Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2020

Project Title:	**Event-Driven Deep Spiking Neural Network**
Student:	**Josh Jennings**
CID:	**01221510**
Course:	**EEE4**
Project Supervisor:	**Dr D.B. Thomas**
Second Marker:	**Dr J. Wickerson**

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, electronic copies of my final year project report to both Blackboard and the EEE coursework submission system.

I affirm that the two copies of the report are identical.

I affirm that I have provided explicit references for all material in my Final Report which is not authored by me and represented as my own work.

Acknowledgements

I'd like to thank my final year project supervisor, Dr David Thomas, for providing me with advice and feedback throughout this project. I'd also like to thank Johnny Beaumont for providing me with guidance on how to use the POETS hardware.

I'd like to thank my friends and my flatmates for making my time studying at Imperial much more fun and enjoyable. I cannot imagine a better group of people to have spent this long degree with.

I'd like to thank my family for being around to support me throughout my life. Without them I wouldn't be the person I am today.

I'd like to thank Catherine Lincoln for being by my side no matter what and always being there to cheer me up even when things were hard. Without you I would never have survived my time at Imperial, you're the best!

Abstract

This project investigated the viability and possible performance gains of running spiking neural networks on the high performance and asynchronous POETS (Partially-Ordered Event-Driven System). In asynchronous applications, POETS can provide orders of magnitude more performance than conventional architectures and due to the asynchronous nature of spiking neural networks, it was hypothesised that there may be performance increases when running them on POETS. The project involved: assessing the viability of synchronisation algorithms; implementing methods for users to easily create spiking neural network models; and evaluating the performance and accuracy of the different algorithms run on both POETS and a commonly used spiking neural network simulator that runs on conventional hardware, known as BRIAN. Of the synchronisation algorithms, the best were the algorithms that were explicitly clocked. BRIAN consistently provided better performance at the network scales tested in this project. Larger scale networks couldn't be tested due to memory constraints for both POETS and BRIAN.

Contents

1 Project Specification

This project investigates the viability and possible performance gains of running spiking neural networks on POETS[1] (Partially-Ordered Event-Driven System) vs running them using simulators developed for more conventional computing architectures.

1.1 Motivation

Spiking neural networks are a type of artificial neural network that closely mimic the structure of the brain[2]. Unlike normal artificial neural networks, the neurons only fire when a membrane potential reaches a specific value. They can use the timing of spikes that move between neurons to encode their data and learn by modifying the weights of each synapse according to a temporal correlation between spiking events.

POETS is a hardware platform that provides 50000 threads connected by a high speed asynchronous network[3]. It uses a large number of small cores connected in a fast parallel network. The idea is that a problem can be broken down and into a large set of interacting devices. These devices communicate using small packages called messages. For highly asynchronous and graph-based problems, there is evidence that POETS can provide up to a two orders of magnitude increase in performance compared to conventional techniques[3].

Spiking neural networks are suited to the POETS platform due to their asynchronous and event-driven nature and the fact that each node isn't required to do computationally heavy tasks. If a system such as POETS could be used for spiking neural networks, the performance compared to conventional CPUs could vastly increase their viability. This will require new techniques for designing networks for POETS since it is currently very complex and doesn't have the ease of use that simulators designed for normal researches have[4].

1.2 Objectives

The objective for this project was to determine whether running spiking neural networks on POETS was a viable alternative to using simulators on conventional computing architectures.

It was hypothesised that POETS could provide an increase in performance over simulators being run on conventional hardware. Based on prior research papers, POETS can provide two orders of magnitude more performance increase in some applications[3]. This hypothesis would be proven valid if POETS provided a 10x decrease in simulation time. This was chosen because it would drastically reduce the time needed to carry out experiments. Simulating a large network using BRIAN can take days, whereas with a 10x increase in speed it could be done in less than a day, thus changing the way that researchers do their work.

The process of validating the hypothesis was broken into several parts:

- Analysing the different methods used for synchronising spiking neural network differential equations and determining how this affects their asymptotic performance.
- Developing a simple way of generating spiking neural networks for POETS. This generator would have ease of use comparable to BRIAN.
- Modifying the existing algorithms that are currently used for generating spiking neural networks on POETS, with the aim of improving performance.
- Determining whether the models that will be run on hardware are accurate at simulating SNNs, and then evaluating the performance of these models on hardware.

1.3 Overview of deliverables and contributions

The objectives were investigated and met through several deliverables and contributions:

- **Asymptotic analysis of algorithms.** The first contribution was an analysis of the different algorithms that could be used to simulate spiking neural networks. The main point of this contribution was to compare how different synchronisation algorithms might affect the performance of the software being run on POETS. This involved writing up a performance analysis of the asymptotic performance of the possible algorithms. Metrics used included number of messages sent and the time required per spike. The asymptotic performance provides an insight into how well the algorithms will perform when simulating numbers of neurons and synapses that cannot currently be simulated using this version of POETS. Despite being written with POETS in mind, the information gathered in this section will be applicable for future projects investigating similar hypotheses even if they don't use the current version of the POETS platform.
- **Software for generating networks.** Another deliverable was creating a method for easily generating spiking neural networks on POETS. Current spiking neural network simulators built for conventional computing platforms allow the user to simply specify spiking neural networks, whereas the spiking neural network simulators that exist for POETS are hard-coded and only allow for changing the number of neurons, their values, and how they are connected. There are no simple functions or libraries that allow a user to specify custom networks in a high level language such as Python. If a user wants to design new spiking neural networks and use them on POETS, they must write their own graph types and instances in XML which is time consuming and difficult. In comparison, programs such as BRIAN are easily used by both proficient programmers as well as scientists and students[4]. There is no such software for easily creating spiking neural networks on POETS. For POETS to be a viable alternative to BRIAN, there needed to be a better way of designing and generating networks for the platform.

- **An evaluation of spiking neural network performance on POETS hardware.** Whilst it's possible that POETS could be much faster than current spiking neural network simulator implementations due to the asynchronous nature of SNNs and POETS' specific capabilities when it comes to asynchronous problems, not much work had been done to evaluate whether this is the case. The implementations of spiking neural networks currently available for use on POETS were designed to investigate whether the messaging systems worked, not how the networks performed. Implementations of the spiking neural network simulator algorithms were tested on POETS hardware and then compared against a conventional spiking simulator (BRIAN) in areas such as: accuracy; ratio of real time needed against simulated time; how performance changes as the network is scaled up; and overheads in generating the network. This contribution will be useful in the future if similar projects are run. They will be able to easily compare performance with POETS by following the experiments laid out during this project.

- **An analysis of the accuracy of the models.** A final contribution/deliverable was an analysis of the accuracy of the spiking neural network algorithms. The hardware simulations were compared against mathematical models to determine whether the results gathered in the hardware evaluation section were valid. When differences in these models arose, they were investigated and the causes for these differences were analysed and explained.

The deliverables and contributions presented in this section are a very brief overview of the work done in this project. They are expanded upon greatly in the various deliverable sections further on in the report.

2 Background

The spiking neural networks (SNNs) that make up the backbone of this project are very different from conventional artificial neural networks (ANNs) as SNNs more closely mimic brains[2]. This section will provide a brief overview of the various technologies related to this project.

2.1 Artificial neural networks

Artificial neural networks are a collection of layers of nodes called artificial neurons, each with inputs connected to the previous layer and outputs connected to the next layer in a directed weighted graph[5]. Each neuron contains a non linear activation function that is applied to the sum of all of its inputs with the output being sent forward. A simple example of this can be seen in Figure 1.

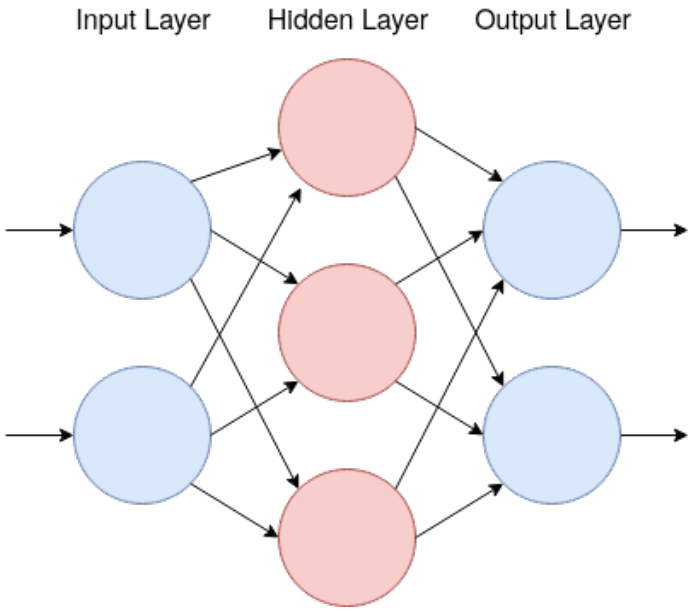

Figure 1: A simple example artificial neural network showing how layers are connected.

Each connection between nodes has an associated weight and any inputs to this edge are multiplied by the weight, either increasing or decreasing the strength of the signal. They are trained by comparing known outputs to the outputs of the network and calculating the error between the output of the network and the known output. A process called backpropogation[6] is then used to determine the contribution of each weight to the overall error and the weight is adjusted accordingly. If a particular weight contributes a lot to the error then it will be changed more than one that contributes less.

Over many thousands of iterations of this process, an artificial neural network can be trained to do a task, such as classification, very well. They can prove useful in a large range of tasks including: computer vision, speech recognition, playing board games[7], spam filtering, image classification, medical diagnosis[8], and many more. Artificial neural networks have had a lot of research put into them and there are many different types such as fully connected, convolutional, and recurrent neural networks. Most of these networks are still made using layers of nodes though and are all synchronous.

2.2 Spiking neural networks

Spiking neural networks are a type of artificial neural network whose structure is much more similar to that of the brain[2]. Whilst originally artificial neural networks were designed to solve problems in the same way that a brain would, over time attention started to focus on making them perform well at specific tasks. This has led to deviations from biology. On the other hand, spiking neural networks are much more similar to biological neural networks. Instead of all neurons firing each propagation cycle, the neurons only fire once their membrane potential reaches a specific threshold value meaning every neuron doesn't fire every time step, this is shown in Figure 2. Similar to a normal artificial neural network, spiking neural networks are connected by many edges with corresponding weights. When signals travel along an edge to a neuron, its membrane potential increases according to the weight of the edge. The state of the neuron is represented as a differential equation that changes over time. This means that instead of regularly firing in a synchronous fashion, the entire network is asynchronous.

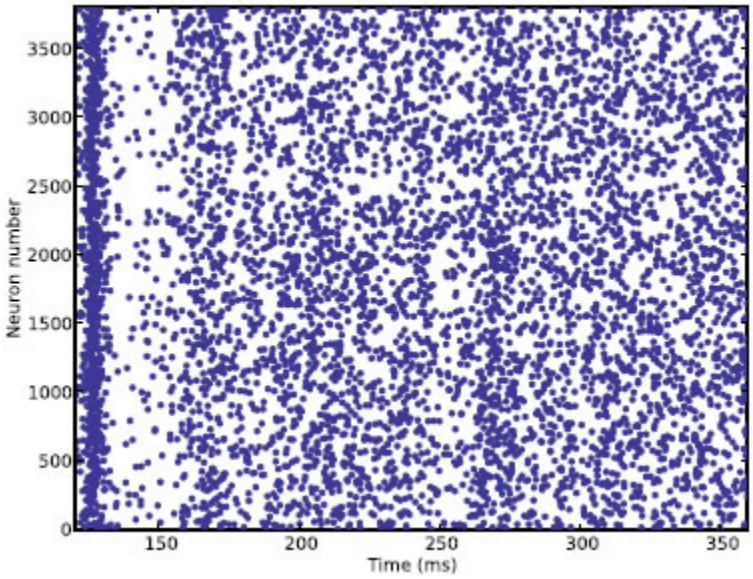

Figure 2: The output from a spiking neural network[4]. It can easily be seen that neuron don't spike every time step, and there are times when neuron spikes are more synchronised.

The asynchronous nature of the neural networks means that information can be encoded using temporal coding. In normal rate coding, information is encoded in the firing rate of a neuron. In temporal coding, information is contained in the fluctuations in the firing rate of a neuron. This allows a single spiking neuron to replace hundreds of hidden neurons in a fully connected network[2].

Similarly to artificial neural networks, spiking neural networks can learn. They cannot use backpropagation due to the use of binary pulses that are not differentiable. They instead use unsupervised learning techniques, two popular ones being Hebbian learning and spike-timing-dependent plasticity.

- **Hebbian learning** is based around the idea that "neurons that fire together wire together"[9]. In practice, this involves increasing the weights between two neurons if the two neurons fire simultaneously, whilst decreasing the weight if they fire separately.
- **Spike-dependent-timing plasticity** (SDTP) is a process that adjusts the weights based on the relative timing of a neurons output and input spikes[10][11]. If an input spike to a neuron occurs immediately before that neuron fires a spike, then that particular weight is increased, whereas if an input spike occurs on average just after the neuron has fired a spike, the weight is decreased. This results in inputs that might be the cause of the a neuron firing are made more likely to contribute in the future whereas neurons that are not the cause will be less likely to contribute. The learning involves a lot of weights becoming zero leaving a subset of the initial weights that are deemed important.

2.3 POETS

POETS (Partial-Ordered Event-Triggered Systems) is an infrastructure based on the idea of having a very large number of small, weak cores[1]. All the cores are connected together in a extremely fast, parallel mesh with inter-core communication effected by small hardware data packets called messages. The process of implementing a program to be run on POETS involves creating a fixed graph representing a problem that can be easily mapped to the processor mesh.

Architecturally, POETS is similar to its predecessor, the SpiNNaker[12] system. POETS consists of tens of thousands of hyper-threaded RISC-V cores that are realised on banks of FPGAs[3]. Each thread has 1MB of high bandwidth off-chip DRAM. Due to the method of caching used by POETS, there is no aliasing between threads and data hazards are eliminated. The threads are connected via a mailbox that stores both incoming and outgoing messages. Several cores can use one mailbox and these are connected with a raw link and a 10 Gbps Ethernet connection containg hardware that automatically detects and drops packets containing errors. On top of this is a POETS specific reliability layer for re-transmitting packets if they are dropped.

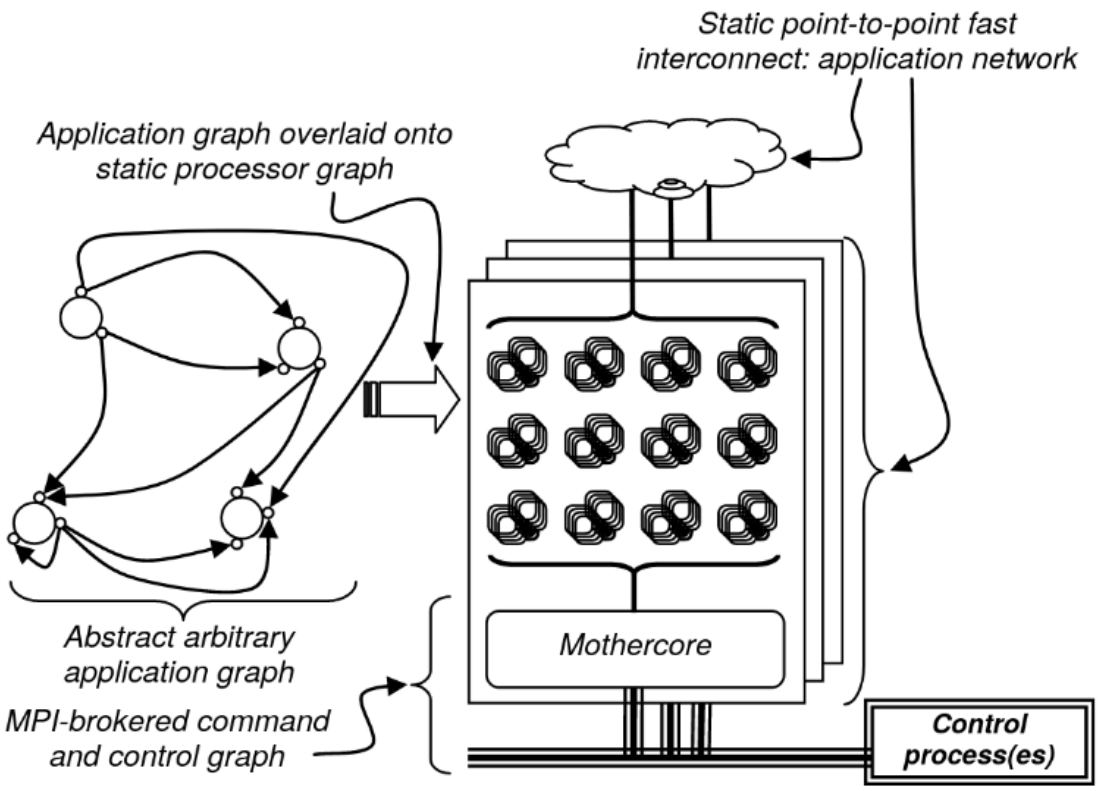

Figure 3: POETS overview showing how a network is mapped to POETS hardware[3].

The software for POETS is controlled by a multi-process program called the Orchestrator[3]. The main tasks this program carries out are: application graph construction - where a model corresponding to the domain specific graph is created for POETS to interpret; application model mapping - where the graph vertices in the model are mapped to the hardware cores such that the overall communication cost is minimised; controlling data in and out - thus allowing interaction with the outside world as seen in Figure 3.

Due to the nature of spiking neural networks being an asynchronous graph of neurons, they are particularly suited to the POETS platform. It is hypothesised that there could be a huge performance increase by using POETS to simulate spiking neural networks as opposed to using conventional computers with software such as BRIAN[4].

3 Related Work

There are several projects that are related to the work that has been done during this project. Of these the most relevant are: the popular SNN simulator BRIAN; the precursor to POETS, SpiNNaker; and the various SNN implementations currently available on POETS.

3.1 BRIAN Simulator

BRIAN is an open source spiking neural network simulator written in Python[4]. It is designed to be easy to learn and use for people whose background isn't in this area of mathematics. For example, this is all that is needed to create, run, and plot a spiking neural network with BRIAN:

```
from brian2 import *

G = NeuronGroup(100, "dv/dt=(1-v)/tau:1",
    threshold="v>=30*mV",
    reset="v=-50*mV")

S = Synapses(G, G, "w:volt", on_pre="v+=w")

spikes = SpikeMonitor(G)
run(1000)

figure()
scatter(spikes.t/ms, spikes.i)
show()
```

The user is given a simple way to specify all the parameters and equations they would like to use. They can name them whatever they want, and aren't limited to preset network types. The syntax also allows for the specification of units such as volts and amperes so that the models can accurately represent real potentials that might occur in a brain, although this is mainly syntactic sugar for researchers and isn't required if the networks are used for data processing. BRIAN also allows learning through techniques like SDTP.

This simple approach to designing neural networks means that researches and students who don't have a software engineering background can easily produce results without having to spend a lot of time learning the syntax. This is bolstered by the fact that it is written for use in Python, a language very popular among students and scientists for data processing. It can therefore link in with many hundreds of packages already available for Python.

BRIAN also has good accuracy and performance when it comes to simulating SNNs[4]. It will, therefore, be used as a baseline that POETS will be compared against. Whilst BRIAN is designed for desktop computers rather than dedicated hardware systems such as POETS, it will still provide a good benchmark. Previous projects have attempted to use BRIAN to POETS[13], but this won't be used.

3.2 SpiNNaker

SpiNNaker (Spiking Neural Network Architecture) was the predecessor to POETS. It is a supercomputer architecture featuring 57,600 ARM9 processors, each with 18 cores and 128MB of SDRAM totalling over 1,000,000 cores. It's primarily designed for simulating Spiking Neural Networks as part of the Human Brain Project[14].

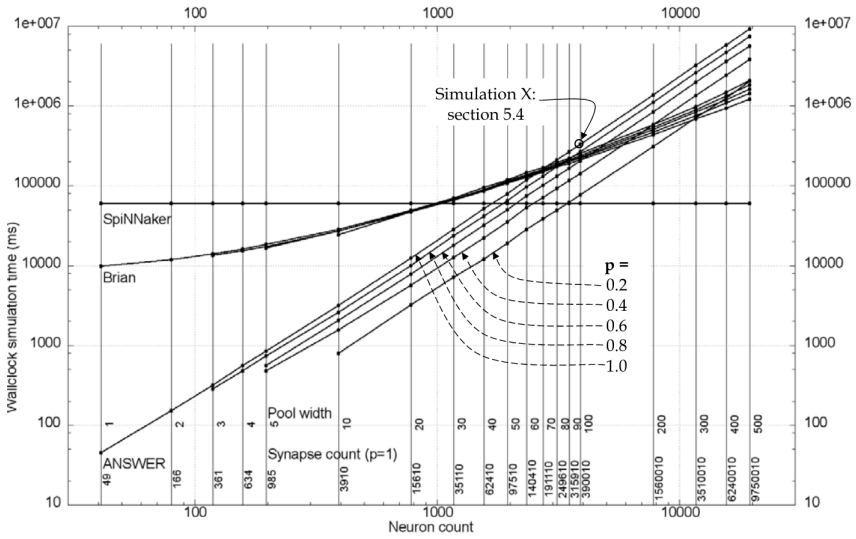

Figure 4: SpiNNaker performance compared to several simulators including BRIAN[12]. It can be seen that SpiNNaker's simulation time isn't dependent on neuron count.

SpiNNaker is made up of triangularly connected nodes, mapped onto a toroid[12]. It is scalable from 1 node to 65532 nodes (that limit due to a 16-bit identifier). The system, like POETS, is event-driven. There is no global clock and computation is done on the arrival of a packet to that node. When a node executes code, it has the opportunity to send messages before going to sleep. Most of the cores should spend most of their time asleep.

SpiNNaker has been compared to BRIAN and its accuracy is extremely similar[12]. For small neuron counts (sub 1000 neurons), BRIAN is faster than SpiNNaker. SpiNNakers simulation time is constant and the overhead doesn't increase the time taken for a simulation when the number of neurons is increased. This can be seen in Figure 4. SpiNNaker is orders of magnitude faster than desktop computers when running significantly large neuron counts.

A disadvantage of SpiNNaker is that it has no floating-point arithmetic meaning it is difficult to compare to floating-point simulators like BRIAN.

3.3 Current POETS spiking neural network implementations

This project is not the first project to develop spiking neural networks for POETS. The graphschema repository contains three versions of the Izikevich Neural Network[15]. This type of network is based around reproducing behavior of types of neurons in the outer layer of the cerebrum. Where v and u are membrane potentials and a, b, c, d are user defined constants, the Izhikevich model is defined as:

$$
\begin{aligned}
&v' = 0.04v^2 + 5v + 140 - u + I \\
&u' = a(bv - u) \\
&\text{if } v \geq 30mV \\
&\text{then } v \leftarrow c,\ u \leftarrow u + d
\end{aligned}
$$

The three versions of Izikevich available in the graphschema repository are:

- GALS Izikevich
- Clocked Izikevich
- Barrier Izikevich Clustered

3.3.1 GALS Izikevich

GALS stands for Globally Asynchronous Locally Asynchronous. It is an architecture that allows a collection of locally clocked modules to communicate with each other. For the GALS Izikevich implementation, a neuron can only progress to its next time step if it has received a number of messages equal to the number of inputs connected to it. The incoming fires are then stored in a list. Despite this, the neuron can fire at any time if it's internal variables are above their threshold no matter whether it has received all the incoming fires.

This technique doesn't require a global clock and is therefore reasonably good for a POETS implementation. Unlike other GALS methods, it doesn't require bi-directional connections. An issue with this is that some neurons can get ahead and have to wait for their neighbours to fire to them. The worst case scenario for this is the length of the maximum cycle in the network.

There are issues with GALS implementations on POETS. For example, in a proper GALS implementation, the lists that store the incoming fires are FIFOs (first in - first out), but due to the nature of POETS, the system actually uses FIRO (first in - random out). This means

that sometimes, a neuron won't get the correct input at the correct time which can reduce the accuracy of the network. Also, the maximum number of cycles (max_t) will be reached when a neuron's internal time equals max_t. Since most neurons don't fire every cycle, it is very hard to stop the network at a specific time step since there are no time steps to speak of at a network level, only at the neuron level.

3.3.2 Clocked Izikevich

An issue with comparing POETS simulations to simulators like BRIAN is that BRIAN is a clocked simulator while POETS is not. There is a provided graph type in the graphschema repository that adds a clock neuron to the network. For the clocked Izikevich implementation, each neuron also has two extra connections as well as any connections to other neurons as well as two connections to a single clock device seen in Figure 5 below:

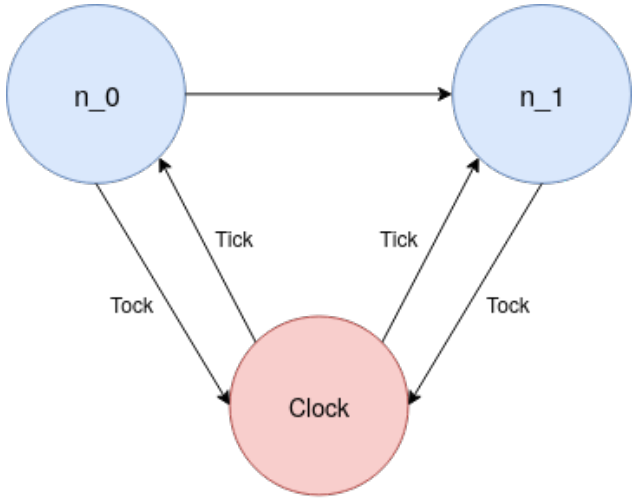

Figure 5: An illustration of how a clock is set up in a simple network. When a neuron fires, messages are sent to all the neurons before the network can proceed.

Each iteration, if a neuron receives an input it sends a *tock* message to a clock device. This clock then sends a *tick* message to every neuron to update each of the neuron's internal clocks. This acts as a global clock and keeps the network synchronised. This avoids the issue where a neuron is many milliseconds behind another neuron and allows the network to act more like a simulator on a platform like BRIAN.

This comes with its own issues however since it vastly increases the overhead. Each time step takes 3 times as long since: the clock has to have a message sent to it, the clock has to send a message to all the other neurons, and the neurons that have reached their threshold potentials have to fire. This means that the overall simulation will take longer. Another issue is that the asynchronous nature of POETS is wasted since, whilst the system is technically asynchronous as not all neurons will be activated each cycle, the overall system has been made into a synchronous one as each neuron receive their clock tick. This goes against the nature of event-driven systems.

3.3.3 Barrier Izikevich Clustered

The final type of Izikevich spiking neural network available in the graphschema repository is a prototype design called a Barrier Clustered Izikevich network. This technique makes assumptions that: the networks are randomly connected; have one billion synapses and one million neurons; the spiking level is between 1% and 10%; there are 25 FPGAs with 1000 threads each; and that the simulation functions well with latency. This causes the implications: each FPGA manages 40000 neurons meaning each thread manages 40 neurons; and each spike fans out to 1000 other neurons over 40 FPGAs. This is an issue because each spike requires 1000 messages which can cause congestion. Most of the messages will also be sent to different FPGAs which is non ideal as cross-FPGA connections are much slower than connections within the FPGAs.

The solution to this is to create clusters of neurons for each FPGA. Each cluster has repeaters connected to a subset of the neurons in the cluster. When a neuron spikes, instead of firing directly to other neurons, they fire to one repeater on every FPGA. This repeater then sends the message to the neurons on its cluster. Advantages of this method are that it reduces the number of messages sent by each neuron to around 40, vastly reducing the number of cross FPGA messages, thus reducing message congestion. Disadvantages to this method are that there is an increase in latency. Due to the repeaters, spikes that are generated at time t will have an effect during time $t + 2$. Also, depending on the number of repeaters, there can be an increase in the number of output ports needed.

4 Deliverable 1: Asymptotic analysis of various synchronisation models

The purpose of this deliverable is to assess the asymptotic performance of the various methods used to synchronise spiking neural networks on POETS. This section is split into five synchronisation models that try to maximise accuracy and two models that sacrifice accuracy for performance. In each analysis, the neuron count is increased whilst the number of synapses is kept the same. The number of synapses per neuron is kept fixed because it provides clearer information and is more similar to an actual brain[16].

POETS is designed for entirely asynchronous tasks. This is in conflict with spiking neural networks that are synchronous due to their temporal nature. The neurons must remain in sync with other neurons so that no neuron can be too far ahead of the rest of the network. For example, it is undesirable for a neuron at time step 100 to affect another neuron that is only at time step 20. Therefore, synchronisation methods are extremely important for simulating spiking neural networks on POETS.

Despite the performance capabilities of POETS being extremely high with the ability to simulate hundreds of thousands of neurons with millions of synapses, this doesn't come close to the tens of billions of neurons in a human brain. The networks that will be simulated on hardware are minuscule compared to an actual brain. In this section we will predict the performance characteristics of the various synchronisation techniques for networks that cannot realistically be simulated.

This is a useful investigation since some synchronisation techniques might perform well on POETS when only simulating millions of neurons, but they may fail to perform well if POETS is ever scaled to hundreds on millions of neurons. This outcome would not be seen by running them on current hardware.

4.1 Accuracy based SNN synchronisation methods

The synchronisation methods used in this section prioritise synchronising neurons for accuracy. The approaches in these methods are specific to the POETS ecosystem and vary in their complexity and added overheads. The various synchronisation methods are:

- Globally Asynchronous Locally Synchronous (GALS)
- Clocked using a specific clock neuron
- Cluster Based
- Clocked using the Hardware-Barrier feature of POETS
- Relaxed GALS.

4.1.1 Globally asynchronous locally synchronous model

In the Globally asynchronous locally synchronous (GALS) model, when a neuron receives an incoming spike, it sends a message to all of its neighbours telling them whether it spiked or not. The neuron can then only progress to its next time step if it has received a number of messages equal to the number of inputs connected to it, with the incoming fires are stored in a list. This is used to synchronise the neurons.

Algorithm 1: GALS spiking neural networks

```
Initialise neurons;
while time < time_max do
    i ← 0 ;
    while i ≤ Number of input edges do
        if Received input from edge then
            i ← i + 1;
        end
        if i = Number of input edges then
            Update neurons;
        end
    end
    if voltage ≥ voltage_thr then
        Reset neurons;
        Message neighbours with has_spiked* = true;
    end
    Message neighbours with has_spiked = false;
end
```

GALS asymptotic analysis

For GALS, the number of messages in one simulation can be seen in Equation 1 whilst the time required for one time step can be seen in Equation 2. Figure 6 shows the effect the neuron count has on the number of messages sent in one simulation.

$$m \;=\; n \cdot d \cdot T \tag{1}$$

$$t = \max{}_{0 \leq i \leq N} \left(t_{i_{synapse}} + t_{i_{return}} \right) \tag{2}$$

where:

m = Number of messages $\qquad$ n = Number of neurons

d = Degree $\qquad$ T = Number of time steps

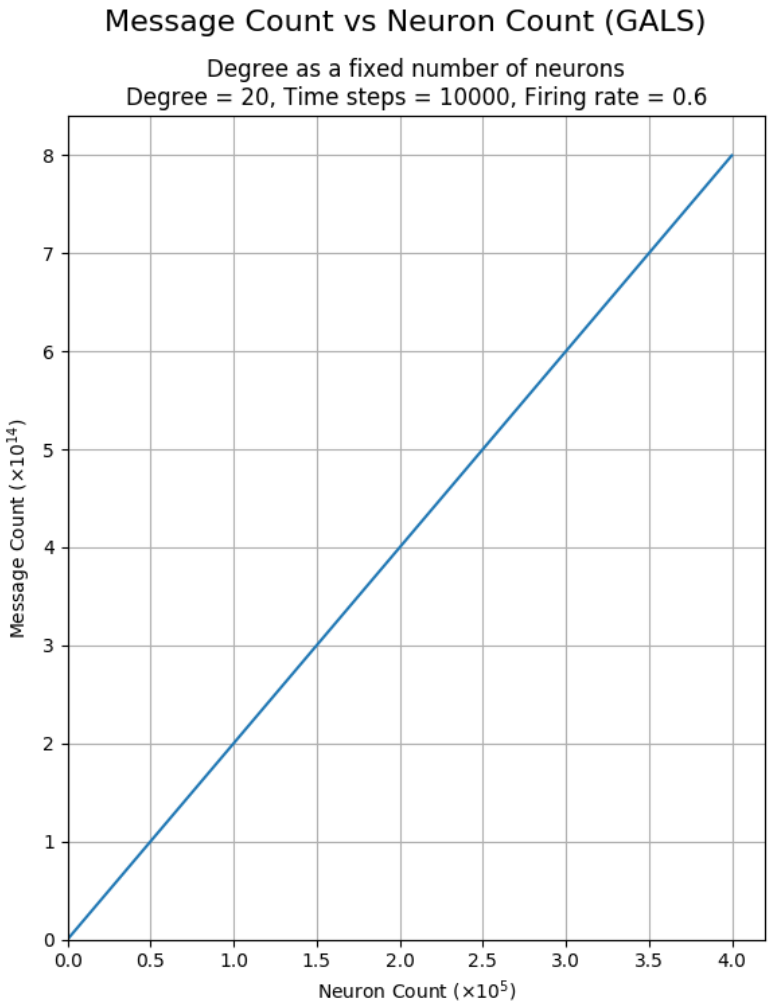

Figure 6: Plot of the GALS asymptotic analysis. It can be seen that the relationship between messages and neuron count is linear.

GALS discussion

GALS should perform reasonably well as can be seen in Figure 6. It doesn't add any extra neurons like the clocked version does but it runs the risk of being slow since messages have to be returned from a neuron's neighbours at every time step. Since the neurons don't have a two way connection between each other, these messages have to travel throughout the network to return to the original neuron. In some cases this is fine, but in certain cases, the message may have to travel extremely far as seen in Figure 7.

A GALS spiking neural network's neurons should maintain synchronisation fairly well. Since a neuron needs to receive a message from all of its input neighbours at time t before moving on to time $t+1$. This means that the neighbouring input neurons cannot be more than one time step behind, although neighbouring neurons can be ahead if they don't share inputs.

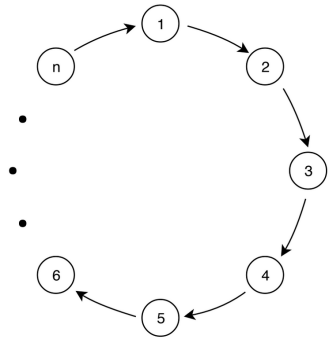

Figure 7: A worst case GALS network where messages must traverse the entire network in order to return to the original neuron[13].

4.1.2 Model clocked with a clock neuron

Each iteration, if a neuron receives an input it sends a tock-message to a specialised clock neuron. This clock then sends a tick-message to every neuron to update each of the neuron's internal clocks. Each neuron must then send another tock-message back to the clock before the network is can continue. This acts as a global clock and keeps the network synchronised.

Algorithm 2: Clocked spiking neural networks

```
Initialise neurons;
while time < time_max do
    if Received input from non-clock edge then
        Update neurons;
        Send tock_1 to clock neuron;
        Clock sends tick to all neurons (to increment clock);
        i ← 0;
        while i < Number of neurons do
            if Clock received tock_2 from neuron_i then
                i ← i + 1;
            end
        end
        Send tick to initial neuron;
    end
    if voltage ≥ voltage_thr then
        Reset neurons;
        Message neighbours with has_spiked = true;
    end
    if Received tick from clock then
        time ← time + 1 ;
        Send tock_2 to clock;
    end
end
```

Clocked asymptotic analysis

For the clocked network, the number of messages in a single run of the simulation can be seen in Equation 3 and the time for one time step in Equation 4. Figure 8 shows how the neuron count affects the number of messages sent in one simulation.

$$m = n \cdot (1 + 2 \cdot n + d) \cdot r \cdot T \tag{3}$$

$$t = \max_{0 \leq i \leq N} (t_{i_{tock}} + \max_{0 \leq i \leq N} (t_{i_{synapse}} + t_{i_{tick}})) \tag{4}$$

where:

m = Number of messages $\quad$ n = Number of neurons

r = Probability of neuron firing $\quad$ d = Degree $\quad$ T = Number of time steps

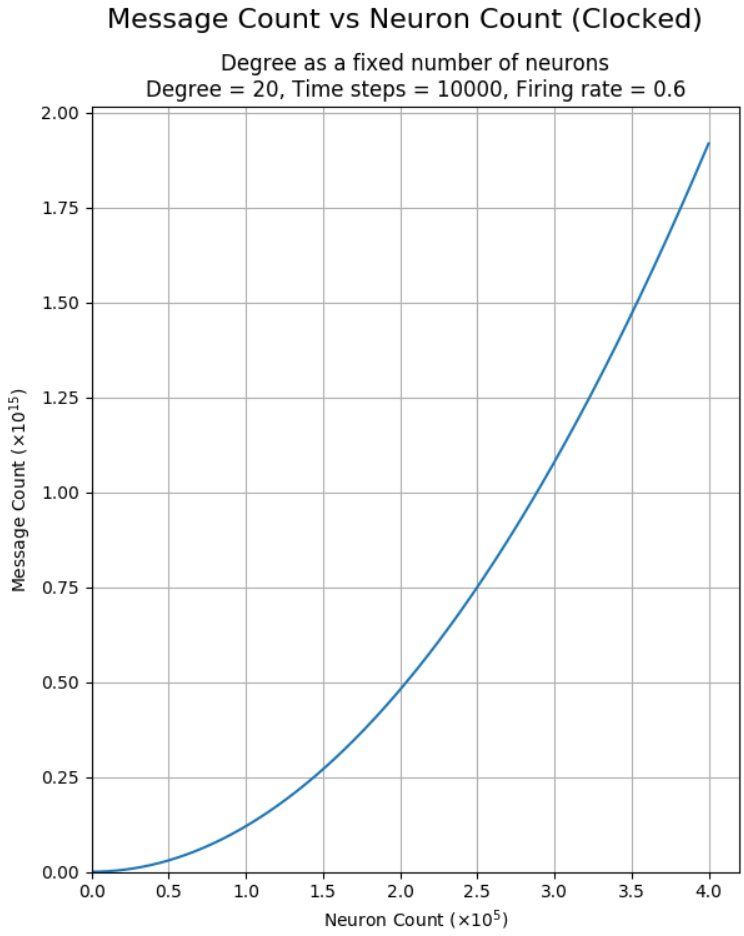

Figure 8: Plot of the clocked asymptotic analysis. Unlike the other networks, the relationship between messages and neuron count is non-linear. This is due to the clock neuron messages.

Clocked model discussion

Networks synchronised using a clock neuron are by far the most complex of the networks. The number of messages increases with $O(n^2)$ as the number of neurons is increased as seen in Figure 8. The issue arises from the fact that every time a neuron fires, it has to send a message to the clock that in turn has to send a message to all other neurons, which then have to send a message back to the clock (Figure 5).

Whilst complex, this network is very likely to be accurate and comparable to BRIAN since it is clocked. It should have reasonable time performance since the synchronisation goes directly to the clock neuron without having to propagate through the network as in GALS. Whilst some of the messages have to travel across FPGAs, the time each message is floating in the network should be relatively constant. An issue with this network type is that it defeats the purpose of POETS since the network is no longer event-driven, it is clocked.

4.1.3 Clustered spiking neural networks

In the clustered model, clusters of neurons are created for each FPGA. Each cluster has repeaters connected to a subset of the neurons in the cluster. When a neuron spikes, instead of firing directly to other neurons, they fire to one repeater which then messages repeaters on every other FPGA. This repeater then sends the message to the neurons on its cluster. This technique leverages the intricacies of the POETS hardware and reduces congestion between the FPGAs. It is not currently proven to work, since there is no way to make sure that specific neurons are put on specific FPGAs.

Algorithm 3: Clustered spiking neural networks

```
Group together neurons that are most interconnected prior to simulation start;
all_repeaters = {repeater_0, ... , repeater_N};
Initialise neurons;
while time < time_max do
    if Received input then
        Update neurons;
    end
    if voltage ≥ voltage_thr then
        Reset neurons;
        repeater_i ← repeater for current cluster;
        other_repeaters ← all_repeaters \ {repeater_i} ;
        Send has_spiked messaged to repeater_i;
        repeater_i sends message to other_repeaters;
        other_repeaters send spikes to the neurons in their clusters;
    end
end
```

Clustered SNN asymptotic analysis

For the clustered network, the predicted number of messages in a simulation are shown in Equation 5 whilst the time required to complete one time step is seen in Equation 6. Figure 9 shows the effect the neuron count has on the number of messages sent in one simulation.

$$m = n \cdot (1 + c + d) \cdot r \cdot T \tag{5}$$

$$t = \max_{0 \leq i \leq N} \left(t_{i_{toRepeater}} + \max_{0 \leq i \leq N} \left(t_{i_{betweenFPGA}}\right) + \max_{0 \leq i \leq N} \left(t_{i_{fromRepeater}}\right)\right) \tag{6}$$

where:

m = Number of messages $\quad$ n = Number of neurons $\quad$ d = Degree

r = Probability of neuron firing $\quad$ c = Number of clusters $\quad$ T = Number of time steps

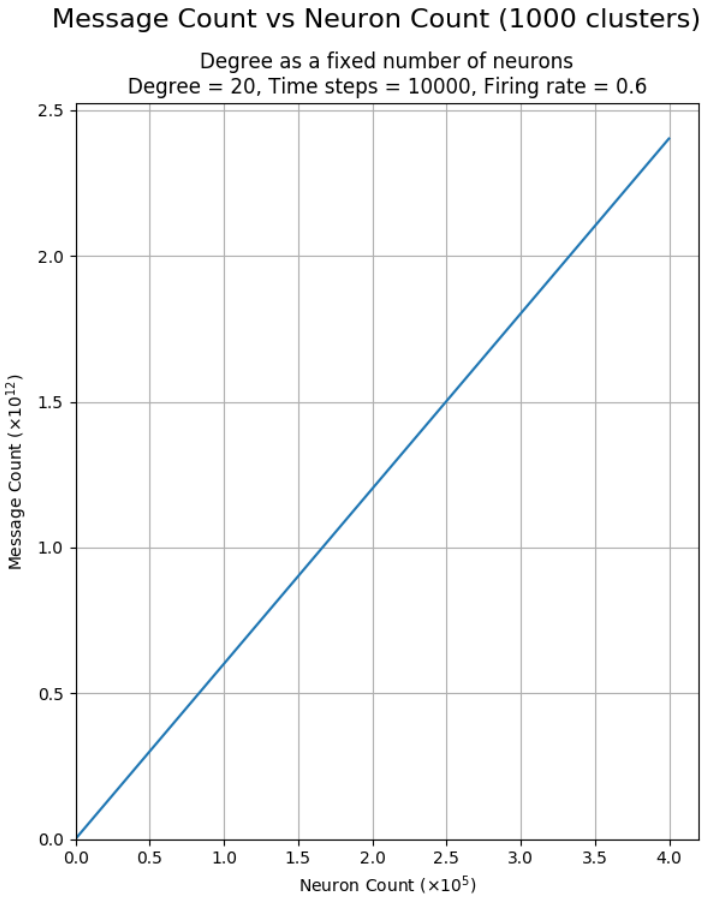

Figure 9: Plot of the clustered network asymptotic analysis with 1000 clusters. The relationship between messages and neuron count is linear, but due to the use of repeaters, significantly more messages are sent. It should be noted that few of these messages will be cross-FPGA so performance won't be as bad as it looks.

Clustered SNN discussion

Clustered networks slightly increase the number of neurons needed since the neurons must be connected to repeaters, however, they don't increase the number of synapses greatly since the connections to the repeaters replace the cross-FPGA connections. The clusters drastically reduce the number of cross-FPGA connections meaning that the number of slower connections is reduced thus increasing the performance of the network. The effect of this is seen in Figure 9.

A significant problem with this network is that it currently is not possible to specify which FPGA a neuron goes to which makes clustering neurons together extremely difficult. Therefore this network isn't feasible with the current version of POETS.

4.1.4 Hardware-Barrier clocked

Hardware barrier clocked networks utilise the hardware idle of the POETS system to synchronise the neurons. The POETS system has a feature known as hardware idle which, when a certain amount of time has passed with no activity on the network, ends the current simulation. This feature can be manipulated to carry out more work when the system reaches hardware idle. Barrier clocked networks use this to create a form of global clock. After all messages for one time step have been sent, the system waits until hardware idle starts. When it starts to idle, the network boots back up again and continues to the next time step. By doing this the hardware idle keeps the neurons of the network in sync, similar to the clocked network type.

Algorithm 4: Hardware-Barrier spiking neural networks

```
Initialise neurons;
while time < time_max do
    if Received input then
        Update neurons;
    end
    if Hardware Idle then
        if voltage ≥ voltage_thr then
            Reset neurons;
            Message neighbours with has_spiked = true;
        end
    end
end
```

Hardware-Barrier asymptotic analysis

For the hardware barrier model, the number of messages in one simulation can be seen in Equation 7 whilst the time required for one time step can be seen in Equation 8. Figure 10 shows the effect the neuron count has on the number of messages sent in one simulation.

$$m = n \cdot d \cdot r \cdot T \tag{7}$$

$$t = \max_{0 \leq i \leq N} (t_{i_{synapse}}) + k_{barrier} \tag{8}$$

where:

m = Number of messages $\quad$ n = Number of neurons

d = Degree $\quad$ $k_{barrier}$ = Time before hardware idle

r = Probability of neuron firing $\quad$ T = Number of time steps

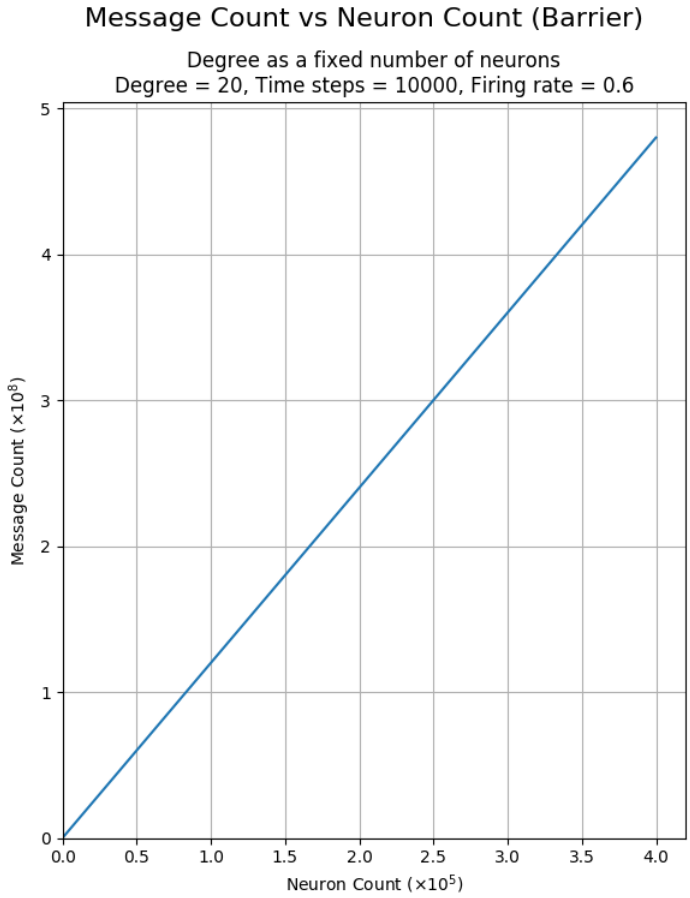

Figure 10: Plot of the barrier Clocked asymptotic analysis. There are no added devices so the relationship between messages and neuron count is linear.

Hardware-Barrier discussion

The barrier model is the model most comparable to BRIAN because it is clocked, yet doesn't add any new synapses or rely on messages being sent to synchronise the neurons. The entire POETS system becomes a clocked system which is good for performance and accuracy since the differential equations in the neurons can be updated correctly and no neuron will ever be out of time with another. It is significantly better than the clocked network in this regard as there is no added overhead in terms of messages, resulting in the performance seen in Figure 10. It does have a small overhead where the network has to be off in order to start the hardware idle, but this is negligible. Unfortunately, the fact it is clocked ruins the asynchronous event-driven nature of POETS.

4.1.5 Relaxed GALS model

Relaxed GALS is similar to normal GALS except it doesn't wait to receive a message from all its input edges before carrying on. This is because it is likely that a neuron receives most of the edge messages quickly and only has a small fraction take a long time to be received. The network in normal GALS has to wait for these neurons to catch up.

Algorithm 5: Relaxed GALS spiking neural networks

```
Initialise neurons;
Set relaxation value r; e.g. r = 0.7 means only wait for 70% of input messages.
 while time < time_max do
    i ← 0 ;
    while i ≤ Number of input edges do
        if Received input from edge then
            i ← i + 1;
        end
        if i = ⌊Number of input edges · r⌋ then
            Update neurons;
            Break;
        end
    end
    if voltage ≥ voltage_thr then
        Reset neurons;
        Message neighbours with has_spiked = true;
    end
    Message neighbours with has_spiked = false;
end
```

Relaxed GALS asymptotic analysis

For relaxed GALS, the performance is similar to standard GALS. This is seen in Equation 9 and Equation 10. Figure 11 shows the how the neuron count will affect the number of messages sent in one simulation.

$$m = n \cdot d \cdot T \tag{9}$$

$$t = \max_{0 \leq i \leq p \cdot N} \left(t_{i_{synapse}} + t_{i_{return}} \right) \tag{10}$$

where:

m = Number of messages $\quad$ n = Number of neurons

d = Degree $\quad$ p = Relaxation value $\quad$ T = Number of time steps

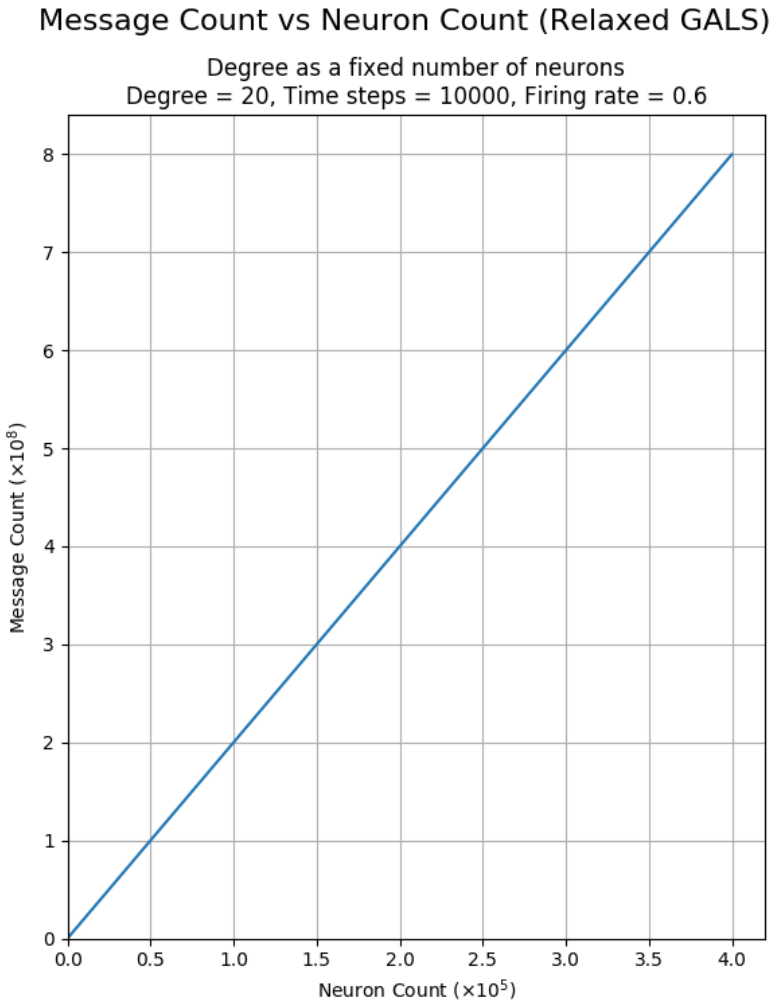

Figure 11: Plot of the relaxed GALS asymptotic analysis. It can be seen that the relationship between messages and neuron count is linear. This is expected since the network is almost identical to the normal GALS network.

Relaxed GALS discussion

This network performs very similarly to normal GALS (Figure 11). The same number of messages will be sent as in GALS, but the network will continue earlier than it would if it was a normal GALS.

It does sacrifice accuracy as it ignores the slowest messages. This may not matter since the most important and linked neurons should escape the culling that the relaxation causes. It is hypothesised that the neurons which take the longest to return their messages will be the same every time step and therefore it will only be those few neurons that are ignored. These neurons will be less important to the running of the network, meaning the accuracy of the network won't be drastically affected. The effect of this is that the synapses whose neuron has to send messages a long way, won't exist.

4.2 Extreme parameter relaxation models

These extremely relaxed parameter models are only loosely based on spiking neural networks. Their analysis is being carried out as a thought experiment on how extreme relaxations on neuron synchronisation effect the overall performance and accuracy of the network.

4.2.1 Non-synchronised spiking neural networks

This is the simplest model of an asynchronous network. It acts as a non-sycnchronised spiking neural network where neurons don't attempt any form of synchronisation with their neighbours.

Algorithm 6: Non-synchronised spiking neural networks

```
Initialise neurons;
while time < time_max do
    if Received input from edge then
        Update neurons;
    end
    if voltage ≥ voltage_thr then
        Reset neurons;
        Message neighbours with has_spiked = true;
    end
end
```

Non-synchronised SNN asymptotic analysis

For the non-synchronised network, the number of messages that occur in one simulation is shown in Equation 11 whilst the time taken to complete one time step is shown in Equation 12. Figure 12 shows the effect the neuron count has on the number of messages sent in one simulation.

$$m = n \cdot d \cdot r \cdot T \tag{11}$$

$$t = \max_{0 \leq i \leq N} (t_{i_{synapse}}) \tag{12}$$

where:

m = Number of messages $\quad$ n = Number of neurons

d = Degree $\quad$ r = Probability of neuron firing $\quad$ T = Number of time steps

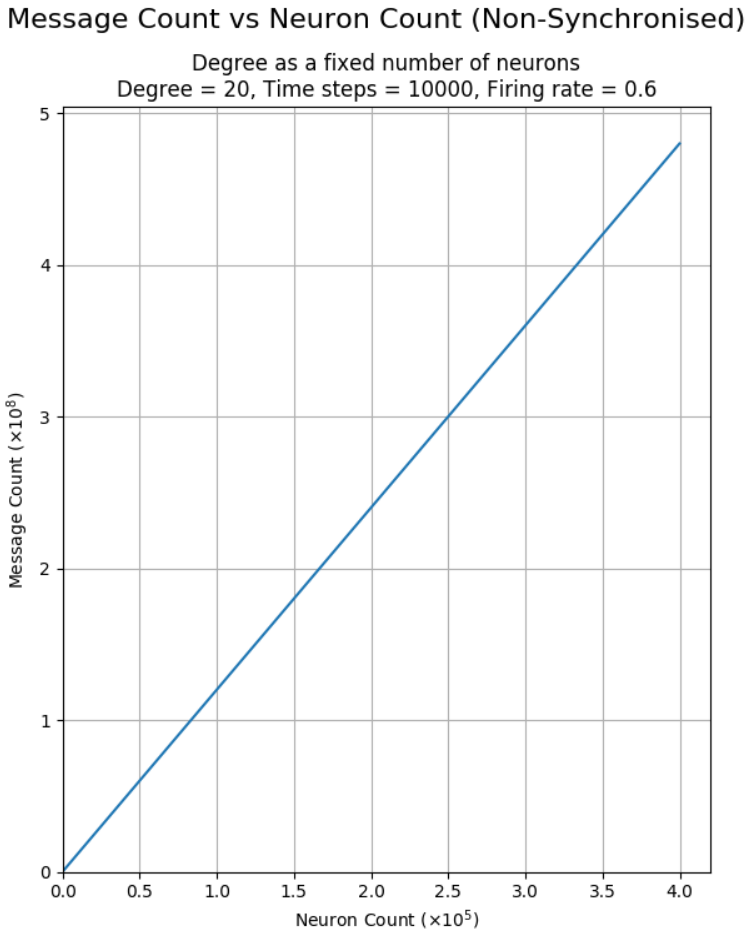

Figure 12: Plot of the non-synchronised network asymptotic analysis. There are no overheads or added devices so the relationship between neuron count and messages is linear.

Non-synchronised SNN discussion

This network will most likely not be accurate. Neurons will fire seemingly randomly since no neuron is synced. One neuron could be 100 time steps behind its neighbour, and then receive a spike from $t + 100$ which would cause the potentials in the neurons to go into disarray. Another issue is that if a node in the POETS platform receives multiple messages at once, it doesn't receive them in order and they are applied randomly. This would further exacerbate the issue of neurons from different time steps sending and receiving messages. In terms of the number of messages fired, this is one of the most optimal since it doesn't add any extra message overhead. This optimality can be seen in Figure 12 where even the largest network only results in five hundred million messages being sent.

4.2.2 Extremely relaxed spiking neural networks

The extremely relaxed spiking neural network attempts to relax synchronisation of neurons whilst also maintaining some kind of synchronisation. It is a thought experiment to try and determine the performance of a graph that is not a true spiking neural network, but *SNN-like*. The idea is to sacrifice accuracy for performance. It does this by passing the most recent timestamp as part of it's message.

Algorithm 7: Extremely relaxed spiking neural networks

```
Initialise neurons;
time ← 0 ;
while time < time_max do
    if Received input from edge then
        if time_message > time_most recent then
            Update neurons;
            time ← time_message ;
        end
    end
    if voltage ≥ voltage_thr then
        Reset neurons;
        Message neighbours with has_spiked = t;
    end
end
```

Extremely relaxed SNN asymptotic analysis

For the extremely relaxed network, the number of messages in one simulation can be seen in Equation 13 whilst the time required for one time step can be seen in Equation 14. Figure 13 shows how the neuron count changes the number of messages sent in one simulation.

$$m = n \cdot d \cdot r \cdot T \tag{13}$$

$$t = \max_{0 \leq i \leq N} (t_{i_{synapse}}) \tag{14}$$

where:

m = Number of messages $\quad$ n = Number of neurons

r = Probability of neuron firing $\quad$ d = Degree $\quad$ T = Number of time steps

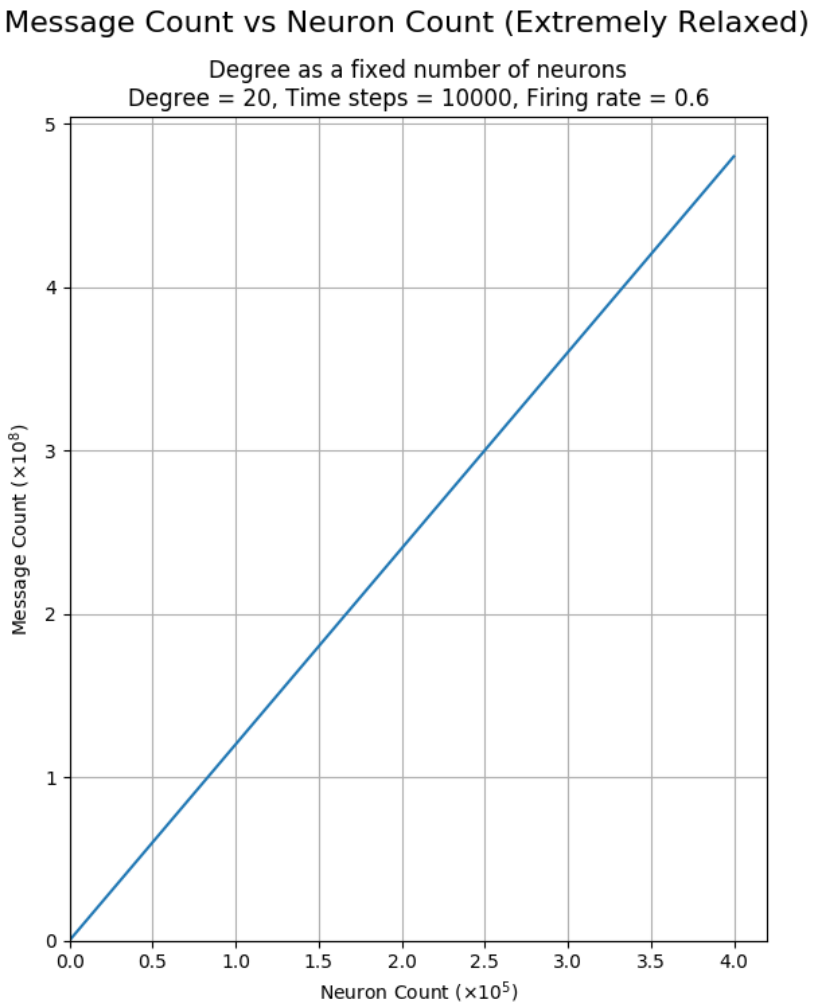

Figure 13: Plot of the extremely relaxed Network asymptotic analysis. For the same reasons as the non-synchronised model (Figure 12), the relationship is linear. It should be noted that this model will likely end before the other models due to the possiblility that the neurons will skip forward many time steps.

Extremely relaxed SNN discussion

Unlike the non-synchronised network, this network attempts to synchronise the neurons. Like the previous network, it doesn't add any messaging overheads which results in a low total message count as seen in Figure 13. It does, however, increase the size of each message since each message has to contain the current time step, rather than just a Boolean value.

It won't be fully accurate since it discards information. If the message received is from a previous time step, then it is ignored. If a neuron receives a new time that is far in the future, recalculating the new value for the differential equation will ignore any possible increases in potential between those two time steps. It is likely that this could cause the network to die out very quickly.

4.3 Evaluation

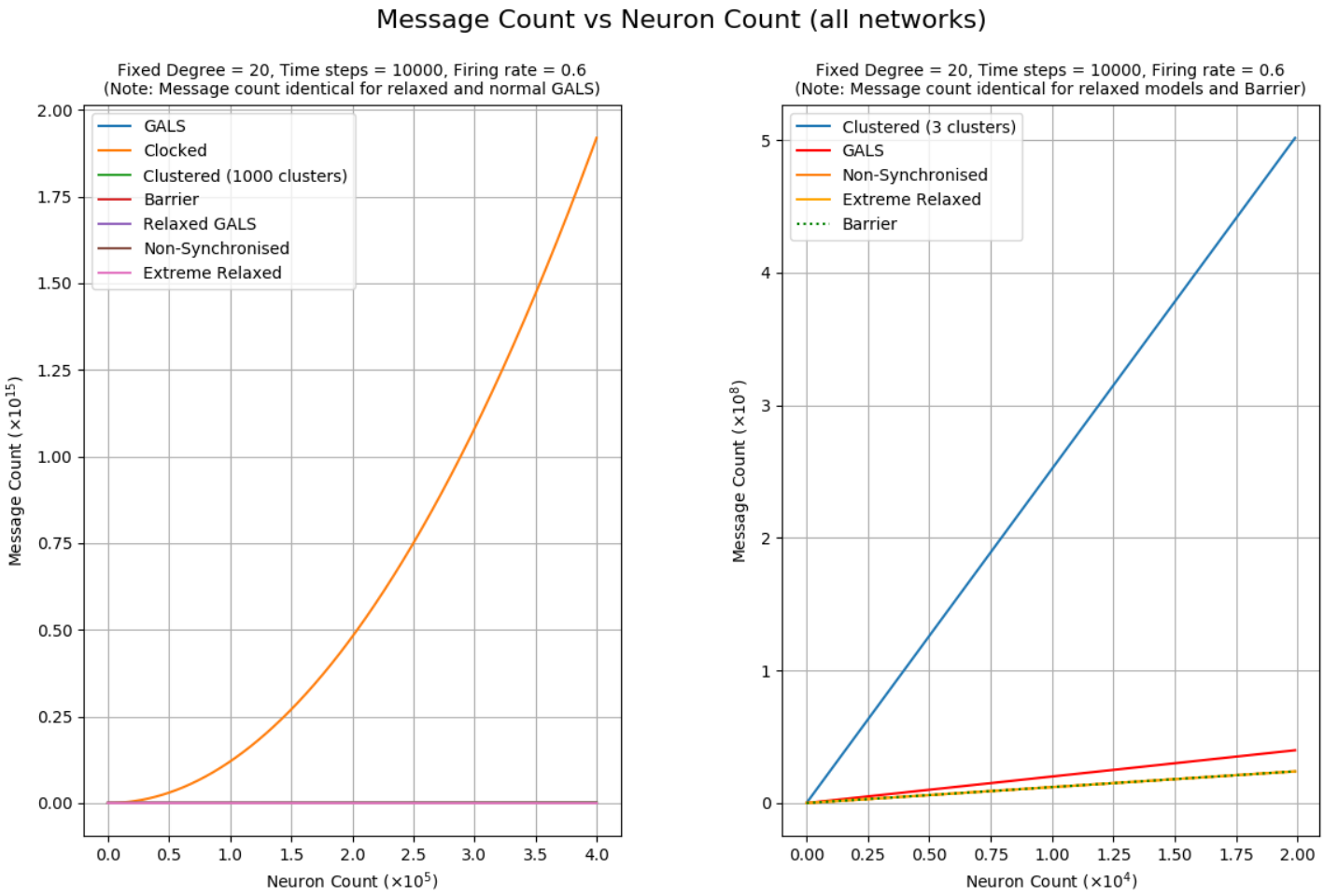

Figure 14: Plot of messages against neuron count for all algorithms. Barrier and relaxed models have very similar asymptotic performance meaning they appear on top of each other. The clock neuron model is removed on the right plot to make their differences clearer.

The best networks in terms of anticipated accuracy are the clocked models. Spiking neural networks, whilst seemingly asynchronous are actually quite synchronous. They heavily feature time based differential equations and the neurons need to be synchronised so that the decay of the differential equations and the increases in potential can be modelled correctly. This, combined with the fact that BRIAN is clocked, mean they are most likely to return results most similar to BRIAN. Unfortunately though, these models aren't ideal since they go against the asynchronous event-driven nature of POETS. Another issue is the overheads caused by having clock neurons connected to every other neuron which vastly increases the number of sent messages as seen in Figure 14. On the other hand, the barrier clocked method doesn't add any overheads in terms of messages, only adding a small constant time between each time step. This can be seen in Figure 15. Based on the lack of overheads and that it is clocked, the barrier clocked method should be the best method.

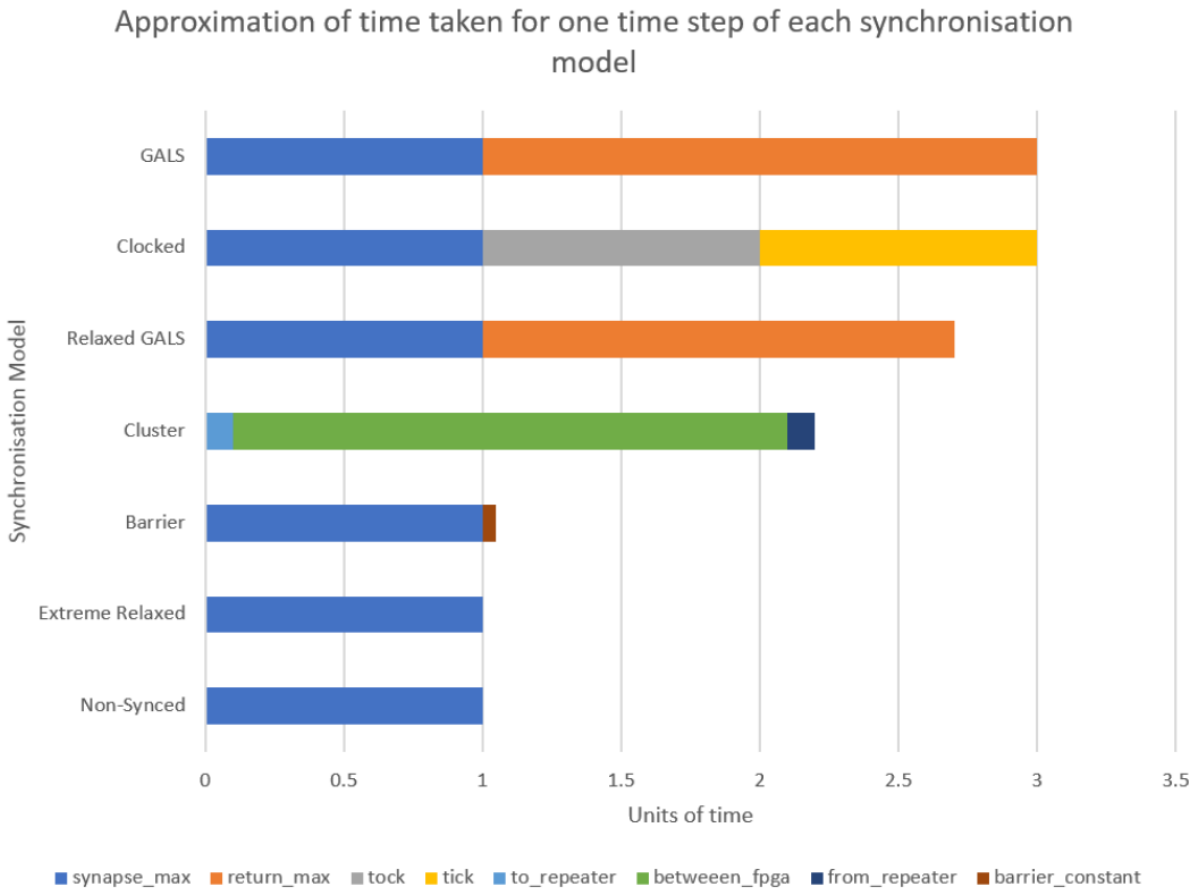

Figure 15: Comparison of the approximate time taken for each model to perform one time step. The best networks are the ones that add no overheads or extra devices, this suggests that the best networks are the extremely relaxed models. Due to the lack of extra devices and the very small overhead, the barrier-clocked model is likely the best accuracy based one.

GALS networks are a good alternative to clocked, they don't add any extra neurons or synapses but they face the issue of being dependent on their critical paths. This can be remedied by relaxing the GALS methods, but it is unclear whether this sacrifice in accuracy

would be detrimental to the overall simulation. GALS also has the issue of sending messages every time a neuron receives a spike since a neuron has to tell all of its neighbours that it has fired or not. This increases congestion of messages significantly, which is particularly bad for messages travelling between FPGAs.

The cluster approach improves on the other methods by reducing the number of messages crossing FPGAs. The increase in neurons doesn't affect the total number of synapses greatly. As well as this, the performance increase gained by minimising cross-FPGA messages could provide a significant performance increase since the bandwidth of these connections are the slowest and congestion of cross-FPGA synapses could grind the system to a halt. A major issue, however, is that it currently isn't possible to control the location of neurons on FPGAs. This means that this model is impossible to actually simulate on hardware.

The relaxed methods could provide a large performance increase since they don't have any extra synapses to control synchronisation. The non-synchronised model will probably be widely inaccurate due to the possibility that neurons could be many time steps behind their neighbours whilst not taking that into account when sending and receiving messages. The extremely relaxed model may act more similarly to an actual spiking neural network, but since it discards information it too is most likely to be inaccurate. These methods are unlikely to perform much better than the barrier clocked model despite not having the small wait between messages being received and hardware idle that the barrier clocked version has. The barrier clocked model adds no extra synapses whilst still maintaining accuracy. The one advantage that these two relaxed models have is that they are truly asynchronous and so fit the philosophy of POETS better than barrier clocked model.

In conclusion, the best method should be the barrier clocked model. It doesn't add any overhead in terms of extra messages, but maintains synchronicity. Its only downsides are the difficulty of implementing it correctly, and the small time between all messages being completed and the hardware idle kicking in. It also doesn't fit with the asynchronous event-driven philosophy of the other methods. Overall it is the most likely model to achieve accuracy comparable to BRIAN whilst maintaining high performance.

*In this analysis, *has_spiked* refers to a neuron sending a message to all its neighbours. In most cases this is a boolean value telling them if this specific neuron has fired.

5 Deliverable 2: Definition and evaluation of accuracy

If a simulator isn't accurate then it isn't worth using. Whether the simulator is accurate depends on several factors, the most important in this case being the timing technique. POETS is designed for asynchronous applications. Whilst spiking neural networks look asynchronous from a macro viewpoint, the are very synchronous as the membrane potentials of each neuron are defined using time based differential equations. This means that to simulate spiking neural networks accurately, they need some method to synchronise the neurons. Since POETS isn't clocked, it is possible that the network's output won't match a clocked simulator like BRIAN. This might not be an issue if a clock neuron is used, but in the case of a GALS approach, there are known issues where messages are received in incorrect time steps[13]. This is due to the fact that if multiple messages reach a neuron at the same time, the order in which the messages are read is chosen at random by the POETS orchestrator.

Other issues that affect accuracy are the lack of a math library in the C++ used by POETS. This requires approximations for several common functions to be made, as well as the use of a single point precision for floating-point values. BRIAN is written in Python and has access to both higher precision floating-point values and accurate high performance math functions.

5.1 Determining accuracy of hardware algorithms

The metric used for determining the accuracy of the different hardware approaches will be a comparison between the different neurons that are firing every time step for a specific hard coded network that is run for a fixed simulation time on both POETS and a separate mathematical model. Overhead costs like initialisation won't be included since they aren't relevant to the actual accuracy. The hard coded network that will be used is Izhikevich since it is widely known and is relatively simple, not requiring complex features such as learning or inputs/outputs.

The networks used for testing accuracy will be small, fully-connected networks with 100 neurons, and will be run for 1000ms. This will give a good approximation for how larger models will act whilst not requiring huge amounts of computing resources. This baseline accuracy simulation will be implemented using a NumPy model[17]. NumPy was chosen for this application instead of BRIAN for several reasons:

- BRIAN won't allow seeding for the C++ compiled networks if there are random number generators used in the equations. This means it isn't possible to reproduce the results.
- NumPy is highly optimised for matrix-based operations[17]. Since spiking neural network connections can be modelled easily with adjacency matrices for their synapses, it makes sense to use NumPy.

- NumPy is widely used for scientific research[17] and thus is a good baseline for experiments. The choice of what baseline to use for accuracy is arbitrary since no model will be completely accurate due to floating-point issues etc.

- NumPy can support 64-bit floats for its arrays which should reduce errors caused by floating-point mathematical operations.

5.2 Evaluation of hardware accuracy

In order to test the accuracy of the hardware model, multiple tests were carried out. The spiking behaviour of both the hardware model and the NumPy model were tracked and saved. The hardware tests were carried out using the fixed versions of GALS, Barrier, and clocked network types. The non-accuracy based methods weren't used since they are likely to be incorrect. The results from the two tests were then both plotted on Figure 16.

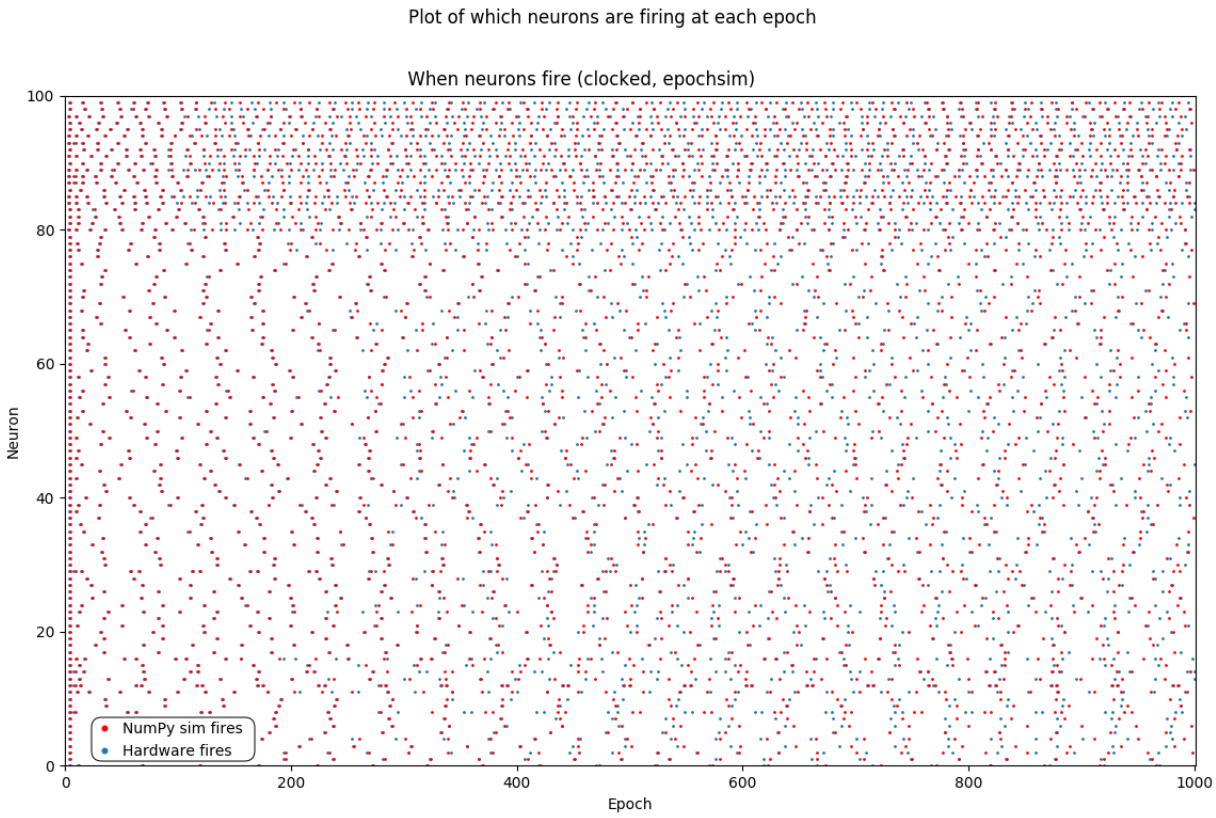

Figure 16: Comparison of when neurons fires on a POETS hardware simulation (blue) and fires in the NumPy simulation (red). Both models are synchronised using a clock neuron.

The same weights and initial values were used for all parameters in both networks. Despite this, it is clear that the two networks diverge after approximate 100 time steps for the excitatory neurons (top 20%) whilst the inhibitory neurons (bottom 80%) start to diverge after approximate 250 epochs.

The difference between the NumPy simulation and the hardware output can be clearly seen in Figure 17. This figure shows the closest fire for a particular neuron between both simulations.

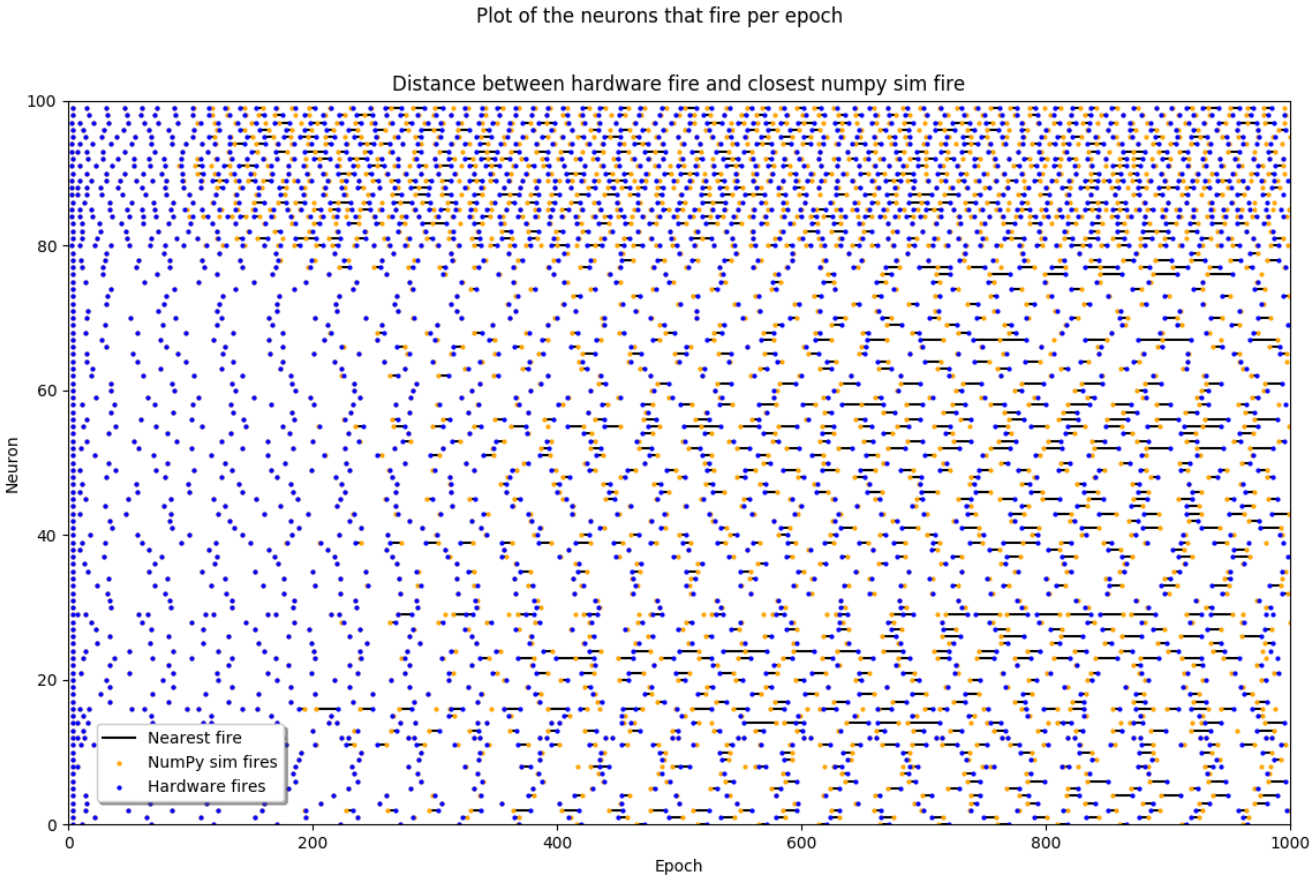

Figure 17: Comparison of the difference in time steps between fires for the POETS hardware simulation and the NumPy simulation. Each fire on the hardware is linked to the closest fire to it on the NumPy simulation thus showing how the networks deviate over time.

It is clear that the error between the NumPy model and the hardware model increases before reaching a steady state where the error between each neuron appears to be fairly similar between time steps and there is a regular difference between hardware fires and NumPy simulation fires. This can be seen in Figure 18 where the error is plotted against each time step. A fitted curve was generated for the parts of the function where the error was not zero.

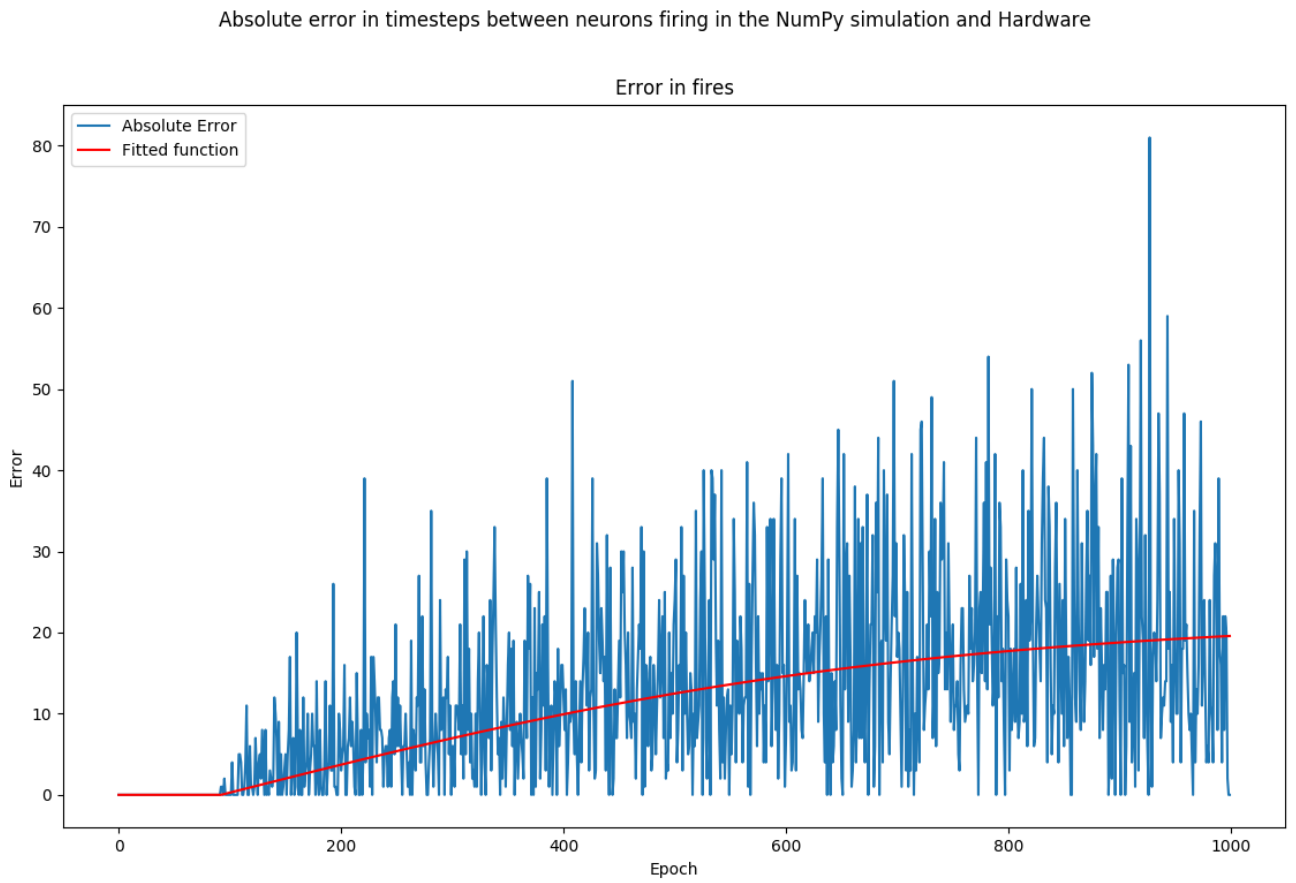

Figure 18: Function showing the size of the error between fires for the POETS hardware simulation and NumPy simulation. The curve was generated using SciPy's `optimize.curve_fit` function.

At first glance it seems that this result means that the hardware simulation is not accurate and that its use cannot be justified. However, through further investigation it can be determined that this behaviour might actually be expected and that the use of the hardware model is still valid.

5.2.1 Divergence explanations

Some of the divergences arise from floating-point issues. Computers use floating-point arithmetic[18] to represent real numbers during operations. These are approximations that trade precision for processing time. They are represented using two integers to form a *significand* and an *exponent* allowing for the representation of numbers in the form:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

The amount of precision limits the size/precision of numbers that can be represented exactly. Integers that contain more digits than the architecture of the computer can represent will be rounded. For example: 123456789 cannot be exactly represented with eight digits and would be rounded to 123456790 where the final 0 is not explicitly stored, thus creating a slight error wherever it is used.

If a number is rounded and then used in further operations, then the error between the rounded number and the actual number is compounded and propagated through the program gradually increasing the error in each result. This is expected and cannot be avoided unless fixed point arithmetic is used. Unfortunately, Python and NumPy use floating-point arithmetic. Although fixed point arithmetic is possible on POETS, it is extremely difficult to implement and is actively discouraged.

The three main hypothesised causes of error arise from floating-point errors and are discussed in the rest of this section.

Pairwise summation

The most common mathematical operation used in either of the simulations is summation. The inputs to a neuron are summed together millions of times during a full simulation. This means that any rounding is will cause larger and larger errors as the simulation progresses. There are several methods that can be used to alleviate the size of the error in floating-point operations, including the one used by NumPy: pairwise summation[19]. However, this can cause other issues to arise.

Pairwise summation[19] is defined in the following algorithm:

```
def pairwise(x):
    m = floor(len(x) / 2)
    return pairwise(x[:m]) + pairwise(x[m + 1:])
```

It is a technique to sum arrays of floating-point numbers that reduces the floating-point error by using a divide and conquer algorithm. The worst case error when using pairwise summation increases at a rate of $O(p \log n)$ (where p is the precision of the computer) whilst with the naive summation (where elements are added one at a time) has a worst case error that grows at $O(pn)$[19]. There are alternative approaches such as Kahan summation, where the worst case error grows at $O(\sqrt{n})$[20], but pairwise summation has superior performance and this performance outweighs the difference in growth of error.

As the algorithm uses divide and conquer, the order of the inputs can affect the result of the summation. A test was done comparing the naive summation function used by Python with the pairwise summation using NumPy. In this test, the sum of different length lists was calculated, and then their elements shuffled and the sum re-calculated. Figure 19 shows that the order of summation clearly affects the outputs for both approaches.

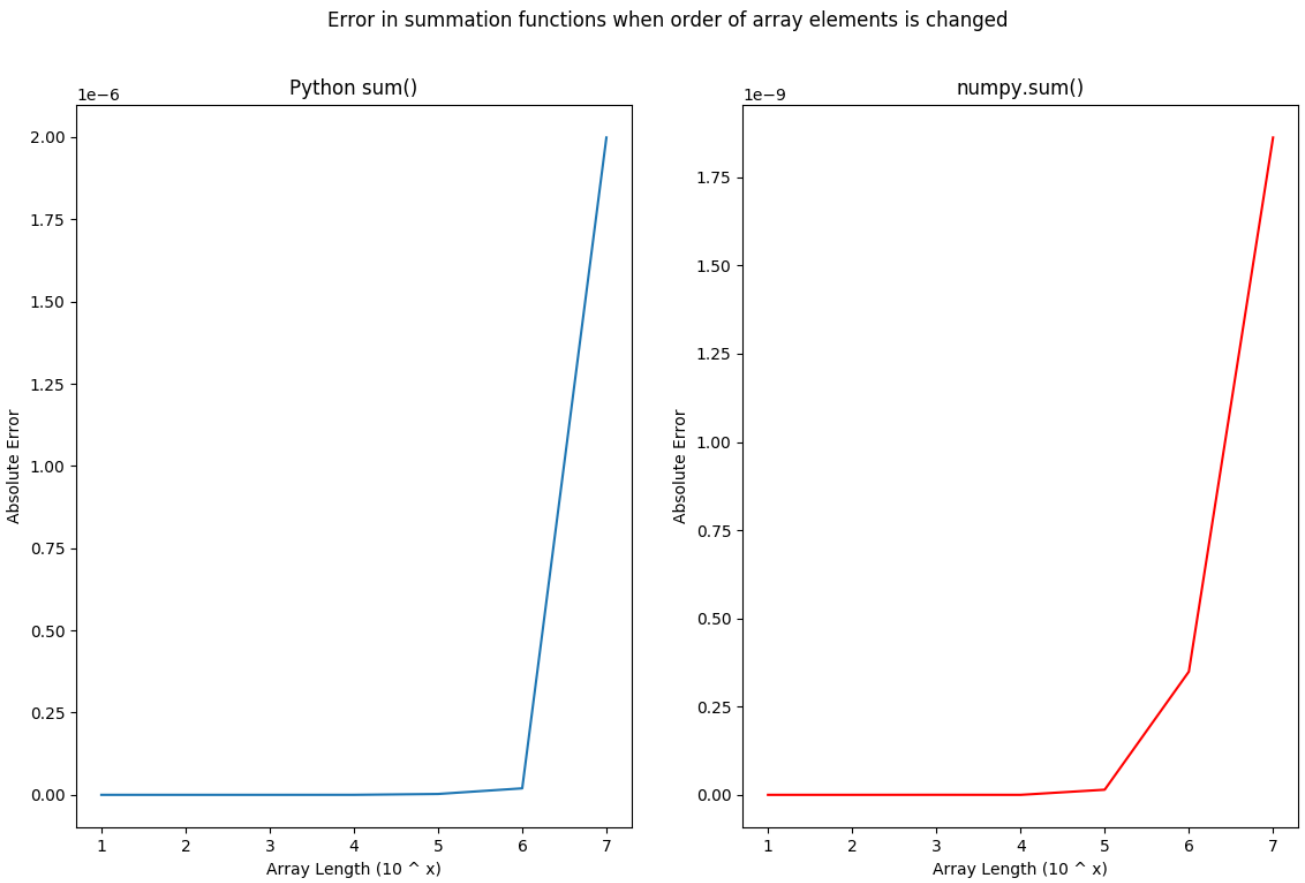

Figure 19: Comparison of pairwise summation vs the naive summation. The error in the `numpy.sum()` grows at a significantly lower rate than the normal Python `sum()`. Note that there is still an error, so pairwise summation will still cause errors in the final simulation.

To investigate how this affects the output of the simulation when the order of neurons were changed, another test was devised. In this test, all parameters in the simulation model were kept identical, but the order in which the neurons appeared in the NumPy array was modified. Care was taken to make sure the adjacency matrix was modified accordingly so that mathematically, the networks were identical. In the case where only two neurons were swapped, the networks begin to diverge after around 50 time steps. This suggests that the order in which the internal operations of a neuron are calculated is very important and that extremely small changes to how the network is connected can lead to large differences. Figure 20 shows that the errors are quite pronounced on a small 100 neuron network. Therefore, the error would be even larger for networks that contain thousands or millions of neurons.

The order of the neurons on the hardware model cannot be controlled by the user. The orchestrator program chooses where neurons are placed and therefore the results of a POETS simulation would be expected to differ from the NumPy model over time based on the arguments above. The node placement on POETS graphs cannot be controlled by the user and thus errors caused by the neuron order cannot be fixed and must be accepted.

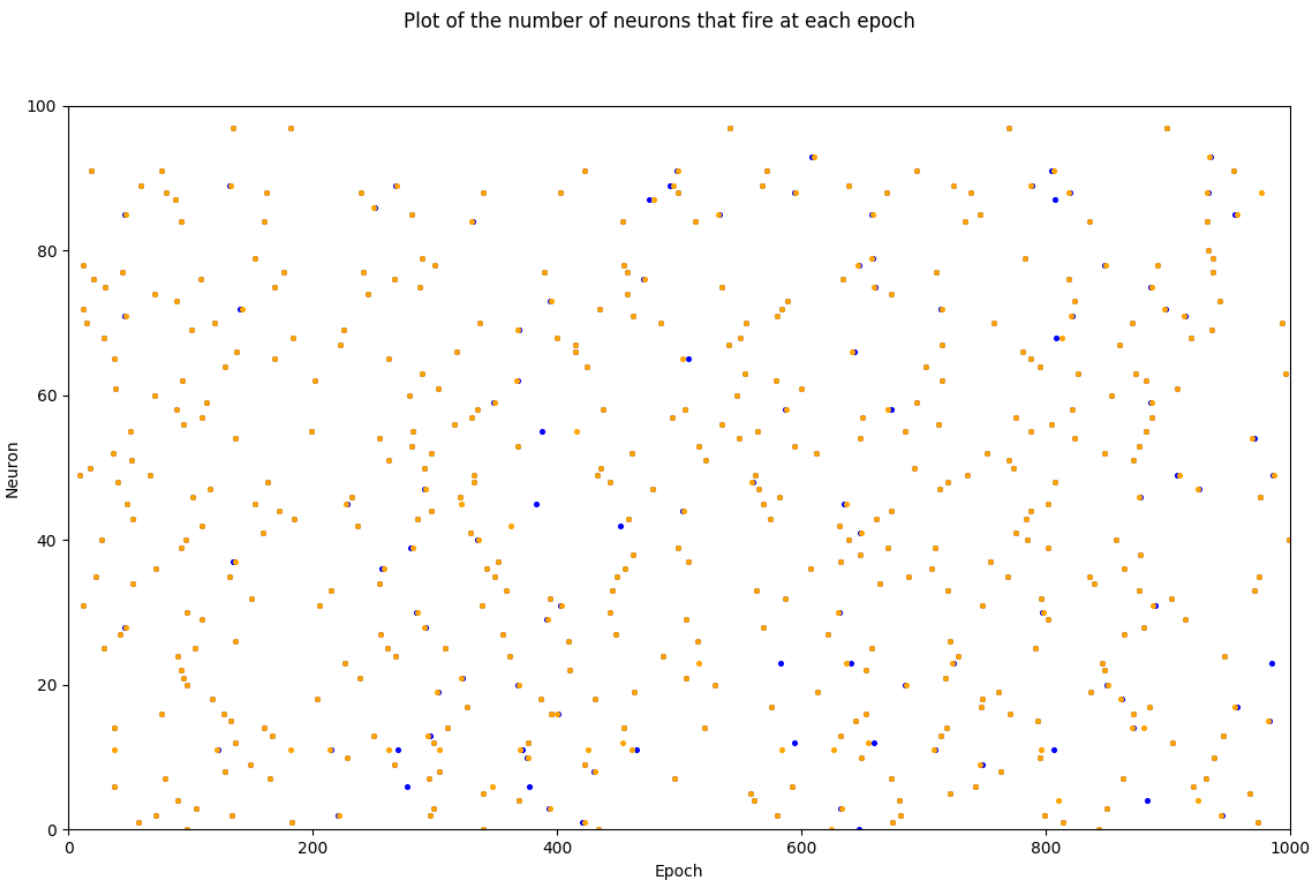

Figure 20: Comparison of when the various neurons spike in two different networks where two neurons have had their indexes swapped. It can be easily seen that the two networks don't behave the same way despite all the network parameters being identical.

Underlying maths functions

Another issue is the lack of a maths library for the hardware simulator. The maths functions that have been used in the implementation of this software will be explored in the implementation deliverable (deliverable 3), but the basic issue is that they make heavy use of approximations to calculate trigonometric functions, logarithms, and square roots. As will be seen in the implementation section, care was taken to make sure that they were sufficiently accurate for the chosen application, but since they are just approximations they are going to cause small errors.

In a similar way to how the summation causes floating-point errors to propagate, the error in these functions will gradually propagate as the simulation progresses. But since for the first 100+ epochs, the networks are identical, the error that arises from this isn't large enough to discourage their use. If the error in these functions was significant then the effect of their error would be seen much earlier. All the approximations chosen were accurate to within at least three decimal places when compared to the C++ standard library math functions.

Floating-point precision

The final issue that can cause a discrepancy between the NumPy model and the POETS hardware model is the size of the floats used. The hardware model uses C++ floats which are 32 bits in size and have 7 bits of precision. The NumPy model uses `numpy.float64` which offers 16 bits of precision, thus a difference in the outputs would be expected since the hardware model has more pronounced rounding than the NumPy model.

A problem with floating-point numbers that has not been mentioned yet is the problem of scale. Each number has an exponent to determine the order of magnitude of the number. This allows you to easily represent either very small or very large numbers easily. An issue arises however when a number with a large exponent is added to a number with very small exponent. In some cases, adding two numbers of different scales will end up with the smaller number vanishing since there is no way to fit it into the representation of the larger number. This will result in the addition doing nothing, causing mathematical errors on top of the inherent errors caused by using floating-point operations talked about earlier.

With spiking neural networks of large sizes, this could cause issues. If there are sufficient inputs to a neuron then after the potential of a neuron gets large enough, adding more very small values to it won't affect the potential. This is more likely to be a problem in the NumPy simulation than in the actual hardware since NumPy will wait for all inputs to be summed before doing any calculations on the internal logic, whereas since POETS is asynchronous, it will recalculate the logic every time it receives an input. This means that the values inside the neurons shouldn't get much larger than the threshold value, whereas using NumPy they might.

5.3 Overall assessment of hardware accuracy

Based on the fact that the first 100+ time steps of the two networks are the same, as can be seen in Figure 17, it can be postulated that the resulting error arises from a combination of the summation techniques used by the different software, the order of the neurons, how the underlying mathematical functions are implemented, and the floating-point precision of the underlying hardware of the two systems.

There are improvements in the POETS version that can be made, such as increasing the accuracy of the underlying maths functions as well as the amount of precision used. In reality, it is likely that both the NumPy and BRIAN models also have errors. In most cases, these errors will be caused by the same underlying problems that are causing the POETS version to diverge from the software models since issues with floating-points come down to computer hardware and the languages that the programs are written in. It is impossible to ever get a model that is 100% accurate without infinite precision.

Whilst it would be interesting future work to improve the accuracy of hardware simulator so that it more closely matches the NumPy (or another) model, fixing these problems is out of the scope of this project. It would involve diving further into the source code for NumPy as well as modifying how the POETS version of the code is compiled and run on the hardware. For these reasons, the current hardware model is sufficiently accurate for use as a spiking neural network simulator for the purposes of this project.

6 Deliverable 3: Devising simple methods for specifying spiking neural networks for POETS

The primary software deliverable is a piece of software for simplifying the process of specifying spiking neural networks for POETS. The goal of the project was to evaluate whether spiking neural networks are better simulated on POETS or alternatives such as BRIAN. Thus, usability must be comparable between POETS and software like BRIAN.

6.1 Motivation

Currently, to specify a network on POETS hardware the user must provide two things:

1. A `GraphType`. This is an XML based description of the logic that runs the network. It contains information on the types of device (in this case a neuron) that are present in the graph and their logic. Each device has input and output connections. The behavior when receiving an input message is specified here with options for logic to be carried out during when a message is received, when a message is sent, when the device is ready to send a message, and when the hardware is idling. For a spiking neural network application, the logic usually consists of: summing the inputs to a neuron as they are received; updating the internal differential equations; firing a spike; and resetting if necessary. The logic is defined using C++ code inside the XML file meaning it is difficult to write the logic for a network. It cannot easily be debugged and tested.

2. A `GraphInstance`. This is an XML based description of how the devices of the network are connected to one another. This is done by specifying edge devices that connect the normal devices. They have various parameters, synapse weight being the relevant one in the case of a spiking neural network. These parameters can be used by the devices they are connected to. This part of the network is generally the largest part for spiking neural network applications since most networks contain hundreds of thousands of synapses. It is therefore infeasible to hand code this section.

Most users of POETS create the graph type code by hand and then use scripts to generate the graph instance. This isn't very user friendly and is vastly inferior to the ease of use that simulators such as BRIAN provide. In order to improve the experience for a user that just wants to run spiking neural networks on POETS, I developed software that allows users to specify code in a simple configuration file and then have it automatically converted into the necessary XML format.

6.2 Implementation

In order to provide an experience more similar to using a simulator like BRIAN, software developed that allows a user to specify different parameters of the network in a configuration file. In this file, a user specifies all neuron and network variables. Multiple sets of variables can be used for different types of neurons. The user of this file can then run the Python code, and an XML file containing the graph type and graph instance files will be produced. A diagram of the toolchain can be seen in Figure 21.

An important feature of this program is its ability to randomise parts of the parameters. If a user specifies R in place of any inputs, then when the file is parsed a different random number is chosen and evaluated for each neuron. This is done instead of the random number being evaluated when the configuration file is read and then used for multiple neurons. In order for this to work the software has a maths parser included. This can read strings containing mathematical expressions and evaluate them when needed. This improves the capabilities of the simulator and provides better support for users as they no longer need to write their own scripts to produce networks.

Users can save the outputs to a log file and then print them onto a graph. Whilst there is software in the POETS repository for printing the outputs of networks, it is very slow and the data produced is extremely difficult to work with. The new implementation works best with smaller networks as the log files produced can end up being very large which slows the program down. By using matplotlib[21] users can produce graphs for use in reports and papers. This will be useful for researchers as they are the primary users of spiking neural network simulation software.

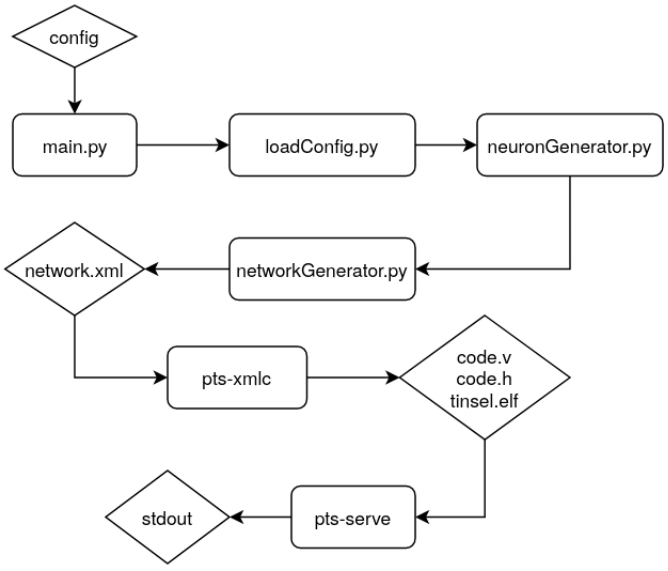

Figure 21: The toolchain of the implementation containing all the steps from specifying the initial config file to running the generated code on POETS hardware.

6.3 Main development hurdles and how they were overcome

When the algorithms were first implemented the resulting networks did not run as expected. The GALS networks wouldn't ever spike, whilst the clocked networks seemed to spike with a regular pattern. Whilst it is expected that neurons should be firing almost every epoch, neurons were only firing at a few regular time steps during simulation as can be seen in Figure 22.

Since the neurons only fire in certain time steps, this implies that the major source of membrane potential increases is the thalamic inputs to the network. These are small, random inputs that stimulate each neuron very slightly every time step to simulate electrical noise in the brain. Without these, the neurons would never fire as the membrane potential would never increase above the threshold value. No issues were found with the logic behind the rest of the network and therefore the issue must have been with the thalamic input, which is based off a custom random number generator.

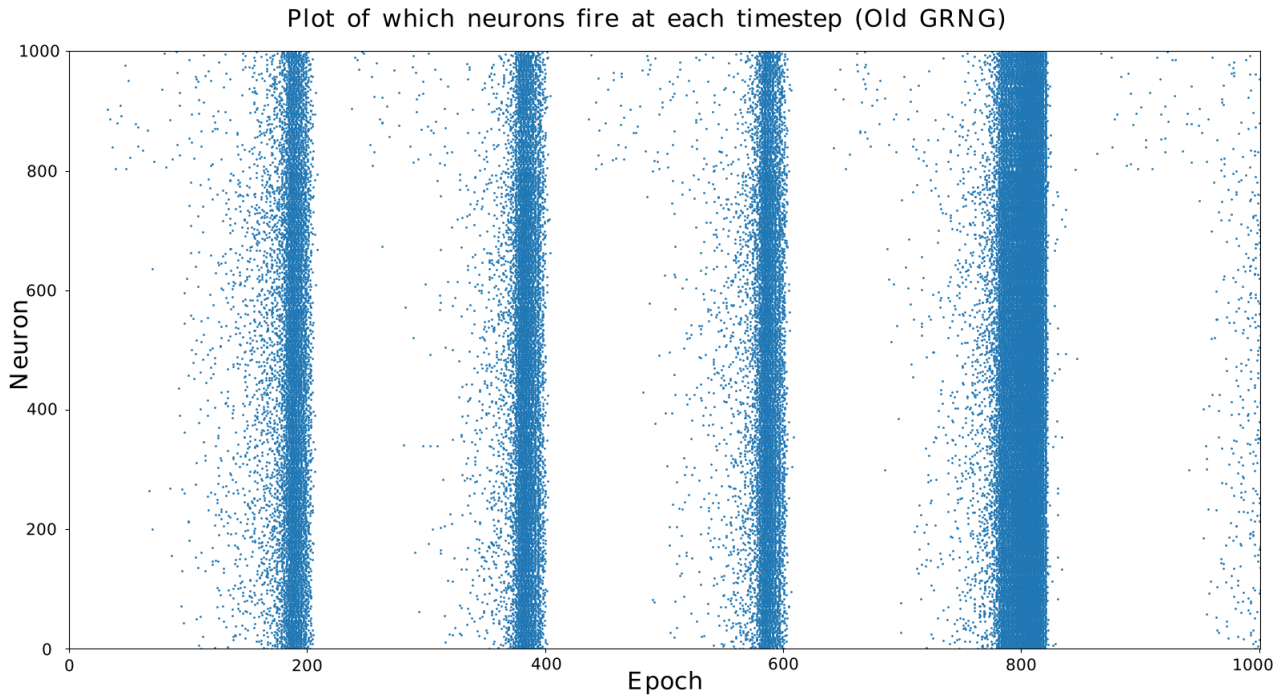

Figure 22: The original behavior of the clocked SNN. It can be seen that whilst the neurons are spiking at regular intervals, there are large sections of time when no neurons fire. The behavior exhibited here isn't how the network should act.

6.3.1 Random number generation

Unfortunately, whilst the POETS graph logic is written in C++, the use of the C++ maths library from the standard library doesn't have an implementation on POETS. Due to this lack of a mathematics library, all functions such as various trigonometric functions and random number generators need to be implemented from scratch. There were two random number generators used in the various POETS spiking neural network applications. One was a uniform random number generator that utilised a simple linear congruent generator. The second was a Gaussian random number generator that used the linear congruent generator and some shifts, to generate values from the normal distribution.

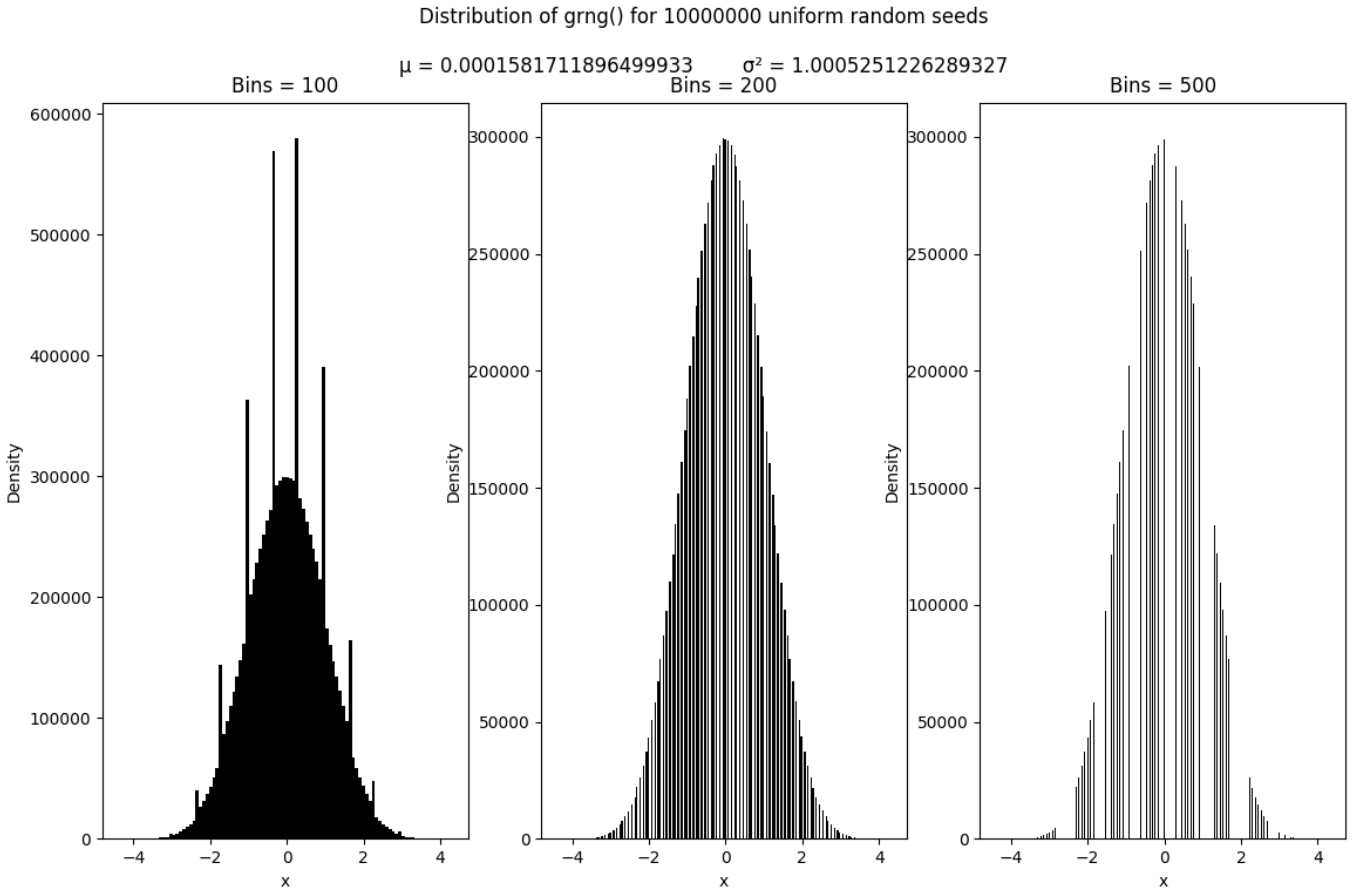

Figure 23: The original Gaussian random number generator. Whilst the mean and standard deviation are accurate, some values are more likely to occur than others and some values don't occur at all.

Tests were carried out on the two random number generators to determine their accuracy. It was initially hypothesised that the uniform random number generator would be at fault due to the short cycles of the lower order bits that can cause the pseudo random number

generation to act in unexpected ways[22]. It was actually found that whilst the linear congruent generator functioned correctly, the Gaussian random number generator caused some interesting problems. In Figure 23 containing a histogram of the Gaussian generator, it can be seen that whilst the mean and standard deviation are correct, there are certain values that occur more often than they should whilst some values never occur.

The alternative Gaussian random generator chosen was the Box-Muller transform[23]. It is defined in Equations 15 and 16. If U_1 and U_2 are drawn from a uniform distribution then Z_0 and Z_1 are independent random variables with a standard normal distribution:

$$Z_0 = \sqrt{-2 \ln U_1} \cos(2\pi U_2) \tag{15}$$

$$Z_1 = \sqrt{-2 \ln U_1} \sin(2\pi U_2). \tag{16}$$

This method was chosen because it is known to be effective at generating Gaussian random numbers and unlike the previous generator, the distribution that it produces is mathematically identical to the normal distribution. The Cartesian form of the Box-Muller transform was chosen over the polar form because it was desired that the generator be as accurate as possible. The polar form utilises rejection sampling and so some generated values as discarded. Since this may have resulted in a distribution similar to the one produced by the old generator, it was decided that the small performance increase of the polar form wasn't worth it.

The overall performance of the Gaussian random number generator using the approximations outlined in the next section was very similar to the old generator, and in some cases faster, as seen in Table 1, and so it was decided that this Box-Muller implementation was suitable for POETS.

Table 1: Time taken to run a 1000 neuron, fully-connected network with different GRNGs.

	Old GRNG	**Box-Muller GRNG**
Real	2m41.178s	2m39.681s
User	2m40.982s	2m39.102s
Sys	0m2.681s	0m2.206s

6.3.2 Lack of mathematical operations

Due to the lack of a maths library for POETS, and the fact that maths functions are required for the Gaussian random number generator, approximations of these functions had to be implemented. Approximations were chosen that maximised performance whilst maintaining reasonable accuracy. The error between the approximations and the actual C++ implementations was negligible.

Trigonometric functions

Sine and cosine functions are required for the implementation of the Box-Muller transform. The only necessary trigonometric function to implement them is the cosine function since sine can be represented with a cosine with an input shifted by half pi. The polynomial approximation that was used for the cosine approximations can be seen in Equation 17:

$$\cos(x) \approx 0.99940307 + x^2 * (-0.49558072 + x^2 * 0.03679168) \tag{17}$$

The source of these coefficients is the book "Computer Approximations"[24]. There were multiple precision approximations provided with the lowest approximation providing 3 digits of precision over the range 0 to $\frac{\pi}{2}$. During testing, the difference between this approximation and ones with higher precision wasn't enough to justify their use. The maximum error produced by the chosen approximation was 0.0006 for cos(1).

Table 2: The transforms needed to use the cosine function that is accurate over the 0 to $\frac{\pi}{2}$ range for the entire 0 to 2π range.

Quadrant	**Cosine**
0	$\cos(x)$
1	$-\cos(\pi - x)$
2	$-\cos(x - \pi)$
3	$\cos(2\pi - x)$

In order to stretch this cosine interval over the full 2π period, we can easily just shift the inputs depending on the quadrant. This is shown in Table 2. The use of this mapping allows for the same code that works over a $\frac{\pi}{2}$ interval to be used for the whole range.

The performance of the new function is very good as shown in Table 3. Figure 24 shows that the error between the two functions is negligible.

Table 3: The time required for cmath and custom cosines to evaluate different sets of samples.

Size of set	**CMath cosine time taken**	**Custom cosine time taken**
$1e6$	5490 μs	5305 μs
$1e7$	53774 μs	49971 μs
$1.6e7$	89765 μs	88671 μs
$2.6e7$	109155 μs	98825 μs

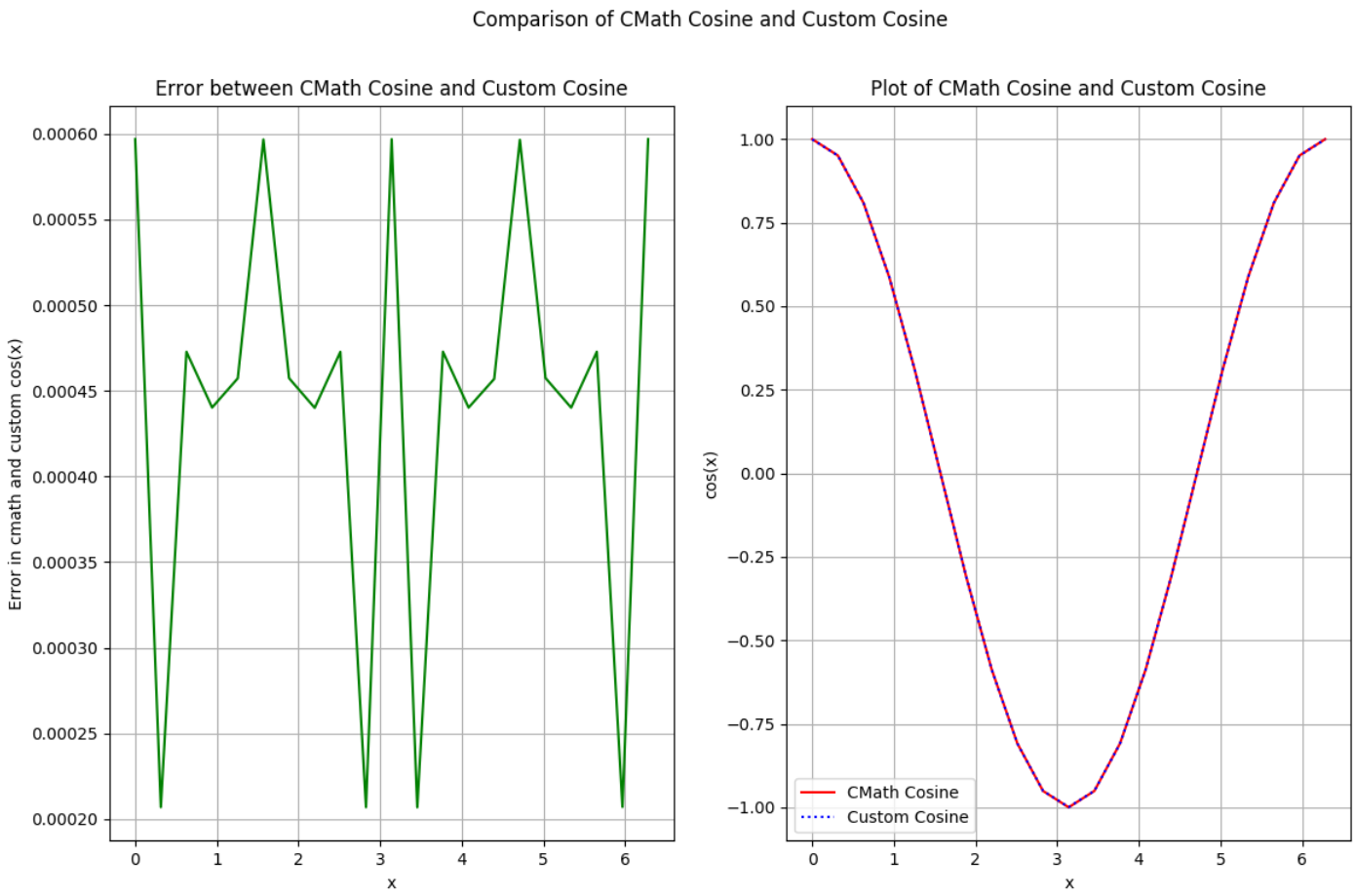

Figure 24: A comparison of the error between the CMath cosine and the custom cosine. It can clearly be seen that for the range 0 to 2π, the error is negligible.

Square root

The square root approximation used in this simulator has a precision of 32 bits and can perform better than the default standard library square root by almost 95%[25]. The reason for this is that the standard library square root has a precision of 100 bits[25]. For POETS, 32 bits provides enough precision for the simulation to behave accurately, as seen in Figure 27. Whilst the algorithm performed well, it didn't perform 95% better. Its performance can be seen in Table 4.

IEEE-754 defines a 32-bit, binary, floating-point number as a number consisting of a 24-bit significand (one bit containing the sign) with an 8-bit exponent[26]. Since the exponent can be negative, 127 added to the exponent to ensure it is positive. Since $\sqrt{x} = x^{\frac{1}{2}}$ it would seem that all that needs to be done to calculate the square root is to subtract 127 from the

exponent and shift it right once and re-add 127 to keep it a float. This is accurate unless the exponent is odd since the least significant bit of the exponent will be shifted into the significand, this must be shifted to remove it from the significand[27].

Table 4: The time needed for the cmath and custom square root functions to evaluate various sized sample sets.

Size of set	**CMath square root time taken**	**Custom square root time taken**
$1e7$	14165 μs	13923 μs
$1e9$	1200053 μs	1199565 μs
$1e10$	11909312 μs	11982789 μs
$1e11$	118895426 μs	119773190 μs

The performance of the new square root function is very good as shown in Table 4 and its accuracy is good as shown in Figure 25.

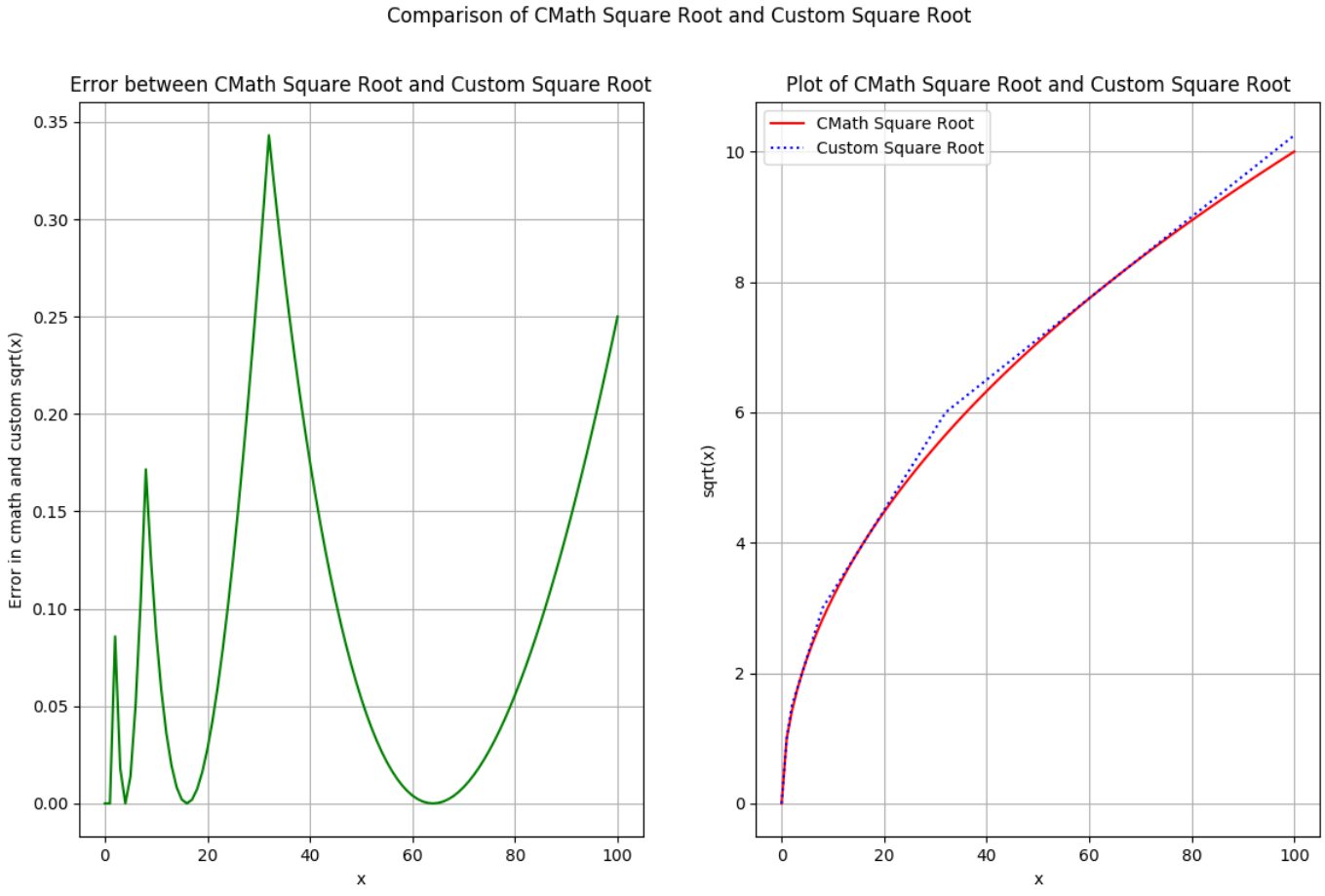

Figure 25: A comparison of the error between the CMath square root and the custom square root. It can clearly be seen that the error is low for all values.

Logarithm

The final approximation needed was a natural logarithm approximation. The logarithm approximation used only contains multiplications and bit-wise shifts to provide a highly accurate and quick logarithm approximation[28]. This approximation has an accuracy of within $1.5e^{-4}$ and the time taken to execute is negligible as seen in Figure 26 and Table 5 respectively.

Table 5: The time needed for the cmath and custom logarithm functions to evaluate different sized sample sets.

Size of set	CMath logarithm time taken	Custom logarithm time taken
$1e7$	14205 μs	13824 μs
$1e9$	121178 μs	119086 μs
$1e10$	12109882 μs	11781912 μs
$1e11$	120580224 μs	120491812 μs

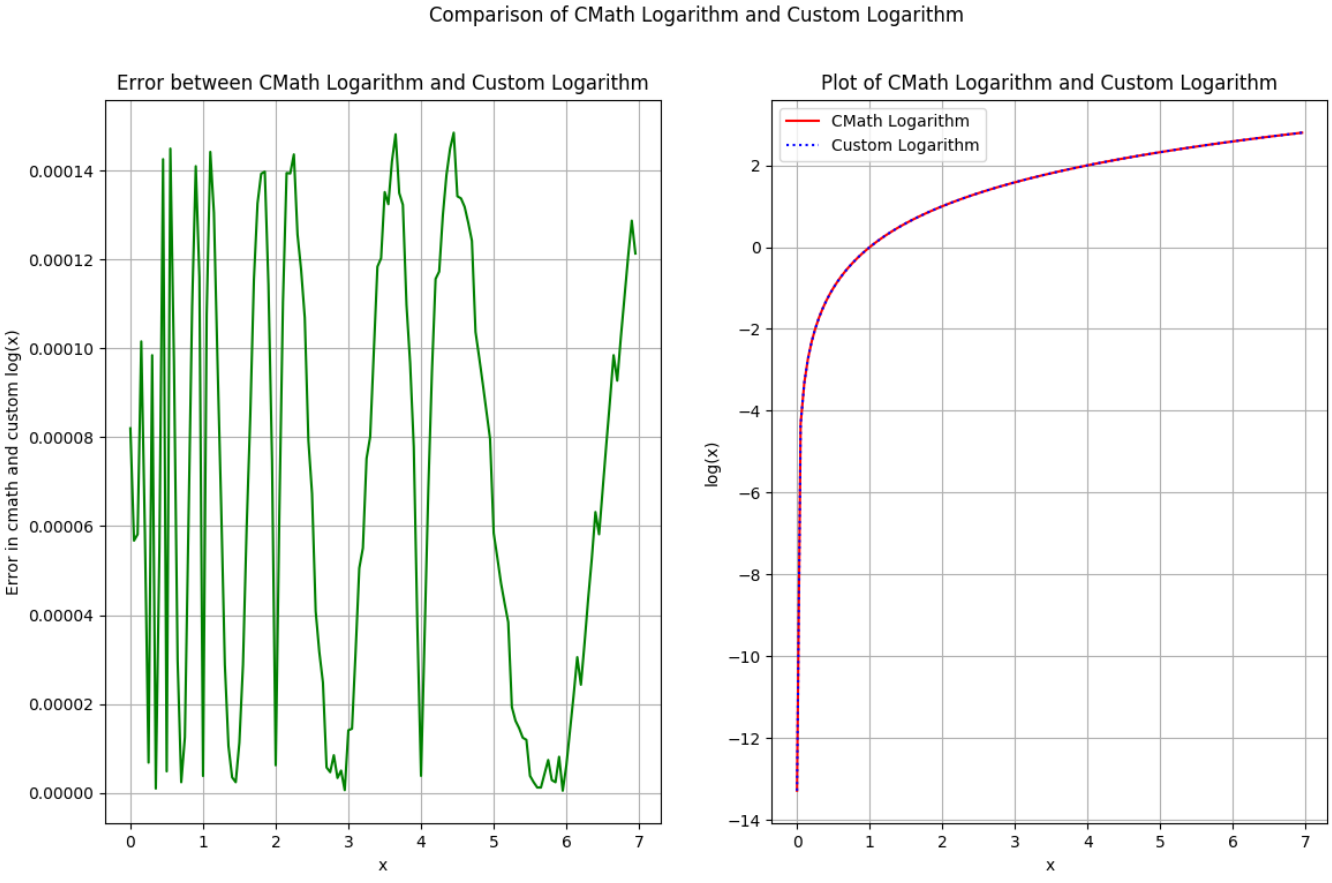

Figure 26: A comparison of the error between the CMath logarithm and the custom logarithm. It can clearly be seen that the error is negligible for all values with the two graphs overlapping each other significantly.

Network output with Box-Muller and approximations

After implementing all the approximations and thus fixing the Gaussian random number generator, the output of the network is closer to the expected firing pattern, as seen in Figure 27. In the broken model, seen in Figure 22, the spikes only happened during specific intervals during the simulation. In the model that utilises the new Gaussian random number generator, spikes occur frequently throughout the simulation and the output of this network is comparable to the network output that was presented in the original Izhikevich paper[15], seen in Figure 28.

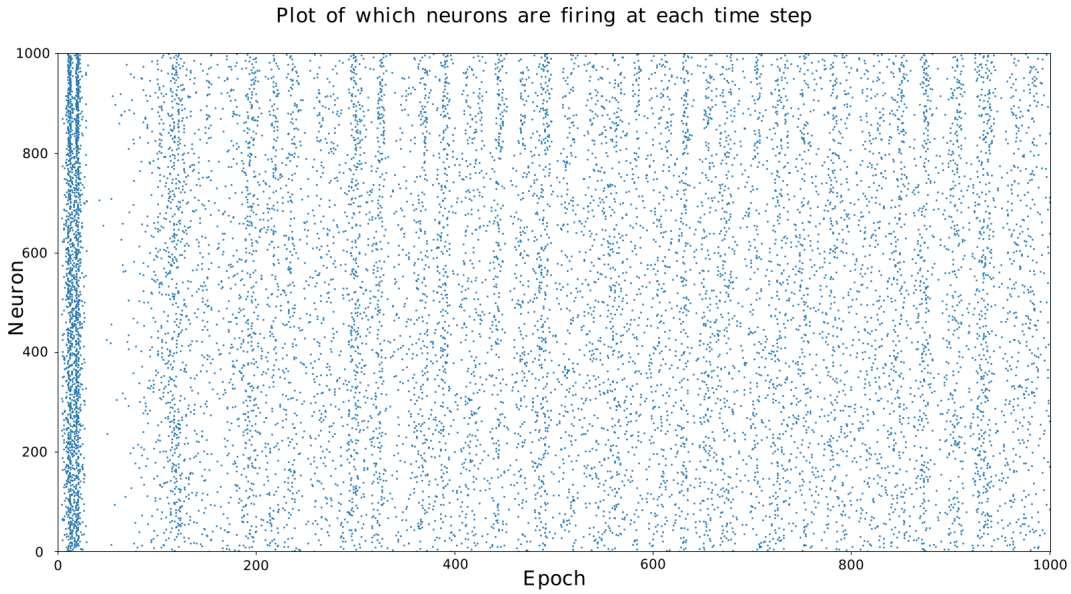

Figure 27: The results of the simulation with the custom functions implemented. This plot is very similar to Figure 28 and completely different to the old plot seen in Figure 22.

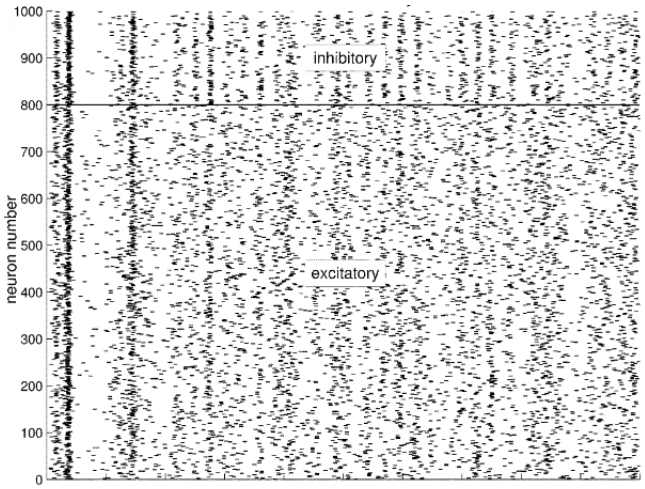

Figure 28: Simulation of 1000 randomly coupled spiking neurons from the original Izhikevich paper: Simple Model of Spiking Neurons[15].

6.3.3 Memory issues for XML generation

The final issue that arose was a minor one. The challenge was making sure that the software wouldn't run out of memory when being used to generate large networks. Memory limitations of POETS have previously been identified by Pomienski[13, p.19] and proved a similar issue in this project. During development, some reasonably small networks cause the computer to run out of memory, as a network was being compiled to XML.

For large networks, there could be tens of thousands to tens of millions of neurons and if these are stored as a list of objects then the amount of memory required would become excessive. For small networks this isn't significant, but POETS and therefore my software are primarily intended for larger networks. The obvious solution to this problem was the use of generator routines. These don't hold the entire lists in memory and yield values only as they are needed. Through the use of generators, the memory issues were solved, allowing large networks of billions of synapses to be compiled to XML.

6.4 Future work

Over time, the scope of the project changed meaning that features that were initially planned weren't implemented. The software in its current state meets the objectives of this project. Features have been identified as possible opportunities to further improve the tools:

- **Allowing the simulator to adapt and learn** - This would allow for the use of spiking neural networks on POETS to be used in applications such as classification. With the possible performance increases that come with platforms like POETS, the time to train could be exceptionally quick. Some ideas for learning would be spike-dependent-timing-plasticity or Hebbian learning, as described in section 2.2. It is more difficult to train spiking neural networks due to a lack of techniques like backpropagation[29]. However, capability to learn would be useful in increasing the applications of SNNs on POETS.

- **Different network connections** - A feature of BRIAN is the ability to connect neurons together in ways that aren't related to probability[30]. Whilst this can be done by hard coding connections now, it cannot yet be done automatically. To achieve feature parity with BRIAN, this should be implemented.

- **Adding input** - The current implementation doesn't contain any way to provide input to the network. It would be useful for users who want to run spiking neural networks on POETS hardware, it should include features that allow neurons to be stimulated with external measures (to allow dopamine injection simulations for example). This is also important if the simulator is to be used for deep learning methods.

- **Interface improvements/better Python integration** - Whilst the current Python functions that have been implemented for specifying spiking neural networks are an improvement over writing the networks in XML, they still use slightly confusing techniques for specifying parameters. A user should be able to specify things in a way much more similar to BRIAN or even normal Python variables.

- **Compressed XML outputs** - The size of some of the networks results in XML files with over one hundred million lines for a reasonable small network. The size of these files can exceed 8GB which aren't easy to work with or open in other editors if you desire to modify the XML directly. Larger networks would result in even bigger files. In order to improve the experience of working with these files, XML compression could be used. There are techniques available that can get a compression ratio of up to two or more[31] which could be applied to good use here.

- **Further memory improvements** - Whilst care was taken to use generators and other techniques to minimise memory use, whether each neuron is connected to another or not is determined using an adjacency matrix. For extremely large networks, this Python object storing the adjacency matrix can grow to tens of gigabytes in size[32]. This is not an ideal situation as users without the necessary memory won't be able to use the software.

7 Deliverable 4: Hardware testing of algorithms on POETS

POETS would be a suitable alternative to more conventional spiking neural network simulators if the POETS simulator can perform an order of magnitude better than BRIAN in the time taken to simulate a network. This metric was chosen because it would change the way that researchers in the field of spiking neural networks do their work, the time taken to run a test would drop dramatically.

7.1 Evaluation methods

During evaluation, the number of synapses each neuron was scaled from 10% to 100%. The number of neurons was also scaled from one hundred up to one million. Increasing the number of synapses should be more computationally complex since the number of messages will be vastly increased. The performance was measured by checking the time taken to complete 1000ms of simulated time, as well as monitoring the number of messages sent.

The hardware tests were run on a box known as Ayres containing two FPGA triplets with sixteen four-core tiles containing 64 threads each[33]. The total number of threads available to this box is therefore 6144. The BRIAN models are being run on a server with 135GB of memory, and an i7 7800X with 12 cores. This should provide enough performance for BRIAN's multi-threading abilities to be shown. Since POETS is a very expensive research computer, it makes sense for BRIAN to be run on hardware with more power than a laptop computer so that the results are more comparable.

7.1.1 Scaling performance

There are two possible types of scaling metrics that can be looked at:

- **Weak scaling** involves monitoring how the time to run the simulation varies with the number of threads that are used. This requires increasing the size of network so that the number of threads it uses increases proportionally, and then monitoring the changes in performance. The overheads required to generate the network were considered separately in the overheads section as they aren't relevant to performance.
- **Strong scaling** would involve monitoring how the time to simulate varies with the number of threads for a fixed network size. In this benchmark, the aim would be to try and split the workload between multiple threads in different ways to check the differences in performance. Due to the nature of POETS, it isn't possible to control the placement of devices so strong scaling wasn't used as a metric.

7.1.2 Overheads

This involved finding out how much time it takes to generate the model for POETS and how long it takes to get the results after running the network. This was done on by comparing a fixed size network on both POETS and BRIAN. With POETS, there are more stages in network generation than with BRIAN, as seen in Figure 21. Throughout the development of the simulator, it was discovered that there are a lot of overheads. This is because the network is required to be compiled to an XML file (for large network sizes the file generation can take nearly an hour), and then compiled to C++ for running on the POETS hardware.

It was expected that the overheads for POETS will be significantly higher than for BRIAN, however this wasn't the case as will be seen. It is a useful benchmark to determine whether the performance of POETS can significantly outweigh the overheads associated with compiling its networks.

7.2 Issues that arose during evaluation

The memory required for compiling turned out to be significant and limited the capability to simulate large networks containing more than around ten million total synapses. Despite the XML files being less than 10GB in size, the compiler requires more than the maximum 135GB of memory available to it on the Ayres box, and crashes before compilation can complete. This unfortunately means that the largest, most useful, and most interesting networks couldn't be tested. This is especially disappointing as despite compile times being more than five times as long as the simulations, the simulations themselves performed well.

Another unfortunate major issue that arose was the failure of the Hardware-Barrier method. This method had worked fine on the software simulator performing better than the clocked method and rivalling the GALS method, but when run on the POETS hardware, this method ended up hanging. All neurons initialised correctly but beyond that the network never progressed. The initial starting value of the membrane potential was increased such that it would force a fire on the first time step but this still didn't cause the network to function. The network never went into hardware idle mode and therefore it never continued. Other than this issue, the networks all functioned as expected.

The final problem was getting the total number of messages sent. To keep track of messages sent, the neurons must have an internal parameter that is incremented when they send a message. Issues arise when at the end of a simulation a user wishes to print these messages. This is extremely difficult to do as the simulation terminates when any neuron reaches the maximum number of time steps specified for a simulation. If the neurons are set up to log their message count state when the function terminates they won't. This is because the function that ends the simulation is only called on the device that activates it. This means that in the case of GALS, it isn't possible to return the message count of every neuron

without constantly logging every time a message is sent. This is alright for small networks, but for large networks there are bottlenecks as the logging tools are slower than the rest of the hardware. This causes the simulation to slow down which isn't desired. It therefore isn't possible to track messages for most of the network types. The exception to this is the clocked network. Since the clock neuron is connected to all the other neurons, it can keep track of the messages sent. In theory, it is possible to add a device whose only job is to gather all the message values from every neuron at the end of the simulation, but this wasn't implemented due to time constraints. It is also possible there is a way to dump the state variables of all devices when the simulation ends but there was no documentation on that.

7.3 Results and evaluation

7.3.1 Scaling performance

The time required for BRIAN to run a simulation varies only slightly with the specified size. All networks with less than 100 million synapses take less than 1 minute to run. The largest BRIAN model that could be generated contained 900 million synapses and took 30 minutes to run. There is a fixed amount of time that is needed to start the simulation of around 3 seconds. After that, the amount of time to run a network on BRIAN barely changes until the network gets very large. This can be seen in Figure 29:

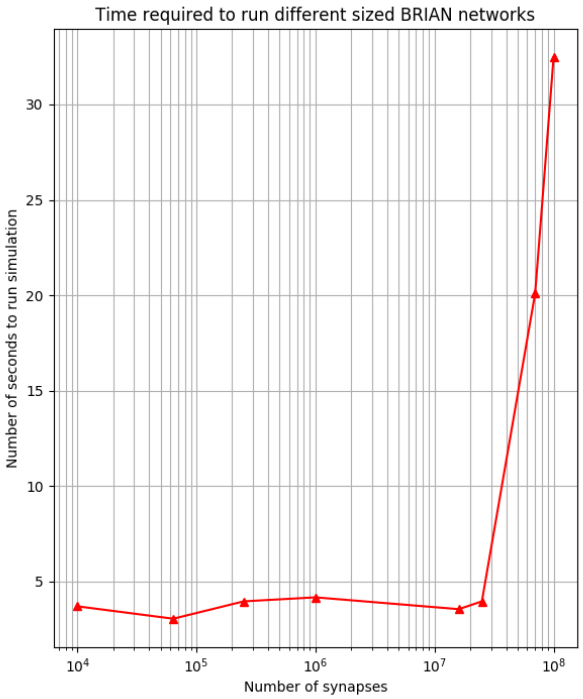

Figure 29: The time needed to run BRIAN networks of different sizes. It is easy to see that there is a startup time for even the small networks. The 900 million synapse network result wasn't shown since it skews the graph. That network takes 8840 seconds to run.

The scaling performance for the various POETS networks generally matches up with the asymptotic analysis. This can be seen in Figure 30. The clocked network requires more time to run than the other network types due to it's use of a clock neuron. The largest POETS networks that were run were 100 thousand neurons in size with 100 synapses per neuron (10 million synapses). The clocked network took 33 minutes to run whilst the same sized GALS network took 20 minutes to run. Smaller networks were much more similar in the time needed to run with. A fully-connected, 1000 neuron clocked network takes 2 minutes 50 seconds to run whilst the equivalent GALS network took 2 minutes 39 seconds to run.

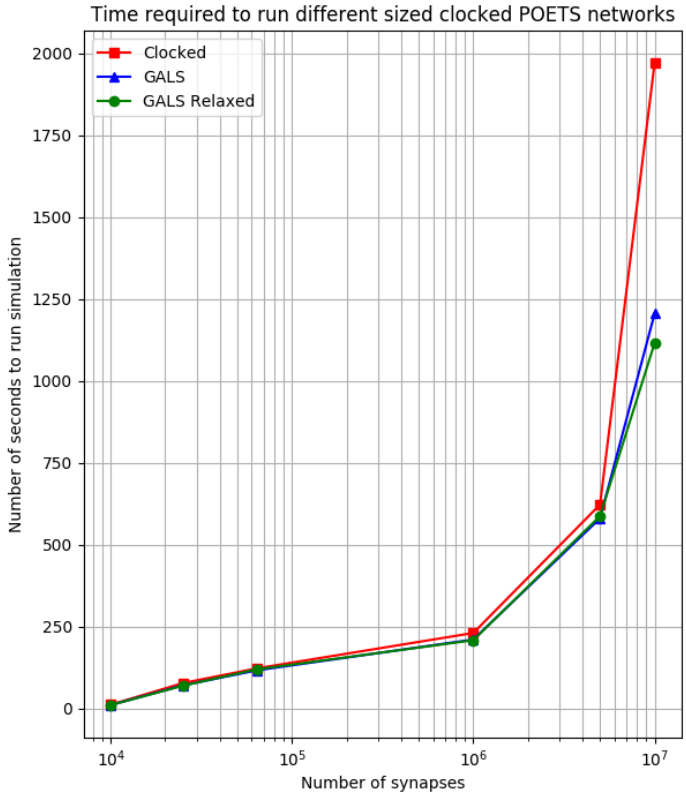

Figure 30: The time needed to run a clocked POETS network of several different sizes. Despite the relaxations used on relaxed GALS, it still performs very similarly to normal GALS.

The networks that sacrifice accuracy for performance don't outperform the accuracy based networks, the largest non-synchronised network took a very similar amount of time to run as GALS at 19 minutes. The extremely relaxed model hangs during execution no matter the size of network. This seems to happen when the difference between the current maximum time step and the time step received is zero. This continued to occur even when the algorithm was modified. Whilst the non-synchronised network performed well, it only spiked around 1 thousand times which is significantly lower than 3 thousand plus that it should have been. All the other networks that ran successfully had final spike counts of within 5% of each

other which implies that they were running correctly. It is expected that the message counts between simulation runs won't be identical because POETS cannot be seeded and thus even identical runs of the same network won't necessarily result in the same message count. The clocked networks had exponentially more total messages sent which matches up with the asymptotic analysis.

Varying the duration of a simulation results in a linear increase in time for the POETS networks. For the BRIAN networks the duration only increases very slightly. This can be seen in Figure 31:

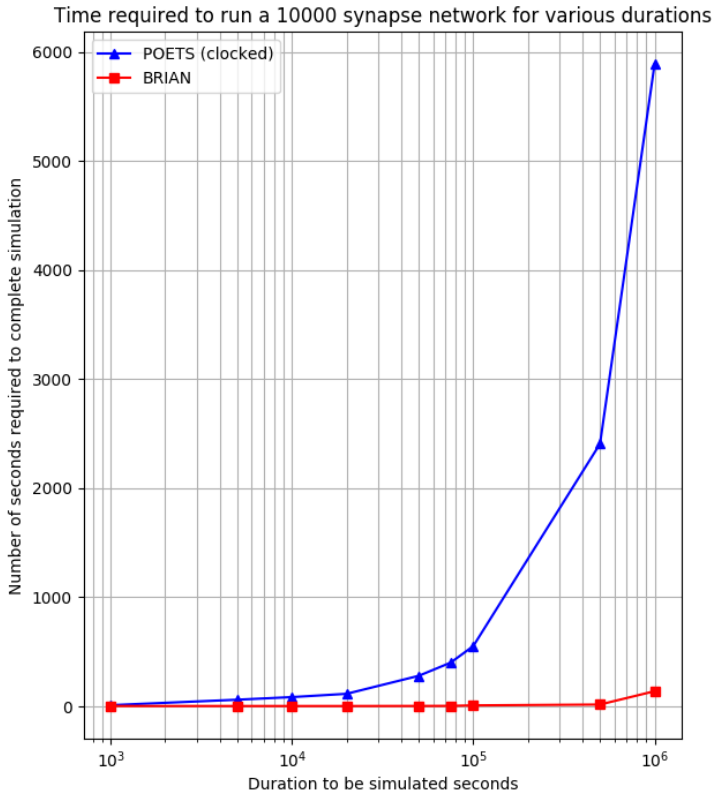

Figure 31: The time needed to run a 10000 synapse clocked network for several different durations. The difference in time needed to simulate networks for different durations is much greater for POETS' than for BRIAN.

7.3.2 Overheads

The time taken to compile a POETS hardware model is independent of the amount of time to be simulated. This is expected as the only difference between the models when the simulation time is changed is a single parameter representing duration. The BRIAN model takes a different amount of time to generate a network based on duration. This can be seen in Figure 32. The POETS compile time includes the XML generation as well as the time needed to compile the XML for POETS.

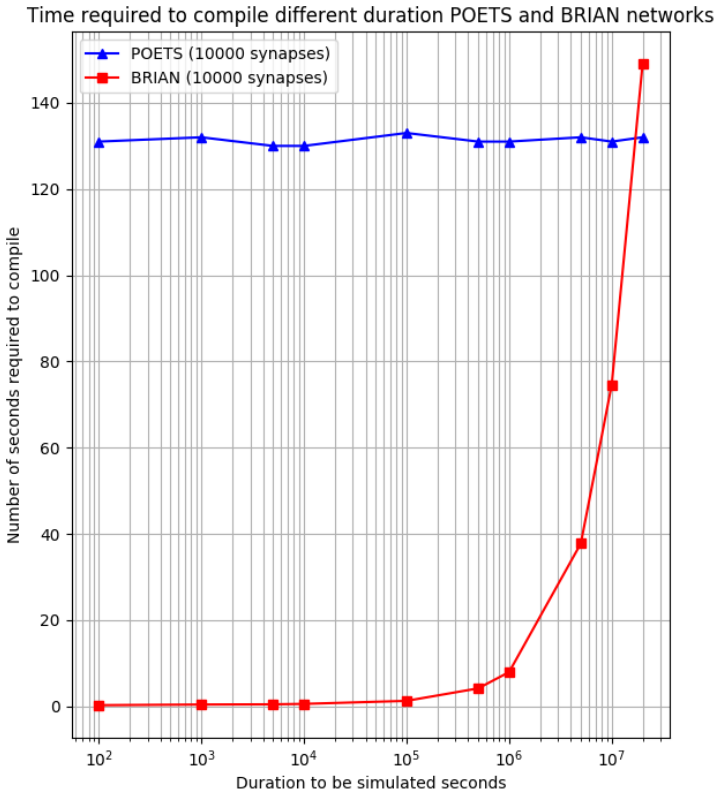

Figure 32: The compile times for 10000 synapse BRIAN and POETS networks for different durations. POETS compilation times are independent of the time to be simulated, whilst BRIAN's aren't. Above a duration of 20000000 seconds, POETS is quicker at compiling (in the case of a 10000 synapse network).

For small networks, BRIAN has better performance than the POETS hardware when it comes to compiling the models. A fully-connected, 10 thousand neuron BRIAN network takes 78 minutes to run whilst a fully-connected, 10 thousand neuron network on POETS fails to compile due to memory issues after just over 4 hours. The largest POETS networks at 10 million synapses took 180 minutes to compile and an equivalently sized BRIAN network took less than 15 minutes. The largest BRIAN network that was compiled was a fully-connected, 30 thousand neuron network and took 11.8 hours to compile.

Whilst BRIAN can compile networks with more total synapses, it falls short when compiling models with large numbers of neurons, as seen in Figure 33. The reason for this is that when the synaptic connections are specified with a probability, BRIAN must go through every pair of neurons and calculate whether a synapse should be connected. It must do this regardless of whether the neurons are connected or not. POETS neurons are specified using a list and therefore only synapses that are specified are evaluated. BRIAN doesn't allow the

synapses to be specified using an adjacency matrix instead requiring either a probability for the connections or a generator that specifies how the neurons should be connected. For fully-connected networks, using a generator results in a performance increase as the connection between a pair of neurons is guaranteed. When using a probability of 1, BRIAN will still generate a random number and check whether it's less than 1 which is a pointless calculation. For any other probabilities, the generator takes the same amount of time to compile as just specifying the probability.

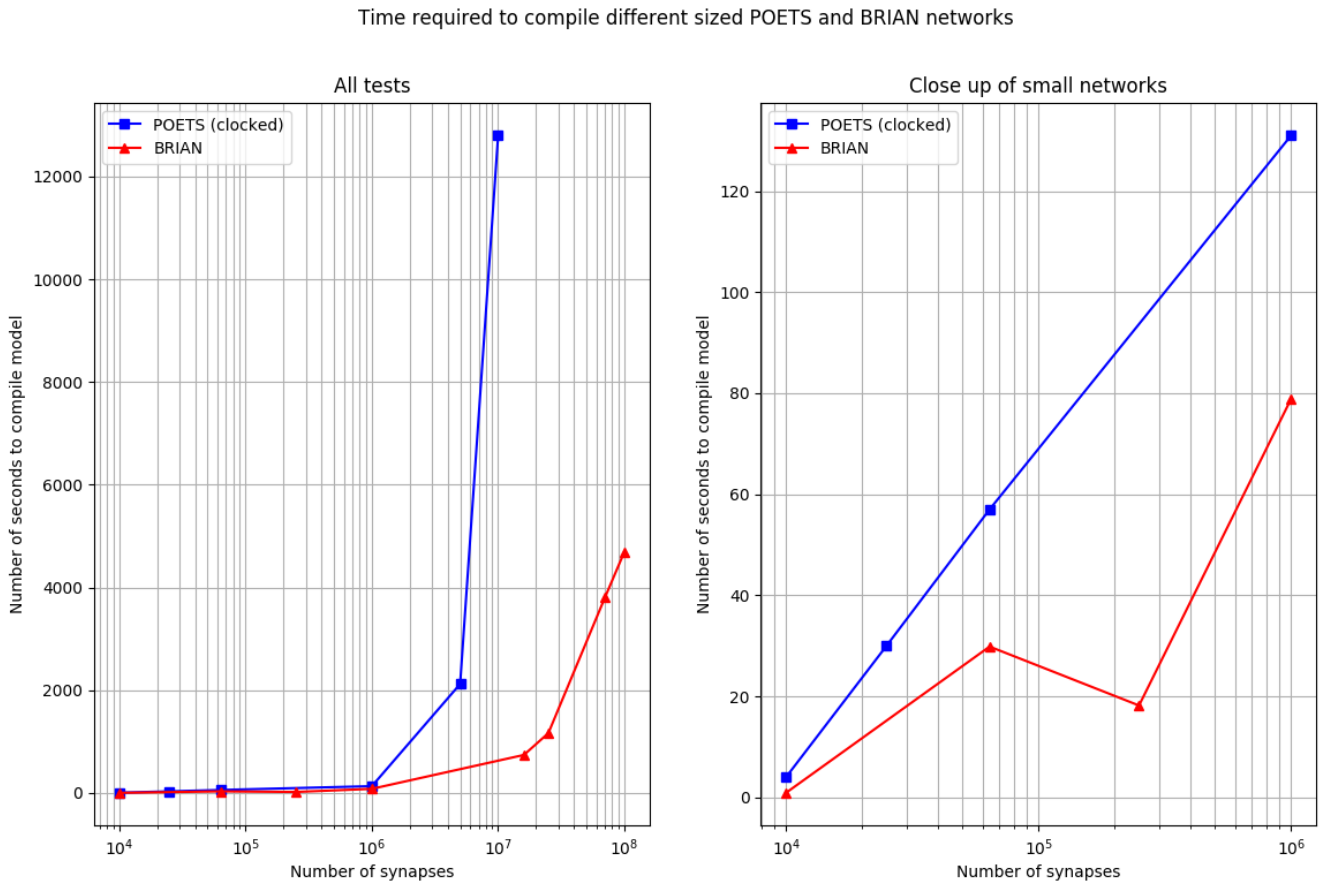

Figure 33: BRIAN has better compile times than the POETS compiler when the duration is kept the same.

8 Conclusions and Further Work

Some elements of the results were surprising. Neither POETS nor conventional simulators came out as having clearly better performance when overheads were included in the evaluation. For small duration networks, BRIAN is significantly better, but as the duration is increased, BRIAN requires more resources than POETS. It is possible that at scales beyond those tested in this project, POETS could prove to be superior to BRIAN. This would require improvements to be made to the POETS compiler and memory.

8.1 Conclusions

In conclusion, whilst POETS is a powerful piece of hardware, it isn't significantly better than conventional simulators such as BRIAN. POETS isn't suitable for running spiking neural networks. The reason for this is that POETS is designed for asynchronous tasks and spiking neural networks are not entirely asynchronous. Therefore, they aren't suitable to be run on POETS if one wants to utilise POETS asynchronous capabilities.

Both BRIAN and POETS ran into memory issues when generating the networks. The networks that were being generated weren't particularly large but required significantly more memory than was available to the systems running the simulations. This may not be an issue in the coming years as the density of memory chips increases but for the time being it proves to be a large bottleneck.

After BRIAN networks have been compiled, they run a lot faster than similarly sized POETS networks. BRIAN also use less memory than POETS for generating its networks and compiles them in less time. Thus, when considering the initial hypothesis that POETS may be better for running spiking neural networks than conventional simulators such as BRIAN, the answer is no. This isn't entirely unexpected as BRIAN has had significant amounts of effort spent developing it and is also designed specifically for spiking neural networks whilst POETS is not. There is a chance for POETS. When the BRIAN network was scaled from ten million synapses to nine hundred million synapses, the time taken to run a simulation increased more than was expected as the simulation time rose from thirty seconds to nearly three hours. This was an extremely large jump in the time required to run as previous increases in BRIAN model sizes were significantly smaller. The time taken for POETS to complete a simulation consistently increased proportional to a fraction of the increase in synapses. It is, therefore, possible that POETS networks of around a billion synapses may start to compare favourably with BRIAN models. Beyond this scale, POETS may overtake BRIAN. This would require improvements to the POETS compiler since it isn't currently possible to compile networks of that size.

Despite performance of POETS networks not being great, it is now much easier to specify them using the software provided in section 6. The Gaussian random number generator has been improved and now functions correctly. Networks that are too large to be compiled on POETS can be specified and have their XML generated so that if the compiler improves in the future, this software can still be used. The algorithms that were implemented for synchronising the networks functioned as planned in section 4 and based on the asymptotic analysis, the hardware barrier model could provide an extremely good approach if it can be implemented on the POETS hardware.

In the future, improvements to POETS and the various tooling used to generate spiking neural networks may mean that POETS surpasses conventional simulators such as BRIAN, but currently that is not the case. For hardware to be superior to conventional simulators, it needs to be specifically designed for spiking neural networks like SpiNNaker was. A system for simulating spiking neural networks needs to have support in place for both the macro level asynchronous network, as well as the micro level synchronous neurons. POETS currently doesn't have the capabilities for this.

8.2 Further work to improve POETS

Better POETS compiler

The initial plan for the project involved simulating large networks containing one million to ten million neurons, each with about one thousand synapses. Whilst the software developed to create the XML files is able to produce networks of this size, the software used to compile the files to C++ cannot successfully compile them. Despite the XML files being less than 10GB in size, the compiler reaches the maximum 135GB of memory available to it before crashing. This means that large networks cannot be properly tested. Improvements to compiler memory usage were not solved in the course of this project, as they are reliant on researchers who host POETS at the Cambridge computer laboratory implementing features such as these.

An alternative to this could be using IO to change parameters of the network. If the logic of a network is sound then it might be more viable to allow the option of passing configuration to the network instead of compiling from XML every sing time a parameter is modified. This would be very useful if the compiler couldn't be improved and could make spiking neural networks on POETS significantly more viable for deep learning.

Better evaluation tools

Currently, it is only feasible to test the output of small networks. The only logging available is a function which prints information to stdout. This means it is difficult to graph results. Code was developed that would parse the log files and plot the information as a graph since there is no software available. For easier debugging and to make it easier to interpret results, better evaluation tools would be useful that didn't rely on parsing a log file.

"String not found at address" error

For small networks, the logging tool sometimes outputs the error `String not found at address 0x81ea`. This error only seems to occur for small networks with less than one thousand neurons. This hasn't caused issues for this project since it doesn't cause crashes it just causes the logging tools to not work properly. The existence of this error could imply that there are more bugs within POETS.

POETS is an old project and has been through many iterations since its inception. Many features have become deprecated and the software has changed significantly over time. It is therefore expected that there are bugs in the system. Whilst these are most likely to be minor, if they could be fixed then that would be desirable.

Improving device level clocking

Spiking neural networks are only asynchronous at the macro level. Since neuron membrane potential is dependent on time based differential equations, there needs to be some system in place to keep the neurons synchronised. POETS doesn't have that capability by default meaning that a significant part of this project was attempting to keep the neurons synchronised. As seen in this project, the overheads caused by this were too large for it to successfully compete with BRIAN.

An improvement to POETS would be a method to clock individual devices. If this was done in hardware then it would vastly improve performance. Unfortunately, this is not what POETS is designed for and therefore this version of POETS probably won't include it.

Fixing hardware-barrier model

The hardware-barrier model was the most promising network throughout the project. It outperformed the other networks when it was run on the software based simulator but it unfortunately failed to run on the actual POETS hardware. If this model could be made to work on the POETS hardware, it would likely outperform the other methods in both time to run as well as accuracy.

8.3 Further work to improve tooling for SNN simulator

Further improve accuracy of simulator

Whilst I have proposed some reasons why the POETS network didn't produce results exactly the same as the NumPy model, there wasn't time to dive deep into how POETS evaluates its internal logic. Given the complexity of POETS, this would be a considerable body of work. Such an investigation into the root cause of the difference between the models could be the subject of a future Imperial final year project in itself.

Adding input and output

The current implementation doesn't contain any way to provide input to the network. If POETS is to be used with broader tasks such as classification or other tasks usually done with artificial neural networks, this would be important to implement.

Allowing the simulator to adapt and learn

A common feature of spiking neural networks is their ability to change their weights in a similar way to how classical artificial neural networks work. With this ability, POETS could be used to run spiking neural networks in applications such as classification and regression. It might be the case that POETS high performance nature allows spiking neural networks to be used in more applications than currently. The time to train could be significantly quicker than on normal computers.

Some ideas for learning would be spike-dependent-timing-plasticity or Hebbian learning that were described earlier in the report. It is significantly more difficult to train spiking neural networks than artificial neural networks due to a lack of techniques like backpropagation but if this feature could be included in the simulator, it would make the simulator more useful. Using spiking neural networks on POETS for tasks such as these could be a good final year project for a future student, provided that the compilation time could be improved. Without improving compilation time, it would be very difficult to rapidly test different network types. Another issue is the lack of IO currently implemented. POETS does have the ability to use inputs and outputs but it wasn't used during this project due to not being within this projects scope.

9 References

[1] *Partial Ordered Event Triggered Systems About Page*, 2020. [Online]. Available: https://poets-project.org/about/.

[2] W. Maass, "Networks of Spiking Neurons: The Third Generation of Neural Network Models," Tech. Rep. 9, 1997, pp. 1659–1671.

[3] A. Brown, D. Thomas, J. Reeve, G. Tarawneh, A. De Gennaro, A. Mokhov, M. Naylor, and T. Kazmierski, "Distributed event-based computing," Tech. Rep., 2018.

[4] D. Goodman and R. Brette, "Brian: A simulator for spiking neural networks in python," *Frontiers in Neuroinformatics*, vol. 2, no. NOV, Nov. 2008, ISSN: 16625196. DOI: 10.3389/neuro.11.005.2008.

[5] A. Zeil, R. O. Verlag, and M. Wien, "Simulation neuronaler Netze," Tech. Rep., 1997.

[6] Yann Cun, "A Theoretical Framework for Back-Propagation," *The proceedings of the 1988 Connectionist Models Summer School*, pp. 21–28, 1988.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, ISSN: 14764687. DOI: 10.1038/nature16961.

[8] L. Bottaci, P. J. Drew, J. E. Hartley, M. B. Hadfield, R. Farouk, P. W. R. Lee, I. M. C. Macintyre, G. S. Duthie, and J. R. T. Monson, "Artificial neural networks applied to outcome prediction for colorectal cancer patients in separate institutions," Tech. Rep., 1997.

[9] S. Lowel and W. Singer, "Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity," *Science*, vol. 255, no. 5041, pp. 209–212, 1992. DOI: 10.1126/science.1372754.

[10] P. R. Montague and T. J. Sejnowski, "Regulation of Synaptic Efficacy by Coincidence of Postsynaptic APs and EPSPs The increase in," Tech. Rep., 1982, p. 253.

[11] T. Iakymchuk, A. Rosado-Muñoz, J. F. Guerrero-Martínez, M. Bataller-Mompeán, and J. V. Francés-Víllora, "Simplified spiking neural network architecture and STDP learning algorithm applied to image classification," *Eurasip Journal on Image and Video Processing*, vol. 2015, no. 1, 2015, ISSN: 16875281. DOI: 10.1186/s13640-015-0059-4.

[12] A. D. Brown, J. E. Chad, R. Kamarudin, K. J. Dugan, and S. B. Furber, "SpiNNaker: Event-based simulation - Quantitative behavior," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 450–462, Jul. 2018, ISSN: 23327766. DOI: 10.1109/TMSCS.2017.2748122.

[13] P. Pomienski, "Compiling Spiking Neural Networks to Event Driven Hardware," Tech. Rep., 2018.

[14] *Human Brain Project*, 2017. [Online]. Available: https://www.humanbrainproject.eu/en/.

[15] E. M. Izhikevich, *Simple model of spiking neurons*, Nov. 2003. DOI: 10.1109/TNN.2003.820440.

[16] F. R. Freemon, "The Synaptic Organization of the Brain," *JAMA*, vol. 243, no. 18, p. 1850, May 1980, ISSN: 0098-7484. DOI: 10.1001/jama.1980.03300440052037. [Online]. Available: https://doi.org/10.1001/jama.1980.03300440052037.

[17] NumPy Team, *NumPy*, 2020. [Online]. Available: https://numpy.org/.

[18] *Floating Point Numbers*, 2020. [Online]. Available: https://floating-point-gui.de/formats/fp/.

[19] N. J. Highamt, "THE ACCURACY OF FLOATING POINT SUMMATION*," Tech. Rep. 4, 1993, pp. 783–799. [Online]. Available: http://www.siam.org/journals/ojsa.php.

[20] W. Kahan, "Pracniques: Further Remarks on Reducing Truncation Errors," *Commun. ACM*, vol. 8, no. 1, p. 40, Jan. 1965, ISSN: 0001-0782. DOI: 10.1145/363707.363723. [Online]. Available: https://doi.org/10.1145/363707.363723.

[21] *Matplotlib*, 2020. [Online]. Available: https://matplotlib.org/.

[22] Meng Xiannong, *Linear Congruential Method*, 2002. [Online]. Available: https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node40.html.

[23] G. E. P. Box and M. E. Muller, "A Note on the Generation of Random Normal Deviates," *Ann. Math. Statist.*, vol. 29, no. 2, pp. 610–611, Jun. 1958. DOI: 10.1214/aoms/1177706645. [Online]. Available: https://doi.org/10.1214/aoms/1177706645.

[24] J. F. Hart, *Computer Approximations*. USA: Krieger Publishing Co., Inc., 1978, ISBN: 0882756427.

[25] Mahmoud Hesham El-Magdoub, *Best Square Root Method - Algorithm - Function (Precision VS Speed)*, 2010. [Online]. Available: https://www.codeproject.com/Articles/69941/Best-Square-Root-Method-Algorithm-Function-Precisi.

[26] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, Jul. 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[27] Brumme Stephan, *Float square root approximation*, 2020. [Online]. Available: https://bits.stephan-brumme.com/squareRoot.html.

[28] Mineiro Paul, *Fast Approximate Logarithm, Exponential, Power, and Inverse Root*, 2011. [Online]. Available: http://www.machinedlearnings.com/2011/06/fast-approximate-logarithm-exponential.html.

[29] M. Pfeiffer and T. Pfeil, "Deep Learning With Spiking Neurons: Opportunities and Challenges," *Frontiers in Neuroscience*, vol. 12, Oct. 2018, ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00774.

[30] *Introduction to Brian part 2: Synapses*, 2016. [Online]. Available: https://brian2.readthedocs.io/en/stable/resources/tutorials/2-intro-to-brian-synapses.html.

[31] Suciu Dan, "XML Compression," in *Encyclopedia of Database Systems*, Springer, 2016, pp. 1–6.

[32] Gigi Sayfan, *Understand How Much Memory Your Python Objects Use*, 2016. [Online]. Available: https://code.tutsplus.com/tutorials/understand-how-much-memory-your-python-objects-use--cms-25609.

[33] M. Naylor, S. W. Moore, and D. Thomas, "Tinsel: a manythread overlay for FPGA clusters," Tech. Rep., 2019.

10 Appendix

10.1 Running the XML generator

The code is available on GitHub at `https://github.com/joshjennings98/fyp`.

To run the XML generator, a configuration file must be specified. The repository contains a config file that can be used to generate the XML needed to run a spiking neural network. Once a configuration file has been made, pass the name of the configuration file as a parameter to `network_generator/main.py`

Network parameters should be split using semicolons and in cases where parameters can have multiple expressions, these expressions should be separated by commas. Things that must be specified include: name, equations, threshold, resets, inits, neuron parameters, number, maxt, and type. To specify multiple sets of neuron parameters, list them as `parameters 1` and `parameters 2` etc. For a more thorough examples, see the example configuration files in the GitHub repository.

There is no set way to produce a config file as in theory any network can be specified. To see an example that implements the Izhikevich spiking neural network, look at `config` in the GitHub repository.