
247     simulation_world(vector<vector<float> > obs, vector
        <laser_robot> lbots, vector<signal_robot> sbots){ //
        Constructor - given the obstacles and robots
248         for(int i=0;i<obs.size();i++){
249             //Makes sure obstacles have an even number of
        coordinates and there are at least 3 of them
250             assert((obs[i].size()%2)==0);
251             assert(obs[i].size() >= 6);
252         }
253         obstacles = obs;
254         laserbots = lbots;
255         sigbots = sbots;
256     }
257     void operator= (const simulation_world& right){
258         if (this != &right){
259             obstacles = right.obstacles;
260             laserbots = right.laserbots;
261             sigbots = right.sigbots;
262         }
263     }
264     void build_obstacle(vector<float> obs){
265         //Makes sure obstacle has an even number of
        coordinates and there are at least 3 of them
266         assert((obs.size()%2)==0);
267         assert(obs.size() >= 6);
268         obstacles.push_back(obs);
269     }
270     void build_obstacle(float obs[],int obs_size){
271         //Makes sure obstacle has an even number of
        coordinates and there are at least 3 of them
272         assert((obs_size%2)==0);
273         assert(obs_size >= 6);
274         vector<float> new_obstacle;
275         for(int i=0;i<obs_size;i++){
276             new_obstacle.push_back(obs[i]);
277         }
278         obstacles.push_back(new_obstacle);
279     }
280     void build_obstacle(const float obs[],int obs_size){
```

```
281         //Makes sure obstacle has an even number of
        coordinates and there are at least 3 of them
282         assert((obs_size%2)==0);
283         assert(obs_size >= 6);
284         vector<float> new_obstacle;
285         for(int i=0;i<obs_size;i++){
286             new_obstacle.push_back(obs[i]);
287         }
288         obstacles.push_back(new_obstacle);
289     }
290     int no_of_obstacles(){
291         return(obstacles.size());
292     }
293     void get_obstacle(int n,vector<float>& obs){
294         assert(n<obstacles.size());
295         obs.clear();
296         obs = obstacles[n];
297     }
298     void get_all_obstacles(vector< vector<float> >& obs){
299         obs = obstacles;
300     }
301     void clear_all_obstacles(){
302         obstacles.clear();
303     }
304     void clear_internal_obstacles(){
305         vector<float>   border = obstacles[0];
306         obstacles.clear();
307         obstacles.push_back(border);
308     }
309     int get_no_of_laser_robots(){
310         return(laserbots.size());
311     }
312     void add_laser_robot(laser_robot new_bot){
313         laserbots.push_back(new_bot);
314     }
315     void add_signal_robot(signal_robot new_bot){
316         sigbots.push_back(new_bot);
317     }
318     void move_laser_robot(int n,float x,float y,float ang)
        {
319         assert(n<laserbots.size());
320         laserbots[n].set_position(x,y,ang);
321     }
322     void update_laser_bot_actuators(int n,float x1,float
        x2){
323         assert(n<laserbots.size());
324         laserbots[n].set_left_wheel(x1);
325         laserbots[n].set_right_wheel(x2);
326     }
327     void move_signal_robot(int n,float x,float y){
328         assert(n<sigbots.size());
329         sigbots[n].set_position(x,y);
```

```
330     }
331     laser_robot get_laser_robot(int n){
332         assert(n<laserbots.size());
333         return(laserbots[n]);
334     }
335     signal_robot get_signal_robot(int n){
336         assert(n<sigbots.size());
337         return(sigbots[n]);
338     }
339     void clear_all_laser_robots(){
340         laserbots.clear();
341     }
342     void clear_all_signal_robots(){
343         sigbots.clear();
344     }
345     void clear_all_robots(){
346         laserbots.clear();
347         sigbots.clear();
348     }
349     bool update_world(float dt){
350         //Gets actuator states and moves each bots, then ↙
        updates sensor states for each bot.
351         laser_robot bot, bot_2;
352         vector<float> obs;
353         bool collision;
354         //The following vectors are labeled in Vol 4 pg 10 ↙
        -13
355         float diameter,v1,v2,x,y,heading,r;
356         float alpha,gamma,beta,theta; //Used for ↙
        determining new states
357         float test_x,test_y,bot_2x,bot_2y;
358         //The following vectors are labeled in Vol 3 pg 98
359         float p0x,p0y,p1x,p1y,p2x,p2y,bx,by,ax,ay,A;
360         //The following vectors are labeled in Vol 3 pg 95 ↙
         & 100
361         float lx,ly,lpx,lpy,v1x,v1y,v2x,v2y,hx,hy,range, ↙
        test_range,phi;
362         //The following vectors are labeled in Vol 4 pg ↙
        109
363         float nx,ny;
364         int ob_hit; //Used for debugging
365         int no_lasers;
366         vector<float> new_lasers;
367         int no_goal_sensors = bot. ↙
        get_number_of_goal_sensors();
368         assert(no_goal_sensors==3); //As of now, logic ↙
        works ONLY if there are 3 sensors
369         vector<float> new_goal_sensors(no_goal_sensors,0. ↙
        0);
370         float sx,sy,sensor_angle,goal_dist,swarmx,swarmy;
371         bool swarm_sees_goal;
372         int no_of_swarm_sees_goal;
```

```
373         for(int i=0;i<laserbots.size();i++){
374             collision = false;
375             //Getting actuator states
376             bot = laserbots[i];
377             diameter = bot.get_diameter();
378             r = diameter/2;
379             v1 = dt*bot.get_left_wheel();
380             v2 = dt*bot.get_right_wheel();
381             x = bot.get_x();y = bot.get_y();heading = bot. ↙
    get_heading();
382             //Finds new spot - logic on Vol 4 pg. 10 - 13
383             assert((1-((v1-v2)*(v1-v2))/(2*diameter))>=-1) ↙
    ;
384             assert((1-((v1-v2)*(v1-v2))/(2*diameter))<=1);
385             alpha = acos(1-((v1-v2)*(v1-v2))/(2*diameter*  ↙
    diameter));
386             //Accounts for CW or no rotations
387             if(v1==v2){
388                 alpha = 0;
389             }else if(v1>v2){
390                 alpha = -alpha;
391             }
392             gamma = atan2(-cos(heading),sin(heading));
393             beta = (pi()-fabs(alpha))/2;
394             if(v1 > v2){
395                 theta = gamma + beta;
396             }
397             else{
398                 theta = gamma + pi() - beta;
399             }
400             test_x = x + (v1*cos(theta) + v2*cos(theta))/2 ↙
    ; //New test position
401             test_y = y + (v1*sin(theta) + v2*sin(theta))/2 ↙
    ;
402             heading = heading + alpha; //New heading.      ↙
    Robot can always turn even if it hits an obstacle
403             //The follow ensure heading is within +/- pi
404             while(heading > pi()){
405                 heading -=  2*pi();
406             }
407             while(heading <= -pi()){
408                 heading +=  2*pi();
409             }
410             //Check to make sure new spot isn't within an  ↙
    obstacle
411             //Logic on Vol 3 pg 97-98
412             p0x = test_x; p0y = test_y;
413             for(int j=0;j<obstacles.size();j++){
414                 if(collision){
415                     break;
416                 }
417                 obs = obstacles[j];
```

```
418                     for(int k=0;k<obs.size();k+=2){
419                         if((k+3)<obs.size()){
420                             p1x = obs[k];
421                             p1y = obs[k+1];
422                             p2x = obs[k+2];
423                             p2y = obs[k+3];
424                         }
425                         else{
426                             p1x = obs[k];
427                             p1y = obs[k+1];
428                             p2x = obs[0];
429                             p2y = obs[1];
430                         }
431                         bx = p2x - p1x;
432                         by = p2y - p1y;
433                         gamma = ((bx*p0x+by*p0y)-(bx*p1x+by*  ↙
    p1y))/(bx*bx+by*by);
434                         if((0<gamma)&&(gamma<1)){
435                             //Bot may hit the wall
436                             ax = p1x + gamma*bx - p0x;
437                             ay = p1y + gamma*by - p0y;
438                             A = ax*ax + ay*ay;
439                             if (A < (r*r)){
440                                 collision = true;
441                                 /*
442                                 cout<<"COLLISION!"<<endl;
443                                 cout<<"Robot "<<i<<" hit      ↙
    obstacle "<<j;
444                                 cout<<" wall with verticies   ↙
    defined at (";
445                                 cout<<p1x<<","<<p1y<<") and (" ↙
    <<p2x<<","<<p2y<<")"<<endl;
446                                 */
447                             }
448                         }
449                         else{
450                             //Bot may still hit a vertex
451                             ax = p0x - p1x; ay = p0y - p1y;
452                             A = ax*ax + ay*ay;
453                             if (A < (r*r)){
454                                 collision = true;
455                                 /*
456                                 cout<<"COLLISION!"<<endl;
457                                 cout<<"Robot "<<i<<" hit      ↙
    obstacle "<<j;
458                                 cout<<" at vertex ("<<p1x<<"," ↙
    <<p1y<<")"<<endl;
459                                 */
460                             }
461                             ax = p0x - p2x; ay = p0y - p2y;
462                             A = ax*ax + ay*ay;
463                             if (A < (r*r)){
```

```
464                                 collision = true;
465                                 /*
466                                 cout<<"COLLISION!"<<endl;
467                                 cout<<"Robot "<<i<<" hit     ↙
    obstacle "<<j;
468                                 cout<<" at vertex ("<<p2x<<"," ↙
    <<p2y<<")"<<endl;
469                                 */
470                             }
471                         }
472                     }
473                 }
474                 //Checks to make sure it won't hit another     ↙
    robot
475                 for(int j=0;j<laserbots.size();j++){
476                     if(i!=j){
477                         bot_2 = laserbots[j];
478                         bot_2x = bot_2.get_x();bot_2y = bot_2. ↙
    get_y();
479                         ax = test_x - bot_2x;
480                         ay = test_y - bot_2y;
481                         A = ax*ax + ay*ay;
482                         if (A < (4*r*r)){
483                             collision = true;
484                             /*
485                             cout<<"COLLISION!"<<endl;
486                             cout<<"Robot "<<i<<" hit Robot "<  ↙
    <j<<endl;
487                             */
488                         }
489                     }
490                 }
491                 if(!collision){
492                     x = test_x; y = test_y;
493                 }
494                 laserbots[i].set_position(x,y,heading);//      ↙
    Updates the robot's position
495             }
496             //After each bot has moved, the sensors of each    ↙
    bot are updated
497             for(int i=0;i<laserbots.size();i++){               ↙

498                 //Updates the robots lasers - Logic on Vol 3   ↙
    pg 95 & 100
499                 //cout<<"Robot: "<<i<<endl;
500                 bot = laserbots[i];
501                 no_lasers = bot.get_number_of_lasers();
502                 new_lasers.clear();
503                 new_lasers.resize(no_lasers);
504                 phi = pi()/(no_lasers+1);
505                 p0x = bot.get_x(); p0y = bot.get_y();
506                 heading = bot.get_heading();
```

```
507                 for(int j=0;j<no_lasers;j++){
508                     theta = heading + pi()/2 - (j+1)*phi;
509                     lx = cos(theta); ly = sin(theta);
510                     lpx = -sin(theta); lpy = cos(theta);
511                     range = RAND_MAX;
512                     ob_hit = -1;
513                     //Checks obstacles
514                     for(int k=0;k<obstacles.size();k++){
515                         obs = obstacles[k];
516                         for(int m=0;m<obs.size();m+=2){
517                             if((m+3)<obs.size()){
518                                 p1x = obs[m];
519                                 p1y = obs[m+1];
520                                 p2x = obs[m+2];
521                                 p2y = obs[m+3];
522                             }
523                             else{
524                                 p1x = obs[m];
525                                 p1y = obs[m+1];
526                                 p2x = obs[0];
527                                 p2y = obs[1];
528                             }
529                             v1x = p1x-p0x; v1y = p1y-p0y;
530                             v2x = p2x-p0x; v2y = p2y-p0y;
531                             if(((lpx*v1x+lpy*v1y)*(lpx*v2x+lpy
        *v2y))<=0){
532                                 if(((lpx*v1x+lpy*v1y)*(lpx*v2x
        +lpy*v2y))==0){
533                                     if(((lpx*v1x+lpy*v1y)==0)&
        &((lpx*v2x+lpy*v2y)==0)){
534                                         test_range = min((v1x*
        v1x+v1y*v1y),(v2x*v2x+v2y*v2y));
535                                         if(range > pow
        (test_range,0.5)){
536                                             range = pow
        (test_range,0.5);
537                                             ob_hit = k;
538                                         }
539                                     }
540                                     else if ((lpx*v1x+lpy*v1y)
        ==0){
541                                         test_range = pow((v1x*
        v1x+v1y*v1y),0.5);
542                                         if(range > test_range)
        {
543                                             range = test_range
        ;
544                                             ob_hit = k;
545                                         }
```

```
546                                     }
547                                     else if ((lpx*v2x+lpy*v2y) ↙
    ==0){
548                                         test_range = pow((v2x* ↙
    v2x+v2y*v2y),0.5);
549                                         if(range > test_range) ↙
    {
550                                             range = test_range ↙
    ;
551                                             ob_hit = k;
552                                         }
553                                     }
554                                 }
555                                 else{
556                                     hx = v2x-v1x; hy = v2y-v1y ↙
    ;
557                                     test_range = (v1x*hy-v1y*  ↙
    hx)/(lx*hy-ly*hx);
558                                     if((range > test_range)&&  ↙
    (test_range>0)){
559                                         range = test_range;
560                                         ob_hit = k;
561                                     }
562                                 }
563                             }
564                         }
565                     }
566                     //Checks other robots
567                     for(int k=0;k<laserbots.size();k++){
568                         if(k!=i){
569                             nx = laserbots[k].get_x();
570                             ny = laserbots[k].get_y();
571                             test_range = (lx*(nx-p0x)+ly*(ny-  ↙
    p0y))/(lx*lx+ly+ly);
572                             ax = nx - p0x - test_range*lx;
573                             ay = ny - p0y - test_range*ly;
574                             if(((ax*ax+ay*ay)<(r*r))&&(range>  ↙
    test_range)&&(test_range>0)){
575                                 range = test_range;
576                                 ob_hit = k;
577                             }
578                         }
579                     }
580 
581                     if (range == RAND_MAX){
582                         cout<<"WARNING: Laser "<<j<<" did not  ↙
    detect any obstacles"<<endl;
583                         new_lasers[j] = 0;
584                     }
585                     else{
586                         new_lasers[j] = range;
587                     }
```

```
588                 /*
589                 cout<<"Laser "<<j<<": Range = "<<range<
    <endl;
590                 cout<<" obs = "<<ob_hit<<": l = ["<<range*
    lx<<" "<<range*ly<<"]"<<endl;
591                 */
592             }
593             laserbots[i].set_lasers(new_lasers);
594             //Goal sensors are updated
595             p1x = sigbots[0].get_x();
596             p1y = sigbots[0].get_y();
597             sx = p1x - p0x; sy = p1y - p0y;
598
599             goal_dist = pow((sx*sx+sy*sy),0.5);
600
601             //Finds if goal is within robot line of sight
602             theta = atan2(sy,sx);
603             lx = cos(theta); ly = sin(theta);
604             lpx = -sin(theta); lpy = cos(theta);

605             range = RAND_MAX;
606             ob_hit = -1;
607             for(int j=0;j<obstacles.size();j++){
608                 obs = obstacles[j];
609                 for(int m=0;m<obs.size();m+=2){
610                     if((m+3)<obs.size()){
611                         p1x = obs[m];
612                         p1y = obs[m+1];
613                         p2x = obs[m+2];
614                         p2y = obs[m+3];
615                     }
616                     else{
617                         p1x = obs[m];
618                         p1y = obs[m+1];
619                         p2x = obs[0];
620                         p2y = obs[1];
621                     }
622                     v1x = p1x-p0x; v1y = p1y-p0y;
623                     v2x = p2x-p0x; v2y = p2y-p0y;

624                     if(((lpx*v1x+lpy*v1y)*(lpx*v2x+lpy*
    v2y))<=0){
625                         if(((lpx*v1x+lpy*v1y)*(lpx*v2x+lpy
    *v2y))==0){
626                             if(((lpx*v1x+lpy*v1y)==0)&&(
    (lpx*v2x+lpy*v2y)==0)){
627                                 test_range = min((v1x*v1x+
    v1y*v1y),(v2x*v2x+v2y*v2y));
628                                 if(range > pow(test_range,
    0.5)){
629                                     range = pow(test_range
    ,0.5);
```

```
630                                             ob_hit = j;
631                                         }
632                                     }
633                                     else if ((lpx*v1x+lpy*v1y)==0) ↙
    {
634                                         test_range = pow((v1x*v1x+ ↙
    v1y*v1y),0.5);
635                                         if(range > test_range){
636                                             range = test_range;
637                                             ob_hit = j;
638                                         }
639                                     }
640                                     else if ((lpx*v2x+lpy*v2y)==0) ↙
    {
641                                         test_range = pow((v2x*v2x+ ↙
    v2y*v2y),0.5);
642                                         if(range > test_range){
643                                             range = test_range;
644                                             ob_hit = j;
645                                         }
646                                     }
647                                 }
648                                 else{
649                                     hx = v2x-v1x; hy = v2y-v1y;
650                                     test_range = (v1x*hy-v1y*hx)/  ↙
    (lx*hy-ly*hx);
651                                     if((range > test_range)&&      ↙
    (test_range>0)){
652                                         range = test_range;
653                                         ob_hit = j;
654                                     }
655                                 }
656                             }
657                         }
658                     }
659 
660                     if(range < goal_dist){
661                         laserbots[i].set_goal_visible_off();
662                     }
663                     else{
664                         laserbots[i].set_goal_visible_on();
665                     }
666 
667                     if(laserbots[i].get_goal_visible()){//If it    ↙
    can see the goal, go to it
668                         sensor_angle = atan2(sy,sx) - heading;
669                         //The follow ensure sensor_angle is within ↙
     +/- pi
670                         while(sensor_angle > pi()){
671                             sensor_angle -=  2*pi();
672                         }
673                         while(sensor_angle <= -pi()){
```

```
674                 sensor_angle +=  2*pi();
675             }
676             //cout<<"sensor_angle = "<<(sensor_angle*
    180/pi())<<endl;
677             if(fabs(sensor_angle)<=pi()/8){
678                 new_goal_sensors[0] = 0;
679                 new_goal_sensors[1] = 1;
680                 new_goal_sensors[2] = 0;
681             }
682             else if(sensor_angle < 0){
683                 assert(fabs(sensor_angle)>pi()/8);
684                 new_goal_sensors[0] = 0;
685                 new_goal_sensors[1] = 0;
686                 new_goal_sensors[2] = 1;

687             }
688             else if(sensor_angle > 0){
689                 assert(fabs(sensor_angle)>pi()/8);
690                 new_goal_sensors[0] = 1;
691                 new_goal_sensors[1] = 0;
692                 new_goal_sensors[2] = 0;
693             }
694         }
695         else{
696             swarm_sees_goal = false;
697             no_of_swarm_sees_goal = 0;
698             swarmx = 0;
699             swarmy = 0;
700             for(int j=0;j<laserbots.size();j++){//See
    if others see the goal...
701                 if(laserbots[j].get_goal_visible()){
702                     swarm_sees_goal = true;
703                     no_of_swarm_sees_goal++;

704                     swarmx += laserbots[j].get_x();
705                     swarmy += laserbots[j].get_y();
706                 }
707             }
708             if(swarm_sees_goal){ //If so, go to center
     of others
709                 assert(no_of_swarm_sees_goal != 0);
710                 swarmx = swarmx/float
    (no_of_swarm_sees_goal) - p0x; //Gives relative
    position
711                 swarmy = swarmy/float
    (no_of_swarm_sees_goal) - p0y; //Gives relative
    position
712                 sensor_angle = atan2(swarmy,swarmx) -
    heading;
713                 //The follow ensure sensor_angle is
    within +/- pi
714                 while(sensor_angle > pi()){
```

```
715                          sensor_angle -=  2*pi();
716                      }
717                      while(sensor_angle <= -pi()){
718                          sensor_angle +=  2*pi();
719                      }
720                      //cout<<"swarmx = "<<swarmx<<endl;
721                      //cout<<"swarmy = "<<swarmy<<endl;
722                      //cout<<"sensor_angle = "<<
     (sensor_angle*180/pi())<<endl;
723                      if(fabs(sensor_angle)<=pi()/8){
724                          new_goal_sensors[0] = 0;
725                          new_goal_sensors[1] = 1;
726                          new_goal_sensors[2] = 0;
727                      }
728                      else if(sensor_angle < 0){
729                          assert(fabs(sensor_angle)>pi()/8);
730                          new_goal_sensors[0] = 0;
731                          new_goal_sensors[1] = 0;
732                          new_goal_sensors[2] = 1;

733                      }
734                      else if(sensor_angle > 0){
735                          assert(fabs(sensor_angle)>pi()/8);
736                          new_goal_sensors[0] = 1;
737                          new_goal_sensors[1] = 0;
738                          new_goal_sensors[2] = 0;
739                      }
740                  }
741                  else{ //no one knows nothin'
742                      new_goal_sensors[0] = 0;
743                      new_goal_sensors[1] = 0;
744                      new_goal_sensors[2] = 0;
745                  }
746              }
747              //cout<<"goal sensors = ";
748              //print_vector(new_goal_sensors);
749              laserbots[i].set_goal_sensors
     (new_goal_sensors);
750          }
751      return(collision);
752      }
753  };
754
755  //---------------------------End of World Class-----------
     ---------------------
756
757
```