Entropy Bounds for Glass Networks

by

Benjamin Walter Wild
B.Sc. Honours, University of Victoria, 2022

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Mathematics and Statistics

Entropy Bounds for Glass Networks

by

Benjamin Walter Wild
B.Sc. Honours, University of Victoria, 2022

Supervisory Committee

Dr. Roderick Edwards, Supervisor
(Department of Mathematics and Statistics)

Dr. Anthony Quas, Departmental Member
(Department of Mathematics and Statistics)

ABSTRACT

Electronic circuitry based on chaotic Glass networks, a type of piecewise smooth dynamical system, has recently been proposed as a potential design for true random number generators. Glass networks are good designs due to their potential for chaotic behaviour and because their analytic tractability allows us here to propose a method for approximating their entropy, a measure of irregularity in dynamical systems. We discuss some of the historical developments that led to the interest in the model that we consider within the context of random number generation. Additionally, we discuss a method for converting a Glass network's governing piecewise-smooth differential equations into discrete-time dynamical systems, and then into symbolic dynamical systems. We also detail how the symbolic entropy of the given Glass network is bounded above by the entropy of the symbolic dynamical system formed from its transition graph, a type of directed graph that represents the possible transitions in phase space between regions not containing discontinuities. We then extend previous results by detailing our new method of refining the transition graph to be a more accurate depiction of the true system's dynamics, making use of more specific information about trapping regions in phase space. Refinements come in the form of splitting nodes and duplicating/partitioning edges on the transition graph and removing those that are never realized by the continuous dynamics. We show that refinements can be done to arbitrary levels and in the limit as the level of refinement goes to infinity, the entropy of the refined transition graphs converges to the true entropy of the system. Along with this, since it is not possible to calculate the limiting value, approximation is necessary. Doing this by hand is tedious and difficult, so as a result, we also detail here an algorithm we devised that automates the refinement process, allowing for approximation (from above) of symbolic entropy. Various examples are considered throughout and we also discuss how numerical simulation can be used to non-rigorously estimate symbolic entropy, as an independent (approximate) verification of our results. Finally, we detail some unfinished and future work which could extend our results further, along with alternative methods to achieve similar and potentially even stronger results. With our results and algorithm, using upper bounds on a Glass network's symbolic representation's entropy is now a viable method for assessing the irregularity of its dynamics.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

I want to acknowledge my parents Norman and Julietta, and my sister Sally for their love and support throughout my whole life. My countless uncles, aunts, and cousins. My friends Alex, Will, Nick, Donovan, and all my friends I've made while at UVic. My supervisor Dr. Rod Edwards for putting up with me during the time that we worked together. The professors of the math department that taught and helped me during my time as a student here. Many helpful conversations with Dr. Anthony Quas. Countless helpful conversations with Rod. My parents, the faculty of Graduate studies, Rod, and my BCGS scholarship for financial support throughout my time as a graduate student at UVic. Finally, I want to acknowledge the students attending UVic that work hard and deserve to be there.

DEDICATION

I dedicate this entire body of work to my parents, Norman and Julietta, whose love and support throughout my whole life is the reason this was possible.

I love you, Mum and Dad.

La famiglia è tutto

Chapter 1
Introduction to the problem

In the modern world, computers are a part of almost every aspect of life. From banking information to health care, almost everything has some amount of data attached to it. Many data storage systems have aspects that users wish to keep restricted from public access. To keep data private, cryptography is at the forefront. In the years since the first data storage systems, cyber security threats have advanced dramatically, necessitating continual advancements in encryption methods. Modern data encryption often involves using random number generators (RNGs) to generate random bit sequences. The metric by which the randomness of a bit sequences is measured is entropy [1]. The higher the entropy, the more random the sequence. Hence, RNGs with a high entropy production rate are of interest to anyone who wishes to keep encrypted data safe. There are many different designs for RNGs, however, the most robust to hacking attempts are the class of RNGs known as true random number generators (TRNGs). TRNGs are usually constructed as physical devices, typically electronic circuits. They often rely on thermal and physical noise, present in all electric circuits, to incorporate random "jitter" to make generated bit sequences unpredictable. However, most RNGs have inherent deficiencies that clever hackers and scammers can take advantage of to disrupt random bit sequences and extract data. As a result, it is important to design physical implementations of these TRNGs to be as robust to attack as possible. A common TRNG design is that of the ring oscillator [57, 35]. Ring oscillators rely on thermal noise in the physical circuit to cause a random drift in phase to generate entropy. However, these circuits are inherently flawed in that they can be easily entrained to display predictable behaviour [48, 47, 2]. It has been suggested that using circuitry with inherently chaotic underlying dynamics [6, 37, 55, 56, 58, 16] may be more robust than classical ring oscillators to entrainment based attacks. The most notable discussion around chaotic free running circuitry has been in the context of the Chua circuit [49, 20, 42], but more recently, circuit designs based on chaotic Glass networks have been proposed as potential designs for TRNGs [16, 46, 59]. Glass network designs have the advantage of being based on standard logic gates that are readily and cheaply accessible, rather than special purpose hardware. It is believed that due to their

potential for irregular dynamics [9, 10, 11, 44, 51, 33, 4, 60], along with a dynamically diverse set of network structures, many different designs [16] could be constructed that in theory could generate enough entropy to effectively encrypt data and be robust to common hacking attempts. To better understand why using circuits based on Glass networks are being suggested as designs for TRNGs, it is important to understand some of the history that lead to this point. In this chapter we will discuss where these models originally came from along with some of the history[1] around analysis that lead to the idea of using them as TRNGs. However, before we can discuss Glass networks or their analysis in any detail, we must first go back their genesis. This comes from the study of genetic regulation.

1.1 Glass networks: a model of genetic regulation

A seminal work in the theory of genetic regulation is that of Jacob and Monod [38]. They argued that genetic regulation can be divided into components that, when linked together, may be responsible for many of the complex cellular functions present in genetics. Much of the early mathematical study of these regulatory systems involved models that contain complex functional responses [28, 29] that mimic the switching-like behaviours seen in genetic regulation[2]. A simple example of this is presented by Glass and Kauffman [29], wherein the model's switching phenomena are described using Hill functions [22, 36] as follows:

$$\frac{dx_1}{dt} = -\lambda x_1 + \gamma \frac{\theta^n}{\theta^n + {x_2}^n},$$
$$\frac{dx_2}{dt} = -\lambda x_2 + \gamma \frac{\theta^n}{\theta^n + {x_1}^n}.$$

Hill functions, first derived in modelling of chemical kinetics, are similarly applicable to gene regulation and take the form

$$h_1(x) = \frac{x^n}{\theta^n + x^n}, \text{ and } h_2(x) = \frac{\theta^n}{\theta^n + x^n},$$

where θ is the switching threshold and n is the Hill coefficient. Note that $h_2(x) = 1 - h_1(x)$.

While there were many different types of models that were studied early on, a

[1] Within this chapter we will not provide much mathematically rigorous description of Glass networks, but instead will provide a historical overview of the work that has lead to the starting point for work we have done in later chapters. A mathematically rigorous description of Glass networks is provided in chapter 2.

[2] Much of the inspiration for many of the early models that we are studying came from the study of chemical kinetics in which similar switching behaviours could be observed to those present in genetic regulation.

simplified version that is of particular interest here can be written as follows:

$$\frac{dx_i}{dt} = -\lambda_i x_i + S_i(x),\ i \in \{1, \ldots, n\}$$

where $x \in \mathbb{R}^n$, $\lambda_i \in \mathbb{R}$, and $S_i(x)$ is a functional response modelling switching, typically in the form of sigmoid or Hill functional type responses. Hill functions are commonly used to study simplified models of gene regulation due to their simple structure and from more detailed mass-action models of catalytic reactions. However, even in this simplified model, analysis becomes very difficult when the systems of equations become large, which is often the case when studying gene regulation. To make analysis more tractable, an ingenious variant of the model can be considered.

With a focus on Hill functions, taking the limit as $n \to \infty$, the Hill function h_1 (and similarly for h_2) becomes the Heaviside step function

$$H(x) = \begin{cases} 1,\ x \geq \theta \\ 0,\ x < \theta, \end{cases}$$

offset to the switching threshold value θ, and gives rise to the model called Glass networks. Written as follows

$$\frac{dx_i}{dt} = -\lambda_i x_i + \Gamma_i(x),\ i \in \{1, \ldots, n\} \tag{1.1}$$

where $x \in \mathbb{R}^n$, $\lambda_i \in \mathbb{R}$, and $\Gamma_i(x)$ is a piecewise constant coupling function. Edwards, Farcot, Glass, and others have published many articles focusing particular attention on this simplified model due to its potential for rich dynamics along with simplicity in structure.

Using the Heaviside step function in place of a continuous switching functional response gives rise to a Boolean structure in the model. The idea of using a Boolean structure to try and understand the complex phenomena present in genetic regulation was first introduced by Kauffman in the late 1960's [39]. However, it was not until works by Kauffman and Glass [39, 28, 29, 23, 25, 30, 24] in the 1970's that a continuous-time mathematical formalism developed[3].

Most of the switching behaviour that happens in biological systems is not instantaneous; it often takes significant time to progress. Step functions were introduced as a way to develop a qualitative model that would allow progress in models where continuous switching terms make analysis difficult. Many of the common continuous switching functional responses previously discussed come with severe difficulties that make classical methods of analysis difficult in most cases. The piecewise constant functions

[3]Kauffman championed the discrete-time synchronous update Boolean model; Glass followed up on the continuous-variable, continuous-time, Boolean response function model.

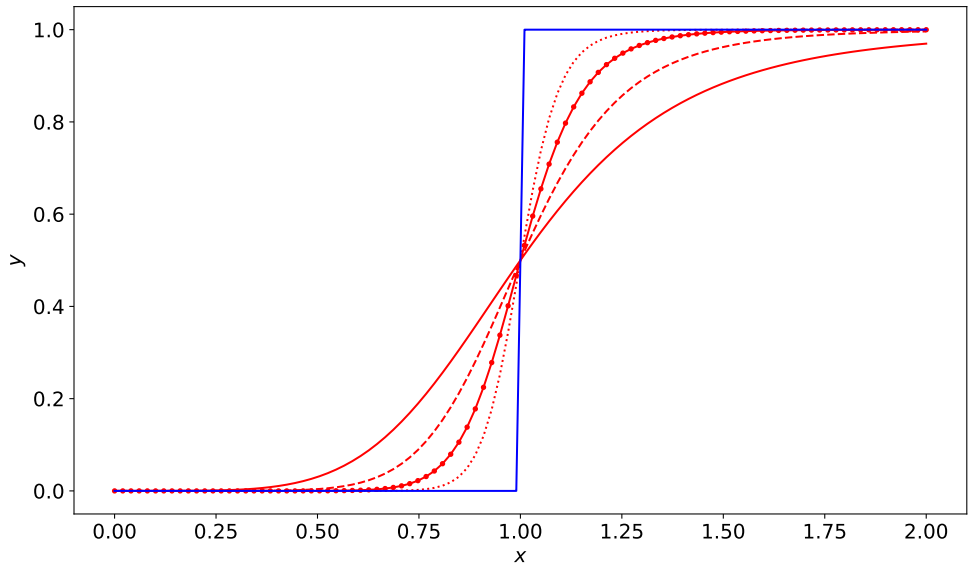


Figure 1.1: Example $h_1(x)$ Hill functions (red) and Heaviside step function (blue), where switching thresholds are $\theta = 1$ and the Hill parameters are $n = 5, 8, 13$, and 21 for the solid, dashed, dashed dotted, and dotted lines respectively.

used in place of continuous switching functions give systems of non-smooth ordinary differential equations (ODEs) that are linear between discontinuities. Hence, between discontinuities solutions are easy to construct and analysis is more easily conducted.

In these simple models of gene regulation, it has been argued that the biologically relevant dynamics produced by models using step function approximations behave qualitatively similarly to those that use the standard Hill or sigmoid type functional responses. However, since the models that Glass networks are used to approximate are often too difficult to make progress with, the loss in information from the step function is made up for in the tractability of analysis. As a result, Boolean structures can be used in this case to derive qualitative results about the genetic systems.

1.2 Basic structure used in analysis

We now focus our discussion on the main method of analysis. Since their inception, the interest in Glass networks has spread beyond just genetic regulation, but also to their properties as mathematical objects. While originally Glass networks were proposed in the context of their application to genetics, much of the body of work on them since is concerned with their mathematical properties and potential for complex dynamics.

Glass networks fall into the vast category of dynamical systems labeled as non-

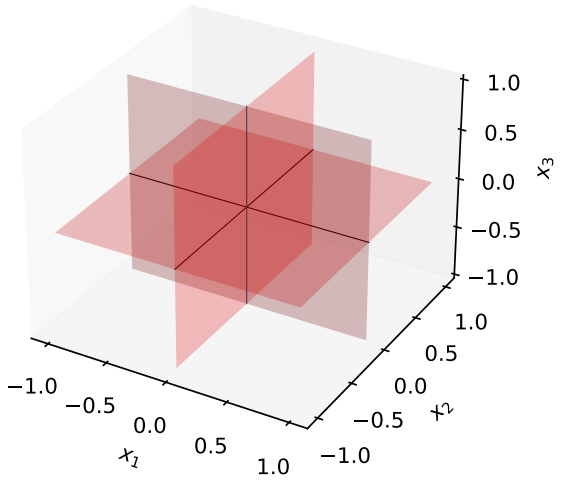


Figure 1.2: Example of how boxes partition space in a 3 dimensional Glass network where each variable has a single switching threshold, all at 0. In this case, each box is just one of the 8 octants of the classical cartesian coordinates of $\mathbb{R}^3$. The red planes indicate the threshold boundaries.

smooth [17], meaning there exist sudden changes and discontinuities within the governing system and dynamics. Non-smooth dynamical systems present many challenges that are unheard of in classical dynamical systems. However, within non-smooth dynamical systems, Glass networks fall into the sub-category labeled piecewise-smooth [7]. As far as non-smooth dynamical systems go, piecewise-smooth systems are more reasonable to deal with because outside the regions in their domain where discontinuities exist, they are smooth.

In Glass networks, the discontinuities come from step functions in the ODEs. As a result, these discontinuities are represented by hyperplanes in phase space that form rectangular-shaped regions historically referred to as boxes. An example of this is shown in figure 1.2. The hyperplanes that form the boundaries of boxes have historically been referred to as walls. Within a given box, Glass networks are smooth, so the ODEs can be easily solved to construct solutions between the hyperplane boundaries. In the case that trajectories are not directed towards a point within the given box the solutions will eventually intersect a switching boundary, dictated by the flow from the governing ODEs. Once a trajectory has intersected a switching boundary, the solution changes to match the conditions within the new box and continues, unless the flow is towards the wall from both sides, where special methods of analysis are required, but this is not considered here. It is usually of interest to follow the flow of trajectories in Glass networks from wall to wall across boxes. In this case, the time for a trajectory to cross a given box is simple to calculate and solutions across the box can be represented by discrete mappings from wall to wall. Then, they can be stitched together to form

a global mapping for the system. Using the walls as a domain for this mapping, Glass network dynamics can then be represented using a discrete dynamical system[4]. Not every wall can be reached from every wall. The allowed wall-to-wall transitions are ultimately dictated by the flow from the governing ODEs. This flow can be represented by a directed graph, commonly referred to as a transition graph (TG). Boxes are represented by nodes and walls by directed edges where the direction of the edge portrays the direction of flow across that particular threshold boundary.

Once a Glass network is converted into a discrete dynamical system, all the tools from classical discrete dynamical systems are at one's disposal and thus open up a vast sea of potential analysis. This, along with the relevance to genetics and biology, have been much of the reason for the advancement in Glass network theory. Much of the early work with Glass networks was concerning periodic behaviour [30, 31, 32] within the context of gene regulation. That is one of the types of behaviour that is expected in those systems. However, the structure of Glass networks does not exclude the possibility for more complex behaviour. While there is not much known relevance of the discussion of chaos in Glass networks to their original use in genetic regulation, discussions of chaos in Glass networks [14, 43, 51, 11, 4, 60, 10, 44, 50] have garnered particular attention over the decades of their study. Even with the advance in analytical tractability that comes with the discrete dynamical system representation of Glass networks, analysis in the context of chaos does present many challenges. Because of the difficulty of the mathematically rigorous study of dynamical chaos, many non-trivial techniques have been employed over the years, one of the most powerful within the context of Glass networks being symbolic dynamics [14, 45].

1.3 Chaos and symbolic representation

The equivalent description of Glass network dynamics by discrete dynamical systems, put many more analytical tools at ones disposal, one of the most powerful being Poincaré sections and mappings. Walls and appropriate compositions of wall-to-wall mappings can be used to define Poincaré sections and mappings respectively. Many results on stable periodic orbits and positively invariant regions can be obtained in this way, positively invariant regions opening the door for discussions around irregular dynamics.

For Glass networks, an effective way of deducing that an example may have rich dynamics is to show that there exist no stable fixed points, no stable periodic orbits, and that there exists a positively invariant region that all trajectories must eventually fall into. An excellent demonstration of this is provided by Edwards in his 2001 article

[4]The details of this conversion to a discrete dynamical system are discussed in great detail by Farcot [15] and a summary of his general description is given in Chapter 2.

discussing chaos in Glass networks [10]. This is not a proof of chaos and such a proof is non-trivial and requires some work. However, it is a way of deducing aperiodic behaviour.

As defined by Devaney in his book on chaotic dynamical systems [5], a dynamical system can be said to be chaotic if its mapping satisfies the following definition:

Definition 1.1. Let V be a set. $f : V \to V$ is said to be chaotic on V if f has the following three properties:

1. periodic points are dense in V;
2. f is topologically transitive;
3. f has sensitive dependence on initial conditions.

However, it is often difficult to show these properties explicitly in complicated, high dimensional, systems. In discussions of chaotic dynamics in Glass networks, the discrete dynamical system representation is particularly useful as it can be used to demonstrate the existence of a horseshoe map. Demonstration of a horseshoe mapping in a dynamical system is one way to conclude that chaotic dynamics is present. Explicit demonstration of horseshoe maps in Glass networks has been shown by Edwards, Farcot, and Foxall [11], and by Li and Yang [44]. In most cases, however, demonstrations such as these are non-trivial and require a lot of work.

Horseshoe maps are not the only indicator of chaos in a dynamical system. A common method for demonstrating chaos in classical dynamical systems is the calculation of a positive maximal Lyapunov exponent. While not a direct proof of the phenomenon, a positive maximal Lyapunov exponent does indicate sensitive dependence on initial conditions. However, there are many situations where this information is not sufficient for irregular dynamics and can be misconstrued, e.g. linear systems with positive eigenvalues. One of the best indicators of chaotic dynamics in a dynamical system is the presence of positive topological entropy. Defined by Adler, Konheim, and McAndrew in their original paper from 1965 [1], topological entropy is defined as follows:

Definition 1.2. Let X be a compact topological space. For any open cover $\mathscr{U}$ of X, let $N(\mathscr{U})$ denote the number of sets in the subcover of minimal cardinality. Define $H(\mathscr{U}) = \log N(\mathscr{U})$ to be the entropy of $\mathscr{U}$. For any two covers $\mathscr{U}$, $\mathscr{B}$, their join is defined by $\mathscr{U} \vee \mathscr{B} = \{A \cap B | A \in \mathscr{U}, B \in \mathscr{B}\}$. The entropy $h(f, \mathscr{U})$ of a mapping f with respect to a cover $\mathscr{U}$ is defined as

$$\lim_{n\to\infty} \frac{H(\mathscr{U} \vee f^{-1}(\mathscr{U}) \vee \cdots \vee f^{-n+1}(\mathscr{U}))}{n}. \tag{1.2}$$

The entropy $h(f)$ of a mapping f is defined as $\sup h(f, \mathscr{U})$ where the supremum is taken over all open covers $\mathscr{U}$.

Now, entropy is a topological invariant. Hence, it is a good quantity to use when trying to assess irregularity of dynamical systems and compare against other dynamical systems. However, it is not a trivial quantity to calculate and within the context of the generalized framework that we wish to study (Glass networks), this is even more the case. There have been some discussions on topological entropy within the context of Glass networks. However at the point of writing this, there exists no comprehensive survey in which a general methodology is presented for practical calculation of topological entropy. Thus, taking advantage of its ability to identify potentially irregular dynamics is less feasible within the context of Glass networks. There is a more tractable quantity within the context of Glass networks that is of particular relevance to RNGs: symbolic entropy.

Within dynamical systems, symbolic dynamics is a powerful tool that has seen success in many areas of study. The mathematical objects of interest in symbolic dynamics are called shift spaces, and as it happens, Glass networks can be represented by shift spaces. As defined by Lind and Marcus [45], the full shift is defined as follows:

Definition 1.3. If $\mathcal{A}$ is a finite alphabet, the full $\mathcal{A}$-shift is the collection of all bi-infinite sequences of symbols from $\mathcal{A}$. The full r-shift (or simply r-shift) is the full shift over the alphabet $\{0, 1, \ldots, r\}$. The full $\mathcal{A}$-shift is denoted by

$$\mathcal{A}^{\mathbb{Z}} = \{x = (x_i)_{i \in \mathbb{Z}} : x_i \in \mathcal{A} \text{ for all } i \in \mathbb{Z}\}$$

A shift space [45] is defined as follows:

Definition 1.4. A shift space (or simply shift) is a subset X of a full shift $\mathcal{A}^{\mathbb{Z}}$ such that $X = X_{\mathcal{F}}$ for some collection $\mathcal{F}$ of forbidden blocks over $\mathcal{A}$. A block (or word) over $\mathcal{A}$ is a finite sequence of symbols from $\mathcal{A}$. A forbidden block for a shift space X is simply a block that never appears in any point in X.

A shift space on its own is not a dynamical system. However, there exists the shift map σ, defined below, that paired with a shift space defines a type of dynamical system commonly referred to as a symbolic dynamical system.

Definition 1.5. The shift map σ on the full shift $\mathcal{A}^{\mathbb{Z}}$ maps a point x to the point $y = \sigma(x)$ whose i^{th} coordinate is $y_i = x_{i+1}$.

Using shift spaces, dynamical properties are transferred from the original system to the infinite symbol sequences. This often makes interpretation simpler, especially when studying chaos. An impressive use of symbolic dynamics in Glass network analysis was performed by Edwards et al [14]. They showed that bifurcations of chaotic dynamics correspond to changes in the associated language.

Since symbolic dynamical systems are themselves dynamical systems, there is also a notion of topological entropy. As defined by Lind and Marcus [45], the entropy of a symbolic dynamical system is defined as follows:

Definition 1.6. Let X be a shift space. The entropy of X is defined by

$$h(X) = \lim_{n\to\infty} \frac{1}{n} \log_2 |\mathcal{B}_n(X)| \tag{1.3}$$

where $\mathcal{B}_n$ denotes the set of all n-blocks that occur in points in X. Note that for shift spaces this definition is equivalent to the one in definition 1.2.

In a shift space, entropy can be thought of as a measure of the system's capacity to display complex information. In general, even for shift spaces entropy is difficult to calculate. However, if a shift space can be represented by a directed graph, the entropy is simple to calculate. It is just $\log_2$ of the largest eigenvalue of the graph's adjacency matrix [45].

The piecewise structure of Glass networks that partitions phase space into boxes allows a natural definition of an alphabet of symbols. Associating points on walls with the boxes they are sandwiched between is a natural way for trajectories to be encoded. Encoding dynamics in this way is not a new idea and (in the context of Glass networks) has been discussed at length by Farcot [15], and more generally by Fu et al [19].

Since the TG is a directed graph, it can also be used to define a shift space. However, while dynamics in phase space have corresponding dynamics on the TG, the converse is often not true. Hence, the shift space representation of a network's trajectories is merely a subset of the shift space defined using the TG. Hence, the entropy of the TG is an upper bound for the entropy of the true dynamics. Entropy estimates have been discussed by Glass et al. [14], but most in depth by Farcot [15]. The entropy estimate provided by Farcot, while an impressive result in its own right, leaves much to be desired. Using the entire TG as an upper bound often times allows for significantly more dynamical possibilities than the actual dynamics will ever realize. In particular, it is possible for the TG to give a positive upper bound when the dynamics is periodic and the true entropy is 0. As a result Farcot's upper bound cannot really be used as an effective method for quickly assessing the symbolic irregularity of a network. In order to make symbolic entropy a viable metric for Glass networks, theory that allows for actual computation of entropy or at least much better methods of computing upper bounds are required.

1.4 Glass Networks as RNGs

As we have just seen, the theory and applicability of Glass networks extends much beyond just simple models of gene regulation. In genetic regulatory networks, the Glass network approximation is a bit crude and leads to the aforementioned qualitative results. However, in systems where switching occurs on much faster time scales, the non-smooth approximation is much more appropriate and has been shown to exhibit

dynamics that are very similar. This is due to the speed at which the switching takes place. In an electronic circuit the switching happens very quickly, whereas, in the context of genetic regulatory networks, switching time can be significant. This makes Glass networks of particular interest in cybersecurity applications. The results achieved can be taken more at face value since the switching speed of logical gates is quite accurately modelled using step functions.

Since there exist many example Glass networks that display potential for chaotic dynamics, circuit designs based on Glass networks have been proposed as potential designs for robust TRNGs [16]. The idea behind this is that using an already chaotic system in a physical circuit implementation with thermal noise present in the circuits make it more difficult to entrain to a strong periodic signal in a frequency-injection attack than is the case for a simple noisy oscillator. Glass networks have the potential for chaotic dynamics and are easy to construct using standard Boolean logic gates, and so they are a perfect candidate for TRNGs especially because of the potential variability in designs. There exist many simple examples that vary dramatically in their dynamics [12].

RNG randomness is typically assessed using the amount of entropy generated by extracted bit sequences. Currently, there exist no general methods for finding the entropy of a Glass network. This makes the process of trying to design TRNGs from Glass networks a significant challenge since other methods used to demonstrate chaos are by no means trivial. So, in order to design circuits based on Glass networks with the highest possible entropy, a method of calculating, or at least approximating, the topological entropy of a given Glass network would be greatly beneficial in the design process of these TRNGs. We suggest that symbolic entropy is a good metric by which to do this. In many examples, the upper bound achieved by Farcot [15] is too loose to be useful and no results on lower bounds exist. Hence, if we wish to use symbolic entropy, we require a better method of estimating upper bounds.

This is what we have done.

Chapter 2
Entropy bounds for Glass networks

In the previous chapter we introduced Glass networks from a historical context surrounding their use in gene regulation and their proposed application to cyber security as TRNGs. Along with a discussion on their analysis, we also introduced the notion of applying symbolic dynamics to Glass networks, an invaluable tool for dealing with chaos. The contents of this chapter will consist of our paper [59] that we submitted to the Society for Industrial and Applied Mathematics Journal on Applied Dynamical Systems for review. Herein, we discuss the structure of Glass networks from a rigorous view point along with providing a method for approximating entropy to any level of accuracy.

The paper included below begins with a brief account of the generalized construction by Farcot [15] that allows for Glass networks to be described as one-sided discrete dynamical systems along with a symbolic representation, whose topological entropy is bounded by the entropy generated by its TG. We then expand upon this by considering more information about dynamics from the continuous system to arbitrarily refine this upper bound by considering separated cycle representations of the TG that forbid cycle concatenations not realized in the network. We conclude the results of our paper by showing that these refinements converge to the symbolic entropy of the true dynamics. Additionally, we provide a numerical check to see how fast an example network's refinements converge.

This paper is significant because our method is simple and the convergence result gives a simple method of approximating a network's symbolic entropy. This work puts forth a mathematical formalism that allows for assessment of the irregularity of a network as compared to others with similar structure. It is our hope that our method can be used to gain easy insight and serve as an analysis tool for those wishing to study complex dynamics in Glass networks.

Entropy bounds for Glass networks

Benjamin W. Wild and Roderick Edwards

First submission submitted to SIADS on 2024-04-23

First revision submitted to SIADS on 2024-10-29

Abstract

We develop a procedure here to calculate good upper bounds on the entropy of Glass networks (a class of piecewise-linear Ordinary Differential Equations), by means of symbolic representations of the continuous dynamics. Our method improves on a result by Farcot (2006), and allows in principle arbitrary refinements of the estimate, and we show that in the limiting case, these estimates converge to the true entropy of the symbolic system corresponding to the continuous dynamics. As a check on the method, we demonstrate for an example network that our upper bound after only a few refinement steps is very close to the entropy estimated from a long numerical simulation. This procedure could be applied to estimating entropy of free-running electronic circuits built from standard Boolean logic gates, which can be modelled by Glass networks.

Keywords

Glass networks, entropy, symbolic dynamics, chaos, piecewise smooth ODEs, dynamical systems

MSC codes

34A36, 34C28, 37B10, 37B40, 94C30

2.1 Introduction

Glass networks are a class of piecewise-linear systems of ordinary differential equations (ODEs), which were originally proposed as qualitative models of gene regulation [13, 23, 24, 25, 26, 27, 30], but have also been implemented in electronic circuitry [16, 33, 46] where the sharpness of the switching between high and low states makes the

model a more accurate approximation of such circuit behaviour than of typical gene networks where switching may not be so steep. Demonstration of chaotic, or at least aperiodic behaviour in low-dimensional examples of such networks has been shown in several ways [9, 10, 11, 44, 51], and there is a renewed interest in establishing such complex behaviour because of their potential application as robust True Random Number Generators (TRNGs) [16].

Entropy is one measure of complexity in a dynamical system [1]. It is particularly relevant in the context of TRNGs, since the objective in that situation is to generate sequences of bits with positive entropy. For Glass networks, entropy estimates have been discussed by Glass et al. [14] and more rigorously by Farcot [15]. Farcot obtains an upper bound on entropy of the dynamics of a Glass network, by means of the entropy of a Transition Graph (TG) associated with the equations. The TG captures the transitions between states (rectangular regions of phase space, called *boxes*) that are allowed by the structure of the equations. The entropy of this TG is shown, using a corresponding symbolic dynamics, to be an upper bound for the actual dynamics of the network. However, it is often not a very sharp upper bound, since the TG typically allows a great many more transitions than actually occur in a trajectory of the differential equations, especially if the latter is restricted to an attractor (*i.e.*, ignoring any transient behaviour), which is the pertinent object for the behaviour of a circuit acting as a TRNG.

Here we show how Farcot's approach to producing an upper bound on entropy for a Glass network can be improved, in some cases dramatically improved, by taking into consideration more information about the actual dynamics of the network. As a first step, the TG can be pruned to remove transitions that do not occur on the attractor of interest. Periodic attractors in Glass networks can often be identified directly, and proven to exist and to be asymptotically attracting. Of course, they must have zero entropy, but one can also sometimes prove that no such stable periodic orbit exists. A trapping region on a *wall* between boxes, serving as a Poincaré section, can often be identified, even when there is no stable periodic orbit, which restricts trajectories to a certain subgraph of the TG. Furthermore, it is often possible to refine the entropy estimate further by identifying sequences of cycles on the Poincaré section that do not occur, and thus correspond to forbidden blocks in the corresponding symbolic dynamics. We use a particular Glass network as a test example, but the methodology is applicable to the entire class of networks considered.

In Section 2.2, we introduce briefly Glass networks and the analysis of their dynamics, mainly following the notation of Farcot [15]. In Section 2.3 we outline Farcot's proof that the entropy of the TG is an upper bound for the entropy of the network dynamics. Then in Section 2.4 we adapt this approach to show how improved upper bounds may be obtained. In Section 2.5 we apply this analysis to a particular Glass network. Finally, in Section 2.6 we show by means of numerical simulations how much

closer our upper bound is to numerical estimates of entropy than the entropy based on the entire TG.

2.2 Glass Networks

2.2.1 General theory

Glass networks are of the form

$$\frac{dx}{dt} = -\Lambda x + \Gamma(x) \tag{2.1}$$

where $x \in \mathbb{R}^n$, $\Lambda \in \mathbb{R}^{n\times n}$ is a positive diagonal matrix with diagonal entries λ_i, $i = 1, \ldots, n$, and $\Gamma : \mathbb{R}^n \to \mathbb{R}^n$ is constant on each of a finite number of rectangular regions that partition phase space defined by thresholds, as detailed below. The following background on the structure and analysis of Glass networks closely follows Farcot [15] since this work builds upon what is discussed there.

Since Γ takes only a finite number of values, Γ is bounded and so is the flow. The dynamics of interest occur within the n-dimensional rectangular region

$$\mathcal{U} = \prod_{i=1}^{n} \left[\min_{x\in\mathbb{R}^n} \left\{ \frac{\Gamma_i(x)}{\lambda_i} \right\}, \max_{x\in\mathbb{R}^n} \left\{ \frac{\Gamma_i(x)}{\lambda_i} \right\} \right],$$

which is positively invariant. The flow from any (non-negative) initial point outside $\mathcal{U}$ must flow into it and cannot then escape. Note that Farcot [15], defines $\mathcal{U}$ as $[0, 1]^n$, but this would only be true after a normalization that we do not make. Since Λ is a constant diagonal matrix, Equation (2.1) only describes systems with linear degradation rates. Thus, we do not here include systems with coupling in the degradation term. $\mathcal{U}$ is partitioned by the piecewise definition of the coupling term, Γ, which is constant on the interior of *boxes*, n-dimensional rectangular regions that are products of bounded intervals. Thus, in each box Equation (2.1) is a first order linear system.

The set of boundary values are denoted by $\Theta_i = \{\theta_{i,j} \,|\, j \in \{0, ..., p_i\}\}$ for each coordinate x_i of x. Like Farcot [15], we will assume that the sets Θ_i are ordered $\theta_{i,0} < \theta_{i,1} < ... < \theta_{i,p_i-1} < \theta_{i,p_i}$, where $\theta_{i,0} = \min_{x\in\mathbb{R}^n} \left\{ \frac{\Gamma_i(x)}{\lambda_i} \right\}$ and $\theta_{i,p_i} = \max_{x\in\mathbb{R}^n} \left\{ \frac{\Gamma_i(x)}{\lambda_i} \right\}$, and the set of *threshold* values is $\{\theta_{i,j} \,|\, j \in \{1, ..., p_i - 1\}\}$. Boxes can then be written as

$$B_a = B_{a_1 \ldots a_n} = \prod_{i=1}^{n} [\theta_{i,a_i}, \theta_{i,a_i+1}] \tag{2.2}$$

where a belongs to the finite set

$$\mathcal{A} = \prod_{i=1}^{n} \{0, ..., p_i - 1\} . \tag{2.3}$$

The boundary values define the boundaries of a finite number of boxes in phase space, on the interior of each of which Γ is constant. These boundaries in phase space are referred to as *walls*. Since the subscript a uniquely determines a box, $\mathcal{A}$ defines an alphabet of symbols that will later be used in symbolic dynamical systems that qualitatively describe the dynamics of System (2.1). Since Γ is constant on the interior of a box, it will also be convenient later to consider the mapping Γ as $\Gamma : \mathcal{A} \to \mathbb{R}^n$. ($\Gamma$ should really be considered as the composition of a map from $\mathbb{R}^n$ to $\mathcal{A}$ and a map from $\mathcal{A}$ to $\mathbb{R}^n$, but it will not cause confusion to abuse notation and refer to the latter map as Γ).

Equation (2.1) can be solved within a given box B_a, where Γ is a constant vector and the system is first order linear, to give:

$$x(t) = f(a) + e^{-\Lambda t}(x(0) - f(a)) \tag{2.4}$$

where $f = \Lambda^{-1} \circ \Gamma = [\Gamma_1/\lambda_1, ..., \Gamma_n/\lambda_n]^\top$, and $f(a)$ is referred to as a focal point for the box B_a. The flow is clearly attracted towards $f(a)$ (in each variable). If $f(a) \in B_a$, $f(a)$ is an asymptotically stable steady state. Otherwise, before the trajectory can reach $f(a)$, it will intersect a wall between B_a and another box. If the wall is crossed then Γ takes a new constant value (in general) and the trajectory continues. The boundary of B_a is formed by k-faces that are k-dimensional rectangles where $k \in \{0, ..., n-1\}$. When the trajectory crosses an $(n-1)$-dimensional face (a wall) there is at most one adjacent box the trajectory can continue in, and so the value of Γ is unambiguous, unless the flow is towards the wall on both sides. In the latter case (a *black wall*), the wall is not crossed. Rather, the subsequent flow is constrained to the wall, and we need either Filippov theory for discontinuous systems, or singular perturbation theory to determine the flow. If the trajectory crosses a face that has dimension less than $n-1$ there are several possible adjacent boxes that the trajectory could continue into, and again Filippov solutions or singular perturbation is sometimes required. Here, we will restrict the discussion to walls, *i.e.*, faces of dimension $n-1$, that are actually crossed. These are called *transparent* walls.

Continuous trajectories from wall crossing to wall crossing can be easily constructed, as long as walls are transparent. In order to ensure this, we will assume that the following two conditions are met.

Condition 1. $\forall\, a \in \mathcal{A},\ f(a) \in \bigcup_{b \in \mathcal{A}} \mathrm{int}(B_b)$.

Here $f : \mathcal{A} \to \mathbb{R}^n$ gives the focal point for the box B_a, and $\mathrm{int}(B_a)$ is the interior of

B_a. This assumption ensures that no focal points are located on a box boundary.

Condition 2. *$\forall\, i \in \{1, ..., n\}$, $\forall\, a, a' \in \mathcal{A}$ such that $a - a' = \pm \boldsymbol{e}_i$, either*

$$(\boldsymbol{d}_i(f(a)) - a_i)\,(\boldsymbol{d}_i(f(a')) - a'_i) > 0,$$

or

$$(\boldsymbol{d}_i(f(a)) - a_i) = 0 \quad \textit{and} \quad (\boldsymbol{d}_i(f(a')) - a'_i)\,(a_i - a'_i) > 0$$

or

$$(\boldsymbol{d}_i(f(a')) - a'_i) = 0 \quad \textit{and} \quad (\boldsymbol{d}_i(f(a)) - a_i)\,(a_i - a'_i) > 0$$

where $\boldsymbol{e}_i$ is the i^{th} basis vector in $\mathbb{R}^n$ and $\boldsymbol{d} = (\boldsymbol{d}_1, ..., \boldsymbol{d}_n) : \bigcup_a int(B_a) \to \mathcal{A}$ is the discretizing operator, which maps a point lying inside a box to the subscript denoting the box.

This Condition ensures that the flow on both sides of a wall between adjacent boxes is in the same direction, so that black (or white) walls cannot occur. Boxes a and a' are adjacent in the x_i direction when $a - a' = \pm \boldsymbol{e}_i$, and the flow direction in box a of variable x_i is determined by the sign of $(\boldsymbol{d}_i(f(a)) - a_i)$. In the context of gene regulation, Condition 2 is satisfied when there is no direct auto-regulation in the system.

Given a box B_a and the solution trajectory defined by Equation (2.4) within the box, if a wall is crossed, the time and position that the trajectory encounters the wall is easy to calculate. Which wall, if any, is encountered uniquely depends on the position of the focal point for the box. The wall $\{x_i = \theta_{i,a_i}\}$ can be crossed if and only if $f_i(a) < \theta_{i,a_i}$ and the wall $\{x_i = \theta_{i,a_i+1}\}$ can be crossed if and only if $f_i(a) > \theta_{i,a_i+1}$. The set of possible output walls, either on the upper or lower side in each direction, is thus captured by

$$I^+_{\text{out}}(a) = \{i \in \{1, ..., n\} \,|\, f_i(a) > \theta_{i,a_i+1}\}$$

and

$$I^-_{\text{out}}(a) = \{i \in \{1, ..., n\} \,|\, f_i(a) < \theta_{i,a_i}\}$$

with

$$I_{\text{out}}(a) = I^+_{\text{out}}(a) \cup I^-_{\text{out}}(a)\,.$$

Note that if $i \in I^+_{\text{out}}(a)$ then $i \notin I^-_{\text{out}}(a)$ and vice versa. Now for each direction in $I_{\text{out}}(a)$, the time it takes for $x(t)$ to reach the threshold from an initial point, $x^{(0)}$ according to the flow of B_a is given by

$$\tau_i(x^{(0)}) = -\frac{1}{\lambda_i} \ln\left(\alpha^-_i(x^{(0)})\right) \quad \text{if} \quad i \in I^-_{\text{out}}(a)$$

and

$$\tau_i(x^{(0)}) = -\frac{1}{\lambda_i} \ln\left(\alpha_i^+(x^{(0)})\right) \quad \text{if} \quad i \in I_{\text{out}}^+(a)$$

where

$$\alpha_i^-(x^{(0)}) = \frac{f_i(a) - \theta_{i,a_i}}{f_i(a) - x_i^{(0)}} \quad \text{and} \quad \alpha_i^+(x^{(0)}) = \frac{f_i(a) - \theta_{i,a_i+1}}{f_i(a) - x_i^{(0)}}\,.$$

Finally, the time to encounter a wall from $x \in B_a$ is given by

$$\tau(x^{(0)}) = \min_{i \in I_{\text{out}}(a)} \tau_i(x^{(0)}) \tag{2.5}$$

and putting this into Equation (2.4) gives the exit point of B_a from the initial condition $x^{(0)} \in B_a$. The starting point of a trajectory can be chosen to be on a wall, and then a the subsequent sequence of wall transitions can be calculated by repeating the above. It follows from Equation (2.4) and Equation (2.5) that the map from wall to wall through a box is given by $\mathcal{M}_a : \partial B_a \to \partial B_a$ where

$$\mathcal{M}_a x = x(\tau(x^{(0)})) = f(a) + e^{-\Lambda\tau(x^{(0)})}(x^{(0)} - f(a))\,. \tag{2.6}$$

Within each box B_a, $I_{\text{out}}(a)$ determines all boxes that are reachable from B_a. These boxes are adjacent to B_a through walls supported by hyperplanes that take the form $\{x_i = \theta_{i,j}\}$, where $i \in I_{\text{out}}$ and $j \in \{a_i, a_i + 1\}$ depends on whether i belongs to $I_{\text{out}}^-(a)$ or $I_{\text{out}}^+(a)$. Using that distinction, walls can be denoted as

$$W_i^+(a) = \{x \,|\, x_i = \theta_{i,a_i+1}\} \cap B_a \quad \text{and} \quad W_i^-(a) = \{x \,|\, x_i = \theta_{i,a_i}\} \cap B_a\,.$$

Each box can then be partitioned into regions associated with each element of $I_{\text{out}}(a)$ where only a single adjacent box is reachable from each region. The only walls by which trajectories can escape B_a are then $W_i^+(a)$ for $i \in I_{\text{out}}^+(a)$, and $W_i^-(a)$ for $i \in I_{\text{out}}^-(a)$. So, given any initial condition x on a wall, the directions i such that $\tau(x) = \tau_i(x)$ are those for which $\mathcal{M}_a x \in W_i^+(a) \cup W_i^-(a)$. Normally there is only one such direction and a particular wall through which we exit B_a, but it is possible that multiple walls are reached simultaneously (*i.e.*, we reach an intersection of threshold hyperplanes), in which case there is more than one such direction, i. Now ∂B_a can be partitioned into two regions:

$$\partial B_a^{\text{out}} = \left(\bigcup_{i \in I_{\text{out}}^+(a)} W_i^+(a)\right) \cup \left(\bigcup_{i \in I_{\text{out}}^-(a)} W_i^-(a)\right) = \{x \in B_a \,|\, \tau(x) = 0\}$$

(points on the boundary of B_a from which it takes no time at all to exit, *i.e.*, exit

walls), and

$$\begin{aligned}\partial B_a^{\text{in}} &= \overline{\partial B_a \setminus \partial B_a^{\text{out}}} \\ &= \left(\bigcup_{i \in I_{\text{out}}^+(a)} W_i^-(a) \right) \cup \left(\bigcup_{i \in I_{\text{out}}^-(a)} W_i^+(a) \right) \cup \bigcup_{i \notin I_{\text{out}}(a)} \left(W_i^-(a) \cup W_i^+(a) \right) .\end{aligned}$$

Thus, the incoming and outgoing regions are unions of walls, which are closed and cover the boundary of ∂B_a. It then follows that $\partial B_a^{\text{out}} \cap \partial B_a^{\text{in}} \neq \emptyset$ whenever $\partial B_a^{\text{out}} \neq \emptyset$, since it includes some threshold intersections. Under Condition 1, it always holds that $\partial B_a^{\text{in}} \neq \emptyset$ because $i \in I_{\text{out}}^+(a) \Rightarrow W_i^-(a) \subset \partial B_a^{\text{in}}$, $i \in I_{\text{out}}^-(a) \Rightarrow W_i^+(a) \subset \partial B_a^{\text{in}}$, and $i \notin I_{\text{out}}(a) \Rightarrow W_i^-(a) \cup W_i^+(a) \subset \partial B_a^{\text{in}}$. This first partition of the boundary of B_a only allows for a distinction to be made between exit directions and the others. It follows that $\partial B_a^{\text{out}} = \emptyset$ if and only if $f(a) \in \text{int}(B_a)$. Using this partition, it is proven by Farcot [15] that under the assumptions of Condition 1 and provided $\partial B_a^{\text{out}} \neq \emptyset$, when $\mathcal{M}_a$ is restricted to the domain and range ∂B_a^{in} and $\partial B_a^{\text{out}}$ respectively, $\mathcal{M}_a$ is a homeomorphism. It is pointed out by Farcot that in the case where $\lambda_i = \lambda \ \forall i$, all the inequalities that define the domains and ranges of the map $\mathcal{M}_a$ through a box B_a are affine, so that the regions are polytopes. In the examples we investigate later, this will be the case.

Thus far, the transition mapping $\mathcal{M}_a$ has been rigorously defined as a homeomorphism within the confines of a single box. Now we turn our attention to a transition map defined globally on the whole space. Although maps defined within boxes with nonempty outgoing domains are invertible, boxes with no escaping direction are more problematic. It is natural to map the boundary of these types of boxes to a single point whose pre-image is the entire box boundary. As a result, in general a global mapping will not be invertible at all points. This leads us to the consider only forward iterates of the map in the global space.

In fact, Condition 2 implies that any outgoing wall $W \subset \partial B_b^{\text{out}}$, for some b, is part of ∂B_a^{in}, for B_a adjacent to B_b at wall W, unless W is only a wall for B_b, when it lies on the boundary of the whole domain $\mathcal{U}$. However, Condition 1 implies that in this case $W \subset \partial B_b^{\text{in}}$. Thus,

$$\bigcup_{a \in \mathcal{A}} \partial B_a = \bigcup_{a \in \mathcal{A}} \partial B_a^{\text{in}} .$$

So any point on $\bigcup_{a \in \mathcal{A}} \partial B_a$ belongs to ∂B_b^{in} for some $b \in \mathcal{A}$. If $\partial B_a^{\text{out}} \neq \emptyset$, then $\mathcal{M}_a$ is well defined by Equation (2.6), but $\partial B_a^{\text{out}} = \emptyset$ occurs when $f(a)$ is in the interior of B_a. In this case, $f(a)$ is an asymptotically stable steady state, and all points in ∂B_a are in its basin of attraction, so we can define that $\forall x \in \partial B_a$, $\mathcal{M}_a x = f(a)$ and $\mathcal{M}_a f(a) = f(a)$. Then $\{f(a)\}$ has to be added to the domain of $\mathcal{M}_a$.

It is now convenient to introduce the subset of terminal subscripts

$$\mathcal{T} = \{a \in \mathcal{A} \,|\, f(a) \in \mathrm{int}(B_a)\} = \{a \in \mathcal{A} \,|\, \boldsymbol{d}(f(a)) = a\}$$

and one can then define local transition maps in all boxes as

$$\mathcal{M}_a : x \in \mathrm{Dom}(\mathcal{M}_a) \mapsto \begin{cases} f(a) + e^{-\Lambda\tau(x)}((x - f(a)) & \text{if } a \in \mathcal{A} \setminus \mathcal{T} \\ f(a) & \text{if } a \in \mathcal{T} \end{cases}$$

It then follows that the domain $\mathrm{Dom}(\mathcal{M}_a) = \partial B_a^{\mathrm{in}}$ for $a \in \mathcal{A} \setminus \mathcal{T}$ and $\mathrm{Dom}(\mathcal{M}_a) = \partial B_a^{\mathrm{in}} \cup \{f(a)\}$ for $a \in \mathcal{T}$, and globally,

$$\bigcup_{a\in\mathcal{A}} \mathrm{Dom}(\mathcal{M}_a) = \bigcup_{a\in\mathcal{A}} \partial B_a \cup \bigcup_{a\in\mathcal{T}} \{f(a)\}\,.$$

A global mapping may still not properly be defined by the above on all of $\bigcup_{a\in\mathcal{A}} \mathrm{Dom}(\mathcal{M}_a)$. The problem arises where an $x \in \mathrm{Dom}(\mathcal{M}_a)$ maps to the intersection of two or more walls, in which case the choice of local map is not unique, and in some cases the subsequent flow depends on this choice. To avoid this issue, following Farcot [15], we exclude all codimension-2 faces from the analysis, along with all points from which those faces can be reached. On such a domain, a global map can be well defined. Farcot expresses the global map in terms of all the local maps for each box B_a using the indicator function ($\mathbf{1}_C(x) = 1$ for $x \in C$, 0 otherwise) as,

$$\mathcal{M}x = \sum_{a\in\mathcal{A}} \mathbf{1}_{\mathrm{Dom}(\mathcal{M}_a)}(x)\mathcal{M}_a x\,, \tag{2.7}$$

but defines its domain as

$$\mathscr{D}_0 = \bigcup_{a\in\mathcal{A}} \mathrm{Dom}(\mathcal{M}_a) \setminus \bigcup_{k\in\mathbb{N}} \mathcal{M}^{-k}(\mathscr{F}_2)\,, \tag{2.8}$$

where $\mathscr{F}_2$ is the union of all threshold faces of codimension 2 or more, $\mathcal{M}^k$ is the k^{th} iterate of $\mathcal{M}$, and $\mathcal{M}^{-k}(\mathscr{F}_2)$ is the k^{th} pre-image of the set $\mathscr{F}_2$.

2.2.2 Equal decay rates and single thresholds

Now, as Farcot [15] points out, $(\mathscr{D}_0, \mathcal{M})$ is a properly defined one-sided discrete dynamical system, whose orbits are $\{\mathcal{M}^k x\}_{k\in\mathbb{N}}$, for $x \in \mathscr{D}_0$. The iterates of $\mathcal{M}$ are compositions of local maps, as determined by the sequence of walls crossed. The general form of these map compositions are derived by Farcot, but for the purpose of explicit calculations on specific examples, we will need to impose another condition on the networks:

Condition 3. *$\forall\, i$, $\lambda_i = \lambda$ and each variable has only a single threshold value.*

A single threshold in each variable implies that in Equation (2.3), $p_i = 2$ for all i, and all thresholds may be translated to 0, by $y_i = x_i - \theta_i$ for each i.

Under the assumptions of Condition 3, the analysis follows that of Edwards [9], and the local maps are all fractional linear:

$$\mathcal{M}_a y = \frac{\boldsymbol{\beta}_a y}{1 + \psi_a^\top y}\,, \tag{2.9}$$

where

$$\boldsymbol{\beta}_a = I - \frac{f(a)\boldsymbol{e}_j^\top}{f(a)^\top \boldsymbol{e}_j}\,, \quad \psi_a = \frac{-\boldsymbol{e}_j}{f(a)^\top \boldsymbol{e}_j}\,, \tag{2.10}$$

and $\boldsymbol{e}_j$ is again the j^{th} standard basis vector, and j is the index of the exit wall of $\mathcal{M}_a$, starting from x.

Then compositions of these fractional linear maps are also fractional linear, and denoting $\mathcal{M}_{a_k}, \boldsymbol{\beta}_{a_k}$, and ψ_{a_k} more simply as $\mathcal{M}^{(k)}, \boldsymbol{\beta}^{(k)}$, and $\psi^{(k)}$, after m steps the composite map is

$$\mathcal{M}^{(m-1)} \cdots \mathcal{M}^{(0)} y = \frac{\boldsymbol{\beta}^{(m,0)} y}{1 + \psi^{(m,0)^\top} y}\,, \tag{2.11}$$

where

$$\boldsymbol{\beta}^{(m,0)} = \boldsymbol{\beta}^{(m-1)} \ldots \boldsymbol{\beta}^{(1)} \boldsymbol{\beta}^{(0)}\,, \quad \text{and} \quad \psi^{(m,0)} = \psi^{(0)} + \sum_{k=1}^{m-1} \boldsymbol{\beta}^{(k,0)^\top} \psi^{(k)}\,.$$

Thus, for a cycle, where after m steps the trajectory returns to its initial wall (we may say $W^{(m)} = W^{(0)}$, using the same numbering for walls as for the maps), we have

$$\mathcal{M} : W^{(0)} \to W^{(0)}, \quad \mathcal{M} y = \frac{\boldsymbol{\beta} y}{1 + \psi^\top y}\,, \tag{2.12}$$

with $\boldsymbol{\beta} = \boldsymbol{\beta}^{(m,0)}$ and $\psi = \psi^{(m,0)}$.

A cycle map, as in Equation (2.12), is in general not defined on an entire wall, but on a subset of a wall, called a returning cone: $C = \{y \in W^{(0)} \,|\, \mathcal{M}(y) \in W^{(0)}\}$. As detailed in Section 2.4, one way of computing a cycle's returning cone is to follow the starting wall backwards through the cycle using pre-images of $\mathcal{M}$ and intersect these with each wall along the cycle. In practice, this means identifying which region on a wall maps to the correct region in the next wall. Outside this region, a different wall (not the one following the cycle) will be reached at the next or a subsequent step. Thus, whenever an entry wall to a box along the cycle maps to multiple exit walls, an inequality must be satisfied on the entry wall for each alternative exit variable to ensure that such a diversion from the cycle does not occur. These inequalities must

then be mapped back from the relevant entry wall to the starting wall. The inequality to prevent an alternative exit on the k^{th} wall of the cycle is [9]

$$-\frac{\boldsymbol{e_i}^\top}{f_i^{(k)}}\boldsymbol{\beta}^{(k)}y^{(k)} > 0, \tag{2.13}$$

where i represents an alternative exit direction, and the denominator of the map (2.9), which is necessarily positive, has been omitted. Note that there may be more than one such i for a given wall.

The corresponding inequality on the starting wall is

$$-\frac{\boldsymbol{e_i}^\top}{f_i^{(k)}}\boldsymbol{\beta}^{(k)}\boldsymbol{\beta}^{(k-1)}\dots\boldsymbol{\beta}^{(0)}y^{(0)} > 0. \tag{2.14}$$

The region satisfying this inequality on the starting wall gets mapped to the region on the k^{th} wall that satisfies the inequality (2.13). Thus, in order for a trajectory starting on the starting wall to return to the starting wall following the given cycle, it must satisfy the set of inequalities (2.14) for each alternative exit variable around the cycle. This set of inequalities defines the returning cone: $C = \{y \in W^{(0)} \mid Ry \geq 0\}$ where $W^{(0)}$ is the starting boundary and R is a matrix with one row for each alternate exit variable around the cycle, given by

$$R_i = -\frac{\boldsymbol{e_i}^\top}{f_i^{(k)}}\boldsymbol{\beta}^{(k)}\boldsymbol{\beta}^{(k-1)}\dots\boldsymbol{\beta}^{(0)} \tag{2.15}$$

as in the left hand side of (2.14).

2.3 Symbolic Dynamics Approach

As discussed in the previous section, the dynamics of Glass networks can be represented (without loss of information) using a continuous-space, discrete-time dynamical system. This discretization process can be taken a step further to discretize space as well. Glass network dynamics can be qualitatively represented using symbolic dynamical systems and as a result, techniques from symbolic dynamics can be used to draw useful conclusions about the original system. The following is a summary of the construction used by Farcot to analyze the irregularity of Glass network dynamics using symbolic dynamics. For a detailed description of the construction, see Farcot [15].

The partitioning of phase space into boxes along with the transition maps between them allows a natural encoding of the allowable dynamics of the system in terms of a directed graph, called a transition graph and denoted $\mathrm{TG} = (\mathcal{A}, \mathcal{E})$. The TG vertices ($\mathcal{A}$) are box subscripts and the edges ($\mathcal{E}$) correspond to pairs of adjacent boxes that

can be successively crossed by a trajectory. This includes 1-loops, to handle the case of a focal point in the interior of its own box, $f(a) \in \mathrm{int}(B_a)$, and pairs that are adjacent through an $(n-1)$-dimensional wall. By Condition 2, B_a and B_b are adjacent via a single wall if and only if $a - b = \pm \boldsymbol{e}_i$, for $i \in \{1, ..., n\}$. Trajectories traversing a box B_a can only exit through a wall $W_i^{\pm}$ for $i \in I_{\mathrm{out}}(a)$ with the direction given by $\varepsilon_i = \mathrm{sign}(\boldsymbol{d}_i(f(a)) - a_i)$. Thus, the edge set of the TG is

$$\mathcal{E} = \{(a, a) \,|\; a \in \mathcal{T}\} \cup \{(a, a + \varepsilon_i \boldsymbol{e}_i) \,|\, a \in \mathcal{A} \setminus \mathcal{T},\; i \in I_{\mathrm{out}}(a)\} \;.$$

The TG describes transitions between boxes that can occur through faces that have dimension $(n-1)$, *i.e.*, through interiors of walls. Now, attractors of the continuous-space, discrete time system $(\mathscr{D}_0, \mathcal{M})$ have a counterpart in the TG, but the converse does not hold in general since trajectories on the TG may correspond to no trajectories in the original continuous-space system. Some information has been lost in going to the TG dynamics. We will take this point up again in the next Section.

The TG encodes the possible dynamics of $(\mathscr{D}_0, \mathcal{M})$ into a subset of infinite words on the alphabet $\mathcal{A}$. The words are given by infinite paths on the graph and the set of all words is given by

$$\mathscr{J}\,(\mathrm{TG}) = \{\boldsymbol{a} = (a^t)_{t \in \mathbb{N}} \,|\, \forall t \in \mathbb{N},\; (a^t, a^{t+1}) \in \mathcal{E}\} \subset \mathcal{A}^{\mathbb{N}} \,.$$

The space $\mathscr{J}\,(\mathrm{TG})$ can be endowed with a metric to make it a metric space, on which discrete dynamics can be defined by the shift operator $\sigma : \mathscr{J}\,(\mathrm{TG}) \to \mathscr{J}\,(\mathrm{TG})$, where $(\sigma(\boldsymbol{a}))^t = a^{t+1}$. This operator is continuous, for example, in the metric

$$\rho(\boldsymbol{a}, \boldsymbol{b}) = \begin{cases} 0 & \text{if } \boldsymbol{a} = \boldsymbol{b} \,, \\ 2^{-\min\{t | a^t \neq b^t\}} & \text{if } \boldsymbol{a} \neq \boldsymbol{b} \,. \end{cases}$$

The more initial terms of $\boldsymbol{a}$ and $\boldsymbol{b}$ coincide, the smaller the distance between them, according to this metric. Additionally, $\mathscr{J}\,(\mathrm{TG})$ is compact for the metric ρ and is σ-invariant. As a result, $\mathscr{J}\,(\mathrm{TG})$ is a shift space, where $(\mathscr{J}\,(\mathrm{TG}), \sigma)$ constitutes a discrete dynamical system. Since orbits of this system are associated with words defined on the alphabet $\mathcal{A}$, whose elements in turn represent subsets of the state space of the initial dynamical system, the trajectories of $(\mathscr{J}\,(\mathrm{TG}), \sigma)$ represent trajectories or sets of trajectories in $(\mathscr{D}_0, \mathcal{M})$.

Now, $\mathscr{D}_0$ lies in the union of all faces of boxes taken without the boundaries of these faces (as well as points $f(a)$ for $a \in \mathcal{T}$). Any of these open faces is well defined by the two boxes it is a part of, except those on the boundary $\partial\mathcal{U}$, but since this boundary cannot be reached from the rest of $\mathscr{D}_0$, we can ignore it and define $\mathscr{D} = \mathscr{D}_0 \setminus \partial\mathcal{U}$. So now $\forall\, x \in \mathscr{D}$, there is either a unique pair (a, b) such that $x \in \partial B_a^{\mathrm{out}} \cap \partial B_b^{\mathrm{in}}$, or some

$a \in \mathcal{T}$ such that $x \in \partial B_a \cup \{f(a)\}$. We can now define a map $\Phi : \mathscr{D} \to \mathcal{E}$ as

$$\Phi(x) = \begin{cases} (a,b) \text{ if } x \in \partial B_a^{\text{out}} \cap \partial B_b^{\text{in}} \\ (a,a) \text{ if } x \in \partial B_a \cup \{f(a)\}, \text{ for } a \in \mathcal{T} \end{cases}$$

so that Φ maps to edges of the TG instead of vertices (except when $a \in \mathcal{T}$). Note that $\Phi^{-1}(a,b)$ is the open wall between two adjacent boxes B_a and B_b. This then leads us to consider a new shift space obtained from $\mathscr{J}(\text{TG})$ through the 2-block map β_2 defined as

$$(\beta_2(\boldsymbol{a}))^t = \begin{bmatrix} a^t \\ a^{t+1} \end{bmatrix} \in \mathcal{E}$$

where $\mathscr{J}(\text{TG})^{[2]} = \beta_2(\mathscr{J}(\text{TG})) \subset \mathcal{E}^{\mathbb{N}}$ is a shift space. The shift operator on $\mathscr{J}(\text{TG})^{[2]}$ is then denoted as $\sigma_{[2]}$. Since β_2 is a conjugacy [45, p.18] $(\mathscr{J}(\text{TG}), \sigma)$ and $(\mathscr{J}(\text{TG})^{[2]}, \sigma_{[2]})$ are conjugate dynamical systems.

Now to encode the trajectories from $(\mathscr{D}, \mathcal{M})$ there are two steps. First, define the mapping $\xi : \mathscr{D} \to \mathscr{D}^{\mathbb{N}}$ by $\xi(x) = (x, \mathcal{M}x, \mathcal{M}^2x, ...)$. Since $\mathcal{M}$ is continuous on $\mathscr{D}$ it can be proven that ξ is a conjugacy when its range is restricted to $\xi(\mathscr{D})$ (see Farcot [15] for details). The second step is the mapping $\Phi_\infty : \mathscr{D}^{\mathbb{N}} \to \mathscr{J}(\text{TG})^{[2]}$, given by

$$\Phi_\infty\left((x^k)_{k\in\mathbb{N}}\right) = \left(\Phi(x^k)\right)_{k\in\mathbb{N}} .$$

Here Φ_∞ maps sequences in $\mathscr{D}$ to sequences in $\mathcal{E}$. Finally, the map ϕ is defined as

$$\phi = \Phi_\infty \circ \xi : \mathscr{D} \to \mathscr{J}(\text{TG})^{[2]}$$

It is shown by Farcot [15] that the mapping ϕ takes constant values on the domains

$$D_a = \bigcap_{j\in\mathbb{N}} \mathcal{M}^{-j}\left(\Phi^{-1}(a^j, a^{j+1})\right)$$

which are exactly the connected components of $\mathscr{D}$. Hence, ϕ is continuous.

As discussed by Farcot [15], Φ_∞, and thus ϕ, is neither injective nor surjective in general. The non-injectivity of ϕ is an inevitable feature of the system $(\mathscr{D}, \mathcal{M})$, in which the domains D_a associated with admissible itineraries are not single points. The fact that ϕ is not surjective means that some infinite paths on the transition graph do not correspond to any admissible trajectory of the continuous-space system. The TG represents transitions between boxes that are individually feasible, but there is no guarantee that a finite sequence of such transitions is feasible. Thus, $\phi(\mathscr{D}) \subset \mathscr{J}(\text{TG})^{[2]}$ is a proper subset, and exactly the space of admissible trajectories in TG. As detailed by Farcot [15], $\phi(\mathscr{D})$ inherits shift-invariance from $\mathscr{D}$'s $\mathcal{M}$-invariance. Now, in order for the shift space $\phi(\mathscr{D})$ to define a dynamical system when paired with $\sigma_{[2]}$, it must

be compact [45, p.185]. Since $\mathscr{J}(\mathrm{TG})^{[2]}$ is compact and contains $\phi(\mathscr{D})$ it would suffice to show that $\phi(\mathscr{D})$ is closed. However, it turns out that $\phi(\mathscr{D})$ is not closed in general.

As discussed by Lind and Marcus [45, p.180] there are two equivalent characterizations of shift spaces: they can be defined as shift-invariant compact subspaces of the full shift, or as subspaces of all infinite words on the alphabet defined by a set of forbidden blocks. Now since ϕ is constant on each D_a, $\phi(\mathscr{D})$ is the set of words $\epsilon = \beta_2(a)$ such that $D_a \neq \emptyset$. Letting $D_a^i = \bigcap_{k=0}^{i} \mathcal{M}^{-k}\left(\Phi^{-1}(a^k, a^{k+1})\right)$, where $D_a = \bigcap_{i\in\mathbb{N}} D_a^i$ is empty if either one D_a^i is empty or all of them are nonempty but their full intersection is empty. In the latter case, $\beta_2(a)$ is an infinite word that is forbidden in $\phi(\mathscr{D})$ while all its subwords are allowed. Thus, it does not satisfy the first characterization of a shift space. Naturally then the system to consider is $\left(\overline{\phi(\mathscr{D})}, \sigma_{[2]}\right)$. This one is a properly defined symbolic dynamical system since the closure $\overline{\phi(\mathscr{D})}$ is clearly compact. Now, the dynamics of the two symbolic dynamical systems $\left(\overline{\phi(\mathscr{D})}, \sigma_{[2]}\right)$ and $(\mathscr{J}(\mathrm{TG})^{[2]}, \sigma_{[2]})$ may be compared, by means of topological entropy. Topological entropy is a nonnegative number that is a measure of the complexity of a dynamical system. Nonzero values of topological entropy are often taken as an indicator of dynamical chaos.

In symbolic dynamics, topological entropy is conjugacy invariant and can be computed easily for systems described by directed graphs. This makes it a useful tool for comparing symbolic dynamical systems. We first give the classical definition of topological entropy for symbolic dynamical systems.

Definition 2.1. Let X be a shift space. The topological entropy of X is defined by

$$h(X) = \lim_{n\to\infty} \frac{1}{n} \log |\mathcal{B}_n(X)| \tag{2.16}$$

where $|\mathcal{B}_n(X)|$ is the number of blocks of length n for the shift space X, and log is by convention the logarithm with base 2.

By definition $h(X)$ is nonnegative. A dynamical system with positive entropy is chaotic by some definitions (e.g. DC2 [8]). In the case when X is defined by the way of infinite paths on an oriented graph G, let A be the adjacency matrix of G: $A_{i,j} \in \{0, 1\}$, and $A_{i,j} = 1$ if and only if (i, j) is an edge in the graph. Define the irreducible components of A as the equivalence classes for the equivalence relation: $i \sim j$ if $\exists p, q \in \mathbb{N}, (A^p)_{i,j} \neq 0$ and $(A^q)_{j,i} \neq 0$. This corresponds exactly to the strongly connected components in G. Let A_i, $i = 1, ..., k$ be the sub-matrices of A with all indices in the same equivalence class. If there is a single class, A is said to be irreducible. The Perron-Frobenius theorem ensures that any irreducible matrix with nonnegative entries has a dominant positive eigenvalue μ_A, which is simple, and is associated with a nonnegative eigenvector. Following Lind and Marcus [45, p.121], the Perron eigenvalue of A is

$$\mu_A = \max_{i=1,...,k} \mu_{A_i}$$

where each μ_{A_i} is the Perron eigenvalue of each irreducible sub-matrix, A_i, and the entropy is given by $h(X) = \log \mu_A$.

Finally, since $\mathscr{J}(\mathrm{TG})$ and $\mathscr{J}(\mathrm{TG})^{[2]}$ are conjugate and topological entropy is conjugacy invariant it follows that $h(\mathscr{J}(\mathrm{TG})^{[2]}) = h(\mathscr{J}(\mathrm{TG}))$. As $\mathscr{J}(\mathrm{TG})$ is exactly the shift space induced by infinite paths on the TG, one simply defines

$$h_{\mathrm{TG}} = h(\mathscr{J}(\mathrm{TG})^{[2]}) = h(\mathscr{J}(\mathrm{TG}))\,.$$

One can then also define the entropy of the true dynamics ($\phi(\mathscr{D})$) of the network as

$$h_{\phi(\mathscr{D})} = h\left(\overline{\phi(\mathscr{D})}\right)\,.$$

Since we have that $\overline{\phi(\mathscr{D})} \subset \mathscr{J}(\mathrm{TG})^{[2]}$, it follows that $h_{\phi(\mathscr{D})} \leq h_{\mathrm{TG}}$.

2.4 Better Entropy Bounds

In the previous two sections we discussed the construction from [15] that allows for $(\mathscr{D}, \mathcal{M})$ to be mapped into $(\overline{\phi(\mathscr{D})}, \sigma_{[2]})$. The entropy of $(\overline{\phi(\mathscr{D})}, \sigma_{[2]})$ is then shown to be bounded above by the entropy of the TG, which is an indication of the potential irregularity of the system dynamics. However, for many Glass networks the entire set of dynamics allowed by the TG may not be realized in the continuous dynamics after transients, so this upper bound on entropy may be very loose. When there exists a *trapping region* in phase space, *i.e.*, a positively invariant region from which trajectories cannot escape, then the system dynamics may be constrained to a subset of the TG. It may be that all trajectories are eventually confined to one trapping region, but even if there are other attractors, once confined to a trapping region, the entropy of the subsequent dynamics is unaffected by other parts of phase space. Thus, once transients have died out, the long term dynamics can be represented using the subset of the TG that represents only the trapping region, and the corresponding entropy provides a tighter upper bound than the full TG.

Additionally it may be that even within the trapping region some dynamics allowed by the TG subset are also never realized. To account for this, procedures are developed to obtain tighter upper bounds on entropy by eliminating cycles and concatenations of cycles on the TG that are shown to be impossible or transient in computations of returning regions and their images. The key idea is to represent dynamics within the trapping region by more detailed graphs that disallow the forbidden dynamics still allowed by previous graphs. In doing so, a sequence of progressively more detailed graphs can be constructed where each graph captures more closely the actual dynamics allowed by the trapping region. This sequence can then be shown to have entropies converging to the true entropy within the trapping region. For this section, Condition 3

is not necessary and thus will not be imposed.

2.4.1 Trapping regions

In order to improve the upper bound on entropy using information about a trapping region, we must first define trapping regions. To do this, a few definitions will be needed, starting with *cycles*. Based on the conventions of Lind and Marcus [45, p.38],

Definition 2.2. A path $\pi = e_1 \dots e_m$ on a graph G is a finite sequence of edges e_i from G such that $t(e_i) = i(e_{i+1})$ for $1 \leq i \leq m-1$, where $i(e)$ and $t(e)$ here indicate the initial and terminal nodes of an edge. The length of π is $|\pi| = m$, the number of edges it traverses. The path π starts at vertex $i(\pi) = i(e_1)$ and terminates at vertex $t(\pi) = t(e_m)$, and π is a path from $i(\pi)$ to $t(\pi)$. A cycle is a path that starts and terminates at the same vertex. If, in addition, $e_i \neq e_1$ for $i = 2, \dots, m$, we call π a first-return cycle (to edge e_1).

Since each cycle on a network's TG represents a potential sequence of wall transitions in $(\mathscr{D}, \mathcal{M})$, for a given cycle $\pi = e_1 \dots e_m$, the edge e_1 represents a wall from which the path in phase space starts. We will denote this wall as the *starting wall* and e_1 as the *starting edge*. These terms may be used interchangeably if it is clear which context is being referenced.

In order for a trajectory passing through a starting wall to follow a cycle and return back to that wall, the trajectory must be confined to the region in each wall that maps through the rest of the walls in the cycle. To specify the region of phase space occupied by trajectories following the cycle, or at least its intersections with each wall around the cycle, we first define the *returning region* on the starting wall.

Definition 2.3. Let $\pi = e_1 \dots e_m$ be a cycle with starting wall W_1 and walls associated with each edge e_i denoted W_i, $i \in \{1, \dots, m\}$. The returning region of π on wall W_1 is the subset of W_1 that under m applications of the map $\mathcal{M}$ will return back to W_1 following the path π. Letting $R_i(S) = \mathcal{M}^{-1}(S) \cap W_i$, this subset is defined as

$$\mathscr{R}_\pi(W_1) = R_1 \circ \cdots \circ R_m(W_1)\,. \tag{2.17}$$

We can also define the cycle map as follows:

Definition 2.4. The cycle map, $\mathcal{M}_\pi : \mathscr{R}_\pi(W_1) \to W_1$ is defined by

$$\mathcal{M}_\pi(x) = \mathcal{M}^m(x), \quad x \in \mathscr{R}_\pi(W_1),$$

where $\mathcal{M}^m$ is the m^{th} iterate of $\mathcal{M}$, and m is the length of π.

When all decay rates in the Glass network are equal (but not in general), these returning regions are polygonal cones commonly referred to as *returning cones*. The

subset of $\mathscr{D}$ through which trajectories starting in this returning region $\mathscr{R}_\pi(W_1)$ pass is now defined as follows:

Definition 2.5. For a first-return cycle π of length m with starting wall W_1, the subset of the domain $\mathscr{D}$ traversed by trajectories following π is

$$\mathscr{D}_\pi = \mathscr{D} \cap \bigcup_{i=0}^{m-1} \mathcal{M}^i(\mathscr{R}_\pi(W_1)) \tag{2.18}$$

where $\mathcal{M}^0 = id$. We call this the cycle tube associated with cycle π.

Using these definitions we can now define a trapping region for a Glass network.

Definition 2.6. For a given Glass network, and a given starting wall, W_1, a trapping region in wall W_1 is defined as

$$\mathscr{T} = \bigcup_{\pi \in \Pi} \mathscr{R}_\pi(W_1)\,, \tag{2.19}$$

where Π is a set of first-return cycles (possibly infinitely many) on the TG such that every cycle in Π starts at e_1 (the starting edge) and

$$\bigcup_{\pi \in \Pi} \mathcal{M}_\pi(\mathscr{R}_\pi(W_1)) \subseteq \mathscr{T}\,.$$

Its corresponding trapping tube is defined as

$$\mathscr{D}_{TR} = \bigcup_{\pi \in \Pi} \mathscr{D}_\pi\,. \tag{2.20}$$

We note here that transients can often be removed to obtain a smaller trapping region before proceeding with the analysis. In the context of our trapping region and trapping tube, a transient region is any returning region that is non-empty, but after an initial cycle back to the starting wall, is never visited again. This is the case when there are nonempty returning regions that are not mapped into by any $\pi \in \Pi$. If the returning region of a transient cycle is removed from the trapping region, one is left with a smaller trapping region. Transient regions can be defined as follows.

Definition 2.7. For a given Glass network with a trapping region as defined above, if there exists a cycle $\pi_j \in \Pi$ such that

$$\bigcup_{\pi \in \Pi} \mathcal{M}_\pi\left(\mathscr{R}_\pi(W_1)\right) \cap \mathscr{R}_{\pi_j}(W_1) = \emptyset,$$

then π_j is a transient cycle (sequence of box transitions) and its returning region is a transient region on the wall W_1.

We can obtain a trapping region with transients removed for a given Glass network and a given starting wall, W_1, as follows. Given a trapping region for the set of cycles Π as given by (2.19), if there exist a set of transient cycles $\Pi_T \subset \Pi$ then the smaller trapping region is

$$\bigcup_{\pi \in \Pi \setminus \Pi_T} \mathscr{R}_\pi(W_1), \tag{2.21}$$

since

$$\bigcup_{\pi \in \Pi} \mathcal{M}_\pi(\mathscr{R}_\pi(W_1)) \subseteq \bigcup_{\pi \in \Pi \setminus \Pi_T} \mathscr{R}_\pi(W_1).$$

Its trapping tube is then

$$\mathscr{D}_{TR} = \bigcup_{\pi \in \Pi \setminus \Pi_T} \mathscr{D}_\pi. \tag{2.22}$$

Similarly, one could also define transient regions such that the image after a finite number of iterations of the set of cycle maps (rather than just one iteration) does not intersect with $\mathscr{R}_\pi(W_1)$. However, the notation becomes more cumbersome. To simplify notation henceforth, we will write the set of cycles Π forming a trapping region assuming that the transients have already been removed.

We are interested in Glass networks in which such a trapping region occurs. To ensure that we are considering only relevant systems we now impose another condition:

Condition 4. *There exists a wall such that some subset* $\boldsymbol{T} \subseteq \Pi$ *forms a trapping region, where* Π *is the set of all first-return cycles starting at that wall.*

This condition means that there is a starting wall that contains a trapping region, so that at least some trajectories always return to the starting wall, with a graph corresponding to the trapping region that is potentially a proper subgraph of the TG.

We will further assume that we have a minimal trapping region, in the sense that it is not the union of disjoint sets that are themselves trapping regions. Multiple trapping regions correspond to different attractors, each with their own entropy, and we focus here on the entropy of a single attractor.

In order to represent dynamics, we define the subset of the TG that corresponds to the trapping region in the following way:

Definition 2.8. For a given Glass network satisfying Conditions 1, 2, and 4, let $\boldsymbol{T}$ be a set of first-return cycles on the TG that forms a trapping region. For each $\pi \in \boldsymbol{T}$, let G_π be the subgraph of TG representing π. The TG_r is then defined as $\mathrm{TG}_r = (\mathcal{A}_r, \mathcal{E}_r)$ where

$$\mathcal{A}_r = \bigcup_{\pi \in \boldsymbol{T}} \mathcal{V}(G_\pi) \quad \text{and} \quad \mathcal{E}_r = \bigcup_{\pi \in \boldsymbol{T}} \mathcal{E}(G_\pi),$$

and $\mathcal{V}(G_\pi)$ and $\mathcal{E}(G_\pi)$ are the respective vertex and edge sets for the graph G_π.

The TG_r encodes the possible dynamics of $(\mathscr{D}_{\mathrm{TR}}, \mathcal{M})$ into a subset of infinite words on the alphabet $\mathcal{A}_r$. The words are similarly given by infinite paths on TG_r and the set of all words is given by

$$\mathscr{J}(\mathrm{TG}_r) = \{\boldsymbol{a} = (a^t)_{t\in\mathbb{N}} \,|\, \forall t \in \mathbb{N},\ (a^t, a^{t+1}) \in \mathcal{E}_r\} \subset \mathcal{A}^{\mathbb{N}}\,.$$

It should be clear from this definition that $\mathcal{E}_r \subseteq \mathcal{E}$ and $\mathscr{J}(\mathrm{TG}_r) \subseteq \mathscr{J}(\mathrm{TG})$. In order to compare the continuous dynamics to the dynamics of $\mathscr{J}(\mathrm{TG}_r)$ we must apply the 2-block map to obtain $\mathscr{J}(\mathrm{TG}_r)^{[2]} = \beta_2(\mathscr{J}(\mathrm{TG}_r)) \subseteq \mathcal{E}_r^{\mathbb{N}} \subseteq \mathcal{E}^{\mathbb{N}}$. It follows that $\mathscr{J}(\mathrm{TG}_r)^{[2]} \subseteq \mathscr{J}(\mathrm{TG})^{[2]}$. Now we must encode the trajectories of $(\mathscr{D}_{\mathrm{TR}}, \mathcal{M})$ in order to compare the dynamics on the trapping region to $\mathscr{J}(\mathrm{TG}_r)^{[2]}$. The space of admissible trajectories in TG_r is then exactly $\phi(\mathscr{D}_{\mathrm{TR}})$. We encounter the same problem with $\mathscr{D}_{\mathrm{TR}}$ as we did with $\mathscr{D}$, namely, that $\phi(\mathscr{D}_{\mathrm{TR}})$ is not compact. So in order to properly define a symbolic dynamical system we must consider the closure, $\overline{\phi(\mathscr{D}_{\mathrm{TR}})}$. It follows that $\overline{\phi(\mathscr{D}_{\mathrm{TR}})} \subseteq \mathscr{J}(\mathrm{TG}_r)^{[2]} \subseteq \mathscr{J}(\mathrm{TG})^{[2]}$. Finally, defining the entropies $h_{\phi(\mathscr{D}_{\mathrm{TR}})} = h(\overline{\phi(\mathscr{D}_{\mathrm{TR}})})$ and $h_{\mathrm{TG}_r} = h\left(\mathscr{J}(\mathrm{TG}_r)^{[2]}\right) = h\left(\mathscr{J}(\mathrm{TG}_r)\right)$ it follows that $h_{\phi(\mathscr{D}_{\mathrm{TR}})} \leq h_{\mathrm{TG}_r} \leq h_{\mathrm{TG}}$. Thus, the entropy of the TG_r is potentially a better upper bound on the entropy of the true dynamics than is the original TG.

2.4.2 Separated cycle representation of TG_r

The TG_r allows for a more accurate representation of the long-term dynamics of a Glass network. However, it may be that it still misses finer details within the trapping region. For example, if multiple cycles on the TG_r share nodes and edges in common, these can potentially allow additional cycles that have empty returning regions. As a result, while the TG_r does give an improvement on entropy estimation, it still potentially allows for too much. In order to improve further we need to consider graph representations that separate the allowed cycles from each other in such a way that the forbidden cycles are no longer possibilities. This will in turn produce a more accurate entropy estimate.

The idea is to produce a new graph in which all the cycles that contribute to the trapping region are separated from each other, and cycles that exist in TG_r but are not part of the trapping region (because they have empty returning regions) are excluded. In order to do this, however, we will need to impose another condition on the networks that we can consider.

Condition 5. *There exists a finite M such that $|\pi| \leq M$ for all $\pi \in \boldsymbol{T}$ where $\boldsymbol{T}$ is a set of first-return cycles forming a trapping region.*

This condition avoids the situation where there are an infinite number of distinct first-return cycles whose returning regions form a trapping region and an infinite partition of the starting wall. In our analysis, an infinite number of first-return cycles will lead to infinite alphabets in the derived symbolic dynamical systems, and we prefer not

to deal with such difficulties. In most of the examples we have looked at, this situation does not arise, but an example of such a system was presented by Gedeon [21].

Under the assumption of Condition 5, the graph that we propose to use for our new entropy upper bound can be defined as follows:

Definition 2.9. For a given Glass network satisfying Conditions 1, 2, 4, and 5, let $\boldsymbol{T}$ be the set of first-return cycles that forms a trapping region. The $\mathrm{TG}_r(1)$ is then defined as

$$\mathrm{TG}_r(1) = \left(\bigcup_{\pi \in \boldsymbol{T}} \mathcal{A}_\pi,\ \mathcal{E}_{\mathrm{cross}} \cup \bigcup_{\pi \in \boldsymbol{T}} \mathcal{E}_\pi \right), \tag{2.23}$$

where $\mathcal{A}_\pi = \{a_\pi \,|\, a \in \mathcal{V}(G_\pi)\}$, $\mathcal{E}_\pi = \{(a_\pi, b_\pi) \,|\, (a,b) \in \mathcal{E}(G_\pi)\}$, and

$$\mathcal{E}_{\mathrm{cross}} = \bigcup_{\substack{\pi,\tau \in \boldsymbol{T} \\ \pi \neq \tau}} \{(i(e_1)_\pi, t(e_1)_\tau)\},$$

where the starting edge, e_1, is the same for all $\pi \in \boldsymbol{T}$. The subscripts π and τ above now distinguish nodes and edges that belong to different cycles.

Note that under Condition 5, $\mathcal{A}_\pi$, $\mathcal{E}_\pi$, and $\mathcal{E}_{\mathrm{cross}}$ are all finite. From the definition, it should be clear that the $\mathrm{TG}_r(1)$ representation does indeed include each cycle contributing to the trapping region and does separate them from each other, while allowing any cycle to follow any other by means of the cross-cycle edges represented by the set $\mathcal{E}_{\mathrm{cross}}$.

To show that this new representation does improve on the entropy bound achieved by the TG_r, we will need to define a sliding block code from $\mathscr{J}(\mathrm{TG}_r(1))$ to $\mathscr{J}(\mathrm{TG}_r)$ know as an embedding [45, p.18]. For two shift spaces X and Y, an embedding is a sliding block code from X to Y such that it is one-to-one. Embeddings have the convenient property that if X embeds into Y, then $h(X) \leq h(Y)$. Thus, if we can define an embedding from $\mathscr{J}(\mathrm{TG}_r(1))$ to $\mathscr{J}(\mathrm{TG}_r)$, we can prove that the $\mathrm{TG}_r(1)$ produces a potentially lower entropy value than the TG_r.

Up to this point we have exclusively dealt with one-sided shift spaces. The notions of embeddings that we need are defined for two-sided shift spaces by Lind and Marcus [45] and thus cannot be directly applied to our situation. However, we can use the natural extension of a one-sided shift space, defined as follows [18]:

Definition 2.10. For a given one-sided shift space X, the natural extension $\widehat{X}$ is the space of bi-infinite sequences such that $x \in \widehat{X}$ if and only if every sub-word of x is an element in the language of X.

The TG_r has the convenient property that for any two nodes, there exists a path connecting them. As a result, the TG_r language, $\mathcal{L}(\mathscr{J}(\mathrm{TG}_r))$, is the same as that of

its natural extension $\widehat{\mathscr{J}(\mathrm{TG}_r)}$, *i.e.*, $\mathcal{L}(\mathscr{J}(\mathrm{TG}_r)) = \mathcal{L}(\widehat{\mathscr{J}(\mathrm{TG}_r)})$, and therefore

$$h(\mathscr{J}(\mathrm{TG}_r)) = h(\widehat{\mathscr{J}(\mathrm{TG}_r)}).$$

For the purposes of our entropy upper bounds, we can use the natural extension to get the entropy relations we need.

Now we can define an embedding, as discussed above, by taking each subscripted symbol (node) and removing the subscript to recover a symbol in $\mathscr{J}(\mathrm{TG}_r)$, as follows:

Definition 2.11. Let $\boldsymbol{T}$ be the set of first-return cycles making up the trapping region. For any $x \in \widehat{\mathscr{J}(\mathrm{TG}_r(1))}$ there exists a $y \in \widehat{\mathscr{J}(\mathrm{TG}_r)}$ such that $x_i = (y_i)_{\pi_i}$ (where $x_i = (x)_i$), for some $\pi_i \in \boldsymbol{T}$. Define $\psi_1 : \widehat{\mathscr{J}(\mathrm{TG}_r(1))} \to \widehat{\mathscr{J}(\mathrm{TG}_r)}$ as a sliding block code such that $(\psi_1(x))_i = (y_i)$.

In simple terms, the map ψ_1 takes the subscripted symbols from $\mathscr{J}(\mathrm{TG}_r(1))$ and removes their subscripts. Since the $\mathrm{TG}_r(1)$ is more restrictive than the TG_r, every point from $\mathscr{J}(\mathrm{TG}_r(1))$ has a corresponding point in $\mathscr{J}(\mathrm{TG}_r)$. This map is clearly one-to-one and hence an embedding. Thus, $h(\mathscr{J}(\widehat{\mathrm{TG}_r(1)})) \leq h(\widehat{\mathscr{J}(\mathrm{TG}_r)})$. Finally since the $\mathrm{TG}_r(1)$ satisfies the same property as the TG_r that all nodes can be connected by some path, it follows that

$$h(\mathscr{J}(\mathrm{TG}_r(1))) = h(\mathscr{J}(\widehat{\mathrm{TG}_r(1)})) \leq h(\widehat{\mathscr{J}(\mathrm{TG}_r)}) = h(\mathscr{J}(\mathrm{TG}_r).$$

It is clear that the entropy of $\mathrm{TG}_r(1)$ is less than or equal to the entropy of TG_r. However, it remains to be shown that we can use it as an upper bound on entropy for the true dynamics. The map ϕ is defined over the original alphabet and does not distinguish between different regions on walls associated with different cycles. However, we can define a new map consistent with $\mathrm{TG}_r(1)$.

First, we will convert each point in $\mathscr{D}_{TR}$ into a symbol as was done with Φ. Letting $\boldsymbol{T}$ denote the set of first-return cycles that makes up the trapping region, we define map Φ_1 as

$$\Phi_1(x) = (a_\pi, b_\tau) \quad \text{if} \quad x \in (\partial B_a^{\text{out}} \cap \partial B_b^{\text{in}}) \cap \mathscr{D}_\tau,\ \mathcal{M}^{-1}(x) \in \partial B_a^{\text{in}} \cap \mathscr{D}_\pi \tag{2.24}$$

where $\pi, \tau \in \boldsymbol{T}$. Under this definition, $\Phi_1 : \mathscr{D}_{\text{TR}} \to \mathcal{E}_{\text{cross}} \cup \bigcup_{\pi \in \boldsymbol{T}} \mathcal{E}_\pi$ takes a point on a wall and identifies it with the two boxes it separates as well as indicating which cycle it came from and which cycle it is destined to follow. Again analogous to the case with Φ, we define $\Phi_{1_\infty} : \mathscr{D}_{\text{TR}}^{\mathbb{N}} \to \mathscr{J}^{[2]}(\mathrm{TG}_r(1))$ by

$$\Phi_{1_\infty}\left((x^k)_{k\in\mathbb{N}}\right) = \left(\Phi_1(x^k)\right)_{k\in\mathbb{N}}. \tag{2.25}$$

Finally, composing Φ_{1_∞} with ξ gives the map

$$\phi_1 = \Phi_{1_\infty} \circ \xi : \mathscr{D}_{\mathrm{TR}} \to \mathscr{J}^{[2]}(\mathrm{TG}_r(1))\,. \tag{2.26}$$

Under our conditions and on our new graph, ϕ_1 works exactly as ϕ did, except that ϕ_1 differentiates between cycles; $\phi_1(\mathscr{D}_{\mathrm{TR}}) \subseteq \mathscr{J}^{[2]}(\mathrm{TG}_r(1))$. Again we consider the closure to ensure that the space is indeed a shift space. So $\overline{\phi_1(\mathscr{D}_{\mathrm{TR}})} \subseteq \mathscr{J}^{[2]}(\mathrm{TG}_r(1))$ is the shift space of all possible trajectories, now encoded in the alphabet of $\mathrm{TG}_r(1)$. If we define $h_{\phi_1(\mathscr{D}_{\mathrm{TR}})} = h(\overline{\phi_1(\mathscr{D}_{\mathrm{TR}})})$ and $h_{\mathrm{TG}_r}(1) = h(\mathscr{J}^{[2]}(\mathrm{TG}_r(1)))$, and since $\mathscr{J}^{[2]}(\mathrm{TG}_r(1))$ embeds into $\mathscr{J}^{[2]}(\mathrm{TG}_r)$, it follows that

$$h_{\phi_1(\mathscr{D}_{\mathrm{TR}})} \leq h_{\mathrm{TG}_r(1)} \leq h_{\mathrm{TG}_r} \leq h_{\mathrm{TG}}$$

where $h_{\mathrm{TG}_r(1)}$ is the entropy of the separated cycle graph representation TG (*i.e.*, $TG_r(1)$) that only allows for the cycles contributing to the trapping region. This gives an even better upper bound on entropy.

2.4.3 Arbitrary levels of refinement

In the previous subsection, we showed how to remove all cycles not included in $\boldsymbol{T}$ using the separated cycle representation of the TG_r, the $\mathrm{TG}_r(1)$. However, there may also be concatenations of cycles that are forbidden. It is not uncommon in Glass networks that there are cycles π and τ such that $M_\pi(\mathscr{D}_\pi) \cap \mathscr{D}_\tau = \emptyset$ and hence, cycle $\pi\tau$ is forbidden from ever occurring in the true dynamics. It is easy to then modify the $\mathrm{TG}_r(1)$ to reflect this, by simply removing the cross edge between the nodes $i(e_1)_\pi$ and $t(e_1)_\tau$.

Of course, longer concatenations of cycles may also be forbidden, in general. To deal with this, we will need to further generalize the separated cycle representation of the TG and that of a trapping region, beyond returning once to the starting wall as was initially considered. This will allow us to determine concatenations of first-return cycles of length $k \in \mathbb{N}$ that are forbidden. A general way to identify forbidden concatenations of cycles is to calculate their returning regions. Empty returning regions correspond to forbidden cycles. Returning regions for concatenations of cycles, like $\pi\pi$, $\pi\tau$, $\tau\pi$, and $\tau\tau$, can be calculated in the same way as was done for cycles π and τ. We define the tube of a concatenation of first-return cycles as follows:

Definition 2.12. For a concatenation of first-return cycles $\pi_1 \dots \pi_k$, the concatenation tube is given by

$$\mathscr{D}_{\pi_1 \dots \pi_k} = \mathscr{D} \cap \bigcup_{i=0}^{l(\pi_1)-1} \mathcal{M}^i(\mathscr{R}_{\pi_1 \dots \pi_k}(W_1)) \tag{2.27}$$

where $l(\pi_1)$ is the length of cycle π_1, the first cycle in the concatenation.

Under this definition, the returning region of a concatenation of first-return cycles starting with cycle π represents a subset of the returning region for cycle π. The set of all possible concatenations of a fixed number of cycles, k, represents a partitioning of the original trapping region as defined previously. Using this, we can redefine the trapping region $\mathscr{D}_{TR}$ using concatenation tubes in such a way that we can distinguish between regions associated with each concatenation. Thus, it follows that for concatenations of length k

$$\mathscr{D}_{TR} = \bigcup_{\pi_1,\ldots,\pi_k \in \boldsymbol{T}} \mathscr{D}_{\pi_1 \ldots \pi_k} \tag{2.28}$$

It is true that $\mathscr{D}_{TR}$ is partitioned by all the concatenation tubes of all the first-return cycles in $\boldsymbol{T}$. However, it may be that some of the concatenations are transient within $\mathscr{D}_{TR}$. These transient concatenation tubes need to be removed from the domain before we continue. This will result in a new trapping region that is a subset of the original. The new domain is simple enough to define.

Definition 2.13. Let $\boldsymbol{T}$ be the set of first-return cycles that forms a trapping region and let Π_k be the set of length k concatenations of cycles from $\boldsymbol{T}$ that are transient under Definition 4.6. The new domain $\mathscr{D}_{TR}(k)$ is defined as

$$\mathscr{D}_{TR}(k) = \left(\bigcup_{\pi_1,\ldots,\pi_k \in \boldsymbol{T}} \mathscr{D}_{\pi_1 \ldots \pi_k} \right) \setminus \bigcup_{\tau \in \Pi_k} \mathscr{D}_{\tau}, \tag{2.29}$$

where $\mathscr{D}_{\tau}$ is the concatenation tube for the concatenation of first-return cycles, τ.

We can now define a separated representation of TG_r that separates cycle concatenations of length k and that is consistent with the underlying dynamics. As before, in order for this to give us the correct entropy relations, we will need to consider the natural extension where appropriate. The graph representation that forbids certain cycle concatenations of length k is defined as follows:

Definition 2.14. For a given Glass network satisfying Conditions 1, 2, 4, and 5, let $\boldsymbol{T}$ be the set of first-return cycles that together forms a trapping region and let Π_k be the set of length k concatenations from $\boldsymbol{T}$ that are transient under Definition 4.6. The $\mathrm{TG}_r(k)$ is then defined as

$$\mathrm{TG}_r(k) = \left(\bigcup_{\substack{\pi_1,\ldots,\pi_k \in \boldsymbol{T} \\ \pi_1 \ldots \pi_k \notin \Pi_k \\ \mathscr{D}_{\pi_1 \ldots \pi_k} \neq \emptyset}} \mathcal{A}_{\pi_1 \ldots \pi_k} \,,\; \mathcal{E}_{\mathrm{cross}} \cup \bigcup_{\substack{\pi_1,\ldots,\pi_k \in \boldsymbol{T} \\ \pi_1 \ldots \pi_k \notin \Pi_k \\ \mathscr{D}_{\pi_1 \ldots \pi_k} \neq \emptyset}} \mathcal{E}_{\pi_1 \ldots \pi_k} \right), \tag{2.30}$$

where $\mathcal{A}_{\pi_1 \ldots \pi_k} = \{ a_{\pi_1 \ldots \pi_k} \,|\, a \in \mathcal{V}(G_{\pi_1}) \}$,

$$\mathcal{E}_{\pi_1\dots\pi_k} = \{(a_{\pi_1\dots\pi_k}, b_{\pi_1\dots\pi_k}) \,|\, (a,b) \in \mathcal{E}(G_{\pi_1}) \setminus \{e_1\}\},$$

and

$$\mathcal{E}_{\text{cross}} = \bigcup_{\substack{\pi_1,\dots,\pi_k \in \boldsymbol{T} \\ \tau_1,\dots,\tau_k \in \boldsymbol{T} \\ \pi_1\dots\pi_k \notin \Pi_k \\ \tau_1\dots\tau_k \notin \Pi_k \\ \mathscr{D}_{\pi_1\dots\pi_k} \neq \emptyset \\ \mathscr{D}_{\tau_1\dots\tau_k} \neq \emptyset}} \{(i(e_1)_{\pi_1\dots\pi_k}, t(e_1)_{\tau_1\dots\tau_k}) \quad | \quad \pi_2\dots\pi_k = \tau_1\dots\tau_{k-1}\}.$$

Notice that for a given concatenation of first-return cycles of length k, the first $k-1$ letters must match the last $k-1$ letters of the previous cycle. For example, the sequence $\pi\tau\tau\pi$ can only be followed by cycles whose subscript starts with $\tau\tau\pi$. This aligns exactly with iterates within $\mathscr{D}_{TR}(k)$.

In this way, the graph allows for individual cycle traversals as all the other TG representations have, but disallows transient concatenations as well as concatenations with empty returning regions. In effect, this representation allows for exclusion of cycle concatenations of arbitrary length. To demonstrate that the $\text{TG}_r(k)$ does indeed produce an improved entropy bound in the way that we desire will require two more mappings. Another embedding, ψ_k, similar to ψ_1, and another encoder map, ϕ_k, similar to ϕ_1. We will begin with the embedding.

Definition 2.15. Let $\boldsymbol{T}$ be the set of first-return cycles making up the trapping region. For any $x \in \mathscr{J}(\widehat{\text{TG}_r(k)})$ there exists a $y \in \mathscr{J}(\widehat{\text{TG}_r(k-1)})$ such that $x_i = (y_i)_{\pi_i}$ (where $x_i = (x)_i$), for some $\pi_i \in \boldsymbol{T}$. Define $\psi_k : \mathscr{J}(\widehat{\text{TG}_r(k)}) \to \mathscr{J}(\widehat{\text{TG}_r(k-1)})$ as a sliding block code such that $(\psi_k(x))_i = (y_i)$.

In simple terms, the map ψ_k takes the subscripted symbols from $\mathscr{J}(\text{TG}_r(k))$ and removes their k^{th} subscripts. Since the $\text{TG}_r(k)$ is more restrictive than the $\text{TG}_r(k-1)$, every point from $\mathscr{J}(\text{TG}_r(k))$ has a corresponding point in $\mathscr{J}(\text{TG}_r(k-1))$. Note that under this definition where $k=1$, we retrieve ψ_1 as defined previously. This map is also one-to-one and hence an embedding. Thus,

$$h(\mathscr{J}(\widehat{\text{TG}_r(k)})) \leq h(\mathscr{J}(\widehat{\text{TG}_r(k-1)})).$$

Finally since each $\text{TG}_r(k)$ satisfies the same property as the TG_r that all nodes can be connected by some path, it follows that

$$h_{\text{TG}_r(k)} \leq \dots h_{\text{TG}_r(2)} \leq h_{\text{TG}_r(1)} \leq h_{\text{TG}_r} \leq h_{\text{TG}}.$$

At each inequality we omit mention of the natural extension. However, as before, the inequality relations come from the fact that the entropy of the TG one-sided systems and their natural extensions are the same and that in the case of the natural extensions, the embeddings induces the inequality relation.

In order to encode trajectories in a way that is consistent with the $\mathrm{TG}_r(k)$ to show that this sequence does bound the true dynamics, we need to generalize the maps Φ_1 and ϕ_1 to any cycle concatenations of length k.

To generalize to the case of sequences of length k is straightforward. We define new maps: Φ_k, Φ_{k_∞}, and ϕ_k like their $k = 1$ counterparts:

$$\Phi_k(x) = (a_{\pi_1 \ldots \pi_k}, b_{\tau_1 \ldots \tau_k}) \quad \text{if} \quad x \in (\partial B_a^{\text{out}} \cap \partial B_b^{\text{in}}) \cap \mathscr{D}_{\tau_1 \ldots \tau_k},\, \mathcal{M}^{-1}(x) \in \partial B_a^{\text{in}} \cap \mathscr{D}_{\pi_1 \ldots \pi_k} \tag{2.31}$$

where $\pi_1, \ldots, \pi_k, \tau_1, \ldots, \tau_k \in \boldsymbol{T}$. The arbitrary combinations of $\pi_1, \ldots, \pi_k, \tau_1, \ldots, \tau_k$ indicate all possible cycle combinations of length k, even if some may not be realized.

$$\Phi_{k_\infty}\left((x^l)_{l\in\mathbb{N}}\right) = \left(\Phi_k(x^l)\right)_{l\in\mathbb{N}}, \tag{2.32}$$

$$\phi_k = \Phi_{k_\infty} \circ \xi : \mathscr{D}_{\mathrm{TR}}(k) \to \mathscr{J}^{[2]}(\mathrm{TG}_r(k)). \tag{2.33}$$

Note that in Definition 4.12 the set Π_k is finite. This means that in $\phi_k(\mathscr{D}_{TR})$ there are only a finite number of transient words associated with the returning regions of elements from Π_k. Since a finite number of transient words in a shift space will not contribute to the entropy, removing from $\mathscr{D}_{TR}$ the returning regions associated with elements of Π_k will not change entropy. Thus for all k, $h_{\phi_k(\mathscr{D}_{\mathrm{TR}})} = h_{\phi_k(\mathscr{D}_{\mathrm{TR}}(k))}$.

Finally, we obtain the string of inequalities

Proposition 2.16.

$$h_{\phi_k(\mathscr{D}_{TR})} = h_{\phi_k(\mathscr{D}_{TR}(k))} \leq h_{TG_r(k)} \leq \ldots h_{TG_r(2)} \leq h_{TG_r(1)} \leq h_{TG_r} \leq h_{TG}. \tag{2.34}$$

This tells us that we can refine the TG to remove dynamics associated with a forbidden cycle concatenation (one with an empty returning cone) of any length we wish, and we can still use its entropy as an upper bound on the entropy of the true dynamics.

Remark 1. *Since the adjacency matrices are sparse, computation of the eigenvalues is reliable for large k. This gives us a practical method of estimating the entropy of a network attractor to any arbitrary level of refinement, in principle.*

Remark 2. *Above we prove that the cycle-separated graph, $TG_r(k)$, has entropy less than or equal to that of the original TG_r by means of factor codes. An alternative way to prove these inequalities is by means of sequences of state-splittings that in themselves preserve entropy and achieve separation of cycles, followed by pruning of unrealizable cycles and concatenations of cycles. This approach is illustrated in the Appendix.*

2.4.4 The limit of the refinement process

The following theorem states that in the limit as $k \to \infty$, the entropy of the k^{th} refinement will converge to the entropy of the true dynamics. For the proof, we will need the following proposition proven by Lind and Marcus [45, p.123]:

Proposition 2.17. *Let $X_1 \supseteq X_2 \supseteq X_3 \ldots$ be shift spaces whose intersection is X. Then $h(X_k) \to h(X)$ as $k \to \infty$.*

Theorem 2.18. *For a given Glass network, $\lim_{k\to\infty} h_{TG_r(k)} = h_{\phi(\mathscr{D}_{TR})}$.*

Proof. Let $\boldsymbol{T}$ be the set of first-return cycles that forms a trapping region for the network. Consider the sequence of shifts of finite type $X_{\mathcal{F}_k}$, that are all subsets of $\mathscr{J}^{[2]}(\mathrm{TG}_r)$, with sets of forbidden blocks $\mathcal{F}_k$, given by

$$\mathcal{F}_k = \{\pi_1 \ldots \pi_k \mid \mathscr{R}_{\pi_1 \ldots \pi_k}(W_1) = \emptyset \text{ or } \pi_1 \ldots \pi_k \text{ is transient}\} \cup \mathcal{F}$$

where $\pi_1, \ldots, \pi_k \in \boldsymbol{T}$ and $\mathcal{F}$ is the set of forbidden blocks for $\mathscr{J}^{[2]}(\mathrm{TG}_r)$. From the definition of $X_{\mathcal{F}_k}$, it is clear that

$$\bigcap_{k=1}^{\infty} X_{\mathcal{F}_k} = \overline{\phi(\mathscr{D}_{TR})}$$

and that $\mathscr{J}^{[2]}(\mathrm{TG}) \supseteq \mathscr{J}^{[2]}(\mathrm{TG}_r) \supseteq X_{\mathcal{F}_1} \supseteq X_{\mathcal{F}_2} \supseteq X_{\mathcal{F}_3} \ldots$ So by Proposition 2.17 it follows that

$$\lim_{k\to\infty} h(X_{\mathcal{F}_k}) = h(\overline{\phi(\mathscr{D}_{TR})}) = h_{\phi(\mathscr{D}_{\mathrm{TR}})}. \tag{2.35}$$

Now, for the given Glass network consider its $\mathrm{TG}_r(k)$ and the mappings

$$\zeta_k : \mathscr{J}^{[2]}(\mathrm{TG}_r(k)) \to \mathscr{J}^{[2]}(\mathrm{TG})$$

where ζ_k is just the mapping that removes all subscripts from symbols, effectively encoding trajectories from any separated alphabet into the original alphabet. From the definition of ζ_k, it is clear that it is a sliding block code and that it is finite-to-one. Additionally, as detailed by Lind and Marcus [45, p.276], finite-to-one codes on any shift space preserve entropy. So, it follows that

$$h(\mathscr{J}^{[2]}(\mathrm{TG}_r(k))) = h(\zeta_k(\mathscr{J}^{[2]}(\mathrm{TG}_r(k)))).$$

Furthermore, it follows that $\zeta_k(\mathscr{J}^{[2]}(\mathrm{TG}_r(k))) = X_{\mathcal{F}_k}$, and hence

$$h(\mathscr{J}^{[2]}(\mathrm{TG}_r(k))) = h(\zeta_k(\mathscr{J}^{[2]}(\mathrm{TG}_r(k)))) = h(X_{\mathcal{F}_k}).$$

So, we have that $h_{\mathrm{TG}_r(k)} = h(X_{\mathcal{F}_k})$ for all $k \in \mathbb{N}$. Finally, by Equation (2.35) it follows

that $\lim_{k\to\infty} h_{\mathrm{TG}_r(k)} = h(\overline{\phi(\mathscr{D}_{TR})})$. □

Since it is easy to compute the entropy of $\mathrm{TG}_r(k)$ for any k, Theorem 2.18 allows us to find an upper bound arbitrarily close to the true entropy. However, it should be emphasized that we have no way to determine a level of refinement, k, guaranteeing that our estimate is within a specified ε of the true entropy.

2.5 An Example

We now consider an example Glass network to demonstrate the efficacy of these graph refinements. This example will fall into the class of networks that satisfies Condition 3. Note that although the theory of the previous section applies in general, computation of returning regions is only practicable when Condition 3 holds.

Consider the following Glass network:

$$\begin{aligned}
\dot{y}_1 &= -y_1 + 2\left(\bar{Y}_3 Y_4 + Y_2 Y_3\right) - 1 \\
\dot{y}_2 &= -y_2 + 2\left(Y_1 \bar{Y}_3 Y_4 + \bar{Y}_1 Y_3 Y_4 + \bar{Y}_1 \bar{Y}_3 \bar{Y}_4\right) - 1 \\
\dot{y}_3 &= -y_3 + 2\left(\bar{Y}_1 Y_2 + Y_1 Y_4\right) - 1 \\
\dot{y}_4 &= -y_4 + 2\left(Y_2 \bar{Y}_3 + \bar{Y}_1 Y_3\right) - 1
\end{aligned} \tag{2.36}$$

where $Y_i = 0$ if $y_i < 0$, and 1 if $y_i > 0$. $\bar{Y}_i = 1 - Y_i$. This network is simple in the sense that all its degradation rates are 1, all the focal point coordinates are located at ± 1, and each variable has only a single threshold. Its TG can be represented by the 4-dimensional hypercube in Figure 2.1 (see [10]). Projections of example trajectories are shown in Figure 2.2. From the adjacency matrix of the TG, we calculate the approximate entropy (the logarithm of the Perron eigenvalue) to be $h_{\mathrm{TG}} \approx 0.873$.

Now, using the wall between boxes 1111 and 1110 as a starting wall (which we may denote $+++0$ to indicate the sign of each y_i), among the many cycles on the TG that return to this wall are two, denoted A and B, that form a trapping region [10].

$$\begin{aligned}
A &: 1110 \to 1010 \to 0010 \to 0000 \to 0100 \to 0110 \to 0111 \to 1111, \\
B &: 1110 \to 1010 \to 0010 \to 0011 \to 0001 \to 0000 \to 0100 \to 0101 \to 0111 \to 1111.
\end{aligned}$$

Figure 2.3 shows the TG with cycles A and B outlined in red and blue respectively.

As discussed above, to show that the returning cones for these two cycles form a trapping region, one calculates the returning cones for each cycle and shows that their images under their respective cycle maps lie in the union of their returning cones. Thus, any trajectory that follows cycle A or B once will necessarily follow one or the other at each iteration for the rest of time. For cycle A in our example network, from the starting wall there is one alternative exit variable, $i = 3$. On the next wall, $+0+-$

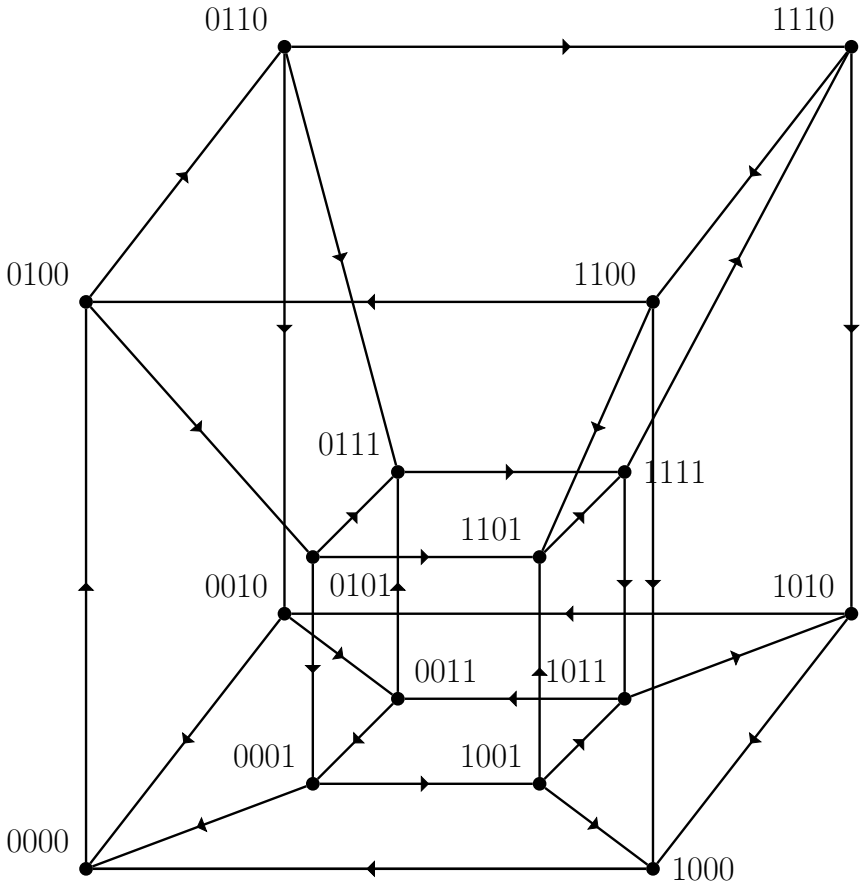


Figure 2.1: TG for example Glass network in Equation (2.36).

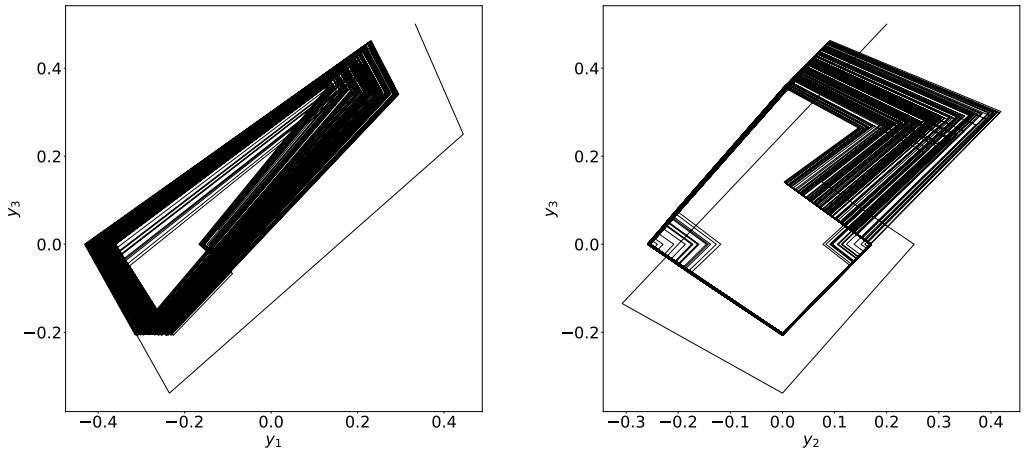


Figure 2.2: Projections of a phase portrait for example Glass network in Equation (2.36).

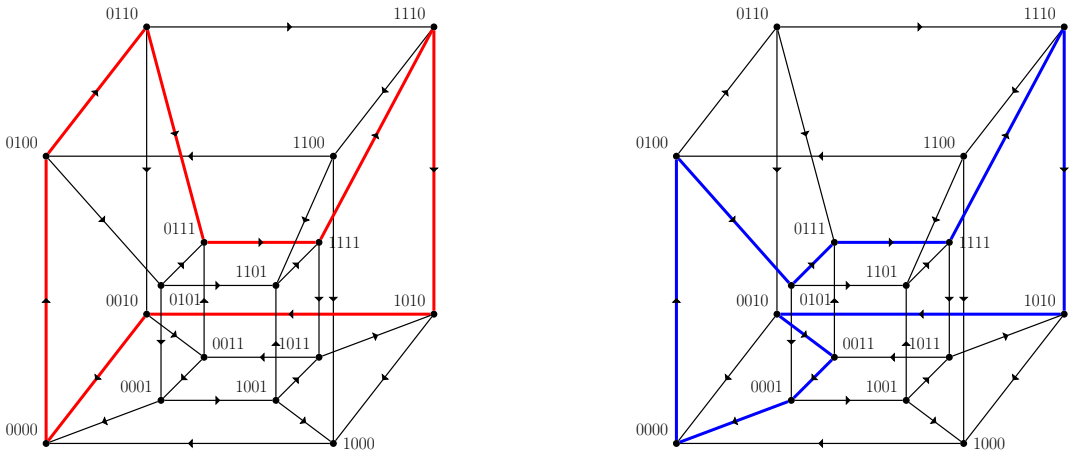


Figure 2.3: TG for example Glass network in Equation (2.36) with cycle A outlined in Red (a) and cycle B outlined in Blue (b).

(between boxes 1110 and 1010), $i = 3$ is again the only alternative exit variable. On the next wall, $0 - + -$ (between boxes 1010 and 0010) the only alternative exit variable is $i = 4$. Then, on $- - 0 -$ (between boxes 0010 and 0000), there are no alternative exit variables so this wall does not contribute a row to the matrix R. On the wall $-0 - -$ (between boxes 0000 and 0100), there is one alternative exit variable, $i = 4$. On $- + 0 -$ (between boxes 0100 and 0110), there are two alternative exit variables, $i = 1$ and $i = 2$. On $- + +0$ (between boxes 0110 and 0111), there are no alternative exit variables. Finally, on the last wall of the cycle, $0 + ++$ (between boxes 0111 and 1111), there is one alternative exit variable, $i = 2$.

Using all of these alternative exit variables for cycle A in equation (2.15), we compute the matrix R for cycle A as

$$R_A = \begin{pmatrix} 0 & -1 & 1 & 0 \\ -1 & -2 & 1 & 0 \\ 2 & 4 & -1 & -1 \\ 2 & 4 & -1 & -1 \\ -3 & -8 & 4 & 1 \\ -2 & -5 & 2 & 1 \\ -2 & -5 & 2 & 1 \end{pmatrix}.$$

Because necessarily $y_4 = 0$ on the starting wall, we can ignore the final column of R_A, and by a slight abuse of notation let y denote just $(y_1, y_2, y_3)^\top$. We can also remove duplicate rows of R_A, and actually we can remove rows for which the inequality is

already implied by another row or rows. Removing all such redundancies, R_A becomes

$$R_A = \begin{pmatrix} 2 & 4 & -1 \\ -2 & -5 & 2 \end{pmatrix}. \tag{2.37}$$

Similarly, following the same procedure, the R matrix that defines the returning cone for cycle B is

$$R_B = \begin{pmatrix} 6 & 11 & -2 \\ -2 & -4 & 1 \end{pmatrix}. \tag{2.38}$$

We can also describe the returning cones by means of their extremal vectors (vertices of a polygonal cross section of the cone) [10]. Berman and Plemmons [3, pp.1–2] define a cone as follows.

Definition 2.19. For a set $S \subseteq \mathbb{R}^n$, the set generated by S is the set of finite non-negative linear combinations of elements of S:

$$S^G = \left\{ \sum_{i=1}^{m} c_i x_i, \text{ for some } c_i \geq 0,\ x_i \in S, m \text{ finite} \right\}.$$

A set K is a cone if $K = K^G$.

So for any S, S^G is a cone. The returning cones for cycles A and B are (respectively):

$$C_A = \left\{ \left(0, \frac{2}{7}, \frac{5}{7}\right)^\top, \left(\frac{1}{2}, 0, \frac{1}{2}\right)^\top, \left(\frac{1}{3}, 0, \frac{2}{3}\right)^\top, \left(0, \frac{1}{5}, \frac{4}{5}\right)^\top \right\}^G, \tag{2.39}$$

$$C_B = \left\{ \left(0, \frac{2}{13}, \frac{11}{13}\right)^\top, \left(\frac{1}{4}, 0, \frac{3}{4}\right)^\top, \left(\frac{1}{3}, 0, \frac{2}{3}\right)^\top, \left(0, \frac{1}{5}, \frac{4}{5}\right)^\top \right\}^G. \tag{2.40}$$

On C_A and C_B their respective cycle maps act together as a Poincaré map. Since the cycle map acts on (a subset of) the starting wall, which is in $\mathbb{R}^{n-1}$, we can express it in terms of $(n-1)$-vectors and an $(n-1) \times (n-1)$ matrix. On the starting wall one of the coordinates is always zero ($y_i = 0$), so we can remove it from the cycle map. Also, since in the computation of the cycle matrix the i^{th} row is all 0's, we can remove that row also. So we can reduce the cycle map for a given cycle by one row and column. For this example we can write the map (2.12) for each of the cycles in terms of 3×3 matrices, and 3-vectors as

$$M_A = \frac{Ay}{1 + \phi^\top y}, \quad M_B = \frac{By}{1 + \psi^\top y},$$

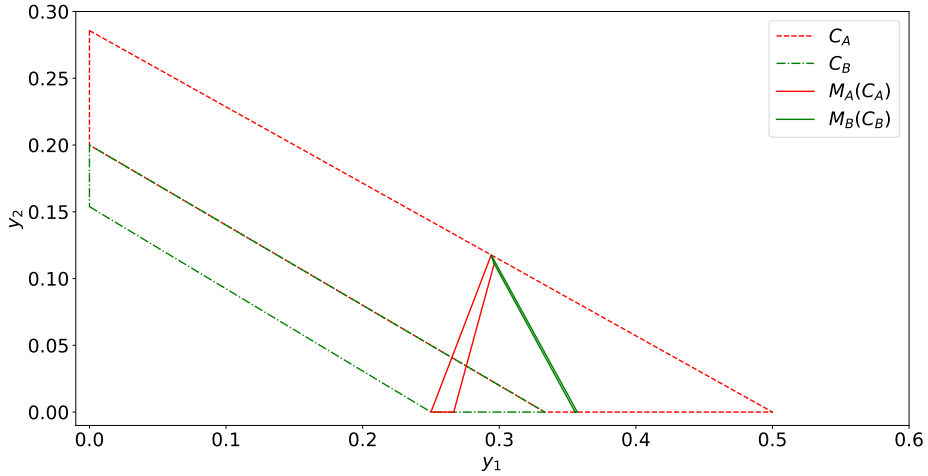


Figure 2.4: Returning cones for cycles A and B from Equations (2.39) and (2.40), represented by the cross section in the plane $y_1 + y_2 + y_3 = 1$ with only the first two coordinates plotted, and their images under their respective cycle maps. Note that although $M_B(C_B)$ is very narrow, it has non-empty interior.

where

$$A = \begin{pmatrix} -3 & -8 & 4 \\ -2 & -5 & 2 \\ -4 & -12 & 7 \end{pmatrix}, \; B = \begin{pmatrix} 5 & 8 & 0 \\ 6 & 11 & -2 \\ 12 & 20 & -1 \end{pmatrix},$$

$$\phi = (-4, -14, 10)^\top, \; \psi = (12, 18, 2)^\top.$$

Applying each of the cycle maps to their respective returning cones, one finds that each returning cone gets mapped into the union of the two. Returning cones are proper (non-empty, pointed) cones in $\mathbb{R}^{n-1}$ (a wall in $\mathbb{R}^n$) with vertex at the origin.

In a Glass network with uniform decay rates, rays map to rays under the mappings from wall to wall and radial dynamics is always convergent (trajectories starting on the same ray converge under iteration of the maps) [9]. So, in order to depict where returning cones, C_A and C_B, in our example, are in $\mathbb{R}^3$ (for a Glass network in $\mathbb{R}^4$), and their images under their respective cycle maps, one can represent a point on a ray with the ray's intersection with the plane of unit L_1 norm, $y_1 + y_2 + y_3 = 1$. Thus, the 3-dimensional returning cone is represented by a two-dimensional polygon, which is the cross section of the cone with the unit L_1-norm plane, with only the first two coordinates plotted in Figure 2.4. The image of each returning cone under its respective map, $M_A(C_A)$ and $M_B(C_B)$ is also depicted in Figure 2.4, showing that all points in $C_A \cup C_B$ map back into $C_A \cup C_B$, which is thus a trapping region. Thus, long term dynamics of this network (or at least the basin of attraction of this trapping region) can be represented on the subset of the TG that is just the union of cycles A and B.

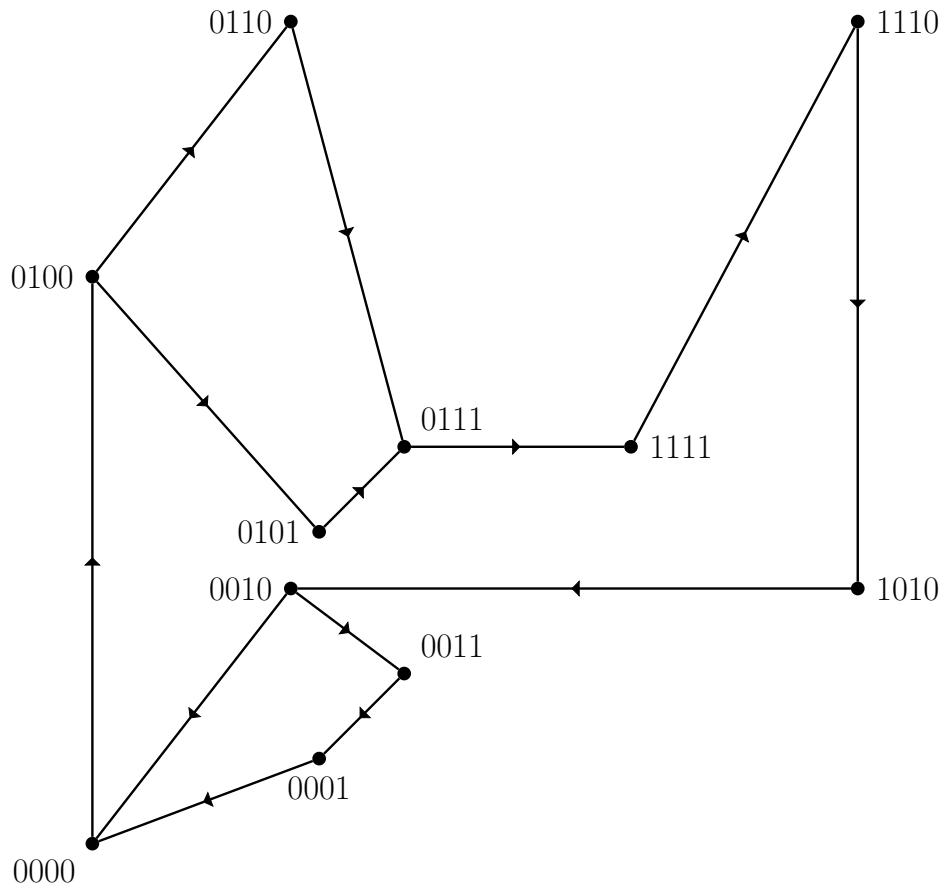


Figure 2.5: TG_r for the example Glass network in Equation (2.36) with only the trapping region.

Figure 2.5 shows the reduced graph, TG_r, that contains only the dynamics allowed by cycles A and B. The entropy calculated from the TG_r's adjacency matrix is $h_{\mathrm{TG}_r} \approx 0.224$, about one quarter the entropy of the original TG. However, the TG_r contains 4 cycles, not just A and B, and so includes dynamics not allowed in the trapping region.

As described in Section 2.4 therefore, we construct a $\mathrm{TG}_r(1)$ in which cycles A and B have been separated from each other. This is shown in Figure 2.6 and its entropy is $h_{\mathrm{TG}_r(1)} \approx 0.111$. This is less than h_{TG_r} because the two impossible cycles have been excluded.

Additionally, it can be seen in Figure 2.4 that the cycle sequence BB is forbidden, since $M_B(C_B) \cap C_B = \emptyset$. In other words, after a circuit of cycle B, the next cycle must be A, since $M_B(C_B) \subset C_A$. We can also deduce this information by considering the returning cones for each of the length two concatenations. These are depicted in Figure 2.7 and the partitioning of the original returning cones is clear. Concatenation BB is missing from the figure since its returning cone is empty. We can then create copies of cycle A and B that correspond to the three concatenations with non-empty

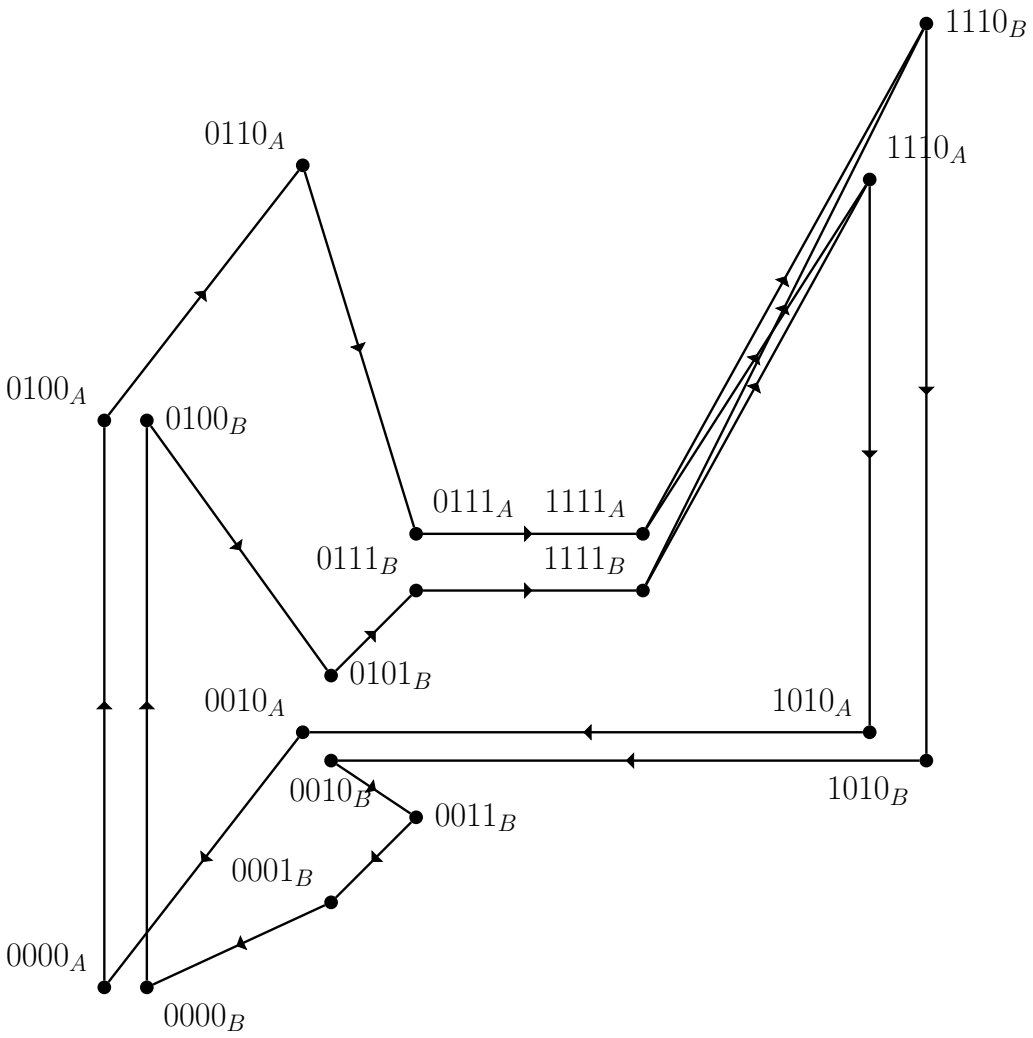


Figure 2.6: $\mathrm{TG}_r(1)$ for the example Glass network in Equation (2.36) where cycles A and B have been separated.

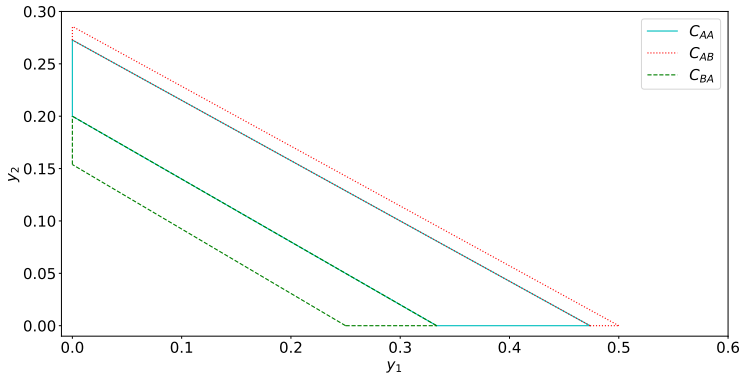


Figure 2.7: Returning cones for AA, AB, BA, and BB, partitioning the returning cones in Figure 2.4.

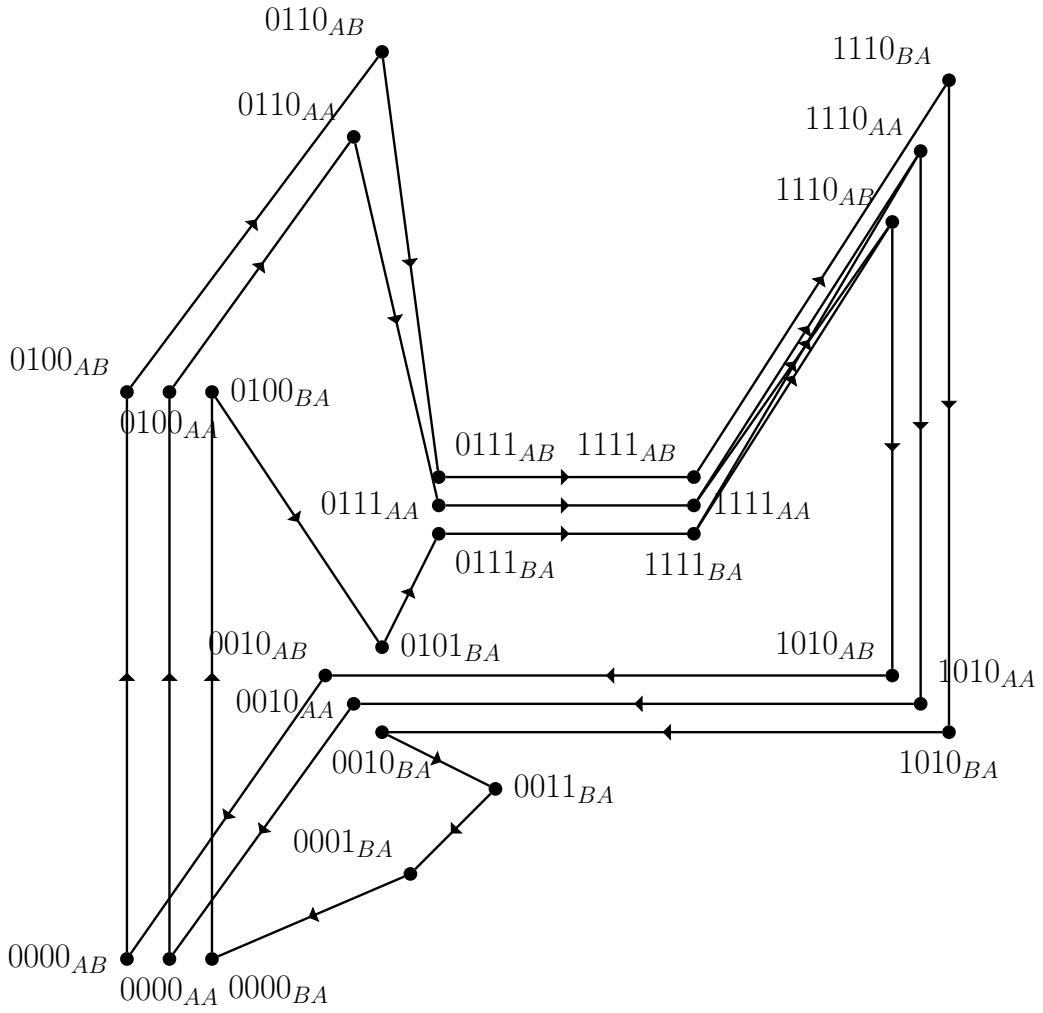


Figure 2.8: $\mathrm{TG}_r(2)$ for the example Glass network in Equation (2.36).

returning cones. The cycle left out is of course BB. Constructing the $\mathrm{TG}_r(2)$ as described previously gives the graph in Figure 2.8. The entropy of this graph is $h \approx 0.0813$.

It is clear from entropy estimates provided by the graph representations in Figures 2.5, 2.6, and 2.8 that our graph refinements quickly improve upon the original entropy bound of Farcot [15], without much extra work. An alternative justification involving state-splitting for the cycle separation procedure is given in the Appendix.

2.6 Numerical Estimation

Here, we numerically simulate our example network and extract the number of blocks from long trajectories to get estimates of the entropy as a check on the results of our refinements above. Numerical integration is done here simply by computing the wall-to-wall maps as a discrete process from a given initial point on the starting wall.

2.6.1 Have we found all of the blocks?

It is reasonable to expect that if we simulate many trajectories from different random initial conditions, we may be able to generate all the elements of $\mathcal{B}_n(\overline{\phi(\mathscr{D}_{TR})})$ for reasonably large n. This is easy to verify for small n since it is simple to calculate all the returning regions and the necessary trajectories are short.

We experimented with blocks of length $n = 50$, where 10^5 steps (wall-to-wall transitions) seemed to generate most of the blocks (the count appeared to have stopped increasing), but a longer simulation showed that after about 10^8 steps there was another jump in the number of blocks. This is likely due to a few blocks of length 50 having very narrow returning cones and thus not occurring often. Increasing the trajectory length did not cause any more increases up to 10^9 steps. Of course, we do not know for certain if there are additional blocks of length 50 or not, without calculating the returning regions of each possible block of length 50 and checking to see which are empty, but for large n that exhaustive check is computationally prohibitive. This experiment suggested that 10^9 steps might be sufficient to get a good estimate of entropy for n somewhat larger than 50.

In Figure 2.9 we plot the number of blocks of length 120 found in simulations of length up to 10^9 transitions (with the number of transitions plotted on a logarithmic scale). It is clear that the number of blocks continues to increase until about 10^8 steps. There are additional small jumps before about 1.5×10^8, but no further visible increases from there until 10^9 steps. If additional increases occur for larger n, their impact on entropy should be small, since that involves taking the logarithm of the number of blocks and dividing by n. Thus, we expect that for $n \leq 120$, 10^9 transitions will give us a reasonably tight lower bound on the number of blocks. If the system is

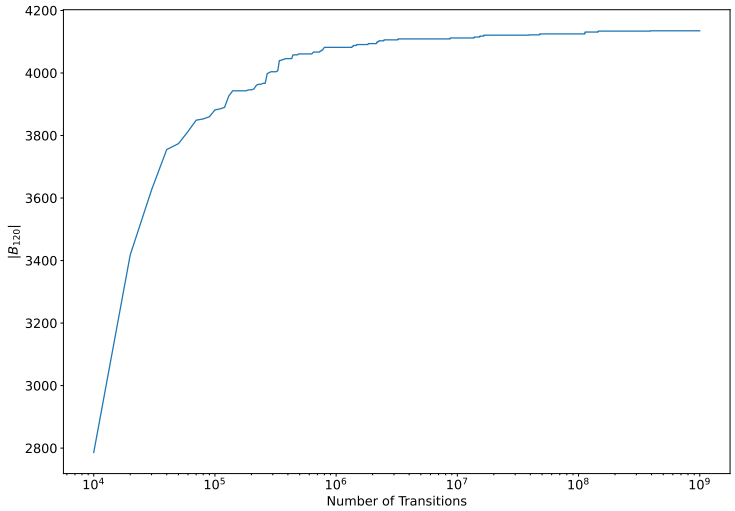


Figure 2.9: Blocks of length 120 vs number of transitions up to 10^9

chaotic, then the number of blocks of length n continues to increase with n, and we can only estimate the number of blocks up to some finite n. The count of blocks of some particular (large) length n from a long simulation gives a lower bound on $\mathcal{B}_n(\overline{\phi(\mathscr{D}_{TR})})$ and hence on $\log(\mathcal{B}_n(\overline{\phi(\mathscr{D}_{TR})}))$ for that particular n and for any larger values of n, but not necessarily on the entropy, since $\frac{1}{n}\log(\mathcal{B}_n(\overline{\phi(\mathscr{D}_{TR})}))$ may still decrease as n increases as the effects of possible longer forbidden blocks become significant. Thus, our lower bound on the number of blocks for a specific large n does not give a rigorous lower bound for the entropy.

One question that should be addressed is "what if this is just a very complicated limit cycle?". For our example it has been proven that there is no stable limit cycle [10]. However, in general this is something that needs to be considered. It has been shown that in example networks with only 4 variables, there can be surprisingly long stable limit cycles: examples with stable limit cycles of length 174 and 252 transitions have been identified [9]. Without knowing of the existence of such long stable limit cycles ahead of time, numerical simulations would need to be long enough to identify that the number of blocks stops increasing at the length of the cycle. However, our method of upper bounds would eventually catch this, if refinement was carried far enough. If there exists a stable limit cycle involving multiple returns to a starting wall, the graph would eventually reduce to a single long loop without any branching, and which crossed the starting wall multiple times. This structure always has an entropy of 0 and hence would make numerical simulation irrelevant.

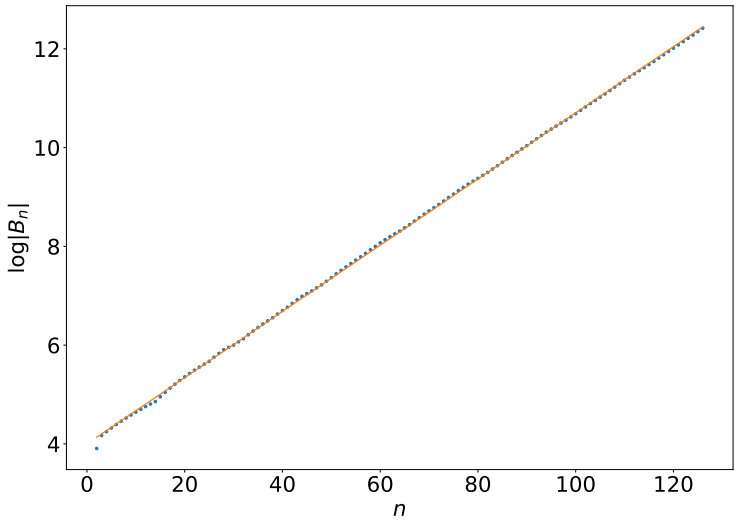


Figure 2.10: Logarithm (base 2) of number of blocks of length n for $2 \leq n \leq 126$ using a simulation of 10^9 transitions (dots) at each n. The solid line is the least squares best fit, which has slope ≈ 0.0670258.

2.6.2 Numerical (non-rigorous) bound on entropy

For a shift space X with nonzero finite entropy, $\mathcal{B}_n(X)$ grows approximately exponentially. For our example, it is reasonable to assume that for sufficiently large n, $|\mathcal{B}_n(\overline{\phi(\mathscr{D}_{TR})})| \approx a \cdot b^n$, where a and b are positive real constants. Hence, $h_{\phi(\mathscr{D}_{TR})} \approx \log b$. Thus for chaotic systems, a plot of $\log |\mathcal{B}_n(\overline{\phi(\mathscr{D}_{TR})})|$ vs n should have a linear trend, at least asymptotically.

Figure 2.10 plots $\log |\mathcal{B}_n(\overline{\phi(\mathscr{D}_{TR})})|$ against n for our example, calculated from numerically generated trajectories. The slope of the best fit line (least squares) is 0.067025, which gives an estimate of the entropy. Our second refinement from Figure 2.8 has an entropy of approximately 0.081. It is likely that there is a longer forbidden block that would only be found by refining further, so that 0.081 is an over-estimate. Alternatively, it may be that there is a very small returning cone that is visited extremely rarely, and was missed by the numerical simulation. If this is the case, then the numerical estimate of 0.067 underestimates entropy. It may also be that transients occur in the numerical simulations. In the theoretical (upper-bound) estimate, transients of some length are discovered as slightly longer sequences of cycles with an empty returning cone (see above). But if the numerical simulations include a transient, they will inflate the number of blocks, and potentially lead to an over-estimate of entropy. However, for a given trajectory, there can be at most one transient block of any given length n, so the effect on entropy is negligible for large n.

An observation from the numerical simulations is that the word $BAAB$ does not appear in any of the generated trajectories. However, its returning cone is nonempty. It may be that $BAAB$ is a rare sequence. The returning cone is very narrow and on the edge of the trapping region. Furthermore, the entropy of a representation that forbids $BAAB$ is 0.0706, close to our numerical estimate of 0.067, so our numerical simulation may have missed a rare occurrence and the true entropy may be closer to 0.081. On the other hand, it may also be that every word of some length $n > 4$ that includes $BAAB$ is forbidden. Given that we know this system is chaotic (or at least, aperiodic), the upper bound of 0.081 may be sufficient. While 0.067 is not a rigorous lower bound, it seems likely that the actual entropy is larger than this. The numerical estimate, 0.067 and the theoretical upper bound, 0.081, are significantly closer in value than any of the first three estimates of 0.873, 0.224, and 0.111 from the TG, TG_r and $\mathrm{TG}_r(1)$ respectively. Additionally, the upper bound 0.081 was achieved with very little work. In general, the refinement process may require more effort, but the process is simple to implement.

2.7 Conclusions

We have shown how to improve on the result of Farcot [15], which allows for the entropy of the dynamics of a Glass network to be bounded above by that of a symbolic dynamical system based on the TG. If one uses more information about the dynamics of such a network, by means of returning regions of cycles and trapping regions, one can construct discrete representations that more faithfully represent the dynamical possibilities, and thus give a tighter upper bound on entropy. The method uses structural changes to separate cycles in the TG and remove cycles that correspond to unrealizable trajectories, thus reducing the estimated entropy. The procedure can be taken to an arbitrary level of precision, giving a sequence of shift spaces that have decreasing entropy, and which we show approach the true entropy in the limit.

It should be noted that one could, in principle, avoid our cycle-separation procedure, and simply identify forbidden sequences in the original alphabet that correspond to forbidden sequences of cycles, or even sequences of boxes that do not necessarily correspond to full cycles. This would avoid the need for defining larger and larger alphabets on the refined graphs. However, the information about the dynamics we have comes from returning regions for cycles and trapping regions on a starting wall, and this is the information we use to identify forbidden cycles or sequences of cycles, so it is natural to structure our symbolic dynamics around them. Additionally, the refined graphs have the advantage of allowing simple calculation of the entropy via the Perron eigenvalue of the graph's adjacency matrix.

We have removed from consideration certain types of network structure that would

make our procedure more difficult, in particular, networks in which one cannot avoid an infinite number of first-return cycles on any starting wall, or at least a potentially infinite number based on the TG alone (Condition 5). It might be possible to extend our work to include such examples. However, the remaining class of networks is large and dynamically diverse, with many good candidates for designs of TRNGs.

Indeed, one potential application of this work is to quantify entropy in free-running electronic circuits based on standard Boolean logic gates, which can be modelled as Glass networks and could serve as designs for TRNGs. Randomness in a TRNG comes mainly from thermal noise, but the strength of the idea of an underlying chaotic (deterministic) circuit design is that there is already positive entropy even without the thermal noise. The method proposed here allows this entropy to be estimated.

We have assumed throughout that we have a minimal trapping region. However, if there are multiple trapping regions in a given starting wall, we must deal with each minimal trapping region (and thus each attractor) separately in order to use the procedure detailed in Section 2.4.

It should be possible to automate our method for estimating entropy by an upper bound based on dynamical information that allows refinement of the state transition graph. Given a network structure, and no prior information about the dynamics, can one automatically detect a trapping region and extract a good upper bound on entropy? This is an idea for future work.

Appendix A. State-splitting

Another justification for the improved upper bounds achieved by separation of cycles (rather than exploiting embeddings) is to achieve the separated cycles by performing step-by-step state-splittings on the TG_r, each step of which preserves entropy. In doing so, one can separate all possible cycles on the TG_r, and then remove the nodes and edges not associated with cycles contributing to the trapping region. Additionally, state-splittings can be continually performed in such a way as to retrieve concatenation representations, which can be pruned in a similar way. This state-splitting and pruning procedure can be done to exclude arbitrarily long concatenations of cycles to achieve exactly the same graph that was achieved by embeddings as described in Section 2.4. To demonstrate this idea we will perform a sequence of state-splittings on our example network and show how the resulting graph can be pruned to retrieve the $\mathrm{TG}_r(1)$ we found in the previous subsection.

We begin by considering the TG_r for our example in Figure 2.5. Starting with the node labeled 1111 (the initial node of the starting edge) and proceeding backwards around the cycles, the first occurrence of a node with multiple outputs is 0100, where we can perform an out-splitting (as defined by Lind and Marcus [45, p.52]), in which the node and incoming edges are duplicated, while the outgoing edges are partitioned.

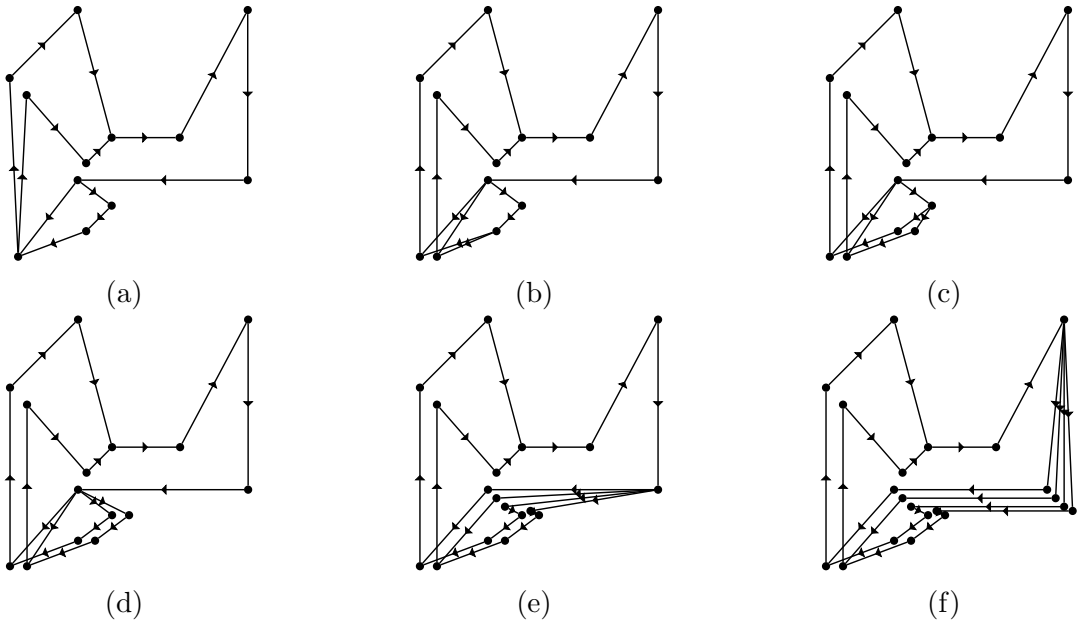


Figure 2.11: The six sequential out-splittings of the TG_s for the example Glass network in Equation (2.36).

This produces the graph in Figure 2.11a. Coninuing backwards around the cycles, and performing out-splittings wherever applicable, we obtain the sequence of graphs depicted in Figures 2.11b – 2.11f, and then Figure 2.12 completes the out-splittings necessary to separate all possible cycles on the TG_r.

Now starting at the nodes labelled 1110^1 through 1111^4, and proceeding forwards around the cycles, the first nodes encountered with multiple input edges are 0000^1 and 0000^2, where we perform an in-splitting (as defined by Lind and Marcus [45, p.53]), in which the node and outgoing edges are duplicated, while the incoming edges are partitioned. This produces the graph in Figure 2.13a. Continuing forwards around the cycles, and performing in-splittings wherever applicable, we obtain the sequence of graphs depicted in Figures 2.13b – 2.13d, and then one final in-splitting produces the graph called the TG_s, shown in Figure 2.14. Since the TG_s was formed from the TG_r by a sequence of out-splittings and in-splittings, they are conjugate shift spaces and hence have the same entropy.

The TG_s has a convenient benefit over the TG_r in that each of the four possible cycles on the TG_r is separated, apart from cross edges in place of the original starting edge in the TG_r. Since cycles with subscripts C and D are never realized in the true dynamics, removing them and their connecting cross edges from the TG_s produces a graph that can still be used for an upper bound on entropy. In fact, removing these cycles produces exactly the $\mathrm{TG}_r(1)$ in Figure 2.6. Performing this same procedure to the $\mathrm{TG}_r(1)$ and then pruning the cycle associated with cycle concatenation BB will produce the $\mathrm{TG}_r(2)$ in Figure 2.8.

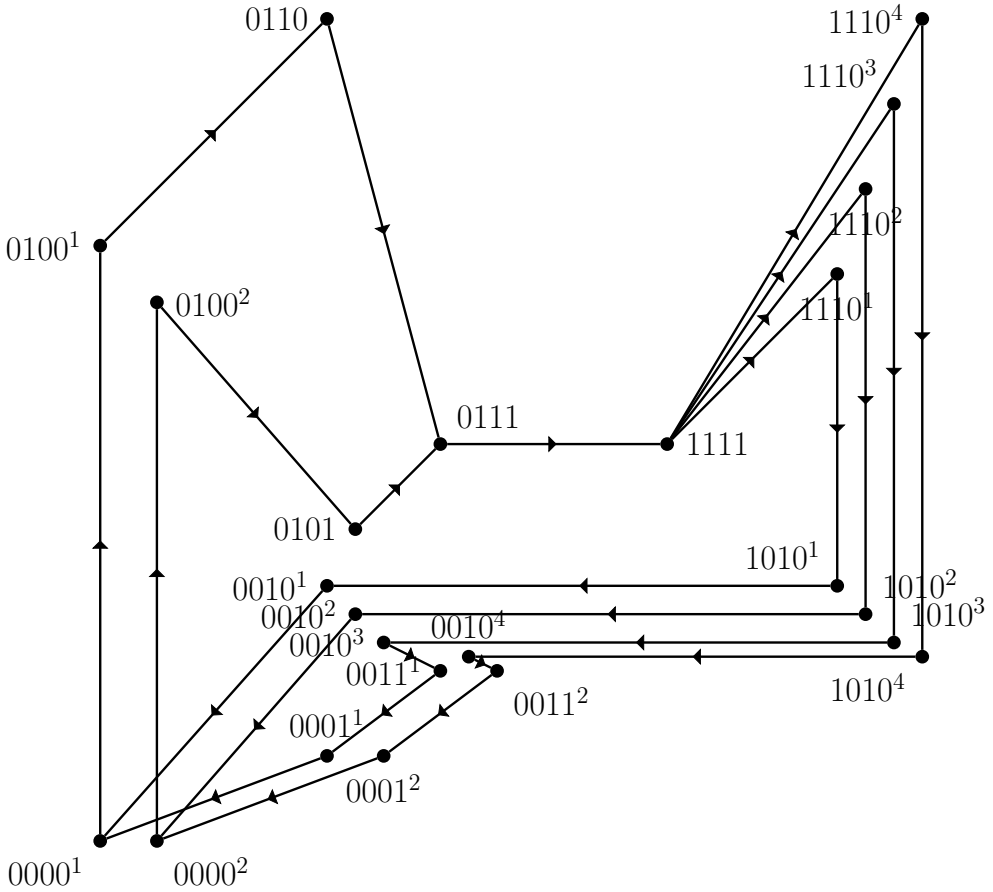


Figure 2.12: Seventh state splitting of TG_r for the example Glass network in Equation (2.36).

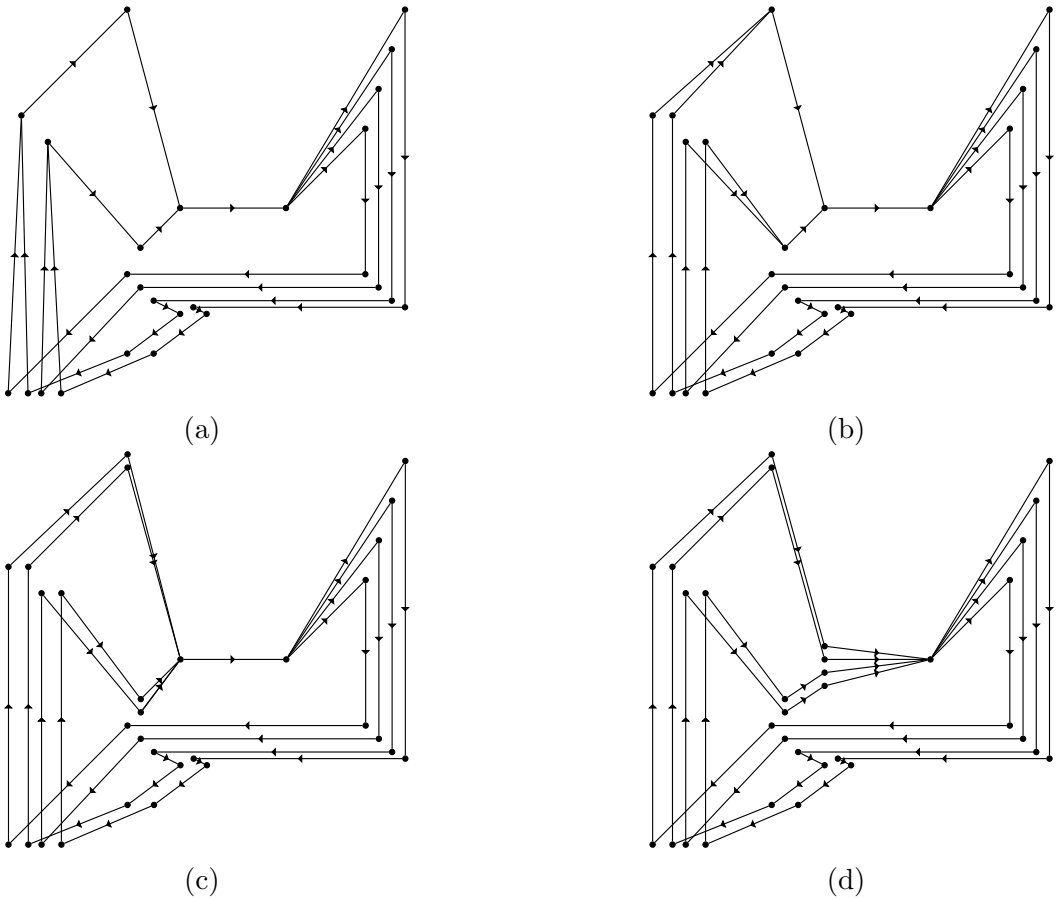


Figure 2.13: The four sequential in-splittings of TG_r for the example Glass network in Equation (2.36).

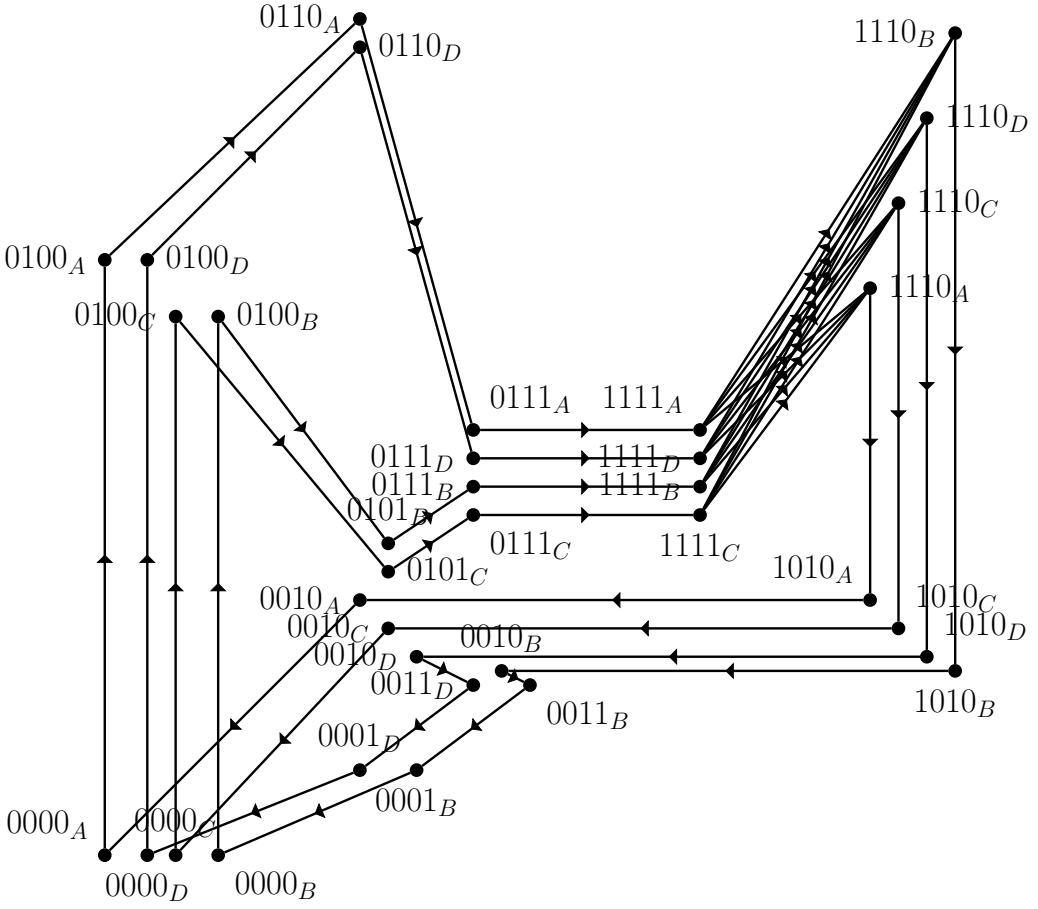


Figure 2.14: TG_s for the example Glass network in Equation (2.36), the result of the state splitting procedure applied to TG_r.

As long as a network satisfies Conditions 1, 2, 4, and 5, this state-splitting and pruning procedure can always be done in such a way that the resulting graph after pruning is a $\mathrm{TG}_r(k)$ for some k. As a result, it is possible to achieve the same results as in Section 2.4.

Acknowledgments

This work was partially supported by a Discovery Grant to RE from the Natural Sciences and Engineering Research Council (NSERC) of Canada, and a British Columbia Graduate Scholarship to BW.

Chapter 3
An algorithmic approach to entropy estimation

The results of the previous chapter provide the necessary theory to assess the potential for Glass networks to have irregular dynamics, and thus aid in the design of TRNGs. Glass networks are dynamically diverse [12] and there are many network structures that could produce dynamics with high entropy. In order to identify which structures result in the desired dynamics, a large survey of network designs would be beneficial.

While our theory gives a reasonably simple procedure for attaining good upper bounds on symbolic entropy, a lot of *a priori* information was used when working on the example. In the example, we knew beforehand that there existed a trapping region and we knew which cycles on the TG made up its composition. If one wishes to perform a large survey of networks to find examples with high symbolic entropy, an in-depth analysis of each network will be a lengthy endeavour and will likely limit the scope of the survey. It would be advantageous to such a survey if the theory we have developed could be quickly applied to any generic network structure with no prior knowledge of any trapping region.

In order to quickly assess the irregularity of an arbitrary Glass network structure, an automation of the process is necessary. A lot of operations are required to perform the reductions. However, all of these operations lend themselves well to computer code implementation. Ideally, a computer implementation of an algorithm could simply take a network's TG and a desired level of refinement for the estimate as inputs, and then the code would produce the entropy estimate. We propose such an algorithm here.

In Section 3.1, we will describe in detail an algorithmic approach to the entropy refinement process, including finding the cycles, forming a trapping region, and pruning the transition graph. These will allow the implementer to take an arbitrary Glass network, specify the level of refinement of the estimate, and then produce the entropy. In Section 3.2 we will describe how all the components from Section 3.1 fit together. In Section 3.3, we will apply the algorithm to some different Glass network examples to demonstrate its efficacy at deciphering different types of dynamics present in Glass

networks. This will also demonstrate the types of results one can expect from our algorithm when applied to networks with different types of dynamics. Note that in this chapter, Condition 5 will not be imposed. This is so we can build as general an algorithm as possible. Additionally, it is impossible to tell which networks satisfy Conditions 4 and 5 from a table of focal points alone. Necessary steps will be taken during the construction of the algorithm to ensure that networks not satisfying Conditions 4 and 5 are removed from consideration.

3.1 Description of the components

This section will primarily be focused on how all the individual components involved in our reduction process work. Additionally, we will discuss how they might fit together within the context of a general algorithm. By the end of this section, one will be able to use these descriptions to automate the entire entropy approximation process discussed in the previous chapter, thus producing a tool that can be used to study complex dynamics in Glass networks.

Before we start any discussion of the algorithm or its components, we will reintroduce the example from Chapter 2. This example is simple in the sense that, after transients die out, all of the dynamics end up being just concatenations of two distinct cycles. However, the dynamics have been shown to display irregular (chaotic) behaviour. Since the network dynamics are quite simple, this makes it the perfect example to use as an instructive tool while developing the algorithm. More complicated examples will be examined in the next section.

The example network is again described as follows:

$$\begin{aligned}
\dot{y}_1 &= -y_1 + 2\left(\bar{Y}_3 Y_4 + Y_2 Y_3\right) - 1 \\
\dot{y}_2 &= -y_2 + 2\left(Y_1 \bar{Y}_3 Y_4 + \bar{Y}_1 Y_3 Y_4 + \bar{Y}_1 \bar{Y}_3 \bar{Y}_4\right) - 1 \\
\dot{y}_3 &= -y_3 + 2\left(\bar{Y}_1 Y_2 + Y_1 Y_4\right) - 1 \\
\dot{y}_4 &= -y_4 + 2\left(Y_2 \bar{Y}_3 + \bar{Y}_1 Y_3\right) - 1
\end{aligned} \tag{3.1}$$

where $Y_i = 0$ if $y_i < 0$, and 1 if $y_i > 0$. $\bar{Y}_i = 1 - Y_i$. The table of focal points for this network is shown in table 3.1 and the TG in figure 2.1.

As was discussed previously [10, 59], the dynamics of this network quickly become confined to a trapping region that restricts dynamics to follow one of the two cycles A and B:

$A : 1110 \to 1010 \to 0010 \to 0000 \to 0100 \to 0110 \to 0111 \to 1111,$

$B : 1110 \to 1010 \to 0010 \to 0011 \to 0001 \to 0000 \to 0100 \to 0101 \to 0111 \to 1111.$

However, arriving at this conclusion involved some prior knowledge of the network's

Y_1	Y_2	Y_3	Y_4	F_1	F_2	F_3	F_4
0	0	0	0	-1	1	-1	-1
0	0	0	1	1	-1	-1	-1
0	0	1	0	-1	-1	-1	1
0	0	1	1	-1	1	-1	1
0	1	0	0	-1	1	1	1
0	1	0	1	1	-1	1	1
0	1	1	0	1	-1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	-1	-1	-1	-1
1	0	0	1	1	1	1	1
1	0	1	0	-1	-1	-1	-1
1	0	1	1	-1	-1	1	-1
1	1	0	0	-1	-1	-1	1
1	1	0	1	1	1	1	1
1	1	1	0	1	-1	-1	-1
1	1	1	1	1	-1	1	-1

Table 3.1: Table of focal point values for example 3.1.

underlying dynamics. If we wish to refine the TG in Figure 2.1 without any prior knowledge of its underlying dynamics, there are a few procedures that we need to discuss in detail:

1. Identifying a suitable starting wall on which to define cycles,
2. Find all the cycles passing through the starting wall,
3. Find which of the cycles passing through the starting wall make up a trapping region,
4. Find which concatenation of cycles of length $2, 3, \ldots$ up to some n are allowed,
5. Prune the TG to be consistent with all of this and extract the entropy.

While simple in theory, each of these procedures have subtleties that, if not accounted for, can cause difficulties later. We start first with the first component of the algorithm, identifying a starting wall.

3.1.1 Identifying a starting wall

The first step in the reduction process is identifying a starting wall on which to define cycles. A perfect candidate for a starting wall would be one that is common to every cycle in the network. However, determining this from just a network's governing ODEs

is not trivial, and is not possible in general. There are many ways one could identify a starting wall; we suggest using the wall that is visited most frequently during a long numerical integration. Since there are only a finite number of walls in the type of systems we are interested in studying, it is simple to associate each vector from a trajectory generated in this manner with a wall[1]. Doing this with each vector in the trajectory, we will have a list of walls that each point in the trajectory is contained in. With this list in hand, it is simple to identify which wall is visited most frequently during the numerical integration. Since trajectories are easy to construct for Glass networks, this part of the algorithm is trivial. Below we have provided a python implementation of how one might do this for our example. Running this code will output the wall between boxes 1111 and 1110. A clever observer will note that this is exactly the wall we used in our paper [59].

We now move to focus on finding cycles about the starting wall.

```
# Python library useful for numerical calculations.
import numpy as np

# Matrix that stores the focal point coordinates for a network.
fpMAT = np.array([[-1,1,-1,-1],[1,-1,-1,-1],[-1,-1,-1,1],[-1,1,-1,1],
          [-1,1,1,1],[1,-1,1,1],[1,-1,1,1],[1,1,1,1],
          [-1,-1,-1,-1],[1,1,1,-1],[-1,-1,-1,-1],[-1,-1,1,-1],
          [-1,-1,-1,1],[1,1,1,1],[1,-1,-1,-1],[1,-1,1,-1]])

# Matrix that stores the binary representation of boxes.
boxMAT = np.array([[0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],
              [0,1,0,0],[0,1,0,1],[0,1,1,0],[0,1,1,1],
              [1,0,0,0],[1,0,0,1],[1,0,1,0],[1,0,1,1],
              [1,1,0,0],[1,1,0,1],[1,1,1,0],[1,1,1,1]])

def t(Y_0,f):

# This function takes in a vector coordinate and a focal point,
   calculates for each component of the vector the time to reach the
   nearest switching boundary, and returns the smallest positive time
    along with the direction.

    T = np.log(np.ones([4]) - Y_0/f)
    t_j = max(T)
    pos = 0

    for i in range(len(T)):

        if T[i] > 0 and T[i] <= t_j:
            t_j = T[i]
```

[1]The hyper cube graph in figure 2.1 is the TG for our example and has exactly 32 edges. In general n-cubes have 2^n nodes and $2^{n-1}n$ edges.

```
            pos = i

    return (t_j,pos)

def B(fp,j):

# This function is used to calculate the matrix in equation (2.10).
   It takes in a focal point and switching direction and returns a
   matrix.

    I = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
    e = np.zeros(len(fp))
    e[j] = 1

    return I - (1/fp[j])*np.outer(fp,e)

def psi(fp,j):

# This function is used to calculate the psi vector in equation
   (2.10). It takes in a focal point and switching direction and
   returns a vector.

    e = np.zeros(len(fp))
    e[j] = 1

    return -e/fp[j]

def Y_kPlus1(Y_k,fp,j):

# This function is the wall-to-wall map in equation (2.9). It takes
   in a vector, focal point, and switching direction and returns the
   vector on the next wall that the input vector is mapped to.

    return (B(fp,j).dot(Y_k))/(1 + psi(fp,j).dot(Y_k))

def find(ell):

# This function takes in a binary box representation and returns the
   position in boxMAT that it occupies.

    boxMAT = np.array([[0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],
             [0,1,0,0],[0,1,0,1],[0,1,1,0],[0,1,1,1],
             [1,0,0,0],[1,0,0,1],[1,0,1,0],[1,0,1,1],
             [1,1,0,0],[1,1,0,1],[1,1,1,0],[1,1,1,1]])

    for i in range(16):

        if np.array_equal(boxMAT[i],ell):
```

```
            return i

def SOL(Y_0,fp_0,box_0,nSteps):

# This function takes in an initial condition, focal point for the
   initial condition, box coordinate for the initial condition and a
   specification for the length of the numerical simulation and
   returns a numerical simulation for the network starting from the
   initial condition.

    fpMAT = np.array
   ([[-1,1,-1,-1],[1,-1,-1,-1],[-1,-1,-1,1],[-1,1,-1,1],
          [-1,1,1,1],[1,-1,1,1],[1,-1,1,1],[1,1,1,1],
          [-1,-1,-1,-1],[1,1,1,-1],[-1,-1,-1,-1],[-1,-1,1,-1],
          [-1,-1,-1,1],[1,1,1,1],[1,-1,-1,-1],[1,-1,1,-1]])

    boxMAT = np.array([[0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],
             [0,1,0,0],[0,1,0,1],[0,1,1,0],[0,1,1,1],
             [1,0,0,0],[1,0,0,1],[1,0,1,0],[1,0,1,1],
             [1,1,0,0],[1,1,0,1],[1,1,1,0],[1,1,1,1]])
    sol = []
    fp_0 = fpMAT[14]
    box_0 = boxMAT[14]
    Y = Y_0
    fp = fp_0
    box = box_0
    sol.append(Y)

    for i in range(nSteps-1):

        if box[t(Y,fp)[1]] == 1:
            box[t(Y,fp)[1]] = 0

        else:
            box[t(Y,fp)[1]] = 1

        Y = Y_kPlus1(Y,fp,t(Y,fp)[1])
        fp = fpMAT[find(box)]
        sol.append(Y)

    return sol



def wall(vec):

# This function converts vectors into their wall representation where
    '+' indicates the variable is positive, '-' indicates the
```

```
    variable is negative and '0' indicates the variable is 0.

    Wall = [0,0,0,0]

    for i in range(len(vec)):

        if vec[i] == 0:
            Wall[i] = '0'

        elif vec[i]/abs(vec[i]) == -1:
            Wall[i] = '-'

        elif vec[i]/abs(vec[i]) == 1:
            Wall[i] = '+'

    return ' '.join(Wall)

def The_Wall(traj):

# This function takes in a trajectory solution and returns the wall
    that is visited most often during the simulation.

    Walls = {wall(traj[0]):1}

    for vec in traj:

        if any(wall(vec) ==  w for w in Walls):
            Walls[wall(vec)] += 1
            continue

        else:
            Walls.update({wall(vec): 1})

        walls = list(Walls.keys())

    thewall = walls[0]

    for w in walls:

        if Walls[w] >= Walls[thewall]:
            thewall = w

    return thewall

def main():

# Here we generate a random initial condition on the wall (+++0),
    generate a numerical simulation of the network of length 3000, and
```

```
    then find the most visited wall during the simulation and print
   its label. We do all of this within a main function to organize
   work flow.

    import random
    t_max = 3000
    traj = SOL(np.array([random.uniform(0.01, 1),random.uniform(0.01,
    1),random.uniform(0.01, 1),0]),fpMAT[14],boxMAT[14],t_max)

    print(The_Wall(traj))

if __name__ == '__main__':
    main()
```

3.1.2 Cycles about a wall on the TG

Before any refinements can be made to the TG, we need to identify a trapping region. Before this however, we need to find all of the complete cycles that pass through the starting wall. Here we suggest the use of a tree structure. While it may not be the most optimal data structure for doing this quickly, it will be crucial later in the algorithm that at each wall instance we can store data pertaining to the number of times the node's name has appeared, returning regions, along with other properties that will be defined as we progress through this section. Additionally, since we will be considering arbitrary concatenations of cycles, using the tree structure facilitates considering these concatenations. To begin this discussion, we will assume that for a given network Conditions 1 through 5 have been satisfied.

We can start constructing the tree. It is convenient to introduce some notation now that will be used throughout the remainder of the algorithm. Since all the cycles that will form a trapping region pass through the same starting wall, there exist at least two nodes on the TG that will be common to all the cycles. As a result, along the flow across the wall, *i.e.*, the direction of the directed edge, it will be convenient to refer to the node that the edge is pointing to as the starting node, n_s, and the node that the edge is pointing away from as the terminal node, n_t. This gives a starting node that every cycle starts from and a terminal node which every cycle ends with. In the context of our instructional example, $n_s = 1110$ and $n_t = 1111$.

Now, Condition 5 specifies that the networks in consideration contain a trapping region that is made up of only a finite number of cycles. This is particularly useful in the event that the TG structure contains a cycle that includes a loop that is disjoint from the starting wall. In this situation, the TG allows for an infinite number of cycles, and then special care is needed to ensure that the algorithm does eventually halt. Below we define a maximum repetition number, which is imposed to ensure that in the case of a cycle containing a disjoint loop from the starting wall, only a finite

number of cycles can be constructed from such a structure[2].

Definition 3.1. The maximum repetition number, $R_{\max}$, for a network is the specified number of times any cycle can return to any given node contained within the cycle.

Now that n_s, n_t, and $R_{\max}$ are all specified, we can focus on actually identifying all the cycles passing throughout the staring wall. We now introduce the tree structure that is the basis for the entire algorithm.

To begin, we start with what is needed from the tree. The entire tree structure is based on nodes that should be defined in the following way:

```
class Node():
# Here we define the node class that we will use to define our tree
   structure along with all of its data type properties.
    def __init__(self, name, pos, parent):
        self.name = name
        self.parent = parent
        self.pos = [0,0]
        self.children = []
        self.rep = 0
        self.cone_data = []
        self.seq_pos = 0
        self.cone_image = []
```

Each of the node properties has a specific function within the context of the algorithm. Here is a brief description of each:

name: A label for the node in the tree.

parent: Specifies which node is the current node's parent in the tree.

pos: Specifies where in the tree the node lies. Level is top down and row is left to right.

children: Specifies a list of nodes that are connected to the current node as children.

rep: Specifies the node's repetition number.

cone_data: A place to store data for the returning regions. Note this will only be used on instances of n_t.

seq_pos: Specifies how many sequences of cycles a particular branch has. Note this will only be used on instances of n_t.

[2]It may be the case that one chooses a maximum repetition number that is not large enough to find a trapping region within the set of cycles passing through the starting wall. If this is the case, one can simply increase the repetition number until a trapping region emerges, or discard the network from consideration. For our example network, a repetition number of 1 was sufficient; this may not always be the case.

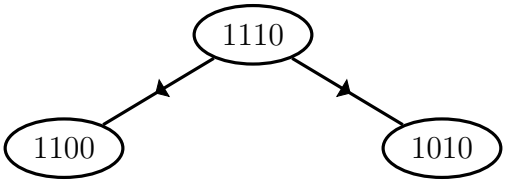


Figure 3.1: Starting node at the root of the tree with child leaf nodes for example Glass network.

cone_image: A place to store data for the image of returning regions under their respective mappings. Note this will only be used on instances of n_t.

Using n_s as a starting point, we begin construction of the tree by putting n_s at the root. We then attached the nodes that can be reached from n_s, as given by the TG, as child nodes. For our example, this is depicted in Figure 3.1. When a new node is added to the tree, the repetition number is calculated by simply counting how many times a node with that same name appears in the path back to the root. Once a leaf node's children are calculated and added to the tree, they then each have their paths to root retrieved and the number of times the name occurs is appended as the repetition number. If that number is larger than R_{Max}, no more child nodes will be added to that particular leaf.

As we move down the tree, it will be convenient to refer to it in levels. So, for example, in Figure 3.1, the node 1110 is on level 0 and the child nodes 1100 and 1010 are on level 1. With the first set of child nodes added to the root, we can now move down a level to focus our attention on them. For this part of the algorithm, there are only two conditions to check when adding children to each node on the current "bottom" level of the tree:

1. Is the given node's repetition number equal to R_{Max}?

2. Is the given node the terminal node?

For a given leaf node, if either one of these statements is true, that branch halts growth. If neither of these statements is true, then the node has its appropriate children added on the next level down. This continues for each node on each level until either the repetition number is reached or the terminal node is reached. Tree growth stops entirely when there exist no child nodes that can be added to the bottommost level of the tree.

The structure that we will be left with is a tree that has n_s at the root, and branches that lead to leaf nodes that are either some node midway around a cycle that reached R_{Max} or the terminal node. Since we are only concerned with complete cycles, this is where the first pruning of the tree takes place. Each leaf node that does not have the same name as the terminal node represents an incomplete cycle. These branches can be pruned back until reaching a branching point. To prune the tree, we systematically

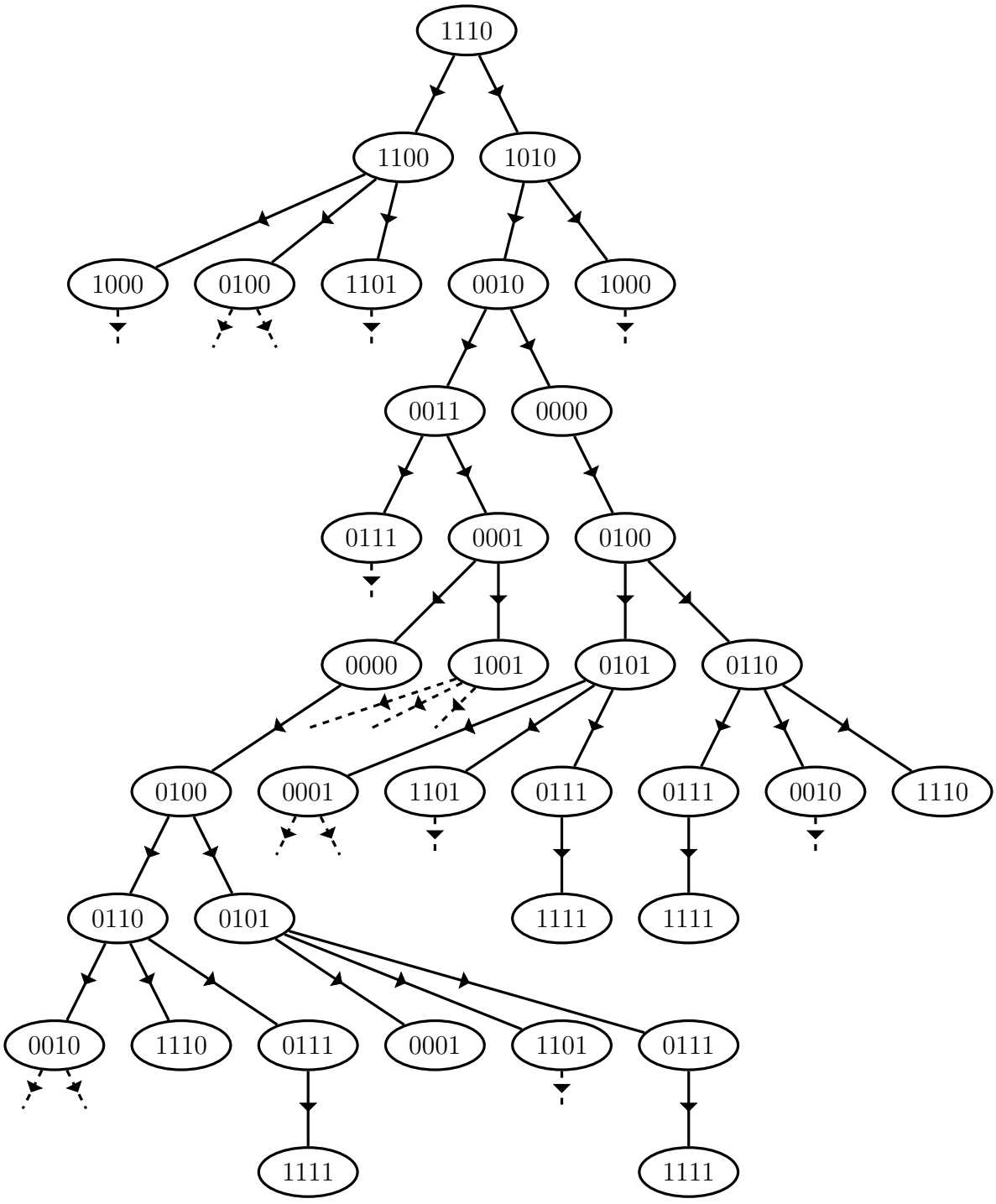


Figure 3.2: Halted tree construction for example Glass network, $R_{\text{Max}} = 1$. Dashed directed edges indicated that the branch continues on to some later halting point not shown. Only four of the many possible complete cycles are shown here. The rest are assumed by dashed lines with arrows. This shows the tree before any pruning is done.

remove leaf nodes that do not match until the only leaves that remain all have the same name as the terminal node. To do this is straightforward. Since we will refer to it again later we will call it pruning step 1. It works as follows:

1. Generate a list of leaf nodes,
2. Then remove any leaf nodes from the tree that do not have the same name as the terminal node,
3. Repeating this process of removing non-terminal leaf nodes until only terminal leaf nodes remain.

Removing all the incomplete cycles in this fashion results in a tree that has the starting node as the root and following each branch down to completion gives a cycle on the TG. Counting the number of instances of the terminal node will reveal how many complete cycles, under R_{Max}, the TG permits. The next step is to consider returning regions for each cycle and use those to identify a trapping region.

3.1.3 Pruning the tree to a trapping region

Up to this point, everything we have discussed has been left as general as possible. For our discussion on returning regions we will restrict our focus to the case of equal decay rates by imposing Condition 3. Thus, in equation 2.1, $\lambda_i = \lambda$ for all i. In the case of unequal decay rates, the theory developed in the previous chapter does not translate well to computation. Pre-images of walls under the mapping $\mathcal{M}$ are no longer linear subspaces, but are curved. It is however easy to calculate returning regions in the case of equal decay rates [9]. In this case, the regions are polygonal cones. The theory discussed by Edwards lays out the procedure to calculate returning cones by way of systems of inequalities that translate perfectly to computation. See equations (2.32) and (2.33) in Section 2.5. So for the purposes of our implementation, we will restrict to the case of equal decay rates. The description of the algorithm will be left as general as possible, but for our code implementation, we will follow the theory of Edwards.

We have found all complete cycles passing through the starting wall from the tree. Now, we find each cycle's returning region and prune those which do not contribute to a trapping region. To do this, we calculate returning regions for each cycle, their respective cycle maps, and the region's respective images under their cycle mappings. We can determine which terminal leaf nodes correspond to cycles that are a part of the trapping region by checking first that a leaf node's corresponding cycle does not have an empty returning region, and if that is the case, checking that other returning regions get mapped into it. If a cycle has an empty returning region, then it is an impossible cycle and must be removed from consideration. If a returning region does not intersect with the images of any of the returning regions of any of the cycles, then it is transient

behaviour and must also be removed from consideration. Then we remove leaf nodes of cycles that do not form part of the trapping region, and re-apply pruning step 1 to remove the rest of the cycle back to a branching point.

To determine if a returning cone is empty is simple. We show below how to do this when $n = 4$ but generalization to higher dimensions is straightforward. For rows R_i and R_j of the matrix R, where $i \neq j$, find the solution to the matrix equation

$$\begin{pmatrix} R_i \\ R_j \\ P \end{pmatrix} \cdot y = \begin{pmatrix} 0 \\ 0 \\ b \end{pmatrix} \tag{3.2}$$

where $P = (p_1, \ldots, p_{n-1}) \in \mathbb{R}^{n-1}$ and $b \in \mathbb{R}$ define the plane that one wishes to find the intersection with. We suggest using the plane that defines a boundary of the 1-ball that lies within the wall of interest that intersects all the cones. In our example this is just the 1-ball centred at the origin. After doing this for every possible i and j such that $i \neq j$, take all the solutions and see which satisfy the matrix equation $R \cdot y = 0$. This will give the vertices of the polygon that is formed when taking the cycle's returning cone and intersecting it with the plane previously mentioned. If less than three solutions satisfy $R \cdot y = 0$, then the cone must be empty and that cycle can be removed from consideration. This is due to the fact that if there are two or fewer vertices in the plane intersection with the "cone", it isn't a proper cone. It is either a plane, a line, or non-existent. Since in the domain $\mathscr{D}$ all threshold faces of codimension 2 or more have been removed, the boundaries of cycle returning cones are not included in $\mathscr{D}$. Hence if a cycle's "returning cone" has less than three vertices when intersecting the plane of interest in the wall, its interior is empty and since the boundaries are also not included, the cone is just the empty set. Hence the cycle is impossible.

To check for intersections of a returning cone with images of other returning cones, first find the images of each returning cone by applying the corresponding cycle map to its vertices and project them back onto the plane of interest. Then check to see if any of the vertices of the cone images satisfy $R \cdot y > 0$ for the returning cone. If any of them do, then there is a non-trivial intersection. If none of them do, then take each linear inequality defining a linear boundary of the first cone ($R_i \cdot y = 0$) and find its intersection with each linear boundary of the image of the other cone ($R'_j \cdot y = 0$, formed from pairs of vertex images), and with the hyperplane of interest ($P \cdot y = 0$). The set of intersection points thus obtained that satisfy $R \cdot y = 0$, along with vertices of one cone that lie inside the other, form the convex hull of the intersection of the two cones. If the set of intersection points is empty, then there is no intersection.

For each leaf node on the tree, we retrieve its "root to leaf"-path, which is a cycle, and store its returning region, cycle map, and region image under its cycle mappings. Finding the cycle mappings given a returning cone, its image can be calculated, by

applying the cycle map to the extremal vectors of the cone (*i.e.*, pairwise intersections of linear inequalities defining the returning cone that satisfy all other returning cone inequalities. The computation of returning cones in the case of equal decay rates is not obvious. As a result, we have provided below a code implementation along with how this works for cycles A and B in our example. The final output of this program is a list of points on the plane $x_1 + x_2 + x_3 = 1$, since in the example, the starting wall is in the first octant of the hyperplane in the first three variables. The points represent the vertices of the cross section of the cone with this plane and the cone can be identified by these points. The convex hull of these points can be used to check for intersections of cones and their images. Programming languages like python have many packages that can do this, so it is trivial to check for cone intersections using these point descriptions. The final output of our code at variables $L1$ and $L2$ are the vertices that define the returning cones on the starting wall for cycles A and B in our example.

```
# Python numerical package
import numpy as np
# Python graphics package good for making plots and graphics
from matplotlib import pyplot as plt
# Python package for calculating convex hulls
from scipy.spatial import ConvexHull

# Matrix of boxes in binary
boxMAT = np.array([[0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],
             [0,1,0,0],[0,1,0,1],[0,1,1,0],[0,1,1,1],
             [1,0,0,0],[1,0,0,1],[1,0,1,0],[1,0,1,1],
             [1,1,0,0],[1,1,0,1],[1,1,1,0],[1,1,1,1]])

# Key word used for when an alternative exit is not possible.
NA="NA"

def B(fp, j):

# Matrix from equation (2.10), same as above.

    I = np.diag(np.ones([len(fp)]))
    e = np.zeros([len(fp)])
    e[j] = 1

    return I-(1/fp[j])*np.outer(fp, e)

def psi(fp, j):

# Vector from equation (2.10), same as above.

    e = np.zeros([len(fp)])
```

```
    e[j] = 1

    return (-1/fp[j])*e

def M(fp, j, y):

# Wall-to-wall map from equation (2.9), same as Y_kPlus1 from above

    return (1/(1+psi(fp, j).dot(y)))*B(fp, j).dot(y)

def B_k(cycle, fpmat):

# This function takes in a list of box numbers that forms a cycle and
    a focal point matrix and returns a list of compositions of
   matrices from the B function. The list returned is a list of B
   matrices associated with a traversal of the cycle necessary for
   calculating the rows for the matrix in equation (2.15).

    Bl =[]
    j = 0
    for i in range(len(cycle)-1):
        if i < len(cycle):
            c = boxMAT[cycle[i]]-boxMAT[cycle[i+1]]
        for k in range(len(c)):
            if abs(c[k]) == 1:
                j = k

        Bl.append(B(fpmat[cycle[i]],j))
    c = boxMAT[cycle[-1]]-boxMAT[cycle[0]]
    for k in range(len(c)):
            if abs(c[k]) == 1:
                j = k

    Bl.append(B(fpmat[cycle[-1]],j))

    return np.array(Bl)

def R(cycle, fpmat, altext):

# This function takes in a list of box numbers as a cycle, a focal
   point matrix, and list of alternative exits for the cycle and will
    return the matrix R from equation (2.15).

    B_k_list=B_k(cycle, fpmat)
    B_k0_list=[]
    for i in range(len(cycle)):
        B_k0_list.append(np.diag(np.ones([len(fpmat[0])])))
        for j in range(i+1):
```

```
            B_k0_list[i]=B_k_list[j].dot(B_k0_list[i])

    lsd=[]

    for i in range(len(cycle)):

        if altext[i] == ["NA"]:
            continue

        else:
            for j in altext[i]:
                e=np.zeros(len(fpmat[0]))
                e[j]=1
                lsd.append((-1/fpmat[cycle[i]][j])*e.dot(B_k0_list[i
   ]))

    return np.array(lsd)

def trunc(val):

# This is a truncating function that will help when python returns a
   very small number instead of zero.

    return int(val*1e14)/1e14

def arraytrunc(arr):

# This is an array version of the truncation function above.

    for i in range(len(arr)):
        arr[i]=trunc(arr[i])

    return arr

def pln_intr(Ri, Rj, plane):

# this creates a matrix with row i and j of the R matrix and the
   plane then finds if there is a point in common between the 3

    A = np.array([Ri,Rj,plane])

    b = np.array([0,0,1])

    if np.linalg.det(A) == 0:
        return np.array([])

    else:
        return np.linalg.solve(A,b)
```

```
def ret_vert(R, plane):

# This function takes in a returning cone matrix, R, and a plane and
   finds the vertices of the returning cone intersected with the
   plane.

    inter, vert = [], []

# these are the axes

    Axes = [[plane[0],0,0],[0,plane[1],0],[0,0,plane[2]]]

#This checks for rows that are different by only a sign. In this case
    the function returns an empty list as the Returning cone is empty
   .
    for i in R:
        for j in R:
            if all(i[k] == -j[k] for k in range(len(i))):
                return []

#This checks for duplicate rows and then removes the duplicates

    Temp_R = [R[0]]
    for i in R:
        if all(r != i for r in Temp_R):
            Temp_R.append(i)

    R = Temp_R

    for r in R:
        for a in Axes:
            if r == a:
                continue
            else:
                x = pln_intr(r, a, plane)
                if len(x) == 0:
                    continue
                else:
                    inter.append(x)

    for Ri in R:
        for Rj in R:

            if Ri == Rj:
                continue
            else:
                x = pln_intr(Ri, Rj, plane)
```

```
                if len(x) == 0:
                    continue
                else:
                    inter.append(x)

# Need to generalize this portion of the code to deal with any octant
   . Right now it currently is set to deal with (+++)
    for i in inter:
        if all(np.array(r).dot(i) >= 0 for r in R):
            i = arraytrunc(i)
            if any(ell < 0 for ell in i):
                continue
            else:
                vert.append([trunc(i[0]),trunc(i[1]),trunc(i[2])])

    return vert

def rem_reps(R):
    Temp_R = [R[0]]
    for i in R:
        if all(r != i for r in Temp_R):
            Temp_R.append(i)

    R = Temp_R

    return R

def main():

# Here we define a matrix of focal points, two cycles, and the list
   of alternative exits for each cycle and then calculate the
   returning cones for each cycle.

    fpMAT = np.array
   ([[-1,1,-1,-1],[1,-1,-1,-1],[-1,-1,-1,1],[-1,1,-1,1],
                    [-1,1,1,1],[1,-1,1,1],[1,-1,1,1],[1,1,1,1],

   [-1,-1,-1,-1],[1,1,1,-1],[-1,-1,-1,-1],[-1,-1,1,-1],
                    [-1,-1,-1,1],[1,1,1,1],[1,-1,-1,-1],[1,-1,1,-1]])

    Cycle_A = np.array([14,10,2,0,4,6,7,15])
    Cycle_B = np.array([14,10,2,3,1,0,4,5,7,15])

    alt_ext_A = [[2],[2],[3],["NA"],[3],[0,1],["NA"],[1]]
    alt_ext_B = [[2],[2],[2],[1],[0],["NA"],[2],[0,1],["NA"],[1]]

    R_A = R(Cycle_A,fpMAT,alt_ext_A)
    R_B = R(Cycle_B,fpMAT,alt_ext_B)
```

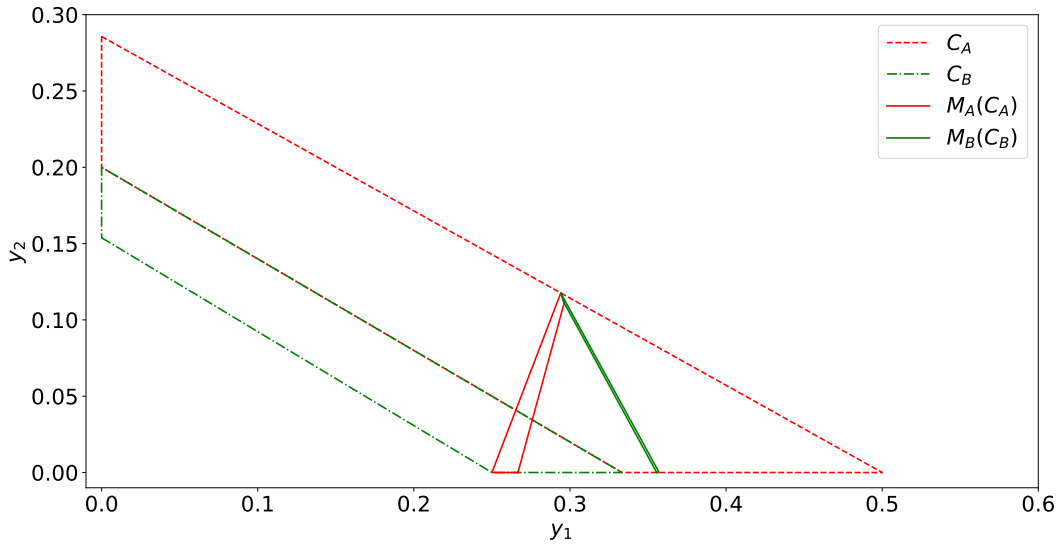


Figure 3.3: Returning cones and their respective images forming a trapping region for example Glass network.

```
    L1 = ret_vert(R_A[:,0:3].tolist(), [1,1,1])
    L2 = ret_vert(R_B[:,0:3].tolist(), [1,1,1])

    L1 = rem_reps(L1)
    L2 = rem_reps(L2)

if __name__ == '__main__':
    main()
```

After the above steps we should be left with a tree that only contains cycles allowed by the TG, that have non-empty returning regions, all of which map into the union of themselves with no individual region not being mapped into. The union of these returning cones is therefore a trapping region. An example of what the tree and TG look like after this step can bee seen in Figures 3.4 and 3.5 respectively. From the tree we can easily form the TG and $\mathrm{TG}_r(1)$. At this level of the algorithm, and all of the other levels going forward, since all of the data required to prune the tree is stored in the leaf nodes, we will mainly focus on the leaf nodes. The removal of leaf nodes requires knowledge of returning regions, cycle mappings, and their images. However, once the leaf nodes are removed, removing the rest of the branch is simple by using pruning step 1.

Now that the tree has been pruned to include only the trapping region, we can continue growing the tree by considering concatenations of these cycles as these cycle will be the only cycles possible going forward.

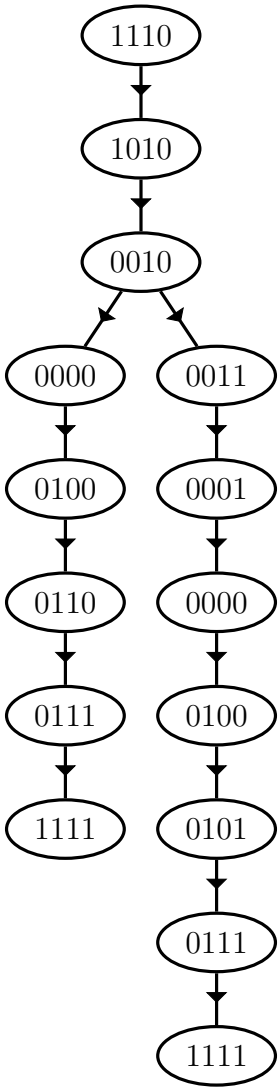


Figure 3.4: Tree structure for the example after returning cone and trapping region pruning. What is left is just a tree that contains branches that are associated with cycles A and B (left and right respectively). In each terminal node, the data for the returning region and cycle map are stored for future use. Note that adding connecting edges from the terminal nodes back to the start node and performing out-splittings at 0010 and 1010 produces the graph that results after all the out-splittings on the TG_r in chapter 2. If in-splittings are then performed as described in chapter 2, then the resulting structure is the $\mathrm{TG}_r(1)$.

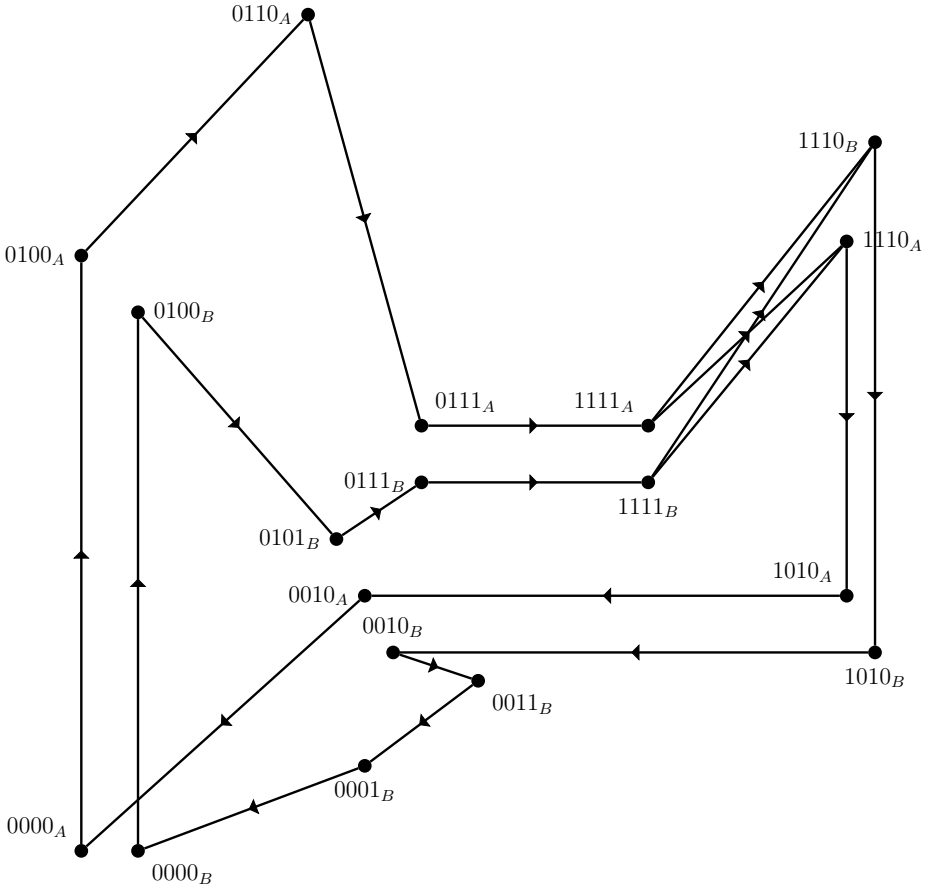


Figure 3.5: TG representation of the tree in Fig 3.4

3.1.4 Cycle concatenations and Pruning

Now the tree only contains branches with leaf nodes associated with the cycles making up the trapping region. The next step is to grow the tree to account for concatenations of cycles and deduce which longer sequences might be forbidden in the trapping region. To each leaf node, which corresponds to a cycle, we add a copy of every cycle in the trapping region and then at each resulting leaf node, calculate the returning cone for the entire concatenation. An example of this is shown in Figure 3.6. Then we remove the empty ones, and repeat until the algorithm halts.

We no longer use the repetition number from the previous step, which was only needed to keep the number of first-return cycles finite. Instead, we use sequence position to halt tree growth. To implement this in the algorithm, and thus consider arbitrarily long concatenations, at this stage we set all node repetition numbers to zero and do not change them again. The new quantity, sequence position, is used to keep track of how many concatenations we consider and hence the level of refinement for the entropy estimate. This is our halting condition: when the leaf nodes reach a sequence position equal to some maximum preset by the user, the algorithm halts. Leaf node sequence position is given by the number of times a node with the same name as the leaf node appears in a path back to the root.

At each concatenation stage, we calculate the returning regions for each resulting path from root to leaf, since these represent all possible concatenations in the trapping region, and add this data to each leaf node. Additionally, we store a representation of the concatenated cycles in each leaf node as they will be used later on to construct the adjacency matrix for the $\mathrm{TG}_r(n)$ at the level when tree growth halts. For our example this means calculating returning cones for AA, AB, BA, and BB. This uses the same code as for the first-return cycles. Below is an implementation. $L1$ through $L4$ are the sets of vertices defining the returning cones for AA, AB, BA, and BB respectively. Note that $L4$ is empty since BB is a forbidden cycle.

```
def main():

# Here we define a focal point matrix for the network.
    fpMAT = np.array
   ([[-1,1,-1,-1],[1,-1,-1,-1],[-1,-1,-1,1],[-1,1,-1,1],
                    [-1,1,1,1],[1,-1,1,1],[1,-1,1,1],[1,1,1,1],

   [-1,-1,-1,-1],[1,1,1,-1],[-1,-1,-1,-1],[-1,-1,1,-1],
                    [-1,-1,-1,1],[1,1,1,1],[1,-1,-1,-1],[1,-1,1,-1]])

# Here define box lists for the cycle concatenations AA, AB, BA, and
   BB.
    Cycle_AA = np.array([14,10,2,0,4,6,7,15,14,10,2,0,4,6,7,15])
    Cycle_AB = np.array([14,10,2,0,4,6,7,15,14,10,2,3,1,0,4,5,7,15])
    Cycle_BA = np.array([14,10,2,3,1,0,4,5,7,15,14,10,2,0,4,6,7,15])
```

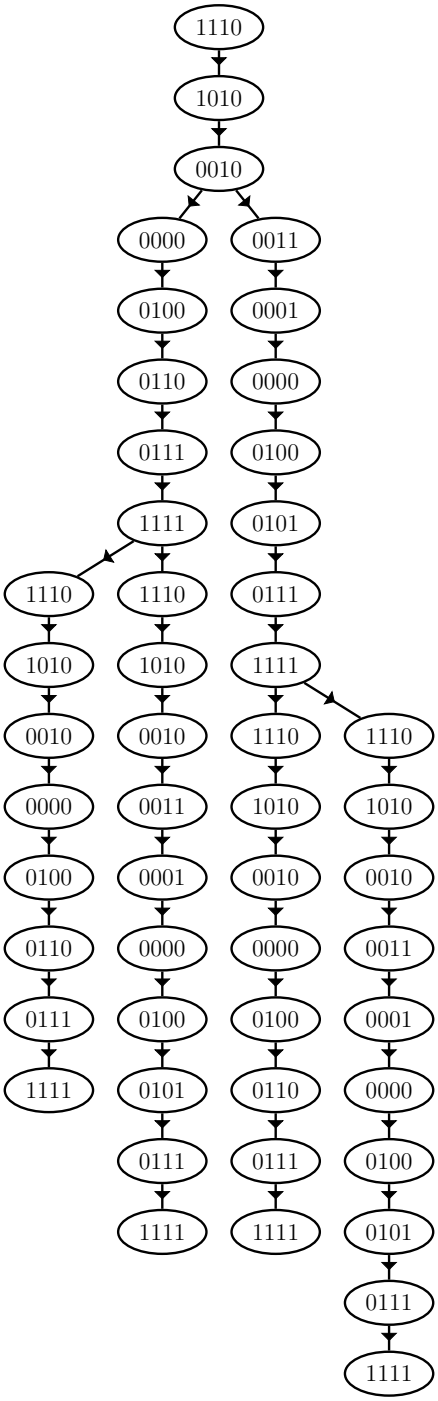


Figure 3.6: Second level tree before pruning.

```
    Cycle_BB = np.array
   ([14,10,2,3,1,0,4,5,7,15,14,10,2,3,1,0,4,5,7,15])

# Here we define lists of alternative exits for cycle concatenations
   AA, AB, BA, and BB.
    alt_ext_AA = [[2],[2],[3],["NA"],[3],[0,1],["NA"
   ],[1],[2],[2],[3],["NA"],[3],[0,1],["NA"],[1]]
    alt_ext_AB = [[2],[2],[3],["NA"],[3],[0,1],["NA"
   ],[1],[2],[2],[2],[1],[0],["NA"],[2],[0,1],["NA"],[1]]
    alt_ext_BA = [[2],[2],[2],[1],[0],["NA"],[2],[0,1],["NA"
   ],[1],[2],[2],[3],["NA"],[3],[0,1],["NA"],[1]]
    alt_ext_BB = [[2],[2],[2],[1],[0],["NA"],[2],[0,1],["NA"
   ],[1],[2],[2],[2],[1],[0],["NA"],[2],[0,1],["NA"],[1]]

# Here we calculate the returning cones for cycle concatenations AA,
   AB, BA, and BB.
    R_AA = R(Cycle_AA, fpMAT, alt_ext_AA)
    R_AB = R(Cycle_AB, fpMAT, alt_ext_AB)
    R_BA = R(Cycle_BA, fpMAT, alt_ext_BA)
    R_BB = R(Cycle_BB, fpMAT, alt_ext_BB)

    L1 = ret_vert(R_AA[:,0:3].tolist(), [1,1,1])
    L2 = ret_vert(R_AB[:,0:3].tolist(), [1,1,1])
    L3 = ret_vert(R_BA[:,0:3].tolist(), [1,1,1])
    L4 = ret_vert(R_BA[:,0:3].tolist(), [1,1,1])

    L1 = rem_reps(L1)
    L2 = rem_reps(L2)
    L3 = rem_reps(L3)
    L4= rem_reps(L4)

if __name__ == '__main__':
    main()
```

At each concatenation step, we remove all the leaf nodes that have empty returning regions, and then re-apply pruning step 1. This again will systematically remove the leaf nodes that are not terminal until a branching point is reached. Again, this function halts when only leaf nodes are terminal.

After the first concatenation step in our example, what we are left with a tree structure that contains branches that are the allowed sequences of two cycles within the trapping region. See Figures 3.8 and 3.9. Since BB is the only concatenation that has an empty returning cone, it is the only sequence that has been removed from the example at this point. Figure 3.7 shows a projection of the starting wall and the three remaining returning cones.

At each additional concatenation step, we again add to each leaf node a copy of each allowed cycle (single cycles), calculate the returning regions, and then remove the

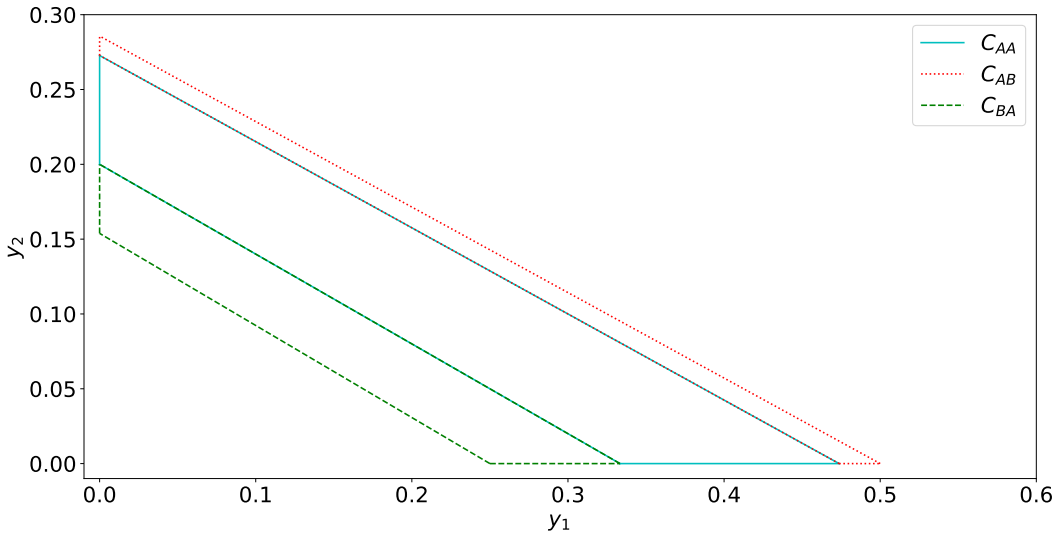


Figure 3.7: Length two returning cones. BB is empty and hence a forbidden sequence. This is a projection of the starting wall on the plane $y_1 + y_2 + y_3 = 1$. Only y_1 and y_2 are shown.

leaves/branches with empty cones. The process can now be repeated arbitrarily to sequences of any length. With the tree constructed and the algorithm halted, the only thing left is to extract a value for entropy.

3.1.5 Extracting entropy

The final component of the algorithm, after tree growth has halted, is extracting a value of entropy for the network. To do this, we need to construct the $\mathrm{TG}_r(n)$ adjacency matrix. This will require knowledge of which transitions (i.e edges) are allowed at that level. The edges contained within an individual cycle are clearly included and are easy to find from the tree. However, the cross edges that complete cycles by joining terminal nodes of cycles to start nodes of some others are not so trivial. Luckily, the construction of the refinement process has a very simple property baked in that makes this not too tricky.

Since the inner edges of a single cycle are just a chain, each node adjacent to the next, the submatrix corresponding to this cycle simply has 1's on the super-diagonal. See the example below. Note that not all cycles have an edge between their terminal node and starting node. Cycles that have a connecting edge between their terminal and starting node are referred to here as complete cycles. For example, after the first stage of tree building (before concatenation of first-return cycles), for our example BB is forbidden so there is no edge connecting B's terminal node to its starting node. However,

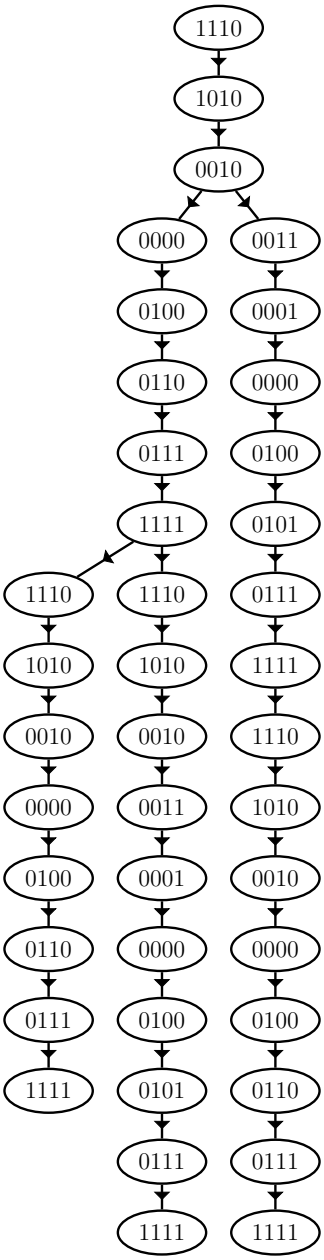


Figure 3.8: Second level tree after pruning. BB was the only concatenation that had an empty returning cone, hence, its branch was removed from the tree.

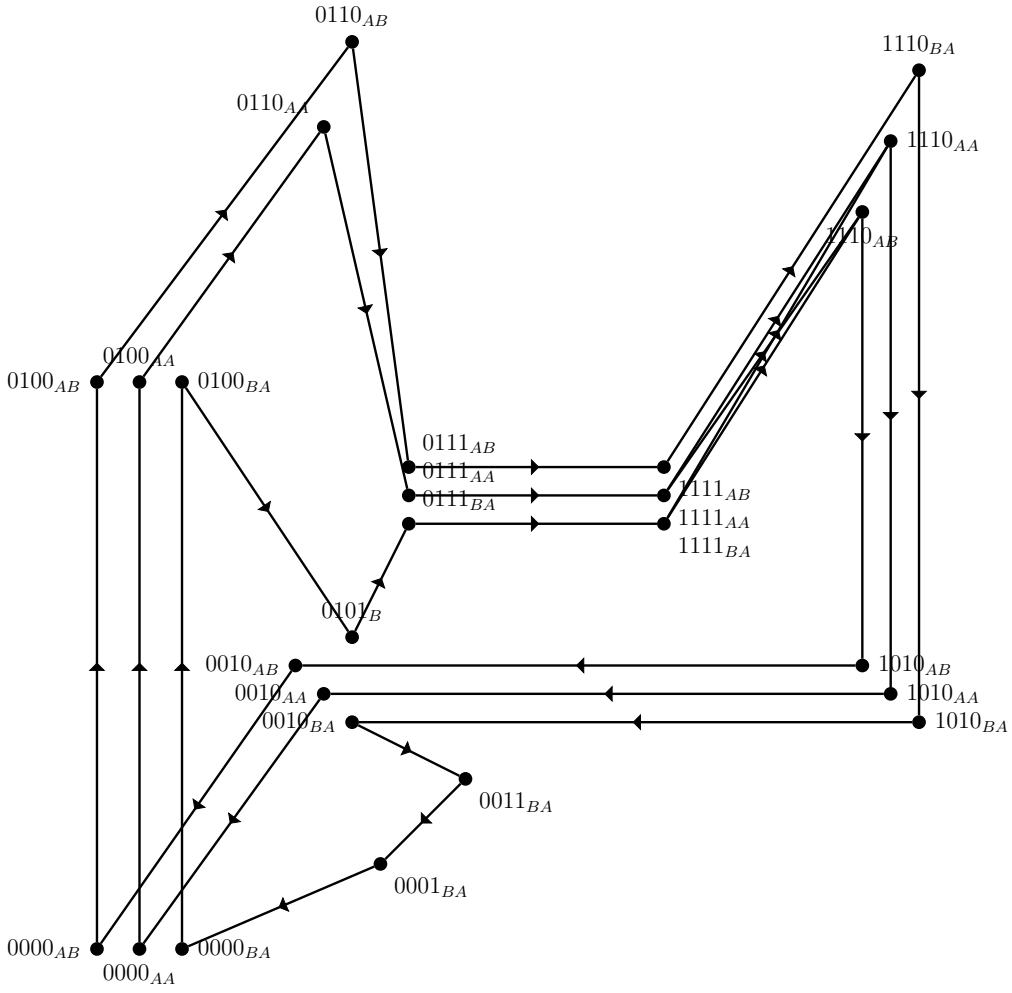


Figure 3.9: TG for tree in Fig 3.8

AB is a complete cycle. The cross edges (edges connecting terminal nodes to starting nodes) can be found using the branches of the tree. For each branch, retrieve the root-to-leaf path and remove the first-run cycle. Then take all cycle concatenations on the tree and remove the last cycle from all of them. Checking to see which concatenations with first-run cycle removed match the concatenations with their final cycles removed, the ones that match represent possible cross edges. This gives a simple method of finding the cross edges on the level of refinements TG from the tree. Using these two ideas, we will be able to decompose the adjacency matrix into sub-matrices associated with each cycle; these will give the off-diagonals. Then we will define sub-matrices for cross edges.

To do all this, we will first need to retrieve all paths starting from the root of the tree to each of the leaf nodes. For each sequence retrieved, the first cycle present in the sequence represents a path from the starting wall back to itself on the TG_r for that level. So in order to construct the adjacency matrix for the level of refinement, we will need to have a position within the matrix for each node in these initial cycles. Let N be the number of leaf nodes once tree growth has halted. Order the cycles as $C_1, C_2, \ldots, C_N$. Then within each cycle, if we say $C_i = (c_{i_1}, c_{i_2}, \ldots, c_{i_{len(C_i)}})$ we can order our adjacency matrix as follows:

$$A = \begin{array}{c} \begin{matrix} c_{1_1} & \cdots & c_{1_{len(C_1)}} & \cdots & c_{N_1} & \cdots & c_{N_{len(C_N)}} \end{matrix} \\ \begin{pmatrix} & & & & & & \\ & C_{11} & & \cdots & & C_{1N} & \\ & & \ddots & & & & \\ & \vdots & & C_{ii} & & \vdots & \\ & & & & \ddots & & \\ & C_{N1} & & \cdots & & C_{NN} & \\ & & & & & & \end{pmatrix} \end{array} \begin{matrix} c_{1_1} \\ \vdots \\ c_{1_{len(C_1)}} \\ \vdots \\ c_{N_1} \\ \vdots \\ c_{N_{len(C_N)}} \end{matrix}$$

Each C_{ij} is a sub matrix of the adjacency matrix, where, if $i = j$ then

$$C_{ii} = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ a & 0 & 0 & \ldots & 0 \end{pmatrix}$$

with $a = 0$ if the cycle cannot be followed by itself and $a = 1$ if it can. If $i \neq j$ then

$$C_{ij} = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \\ a & 0 & 0 & \dots & 0 \end{pmatrix}$$

where $a = 1$ if C_i can follow cycle C_j and $a = 0$ if it cannot. The value of a in both of these matrices is determined by the last $n-1$ symbols in the current cycle's subscript and the first $n-1$ symbols in the next cycle's subscript. Below we have provided some code that does this for our example from a list of allowed cycle sequences of length two extracted from the tree. Here we extracted, from the tree, a list of which cycle sequences of length two are allowed and then knowing the length of each cycle, we constructed the adjacency matrix from the list.

```
# Python library useful for numerical calculations.
import numpy as np
# Python library that contains mathematical functions and operations.
import math

def main():

# Here we have a list of the allowed cycle concatenations of length
   two for the example.
    NOTEMPTY = ['AA','AB','BA']

# Here we create a variable that will be used to calculate the
   necessary dimension of the adjacency matrix for the entropy
   estimate.
    MAT_DIM = 0

    for i in range(len(NOTEMPTY)):
        if NOTEMPTY[i][0] == 'A':
            MAT_DIM += 8
            continue
        if NOTEMPTY[i][0] == 'B':
            MAT_DIM += 10

# Here we define a matrix that we will use build the adjacency matrix
   .
    A = np.zeros([MAT_DIM,MAT_DIM])

# Here we use the cycle concatenations to find which nodes are
   connected by edges around cycle concatenation loops and add ones
   to the adjacency matrix accordingly.
```

```
    DIAG = 0

    for i in range(len(NOTEMPTY)):

        if NOTEMPTY[i][0] == 'A':
            A_i = np.zeros([8,8])

            for j in range(7):
                A_i[j,j+1] = 1

            A[DIAG:DIAG+8,DIAG:DIAG+8] = A_i

            DIAG += 8


        if NOTEMPTY[i][0] == 'B':
            A_i = np.zeros([10,10])

            for j in range(9):
                A_i[j,j+1] = 1

            A[DIAG:DIAG+10,DIAG:DIAG+10] = A_i

            DIAG += 10

# Here we use the allowed concatentations to deduce which cross edges
    are allowed and add ones to the adjacency matrix accordingly.
    row = -1
    for i in range(len(NOTEMPTY)):

        if NOTEMPTY[i][0] == 'A':
            row += 8

        if NOTEMPTY[i][0] == 'B':
            row += 10
        col = 0
        for j in range(len(NOTEMPTY)):

            if NOTEMPTY[i][1:] == NOTEMPTY[j][:-1]:

                A[row,col] = 1
            if NOTEMPTY[j][0] == 'A':
                col += 8

            if NOTEMPTY[j][0] == 'B':
                col += 10

# Here we calculate the eigen values and eigen vectors for the
```

```
    adjacency matrix A.
     v, w = np.linalg.eig(A)

# Here we print the log base 2 of the largest eigen value for the
    adjacency matrix A.
     print(math.log2(abs(max(v))))

if __name__ == '__main__':
     main()
```

Finally, we calculate the base-2 logarithm of the Perron eigenvalue for the matrix, which is the entropy estimate for that level of refinement.

Although the adjacency matrix becomes very large as we further refine, the adjacency matrices are quite sparce and the numerical calculation of this eigenvalue is quite accurate for very large matrices. Thus, for a given network, we can specify what level of refinement we wish to use to estimate entropy and using the above, we can accurately calculate an estimate of the network's entropy that is accurate up to the level of refinement. Finally, due to the convergence result of Theorem 2.16, we have effectively developed a methodology of approximating entropy for Glass networks.

Of course, we only obtain upper bounds, so you never really know if the entropy is nonzero. However, with the ability to specify an arbitrarily large level of refinement, we have effectively found an algorithm that can compute upper bounds arbitrarily close to the true value. This algorithm can therefore be used to deduce if a given network is predictable (periodic, quasi-periodic, etc) or irregular (chaotic and beyond!).

3.1.6 Multiple disjoint trapping regions

Up to this point we have avoided discussing the possible presence of multiple disjoint trapping regions. This is not a particularly difficult situation to deal with. However, if one does not account for this, it will affect the quality of the estimate that the algorithm can produce by combining the entropy of two separate attractors into a single number.

Once we identify disjoint trapping regions, we suggest splitting the tree, to creating multiple sub-trees in which the only branches are those pertaining to the particular sub-tree's trapping region.

To identify disjoint trapping regions at a given level of refinement, at each leaf node, take the image of the returning region for the cycle map associated with the leaf node, and look for intersections with the returning region of every other leaf node. Using the non-empty intersections we can then simply determine which returning regions form the disjoint trapping regions within the greater trapping region. Then we can organize the returning regions into disjoint lists to categorize which cycles form which trapping region. Each list will represent a disjoint trapping region.

Now use each disjoint list to create a new tree, one for each list and then apply the

algorithm to each tree independently in accordance to the above description. Continue to grow all of the trees until the level of refinement is to satisfaction. The entropy calculated from each tree will be associated with the region of phase space occupied by trajectories through the set of returning cones covered by that tree. Entropy is thus not that of all space, but rather of the attractor.

3.2 Putting it all together

Now we put the individual pieces together into an executable algorithm that can approximate entropy in Glass networks. We will also include a pseudo-code description of the algorithm to make it as clear as possible. Given a Glass network we wish to investigate, the algorithm is as follows.

Step 1. To begin the algorithm, we define the network's focal point matrix and set a value for R_{Max}. Then using The_Wall function (a function designed to find the wall most often visited during numerical simulation) we identify a starting wall along with the starting and terminal nodes.

```
# This defines a network by its focal points matrix
GNET = fpMAT
# This defines a maximum number of times that a node can appear
  in a cycle.
R_Max = K
# This is a simulation that will be used to determine the
 starting wall.
sim = GNETTraj

# Here we use the simulation to find the starting wall, the
 starting node, and the terminal node.
sWall = The_Wall(sim)
sNode = box(sWall)[0]
tNode = box(sWall)[1]
```

The function, box, is a simple function that finds the boxes that sandwich a wall. This step of the algorithm identifies the starting and terminal nodes. In the above section of code, GNETTraj represents a trajectory from a numerical simulation of the network. We do not describe how to numerically simulate networks here since this is a simple matter with the theory discussed in previous Chapters.

Step 2. Placing sNode at the root of the tree, we grow the tree until each leaf node's repetition number is equal to R_Max or has a name matching tNode.

```
# Here we define a level number and level list.
Level = 0
```

```
Lvllst = []
# Here we start building the tree by defining the starting node
  in the tree data type and place the starting node at the root
  of the tree.
Tree = GT.Node(sNode, [0,0], None)
Lvllst.append(Tree)
# Here we start building the tree by systematically adding
 nodes and use the level number and level list to keep track.
 We use a while loop to check to see if the level list is empty
 , if it is we stop building as there is nothing left to be
 added.
while Lvllst != []:
# if the level list is not empty we increment the level
  Level = Level + 1
# we define the next level
  Nxtlvl = []
# Then we check each node in the level list and if it meets the
  criteria to continue growing, i.e not at rep max or the
 terminal node, then we add its possible children to the next
 level
  for node in Lvllst:
    node.rep = GT.root_path_reps(node)
    if node.name == tNode:
      Lvllst.remove(node)
      continue
    if node.rep == repMAX:
      Lvllst.remove(node)
      continue
    else:
      Clst = ext(GNET, node.name)
      node.children = [GT.Node(Clst[i], [Level,len(Nxtlvl)+i],
 node) for i in range(len(Clst))]
      Nxtlvl.append(node.children)
      Lvllst.remove(node)
      continue
# Once the above finishes, then we make the level list the next
  level list and continue. The loop stops when there are no
 more possible nodes to be added to the tree.
  Lvllst = Nxtlvl
  continue
# Once the tree is done growing for the moment we then remove
 all branches that do not end with a leaf node matching the
 terminal node.
leafnodelst = GT.leaf_nodes(Tree)
while any(l.name != tNode for l in leafnodelst):
  for l in leafnodelst:
    if l.name == tNode:
      continue
```

```
    else:
      Tree.remove(l)
```

This step grows the initial part of the tree producing a tree with every complete first-run cycle that passes through the starting wall that does not exceed R_{Max}.

Step 3. Now, in the next two steps we refine the tree to include just a trapping region. First we find all the returning cones and remove cycle branches of those that are empty.

```
# Here we define a cone function to calculate all the necessary
  data for returning cone calculations.
def CONE(fp, swall, cyc, alt):
  plane = []
  for i in range(len(swall)):
    if swall[i] == 0:
      j = i
    else:
      if swall[i] == "+":
        plane.append(1)
      if swall[i] == "-":
        plane.append(-1)
  r = R(cyc, fp, alt)
  L = ret_vert(np.delete(r, j, 1).tolist(), plane)
  return L
# Here we check each cycles returning cones and if they are
 empty we remove the leaf node. If they are not empty then we
 add the cone data and the cone image data to the node.
for leaf in GT.leaf_nodes(Tree):
  cyc = Tree.path_from_root(leaf)
  alt = Tree.alt_ext(cyc)
  cone = CONE(fpMAT, sWall, cyc, alt)
  if cone == []:
    Tree.remove(leaf)
    continue
  else:
    leaf.cone_data = cone
    leaf.cone_image = cyc_map(cyc, cone)
    continue

# Here we remove all branches with leaf nodes not matching the
 terminal node.
leafnodelst = GT.leaf_nodes(Tree)
while any(l.name != tNode for l in leafnodelst):
  for l in leafnodelst:
    if l.name == tNode:
      continue
```

```
    else:
      Tree.remove(l)
```

Step 4. Now we locate a trapping region. For brevity, we will assume that there is only one trapping region.

```
# Here we retrieve each leaf node and use the returning cone
 data to find a trapping region. For the branches not
 contributing to the trapping region we remove the leaf nodes.
leafnodelst = GT.leaf_nodes(Tree)
for l in leafnodelst:
  templst = leafnodelst.remove(l)
  intersectionlst = []
  for L in templst:
    intersectionlst.append(l.cone_data,L.cone_image)
  if intersectionlst == []:
    Tree.remove(l)
# Here we remove all branches with leaf nodes not matching the
 terminal node.
leafnodelst = GT.leaf_nodes(Tree)
while any(l.name != tNode for l in leafnodelst):
  for l in leafnodelst:
    if l.name == tNode:
      continue
    else:
      Tree.remove(l)
```

At this point, the only cycles left on the tree are those that make up the trapping region.

Step 5. We now can add entire branches to each leaf node, reflecting cycle concatenations, and prune according to which cones have empty interiors. This is also where we define the level of refinement.

```
# This defines the level of refinement.
stop = N
leafnodelst = GT.leaf_nodes(Tree)
cycles = []
# Using the leaf node list we add cycle copies to each leaf
 node
for l in leafnodelst:
  cycles.append(Tree.path_from_root(leaf))
count = 1
# Here we use a while loop to continue adding cycles to the
 branches and removing leaf nodes with empty returning cones.
while GT.leaf_nodes(Tree)[0].seq_pos < stop:
```

```
 for l in leafnodelst:
   l.children = cycles
   l.seqpos = count

 for leaf in GT.leaf_nodes(Tree):
   cyc = Tree.path_from_root(leaf)
   alt = Tree.alt_ext(cyc)
   cone = CONE(fpMAT, sWall, cyc, alt)
   if cone == []:
     Tree.remove(leaf)
     continue
   else:
     leaf.cone_data = cone
     leaf.cone_image = cyc_map(cyc, cone)
     continue

 leafnodelst = GT.leaf_nodes(Tree)
 for l in leafnodelst:
   templst = leafnodelst.remove(l)
   intersectionlst = []
   for L in templst:
     intersectionlst.append(l.cone_data,L.cone_image)
   if intersectionlst == []:
     Tree.remove(l)

 # Here we remove branches with leaf nodes that do not match
the terminal node
 leafnodelst = GT.leaf_nodes(Tree)
 while any(l.name != tNode for l in leafnodelst):
   for l in leafnodelst:
     if l.name == tNode:
       continue
     else:
       Tree.remove(l)
 count = count +1
```

Step 6. The final step in the algorithm is to extract entropy. To build the required adjacency matrix, for every first run cycle in each concatenation produced by the tree, we associate a C_{ii} matrix with each. This gives the off diagonals. The remaining values of a in both the C_{ii} matrices and C_{ij} matrices are then found by simple comparing every concatenation excluding the first-run cycle to every other concatenation minus the final cycle. If they match $a = 1$, otherwise $a = 0$. The entropy estimate is then given by the base-2 logarithm of the largest eigenvalue of the adjacency matrix.

The above description could be used to write computer code to automatically calculate entropy estimates for Glass networks. In the next section, we will demonstrate what can be expected of the algorithm when applied to network examples with varying levels and types of complexity in their dynamics.

3.3 Dynamical diversity in Glass networks

We now examine how this algorithm deals with different types of dynamical phenomena. Since the main goal of this algorithm is to identify complex behaviour, this will give a good demonstration of how it deals with true chaos and other types of behaviour that might be confused with chaos in a more naive investigation. We will demonstrate what to expect when investigating complicated limit cycles and of course chaotic dynamics. Complicated limit cycles are the most common form of dynamics that are mislabeled as chaos.

3.3.1 Chaotic example: Revisited

In Chapter 2 we used an example to demonstrate how the theory behind our refinement process works and also used that same example as a guide to build our algorithm in Chapter 3. The algorithm is a powerful tool to investigate this example, further than we did in Chapter 2. We will show how the trapping region changes as longer concatenations of the cycles are considered. Additionally, we will show how the entropy estimate changes as longer sequences of cycles are deduced as forbidden blocks.

We previously showed for the example network that is depicted in Figure 2.1 that for cycle sequences of length two, the sequences AA, AB, and BA were all allowed but the sequence BB had an empty returning cone and is hence a forbidden block for the network's symbolic representation. The $\mathrm{TG}_r(2)$ in that case has entropy approximately 0.081. When compared to our numerical estimate, which gave a non-rigorous estimate of 0.067, it was clear that there were longer sequences of forbidden blocks as yet unidentified. Looking at the numerical simulation directly we saw that $BAAB$ was likely a forbidden block. A TG structure that also forbade $BAAB$ produced an entropy estimate of 0.0706, which is much closer to the numerical bound. However, upon calculating the returning cone for the sequence $BAAB$, we found that it was not empty. This raised the question whether it was truly a forbidden block, or rather just an extremely rare block that did not occur very often. With the algorithm, we can now answer this question.

While the returning cone for $BAAB$ is not empty, when considering sequences of length six, we find that sequences of length six that contains $BAAB$ followed by allowed words are empty, effectively making $BAAB$ a perfect example of a transient word. Since there are no additional sequences of length six beyond the ones containing $BAAB$ that

n	$h_{TG_{sr}(n)}$
1	0.11159015148410427
2	0.08126787770643938
3	0.08126787770643727
4	0.08126787770643908
5	0.08126787770643756
6	0.07056594265838596
7	0.07056594265838352
8	0.07056594265838535
9	0.070565942658382
10	0.070565942658382
11	0.06958939293749207
12	0.06763370755426378
13	0.06763370755426225
14	0.06664450427760658
15	0.06664450427759833

Table 3.2: Table of entropy estimates for chaotic network example. Accurate to what is standard of pythons floating point arithmetic, approximately 10^{-13}. Hence, estimates matching in value up to the 12^{th} decimal point can be considered equivalent.

are forbidden, calculation of the entropy estimate that forbids those sequences of length six produces the same estimate as the TG structure just forbidding $BAAB$. So, we now know that 0.0706 is indeed an upper bound on the entropy of the true dynamic in that example. Returning cones for cycle sequence length up to length six are shown in Figure 3.10.

Refining the TG to remove cycle sequences up to length 12 gives an entropy upper bound for the system of 0.06763370755426378, which is getting very close to the value 0.0670258 that was achieved using the numerical simulations. Refining the TG to remove cycle sequences up to length 15 gives an entropy upper bound of 0.06664450427759833. This is actually lower than that of the numerical simulation, suggesting that the numerical simulation was not long enough to capture all forbidden blocks.

This new upper bound is drastically lower that the original upper bound obtained by Farcot [15]. Our numerical check was used as a "quick" way of seeing how accurate the upper bounds we had were. However, the numerical simulation that gave the entropy estimate of 0.0670258 took a fairly optimized program a few days to produce that value. Our algorithm produced a better bound using unoptimized code in less than an hour. Thus, not only is the upper bound produced by our algorithm rigorous and better than the estimate produced by the numerical simulation, but it produced that bound in a small fraction of the time. This suggests that our algorithm is an effective way of producing good entropy estimates for Glass networks.

Finally, as we remove longer and longer sequences of cycles from contention on the TG, the bound does decrease, but in such a way that strongly suggests convergence to a nonzero value. The original TG produced an upper bound of approximately 0.873, whereas the TG_r gave 0.224. This is quite a significant decrease, approximately 0.649. However, for the next few refinements this decrease starts to get smaller. The difference between the TG_r entropy estimate and the $\mathrm{TG}_r(1)$ is approximately 0.111, between the $\mathrm{TG}_r(1)$ and $\mathrm{TG}_r(2)$ is approximately 0.0297, between the $\mathrm{TG}_r(2)$ and $\mathrm{TG}_r(6)$ is approximately 0.0107, between the $\mathrm{TG}_r(6)$ and $\mathrm{TG}_r(12)$ is approximately 0.003, and finally the difference between the $\mathrm{TG}_r(12)$ and $\mathrm{TG}_r(15)$ is approximately 0.00099. This system has been examined before [10] and we know it has no stable fixed points, no stable limit cycles, has a trapping region, and is believed to be chaotic. Thus, in this example and other chaotic examples like it, our algorithm gives a practical and effective way of approximating the network's true entropy.

3.3.2 Complicated stable limit cycle: a 34 cycle

Now that we have seen a demonstration of how the refinement process handles chaotic networks, it will be a useful exercise to examine a complicated limit cycle. Since the goal of this theory is to be used in the design of intrinsically chaotic true random number generators, it is important to be able to distinguish between true chaos and complicated limit cycles.

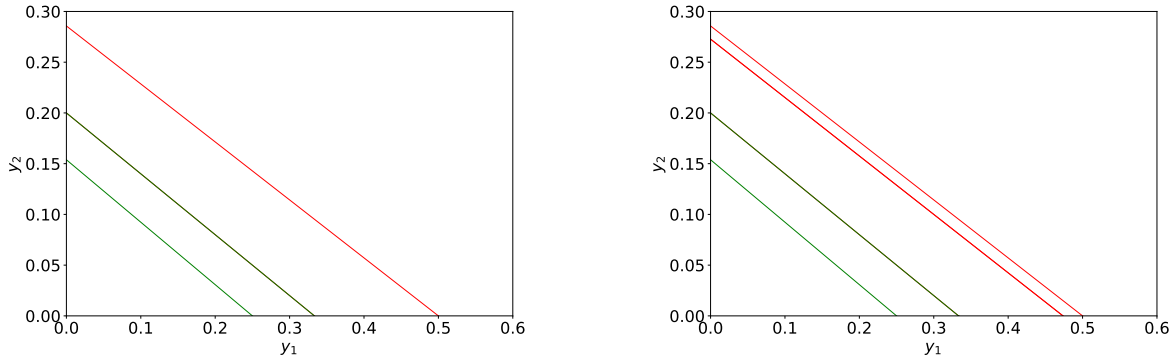
Consider the network defined as follows:

$$\begin{aligned}
\dot{y}_1 &= -y_1 + 2\left(\left(\bar{Y}_2 Y_3 + Y_2 \bar{Y}_3\right)\bar{Y}_4 + Y_2 Y_4\right) - 1 \\
\dot{y}_2 &= -y_2 + 2\left(\bar{Y}_1 \bar{Y}_3 + Y_1 \bar{Y}_4\right) - 1 \\
\dot{y}_3 &= -y_3 + 2\left(\left(\bar{Y}_1 Y_2 + Y_1\right) Y_4\right) - 1 \\
\dot{y}_4 &= -y_4 + 2\left(\bar{Y}_1\left(\bar{Y}_2 \bar{Y}_3 + Y_2 Y_3\right) + Y_1\left(\bar{Y}_2 Y_3 + Y_2 \bar{Y}_3\right)\right) - 1,
\end{aligned} \tag{3.3}$$

The focal points are given by Table 3.3

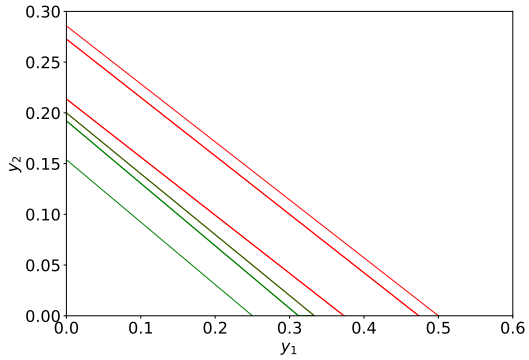
This gives rise to the TG structure in Figure 3.11

At a glance, this network seems like as good a candidate as any for a TRNG design. This is supported by numerical simulations with transient behaviour not removed. The transients in this network are quite long, so if a short numerical simulation is used, it is likely that no cyclic pattern will emerge. However, if we remove the transients from the trajectory (Figure 3.13) we can see that there is indeed something that looks like a stable limit cycle. However, there exist networks with even longer limit cycles. An example of length 174 was presented by Edwards [9]. In that case, and in even more complicated examples, removing transients may not reveal an obvious pattern, since in the case of the 174-cycle, short phase plane simulations on the attractor look just as "messy" as an actually irregular system. Our example 34-cycle network will highlight

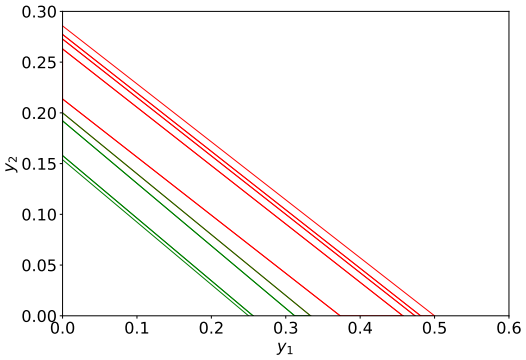


(a)

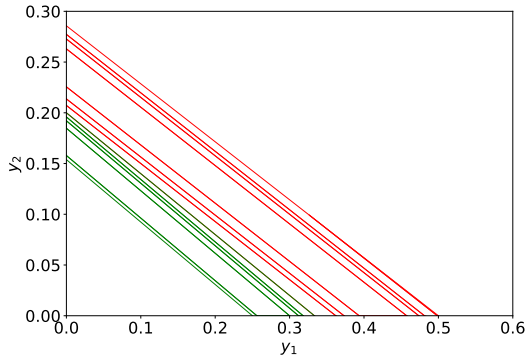
(b)



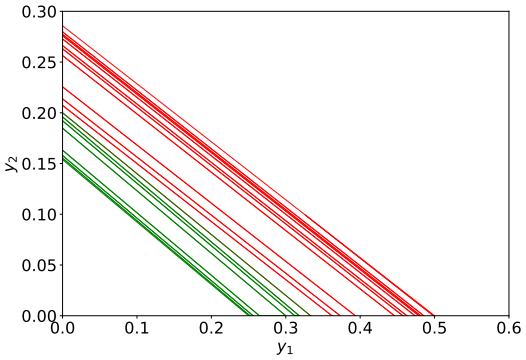
(c)



(d)



(e)



(f)

Figure 3.10: Returning cone projections for equation 3.1 to the plane $y_1 + y_2 + y_3 = 1$ for sequences of cycles of (a) length 1, (b) length 2, (c) length 3, (d) length 4, (e) length 5, and (f) length 6. Red regions indicating sequences starting with cycle A, green with cycle B.

Y_1	Y_2	Y_3	Y_4	F_1	F_2	F_3	F_4
0	0	0	0	-1	1	-1	1
0	0	0	1	-1	1	-1	1
0	0	1	0	1	-1	-1	-1
0	0	1	1	-1	-1	-1	-1
0	1	0	0	1	1	-1	-1
0	1	0	1	1	1	1	-1
0	1	1	0	-1	-1	-1	1
0	1	1	1	1	-1	1	1
1	0	0	0	-1	1	-1	-1
1	0	0	1	-1	-1	1	-1
1	0	1	0	1	1	-1	1
1	0	1	1	-1	-1	1	1
1	1	0	0	1	1	-1	1
1	1	0	1	1	-1	1	1
1	1	1	0	-1	1	-1	-1
1	1	1	1	1	-1	1	-1

Table 3.3: Table of focal point values for 34-cycle example.

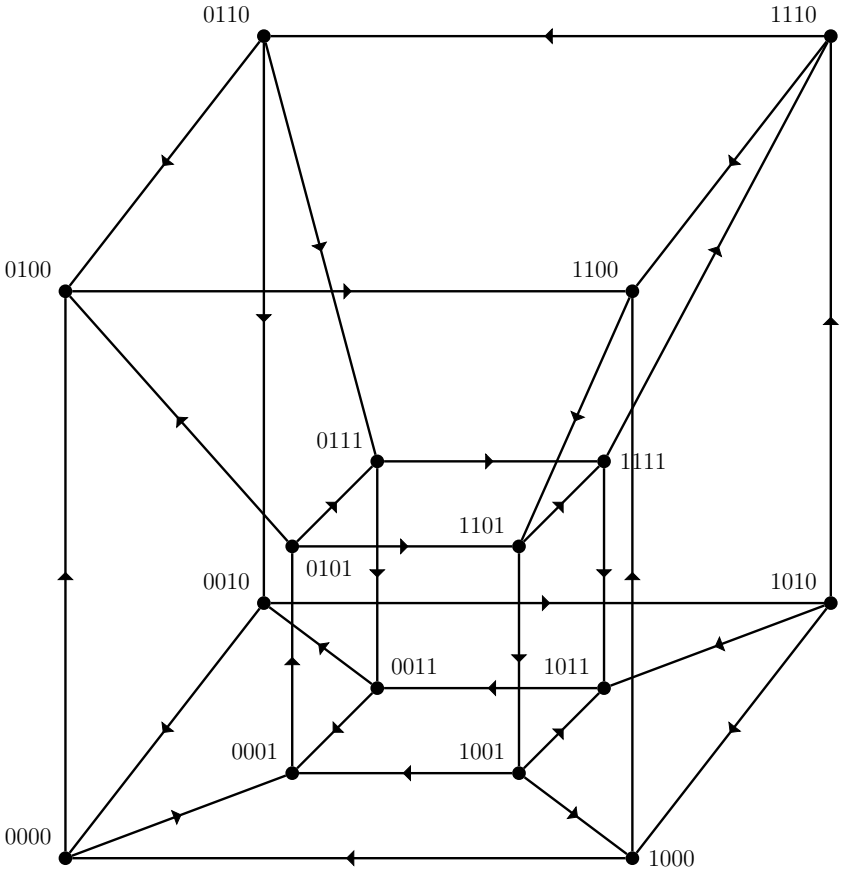


Figure 3.11: 34-cycle example TG

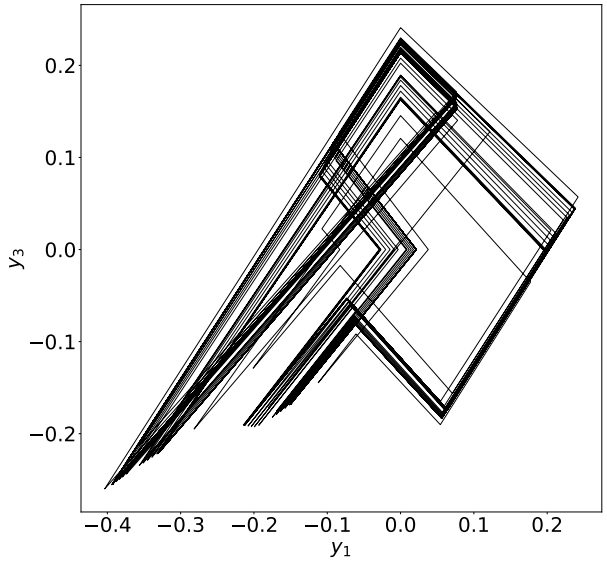


Figure 3.12: Stable limit cycle example phase plane simulation without removing transient behaviour.

a convenient property of our method that will be very useful in more complicated limit cycles.

Using numerical simulation to find the most visited wall in a sequence we find that the wall between boxes 0011 and 1011 is a good choice. Three cycles found on this wall are

$B : 1011 \to 0011 \to 0010 \to 0000 \to 0001 \to 0101 \to 0100 \to 1100 \to 1101 \to 1111 \to 1011$

$F : 1011 \to 0011 \to 0010 \to 1010 \to 1000 \to 0000 \to 0001 \to 0101 \to 0100 \to 1100 \to 1101$
$\to 1001 \to 1011$

$G : 1011 \to 0011 \to 0001 \to 0101 \to 0111 \to 1111 \to 1011$

Numerically, all trajectories eventually are confined to follow the sequence $GBGF$, and it would be reasonable to expect that the returning cones for cycles G, B, and F form a trapping region. However, this is not the case. When calculating their returning cones and respective images we find that while they do map into each other, there are regions of each that gets mapped outside of their union into the returning cones of other cycles. This is shown in Figure 3.14. The prominence of transient behaviour in this network is why this is not sufficient to find a trapping region. Hence, many different cycles need to be considered at refinement level two to find a trapping region. This does not give a very good upper bound and will likely, again, make it seem as if

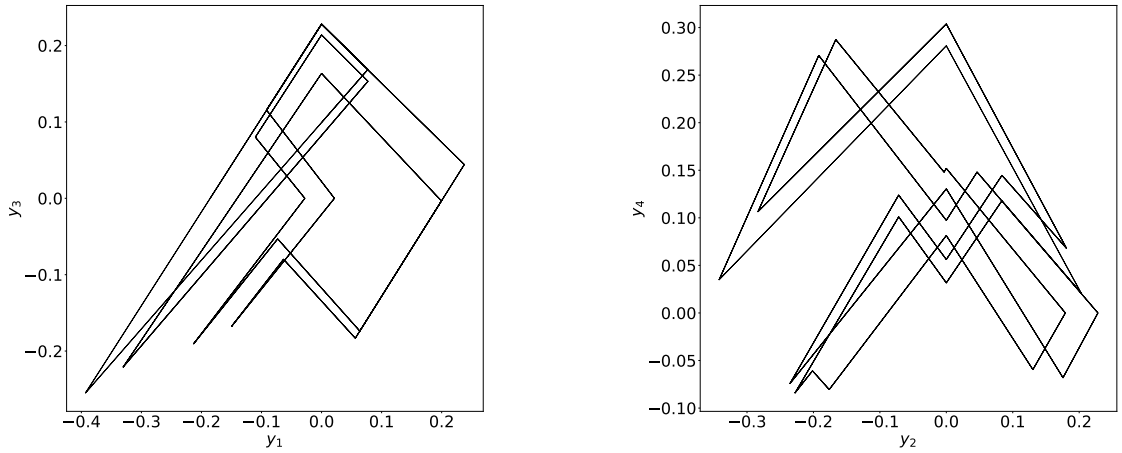


Figure 3.13: Stable limit cycle example phase plane simulation after removing transient behaviour.

this network is more irregular than it is. In fact, this reduction becomes very messy until much later in the refinement process than $\mathrm{TG}_r(1)$. Probing deeper will reveal more.

It would make sense that the returning cone for GBGF would form a minimal trapping region with only itself, however, we find that this is not the case. This is shown in Figure 3.15. Since the algorithm checks at each level for a minimal trapping region, we find what the numerical simulation suggested when we calculate the returning cone for $GBGFGBGF$ and its image and find that the cone is mapped directly into itself and no other. This forms the simplest trapping region, a trapping region of one cycle. As a result, at this point every other cycle sequence of length 8 is either empty or transient or part of a disjoint trapping region and thus can be removed. Then, since there is only one sequence that is in the trapping region the $\mathrm{TG}_r(8)$ is simply a ring where each node has a single edge entering and leaving. This is a special condition that the algorithm should check for. If there exists a returning cone that maps into itself and no other, it is periodic and thus has entropy zero.

While it is less straightforward than our chaotic example, it is clear that at a sufficient level of refinement we were able to deduce that the network contained a single returning cone that acts as a trapping region. Our method and algorithm are effective in finding entropy zero networks. As soon as the algorithm detects that there is a minimal trapping region made up of one cycle sequence, then there is a loop $\mathrm{TG}_r(n)$ that has entropy zero.

This method will always work regardless of how complicated the limit cycle is, since every limit cycle has a cycle map with a returning cone that solely acts as a trapping region. That trapping region will always give a loop $\mathrm{TG}_r(n)$ which has zero entropy.

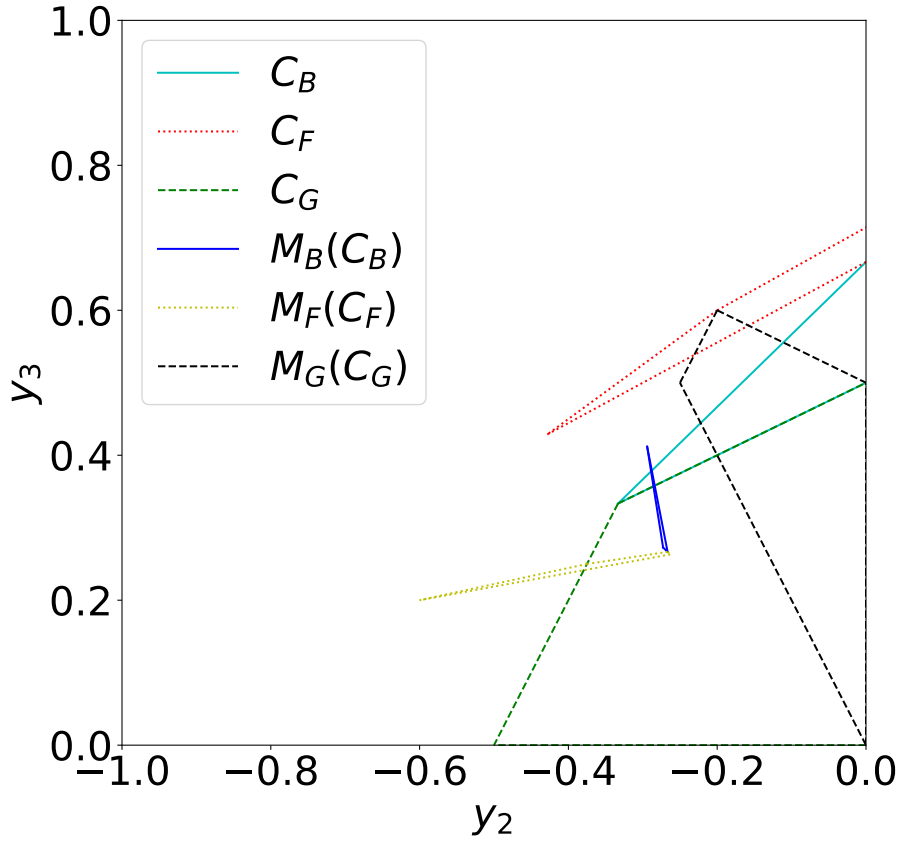


Figure 3.14: Cross sections of returning cones for cycles G, B, and F with the plane $-y_2 + y_3 + y_4 = 1$ projected onto the y_2, y_3-plane.

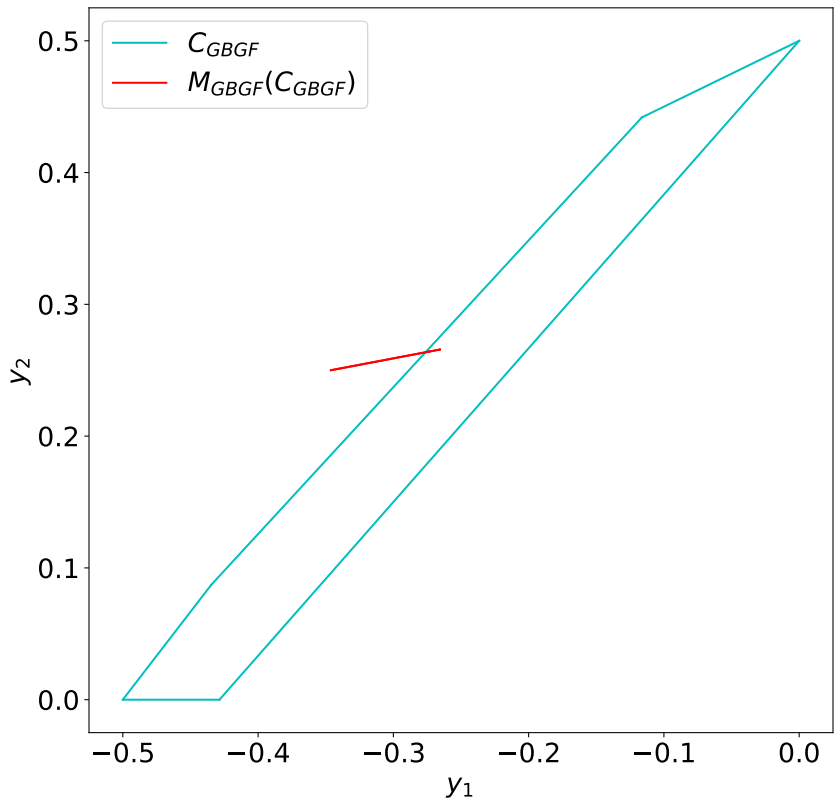


Figure 3.15: Cross sections of returning cones for the sequence of cycles $GBGF$ with the plane $-y_2 + y_3 + y_4 = 1$ projected to the y_2, y_3-plane and its image under its respective mapping. This shows that it does map into itself, but not exclusively. This is not enough to deduce the periodicity of the network.

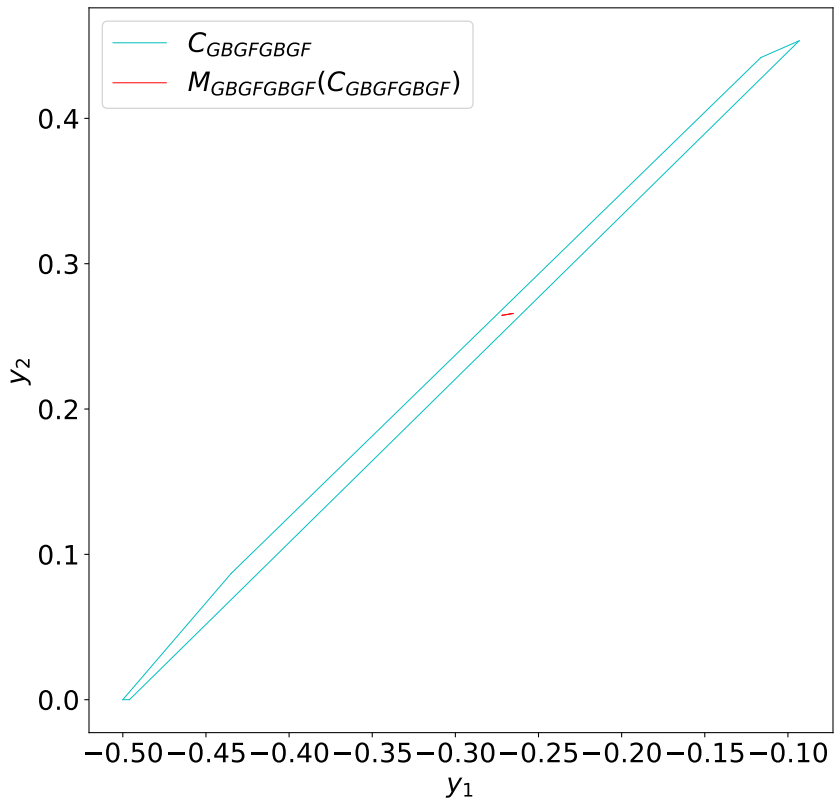


Figure 3.16: Cross sections of returning cones for the sequence of cycles $GBGFGBGF$ with the plane $-y_2 + y_3 + y_4 = 1$ projected to the y_2, y_3-plane and its image under its respective mapping. Showing that it exclusively maps into itself, thus, forming a trapping region of one word.

Chapter 4
Future work

Through the last two chapters, we have seen the theory and original entropy bound presented by Farcot [15] develop quite substantially into a viable method for approximating symbolic entropy in Glass networks. In Chapter 2 we showed that given a Glass network satisfying a few basic properties, we can alter its TG to be a more accurate depiction of the underlying dynamics, that in turn gives a much better upper bound on entropy. Additionally, we showed that this method of improving the upper bound can be done to exclude arbitrary length concatenations of cycles. Chapter 2 concludes with a convergence result that, in the limit as the concatenation length approaches infinity, the upper bounds converge to the system's true entropy. One potential downside to our method is that the reduction process involves a lot of moving parts that all require calculation. Doing all of this by hand becomes impractical for complicated examples. Thus, in Chapter 3 we discussed how all of the moving parts involved in the refinement process work and can be fit together in the form of an algorithm allowing for complete automation of the refinement process. This allows the algorithm to be used as a tool to approximate a Glass network's entropy to any level of precision. We concluded Chapter 3 with some worked examples to detail what to expect when the algorithm is applied to Glass networks displaying different types of dynamics.

We have effectively made symbolic entropy a viable method for assessing the irregularity of Glass network dynamics. It is our hope that our theory and algorithm can be used to not only study irregular dynamics in Glass networks, but also, Glass networks can be seriously investigated as potential designs for TRNGs. Now, while our method of approximating symbolic entropy does give a meaningful way of assessing these networks, it does come with a few caveats. For example, condition 5 assumes that the starting wall is only partitioned by a finite number of cycles. This condition was imposed so that we could avoid the case of an infinite number of nodes on the graph since there are many challenges with computing topological entropy in that case [54]. By tweaking our theory slightly, it is our belief that we can develop a new theoretical approach to the refinement process that may allow for the analysis of these types of networks. Additionally, we have ideas for ways to make numerically generated

bounds more robust and accurate. Finally, we have some ideas for applications of our algorithm. Since we are constrained for time the ideas discussed in this chapter will not be completed, but instead only briefly mentioned as topics for future work. We begin with a discussion on numerically generated bounds.

4.1 More robust numerical bounds

At the end of Chapter 2 we discussed using numerically generated entropy bounds as a way of gauging the accuracy of the rigorous bound we provided for our example. The way we did this was not very sophisticated. Effectively, we took the example network and integrated it for a very long time from many different random initial conditions. Associating each box with a given symbol, we then converted each trajectory into a long concatenation of these symbols and counted the blocks of length n up to 126. Since we know that for a chaotic system, blocks must grow approximately exponentially, we fit a straight line to the logarithm of the number of blocks as a function of n. Then we were able to approximate the entropy of the system using the slope of that line. This has a number of potential problems. For any n we can never be certain we have integrated the system for long enough to generate all of the possible blocks. Additionally, fitting a curve to the number of blocks is only an approximation and may err on either side. As a result, we ended up with a value that we expect to be close to the true entropy, but had no means of rigorous justification. Additionally, as a check of how accurate our rigorous bound was, it was not very quick. The code had to be optimized substantially by parallelization and it still took a long time to run. The bound could have potentially been made a bit more accurate in estimating the slope by considering n from a larger starting point, say 100, since we expect this value to asymptotically approach the true slope. However, this will not improve the bound much as it still takes the same amount of time to generate the trajectories, which is likely where the most error comes from: mainly, the number of blocks. While there is not much that can be done in terms of curve fitting, being certain we have generated all of the blocks for a given n is something that can be made robust by using returning cones.

Once a trapping region has been identified, it is simple to find returning cones for cycles as well as concatenations of cycles. Each cycle in the trapping region represents an allowed word in the alphabet. It is just a simple combinatorial exercise to find the allowed blocks up to the length of the shortest concatenation of cycles up to length n. This gives the exact number of blocks of length n for any finite n. If we use this idea to generate the number of blocks of length n for each n up to some maximum N and then fit a curve to that data, this will give us a much more accurate numerical approximation of entropy. The potential source of error in this is that it may be that for some much larger n than the imposed maximum, there is a change in the trend to the number of allowed blocks. However, since calculating returning cones is a very

quick calculation as compared to numerically simulating trajectories, this method will have the added benefit of being an actual quick check.

This method also has the added benefit of being rigorous in the case of a periodic example. Since in the case of periodic behaviour, eventually the trapping region will be made up of only one region, there will only be one block forever for any larger n. Hence for some finite n you can find all the necessary allowed blocks to find the exact entropy.

Additionally, you can find the exact number of blocks of length n using pre-images of $\mathcal{M}$ and its iterates. Similar to the definition of returning regions, we can just as easily define which regions on walls will map to other walls without reference to cycles or a starting wall. This idea could also be used to develop another theoretical approach to the result achieved in our paper. We will discuss this more in section 4.3.

4.2 A quasi-periodic example

Another very complicated type of dynamics possible in Glass networks is quasi-periodicity. This behaviour, while not as trivial as a classical limit cycle, would be disadvantageous to generating random numbers since quasi-periodic behaviour has zero entropy. As a result, it would be very useful to be able to see how our algorithm behaves when given a network structure that displays quasi-periodic behaviour. Unfortunately due to time constraints we were not able to study the following example network that has been identified as quasi-periodic:

$$\begin{aligned}
\frac{dy_1}{dt} &= 1 - 2Y_2 + \frac{1}{2}Y_1Y_2 - y_1 \\
\frac{dy_2}{dt} &= 2Y_1 - 1 - y_2 \\
\frac{dy_3}{dt} &= 1 - 2Y_4 + kY_3Y_4 - y_3 \\
\frac{dy_4}{dt} &= 2Y_3 - 1 - y_4
\end{aligned} \tag{4.1}$$

where $0 < k < 1$, and k is not $1/2$. As in previous examples, $Y_i = 1$ if $y_i > 0$ and 0 if $y_i < 0$. This is just a pair of 2-unit oscillators that are uncoupled, and if k is not $1/2$ then the periods of the two oscillators will be different, and so as long as they don't happen to be commensurate periods, it will be quasi-periodic. Phase plane simulations, where $k = 1/3$, support this claim and can be seen in Figure 4.2.

For quasi-periodic networks, we believe that the blocks of length n will grow more slowly than exponentially, probably linearly. If this is the case, we will see a sequence of upper bounds that will asymptotically approach zero. We believe that running our algorithm to a sufficiently large refinement level will demonstrate this and the user will

Y_1	Y_2	Y_3	Y_4	F_1	F_2	F_3	F_4
0	0	0	0	1	-1	1	-1
0	0	0	1	1	-1	-1	-1
0	0	1	0	1	-1	1	1
0	0	1	1	1	-1	-2/3	1
0	1	0	0	-1	-1	1	-1
0	1	0	1	-1	-1	-1	-1
0	1	1	0	-1	-1	1	1
0	1	1	1	-1	-1	-2/3	1
1	0	0	0	1	1	1	-1
1	0	0	1	1	1	-1	-1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	-2/3	1
1	1	0	0	-1/2	1	1	-1
1	1	0	1	-1/2	1	-1	-1
1	1	1	0	-1/2	1	1	1
1	1	1	1	-1/2	1	-2/3	1

Table 4.1: Table of focal points for quasi-periodic example.

be able to deduce that a network is quasi-periodic from the sequence of upper bounds.

4.3 Wall-to-wall transitions instead of cycles

As we have hinted previously, it is possible to define which region of a wall maps to another. If we denote a Glass network's collection of walls contained within the domain $\mathscr{D}$ as $\mathscr{W}$, for two walls $\omega_i, \omega_j \in \mathscr{W}$ the region on wall i that gets mapped into wall j is simply

$$\omega_i \cap \mathcal{M}^{-1}(\omega_j).$$

Generalizing this to sequences of wall-to-wall transitions is straightforward. For the collection of n walls $\{\omega_{i_1}, \omega_{i_2}, \ldots, \omega_{i_n}\}$ and defining $S_i(\omega) = \omega_i \cap \mathcal{M}^{-1}(\omega)$, the region on wall i_1 that gets mapped through i_2 to i_n, in order, is just

$$S_{i_1} \circ S_{i_2} \circ \ldots \; S_{i_{n-1}}(\omega_{i_n}).$$

We can then systematically check every possible sequence of n walls in the TG, and remove the impossible ones to make a shift of finite type that forbids impossible blocks of length n. The convergence of the entropy of this sequence of shifts of finite type is known. Practical calculation of entropy can be achieved as follows. As discussed by Lind and Marcus [45], every shift of finite type is conjugate to a vertex shift. So, computing entropy in this case boils down to finding the vertex shift representing each

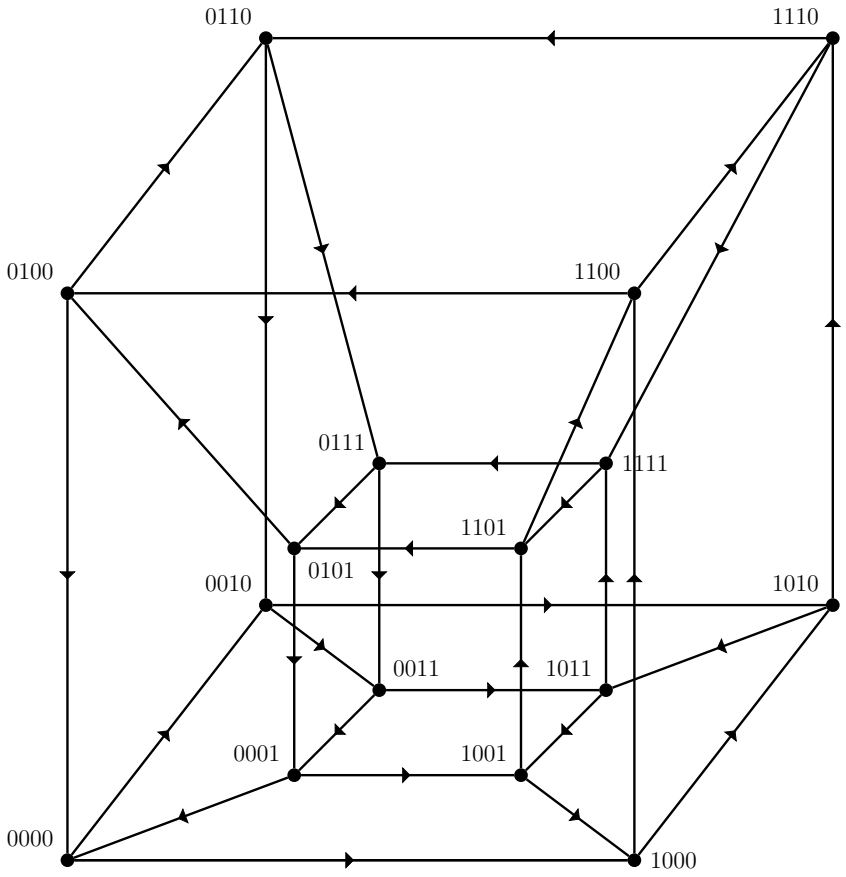
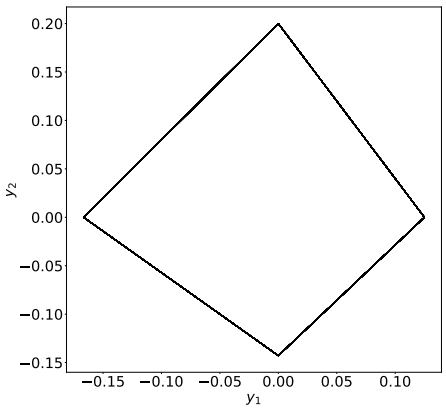
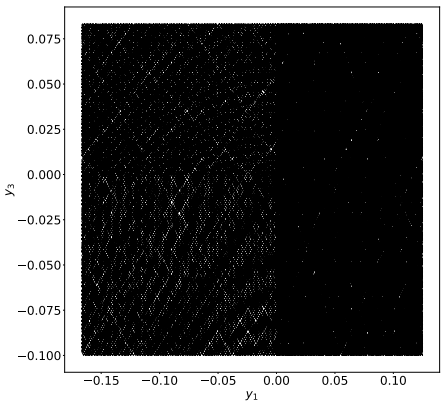


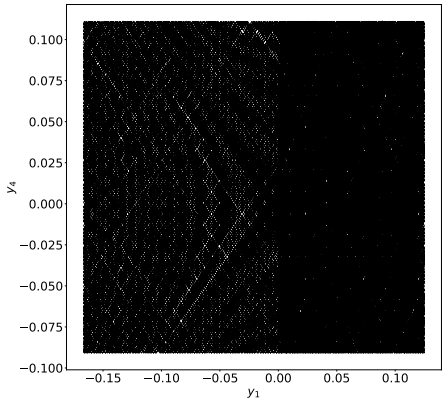
Figure 4.1: Quasi-periodic example



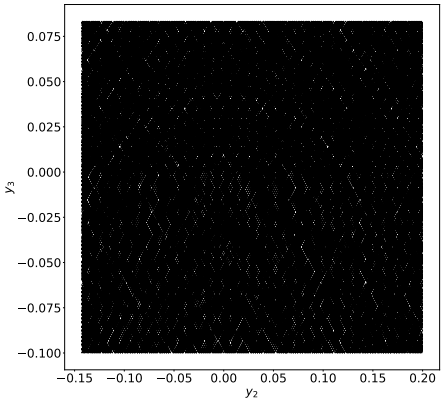
(a)



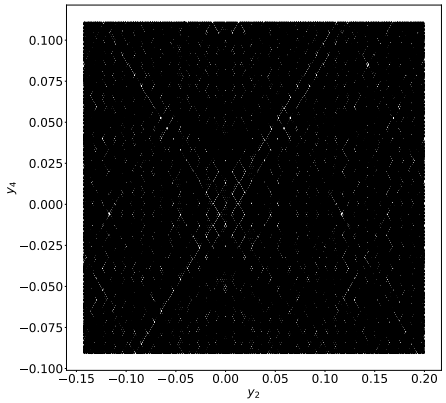
(b)



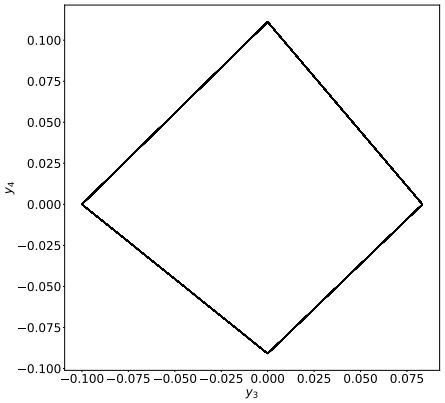
(c)



(d)



(e)



(f)

Figure 4.2: Phase plane projections for quasi-periodic example

shift space.

This comes with some difficulties as this will be much more complicated than using returning cones. So much of the intuition that could be used in the case of returning cones is now not feasible. However, where we lose intuition, we expand our theory. In this situation, there is no point in identifying a trapping region since you consider the whole space instead of just a restriction of the domain $\mathscr{D}$. Additionally, you will not have to remove transients. Possibly the biggest gain from constructing the sequence of upper bounds in this way is that you can also deal with the case of an infinite number of cycles, since no possibility for infinite alphabets will arise since we just remove blocks of length n in general, not centred around a starting wall. Another potential drawback to this approach however is in distinguishing disjoint trapping regions. There is still much in favour using a starting wall and returning cones. The process is simple and as we saw we can get accurate upper bounds quickly with the algorithm implementation of that theory.

4.4 Cycles as alphabets

Up to this point, every way of defining the shift representations has revolved around using a single box or single wall as the symbol with which the alphabet is defined. We now wish to draw attention to another approach, that is, defining an alphabet where each symbol represents groups of boxes or walls. Each symbol in this new alphabet represents finite words in the our previous alphabets. Focusing our attention on our main example with the two cycles A and B, what if instead of bothering with all the separated cycle representations or wall-to-wall mappings, we defined an alphabet using just the symbols A and B? It would not be difficult to develop the theory along these lines and in some ways this might actually be simpler than our previous methods. However, the problem arises when trying to compare two shift spaces with different alphabets.

Since for each example, cycles have different lengths and represent different cycles traversing the systems, one would need to find a way of relating the dynamics in a common language since application of the shift operator in one system will represent a different amount of dynamics in another and entropies calculated on different shift spaces with different alphabets will not be directly comparable. This idea is not unheard of, and some results have been published [53, 41]. However, this idea is not trivial. We believe investigation of this is warranted, but within the context of generating random numbers, this approach may not be ideal. However, further developments will be needed to accurately make that assessment.

4.5 Infinite alphabets

In Chapter 2, Condition 5 was imposed to avoid the starting wall being partitioned by an infinite number of cycles. In this situation, the procedure in our paper would result in a shift space that has an infinite alphabet [34, 45, 54, 52]. As discussed by Lind and Marcus [45], shift spaces can be defined using infinite alphabets. However, they become very challenging mathematical objects. Their definition is the same as the case of finite alphabets, however their analysis is not. To properly analyze shift spaces with infinite alphabets requires C^* algebras [34]. Additionally, the notion of entropy in the case of infinite alphabets is not really well defined. Lind and Marcus have discussed how one might go about computing entropy in this case. Other attempts have been made with some success [54]. However, it is agreed upon that computation of entropy in even the most simple examples of shift spaces with infinite alphabets is non-trivial. For this reason, along with the many alternative solutions we derived while working on this, we did not investigate using infinite alphabets. However, we do believe this could be worth investigating in the future due to the tractability of Glass network analysis. In somewhat of a reverse process idea, Glass networks may be a useful tool to study shift spaces with infinite alphabets. Glass networks have been used to study non-trivial mathematical structures in the past and their future use may include, but not be limited to, shift spaces with infinite alphabets, classically smooth nonlinear differential equations, and even potentially networks with higher order interactions.

4.6 A network survey of dynamical diversity and structural properties

In a previous publication by Edwards and Glass [12], they explored combinatorial aspects of Glass network structure in a simplified model where the governing equations took the form

$$\dot{y}_i = y_i + F_i(\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_n), \quad i = 1, \ldots, n,$$

where

$$\tilde{y}_i = \begin{cases} 0 \text{ if } y_i < 0 \\ 1 \text{ if } y_i > 0. \end{cases}$$

They also restricted this model to the case where all focal point coordinates were at ± 1 (so values of F_i are ± 1), and networks did not contain self input (so F_i does not depend on y_i). Edwards and Glass discussed many combinatorial results that arose from this basic structure. They also performed a random survey of one million 4 dimensional networks, in which they were able to categorize the dynamics that arose into five distinct behaviour classes: nodes, foci, non-degenerate periodic orbits, degenerate

periodic orbits, and irregular. This survey demonstrated the rich potential for dynamical diversity well and suggested that even in the most simple Glass network structure, there exists very complex dynamics and diversity.

Our new method of approximating entropy could now be used to perform our own survey, using entropy as the dynamical identifier. The method used by Edwards and Glass to distinguish between irregular and regular was based on numerically computed trajectories (using exact wall-to-wall maps of course) and some examples may have been misclassified. They also did not deal with possible multistability of different attractors in a given network. Since our algorithm is good at distinguishing between true chaos and other types of dynamics, we believe we could do a more detailed description of the dynamical diversity arising in network structures. Additionally, Edwards and Glass only considered the simplified model above. Our algorithm is capable of dealing with arbitrary production functions, and hence arbitrary focal points (though not unequal decay rates). As a result, we believe that we can not only recreate the results of their survey with more accurate assessments of network irregularity, but also can expand the scope of the survey to consider more complicated network structures. This could be a useful tool for determining the proportion of Glass networks of a given class exhibiting chaotic dynamics.

Another aspect of Glass network structure that we believe our entropy method could be helpful in studying is bifurcation. In a previous publication by Edwards et al. [14], they discussed how symbolic representations of Glass networks can be used to identify chaotic bifurcations by way of a change of alphabet. We believe that similar results can be identified using our entropy method by a sudden change in entropy. Additionally, the they used the same simplified model as in [12] without the focal point restriction to ± 1, so our method has the same advantage of including a larger class of networks. As a result, our entropy method could potentially be used to extend their previous result along with potentially identifying other bifurcation properties [40].

Finally, our algorithm can be used to study high-dimensional network examples. The circuit design proposed by Farcot et al. [16] is a $4n$-dimensional Glass network with $n \geq 5$. Our algorithm could be an effective tool to analyze that model, along with other high-dimensional designs for TRNGs.

Chapter 5
Concluding remarks

At last, we come to the end of the long strange trip that has been our study of entropy bounds for Glass networks. We have seen the theory develop quite substantially from the original bound provided by Farcot [15]. With the work that we have done here, symbolic entropy has become a viable quantity for assessing the potential irregularity of dynamics in Glass networks. It is our hope that using symbolic entropy as a gauge for irregular dynamics will become a useful analytical tool in studying Glass networks. We also believe that symbolic entropy will be an invaluable tool in the design of TRNGs based on Glass networks. Beyond use in Glass networks, we hope that this work will inspire similar ideas in other areas of non-smooth dynamical systems. Symbolic entropy may have similar potential for deciphering irregular dynamics in general piecewise smooth dynamical systems [7]. Beyond symbolic entropy, shift spaces have been successful in studying Glass networks and it is our belief that they have uses in other piecewise smooth systems as well as other complex networks.

Now that symbolic entropy can be accurately approximated in Glass networks, we believe that topological and analytical properties of symbolic entropy warrant attention. For instance, if two Glass networks can be considered topologically equivalent, do they have equivalent entropy? Or maybe there is a scaling factor that relates their symbolic entropy? Additionally, by Farcot's design of the symbolic representation of Glass networks the symbolic entropy and classical topological entropy are not equal. Is there some relationship between symbolic entropy and the true topological entropy of $(\mathscr{D}, \mathcal{M})$? Returning regions could perhaps be used in defining open covers.

Doing our refinement process by hand, while possible, is very tedious and will likely make it more of a chore to use than an actual asset. Automating the refinement process is a key step in making it viable. Within the context of generating random numbers for cyber security applications, our entropy bounds paired with the algorithm gives a reasonably simple way of assessing a large number of Glass networks and hence TRNG designs.

However, if one wants to use Glass networks as TRNGs there are still many things beyond just the intrinsic chaos of the underlying model that need to be taken into

consideration. For example, even if the actual value of entropy could be exactly calculated, it can only be treated as an estimate for the entropy generated by the circuits, because the circuit is a physical implementation of a Glass network and hence will be subject to thermal noise. Thermal noise will contribute additional entropy to the circuit, possibly making it a more irregular random number generator. Additionally, in a physical implementation, the circuit will have continuous outputs that are not discrete like the mappings typically dealt with in Glass network analysis. So in order to use this to generate random bit sequences, a sampling rule will need to be selected. The rate at which random bits are extracted from a trajectory will have the an impact on the entropy of the RNG. Hence the thermal noise, sampling rate, and intrinsic chaos of the network itself will contribute to the randomness of bits extracted from a Glass network TRNG.

It is also likely that the intrinsic chaos of the underlying network will aid in defence against hacking in the classical sense for physical TRNGs, frequency injection. Numerical experiments have suggested that for Glass networks with high entropy, the frequency injection at amplitudes achievable in a physical circuit cannot entrain the dynamics into a predictable pattern as it does in classical ring oscillators. If these numerical experiments are indicative, then being able to design Glass networks to have the highest possible entropy is central in designing a more robust TRNG. The theory we have developed here can be used to gauge how irregular a network is and hence can be used as a valuable tool in finding good designs for TRNGs.

Still on the topic of generating random numbers, our Condition 5 was imposed to avoid the situation of an infinite number of cycles causing an infinite alphabet after splitting. However, it may be more advantageous to generating random numbers that this happens. As a result, we plan to develop the theory suggested previously to redefine our results using wall-to-wall images so that these types of systems can be identified and studied in more detail. While our Condition 5 can likely be relaxed, Farcot's conditions 1 and 2 are absolutely necessary for meaningful solutions. Additionally, removing examples with white and black walls is actually beneficial if high entropy is the goal. In systems with black walls, all of the dynamics gets sucked onto a lower-dimensional attractor and hence will not be as beneficial for irregular dynamics. While these conditions restrict the number of eligible Glass network examples that our theory can examine, there are still many examples to choose from.

If we are trying to maximize the entropy generated from a circuit implementation, multiple disjoint trapping regions, each with its own chaotic attractor, may be a great way to introduce even more entropy. Since each attractor in space will have its own potential for generating entropy, the random "jitter" in the circuit could allow for trajectories to jump between chaotic attractors aperiodically. As a result, circuits could be designed around multiple disjoint trapping regions to take advantage of this and generate higher entropy. Multiple disjoint trapping regions can be identified by a

separating of its $\mathrm{TG}_r(n)$. So, one could use this fact similarly to the way Edwards et al. [14] use a change in alphabet to identify chaotic bifurcations. In fact, the splitting of the graph represents a change in alphabet not too dissimilar to their result.

Finally, we can conclude that our method of approximating entropy has made symbolic entropy a new valuable tool for Glass network analysis and has opened many new avenues for future work in the study of chaotic dynamics in Glass networks!

Bibliography

[1] R. L. Adler, A. G. Konheim, and M. H. McAndrew. Topological entropy. *Transactions of the American Mathematical Society*, 114(2):309–319, 1965.

[2] P. Bayon, L. Bosuet, A. Aubert, V. Fischer, F. Poucheret, B. Robisson, and P. Maurine. Contactless electromagnetic active attack on ring oscillator based true random number generator. In W. Schindler and S. A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 151–166. Springer, Berlin, Heidelberg, 2012.

[3] A. Berman and R. Plemmons. *Nonnegative Matrices in the Mathematical Sciences*. Academic Press, New York, second edition, 1994.

[4] H. L. d. S. Cavalcante, D. J. Gauthier, J. E. Socolar, and R. Zhang. On the origin of chaos in autonomous Boolean networks. *Philosophical Transactions of the Royal Society A*, 368:495–513, 2010. Presented at the International Workshop on Delayed Complex Systems (2009).

[5] R. L. Devaney. *An introduction to chaotic dynamical systems*. Addison-Wesley, Redwood City, Calif, 2nd ed. edition, 1989 - 1989.

[6] S. N. Dhanuskodi, A. Vijayakumar, and S. Kundu. A chaotic ring oscillator based random number generator. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 160–165. IEEE, 2014.

[7] M. di Bernardo, C. J. Budd, A. R. Champneys, and P. Kowalczyk. *Piecewise-smooth dynamical systems*, volume 163 of *Applied Mathematical Sciences*. Springer-Verlag London, Ltd., London, 2008. Theory and applications.

[8] T. Downarowicz. Positive topological entropy implies chaos DC2. *Proceedings of the American Mathematical Society*, 142(1):137–149, 2014.

[9] R. Edwards. Analysis of continuous-time switching networks. *Phys. D*, 146(1-4):165–199, 2000.

[10] R. Edwards. Chaos in neural and gene networks with hard switching. *Differential Equations Dynam. Systems*, 9(3-4):187–220, 2001.

[11] R. Edwards, E. Farcot, and E. Foxall. Explicit construction of chaotic attractors in Glass networks. *Chaos, Solitons & Fractals*, 45(5):666–680, 2012. Chaos and dynamics in biological networks.

[12] R. Edwards and L. Glass. Combinatorial explosion in model gene networks. *Chaos*, 10(3):691–704, 2000.

[13] R. Edwards and L. Glass. A calculus for relating the dynamics and structure of complex biological networks. In R. S. Berry and J. Jortner, editors, *Adventures in Chemical Physics*, Advances in Chemical Physics, vol. 132, pages 585–611. Wiley, Hoboken, 2006.

[14] R. Edwards, H. T. Siegelmann, K. Aziza, and L. Glass. Symbolic dynamics and computation in model gene networks. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 11(1):160–169, 03 2001.

[15] E. Farcot. Geometric properties of a class of piecewise affine biological network models. *J. Math. Biol.*, 52(3):373–418, 2006.

[16] E. Farcot, S. Best, R. Edwards, I. Belgacem, X. Xu, and P. Gill. Chaos in a ring circuit. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 29(4):043103, 04 2019.

[17] A. F. A. F. Filippov and F. M. Arscott. *Differential equations with discontinuous righthand sides*. Mathematics and its applications (Soviet series). Kluwer Academic Publishers, Dordrecht [Netherlands] ;, 1988 - 1988.

[18] T. French. Follower, predecessor, and extender set sequences of β-shifts. *Discrete and Continuous Dynamical Systems*, 39(8):4331–4344, 2019.

[19] X.-C. Fu, W. Lu, P. Ashwin, and J. Duan. Symbolic representations of iterated maps. *Topol. Methods Nonlinear Anal.*, 18(1):119–147, 2001.

[20] Z. Galias. Positive topological entropy of Chua's cirtuit: A computer assisted proof. *International Journal of Bifurcation and Chaos*, 7:331–349, 1996.

[21] T. Gedeon. Attractors in continuous-time switching networks. *Commun. Pure Appl. Anal.*, 2(2):187–209, 2003.

[22] R. Gesztelyi, J. Zsuga, A. Kemeny-Beke, B. Varga, B. Juhasz, and A. Tosaki. The hill equation and the origin of quantitative pharmacology. *Archive for history of exact sciences*, 66(4):427–438, 2012.

[23] L. Glass. Classification of biological networks by their qualitative dynamics. *Journal of Theoretical Biology*, 54(1):85–107, 1975.

[24] L. Glass. Combinatorial and topological methods in nonlinear chemical kinetics. *The Journal of Chemical Physics*, 63(4):1325–1335, 08 1975.

[25] L. Glass. Combinatorial aspects of dynamics in biological systems. In U. Landman, editor, *Statistical mechanics and statistical methods in theory and application (Proc. Sympos., Univ. Rochester, Rochester, N.Y., 1976)*, pages 585–611. Plenum, New York, London, 1977.

[26] L. Glass. Boolean and continuous models for the generation of biological rhythms. In J. Demongeot, E. Golès, and M. Tchuente, editors, *Dynamical systems and cellular automata (Luminy, 1983)*, pages 197–206. Academic Press, London, 1985.

[27] L. Glass and R. Edwards. Hybrid models of genetic networks: Mathematical challenges and biological relevance. *Journal of Theoretical Biology*, 458:111–118, 2018.

[28] L. Glass and S. A. Kauffman. Co-operative components, spatial localization and oscillatory cellular dynamics. *Journal of Theoretical Biology*, 34(2):219–237, 1972.

[29] L. Glass and S. A. Kauffman. The logical analysis of continuous, non-linear biochemical control networks. *Journal of Theoretical Biology*, 39(1):103–129, 1973.

[30] L. Glass and S. A. Kauffman. The logical analysis of continuous, non-linear biochemical control networks. *Journal of Theoretical Biology*, 39(1):103–129, 1973.

[31] L. Glass and J. S. Pasternack. Prediction of limit cycles in mathematical models of biological oscillations. *Bull. Math. Biology*, 40(1):27–44, 1978.

[32] L. Glass and J. S. Pasternack. Stable oscillations in mathematical models of biological control systems. *J. Math. Biol.*, 6(3):207–223, 1978.

[33] L. Glass, T. J. Perkins, J. Mason, H. T. Siegelmann, and R. Edwards. Chaotic dynamics in an electronic model of a genetic network. *J. Stat. Phys.*, 121(5-6):969–994, 2005.

[34] D. Gonçalves and D. Royer. Ultragraphs and shift spaces over infinite alphabets. *Bulletin des sciences mathématiques*, 141(1):25–45, 2017.

[35] A. Hajimiri, S. Limotyrakis, and T. H. Lee. Jitter and phase noise in ring oscillators. *IEEE Journal of Solid State Circuits*, 34(6):790–804, 1999.

[36] A. V. Hill. The possible effects of the aggregation of the molecules of hæmoglobin on its dissociation curves. *The Journal of Physiology*, 40:i–vii, January 1910.

[37] M. Inubushi. Unpredictability and robustness of chaotic dynamics for physical random number generation. *Chaos*, 29:033133, 2019.

[38] F. Jacob and J. Monod. Genetic regulatory mechanisms in the synthesis of proteins. *Journal of molecular biology*, 3(3):318–356, 1961.

[39] S. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467, 1969.

[40] D. B. KILLOUGH and R. EDWARDS. Bifurcations in glass networks. *International Journal of Bifurcation and Chaos*, 15(02):395–423, 2005.

[41] T. Kucherenko and D. J. Thompson. Measures of maximal entropy on subsystems of topological suspension semiflows. *Studia mathematica*, 260(2):229–240, 2021.

[42] N. Kuznetsov, O. Kuznetsova, G. Leonov, and V. Vagaitsev. Analytical-numerical localization of hidden attractor in electrical Chua's circuit. In *Information in Control, Automation and Robotics: 8th International Conference, ICINCO 2011, Revised Selected Papers*, pages 149–158. Springer, Berlin, Heidelberg, 2011.

[43] J. E. Lewis and L. Glass. Steady states, limit cycles, and chaos in models of complex biological networks. *Internat. J. Bifur. Chaos Appl. Sci. Engrg.*, 1(2):477–483, 1991.

[44] Q. Li and X.-S. Yang. Chaotic dynamics in a class of three dimensional Glass networks. *Chaos*, 16(3):033101, 5, 2006.

[45] D. Lind and B. Marcus. *An Introduction to Symbolic Dynamics and Coding*. Cambridge Mathematical Library. Cambridge University Press, Cambridge, second edition, 2021.

[46] Y. Luo, W. Wang, S. Best, Y. Wang, and X. Xu. A high-performance and secure TRNG based on chaotic cellular automata topology. *IEEE Transactions on Circuits and Systems - I: Regular Papers*, 67:4970–4983, 2020.

[47] A. T. Markettos and S. W. Moore. The frequency injection attack on ring-oscillator-based true random number generators. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 317–331. Springer, Berlin, Heidelberg, 2009.

[48] H. Martín, T. Korak, E. S. Millán, and M. Hutter. Fault attacks on STRNGs: Impact of glitches, temperature, and underpowering on randomness. *IEEE Transactions on Information Forensics and Security*, 10(2):266–277, 2015.

[49] T. Matsumoto. A chaotic attractor from Chua's circuit. *IEEE Transactions on Circuits and Systems*, CAS-31:1055–1058, 1984.

[50] T. Mestl, R. J. Bagley, and L. Glass. Common chaos in arbitrarily complex feedback networks. *Phys. Rev. Lett.*, 79:653–656, Jul 1997.

[51] T. Mestl, C. Lemay, and L. Glass. Chaos in high-dimensional neural and gene networks. *Phys. D*, 98(1):33–52, 1996.

[52] W. Ott, M. Tomforde, and P. Willis. One-sided shift spaces over infinite alphabets. *arXiv.org*, 2013.

[53] W. Parry. Intrinsic markov chains. *Transactions of the American Mathematical Society*, 112(1):55–66, 1964.

[54] S. Rezagholi. Subshifts on infinite alphabets and their entropy. *Entropy*, 22(11), 2020.

[55] D. P. Rosin. *Dynamics of Complex Autonomous Boolean Networks*. Springer Theses. Springer Cham, Heidelberg, 2015.

[56] D. P. Rosin, D. Rontani, and D. J. Gauthier. Ultrafast physical generation of random numbers using hybrid Boolean networks. *Physical Review E*, 87:040902(R), 2013.

[57] S. Srivastava and J. Roychowdhury. Analytical equations for nonlinear phase errors and jitter in ring oscillators. *IEEE Transactions on Circuits and Systems: I Regular Papers*, 54(10):2321–2329, 2007.

[58] C. Tanougast. Hardware implementation of chaos based cipher: Design of embedded systems for security applications. In L. Kocarev and S. Lian, editors, *Chaos-Based Cryptography*, pages 297–330. Springer, Berlin, Heidelberg, 2011.

[59] B. W. Wild and R. Edwards. Entropy bounds for glass networks. *arXiv preprint arXiv:2406.04435*, 2024.

[60] R. Zhang, H. L. d. S. Cavalcante, Z. Gao, D. J. Gauthier, J. E. Socolar, M. M. Adams, and D. P. Lathrop. Boolean chaos. *Physical Review E*, 80:045202(R), 2009.