Universidade Técnica de Lisboa
Instituto Superior Técnico

On the Computational Power of Sigmoidal Neural Networks

Ricardo Joel Marques dos Santos Silva

Diploma Thesis

Supervisor:
Prof. Dr. José Félix Costa

July 2002

Acknowledgments

Nine months ago, when I first started to work on this project, I kept thinking "Will I be able to do anything good?" I looked into the future and felt troubled by the possibility of spending months investigating without reaching any significant conclusion. Now, that the work is done, I look back and see how this work was only possible thanks to so many people. I know that many and long acknowledgements are awkward and that being capable of writing the essential acknowledgements in a concise way is a quality and a sign of distinction, but I choose to be uncouth rather than ungrateful.

Most of this work is a fruit of Felix. First of all, the idea to clarify the question of the computational power of sigmoidal networks was his. He taught me almost everything I know about neural networks. During this year, not only he kept providing me with the necessary material (every time I entered his office, I came out with a new stack of books) and guidance, but also with the important tips for a young "rookie" doing scientific research for the first time on his life. For all this, thank you Félix! But more than for all the reasons stated above, thank you for your patience revising so carefully everything I wrote, for turning our work meetings into a chance to learn and discuss; and for all the understanding and goodwill as I kept breaking all the deadlines we established.

I want to thank Cristopher Moore for his generosity and his helpfulness, both when, on his passing by Lisbon, he explained me the point at which he suspected Hava's work could be wrong and his and Pascal's conjecture; and last May when he indicated me the statement of their conjecture, so I might quote it properly.

I must thank my classmates and workfellows Alexandre and Pedro Adão for their efforts in teaching me LaTeX. I am also indebted to Alexandre for all the times when the preparation of EDA lessons laid on him.

João Boavida helped out with his precious LaTeXadvice.

I am grateful to all my friends of the Emmanuel Community, for helping me to deepen my relationship with God, to maintain my health of mind, and for being an endless source of true friendship.

Gerald and N'Cok (that certainly can't imagine to have anything to do with this work), have been for me the best examples of what one's willpower can do, and made me remember that things we usually take for granted (to have a house to manage, to be in College, ...), so much that we tend to feel them as a burden when they start to demand an extra bit of work, are in fact opportunities we mustn't waist.

To all my pupils I thank the almost unbelievable way how they stayed in class and managed to understand what I was saying, in spite of my failures to display the subjects I addressed in a clear way, at 8 am after a night spent working.

I thank also to Filipa, Rita and João, the workgroup of BMG laboratory classes, for allowing me to do so little in the reports.

I dedicate this thesis to my parents as a *tiny* sign of my gratitude, not just because they are paying for my College studies, not just because they have raised me and have been my most influential educators, not just because they gave me life, but especially for all the times when they had to love me without understanding me (and all the future times when they will have to do it again).

Agradecimentos

Há cerca de nove meses, quando peguei neste trabalho pela primeira vez, afligia-me pensar: "Será que vou conseguir fazer alguma coisa de jeito?" Olhava para a frente e assustava-me perante a possibilidade de passar meses a investigar sem chegar a conclusão nenhuma. Agora, que o trabalho está acabado, olho para trás e vejo como este trabalho só foi possível graças a um conjunto de muitas pessoas. Sei que muitos e longos agradecimentos são deselegantes, e que ser capaz de escrever de forma sucinta os agradecimentos essenciais é uma qualidade e um sinal de distinção, mas antes quero ser bronco que mal-agradecido.

Grande parte deste trabalho se deve ao Félix. Antes de mais, a ideia de fazer um trabalho a esclarecer a questão do poder computacional das redes sigmoidais foi dele. Depois, foi ele que me ensinou quase tudo o que sei sobre redes neuronais. Ao longo do ano, não só me foi dando todo o material necessário (cada vez que entrava no seu gabinete saía com uma pilha de livros nova), mas principalmente as dicas importantes para um jovem "maçarico" a fazer investigação pela primeira vez na vida. Por tudo isto, obrigado Félix! Mas mais do que todos os motivos atrás citados, obrigado pela paciência que foi necessária para ler minuciosamente tudo o que eu escrevi, obrigado por as reuniões em que discutíamos o trabalho terem sido não só reuniões de trabalho, mas também ocasião de aprender e conversar; e obrigado pela compreensão e boa vontade quando eu quebrei sistematicamente todos os prazos que íamos combinando.

Gostava de agradecer ao Cristopher Moore a sua boa-vontade, o ter sido tão prestável, quer quando, de passagem por Lisboa, me esteve a explicar os pontos em que suspeitava que o trabalho da Hava pudesse falhar e a conjectura que ele e o Pascal tinham, quer agora em Maio quando me indicou a conjectura para que eu a pudesse citar convenientemente.

Ao Alexandre e ao Adão pela paciência que tiveram para me ensinar LaTeX. Ao Alexandre agradeço também as vezes em que a preparação das aulas de EDA recaiu sobre ele.

Ao João Boavida agradeço os conselhos de LaTeX.

A todos os meus amigos da Comunidade Emanuel, por ajudarem a manter a minha sanidade mental, a minha relação com Deus, e por serem uma fonte inesgotável de verdadeira amizade.

Ao Gerald e ao N'Cok (que nem fazem a mínima ideia de me terem ajudado a fazer este trabalho), por terem sido para mim o maior exemplo do que pode fazer a força de vontade, e por me fazerem recordar que as coisas que normalmente temos como certas (ter uma casa para governar, estar na Universidade, ...) ao ponto de as sentir como um fardo quando dão um pouco mais de trabalho, são na verdade uma oportunidade que não devemos desperdiçar.

Aos meus alunos de EDA e de PR agradeço a forma quase inacreditável como se conseguiram manter nas aulas e perceber aquilo que lhes queria transmitir, apesar da minha incapacidade de expor a matéria de forma clara às oito da manhã após uma noitada de trabalho.

Agradeço também à Filipa, à Rita e ao João, o grupo de laboratório de BMG, por permitirem que eu participasse de forma tão ligeira (fica mal agradecer por permitirem que eu me baldasse) nos relatórios.

Dedico este trabalho aos meus pais como *minúsculo* sinal do meu reconhecimento, não só porque são eles que me estão a pagar o curso, não só porque foram eles os meus primeiros e principais educadores, não só porque me deram a vida, mas principalmente por todas as vezes que tiveram (e vão continuar a ter) que me amar sem me perceber.

Contents

Chapter 1

Introduction

Simulation of Turing machines by neural networks was done first by McCulloch and Pitts in 1943 [MP43], and then revisited by Kleene in [Kle56]. The result that neural networks can simulate Turing machines is well-known. Besides the articles already cited, this result appears in a great number of books. The reader interested in this result can find a more recent and elegant demonstration in [Arb87], for instance.

The neural networks introduced by McCulloch and Pitts were assumed to be connected to an external tape. Neural networks that compute without recurring to a tape were shown to be Turing universal in [SS95].

Several alternative models and variants of neural networks have been proposed over the years, for a variety of different purposes. In 1996, Hava Siegelmann and Joe Killian wrote an article ([KS96]) on the simulation of Turing machines by analog recurrent neural networks (ARNN) with a sigmoidal activation function. These networks possess the interesting and useful property that the activation function of the neurons is analytical. It was the first time that a neural network with analytical update was proven to have the computational power of Turing machines. However, the authors, at several points of their work, merely sketched their demonstrations, giving the general ideas and letting their readers fill in the gaps. Quoting the referred article, the demonstration of the main result was only "sketched to a point at which we believe it will be clear enough to enable the reader to fill in the details".

Some scientists working in the field, and amongst them Cristopher Moore and Pascal Koiran, were not satisfied with the error analysis of that result. Moreover, Cristopher Moore and Pascal Koiran had a conjecture that contradicted this result. They conjectured that the simulation of Turing machines by an analytic function, or equivalently, by a recurrent neural network with analytic update should be impossible. This question was left has an open question in [MK99]:

Can Turing machines be simulated by an analytic function on a compact space? TMs generally have a countably infinite number of fixed points and periodic points of each period, while compact analytic maps can have only a finite or uncountable number. In one dimension this creates a contradiction, ... but such a simulation may still be possible in more than one dimension.

Another important fact concerning this kind of network was the result concerning the behavior of such networks in the presence of noise. This subject was addressed in 1998, both by [SR98] and [MS98]. In the presence of noise, the computational power of the networks falls to a level below the computational power of finite automata. This collapse of computational power gave even more plausibility to the hypotheses that there was an error in the paper, and the result might not be valid.

In 1999, the article was incorporated in the book [Sie99], but no reviewing was made, the article became the seventh chapter of the book almost unaltered.

Professor Félix Costa, a friend and collaborator of both Hava Siegelmann and Cris Moore, was told by Hava that there were doubts about the veracity of the result, raised by Cris. In the Spring of 2001, when I was looking for a subject for my diploma thesis, I first talked with professor Félix, who had been my teacher before, and he proposed me as a subject to work on this question.

The aim of this thesis is therefore very simple: solve the question of knowing whether or not ARNNs with a sigmoidal activation function have the computational power of Turing machines. In the affirmative case, obtain a clear and complete demonstration of that fact, and in the contrary case find out exactly where things fail, and why.

The problem at hand was hard, due to the heavy and cumbersome calculations and verifications necessary and to the lot of details that needed to be clarified. Although some alterations had to be made to the original paper, the result is valid, or at least so I hope to have proved with this thesis. If the result is not true, then I have improved proof-checking to almost 100%.

The results of this thesis will also appear in Computing Reviews briefly.

To finish this introduction, we would like to summarize the contents and organization of this thesis. In chapter 2 we present the background concepts to the reader unfamiliar with the concepts in the field of general Computer Science, necessary to understand the rest of the thesis. The reader familiar with Turing machines, stack machines and counter machines may simply skip this chapter.

In the third chapter we present the concepts introduced in [KS96], notably alarm clock machines with restless counters, which we shall simulate via neural networks, and the new type of automaton we shall use as intermediaries in order to prove our result, the adder machine.

In the fourth chapter we describe in detail the network whose universality we shall prove, and explain its functioning.

In the fifth chapter we prove that the network is in fact universal. Most of the demonstration consists in error analysis.

Finally, in the last chapter, we present a summary of the conclusions drawn during and after the elaboration of this thesis.

Chapter 2

Basic Definitions

Although the classical work of McCulloch and Pitts[MP43] on finite automata was concerned with computability by finite networks only, most of the models used over the years to study the computational power of recurrent neural networks assume networks of infinite size, i.e., with an infinite number of neurons.

Ralph Hartley and Harold Szu showed in [HS87] that Turing machines are computationally equivalent to countably infinite neural networks and also to finite networks of neurons with a countable infinity of states. Max Garzon (see [Gar95], chapter 6) used a model with a countable and locally-finite (each neuron can only be connected to a finite number of neurons, although the network may be infinite) number of finite-state neurons and studied the relationship between the computational power of these devices and that of countably infinite *cellular automata*. Wolpert and McLennan ([WM93]) studied the computational power of *field computers*, a device that can be viewed as a neural net continuous both in space and time, with an infinite, uncountable number of neurons. They have shown the existence of a universal network of this kind with a linear activation function.

An exception to this tendency was the result due to Pollack in [Pol87]. Pollack argued that a certain recurrent net model, which he called a *neuring machine*, is universal. The model in [Pol87] consisted of a finite number of neurons of two different kinds, having identity and threshold responses, respectively. Its machine was *high order*, that is, the activations were combined using multiplications as opposed to just linear combinations.

Pollack left as an open question whether high-order connections are really necessary in order to achieve universality, though he conjectured that they are.

> We believe however, that even when a more reasonable bound on analog values is imposed, multiplicative connections[1] remain a

[1]Equivalently, high-order neurons.

critical, and underappreciated, component for neurally-inspired computing.

High order networks were often used in applications, and one motivation often cited for the use of high-order nets was Pollack's conjecture regarding their superior computational power.

In [SS91] the simulation of Turing machines by ARNNs was the first simulation of Turing machines by neural networks using only a finite number of first-order neurons. Hava Siegelmann and Eduardo Sontag proved then the Turing universality of ARNNs with the saturated linear activation function:

$$\sigma(x) = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{otherwise.} \end{cases}$$

Pascal Koiran, in his PhD thesis ([Koi93]) extended the results in [SS91] to a larger class of functions[2]. However, this expansion did not include any analytic functions. Quoting [Koi93]:

> Le chapitre 4 généralise ce résultat[3] à une classe assez large de fonctions de sorties (qui n'inclut malheureusement pas la tangente hyperbolique, souvent utilisée en pratique, ni d'ailleurs aucune fonction analityque).[4]

so the question of knowing whether or not this result could be extended to analytical functions remained an open question.

The importance of the existence of universal sigmoidal neural networks and the reason why sigmoidal networks are so often used in practice, lies in two useful properties: it is a function that is biologically plausible, in contrast with the linear saturated function, and it also verifies the nice property that its derivative can be expressed in terms of the function, therefore it becomes very useful in backpropagation algorithms.

However, the simulation of Turing machines by sigmoidal neural networks raises problems and difficulties that do not occur when we use the saturated linear activation function. σ has one important property useful in the proof of Turing universality: within the unit interval, (0,1), it computes the identity function. This feature can be used to implement long term memory, in particular it can be used to store the contents of Turing machine tapes. With the sigmoid activation function,

[2]This class of functions consists of all functions ϕ that verify the following three conditions: $\phi(x) = 0$ for $x \leq 0$; $\phi(x) = 1$ for $x \geq 1$; and there is an open interval I such that ϕ is C^2 in I and $\forall x \in$ I, $\phi'(x) \neq 0$.

[3]Turing universality, a concept defined in section 2.2.

[4]In English: " Chapter 4 generalizes this result[3] to a very large class of output functions (which unfortunately doesn't include the hyperbolic tangent, often used in practice, nor any analytical function)".

$$\varrho\ (x) = \tfrac{2}{1+e^{-x}} - 1 = Tanh(\tfrac{x}{2})$$

it is no longer clear how to maintain long term memory stored in values of the neurons. Therefore, we will use a new type of automaton, which is Turing universal and does not rely on long term memory, the alarm clock machine, introduced in [KS96]. Once the alarm clock machine needs no long-term memory, it will be easier to simulate Turing machines via alarm clock machines than to do it directly.

In this chapter, following the same scheme used in [KS96], we introduce the concepts of Turing machine, k-stack machine, and k-counter machine. We prove that each new type of automaton introduced can simulate the previous one, thus establishing that k-counter machines are Turing universal. These are well-known results in automata theory and therefore this chapter is included in the thesis only in order to make it self-contained and to establish notation to be used throughout the whole thesis.

In the next chapter we will introduce adder machines and alarm clock machines, and prove their universality. We introduce also restless counters and see how to simulate alarm clock machines replacing the clocks by restless counters. The aim of this reduction is to facilitate the simulation of alarm clock machines because, as we will see in chapter 4, restless counters can be implemented by sigmoidal neurons. The complete series of simulations between different types of automata is shown in the diagram in figure 2.1.

$$\begin{tikzcd}
Turing\ machines \arrow[d] \\
stack\ machines \arrow[d] \\
counter\ machines \arrow[d] \\
adder\ machines \arrow[d] \\
alarm\ clock\ machines \arrow[d] \\
restless\ counters \arrow[d] \\
neural\ networks
\end{tikzcd}$$

Figure 2.1: Schema for the successive simulations to be proven throughout this thesis.

2.1 Turing machines and general definitions

We begin by introducing the concept of Turing machine. There are many computationally equivalent definitions of a Turing machine. First we will present a single tape model, and afterwards an equivalent[5] multi-tape model. We will also use this section to define a few concepts sufficiently general to be applied to distinct types of automata, and to establish notation to be used. The reader familiar with this notion may skip this section.

A Turing machine consists of a finite automaton, called the *finite control* of the machine, and a one-way infinite *tape*. The tape is divided in *cells* and is accessed by a *read-write head*. By one-way infinite tape, we mean that the tape has a leftmost cell, but there is an infinite number of cells to its right. When the head is on the leftmost cell, it is not allowed to move left.

Before the beginning of a *computation*, an input sequence w is written on the first cells of the tape, followed by an infinite number of blanks. The tape head is placed at the leftmost cell of the tape. The internal state is the initial state q_0.

At each moment, the device is in one of its states. At each step, the machine reads the tape symbol under the head, checks the state of the control and executes three operations: writes a new symbol into the current cell under the head of the tape, moves the head one position to the left or to the right, or makes no move, and finally changes the state of the control. We will denote the movement of tape head in the following manner: R means moving one cell to the right, N means not to move, and L means moving one cell to the left if there are cells to the left, otherwise do not move.

Whenever the transition function is undefined, the machine stops, i.e., it interprets the absence of instructions as a stop order. If given input w the machine computes eternally, then we say that the function computed by $\mathcal{M}$ is undefined at w. If not, the output of the computation is the sequence written on the tape when it stops.

We will now present the formal definitions:

Definition 2.1.1 An *alphabet* is any non-empty finite set, including the blank symbol B, but not the symbols §, \$, and #.[6]

Definition 2.1.2 A *single tape*[7] *Turing machine is a 4-tuple* $\mathcal{M} =<Q, \Sigma, \delta, q_0>$[8]*, where :*

[5]Equivalent only in the sense that the classes of computable functions are the same. From the point of view of complexity classes, these models are not equivalent at all. See also page 18.

[6]The reasons for demanding that §, \$, and # cannot be symbols of our alphabet are merely notational issues and will become clear with definition 2.1.4 and the introduction of endmarkers, on page 16.

[7]In opposition to multi-tape Turing machine, definition 2.1.10. Whenever we refer simply to Turing machines, we mean single tape Turing machines.

[8]We will be concerned with Turing machines as devices for computing functions, not

- *Q is a non-empty finite set (of states),*
- *Σ is the tape alphabet (Q and Σ are assumed to be disjoint, to avoid confusion),*
- *$\delta : Q \times \Sigma \to Q \times \Sigma \times \{\mathsf{L}, \mathsf{N}, \mathsf{R}\}$, is a partial function, called the transition function,*
- *$q_0 \in Q$ is the initial state*

Definition 2.1.3 A *tape* for a Turing machine $\mathcal{M} =<Q, \Sigma, \delta, q_0>$ is a total map $f : \mathbb{N} \to \Sigma$ such that the set $\{n : n \in \mathbb{N}$ and $f(n) \neq \mathsf{B}\}$ is finite. The elements of the domain of f are denoted *cells*; $n \in \mathbb{N}$ is the n-th cell of the tape.

Situations in which the transition function is undefined indicate that the computation must stop. Otherwise the result of the transition function is interpreted as follows: the first component is the new state; the second component is the symbol to be written on the scanned cell; the third component specifies the movement of the tape head according to the convention earlier described.

Definition 2.1.4 Given a Turing machine $\mathcal{M}$, an *instantaneous description* of $\mathcal{M}$, also called a *configuration*, is a pair (q, x), where q is the current state of $\mathcal{M}$, and $x \in \Sigma^*\#\Sigma^*$ represents the current contents of the tape. The symbol $\#$ is supposed not to be in Σ, and marks the position of the tape head: by convention, the head scans the symbol immediately at the right of $\#$. All the symbols in the infinite tape not appearing in x are assumed to be the blank symbol B[9].

Definition 2.1.5 The *initial configuration* of a Turing machine $\mathcal{M}$ on an input w is $(q_0, \#w)$.

We will now define a few concepts that we wish to apply not only to Turing machines, but also to any other kind of machines we may introduce throughout this thesis, as long as we provide adequate notions of transition function and configuration and specify the initial configuration.

as language recognizers. Therefore, we have chosen not to include a set of final states in the definition of Turing machine, because its existence is not necessary. This remark is valid to the other kinds of automata considered in this thesis.

[9]We point out the fact that there is more than one possible instantaneous description for a Turing machine at a given moment. Usually, the blank symbols to the right of the rightmost non-blank symbol are all omitted, but this definition does not force us to follow this rule. In fact, the sequence to the right of # may contain any finite number of these "useless" blank symbols. In the examples given we have chosen sometimes to maintain them, for the sake of an easier understanding.

Definition 2.1.6 Given a machine $\mathcal{M}$, a *partial computation* of $\mathcal{M}$ is a sequence (possibly infinite) of configurations of $\mathcal{M}$, in which each step from a configuration to the next obeys the transition function. Given also an input sequence w, the *computation of $\mathcal{M}$ on w* is a partial computation which starts with the initial configuration of $\mathcal{M}$ on w, and either is infinite, or ends in a configuration in which no more steps can be performed.

In order to have Turing machines performing numerical computations, it is necessary that we introduce a symbolic representation for numbers. Once we have chosen a way to encode the natural numbers into an input sequence and to decode the final contents of the output tape back into the natural numbers, a function f will be said to be computable if its values can be computed by some Turing machine whose tape is initially blank except for the coding of its arguments $m_1, ..., m_n$. The value of $f(m_1, ..., m_n)$ is the decoding of what remains on the output tape when the machine stops.

There are several possible ways to encode the input and the output of a Turing machine, however the coding function γ must be injective, otherwise we cannot design Turing machines to compute injective functions, and the decoding function γ' must be surjective, otherwise we cannot represent functions that return values outside the range of γ'. Once more, this notion can be defined in a broader sense, so that it may apply not only to Turing machines, but to any kind of machines provided with a notion of computation and a convention for the inputs and outputs.

Definition 2.1.7 Let $\mathcal{M}$ be a machine with alphabet Σ, and $\gamma : \mathbb{N}^{*}$[10] $\to \Sigma^*$, $\gamma' : \Sigma^* \to \mathbb{N}$ two previously chosen functions, called coding function and decoding function, respectively, such that γ is injective and γ' is surjective. For each $n \in \mathbb{N}$, the n-ary function computed by $\mathcal{M}$, $\phi^n_{\mathcal{M}}$, is defined in such a way that for every $m_1, ..., m_n \in \mathbb{N}$, $\phi^n_{\mathcal{M}}(m_1, ..., m_n)$ equals the natural number obtained by decoding $\mathcal{M}$'s output when it stops at the end of the computation of $\mathcal{M}$ on $\gamma(m_1, ..., m_n)$, or is undefined if this computation never stops. For every n-ary function f, if there is an $\mathcal{M}$ such that $\phi^n_{\mathcal{M}} = f$, f is said to be partially computable. In this case we say that $\mathcal{M}$ computes f. If f is also a total function, then f is said to be computable.

In the specific case of Turing machines, we will encode the inputs by choosing one symbol as basic and we will denote a number by an expression consisting entirely of occurrences of this symbol. We will represent each $n \in \mathbb{N}$ by a sequence of $n+1$ 1's (the extra 1 is to allow us to distinguish between zero and the empty tape). To codify a tuple of natural numbers we will simply separate each one by a blank symbol. We will therefore assume the coding function γ defined by $\gamma(m_1, m_2, ..., m_2) = 1^{(m_1+1)}\mathsf{B}1^{(m_2+1)}\mathsf{B}...\mathsf{B}1^{(m_n+1)}$. As for the decoding function γ', we will define it simply by $\gamma'(w) =$ number

[10]By $\mathbb{N}^*$ we mean, naturally, the set of finite sequences of natural numbers.

of 1's in w, for every $w \in \Sigma^*$. Trivially, these functions satisfy the required properties.

Definition 2.1.8 Let $\mathbf{M}$ be a class of machines. The class of functions computed by $\mathbf{M}$ is the set $\{\phi^n_{\mathcal{M}} \ ; \ \mathcal{M} \in \mathbf{M}$ and $n \in \mathbb{N}\}$.

The class of functions computed by Turing machines is the class of partial recursive functions of Kleene. For a full discussion of this class of functions, see [Min67], chapter 10.

Definition 2.1.9 Any Turing machine $\mathcal{M}$ can be encoded by a natural number $\tau(\mathcal{M})$, using techniques developed by Gödel and Turing. This process is known as gödelization of Turing machines, and the number corresponding to each machine is its Gödel number. The Turing machine $\mathcal{U}$ is said to be universal for n-ary functions if $\phi^{n+1}_{\mathcal{U}}(\tau(\mathcal{M}), m_1, ..., m_n) = \phi^n_{\mathcal{M}}(m_1, ..., m_n)$, for every Turing machine $\mathcal{M}$ and every $m_1, ..., m_n \in \mathbb{N}$.

It is a classical result of Computer Science, proven in [Sha56], that there is a universal Turing machine with an alphabet of only two symbols[11], one of them being the blank symbol. Therefore we may assume the alphabet $\Sigma = \{1, \mathsf{B}\}$ without any loss in computational power. From now on, we will represent Turing machines as 3-tuples, and adopt the convention that Σ is $\{1, \mathsf{B}\}$, unless otherwise indicated.

The following examples are adapted from [Dav82].

Example 2.1.1 Let us consider as an example the single tape Turing machine $\mathcal{M} =<\{q_0, q_1, q_2\}, \delta, q_0\}>$, with δ given by the table:

δ	1	B
q_0	q_1, B, R	$-$
q_1	$q_1, 1, R$	q_2, B, R
q_2	q_2, B, N	$-$

Its graph is shown in figure 2.2. The meaning of the 3-tuples shown in the figure is the following: the first element is the symbol read, the second one the symbol written and the last one the movement of the tape head.

This Turing machine computes the function $\lambda xy.x + y$. Given the input $w = 11^m B 11^n$, the computation performed by the machine is:

[11] A classical result concerning small universal Turing machines is the existence of a universal Turing machine (for unary functions) with only 4 symbols and 7 states, proven in [Min67]. Recently, Yurii Rogozhin showed in [Rog96] the existence of universal Turing machines with the following number of state-symbols: (24,2),(10,3),(7,4),(5,5),(3,10) and (2,18).

$(q_0, \#11^m B11^n)$
$(q_1, B\#1^m B11^n)$
$(q_1, B1\#1^{m-1} B11^n)$
$(q_1, ...)$
$(q_1, B1^m \# B11^n)$
$(q_2, B1^m B\#11^n)$
$(q_2, B1^m BB\#1^n)$

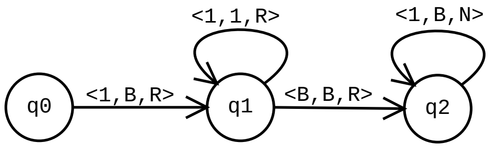

Figure 2.2: Graph for the Turing machine of example 2.1.1.

Example 2.1.2 Let us consider as an example the single tape Turing machine $\mathcal{M} = <\{q_0, q_1, q_2\}, \delta, q_0\}>$, with δ given by the table:

δ	1	B
q_0	q_1, B, R	$-$
q_1	$q_1, 1, R$	q_2, B, R
q_2	$q_2, 1, R$	q_3, B, L
q_3	$-$	q_4, B, L
q_4	$-$	$q_5, 1, L$
q_5	q_6, B, L	$q_5, 1, L$
q_6	q_8, B, R	$q_7, 1, L$
q_7	q_1, B, R	$q_7, 1, L$
q_8	q_8, B, R	$q_8, 1, L$

This Turing machine computes the function $\lambda xy.x - y$. This is a partial function defined on the naturals with the result $x - y$, if $x \geq y$ and undefined elsewhere. The graph for this machine is shown in figure 2.3. Given the input $w = 11^m B11^n$, the computation performed by the machine is:

$(q_0, \#11^m B11^n)$
$(q_1, B\#1^m B11^n)$
$(q_1, B1\#1^{m-1} B11^n)$
$(q_1, ...)$
$(q_1, B1^m \# B11^n)$
$(q_2, B1^m B\#11^n)$
$(q_2, B1^m B1\#1^n)$
$(q_2, ...)$
$(q_2, B1^m B1^{n+1} \#)$

($q_3, B1^mB1^n\#1$)
($q_4, B1^mB1^{n-1}\#1B$)
($q_5, B1^mB1^{n-2}\#11B$)
($q_5, B1^mB1^{n-3}\#111B$)
($q_5, ...$)
($q_5, B1^m\#B1^nB$)
($q_6, B1^{m-1}\#1B1^nB$)
($q_7, B1^{m-2}\#11B1^nB$)
($q_7, B1^{m-3}\#111B1^nB$)
($q_7, ...$)
($q_7, \#B1^mB1^nB$)
($q_0, B\#1^mB1^nB$)

Now, except for the initial and the final 1's we are back to the beginning. The process will be repeated and one of two things will happen, depending on whether $m \geq n$ or $m < n$.

If $m \geq n$, then we will have

($q_0, \#1^{m+1}B1^{n+1}$)
...
($q_0, B\#1^mB1^nB$)
...
($q_0, BB\#1^{m-1}B1^{n-1}BB$)
...
($q_0, B^n\#1^{m-n+1}B1B^n$)
...
($q_1, B^{n+1}1^{m-n}\#B1B^n$)
...
($q_3, B^{n+1}1^{m-n}B\#1B^n$)
($q_4, B^{n+1}1^{m-n}\#BB^{n+1}$)

At this point the machine halts and the output is m-n. If, on the other hand, $m < n$, then:

($q_0, \#1^{m+1}B1^{n+1}$)
...
($q_0, B\#1^mB1^nB$)
...
($q_0, BB\#1^{m-1}B1^{n-1}BB$)
...
($q_0, B^m\#1B1^{n-m+1}B^m$)
($q_1, B^{m+1}\#B1^{n-m+1}B^m$)
...
($q_2, B^{m+2}\#1^{n-m+1}B^m$)
...

($q_3, B^{m+2}1^{n-m}\#1B^m$)
($q_4, B^{m+2}1^{n-m-1}\#1B^{m+1}$)
...
($q_5, B^{m+1}\#B1^{n-m}B^{m+1}$)
($q_6, B^m\#BB1^{n-m}B^{m+1}$)
($q_8, B^{m+1}\#B1^{n-m}B^{m+1}$)
($q_8, B^{m+2}\#1^{n-m}B^{m+1}$)
($q_8, B^{m+1}\#B1^{n-m}B^{m+1}$)
($q_8, B^{m+2}\#1^{n-m}B^{m+1}$)
($q_8, B^{m+1}\#B1^{n-m}B^{m+1}$)
...

At this point, the last two instantaneous descriptions shown are transformed back and forth into each other, causing the computation to go on indefinitely. We conclude that the machine behaves as desired.

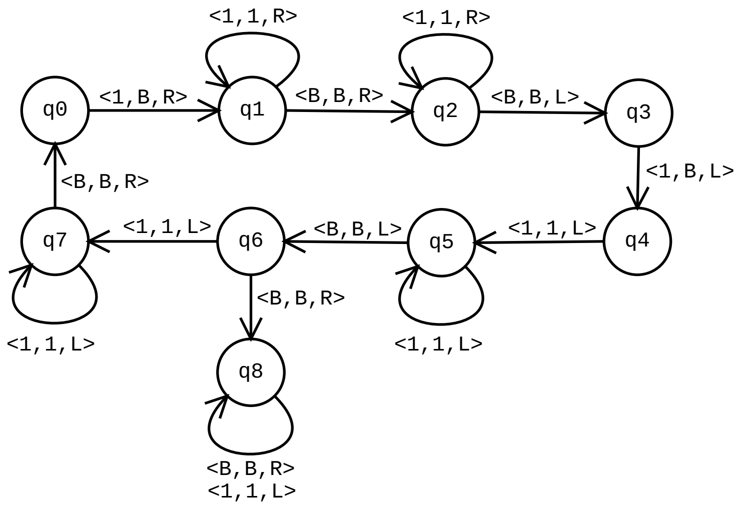

Figure 2.3: Graph for the Turing machine of example 2.1.2.

Example 2.1.3 Now we present an example of a Turing machine with a 4-symbol alphabet $\Sigma = \{0, 1, \epsilon, \eta\}$, that computes function $\lambda mn.(m+1) \times (n+1)$. $\mathcal{M}$ has 10 states, its graph can be seen in figure 2.4, and we will begin to describe its functioning.

Given arguments m and n as input, the machines' behavior is the following:

$\delta(q_0, 1) = (q_1, B, R)$	(erase a single 1, leaving m 1's to be counted)
$\delta(q_1, 1) = (q_2, \epsilon, R)$	(if the m 1's have all been counted, stop; otherwise count another one of them)

$\delta(q_2, 1) = (q_2, 1, R)$
$\delta(q_2, B) = (q_3, B, R)$

$\delta(q_3, 1) = (q_2, 1, R)$	(go right until a double blank is reached)
$\delta(q_3, B) = (q_4, B, L)$	
$\delta(q_4, 1) = (q_5, 1, L)$	
$\delta(q_4, B) = (q_5, B, L)$	
$\delta(q_5, 1) = (q_6, \eta, R)$	(count another 1 from a group of $n + 1$ 1's)
$\delta(q_5, B) = (q_9, B, N)$	($n + 1$ 1's have been counted; prepare to repeat the process)
$\delta(q_6, 1) = (q_6, 1, R)$	
$\delta(q_6, B) = (q_7, B, R)$	
$\delta(q_7, 1) = (q_7, 1, R)$	
$\delta(q_7, B) = (q_8, 1, N)$	(write another 1 to correspond to the 1 that has just been counted)
$\delta(q_8, 1) = (q_8, 1, L)$	
$\delta(q_8, B) = (q_8, B, L)$	(go left until η has been reached; prepare to count again)
$\delta(q_8, \eta) = (q_4, 1, N)$	
$\delta(q_9, 1) = (q_9, 1, L)$	
$\delta(q_9, B) = (q_9, 1, L)$	(go left until ϵ is reached)
$\delta(q_9, \epsilon) = (q_0, B, N)$	

Now we introduce the notion of *k-tape Turing machine.* The difference to the previous version is that the machine has a finite number k of tapes, $k \geq 2$. Each tape is accessed by its own read-write head. The first tape is called the *input tape*, the second tape is called the *output tape*, and all the other tapes are called *work tapes.* In terms of computational power this model is equivalent to the single tape one, so there is no advantage in computational power. The reason for presenting this kind of machine is that some of the machines considered in the next sections are special cases (or can be regarded as such) of k-tape Turing machines.

At the beginning of a computation, the input sequence w is written on the input tape, surrounded by endmarkers, § before the sequence, and \$ after. The Turing machine is not allowed to move the input tape head beyond these endmarkers. All the cells of the other tapes are blank, and all the tape heads are placed at the leftmost cell of each tape.

At each step, the machine reads the tape symbols under all the heads, checks the state of the control and executes three operations: writes a new symbol into the current cell under the head of each of the tapes, moves each head one position to the left or to the right, or makes no move, and finally

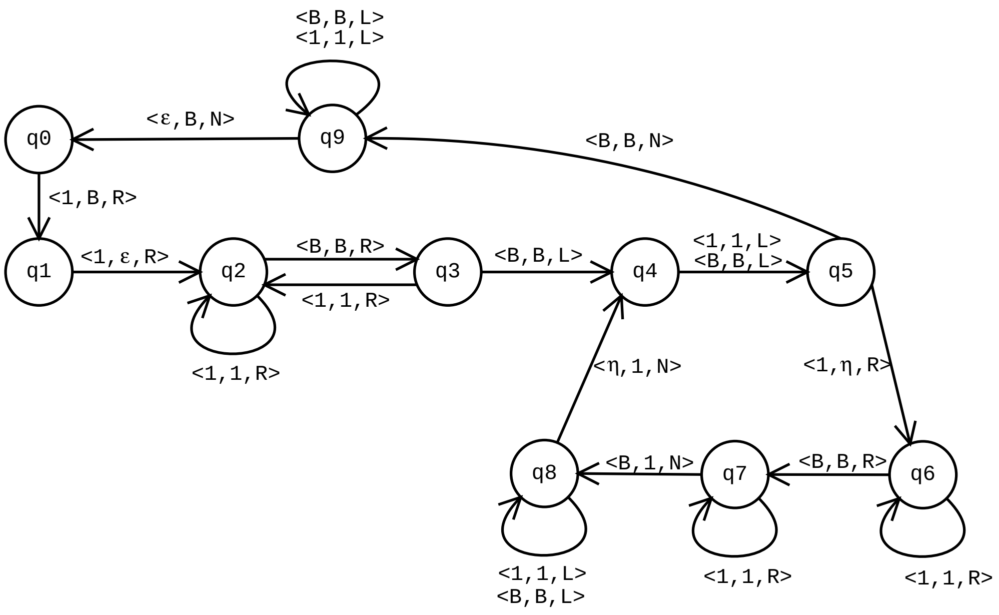

Figure 2.4: Graph for the Turing machine of example 2.1.3.

changes the state of the control. There are, however, two restrictions: the input tape cannot be written to, and the head of the output tape can only write on it and move one cell to the right after writing. For k-tape Turing machines, the output is the number of 1's written on the output tape when it halts.

In all other aspects, the machine performs exactly as the single tape version. In the remaining definitions of this section, k is implicitly assumed to be greater than 1.

Definition 2.1.10 A k-tape *Turing machine* is a 4-tuple $\mathcal{M} = <Q, \Sigma, \delta, q_0>$, where :

- Q is a non-empty finite set (of *states*),
- Σ is the tape alphabet,
- $\delta : Q \times \Sigma^{k-1} \to Q \times \Sigma^{k-1} \times \{\mathsf{L}, \mathsf{N}, \mathsf{R}\}^{k-1} \times \{\mathsf{N}, \mathsf{R}\}$, is a partial function, called the *transition function*,
- $q_0 \in Q$ is the *initial state*.

The result of the transition function consists on: the new state, the $k-1$ symbols to be written on the work tapes and the output tape, the $k-1$ movements to be performed on the input tape and the work tapes, and finally the movement on the output tape.

The alterations to the previous definitions given for the single tape model are straightforward. We state the adapted definitions 2.1.4 and 2.1.5. All the others remain unchanged. By multi-tape Turing machine we will mean a k-tape Turing machine, with k unspecified.

Definition 2.1.11 Given a k-tape Turing machine $\mathcal{M}$, an *instantaneous description* of $\mathcal{M}$, also called a *configuration*, is a $k+1$ tuple

$$(q, x_1, x_2, ..., x_{k-1}, x_k)$$

where q is the current state of $\mathcal{M}$, and each $x_j \in \Sigma^*\#\Sigma^*$ represents the current contents of the j^{th} tape.

Definition 2.1.12 The *initial configuration* of a k-tape Turing machine $\mathcal{M}$ on an input w is $(q_0, \#w, \#, ..., \#)$.

Proposition 2.1.1 The single tape model and the multi-tape model are equivalent in the sense that the class of computable functions by one and by the other model are the same.

This result is well-known and we will not prove it here. The reader who wishes may find a proof in any Theoretical Computer Science introductory book, such as [HU01] or [FB94].

By reducing the number of tapes, we cause a slowdown in the computation time, that is, in the number of steps necessary to perform a computation. For every $k \in \mathbb{N}$ and every k-tape Turing machine $\mathcal{M}$ working in time T, there is a single-tape Turing machine $\mathcal{M}'$ working in time $\mathrm{O}(\mathrm{T}\cdot\log(\mathrm{T}))$ such that $\phi^n_{\mathcal{M}} = \phi^n_{\mathcal{M}'}$ for every $n \in \mathbb{N}$.

2.2 Stack machines

In this section we present the concept of p-stack machine and prove that a 2-stack machine can simulate an arbitrary Turing machine. A stack machine consists of a finite control, an input tape, an output tape and p stacks. Its functioning is similar to that of a Turing machine. On each step, the machine reads the symbol under the input tape head, and also the symbols on the top of the stacks. If a stack is empty, the machine may also use that information to decide what to do next. As for the stacks, the actions performed by the stack machine are adding a symbol to the top of the stack (Push(symbol)), removing a symbol from the top of a stack (Pop), or do nothing.

A p-stack machine may also be defined as a $(p+2)$-tape Turing machine. Each stack corresponds to a work tape, with the extra 2 tapes being for input and output. We must add to the definition of Turing machine the restrictions that for all its work tapes, every time a head moves left on either tape, a blank is printed on that tape just before the move. This way we guarantee that the read-write head is always scanning the rightmost non-blank symbol, and we can think of it as the top of the stack. Writing a new symbol on the stack and moving right corresponds to pushing the symbol, erasing a symbol and moving left corresponds to popping.

Definition 2.2.1 A *p-stack machine* is a 4-tuple $\mathcal{A} = <Q, \Sigma, \delta, q_0>$, where Q, Σ, and q_0 are defined as for Turing machines, and the transition function δ is defined as a partial function

$$\delta : Q \times (\Sigma \cup \{\epsilon\})^{p+1} \to Q \times \{\mathsf{L}, \mathsf{N}, \mathsf{R}\} \times (\Sigma \cup \{\mathsf{N}\}) \times (\{\mathsf{N}, \mathsf{Pop}\} \cup \{\mathsf{Push}(\sigma) : \sigma \in \Sigma\})^p.$$

The interpretation of the transition function for the stack machine is simple. The stack machine computes its output from the elements on top of each stack (or the information that the stack is empty), as well as the value in the input tape currently scanned and the present state. The output of δ consists on the new state, the movement of the input tape head, the symbol to write in the output tape (or N for none), and the operations to be performed on the stacks. Just as we have done for Turing machines, we assume Σ to be $\{1, B\}$ and denote each machine by its remaining 3 components, unless otherwise explicitly mentioned.

Definition 2.2.2 We denote an *instantaneous description* of a p-stack machine $\mathcal{A}$ by $(q, \alpha_1 \# \alpha_2, \alpha_3, s_1, s_2, ..., s_n)$. $\alpha_1, \alpha_2, \alpha_3, s_1, ..., s_n \in \Sigma^*$. q represents the current state, $\alpha_1 \# \alpha_2$ represents the contents of the input tape and the position of the tape head, α_3 represents the contents of the output tape. $s_1, s_2, ..., s_n$ represent the sequences stored in the stacks. We will represent the empty sequence by ϵ.

Definition 2.2.3 The *initial configuration* of a p-stack machine $\mathcal{A}$ on an input w is $(q_0, \#w, \epsilon, \epsilon, ..., \epsilon)$.

We will assume for k-stack machines the same input and output conventions that we have assumed for Turing machines, on page 11, therefore the following definition arises naturally.

Definition 2.2.4 We denote by $\mathcal{A}[k]$ the class of functions that can be computed by a k-stack machine[12], i.e., given a function $f : \mathbb{N}^n \to \mathbb{N}$, $f \in \mathcal{A}[k]$ iff there exists a k-stack machine $\mathcal{M}$ such that $\phi^n_{\mathcal{M}} = f$.[13] We denote by $\mathcal{A}[k](T)$ the *class of functions that can be computed by a k-stack machine in time T.*

Example 2.2.1 Let us see an example of a 1-stack machine $\mathcal{A}$ designed in such a way that $\phi^2_{\mathcal{A}} = (\lambda xy.x+y)$. This machine has 7 states (Q=$\{q_0, ..., q_6\}$) and the usual alphabet. The transition function is given by:

[12] Recall definition 2.1.7.

[13] It is possible to simulate 3 stacks with only 2, therefore $\mathcal{A}[2] = \mathcal{A}[n], \forall\, n \geq 2$.

$\delta(q_0, 1, \_) = (q_0, R, N, N)$,	(we move the head of the input tape to the final endmarker)
$\delta(q_0, B, \_) = (q_0, R, N, N)$,	
$\delta(q_0, \$, \_) = (q_1, L, N, N)$,	
$\delta(q_1, 1, \_) = (q_1, L, N, \mathsf{Push}(1))$,	(now we move the head of the input tape backwards, and start putting the symbols read into one stack)
$\delta(q_1, B, \_) = (q_1, L, N, \mathsf{Push}(B))$,	
$\delta(q_1, \S, \_) = (q_2, R, N, N)$,	
$\delta(q_2, \_, \_) = (q_3, N, N, \mathsf{Pop})$,	(we pop the exceeding 1)
$\delta(q_3, \_, 1) = (q_3, N, 1, \mathsf{Pop})$,	(we pop the 1's that codify the first argument, and write the corresponding 1's on the output tape, until we find the blank separating the arguments)
$\delta(q_3, \_, B) = (q_4, N, N, \mathsf{Pop})$,	
$\delta(q_4, \_, \_) = (q_5, N, N, \mathsf{Pop})$,	(for the second argument, once again we pop the extra 1 before continuing)
$\delta(q_5, \_, 1) = (q_5, N, 1, \mathsf{Pop})$,	(finally we proceed as for the first argument; once the stack is empty, we change into a state without transitions and stop)
$\delta(q_5, \_, B) = (q_5, N, 1, \mathsf{Pop})$,	
$\delta(q_5, \_, \epsilon) = (q_6, N, N, N)$	

In figure 2.5, we can see a graph for that machine. The meaning of the 5-tuples on the arrows is the following: the first value is the symbol read on the input tape; the second one the value on the top of the stack; the third the movement on the input tape; the fourth the action on the output tape; and the last one the operation on the stack. An underscore represents all possible values for that component of the tuple.

We will now show that for any given Turing machine $\mathcal{M}$, there is a 2-stack machine $\mathcal{A}$ that simulates $\mathcal{M}$, i.e., $\forall\, n \in \mathbb{N}\ \phi^n_{\mathcal{A}} = \phi^n_{\mathcal{M}}$. In particular, if $\mathcal{M}$ is a universal Turing machine for functions of arity n, then there exists a 2-stack machine $\mathcal{A}$ such that is Turing universal, in the sense that for every Turing machine $\mathcal{M}'$, $\phi^{n+1}_{\mathcal{A}}(\tau(\mathcal{M}'), m_1, ..., m_n) = \phi^n_{\mathcal{M}'}(m_1, ..., m_n)$. This proposition can be found in [HU01], Theorem 8.13.

Proposition 2.2.1 An arbitrary Turing machine can be simulated by a 2-stack machine.

Proof:

To simulate the computation of a single tape Turing machine, we will store the symbols to the left of the read-write head on one stack, while the symbols to the right of the head, including the symbol being read, will be placed on the other stack. On each stack, we will place the symbols closer to the Turing machine's head, closer to the top of the stack.

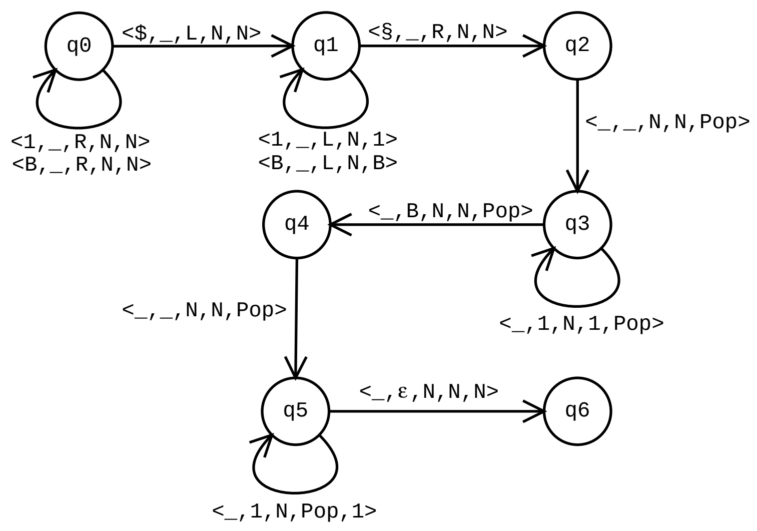

Figure 2.5: Graph for the 1-stack machine of example 2.2.1.

Let $\mathcal{M} =< \{q_0, q_1, ..., q_n\}, \delta, q_0 >$ be a single tape Turing machine.

We simulate $\mathcal{M}$ with a 2-stack machine in the following manner.

We initialize our 2-stack machine $\mathcal{A}$ moving right along the input tape until we reach the end of the input tape, \$. After reaching \$, the stack machine starts moving to the left, pushing the elements of the input tape into the second stack, until it reaches the beginning of the tape again. At this point, the first stack is empty and the contents of the input tape is stored on the second stack, and we are now ready to start simulating the moves of $\mathcal{M}$. This initialization requires that $\mathcal{A}$ must have 2 new states q_a and q_b, not states of $\mathcal{M}$, to sweep the input tape in both directions.

If $\mathcal{M}$ overwrites symbol s with symbol s', changes into state q and moves right, then what we want $\mathcal{A}$ to do is simply to remove s from stack$_2$ and to push s' into stack$_1$. This can be done immediately by performing Pop on stack$_2$ and Push(s') on stack$_1$.

If, on the other hand, $\mathcal{M}$ moves left, then we want to replace s with s' in stack$_2$, and we want to transfer the top of stack$_1$ to stack$_2$. We must do this operation in three steps: first, we pop stack$_2$ and push s' into stack$_1$ (otherwise the value of s' could be lost); second we push s' into stack$_2$ and pop stack$_1$; third, we pop stack1 again and push its top into stack$_2$. This forces us to have 3 different states in $\mathcal{A}$ for every state of $\mathcal{M}$.

At any time step the symbols of the first stack are the symbols to the left of $\mathcal{M}$'s control and the symbols to its right are those of the second stack; furthermore: the closer they are to the top of the stack, the closer they are to the tape head.

In particular, the symbol read by $\mathcal{M}$ is the top of the second stack.

When $\mathcal{M}$ halts, the contents of its tape is represented in the contents of both stacks. Therefore we need another additional 2 states for $\mathcal{A}$. When $\mathcal{M}$ halts, $\mathcal{A}$ changes its state to q_c and starts transferring the contents of stack$_2$ into stack$_1$. Once stack$_2$ is empty, $\mathcal{A}$ changes its state to q_d and starts transferring the contents of stack$_1$ to the output tape. When stack$_1$ becomes empty, $\mathcal{A}$ halts, because the contents of its output tape equals the contents of $\mathcal{M}$'s tape.

It should be obvious from the discussion above that
$\mathcal{A} =< \{q_a, q_b, q_c, q_d, q_0, q_1, ..., q_n, q_0', ..., q_n', q_0'', ..., q_n''\}, \delta', q_a >$ can simulate $\mathcal{M}$, if we define:

$\delta'(q_a, i, s_1, s_2) = (q_a, \mathsf{R}, \mathsf{N}, \mathsf{N}, \mathsf{N})$, if $i \neq \$$
$\delta'(q_a, \$, s_1, s_2) = (q_b, \mathsf{L}, \mathsf{N}, \mathsf{N}, \mathsf{N})$
$\delta'(q_b, i, s_1, s_2) = (q_b, \mathsf{L}, \mathsf{N}, \mathsf{N}, \mathsf{Push}(i))$, if $i \neq \S$
$\delta'(q_b, \S, s_1, s_2) = (q_0, \mathsf{R}, \mathsf{N}, \mathsf{N}, \mathsf{N})$
$\delta'(q_k, i, s_1, s_2) = ((\delta(q_k, i))_1, \mathsf{N}, \mathsf{N}, \mathsf{Push}((\delta(q_k, i))_2), \mathsf{Pop})$,
if $(\delta(q_k, i))_3 = \mathsf{R}$
$\delta'(q_k, i, s_1, s_2) = ((\delta(q_k, i))_1'', \mathsf{N}, \mathsf{N}, \mathsf{Push}((\delta(q_k, i))_2), \mathsf{Pop})$,
if $(\delta(q_k, i))_3 = \mathsf{L}$
$\delta'(q_k'', i, s_1, s_2) = (q_k', \mathsf{N}, \mathsf{N}, \mathsf{Pop}, \mathsf{Push}(s_1))$
$\delta'(q_k', i, s_1, s_2) = (q_k, \mathsf{N}, \mathsf{N}, \mathsf{Pop}, \mathsf{Push}(s_1))$
$\delta'(q_k, i, s_1, s_2) = (q_c, \mathsf{N}, \mathsf{N}, \mathsf{N}, \mathsf{N})$, if $\delta(q_k, i)$ is undefined
$\delta'(q_c, i, s_1, s_2) = (q_c, \mathsf{N}, \mathsf{N}, \mathsf{Push}(s_2), \mathsf{Pop})$
$\delta'(q_c, i, s_1, \epsilon) = (q_d, \mathsf{N}, \mathsf{N}, \mathsf{N}, \mathsf{N})$
$\delta'(q_d, i, s_1, s_2) = (q_d, \mathsf{N}, s_1, \mathsf{Pop}, \mathsf{N})$,

where $k \in \{0, ..., n\}$, $i \in \{1, \mathsf{B}, \S, \$\}$, $s_1, s_2 \in \{1, \mathsf{B}\}$.

□

We conclude from this result that there is a Turing universal 2-stack machine.

From the proof of this result, we can easily see that it is possible to store the input and output of the stack machine in the stacks, i.e., we can define a p-stack machine without input and output tapes, with input/output conventions that the input to the machine is the initial contents of the first stack instead of the initial contents of an input tape, and its output is the contents of the same stack when the machine halts. The formal definition is merely a reformulation of definition 2.2.1.

Definition 2.2.5 A *tapeless p-stack machine* is a 3-tuple $\mathcal{A} =< Q, \delta, q_0 >$, where Q and q_0 are a set of states and an initial state, and the transition function δ is defined as a partial function

$$\delta : Q \times (\Sigma \cup \{\epsilon\})^p \to Q \times \{\mathsf{N}, \mathsf{Pop}, \mathsf{Push(B)}, \mathsf{Push(1)}\}^p.$$

All the previous definitions and results can be adapted trivially to tapeless stack machines. We therefore conclude that we don't need input and output tapes to simulate a Turing machine with a 2-stack machine.

2.3 Counter machines

A k-counter machine consists of a finite control, an input tape, an output tape, and a finite number k, of counters. At each step, the machine's behavior is determined by the current state, the symbol under the input tape head, and the result of testing the counters for zero. The operations on the counters, besides the test for 0, are increase one unit, I, decrease one unit, D, and do nothing N.

A k-counter machine may be defined as a $(k+2)$-tape Turing machine, with the restrictions that all its work tapes are read-only, and all the symbols on the work tapes are blank, except the leftmost symbol. In each of the work tapes, the read head can only move towards left or right, and read the symbol under it. Since all the symbols are blank except for the first, the only thing we can store on the tape is an integer i, by moving the tape head i cells to the right of the leftmost cell. Therefore, we have operations increase, decrease and test for zero (read the cell and see if the symbol stored in it is 1), but we have no way to directly compare two counters, or to know instantly the value of a counter.

An instantaneous description of a counter machine can be given by the state, the input tape contents, the position of the input head, the output tape contents, and the distance of the storage heads to the leftmost cell. We call these distances the *counts* of the tapes.

Definition 2.3.1 A *k-counter machine* is a 4-tuple $\mathcal{C} = <Q, \Sigma, \delta, q_0>$, where Q, Σ, and q_0 are defined as for Turing machines, and the transition function δ is defined as a partial function

$$\delta : Q \times \Sigma \times \{True, False\}^k \rightarrow Q \times \{\mathsf{L}, \mathsf{N}, \mathsf{R}\} \times (\Sigma \cup \{\mathsf{N}\}) \times \{\mathsf{I}, \mathsf{D}, \mathsf{N}\}^k.$$

The counter machine computes its output from the counts, as well as the value in the input tape currently scanned and the present state. The output of δ consists on the new state, the movement of the input tape head, the symbol to write in the output tape (or N for none), and the operations to be performed on the counters.

Definition 2.3.2 We denote an *instantaneous description* of a k-counter machine $\mathcal{C}$ by $(q, \alpha_1 \# \alpha_2, \alpha_3, c_1, c_2, ..., c_n)$. $\alpha_1, \alpha_2, \alpha_3 \in \Sigma^*$ and $c_1, c_2, ..., c_n \in \mathbb{N}$. q represents the current state, $\alpha_1 \# \alpha_2$ represents the contents of the input tape and the position of the tape head, α_3 represents the contents of the output tape. $c_1, c_2, ..., c_n$ represent the counts stored by the machine.

Definition 2.3.3 The *initial configuration* of a k-counter machine $\mathcal{C}$ on an input w is $(q_0, \#w, \epsilon, 0, 0, ..., 0)$.

Definition 2.3.4 We denote by $\mathcal{C}[k]$ the *class of functions that can be computed by a k-counter machine*[14], i.e., given a function $f : \mathbb{N}^n \to \mathbb{N}$, $f \in \mathcal{C}[k]$ iff there exists a k-counter machine $\mathcal{M}$ such that $\phi^n_{\mathcal{M}} = f$.[15] We denote by $\mathcal{C}[k](T)$ the *class of functions that can be computed by a k-counter machine in time T.*

Example 2.3.1 Let us consider a 4-counter machine to compute the sum of two integers. In figure 2.6 we can see the graph for this machine. We could design a much simpler counter machine to perform this task, but we believe this example may help to understand the proof of proposition 2.3.1.

Before analyzing this machine, let us see how to interpret the information in figure 2.3.1. Instead of the ordinary representation of tuples, we indicate only the operations performed on the counters ($\mathsf{I}n$ denotes increase counter n and $\mathsf{D}n$ denotes decrease counter n) and the changes caused by a counter reaching 0. Whenever the action to be performed depends on the symbol read from the input tape, we denote it by $\mathsf{In}(s)$, and whenever we write a symbol s onto the output tape, it is denoted by $\mathsf{Out}(s)$.

We will now try to explain the functioning of this machine. Given two integers m and n, we encode them as the string $1^{m+1}B1^{n+1}$, so this is the initial contents of the input tape. One difficulty that will arise naturally when we try to simulate a Turing machine by a k-counter machine is how to store the contents of the tape. As we saw in the previous section, stack machines do not have that problem, because the symbols can be stored directly into the stacks. But counter machines can only store integer values, therefore we must find a correspondence between the sequences of symbols of the alphabet $\Sigma = \{1, B\}$ and the natural numbers. We can do that by representing each string $s_1 s_2 ... s_m \in \Sigma^*$ by the count

$$j = C(s_m) + 3C(s_{m-1}) + 3^2C(s_{m-2}) + ... + 3^{m-1}C(s_1)$$

, where $C(1) = 1$ and $C(B) = 2$. The first part of the computation is aimed at setting the value of the first counter at j. That will be the value of counter 1 after this cycle, when the machine enters state q_7. All other counters will have the value 0 at that moment. The machine will read the symbols one by one, every time it is in q_0, until it reaches the final endmarker, \$. Having read symbol s from the input tape, the machine stores $C(s)$ on the second counter, and then cycles through states q_3, q_4, and q_5 decreasing the first counter by 1 unit while increasing the second one by 3. When counter 1 reaches 0, the value on counter 2 is the sum of $C(s)$ and the triple of the value stored on counter 1 before reading s. The role of state q_6 is to transfer this value back to the first counter, in such a way that when the second

[14]Recall definition 2.1.7.

[15]We will see later on this section that it is possible to simulate any number of counters with only 2 counters, therefore $\mathcal{C}[2] = \mathcal{C}[n], \forall\, n \geq 2$.

counter reaches 0, we are in good conditions to read the next symbol. This way, by the time \$ is read, the first count is j.

At a second stage of the computation, the machine cycles through states q_7, q_8, and q_9, decreasing continually the first counter and increasing the second at every 3 time steps. When counter 1 reaches 0, the value stored in counter 2 is $\llcorner j/3 \lrcorner$, and corresponds to the encoding of the above described string without the last element. If $j \ mod \ 3 = 1$, then counter 1 reaches 0 in state q_8, if on the contrary $j \ mod \ 3 = 2$, then counter 1 reaches 0 in state q_9. In the first case, it means that the last element of the string was a 1, in the second case, it means it was a blank. In the first case, we increment counter 3, and in both cases we transfer the value of counter 2 back to the first counter and start the cycle all over again. That is the role of states q_{10}, q_{11}, and q_{12}.

When counter 2 reaches 0 on state q_{11}, it means that the value of the first counter was smaller than 3 on the cycle previously described. That means we have just finished "popping out" all the elements of the string, and the value on counter 3 equals $m + n + 2$. Therefore, we decrease that value twice, so when we reach q_{14} we have $m + n$ stored on counter 3.

Here we begin a different cycle, having as our goal to output an integer representing a string $\in \Sigma$ with $m + n$ 1's. The simplest way to do it is to encode 1^{m+n}. We proceed in the following manner: in state q_{14}, the value stored on counter 3 is $m + n$, and all the other counters are at 0. We enter the cycle increasing counter 1 and decreasing counter 3. In each step of the cycle we begin by transferring the result on counter 1 to counter 4 while storing its triple in counter 2. This is the role of states q_{15}, q_{16}, and q_{17}. Then we add the contents of counters 2 and 4 back to counter 1, and decrease counter 3 again. Each step of the cycle corresponds to concatenate another 1 to the string encoded by the first counter, and the number of times we do it is the value $m + n$. So when the machine reaches state q_{20}, the 3^{rd} count is the coding of 1^{n+m}.

Finally, we transfer the result to the output tape and stop. The cycle in states q_{20} to q_{24} is identical to the one of states q_7 to q_{12} explained above with the only difference that instead of storing the 1's found into a counter, they are written into the output tape.

The reason why we chose to present this example is that in the simulation of a stack machine by a counter machine, the contents of the stacks is stored in the counts as we have described here, and this example describes all the necessary mechanisms to use that information.

We now show that for any 2-stack machine there is a 4-counter machine that simulates its behavior. Therefore there is a Turing universal 4-counter machine. The following demonstration is adapted[16] from [HU01].

[16]The idea of the demonstration remains the one used in [HU01], but the definition of Turing machine is different, and we assume $\Sigma = \{1, \mathsf{B}\}$ instead of an arbitrary alphabet.

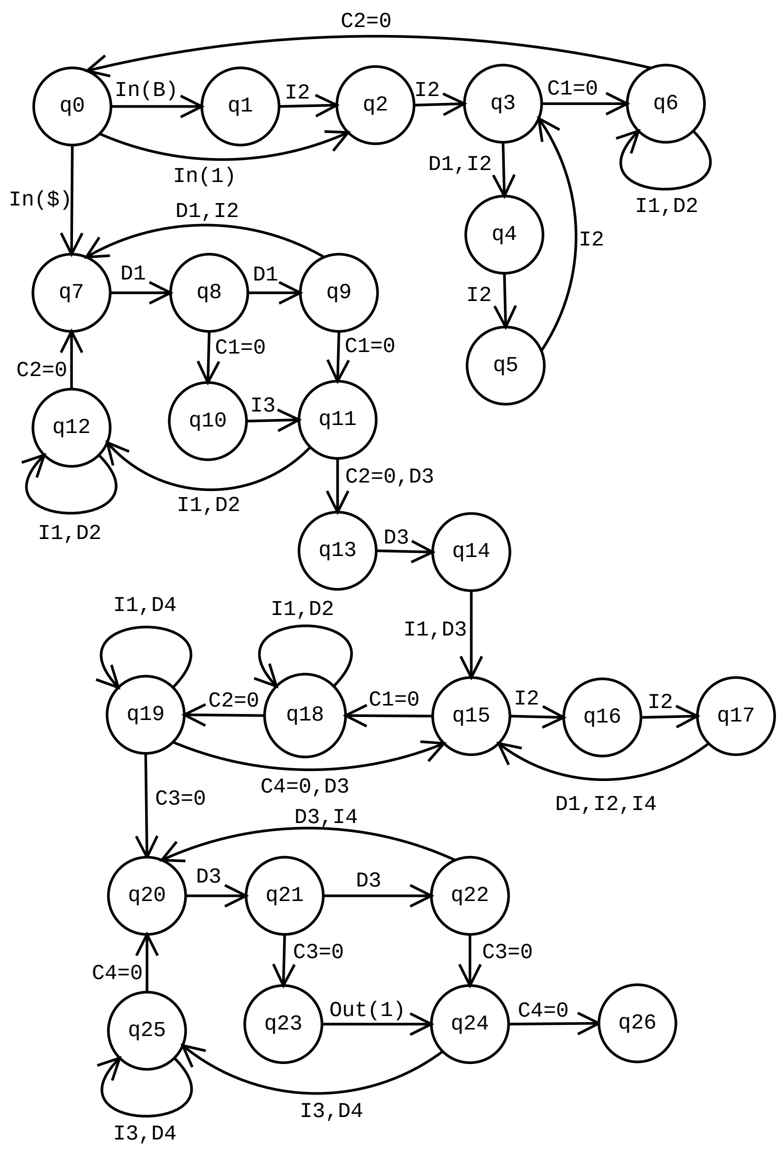

Figure 2.6: Graph for the 4-counter machine of example 2.3.1.

Proposition 2.3.1 A four-counter machine can simulate an arbitrary Turing machine.

Proof:

From the previous result, proposition 2.2.1, it suffices to show that two counters can simulate one stack. Let $\mathcal{A} = <Q, \delta, q_0>$ be a 1-stack machine. We can represent a stack $s_1 s_2 ... s_m$, uniquely by the count

$$j = C(s_m) + 3C(s_{m-1}) + 3^2 C(s_{m-2}) + ... + 3^{m-1} C(s_1)$$

where C is given by C(1)=1 and C(B)=2.

Suppose that a symbol is pushed onto the top of the stack. The count associated with $s_1 s_2 ... s_m s_{m+1}$ is $3j + C(s_{m+1})$.

To obtain this new value we begin by adding $C(s_{m+1})$ to the second count and then decrease repeatedly one unit to the first counter while increasing 3 units to the second.

If, on the contrary, we wish to pop the symbol s_m from the stack, then we must replace j by $\lfloor j/3 \rfloor$. We repeatedly decrease the first counter by 3 and add the second counter by 1. When the first counter reaches 0, the value in the second counter is $\lfloor j/3 \rfloor$.

The transition made by $\mathcal{A}$ depends on the value of the element on top of the stack, therefore it is necessary to show how our counter machine obtains the value of s_m. The value of s_m is simply j mod 3 and this value is computed by the machine's finite control, while the result from the second counter is transferred to the first.

The set of states of the counter machine will be $\{(q, read, 1), (q, read, \mathsf{B}),$ $(q, add, 1), (q, add, \mathsf{B}, (q, push, 0), (q, push, 1), (q, push, 2), (q, pop, 0),$ $(q, pop, 1), (q, pop, 2), (q, top, 0), (q, top, 1), (q, top, 2)) : q \in Q\}$.

At the beginning of each step of the 1-stack machine, the 2-counter machine is in state $(q, read, a)$, with q being the present state of the one-stack machine, and a the top of the stack. At this point there are only three possible outcomes, depending on which actions the 1-stack machine executes. The simplest case is when the action on the stack is N. In this case the action on both counters is also N, and the new state is $(q', read, a)$, where q' is the new state of the 1-stack machine.

If the next operation on the stack is $\mathsf{Push}(s)$, then we change state into (q', add, s). From here we change to state $(q', push, 0)$, storing $C(s)$ on the second adder, and then we cycle through states $(q', push, 0), (q', push, 1)$, and $(q', push, 2)$ transferring the triple of count 1 to the second counter. For a detailed description see example 2.3.1, the functioning of this cycle is exactly the same as for the states $q_1, ..., q_5$ of the example.

Finally we cycle through the states $(q', top, 0), (q', top, 1), (q', top, 2)$ executing the operations, I on counter 1, D on counter 2 and testing counter 2 for 0. This has the double effect to determine the top of the stack and to transfer the value of counter 2 to counter1. When counter 2 reaches 0, we move from state (q', top, i) to $(q', read, C_i^{-1}(s))$ and we are ready to start again.

If the next operation on the stack is Pop, we change to state $(q', pop, 0)$. We cycle through states $(q', pop, 0), (q', pop, 1), (q', pop, 2)$, executing D on counter 1 and testing it for 0, and also I on counter 2 when passing from $(q', pop, 2)$ to $(q', pop, 0)$. When the first counter reaches 0, the value stored on the second counter is $\lfloor j/3 \rfloor$, and now we go to $(q', top, 0)$ and from there on we repeat what we did for Push.

$\square$

Proposition 2.3.2 Any k-counter machine can be simulated by a 2-counter machine.

We will not prove this result here. The proof consists on showing that 3 counters can be simulated with only 2, which is achieved by encoding the 3 counts into the first counter and using the second as an auxiliary for the operations of increasing and decreasing. This result and its proof may be found in [FB94], section 6.6. We may conclude immediately:

Proposition 2.3.3 There is a Turing universal 2-counter machine.

We have seen at the end of section 2.1 that the input and output of a p-stack machine could be encoded in a stack. In the proof of result 2.3.1 we saw how to simulate one stack with two counters. Therefore, it is immediate to conclude that we can have an universal tapeless 4-counter machine, if we encode the 2 stacks on 4 counters. The input to the 4-counter machine is then the initial value of the first counter, while all the other counters start with the value 0. This initial value is given by the count j used in the proof of 2.3.1. Once that at the end of the computation of the stack machine, the second stack is empty, then we can read the output from the first counter.

The simulation of Turing machines by counter machines is done with an exponential slowdown, therefore, all the simulations we are going to show possible in the following chapters will allow us to conclude that the devices considered can simulate Turing machines with an exponential slowdown.

We can use only 3 counters to simulate 3 stacks maintaining the same input/output conventions, by using 2 counters to store the integers encoding the stacks, and using the third stack as a shared auxiliary counter to perform the operations on the stacks.

Of course it is also possible to simulate a Turing machine with a tapeless 2-counter machine, but the i/o conventions have to be changed. In this case the initial value would be 2^c, with c being the initial value of the 3-counter machine.

In this chapter we have introduced the background of notions from Computer Science necessary to the following chapters, we have shown a few examples of how different kinds of machines can be used to compute and we presented the proofs of some well-known results on simulations between different kinds of machines that we believe may be useful as a motivation to the results to be shown for the other kinds of automata we will consider, and that have not been extensively studied, as those presented so far.

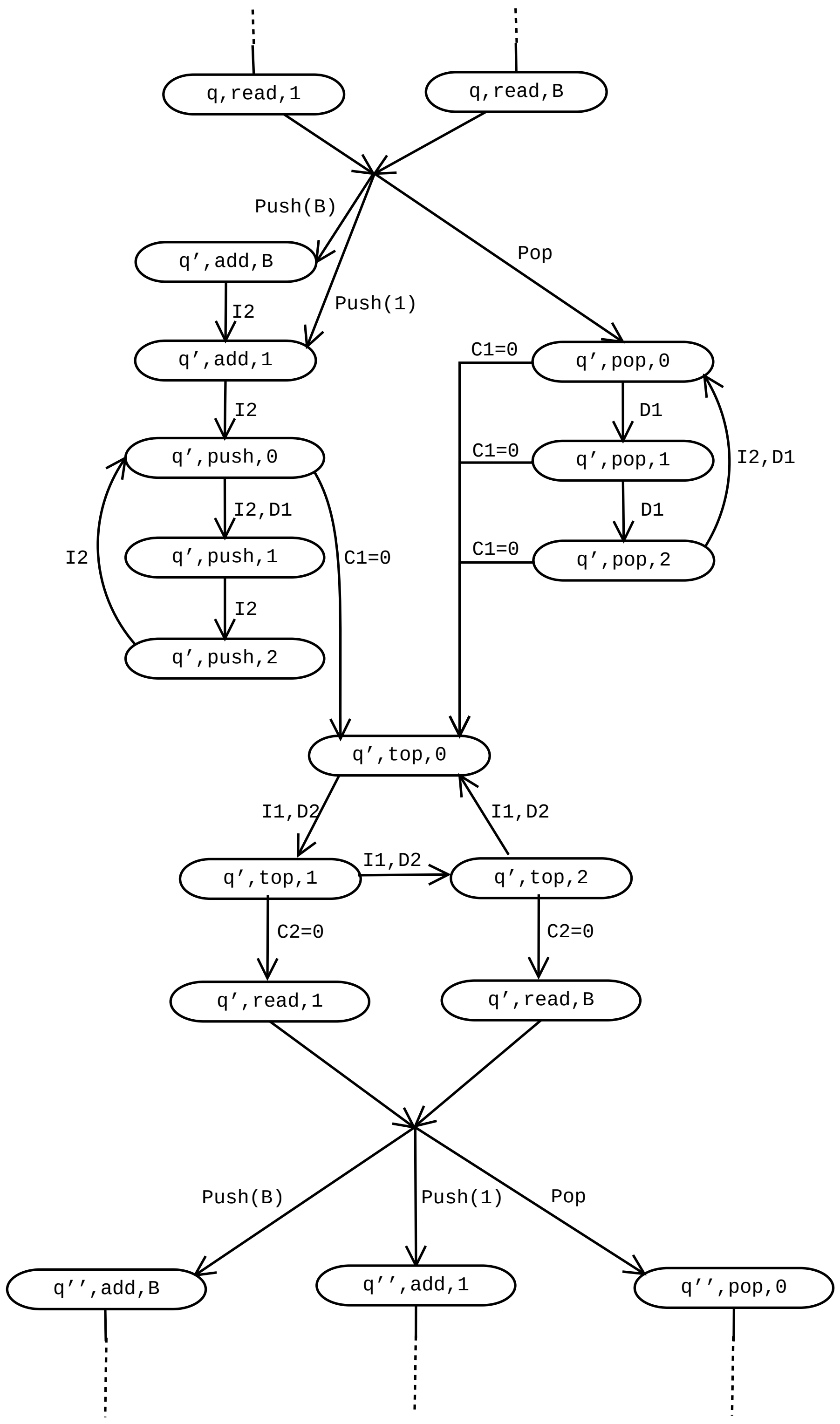

Figure 2.7: Flowchart illustrating the proof of proposition 2.3.1.

Chapter 3

Adder Machines, Alarm Clocks Machines and Restless Counters

3.1 Adder machines

A k-adder machine consists of a finite automaton, an input tape, an output tape, and a finite number k, of adders. Each adder stores a natural number. At each step, the machine's behavior is determined by the current state, the symbol under the input tape head, and the result of comparing the adders. For each pair of adders (D_i, D_j), the test Compare(i,j) has a range $\{\leq, >\}$. The operations on the adders are Compare, already described, and increase one unit, Inc.

An instantaneous description of an adder machine can be given by the state, the input tape contents, the position of the input head, the output tape contents, and the values stored on the adders.

Definition 3.1.1 A *k-adder machine* is a 4-tuple $\mathcal{D} = <Q, \Sigma, \delta, q_0>$, where Q, Σ, and q_0 are defined as for Turing machines, and the transition function δ is defined as a partial function

$$\delta : Q \times \Sigma \times \{\leq, >\}^{k^2} \to Q \times \{\mathsf{L}, \mathsf{N}, \mathsf{R}\} \times (\Sigma \cup \{\mathsf{N}\}) \times 2^{\{D_1, \ldots, D_k\}}.$$

The adder machine computes its output from the results of the comparisons between the adders, as well as the value in the input tape currently scanned and the present state. The output of δ consists on the new state, the movement of the input tape head, the symbol to write in the output tape (or N for none), and the set of adders to be increased.

Definition 3.1.2 We denote an *instantaneous description* of a k-adder machine $\mathcal{D}$ by $(q, \alpha_1 \# \alpha_2, \alpha_3, c_1, c_2, ..., c_n)$. $\alpha_1, \alpha_2, \alpha_3 \in \Sigma^*$ and $c_1, c_2, ..., c_n \in \mathbb{N}$.

q represents the current state, $\alpha_1\#\alpha_2$ represents the contents of the input tape and the position of the tape head, α_3 represents the contents of the output tape. $c_1, c_2, ..., c_n$ represent the values of the adders.

Definition 3.1.3 The *initial configuration* of a k-adder machine $\mathcal{D}$ on an input w is $(q_0, \#w, \epsilon, 0, 0, ..., 0)$.

Definition 3.1.4 We denote by $\mathcal{D}[k]$ the *class of functions that can be computed by a k-adder machine*[1], i.e., given a function $f : \mathbb{N}^n \to \mathbb{N}$, $f \in \mathcal{D}[k]$ iff there exists a k-adder machine $\mathcal{M}$ such that $\phi^n_{\mathcal{M}} = f$.[2] We denote by $\mathcal{D}[k](T)$ the *class of functions that can be computed by a k-adder machine in time T*.

Example 3.1.1 Consider the 2-adder machine which graph is shown in figure 3.1. We will show that this machine can compute the function $\lambda xy.x-y$. Once again, we remind that this is the partial function, such as in example 2.1.2.

The four elements of the tuples represent the symbol read from the input tape, the movement of the input tape head, the action to be performed on the output tape, and the set of adders increased. Actions that depend on the comparison between the adders have the condition necessary for that action to occur indicated along with the tuple.

The functioning of this machine is rather simple. Given an input tape with input $1^{m+1}B1^{n+1}$, we begin by increasing D_1 m+1 times and D_2 n+1 times. Then, if $D_2 > D_1$ the machine enters an infinite loop. If, on the contrary $D_2 \leq D_1$, it keeps increasing the second adder and writing 1's to the output tape until equality is reached.

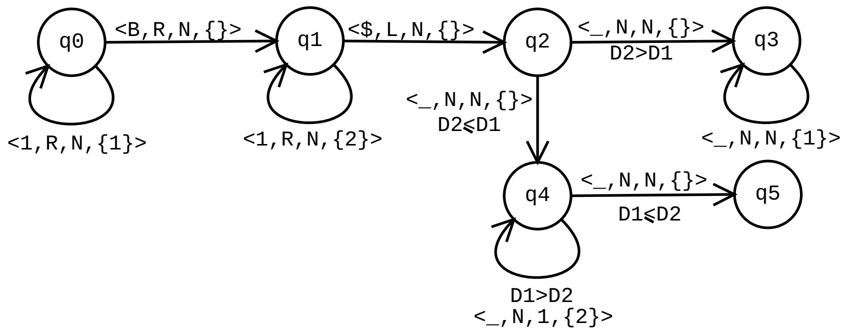

Figure 3.1: Graph for the 2-adder machine of example 3.1.1.

[1]Recall definition 2.1.7.

[2]It is possible to simulate n counters with only $2n$ adders, therefore $\mathcal{D}[4] = \mathcal{C}[n], \forall\, n \geq 4$.

We will now see the relationship between counter machines and adder machines. The following proposition and its demonstration are adapted from [Sie99], Lemma 7.1.5, page 101.

Proposition 3.1.1 Given a k-counter machine C, there exists a $2k$-adder machine that simulates C; and conversely, given a k-adder machine D, there exists a $(k^2 - k)$-counter machine that simulates D. Both simulations can be made in real-time.

Proof:
1.$\mathcal{D}[k](T) \subseteq \mathcal{C}[k^2 - k](T)$

Let D be a k-adder machine.

We define C as a counter machine having the same set of states, the same initial state, and the same alphabet as D, and a pair of counters C_{ij}, C_{ji} for each pair (i,j) of distinct adders in D.

C's computation is performed as follows: given the values of the input symbol read, of C's internal state and the values of the zero tests for the counters, change to the same state and perform the same actions on the input tape and the output tape D would do for the same values of input symbol read and internal state. For every action $\mathsf{Compare}(D_i, D_j)$ D would perform, C checks if C_{ij} is zero. For every action $\mathsf{Inc}(D_i)$ D would perform, C makes $D(C_{ji})$, for all $1 \leq j \leq k$ such that $C_{ji} \neq 0$ and $I(C_{ij})$, for all $1 \leq j \leq k$ such that $C_{ji} = 0$.

At each time step, the counter C_{ij} holds the value $max\{0, D_i - D_j\}$. Hence $\mathsf{Compare}(D_i, D_j) = \leq$ iff $C_{ij} = 0$.

From the above definition, C and D are always in the same internal state and perform always the same actions on the input and output tapes. Moreover, the correspondence between the value of the adders and the value of the counters described above is maintained throughout the computation. We can now conclude that C simulates D.

2. $\mathcal{C}[k](T) \subseteq \mathcal{D}[2k](T)$

Suppose now that C is a k-counter machine. Let D be an $2k$-adder machine with the same set of states, the same initial state, the same set of internal states and the same alphabet as C. For every counter C_i, D has a pair of adders: Inc_i and Dec_i. Inc_i is increased every time C does $\mathsf{I}(C_i)$, and Dec_i is increased every time C does $\mathsf{D}(C_i)$.

At each time step, the internal state and the position of the tape head of the two machines coincide. The value on C's i-th counter is simply the difference between Inc_i and Dec_i, so $\mathsf{Test0}(C_i)$ can be simulated by comparing Inc_i and Dec_i. If $\mathsf{Compare}(Inc_i, Dec_i) = \leq$, then $\mathsf{Test0}(C_i)$ is true, else $\mathsf{Test0}(C_i)$ is false. Therefore, given the results of the comparisons between adders, the state and the input symbol read, D changes to the same state C would do, with the same movement on the input tape, and the same action on the output tape, and adds 1 to the adders corresponding to the counters C would increase.

$\therefore D$ can simulate C.

$\square$

Example 3.1.2 The translation from $k-$counter machines to equivalent $2k$-adder machines is straightforward. In figure 3.2, we can see the 8-adder machine corresponding to the 4-counter machine of example 2.3.1.

Definition 3.1.5 An *acyclic Boolean circuit* is a 5-tuple $B =< V, I, O, A, f >$, where:

V is a non-empty finite ordered set (whose elements are called *gates*),

I is a non-empty set (whose elements are called *inputs*),

V and I are disjoint,

O is a non-empty subset of V (whose elements are called *output gates*),

A is a subset of $(V \cup I) \times V$ (whose elements are called *transitions*),

$< V \cup I, A >$ denotes an oriented acyclic graph and

$f : V \cup I \rightarrow \{\mathsf{Not}, \mathsf{And}, \mathsf{Or}, 0, 1\}$ is an application, called *activity*,

for every $v \in V$ the number of elements in A that have v as second element is at most:

2, if $f(v) = \mathsf{And}$ or $f(v) = \mathsf{Or}$
1, if $f(v) = \mathsf{Not}$
0, if $f(v) = \mathsf{1}$ or $f(v) = \mathsf{0}$.

Each gate in V computes a Boolean function determined by the activity f. I represents a set of inputs, and O represents a set of outputs. A represents the connections between the gates, the inputs, and the outputs. We call the graph $(V \cup I, A)$ the *interconnection graph* of B. The following example is adapted from [Par94].

Example 3.1.3 Let B be a Boolean circuit $B =< V, I, O, A, f >$ with

$V = \{g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_8, g_9, g_{10}\}$

$I = \{x_1, x_2, x_3, x_4\}$

$O = \{g_{10}\}$

$A = \{(x_1, g_1), (x_2, g_1), (x_2, g_2), (x_3, g_2), (x_3, g_3), (x_4, g_3), (g_1, g_4), (g_2, g_7), (g_2, g_5), (g_3, g_6), (g_4, g_7), (g_5, g_8), (g_6, g_8), (g_7, g_9), (g_8, g_9), (g_9, g_{10})\}$

$$\begin{array}{lll}
f(g_1) = AND & f(g_5) = NOT & f(g_8) = OR \\
f(g_2) = AND & f(g_6) = NOT & f(g_9) = AND \\
f(g_3) = OR & f(g_7) = OR & f(g_{10}) = NOT \\
f(g_4) = NOT & &
\end{array}$$

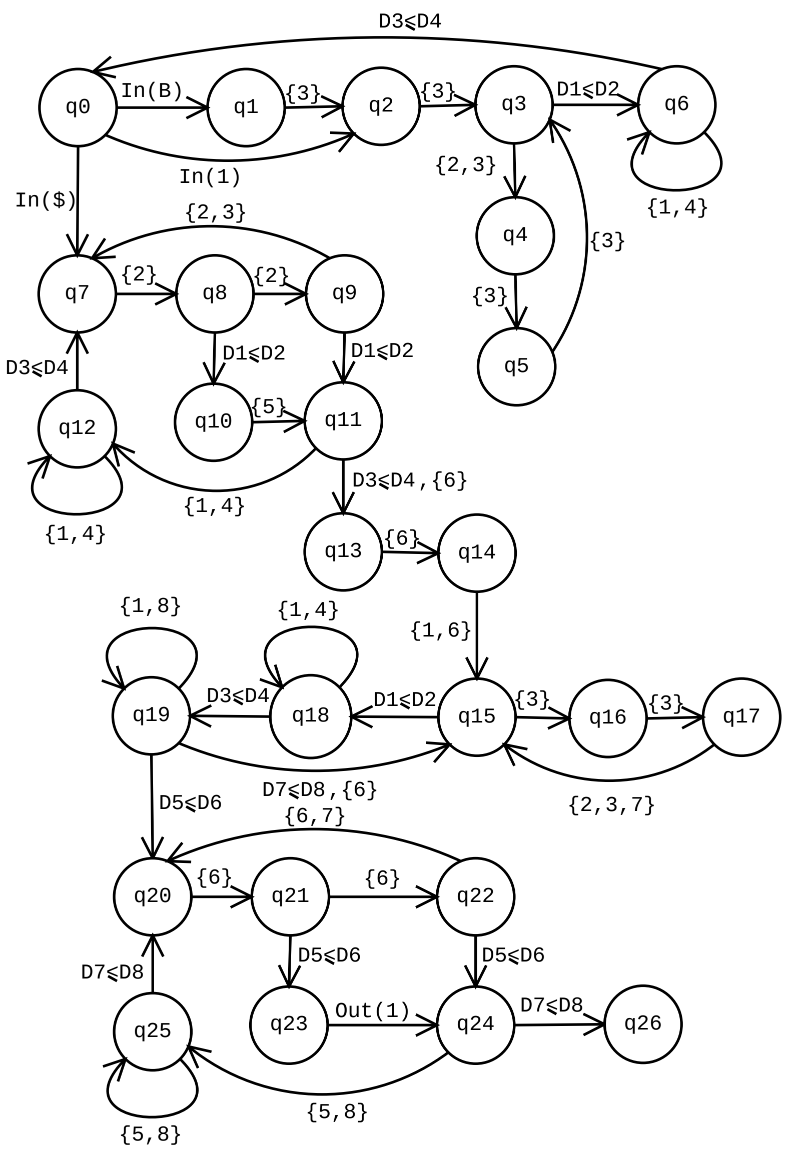

Figure 3.2: 8-adder machine that simulates the 4-counter machine of example 2.3.1.

Definition 3.1.6 A k-adder machine is said to be *acyclic* if its finite control has one single state.

The reason why we call these machines acyclic is because their transition function can be computed by an acyclic boolean circuit as a function of the results of the comparisons between the adders only. We will see now that every adder machine can be transformed into an equivalent acyclic adder machine, by allowing it to have some extra adders.

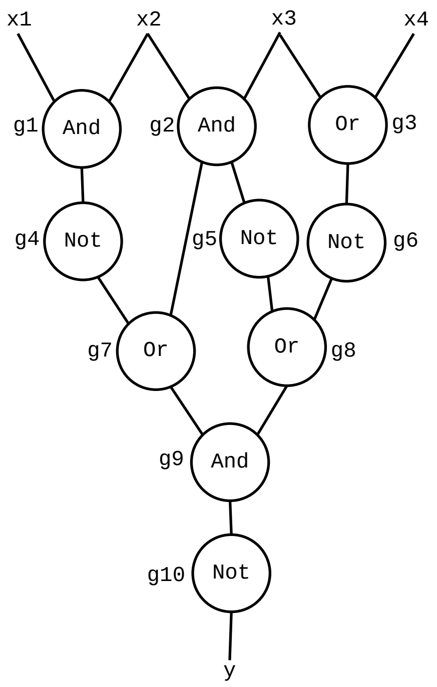

Figure 3.3: Boolean circuit of example 3.1.3.

Proposition 3.1.2 A k-adder machine with c control states can be simulated by an acyclic $(k+2c)$-adder machine.[3]

Proof:

Let $\mathcal{D}$ be a k-adder machine with c control states. During this proof, we shall denote $\mathcal{D}$'s states as $q_1, q_2, ..., q_c$.

$\mathcal{D}$ can be instantaneously described by the present state, the contents of the input and output tapes, the position of the input tape head and the values of the adders.

Analogously, if $\mathcal{D}'$ is an acyclic $(k+2c)$-adder machine without internal states, then $\mathcal{D}'$ can be instantaneously described by the position of the tape head of the input tape, the contents of the two tapes and the values of the adders.

$\mathcal{D}'$ can be used to simulate each step made by $\mathcal{D}$ in two steps.

We will call the first k adders of $\mathcal{D}'$ *storing adders*, the adders $D'_{k+1}, ...,$ D'_{k+c} *state adders*, and the last c adders *final adders*. At the beginning of each step (of $\mathcal{D}$), the values of the storing adders equal the values of the corresponding adders of $\mathcal{D}$; the values of the state adders are equal amongst themselves, except for the value of D'_{k+i}, which is one unit greater than the

[3]This result is adapted from [KS96], where it is claimed that the simulation can be made by an acyclic $(k+c)$-adder machine. We believe this was a lapse, and the authors didn't realize that they were increasing an adder twice on the same time step.

others, with q_i being $\mathcal{D}$'s present state (prior to this step); the values of the final states are all equal. The actions performed on the input and output tapes will be the same for both machines.

In a first step, the Boolean control uses the value of the comparisons between the state adders to determine the corresponding state in $\mathcal{D}$: afterwards performs in the storing adders the changes that $\mathcal{D}$ would do in its adders given that state, and similarly for the actions to be performed on the tapes. Still in this first step, $\mathcal{D}'$ uses the Boolean control to increment all the state adders, except D'_{k+i}, and to increment also D'_{k+c+j}, being j the state $\mathcal{D}$ arrives after its step. After this first step, all the state adders have the same value, and the next state is stored in the final adders.

In a second step, the value of D'_{k+j} is increased, as well as the value of all the final adders, except for D'_{k+c+j}. The value of the storing counters is left unaltered and the actions on both tapes are N (this second step does not perform any computation on the storing adders, it only resets the initial conditions to simulate $\mathcal{D}$'s next step and the reason why it is necessary is that D'_j cannot be increased twice in the same time step).

□

Let us now see an example of how we can transform an adder machine into its acyclic equivalent and how to design a Boolean circuit to compute its transition function, in order to exemplify the construction given in the proof of 3.1.2.

Example 3.1.4 Please recall the 2-adder machine of figure 3.1. We shall refer to it as D. We will call D' to the acyclic machine designed from D by the process described in the proof of proposition 3.1.2. We will now show how the transition function for this machine can be computed by a Boolean circuit. We will begin by dividing this task into subcircuits, and then see how to combine them.

The transition function receives the state, the input symbol read and the results of the comparisons between the adders. D' has only one state and the state information is irrelevant, therefore the inputs to the Boolean circuit will be: an input for every pair of adders D'_i, D'_j, that we shall denote as $\mathbf{D'_i > D'_j}$, and that has value 1 iff $D'_i > D'_j$; and another input **Input** σ for each symbol $\sigma \in \Sigma$, that has value 1 iff the symbol read is σ. D has 2 counters and 6 states, so by proposition 3.1.2, D' will have $2 + 2 \times 6 = 14$ adders. The first 2 of these adders simulate the adders of D, the next 6 store the state, and the last 6 are only auxiliary.

The Boolean circuit must be able to compute: the set of adders to increase, the movement of the input tape head, and the symbol (if any) to be stored in output tape. As for the symbols to be written in the output tape, we will proceed as we did for the input: we will assume the existence of an output gate **Output** σ for each symbol $\sigma \in \Sigma$. For the movement of the input tape head, we will proceed in the same way: there are 3 output gates **MoveR**,**MoveN**, and **MoveL**. Finally, we will have an output gate for each adder D'_i, $\mathbf{Inc_i}$, that will take value 1 iff adder i is to be increased.

We will not need all the comparisons between these adders, otherwise there would be 14^2 input gates with the comparisons of the adders. In fact, given D''s *modus operandi*, we will only need $\mathbf{D'_1 > D'_2}$ and $\mathbf{D'_2 > D'_1}$ for the storing adders, and one more input gate for each of the remaining adders, that will allow us to conclude if that adder has a greater value than the others,i.e., if it is the adder keeping track of the state or not. Therefore we will choose to use $\mathbf{D'_3 > D'_4}$, $\mathbf{D'_4 > D'_3}$, $\mathbf{D'_5 > D'_3}$, ... , $\mathbf{D'_8 > D'_3}$, $\mathbf{D'_9 > D'_{10}}$, $\mathbf{D'_{10} > D'_9}$, ... , $\mathbf{D'_{14} > D'_9}$. Therefore, we will have to use a total of 17 input gates.

From D's transition function it is immediate to see that the first adder is increased if we are in state q_0 and read 1, or if we are in state q_3. The adders that keep track of states q_0 and q_3 are D'_3 and D'_6, respectively, so the Boolean function that determines the value of $\mathbf{Inc_1}$ is

$$\mathbf{Inc_1} = (\mathbf{D'_3 > D'_4}\ \textit{And}\ \mathbf{Input\ 1})\textit{Or}\ \mathbf{D'_6 > D'_3}$$

In a similar manner we quickly conclude that the Boolean function that determines the value of Inc 2 is

$$\mathbf{Inc_2} = (\mathbf{Input\ 1}\ \textit{And}\ \mathbf{D'_4 > D'_3})\textit{Or}(\mathbf{D'_7 > D'_3}\ \textit{And}\ \mathbf{D'_1 > D'_2})$$

So, we can build the sub-circuit of figure 3.4.

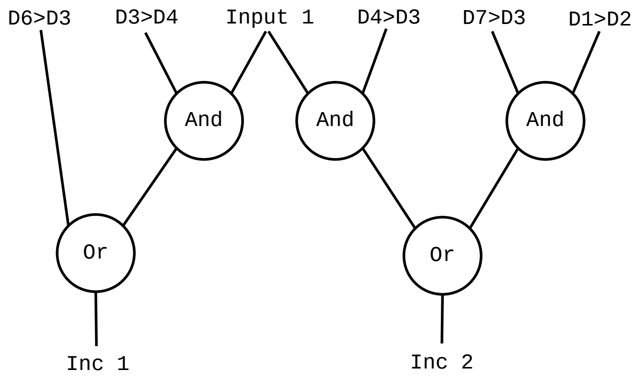

Figure 3.4: Part of the Boolean circuit that computes the increments on the storing adders.

The movement of the tape head is easy to determine. From figure 3.1 we can see that D only moves right if it is in state q_0, or if it is in state q_1 and reads 1. D only moves left if it is in state q_1 and reads \$. If none of these situations happens, D makes no move. Therefore, the circuit in figure 3.5 can compute the movement of the input tape head of D'. We also state

the equations:
$\mathbf{MoveR} = \mathbf{D_3'} > \mathbf{D_4'} Or (\mathbf{D_4'} > \mathbf{D_3'} And \mathbf{Input\ 1})$
$\mathbf{MoveL} = \mathbf{D_4'} > \mathbf{D_3'} And \mathbf{Input}\ \$$
$\mathbf{MoveN} = (Not \mathbf{MoveR}) And (Not \mathbf{MoveR})$

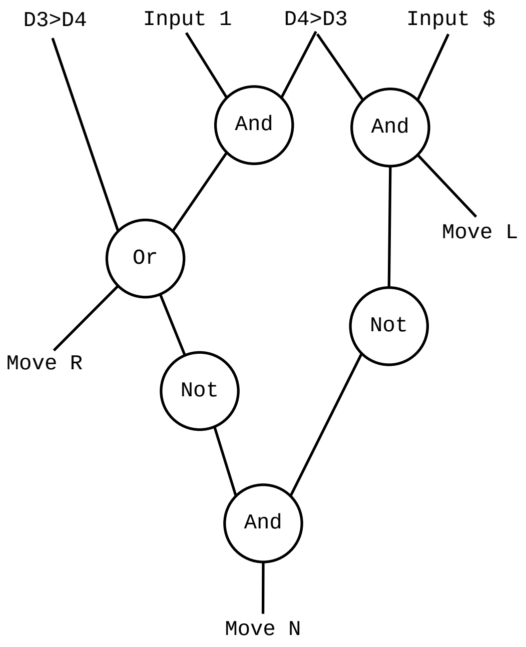

Figure 3.5: Part of the Boolean circuit that computes the movement of the tape head.

The output is even simpler: the machine never writes a blank symbol into the output tape, so we can use the constant 0 to compute **Output B**. D only writes a 1 to the output, when it is in state q_4 and $D_1 > D_2$. If neither **Output B** nor **Output 1** then the output of D''s transition function should be **Output N**. Figure 3.6 shows this part of the circuit.

The only thing we haven't seen yet is how to compute which of the state adders and final adders to increase. One of the things we will always need to know is at which step of the computation we are: if we are simulating a step of D in this step of D', or merely resetting the state adders to do it in the next step of D'. This can be achieved by seeing if any of the state adders holds a value greater than the others, through a series of Or gates, until we have a gate that takes value 1 iff there is an adder verifying this condition. We will call this gate g.

To present the complete sub-circuit that computes all these adders would result in a confusing overfull diagram, so we have chosen to show only how to implement $\mathbf{Inc_3}$, $\mathbf{Inc_4}$, and $\mathbf{Inc_9}$. We believe that this is enough to understand the rest of the circuit. It is shown on figure 3.7.

For $\mathbf{Inc_3}$, $\mathbf{Inc_4}$ and the output gate of any state adder, the procedure is

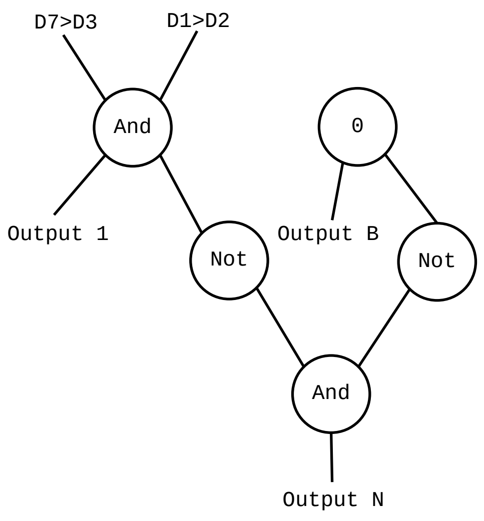

Figure 3.6: Part of the Boolean circuit that computes the output.

very simple. We must increment adder 3 if adder 9 is greater than the other final adders, or if g and adder 3 is not greater that the other state adders. For adder 4 is is exactly the same thing, but using adder 10 instead of 9.

For $\mathbf{Inc_9}$ and any other final adder in general, it is not so simple. We must look at the transition function and design the circuit accordingly. D changes into q_0 iff it is already at q_0 and reads 1. This corresponds to $\mathbf{D'_3} > \mathbf{D'_4}$ *And* **Input 1**. The other situation in which we increase adder 9 is to equalize the final adders again. This is done when g is 0, and D'_9 is not the adder corresponding to the new state.

It is straightforward how to complete this circuit in such a way that for every possible tuple of δ's range, there is an output gate that takes value 1 iff that tuple is the result of D''s computation.

From the proposition 3.1.2 we conclude that acyclic adder machines are Turing universal. In order to achieve this Turing universality, we saw in the previous section, 2.2., that we needed only a 2-counter machine. Therefore, we can conclude that 4-adder machines are Turing universal. As we have already mentioned in page 12, there is a universal Turing machine with only 2 states. These 2 states are maintained in the simulations by stack machines and counter machines[4]. We conclude that there is a universal acyclic 8-adder machine (4 adders to simulate 2 counters, plus 4 adders to keep track of 2 states). It remains as an open question whether this number can still be improved.

In order to maintain simple input/output conventions we will assume 10

[4]This machine has only 2 states but has necessarily 10 symbols, therefore the proof we have given in 2.3.1 does not apply. However all we have to do is to change the count in base 3 to a count in base 11, and the result holds.

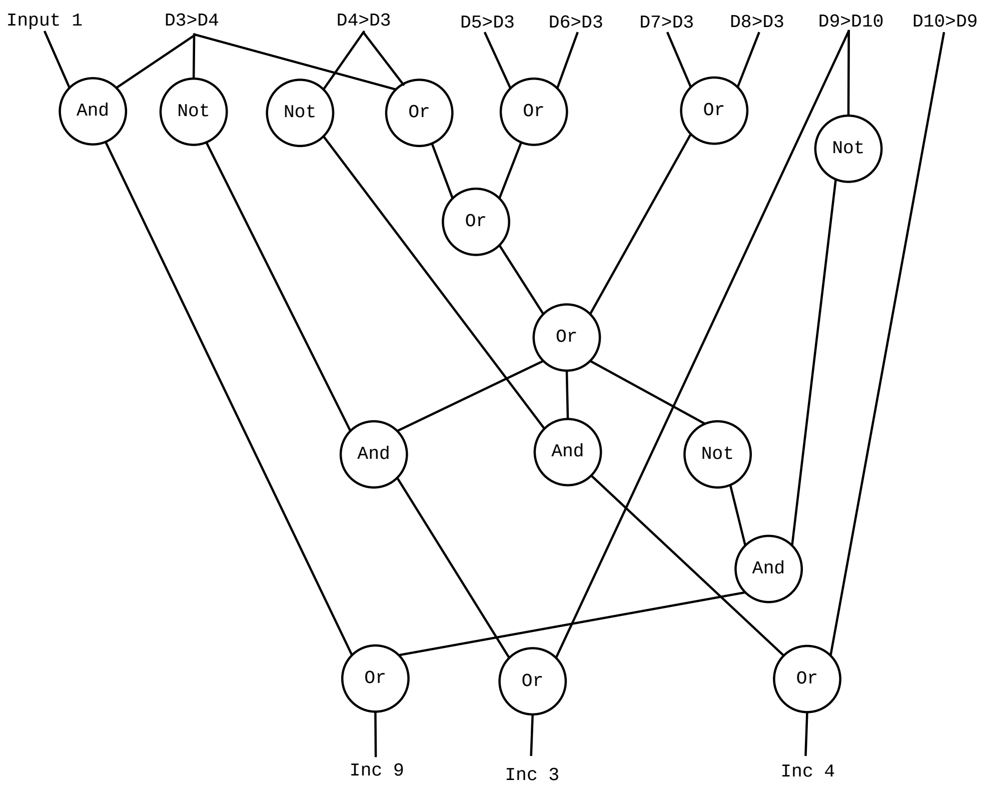

Figure 3.7: Part of the Boolean circuit that computes the increments on adders 3, 4 and 9.

adders, instead of 8, corresponding to simulating 3-counter machines instead of 2. The input and output of the 3-counter machine could be directly written and read from the value of the first counter. The value of the first counter is simulated, as we have just seen, by the difference between two adders. As for the input, we may choose to keep the second adder at 0, and initialize the first adder with the same initial value that the counter. When it comes to output, we will have to check both adders in order to obtain the machine's output.

Definition 3.1.7 A *tapeless k-adder machine* is a 3-tuple $\mathcal{D} = <Q, \delta, q_0>$, where Q and q_0 are the set of states and the initial state, and the transition function δ is defined as a partial function

$$\delta : Q \times \{\leq, >\}^{k^2} \to Q \times 2^{\{D_1,\ldots,D_k\}}.$$

3.2 Alarm clock machines

Finally, we come to the notion of alarm clock machine. This kind of automaton was first introduced in [KS96] and a k-alarm clock machine consists on a transition function $\delta : \{0,1\}^{5k} \rightarrow 2^{\{\mathsf{delay}(i),\mathsf{lengthen}(i)\,:\,1\le i\le k\}\cup\{\mathsf{halt}\}}$. The fact that δ's domain is $\{0,1\}^{5k}$ means that δ's input is simply the information of which alarm clocks have alarmed in the last 5 time steps[5] and when they did so. δ's output is simply which clocks to delay, which clocks to lengthen, and whether the machine halts or not.

We stress out the fact that the alarm clocks are merely a conceptual tool that helps us understand the functioning of this machine. In fact we can define this kind of machines without mentioning alarm clocks altogether, as they are not part of the machine's definition. The input to $\mathcal{A}$ consists of $((p_1,t_1),...,(p_k,t_k))$, where p_i denotes the period of clock i, and time t_i denotes the next time it is set to alarm. For notational ease, we keep arrays $a_i(t)$, for $t \in \mathbb{N}$ and $1 \le i \le k$, with every entry initially set to 0. At time step T, for $1 \le i \le k$, if $t_i = \mathrm{T}$, then $a_i(\mathrm{T})$ is set to 1 and t_i is set to $t_i + p_i$. This event corresponds to clock i alarming. δ then looks at $a_i(\mathrm{t})$ for $1 \le i \le k$ and $T-5 < t \le T$, and executes 0 or more actions. Action $\mathsf{delay}(i)$ sets t_i to t_i+1, action $\mathsf{lengthen}(i)$ sets p_i to p_i+1 and t_i to t_i+1, and action halt halts the alarm clock machine. We make one final restriction to the behavior of an alarm clock machine: when its transition function is applied to a vector of $5k$ 0's, then it outputs the empty set of actions. Intuitively, this corresponds to demanding that the machine must be asleep until it is woken.

The role of the alarm clocks of the alarm clock machine is to store information on the frequency at which they alarm. In the same way as in Turing machines the tapes are potentially infinite, but at any given instant only a finite amount of information is actually stored on the tape is finite, the period of the clocks may increase unlimitedly, but any given instants all alarm clocks have a period limited by some given constant.

Definition 3.2.1 A *k-alarm clock machine* $\mathcal{A}$ is a total function from $\{0,1\}^{5k}$ to a subset of $\{\mathsf{delay}(i), \mathsf{lengthen}(i) : 1 \le i \le k\} \cup \{\mathsf{halt}\}$ that verifies $\mathcal{A}(000...00) = \{\}$.

Definition 3.2.2 Given a k-alarm clock machine $\mathcal{M}$, an *instantaneous description* of $\mathcal{M}$, also called a *configuration*, is a k-tuple $<(p_1,t_1),(p_2,t_2),...,(p_k,t_k)>$, where $p_i \in \mathbb{N}$ denotes the period of the i^{th} clock and $t_i \in \mathbb{N}$ denotes the next time at which the i^{th} clock will alarm.

[5]The reason why we have chosen to consider the last 5 time steps is that 5 is the time span we are going to need in order to simulate adder machines by alarm clock machines, but we might have considered any other integer greater than 5.

A k-alarm clock machine, contrarily to all the other devices we have seen so far, does not receive as an input a sequence over a given alphabet. Instead, the input to the alarm clock machine consists of a k-tuple $<(p_1,t_1),(p_2,t_2),...,$
$(p_k,t_k)>$, where $p_i \in \mathbb{N}$ denotes the period of the i^{th} clock and $t_i \in \mathbb{N}$ denotes the next time at which the i^{th} clock will alarm, i.e., the input to the alarm clock machine is its *initial configuration*.

Definition 3.2.3 Given a k-alarm clock machine $\mathcal{M}$, and an *initial configuration* of $\mathcal{M}$, $<(p_1,t_1),(p_2,t_2),...,(p_k,t_k)>$, the computation of $\mathcal{M}$ on the given configuration is a sequence of configurations that verifies

$$p_i(t+1) = \begin{cases} p_i(t)+1, & if\ \mathsf{lengthen}(i) \in \delta(t) \\ p_i(t), & otherwise. \end{cases} \quad \text{and}$$

$$t_i(t+1) = \begin{cases} t_i(t)+1, & if\ \mathsf{delay}(i) \in \delta(t)\ or\ \mathsf{lengthen}(i) \in \delta(t) \\ t_i(t)+p_i(t), & if\ t_i = t \\ t_i(t), & otherwise. \end{cases}$$

Proposition 3.2.1 Given an acyclic k-adder machine $\mathcal{D}$ that computes in time T, there exists a $(\frac{k^2+k}{2}+2)$-alarm clock machine that simulates $\mathcal{D}$ in time $O(T^3)$.

Proof:

Given an acyclic k-adder machine $\mathcal{D}$ with adders $D_1,...,D_k$ we construct an alarm clock machine $\mathcal{A}$ that simulates $\mathcal{D}$.

$\mathcal{A}$ has the clocks $A_0, A_1, ..., A_k, A_{00}, A_{ij} : 1 \leq i < j \leq k$ and the functioning is as we now describe.

The input to $\mathcal{A}$ can be encoded in the initial values of the period and next alarm of the clocks. Assuming that $\mathcal{D}$ is a k-adder machine without input/output tapes, its input is stored as the initial value on one of its counters. Let s be that value. Initially, all the clocks $A_0,...,A_k$ have period $2s+10$, and the clocks A_{00}, A_{ij} have period $2s+11$. Throughout the entire computation, all the clocks $A_0, A_1, ..., A_k$ will have the same period p (p changes over time), and all the auxiliary clocks A_{00}, A_{ij} will have period p+1. The periods of the clocks will always be lengthened simultaneously. Throughout the computation, the period of the clocks, p, will be large enough to verify $p > 2max_i(D_i)+10$.

If A_0 alarms at time t_0, then A_i alarms at time $t_0 + D_i + 5$. Thus each alarm clock will be used to store the value of a counter. We can set as the initial conditions that A_0 alarms at time 0, A_1 alarms at time $s+5$ and A_i alarms at time 5, for $1 < i \leq k$.

The clocks A_{ij} are initially synchronized with A_{00}, alarming at time step 4 (other choices could be made, as long as the clocks alarm simultaneously, after A_0 and before the first of the A_i's, at the beginning of each step of the adder machine), and in each step the finite control performs the tasks:

i) If A_{00}, A_{ij} and A_i, but not A_j alarm at the same time, then the finite control delays A_{ij} once;

ii) If A_{00}, A_{ij} and A_j, but not A_i alarm at the same time, then the finite control delays A_{ij} twice;

iii) If A_{00} and A_0 alarm simultaneously, that means that all the comparisons are done and their results are stored in the alarming pattern of the auxiliary clocks, and it will be available to the alarm clock machine within the next 2 time steps. The machine then enters a short period of time in which it performs the necessary delays on the clocks holding the values of the adders that are incremented by the adder machine, and resets all the other clocks to the initial conditions, so that the next step of the adder machine may be simulated, via a new cycle of classification of the clocks, corresponding to points i) and ii) above.

Our alarm clock machine only needs to maintain in memory the last 5 time steps (when there has been an alarm in the last 5 time steps) to carry out the tasks described above. Let us see how.

If at present time step clock A_{00} alarms, then the finite control checks all pairs (i,j) with $i < j$, and if one of A_i, A_j alarms at present time step, but not both, then A_{ij} is delayed.

If one time step ago clock A_{00} has alarmed, then the finite control delays all clocks A_{ij} such that A_j alarmed one time step ago, but A_i didn't.

With this we achieve the tasks of points i) and ii) above.

Due to the difference of phase and the initial conditions, A_{00} will alarm simultaneously with the alarms that correspond to an adder with a smaller value, before doing so with the alarm clocks that correspond to an adder with a larger value. After $p^2 + p$ time steps A_{00} has alarmed simultaneously with all the clocks except A_0. At that time A_0 and A_{00} alarm simultaneously and in the next two time steps the finite control receives all the information necessary to compute the next step of the adder machine, that is, the result of all the comparisons between adders, according to the following table:

Adders	Alarm Clocks
$D_i = D_j$	A_{ij} and A_{00} alarm at the same time
$D_i < D_j$	A_{ij} alarms 1 step after A_{00}
$D_i > D_j$	A_{ij} alarms 2 steps after A_{00}

If the clocks A_0 and A_{00} have alarmed two time steps ago, then we check all pairs (i,j), with $i < j$ and determine which of them alarmed 1 or 2 time steps ago and delay them. We also delay A_{00}. We are also in the conditions to create a matrix with the values of the comparisons between the adders ($D_i \leq D_j$ iff A_{ij} doesn't alarm at the present time step) and calculate the next step of the adder machine. If the adder machine would have stopped, then the alarm clock machine halts. If not, we delay all the clocks corresponding to the counters increased by the adder machine.

If the clocks A_0 and A_{00} have alarmed 3 steps ago, then we delay A_{00} and we also delay all the clocks A_{ij} that alarmed 3 time steps ago and also lengthen simultaneously the period of all the clocks. This delay of all clocks assures us that there won't be overlaps.

Finally, if A_0 and A_{00} have alarmed 4 steps ago, we lengthen the periods of all clocks again.[6]

After these delays, it is easy to verify that all the auxiliary clocks are synchronized with A_{00} again, because in the course of the entire cycle, each auxiliary clock was delayed exactly twice (four times if we count the delays from lengthening the period). It is also easy to see that none of the other clocks will alarm before A_{00}, because from the initial condition we know that all the adder clocks have a time phase of 5 steps, clock A_0 is never delayed in the computation, and the period of all clocks is always lengthened at least twice as much as the adder clocks are delayed with respect to A_0. Therefore, we are in the initial conditions to begin the simulation of a new step of the adder machine.

To simulate the t-th step of the adder machine, since p grows with $2T$ ($O(T)$), the alarm clock machine performs the comparisons in time $O(T^2)$ and the delays and lengthenings in time $O(1)$. Thus, to simulate t steps we need $O(T^3)$ steps.

$\square$

Example 3.2.1 Let us consider an alarm clock machine functioning, and look at the simulation of one step of the adder machine. We will assume a 3-adder machine with values 2, 4 and 2 stored on the adders. We will also suppose that the result of the transition function for these values is to increment the first adder only. We will show an 8-alarm clock machine simulating this step, assuming the period of A_0 to be presently 15. Each column of the following tables is a vector of 1's and 0's, representing which clocks have alarmed at that time step. For simplicity, only the 1's are represented, the blank spaces correspond to the 0's.

A_0	1	...												1
A_1		...				1								
A_2		...						1						
A_3		...				1								
A_{00}	1	...												
A_{12}		...												
A_{13}		...												
A_{23}		...												
time	0	1 2 3	4	5	6	7	8	9	10	11	12	13	14	15

[6]In order to avoid overlaps, we only need to lengthen the period of the clocks once, since for any clock there is at most one delay operation performed on it during each cycle. However, in order to implement these alarm clocks via restless counters, as we will see in next section, we must delay them twice. See footnote 7 for further explanations.

In the initial time steps 0 to 5 the previous step of the adder machine is computed and the conditions to start the simulation of a new step are restored, with the synchronization of the auxiliary clocks, as we can see on step 18.

A_0															1													
A_1							1															1						
A_2									1															1				
A_3							1															1						
A_{00}			1																1									
A_{12}			1																1									
A_{13}			1																1									
A_{23}			1																1									
time	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	...	79	80	81

Every time the auxiliary clocks alarm, the phase to the adder clocks is shortened...

A_0													
A_1	1						1						
A_2			1						1				
A_3	1						1						
A_{00}	1							1					
A_{12}	1								1				
A_{13}	1							1					
A_{23}	1									1			
time	82	83	84	85	86	...	97	98	99	100	101	102	...

Until, at time step 82, they reach some of the adder clocks. In this particular example, by time step 100, the classification of the clocks is complete and visible by the pattern of alarms.

A_0		1								1		
A_1							1					
A_2						1						
A_3								1				
A_{00}		1									1	
A_{12}			1								1	
A_{13}		1									1	
A_{23}				1							1	
time	...	210	211	212	...	218	219	220	...	226	227	...

Finally, at step 210, A_0 and A_{00} alarm simultaneously, denoting the end of the classification cycle. The operations on the adder clocks are performed, the auxiliary clocks are synchronized, and the machine can start again.

The output of the adder machine was encoded in the difference between the first two adders. The output value of the alarm clock machine can be encoded by the phase difference between A_1 and A_2 at the end of the computation. We saw in section 2.1 that 8 adders were sufficient to assure Turing universality of acyclic adder machines. In the proof of proposition 3.2.1, we have seen that we need $(\frac{k^2+k}{2}+2)$ alarm clocks to simulate an acyclic k-adder machine. Therefore, we can guarantee the existence of a Turing universal 38-alarm clock machine. In order to maintain the referred input/output conventions, we need 57 clocks.

3.3 Playing with restless counters

In the previous section, we showed how we can simulate acyclic adder machines with a device that doesn't rely on tapes, stacks, counters, adders or any other memory device, but instead keeps the information stored only in the frequency of its alarms. In this section we will introduce a device that will serve as a bridge between the alarm clock machine and neural networks.

As we have discussed in the beginning of chapter 2, page 2, one of the greatest difficulties in using neural networks to simulate Turing machines is the fact that the neurons are continually updating, and therefore it is impossible to store a value on a neuron as its activation value, whenever $\rho(x) \neq x$, for almost all x, which is the case when we are using a sigmoidal activation function.

Therefore, we show in this section how we can simulate a k-alarm clock machine with simple restricted counters, which we call restless counters.

Every restless counter must always either be increased or decreased in each time step, unless it is at 0, in which case it must stay at 0 for two time steps and then be increased. Once a restless counter is instructed by the finite control to start increasing or decreasing, it must continue to do so in each successive time step until it has an opportunity to change its direction (the finite control can "sleep" after giving the command of what direction to continue with). We call the event in which the restless counter i is set to 0 a *zero event for i*, and the event in which some restless counter is set to 0 a *zero event.*

Definition 3.3.1 A *k-restless counter machine* is a total map
$F : ((\mathbb{N} \cup \{0', 0''\}) \times \{Up, Down\})^k \rightarrow \{I, D, N\}^k \cup \{\mathsf{halt}\}$.

Definition 3.3.2 Given a k-restless counter machine $\mathcal{M}$, an *instantaneous description* of $\mathcal{M}$, also called a *configuration*, is a k-tuple $<(n_1, d_1), (n_2, d_2), ..$
$..., (n_k, d_k)>$, where $n_i \in \mathbb{N} \cup \{0', 0''\}$ denotes the value of the i^{th} counter and $d_i \in \{Up, Down\}$ denotes the direction of the i^{th} counter.

As for the alarm clock machines, the input to a k-restless counter machine is simply an instantaneous description, the entire computation is determined by the machine's definition and its initial configuration.

Proposition 3.3.1 An acyclic k-adder machine can be simulated by a $(k^2 + k + 4)$-restless counter machine.

Proof:

We will simulate each alarm clock A_i by two restless counters, to which we call *morning* (M_i) and *evening* (E_i). Each step of the alarm clock machine will be simulated by 4 steps of the restless counter machine. We will see that all zero events of morning counters are separated by multiples of 4 time steps. We want a clock alarming to correspond to a zero event of the morning counter. We also want the finite control to be constrained to the restrictions we have imposed in the previous section: it is allowed to change the directions of the restless counters only during 5 time steps after being awakened. The control is awakened at every zero event. Another restriction is that, at any time step, for any i, there will be at most O(1) zero events before the next zero event for i.

We assume that the universal alarm clock machine runs a valid simulation of an acyclic adder machine, and thus behaves as described in the previous section. In order to prove that we can simulate an alarm clock machine in the conditions of proposition 3.2.1, we will use some particular features of that simulation. To understand this proof it is then necessary to read the proof of proposition 3.2.1.

The behavior of the restless counters is simple: we will consider that M_i and E_i are in a cycle, having a phase shift equal to 2p time steps, where p is the period of the alarm clock machine (this value, as we will see, doesn't have to be kept in memory). Each of the counters counts up to $2p - 1$, then down to 0, stays at 0 for two time steps and starts counting up again:

```
... 0 0 1 2 3 4 ... (2p-1) ... 4 3 2 1 0 0 0 1 2 3 ...
```

When one of the counters reaches zero, at time t, the finite control is awakened, and starts decreasing the other counter at time $t + 1$ and starts increasing the counter that reached 0 at time $t + 2$. This maintains the system in an oscillatory dynamics with period $4p$. This 2-counter system can simulate an alarm clock with constant period p. All we have to do now is to see how we can implement the operations delay and lengthen.

To implement these operations we will assume, without loss of generality, that the period of the clock is greater than 1. This causes no problems because, as we saw in the previous section, in our simulation of adder machines, all clocks begin with period greater than 10. We also assume that none of the restless counters equals 0, and that it is known which restless

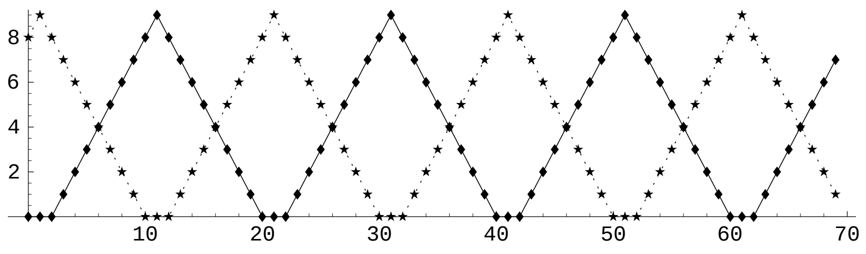

Figure 3.8: Restless counters with period 5 in usual cycle.

counter is increasing and which is decreasing. Once again, these assumptions cause no difficulties, when we restrict our analysis to the alarm clock machines who behave as described in the proof of proposition 3.2.1. In this context, the clocks will be delayed or lengthened in very specific conditions. During the classification phase, only the auxiliary clocks are delayed, and they are delayed just after alarming, so we know that the morning counter is being increasing. At the end of the cycle, all the auxiliary clocks have just alarmed and therefore they are in the "morning", and from our construction, all the adder clocks are in the "evening".[7]

To delay a clock, the finite control increases the restless counter it had previously been decreasing, and decreases the restless counter it had previously been increasing both a duration of two time steps, and then resumes its normal operations, see figure 3.9. We stress the fact that the operation of delaying is performed in 4 time steps. In the figure, the status of the restless counters is exactly the same at time steps 14 and 18, and that corresponds to a delay of one time step in the alarm clock machine.

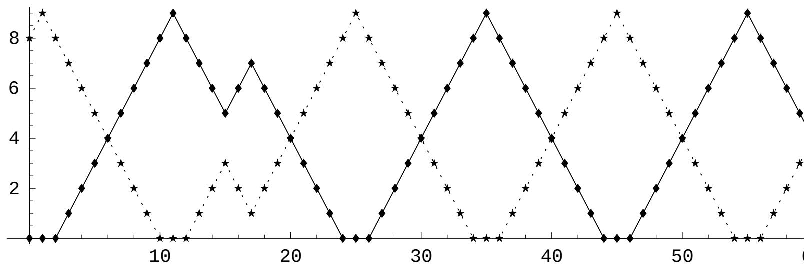

Figure 3.9: Delaying the restless counters.

To lengthen the day's period is not as simple as it may seem at first sight.

[7]Both the operations used in proof 3.2.1 and the functioning of the restless counters we present are different from the ones presented in [KS96]. First of all, the operation of lengthen performed on the restless counters didn't coincide with the one on the clocks. Also, at the end of the cycle, the adder clocks could be either in the morning or the evening, although the previous knowledge of which counter would be increasing was necessary and assumed.

In case we are in the morning period, all we have to do is to increment the evening counter 1 time step and then let it go back to normal, see figure 3.10. once again this operation can be done within the same time step of the alarm clock machine.

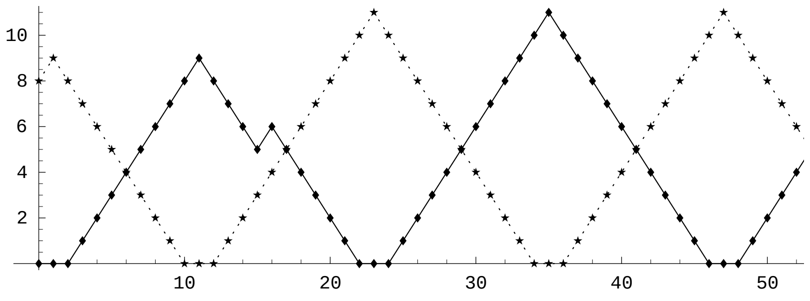

Figure 3.10: Lengthening the period of the restless counters.

If, on the other hand, we are in the evening period, then a similar attempt, see figure 3.11 would cause the lengthening of the period, but only a half delay (please recall from definition 3.2.1 that the operation of lengthening the period of the clocks also delays them)[7]. Therefore, in addition to the time step increasing the morning counter, we have another time step when we are both increasing the morning counter and decreasing the evening counter. Figure 3.12 illustrates this situation.

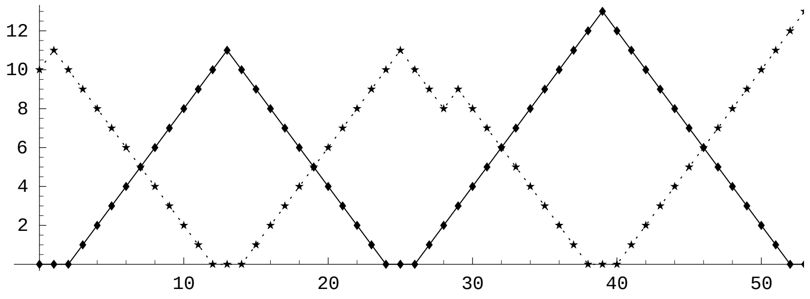

Figure 3.11: Wrong attempt at lengthening the period of the restless counters.

During the phase of comparing the clocks, if clock A_i is not synchronized with A_j, then the alarm clock machine delays A_{ij} once or twice (depending on whether A_i alarms sooner than A_j or not) in the 2 time steps (of the alarm clock machine) after A_{ij} alarms. This can be easily accomplished by using the time steps $k + 4$ to $k + 11$ (assuming A_{ij} alarms at time step k). Once all the comparisons are made, A_0 and A_{00} alarm simultaneously at time step t, and all the information about the comparisons will be available at time step $t + 8$. We will need to perform all the necessary operations between time step $t+8$ and time step $t+19$. At time steps $t+8$ to $t+15$ we

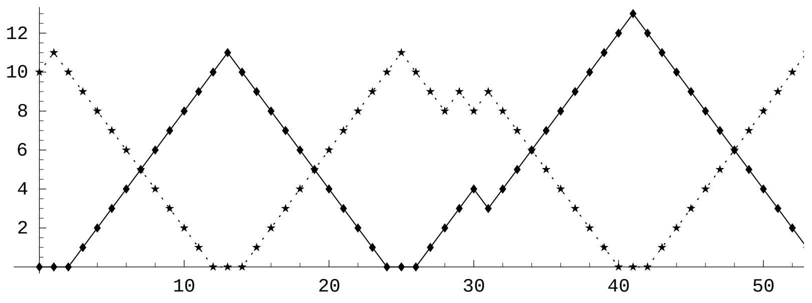

Figure 3.12: Lengthening the period of the restless counters.

can delay A_{00} and the other auxiliary clocks that didn't alarm at time $t+8$ so that at time step $t+15$ all the auxiliary clocks are synchronized with A_{00} again. The delaying of the adder clocks can also be made in time steps $t+8$ to $t+11$. At time step $t+12$ to $t+15$, we can lengthen the period of all clocks, and finally we can do the second lengthening between steps $t+16$ and $t+19$. Notice that we can start the lengthening at time $t+12$ because we have finished lengthening the adder clocks at time $t+11$. For the reasons mentioned before, we cannot simultaneously delay and lengthen the adder clocks, as we do for the auxiliary clocks.

Chapter 4

Implementing Sigmoidal Networks

In this chapter we present the details of the neural network that simulates an alarm clock machine (as introduced in the previous chapter). This network was first introduced in [KS96]. The analysis of this network will be the main concern of this thesis. The main goal is to discover whether the theorem[1] concerning the universality of this network does, in fact, hold.

In the beginning of the chapter, we will address some important features of the sigmoid and other similar functions, that are used in the proof. On the next three sections we will see each of the components of the network separately. Finally, on the last section, we will deal with the functioning of the network as a whole, and we will also address the initialization of the network.

4.1 A few important properties

Before starting the description of the neural network, we begin by stating and illustrating four properties of the sigmoid that we will use in the construction of this neural network. The understanding of these properties is essential to the understanding of the functioning of the network we will describe in the next sections.

These properties are also of extreme importance to understand the generality of the result we are trying to prove. If this result does, in fact, hold, then it can be easily extended to any function that exhibits these properties.

Definition 4.1.1 (Property 1) A function φ has Property 1 if it is monotonically non-decreasing for $|x| > a$, for some given $a \in \mathbb{R}^+$ such that, if $x < -a$, then $\varphi(x) \in [-1, -\frac{1}{2}]$ and if $x > a$, then $\varphi(x) \in [\frac{1}{2}, 1]$.

[1]Theorem 18, [Sie99], page 100.

In the particular case of the function $\varrho(x) = \frac{2}{1+e^{-x}} - 1$, then it is immediate to verify that this property holds for $a = \ln 3$.

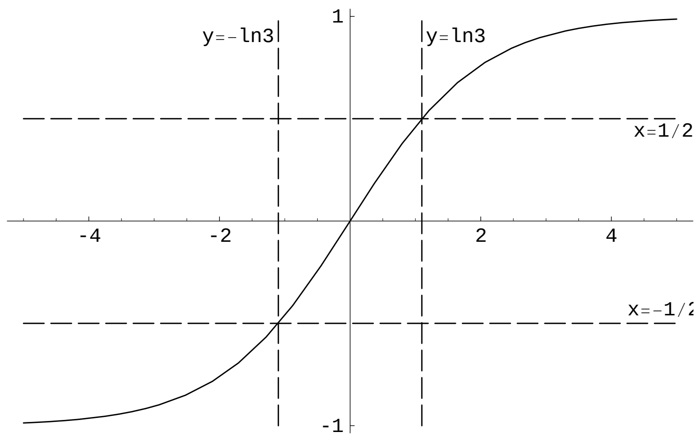

Figure 4.1: Function ϱ.

Definition 4.1.2 $z \in \mathbb{R}$ is a *fixed point* for $f : \mathbb{R} \to \mathbb{R}$ if $f(z) = z$. If z is a fixed point for f, z is said to be a *stable* fixed point for f, or an *attractor* for f, if there is an interval I such that $z \in I$ and $|z - f(x)| \leq |z - x|, \forall x \in I$. Similarly, z is said to be an *unstable* fixed point for f, or a *repulsor* for f, if there is an interval I such that $z \in I$ and $|z - f(x)| \geq |z - x|, \forall x \in I$.

Definition 4.1.3 (Property 2) A function φ has Property 2 if for every constant b, for which the slope of $\lambda x.\varphi(bx)$ at 0 is larger than 1, $\lambda x.\varphi(bx)$ has three fixed points: 0, A and -A, for some constant A and if, moreover, given any constant c, we can choose b sufficiently large so that $\lambda x.\varphi(bx + c)$ has also three fixed points.

Returning to our example, $\frac{\partial(\lambda x.\varrho(bx))}{\partial x} = \frac{2be^{-bx}}{(1+e^{-bx})^2}$.

For $x = 0$, we have $\frac{\partial(\lambda x.\varrho(bx))}{\partial x}|_{x=0} = \frac{b}{2}$.

Therefore the slope at 0 is $\frac{b}{2}$ and this feature requires $b > 2$.

It is immediate that $\lambda x.\varrho(bx)$ has a fixed point in 0, for all values of b.

For $b > 2$ we can guarantee the existence of symmetric fixed points by noting that

$$\varrho(bx) = x \text{ iff } \frac{2}{1+e^{-bx}} - 1 = x$$

After some simple manipulation, and assuming $x \neq 0$, we obtain equivalently $b = \frac{\ln(\frac{1-x}{x+1})}{-x}$. The first thing we show is that, independently of the values of b, the fixed points are always located in the [-1,1] interval. Any values of x that satisfies the previous equality for some b must satisfy also $\frac{1-x}{x+1} > 0$.

$\frac{1-x}{x+1} > 0 \Leftrightarrow \frac{2}{x+1} - 1 > 0$

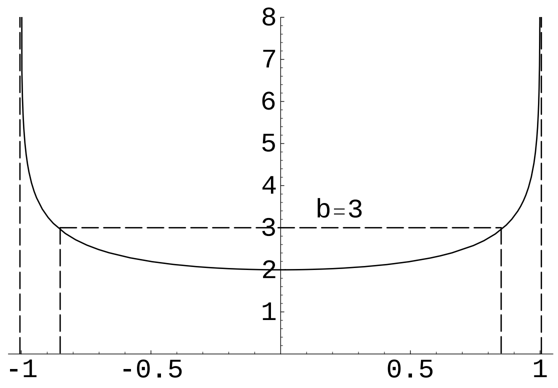

Figure 4.2: $\lambda x.\frac{\ln(\frac{1-x}{x+1})}{-x}$ For $b > 2$, the values of the fixed points are the values in the horizontal axis corresponding to the intersection point between the function and the constant line b.

$\Leftrightarrow 0 < \frac{x+1}{2} < 1$
$\Leftrightarrow -1 < x < 1$

The function $\lambda x.\frac{\ln(\frac{1-x}{x+1})}{-x}$ always assumes values greater than 2, and converges to 2 when x converges to 0. This function is an even function, and therefore, for $b > 2$ there are always two symmetrical solutions to the previous equation (compare with figure 4.2). A is always smaller than 1, and $A \to 1$ when $b \to +\infty$. Using 10 decimal digits in the precision we obtain the values:

b=6	A=0.9949015285
b=12	A=0.9999877098
b=20	A=0.9999999958

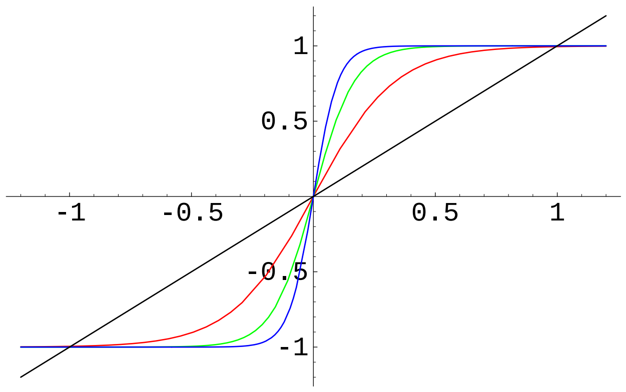

Figure 4.3: Joint plot of $\lambda x.x$, $\lambda x.\varrho(6x)$, $\lambda x.\varrho(12x)$, and $\lambda x.\varrho(20x)$.

In figure 4.3 we can see the plot of $\varrho(6x)$ in red, $\varrho(12x)$ in green, and $\varrho(20x)$ in blue. As b grows, the function becomes more and more similar to the threshold function.

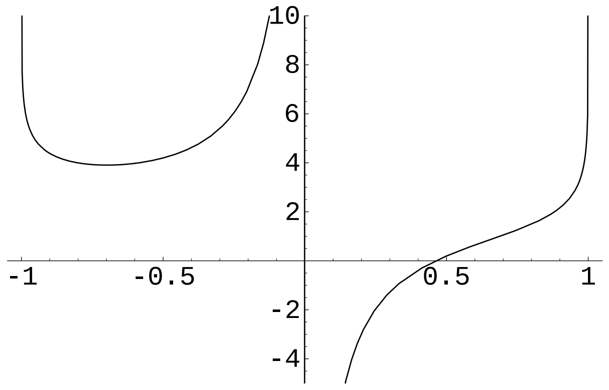

Figure 4.4: $\lambda x.\frac{\ln(\frac{1-x}{x+1})+1}{-x}$.

Let c be a constant. In order to conclude that ϱ has property 2, we want to show that for b sufficiently large, the function $\varrho(bx+c)$ has also three fixed points. We begin by adapting the equation for the fixed points:

$$\varrho(bx+c) = x \text{ iff } x = \frac{2}{1+e^{-bx-c}} - 1 \text{ iff } b = \frac{\ln(\frac{1-x}{x+1})+c}{-x}$$

Once again the fixed points remain in the $[-1,1]$ interval. The behavior of this function, for a fixed c, depends on the signal of c. If c is positive, then $\lambda x.\frac{\ln(\frac{1-x}{x+1})+c}{-x}$ is positive for all $-1 < x < 0$, the function grows to $+\infty$ when x approaches -1, 1, and 0 by negative values. The function decreases to $-\infty$ when x approaches 0 by positive values. See figure 4.4 for a better understanding.

This means that for every b, there is exactly one fixed point x, with $x > 0$ and the number of negative fixed points depends on the value of b. For b greater than a given value we will have two negative fixed points. As b grows, one of them will become increasingly near -1, while the other converges to 0. The positive fixed point converges to 1. Moreover, we can easily determine the smaller value of b such that $\varrho(bx+c)$ has 3 fixed points: all we have to do is to minimize $\frac{\ln(\frac{1-x}{x+1})+c}{-x}$, with $x < 0$.

For the first integer values of c, we show a table of the limit values of b such that the equation has three fixed points beyond these values.

c	Limit value of b	c	Limit value of b
1	3.906557937512	7	11.047911984021
2	5.243962926453	8	12.147268315137
3	6.477395880086	9	13.236854293244
4	7.658935761882	10	14.318482404240
5	8.808536311856	11	15.393494540308
6	9.936245470418	12	16.462913765765

The two external fixed points are not equal in size anymore, provided that $c \neq 0$, and they thus are denoted by A_1(closer to -1) and A_2(closer to 1).[2] For negative values of c, we would have a symmetric behavior and the middle fixed point would be positive. We will only be interested in positive values of c, because in our implementation of the restless counters we will need to store positive values arbitrarily close to 0 and cannot allow this positive fixed point. This should become clear in the next section.

In order to study the order of convergence of these attractors, let us recall a basic theorem of Numerical Analysis.[3]

Definition 4.1.4 A function $f : \mathbb{R} \to \mathbb{R}$, continuous in $[a, b]$, is *contractive in* $[a, b]$ if there is a constant $0 \leq L < 1$ such that

$$|f(x) - f(y)| \leq L|x - y|, \forall x, y \in [a, b].$$

Proposition 4.1.1 (Fixed Point Theorem) If g is a contractive function in $[a, b]$, and if $g([a, b]) \subseteq [a, b]$, then:

- g has one and only one fixed point z in $[a, b]$;
- the sequence $z_{n+1} = g(z_n)$ converges to that fixed point z, given any z_0 in $[a, b]$
- the difference between the exact value z and the approximation z_n can be bounded by:
  $|z - z_n| \leq L^n|z - z_0||$
  $|z - z_n| \leq \frac{1}{1-L}|z_{n+1} - z_n|$.

The fixed point method consists in, given a function in the conditions of the previous theorem, choose an adequate initial value and repeatedly apply the function until we are sufficiently near the exact value of the solution. We now state a corollary of the previous theorem that will allow us to analyze the convergence of the fixed point method when applied to our function ϱ.

Definition 4.1.5 Let (x_n) be a sequence converging to z, and let $e_n = x_n - z, \forall n \in \mathbb{N}$. If there exists the limit $\lim_{m\to\infty} \frac{|e_{m+1}|}{|e_m|^p} = K$, with $0 < K < \infty$, we call p the *order of convergence* and we call K the *asymptotic convergence coefficient.* If $p = 1$ the convergence is said to be linear.

Proposition 4.1.2 Let $g : \mathbb{R} \to \mathbb{R}$ be a function and $z \in \mathbb{R}$ such that $g(z) = z$. If $0 < |g'(x)| < 1$, for every x within an interval containing z,

[2] In [KS96], the middle fixed point is considered always to be zero and the possibility that the function doesn't have the three fixed points is not considered.

[3] The results 4.1.1 and 4.1.2 may be found in any Numerical Analysis introductory book. The statements of these results were taken from [Alv99]. We recommend the reader to refer to [Atk88], especially if he is unfamiliar with Portuguese language.

then z is an attractor for g within the given interval, the convergence of the fixed point method is linear and the asymptotic convergence coefficient equals $g'(z)$.

$\forall x \in \mathbb{R}\ 0 < \varrho'(x) \leq 1/2$: hence, $\forall x \in \mathbb{R}\ 0 < \frac{\partial \varrho(bx+c)}{\partial x} \leq b/2$, regardless of the values of c. It is immediate from proposition 4.1.2, that the convergence to the fixed points is linear[4] if we can show that on a neighborhood of these fixed points $\frac{\partial \varrho(bx+c)}{\partial x} < 1$.

$\frac{\partial \varrho(bx+c)}{\partial x} = \frac{b}{1+Cosh(bx+c)}$

$\frac{b}{1+Cosh(bx+c)} = 1 \Leftrightarrow x = \frac{-c \pm ArcCosh(b-1)}{b}$

In figure 4.5 we can see zones of convergence and divergence of the fixed point method for different values of b, for c=0.[5]

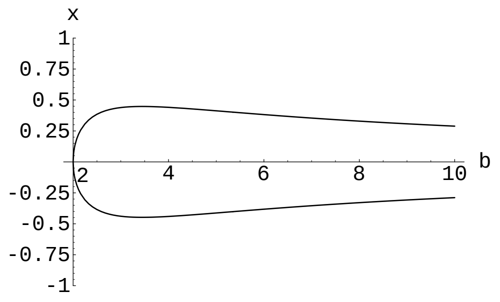

Figure 4.5: $x = \frac{\pm ArcCosh(b-1)}{b}$, $c = 0$.

The fixed points A_1 and A_2 are stable and constitute linear attractors (for all x different from the middle fixed point), and the middle fixed point (x=0, when c=0) is unstable. We can see what happens for c=2 in figures 4.6 and 4.7.

Definition 4.1.6 (Property 3) A function φ has Property 3 if it has Property 2 and it is differentiable twice around its attractors.

In the case of our sigmoid, this is evident by the very definition of the sigmoid function. Explicitly, the second derivative of the sigmoid is given by: $\varrho''(x) = -\frac{2e^x(-1+e^x)}{(1+e^x)^3}$.

[4]The fact that the convergence is linear means that, as defined in 4.1.5, $\lim_{m\to\infty} \frac{|e_{m+1}|}{|e_m|}$ is a non-zero constant. Let k be that constant. That means that we can make $|e_{m+1}| \approx k|e_m| \approx k^2|e_m - 1| \approx ... \approx k^m|e_1|$, so although the convergence is linear, is varies exponentially with m.

[5]In figures 4.5 and 4.7 the extreme case when $b = 2$ is a simple one because if $b = 2$, then there is only one fixed point.

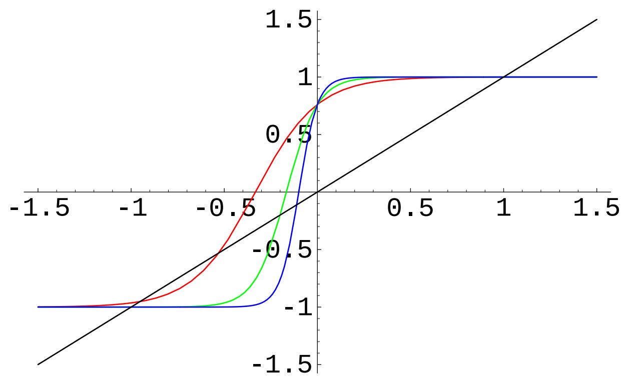

Figure 4.6: Joint plot of $\lambda x.x$, $\lambda x.\varrho(6x{+}2)$, $\lambda x.\varrho(12x{+}2)$, and $\lambda x.\varrho(20x{+}2)$.

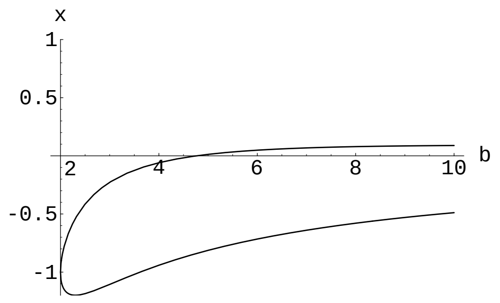

Figure 4.7: $x = \frac{-2 \pm ArcCosh(b-1)}{b}$, $c = 2$.

Definition 4.1.7 (Property 4) A function φ is said to have Property 4 if $\varphi(0) = \varphi''(0) = 0$ and $\varphi'(0) = \frac{1}{2}$. (Hence, if x is a small number, then $\varphi(x) \sim x/2$.)

ϱ verifies the above conditions, i.e., $\varrho(0) = \varrho''(0) = 0$ and $\varrho'(0) = \frac{1}{2}$. By the Taylor series expansion around 0, $\varrho(x) = x/2 + O(x^3)$. [6]

The four properties we have seen in this section will be used to design a neural network to simulate an alarm clock machine with restless counters, as defined in the previous chapter. For any alarm clock machine $\mathcal{A}$, the network $\mathcal{N}$ that simulates $\mathcal{A}$ consists of three parts: a finite control, a set of restless counters and a set of flip-flops. The finite control simulates the finite control of the alarm clock machine, the restless counters will simulate the clocks, and finally the flip-flops will be used as a bridge between the finite control and the restless counters.

[6]In [KS96], it is claimed that $\varrho(x) = x + O(x^3)$, which is obviously a minor mistake.

4.2 Implementing the restless counters

We will implement each restless counter by three sigmoidal neurons: one, called the *rc* neuron, will be used to store the value of the restless counter, and the other two will be used to help performing the Inc/Dec operations.

The value $n \in \mathbb{N}$ will be stored in the *rc* neuron as a value "close" to B^{-n}, where B is a constant greater than 2. In particular, a value 0 in a restless counter will be implemented as an *rc* neuron holding a value close to 1. Given that representation, to increase the counter by 1 is simply to divide by B, and to decrease it is to multiply by B. So, we want our *rc* neuron to be multiplied either by B or $\frac{1}{B}$, at every time step.

As already mentioned in the previous section, our network will have three components. As we will see in the next section, the flip-flops will receive their input from the finite control, and transmit the value A_1 or A_2 to the counter, depending on whether the counter should be increased or decreased.

Let F be the input signal coming from the i^{th} flip-flop and x_i the value of the i^{th} *rc* neuron. We will use the approximation:

$$\varrho(F + cx_i) - \varrho(F) \approx \varrho'(F)cx_i$$

for sufficiently small c and $|x_i| < 1$ (recall Property 3 of the previous section).[7]

Let us define the update of the *rc* neurons by:

$$\begin{aligned} x_i(t+1) &= \varrho[\alpha_{c1}\varrho(\alpha_{c2}F + \alpha_{c3} + \alpha_{c4}x_i(t)) - \alpha_{c1}\varrho(\alpha_{c2}F + \alpha_{c3}) + \alpha_{c5}x_i(t)] \\ &\approx \varrho[(\alpha_{c1}\varrho'(\alpha_{c2}F + \alpha_{c3}))\alpha_{c4}x_i(t) + \alpha_{c5}x_i(t)] \end{aligned}$$

By a suitable choice of constants α_{c1},...,α_{c5}, we have:

$$\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}A_1 + \alpha_{c3}) + \alpha_{c5} = 2B$$
$$\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}A_2 + \alpha_{c3}) + \alpha_{c5} = \frac{2}{B}$$

If the value of x_i is close enough to 0, we can approximate (using Property 4 from the last section)

$$\varrho[(\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}F + \alpha_{c3}) + \alpha_{c5})x_i] \approx (\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}F + \alpha_{c3}) + \alpha_{c5})x_i/2.$$ [8]

Combining these two pieces of information is now trivial: we conclude that x_i is multiplied by B whenever $F = A_1$ and divided by B if $F = A_2$. Note also that when x_i is approximately 1, and is "approximately multiplied" by B, then it will in fact be drawn towards a fixed point of the above equation. This acts as a form of *error correction*, because as we have seen in the functioning of the restless counters, every time a restless counter reaches 0, it stays at zero for 2 time steps. During those time steps, the errors

[7] We will see that we will be able to choose c as small as desired. Therefore, by imposing an upper bound on the absolute value of x_i, we can assure the existence of a c such that the error committed in this approximation is less than any previously determined constant of our choice. This restriction, however is not very important, since we will have to take an extra care with the high values of x_i in another approximation, as we will see very soon.

[8] This approximation is only accurate when the value of x_i is sufficiently small so that its cube can be ignored. This is a problem only in the first steps of each cycle.

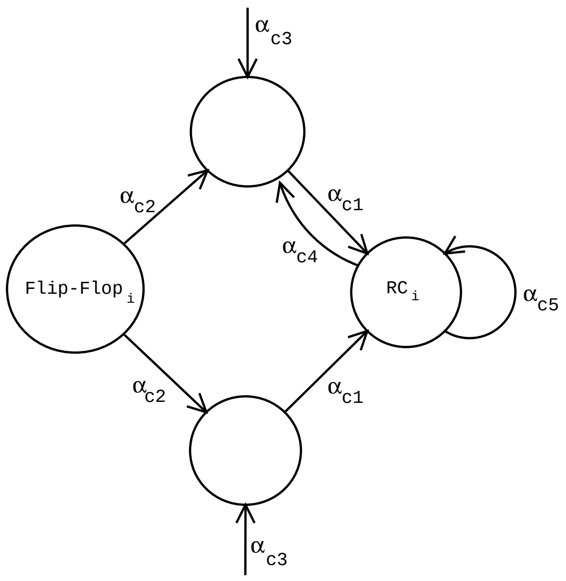

Figure 4.8: Implementation of a restless counter.

committed during the rest of the cycle are corrected, to some extent.

Giving a concrete example for our choice of variables, we must begin by noting that all the other constants are defined based on the fixed A_1 and A_2. Let us use the fixed points of the function $\varrho(30x + 1/2)$. The values of the fixed points are (up to 15 decimal digits) $A_1 = -0.999999999999691$ and $A_2 = 0.999999999999886$.

Next we can start choosing the other constants α_{c1},...,α_{c5} in order to satisfy the previous equations. That is not a totally trivial question. From the equations $\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}A_1+\alpha_{c3})+\alpha_{c5} = 2B$ and $\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}A_2+\alpha_{c3})+\alpha_{c5} = \frac{2}{B}$ we can easily get to $\varrho'(\alpha_{c2}A_1+\alpha_{c3}) = \frac{2B-\alpha_{c5}}{\alpha_{c1}\alpha_{c4}}$ and $\varrho'(\alpha_{c2}A_2+\alpha_{c3}) = \frac{B/2-\alpha_{c5}}{\alpha_{c1}\alpha_{c4}}$.

After viewing the graphic of $\varrho'(\alpha_{c2}A_1+\alpha_{c3})-\varrho'(\alpha_{c2}A_2+\alpha_{c3})$ (the terms in α_{c5} cancel out and we only obtain B multiplied by constants) we quickly conclude that there exists a solution with $\alpha_{c2} = 1.2$ and $\alpha_{c3} = -0.3$.[9] We calculate the values $C_1 = \varrho'(\alpha_{c2}A_1 + \alpha_{c3}) \approx 0.298292904140736$ and $C_2 = \varrho'(\alpha_{c2}A_2 + \alpha_{c3}) \approx 0.411000614684551$.
Next, we are still free to choose B=3 (recall that $B > 2$). Now we have the equations $\frac{6-\alpha_{c5}}{\alpha_{c1}\alpha_{c4}} = C_1$ and $\frac{2/3-\alpha_{c5}}{\alpha_{c1}\alpha_{c4}} = C_2$. With these equations we can determine $\alpha_{c5} \approx 20.1152320553272$ and $\alpha_{c1}\alpha_{c4} \approx -47.3200396636575$.

Finally, we choose α_{c4} to be small (0.001) and take $\alpha_{c1} = 47320$.

If, with these values for the constants, we simulate the functioning of the restless counter in a cycle with p=10 (the restless counter goes from 0

[9]All the numerical examples chosen in this section to illustrate the functioning of the network are original values, they were not taken from [KS96].

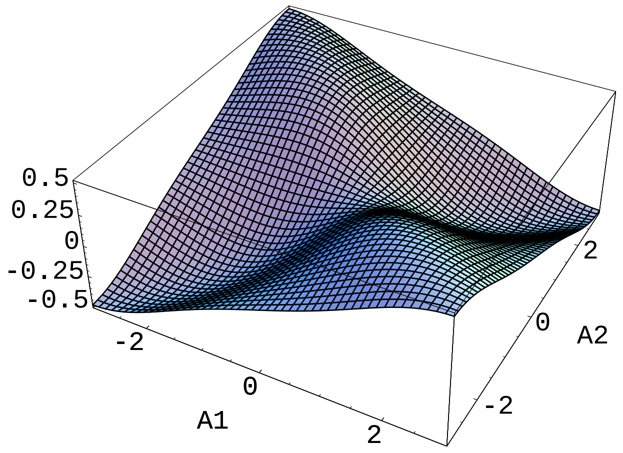

Figure 4.9: Plot of $\varrho'(\alpha_{c2}A_1 + \alpha_{c3}) - \varrho'(\alpha_{c2}A_2 + \alpha_{c3})$.

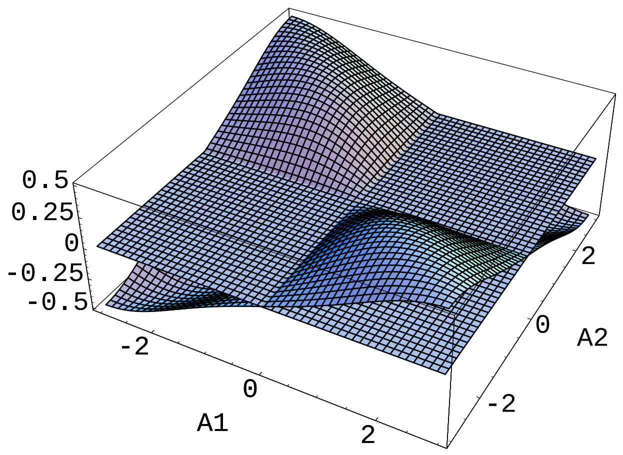

Figure 4.10: Joint plot of $\varrho'(\alpha_{c2}A_1 + \alpha_{c3}) - \varrho'(\alpha_{c2}A_2 + \alpha_{c3})$ and the plan 0, i.e., $\varrho'(\alpha_{c2}A_1 + \alpha_{c3}) = \varrho'(\alpha_{c2}A_2 + \alpha_{c3})$.

to 2p-1=19 and then goes back to 0 and stays at 0 for two time steps, figure 4.11), and we start with the value 0.994878198867128, value of a fixed point for the function $\varrho[\alpha_{c1}\varrho(\alpha_{c2}F + \alpha_{c3} + \alpha_{c4}x_i(t)) - \alpha_{c1}\varrho(\alpha_{c2}F + \alpha c3) + \alpha_{c5}x_i(t)]$, then we obtain successively the following values for the rc neuron:

0	0.99487819886712828433600 4996	21	$7.465564308628740706 * 10^{-9}$
1	0.32180274724412187481009 4142	22	$2.239669292576129 9259 * 10^{-8}$
2	0.10706806019384227855312 5796	23	$6.719007877615952 4634 * 10^{-8}$
3	0.03569769435806996836045 8874	24	$2.015702363183573 95617 * 10^{-7}$
4	0.01190128358585283330633 4497	25	$6.047107088639324 42812 * 10^{-7}$
5	0.00396736426552664640505 9143	26	$1.814132125770212 866631 * 10^{-6}$
6	0.00132248627213452832957 3043	27	$5.442396369880555 776719 * 10^{-6}$
7	0.00044083231654198846098 1680	28	0.00001632718904180370993 1214
8	0.00014694450309827334585 5598	29	0.00004898156648875479085 0410
9	0.00004898154528736963714 6612	30	0.00014694469303125993774 5664
10	0.00001632718668254607646 4624	31	0.00044083400214112184179 7125
11	$5.442396107633100922861 * 10^{-6}$	32	0.00132250079983507699994 3265
12	$1.814132096635515887342 * 10^{-6}$	33	0.00396747766202788566254 8622
13	$6.04710705629521559624 * 10^{-7}$	34	0.01190183567832336827969 4907
14	$2.01570235959958914602 * 10^{-7}$	35	0.03569002423112339197119 5463
15	$6.7190078736666329338 * 10^{-8}$	36	0.10665996730569516302013 75075
16	$2.2396692921482860751 * 10^{-8}$	37	0.30946569224694139949017 13524
17	$7.465564308189926192 * 10^{-9}$	38	0.72974535722751915279863 8671
18	$2.488521436177639030 * 10^{-9}$	39	0.97516332346506945805104 6091
19	$8.29507145405249710 * 10^{-10}$	40	0.99423790986523826450601 5668
20	$2.488521436214206915 * 10^{-9}$		

The values from the table above are the values of the rc neuron while the restless counter performs a cycle with period 10, as in figure 4.11. They show how quickly the value stored on the rc becomes extremely small. Therefore very small errors committed in the middle of the cycle, when the value of the counter approaches $2p - 1$, and the value of the rc neuron approaches 0, may become too large as they are multiplied by B^n in the second half of the cycle.

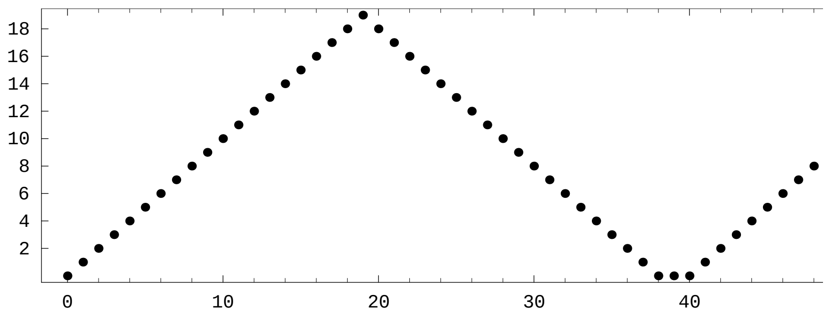

Figure 4.11: Restless counter of period 10.

The ratio between the initial value and the value obtained after this cycle is 0.999356. Therefore we have an implementation (with small errors), of a restless counter! Of course the control of the errors committed is of crucial importance. In fact it is the very central question: are the errors committed controlled? We will not answer this question here, because it is worthy of a chapter of its own. The next chapter will be entirely dedicated to error analysis. For the moment we are only concerned with describing the network as detailed and clearly as possible, and we hope to have made our point: shown how to implement restless counters with sigmoidal neurons in such a way that as long as their inputs from the flip-flops are correct, they will perform the usual cycle of the restless counters.

The delay and lengthen operations will be described when we analyze the finite control.

4.3 Bi-directional flip-flops

As we saw in the previous section, we will need flip-flop devices to give the restless counters their direction. These flip-flop devices must receive their input from the finite control and pass on the information to the restless counters in the manner discussed in the previous section. Therefore, the flip-flop must have a neuron that takes on two distinct values A_1 and A_2, corresponding to increasing and decreasing. We will implement this feature recurring to approximation once more. Instead of having neurons that take one of two values, we will have neurons that will be converging to one of two fixed points.

We will assume, and see in the next section that it is a reasonable assumption, that the finite control has two output lines, implemented as neurons, for each restless counter. We will call these output neurons *Start-Inc$_i$* and *Start-Dec$_i$*. Their expected behavior is very simple. Most of the time these two neurons are in an inactive state (≈ 0), and they will only become active when there is a need for the restless counter i to change direction. Whenever one of these neurons becomes active, it can reset the value of the corresponding flip-flop. During the quiet periods, the flip-flop maintains its current state, converging to one of its two stable states, A_1 or A_2.

The update equation of each flip-flop is

$ff_i = \varrho(\alpha_{f1}(\text{Start-Inc}_i - \text{Start-Dec}_i) + \alpha_{f2} ff_i + \alpha_{f3})$,

where α_{f1}, α_{f2}, and α_{f3} are suitably chosen constants (Property 2).

As we have seen in the previous section (page 59), in our concretization of the appropriate constants for the restless counters, we began by choosing to use the fixed points of $\varrho(30x+1/2)$: this means that we chose α_{f2} to be 30 and α_{f3} to be 1/2. With this choice of variables, the flip-flop will be assured to function properly on the quiet periods, when both Start-Inc$_i$ and Start-Dec$_i$ are inactive. We only have to establish now a constant α_{f1} sufficiently large so that the activity of one of the output neurons changes the state of the flip-flop, and sufficiently small so that for values of the counter close to

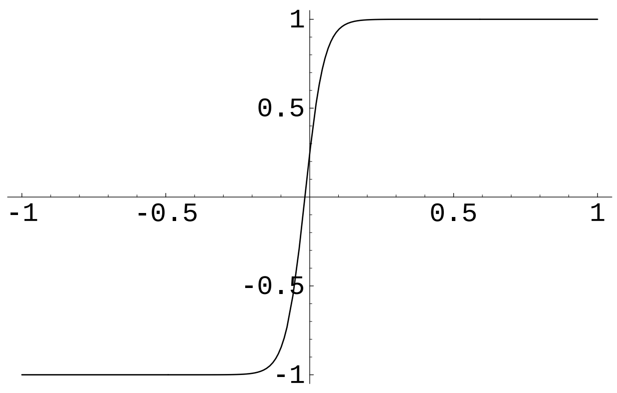

Figure 4.12: $\varrho(30x + \frac{1}{2})$.

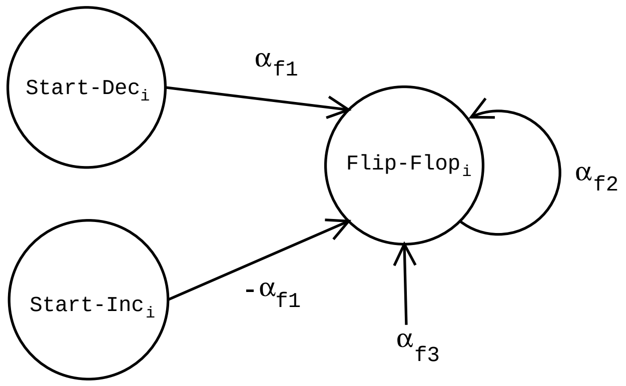

Figure 4.13: Implementation of flip-flops i.

0, the convergence remains unaltered. A suitable value for α_{f1} that would verify this property would be $\alpha_{f1} = 35$. The verification is straightforward. In figure 4.3 we can see an plot of the value of a flip-flop neuron converging to $A_1(\approx -1)$, as a function of the value of *Start-Inc$_i$*-*Start-Dec$_i$* From figure 4.3 we can see clearly that in order to change to the neighborhood of A_2, the result of that difference must be above 0.8. Once again we leave the error analysis for the next chapter.

4.4 The finite control

The only component left unspecified in our neural network is now the finite control. Kleene showed in 1956 [Kle56] that any finite automaton can be simulated by a network of threshold devices. If the finite control has finite memory, i.e., if it depends only on the last O(1) time steps, the resulting network can be made into a feed-forward one.

But we don't want a network of threshold devices, we want a network of

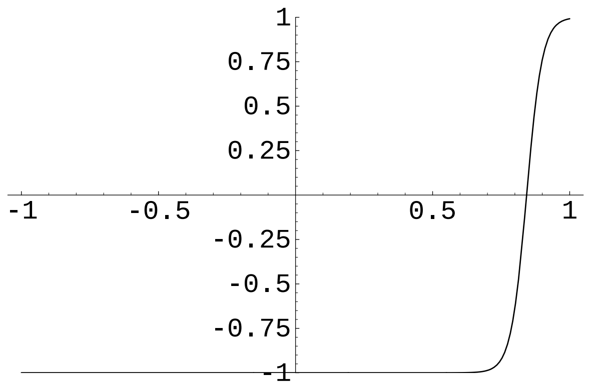

Figure 4.14: $\varrho(30 + 1/2 + 35(\textit{Start-Inc}_i\text{-}\textit{Start-Dec}_i))$.

sigmoidal neurons. Therefore we substitute each threshold device

$$x_i(t+1) = \mathcal{H}(\textstyle\sum_{j=1}^{N} a_{ij}x_j + \theta_i)$$

with a sigmoidal device

$$x_i(t+1) = \varrho(\alpha_a(\textstyle\sum_{j=1}^{N} a_{ij}x_j + \theta_i))$$

for a large and fixed (the same for all neurons) constant α_a. Once this constant α_a is fixed, there is a determined neighborhood of 0 such that if the result of the sum above is outside that neighborhood, then the result of the sigmoid activation function will be very close to the result we would obtain if we were using the Heaviside activation function. In fact, we can make that approximation as close as desired: this claim is simply Property 1 of section 4.1.

We are treating the output values of the activation function as discrete, because we ensure that these values fall within small neighborhoods of 1 and -1; however the range of possible values of ϱ is continuous and so we have to be more careful than that with respect to the control of the errors. This property will be analyzed more carefully in the next chapter.

We also know a few more properties of the finite control. As we have seen in the previous section, for each restless counter i, we demand that the finite control has two output lines, implemented as neurons, and called *Start-Inc*$_i$ and *Start-Dec*$_i$, satisfying the properties we used in the last section: at the instant when *Start-Inc*$_i$ is active (i.e., ~ 1), it means that from that moment on, the restless counter i should be continually increased. Similarly, when *Start-Dec*$_i$ is active, it means that restless counter i should begin to be continually decreased. Most of the time, both output lines are in an inactive state (i.e., ~ 0). It will never be the case that both signals are simultaneously active.

But we know even more about the finite control: its inputs come from the restless counters. We have seen when we dealt with the implementation of the restless counters that a counter being 0 was stored as a neuron being close to 1. Therefore the input lines of the finite control should be such that values relatively close to 1 would be treated as exact 1's and small values would be ignored.

The part of the finite control that simulates the finite control of the adder machine that our alarm clock machine is miming is straightforward. We will assume that part is working and will no longer worry about it.

Performing the specific operations on the restless counters is another story. Will we see these in more detail. One of the features that our finite control must be able to do is to perform the delays that occur during the phase of comparing clocks (recall chapter 3). For that specific purpose there will be two sigmoidal neurons for each auxiliary clock A_{ij}, called Delay1_{ij} and Delay2_{ij}. These neurons will become active only when clock A_{ij} needs to be delayed. These neurons have connections and weights as in the figure 4.15. The weights that are not specified in the diagram are all 1. The neurons that have no name have no other task than to maintain the synchrony of the net.

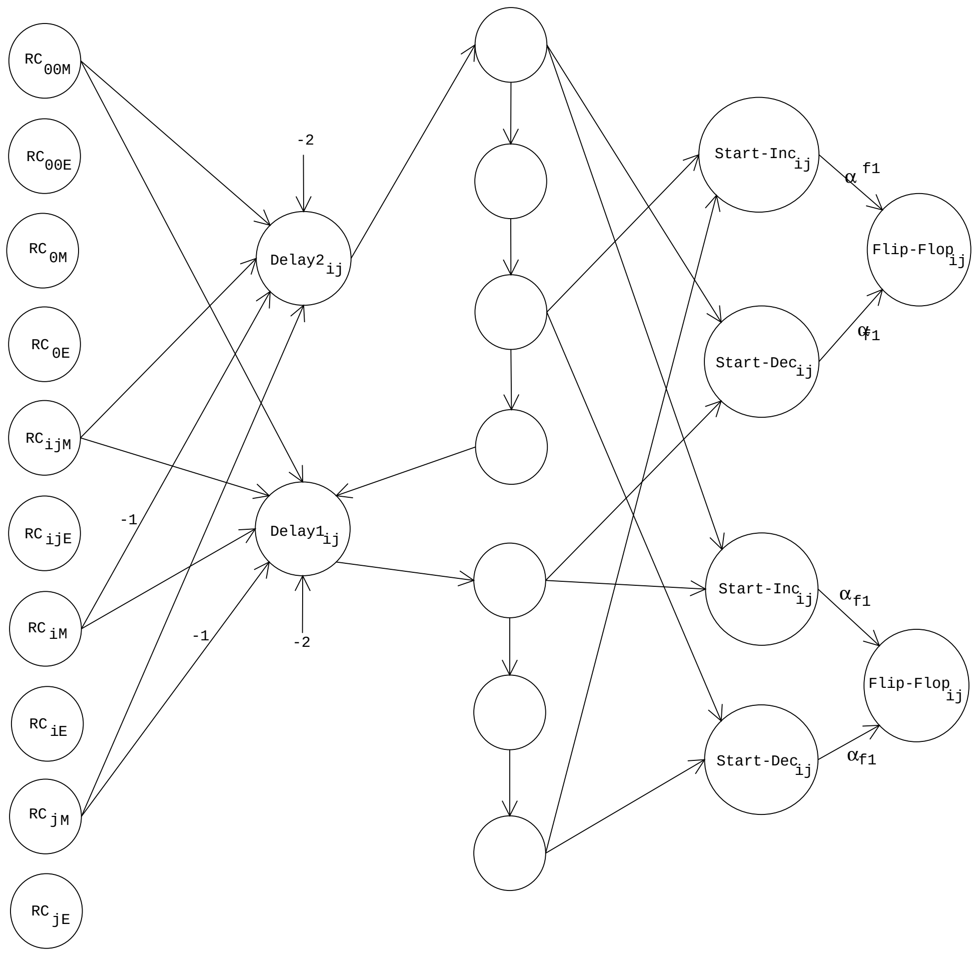

Figure 4.15: Implementation of the classifying of the clocks.

From the discussion above it is easy to see that the delaying of the clocks resulting from the clock comparisons can be achieved with these neurons.

Now we show how to implement in the finite control the part of the cycle that corresponds to the ending of the cycle when the clocks are all classified and we only have to "read" their classifying and perform accordingly. The way to achieve this purpose rests on two basic ideas. First, that action can be made by a Boolean circuit, as in example 3.1.4. The results of the comparisons amongst the clocks that were provided to that boolean circuit are provided in the neural network by the pattern of the rc-neurons becoming active. Second, a Boolean circuit can be computed by an equivalent 3-layer circuit if we allow the And and Or gates to have any finite number of inputs; and it is possible to use sigmoidal neurons as And and Or gates with any finite number of inputs. The possibility of this construction in 3 layers is important, because we cannot have boolean circuits of unbounded depth, otherwise we would not be able to compute the adders to add in time.

Chapter 5

Controlling the Errors

In this chapter we present the main result of our thesis. We prove that sigmoidal networks, as presented in the previous chapter, can simulate Turing machines, because the errors committed by the neurons used in implementing the network described in the last chapter are controllable. A major consequence of this result is that it is possible to simulate Turing machines with an analytic recurrent network. As already referred in the introduction to this thesis, this was believed to be impossible by some cientists working in this area. Concretely, Cristopher Moore, pointed out that the errors committed during the computation of the neural network were not controlled, by showing a flaw in the original paper [KS96], as we will discuss further on page 75, when we address how to control these errors.

Later in this chapter we will prove that the flip-flop devices can send to the restless counters signals with controlled errors. However, in order to simulate correctly the restless counters, this part of the network has to receive these values and store an approximation of the value B^{-v} good enough to always allow us to recover v, where v is the actual value of the restless counter. The restless counters are usually in a periodic cyclic movement, unless there are operations of delaying or lengthening performed on them. Each restless counter starts its "day" at 0, counts up to $2p-1$, then down to 0, stays at 0 for 2 time steps, and starts over again.

According to what we have seen in section 4.2, the update equation of a restless counter x_i is given by:

$$x_i(t+1) \;= \varrho(\alpha_{c1}\varrho(\alpha_{c2}F+\alpha_{c3}+\alpha_{c4}x_i(t)) - \alpha_{c1}\varrho(\alpha_{c2}F+\alpha_{c3}) + \alpha_{c5}x_i(t)).$$

Recall that for small y,

$$\varrho(E+y) - \varrho(E) \approx \varrho'(E)y.$$

We can choose constants $\alpha_{c1}, \alpha_{c2}, \alpha_{c3}, \alpha_{c4}, \alpha_{c5}$ such that

$$\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}A_1+\alpha_{c3}) + \alpha_{c5} = 2B$$
$$\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}A_2+\alpha_{c3}) + \alpha_{c5} = \tfrac{2}{B}.$$

Ideally, our finite control neurons would all have 0,1 values, our flip-flops would take on precisely two values (A_1,A_2) and the rc neuron would have the exact value B^{-v}. Unfortunately, it is inevitable that the neurons' values will deviate from their ideal values. To obtain our result, we show that these errors are controllable. The deviation from this ideal behavior is caused by the use of function ϱ, and it is controlled in three different levels:

1. the error caused by approximating the difference equation by the differential,
2. the error caused by using the approximation $\varrho(x) \approx \frac{x}{2}$ for small x,
3. the error in $\varrho'(\alpha_{c2}F + \alpha_{c3})$ relative to the desired $\varrho'(\alpha_{c2}A_i + \alpha_{c3})$.

Synthesizing what we have just seen...

$$\begin{aligned}
x_i(t+1) &= \varrho(\alpha_{c1}\varrho(\alpha_{c2}F + \alpha_{c3} + \alpha_{c4}x_i(t)) - \alpha_{c1}\varrho(\alpha_{c2}F + \alpha_{c3}) + \alpha_{c5}x_i(t)) \\
&(1) \approx \varrho(\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}F + \alpha_{c3})x_i(t) + \alpha_{c5}x_i(t)) \\
&(2) \approx (\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}F + \alpha_{c3}) + \alpha_{c5})\frac{x_i(t)}{2} \\
&(3) \approx (\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}A_j + \alpha_{c3}) + \alpha_{c5})\frac{x_i(t)}{2} \\
&= \begin{cases} Bx_i(t) & \text{if } j = 1, \\ \frac{x_i(t)}{B} & \text{if } j = 2. \end{cases}
\end{aligned}$$

$$\boxed{x_i(t+1) \quad = \begin{cases} Bx_i(t) & \text{if } j = 1, \\ \frac{x_i(t)}{B} & \text{if } j = 2. \end{cases}}$$

5.1 Error in one step

Let us begin by ignoring the errors caused by the errors of the flip-flops, (3).

During the first half of the restless counter's day, the value of x_i is repeatedly divided by B. We will call the update of the neuron's value during this period *down_update*. Our first goal is to prove that the error committed in this half of the cycle, considering n steps, can be bounded by $\frac{ky^2}{B^n}$, for n sufficiently large and for some constant k. This will in turn allow us to conclude that the error in the whole cycle will be bounded by a constant, for n arbitrarily large. Moreover, we will see that this constant can be made arbitrarily small by choosing the parameters of our network appropriately.

Assuming that the values received by the flip-flops were the exact ones, A_1 and A_2, we would have the error in each step given by:

$down\_error(y) = |down\_update(y) - \frac{y}{B}|$ [1]

[1] y is simply any value being stored.

$$= \dagger^2 |\varrho(\alpha_{c1}\varrho(\alpha_{c2}A_2 + \alpha_{c3} + \alpha_{c4}y) - \alpha_{c1}\varrho(\alpha_{c2}A_2 + \alpha_{c3}) + \alpha_{c5}y) - \tfrac{y}{B}|$$
$$= |\frac{2}{1+e^{\alpha_{c1}(\frac{2}{1+e^{\alpha_{c3}+\alpha_{c2}A_2}} - \frac{2}{1+e^{\alpha_{c3}+\alpha_{c2}A_2+\alpha_{c4}y}}) - \alpha_{c5}y}} - 1 - \frac{y}{B}|$$
$$= |\frac{2}{1+e^{-\frac{2\alpha_{c1}e^{\alpha_{c3}+\alpha_{c2}A_2}(e^{\alpha_{c4}y}-1)}{(1+e^{\alpha_{c3}+\alpha_{c2}A_2})(1+e^{\alpha_{c3}+\alpha_{c2}A_2+\alpha_{c4}y})} - \alpha_{c5}y}} - 1 - \frac{y}{B}|$$
$$= |1 - \frac{2}{1+e^{\alpha_{c5}y+\alpha_{c1}Sech(\frac{1}{2}(\alpha_{c3}+\alpha_{c2}A_2))Sech(\frac{1}{2}(\alpha_{c3}+\alpha_{c2}A_2+\alpha_{c4}y))Sinh(\frac{\alpha_{c4}y}{2})}} - \frac{y}{B}|$$
$$= \dagger|1 - \frac{2}{1+e^{\alpha_{c5}y+\alpha_{c1}Sech(\frac{1}{2}(\alpha_{c3}+\alpha_{c2}A_2))Sech(\frac{1}{2}(\alpha_{c3}+\alpha_{c2}A_2+\alpha_{c4}y))Sinh(\frac{\alpha_{c4}y}{2})}} -$$
$$-\frac{y}{2}(\alpha_{c5} + \frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(\alpha_{c3}+\alpha_{c2}A_2)})|$$
$$= \dagger|1 - \frac{2}{1+e^{\alpha_{c5}y+\alpha_{c1}\frac{Sinh(X)}{Cosh(Y)Cosh(X+Y)}}} - \frac{y}{2}(\alpha_{c5} + \frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)})|,$$
where $X = \frac{\alpha_{c4}y}{2}$ and $Y = \frac{1}{2}(\alpha_{c3} + \alpha_{c2}A_2)$.

First, let us show that $\alpha_{c5}y + \alpha_{c1}\frac{Sinh(X)}{Cosh(Y)Cosh(X+Y)} = y(\alpha_{c5} + \frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)}) - \frac{1}{4}\alpha_{c1}\alpha_{c4}^2 Sech(Y)^2 Tanh(Y)y^2 + O(y^3)$. This can be done by expanding the first expression in Taylor series around 0 and seeing that we obtain exactly the second term of the previous equality.

We would also like to point out the behavior of the function $(\lambda y.\frac{e^y-1}{e^y+1})$. In a neighborhood of 0, this function approximates $(\lambda y.\frac{y}{2})$. Using the Taylor series expansion around 0 for this function we have $\frac{e^y-1}{e^y+1} = \frac{y}{2} - \frac{y^3}{24} + O(y^5)$.

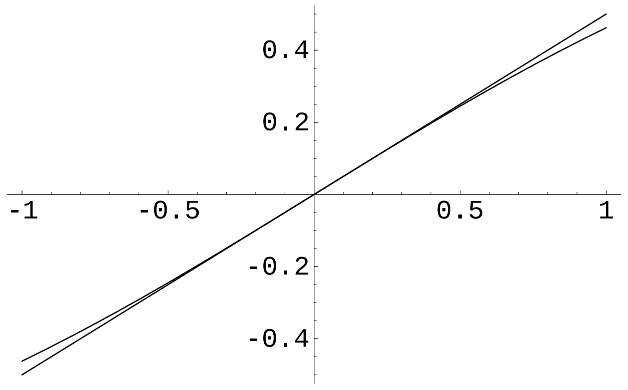

Figure 5.1: $\lambda y.\frac{e^y-1}{e^y+1} \approx \lambda y.\frac{y}{2}$.

Combining this information together we have

$$down\_error(y) = |1 - \frac{2}{1+e^{\alpha_{c5}y+\alpha_{c1}\frac{Sinh(X)}{Cosh(Y)Cosh(X+Y)}}} - \frac{y}{2}(\alpha_{c5} + \frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)})|$$
$$= |\frac{e^{\alpha_{c5}y+\alpha_{c1}\frac{Sinh(X)}{Cosh(Y)Cosh(X+Y)}}-1}{e^{\alpha_{c5}y+\alpha_{c1}\frac{Sinh(X)}{Cosh(Y)Cosh(X+Y)}}+1} - \frac{y}{2}(\alpha_{c5} + \frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)})|$$

[2]Whenever an equality is marked with †it means that the sequence of equalities has been confirmed using Mathematica.

$$= |\frac{\alpha_{c5}y+\alpha_{c1}\frac{Sinh(X)}{Cosh(Y)Cosh(X+Y)}}{2} + O((\alpha_{c5}y + \alpha_{c1}\frac{Sinh(X)}{Cosh(Y)Cosh(X+Y)})^3)$$
$$-\frac{y}{2}(\alpha_{c5} + \frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)})|$$
$$= |\frac{y(\alpha_{c5}+\frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)})-\frac{1}{4}\alpha_{c1}\alpha_{c4}^2 Sech(Y)^2 Tanh(Y)y^2}{2} + O(y^3) - \frac{y}{2}(\alpha_{c5}+\frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)})|$$

$$= | - \frac{1}{8}\alpha_{c1}\alpha_{c4}^2 Sech(Y)^2 Tanh(Y)y^2 + O(y^3)|$$
$$= \frac{1}{8}|\alpha_{c1}|\alpha_{c4}^2 Sech(Y)^2 |Tanh(Y)|y^2 + O(y^3)$$ [3]

Given the fact that $\forall Y \in \mathbb{R}\ Sech(Y)^2 Tanh(Y) < \frac{2}{5}$, we can write

$$down\_error(y) < \frac{|\alpha_{c1}|\alpha_{c4}^2}{20}y^2 + O(y^3).$$

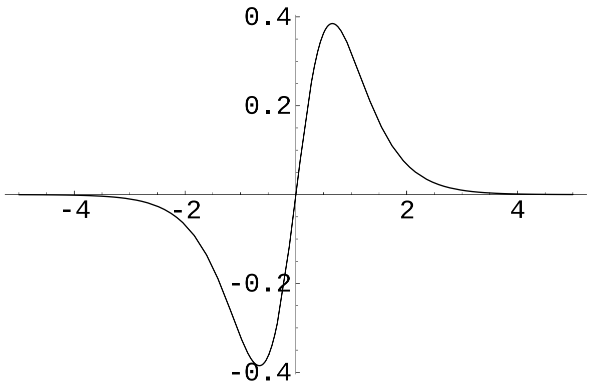

Figure 5.2: Function $\lambda x.Sech(x)^2 Tanh(x)$.

Therefore, for small values of y (after the first few steps), we can control the relative error committed by simply controlling the value of the constants α_{c1} and α_{c4}. If we recall section 4.2, where we saw that once the other constants were fixed, we could freely choose α_{c1} and α_{c4} so that their multiplication was constant, then we conclude that we can make the relative errors as small as desired, by simply diminishing the absolute value of α_{c4}.

5.2 Error in one cycle

As for the error during this entire half of the cycle (except the first few steps), let us define

$$down\_error_n(y) = |down\_update^n(y) - \frac{y}{B^n}|,$$

where $down\_update^n(y)$ denotes the result of n repeated applications of *down_update* to y.

[3]Please recall that $\alpha_1 < 0$.

Once we can guarantee that the error in one step can be bounded by a constant ϵ times the value of y^2, then *down_update*(y) will certainly be between $\frac{y}{B} - \epsilon y^2$ and $\frac{y}{B} + \epsilon y^2$. Let us consider the worst possible case, that *down_update*(y) always equals $\frac{y}{B} + \epsilon y^2$. We can easily obtain the formula

$$down\_update^n(y) = \frac{y}{B^n} + \sum_{i=n-1}^{2(n-1)} \frac{\epsilon y^2}{B^i} + O(y^3)$$

and immediately

$$\begin{aligned} down\_error_n(y) &< |\sum_{i=n-1}^{2(n-1)} \frac{\epsilon y^2}{B^i} + O(y^3)| \\ &= |\frac{\epsilon y^2(B^n-1)}{B^{2n-2}(B-1)} + O(y^3)| \\ &= \frac{|\alpha_{c1}|\alpha_{c4}^2 y^2(B^n-1)}{20B^{2n-2}(B-1)} + O(y^3). \end{aligned}$$

As $n \to \infty$, $\frac{B^n-1}{B^{2n-2}} \to B^{2-n}$ and $\frac{|\alpha_{c1}|\alpha_{c4}^2 y^2(B^n-1)}{20B^{2n-2}(B-1)} \to \frac{|\alpha_{c1}|\alpha_{c4}^2 y^2 B^2}{20(B-1)B^n}$.
As we intended, we have just shown that the error can be bounded by $\frac{ky^2}{B^n}$ for a constant $k = \frac{|\alpha_{c1}|\alpha_{c4}^2 B^2}{20(B-1)}$.
For the other half of the cycle, we begin by adapting the error estimate in a much similar way,

$$\begin{aligned} up\_error(y) &= |up\_update(y) - yB| \\ &= |\varrho(\alpha_{c1}\varrho(\alpha_{c2}A_1 + \alpha_{c3} + \alpha_{c4}y) - \alpha_{c1}\varrho(\alpha_{c2}A_1 + \alpha_{c3}) + \alpha_{c5}y) - yB| \\ &= |\frac{2}{1+e^{\alpha_{c1}(\frac{2}{1+e^{\alpha_{c3}+\alpha_{c2}A_1}} - \frac{2}{1+e^{\alpha_{c3}+\alpha_{c2}A_1+\alpha_{c4}y}})-\alpha_{c5}y}} - 1 - yB| \\ &= |\frac{2}{1+e^{-\frac{2\alpha_{c1}e^{\alpha_{c3}+\alpha_{c2}A_1}(e^{\alpha_{c4}y}-1)}{(1+e^{\alpha_{c3}+\alpha_{c2}A_2})(1+e^{\alpha_{c3}+\alpha_{c2}A_1+\alpha_{c4}y})}-\alpha_{c5}y}} - 1 - yB| \\ &= |1 - \frac{2}{1+e^{\alpha_{c5}y+\alpha_{c1}Sech(\frac{1}{2}(\alpha_{c3}+\alpha_{c2}A_1))Sech(\frac{1}{2}(\alpha_{c3}+\alpha_{c2}A_1+\alpha_{c4}y))Sinh(\frac{\alpha_{c4}y}{2})}} - yB| \\ &= |1 - \frac{2}{1+e^{\alpha_{c5}y+\alpha_{c1}Sech(\frac{1}{2}(\alpha_{c3}+\alpha_{c2}A_1))Sech(\frac{1}{2}(\alpha_{c3}+\alpha_{c2}A_1+\alpha_{c4}y))Sinh(\frac{\alpha_{c4}y}{2})}} - \\ &\quad -\frac{y}{2}(\alpha_{c5} + \frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)})| \\ &= |1 - \frac{2}{1+e^{\alpha_{c5}y+\alpha_{c1}\frac{Sinh(X)}{Cosh(Y)Cosh(X+Y)}}} - \frac{y}{2}(\alpha_{c5} + \frac{\alpha_{c1}\alpha_{c4}}{1+Cosh(2Y)})|, \end{aligned}$$

where $X = \frac{\alpha_{c4}y}{2}$ and $Y = \frac{1}{2}(\alpha_{c3} + \alpha_{c2}A_1)$.

For the same reasons, for this half of the cycle we also have the bound

$$up\_error(y) = \frac{1}{8}|\alpha_{c1}|\alpha_{c4}^2 Sech(Y)^2|Tanh(Y)|y^2 + O(y^3).$$

Obviously, we will now define

$$up\_error_n(y) = |up\_update^n(y) - yB^n|$$

Given the bounding on the error of *up_update*,

$$By - \epsilon y^2 < up\_update(y) < By + \epsilon y^2,$$

we can assure that *up_error*$(y) < \epsilon y^2$. Assuming the error to always have this value, we would obtain

$$up\_update^n(y) = yB^n + \sum_{i=n-1}^{2(n-1)} \epsilon y^2 B^i + O(y^3)$$

and immediately

$$\begin{aligned} up\_error_n(y) &= |\sum_{i=n-1}^{2(n-1)} \epsilon y^2 B^i + O(y^3)| \\ &= |\tfrac{\epsilon y^2(B^n-1)B^{n-1}}{B-1} + O(y^3)| \\ &= \tfrac{|\alpha_{c1}|\alpha_{c4}^2 y^2(B^n-1)B^{n-1}}{20(B-1)} + O(y^3). \end{aligned}$$

Combining the two previous results, we have the total error of the cycle bounded by:

$$\begin{aligned} total\_error_n(y) &= |up\_update^n(down\_update^n(y)) - y| \\ &= |up\_update^n(\tfrac{y}{B^n} + \sum_{i=n-1}^{2(n-1)} \tfrac{\epsilon y^2}{B^i} + O(y^3)) - y| \\ &< |up\_update^n(\tfrac{y}{B^n} + \tfrac{ky^2}{B^n}) - y| \\ &= |up\_update^n(\tfrac{y+ky^2}{B^n}) - y| \\ &= |\tfrac{y+ky^2}{B^n}B^n + \sum_{i=n-1}^{2(n-1)} \epsilon(\tfrac{y+ky^2}{B^n})^2 B^i + O((\tfrac{y+ky^2}{B^n})^3) - y| \\ &< |ky^2 + \sum_{i=n-1}^{2(n-1)} \epsilon(\tfrac{y+ky^2}{B^n})^2 B^i| \\ &= |ky^2 + \tfrac{\epsilon(\frac{y+ky^2}{B^n})^2(B^n-1)B^{n-1}}{B-1}| \\ &= |\tfrac{\epsilon B^2}{B-1}y^2 + \tfrac{\epsilon(\frac{y+ky^2}{B^n})^2(B^n-1)B^{n-1}}{B-1}| \\ &= |\tfrac{\epsilon B^2}{B-1}y^2 + \tfrac{\epsilon y^2(B^n-1)B^{n-1}}{B^{2n}(B-1)} + O(y^4)| \\ &= |(\tfrac{B^2}{B-1} + \tfrac{B^n-1}{B^{n+1}(B-1)})\epsilon y^2 + O(y^4)| \\ &< \tfrac{B^3+1}{B^2-B}\epsilon y^2 \text{ (for sufficiently small } y) \\ &= \tfrac{|\alpha_{c1}|\alpha_{c4}^2(B^3+1)}{20(B^2-B)}y^2. \end{aligned}$$

From the bound above it is now clear that in order to obtain an error smaller than ζ, with an initial value y, all we have to do is to choose α_{c4} small enough so that $\alpha_{c4}^2 < \frac{20\zeta(B^2-B)}{|\alpha_{c1}|(B^3+1)}y$.

5.3 The first steps

As for the beginning and the end of the cycle, the situation is as follows: in order to control the error committed in the first step, all we have to do is to decrease α_{c4}, if B is sufficiently large. For every B, there is a value c such that when we make $\alpha_{c4} \to 0$ the relative error on the first step converges to c. As $B \to \infty$, $c \to 0$. This procedure is valid for the first n steps of the cycle, until the bound previously achieved can be applied (empirically, usually 2 or 3 steps). The same procedure, making α_{c4} vanish with B sufficiently large can be used to control the errors in the last steps, except for the last one. The error committed in this last step is always between 0.2 and 0.25, independently of the constants considered. This is caused by the fact that in a neighborhood of 1 the update function converges to a fixed point instead of keep multiplying by B. We will see that the two additional time steps at which the restless counter remains at 0 will allow to correct this error.

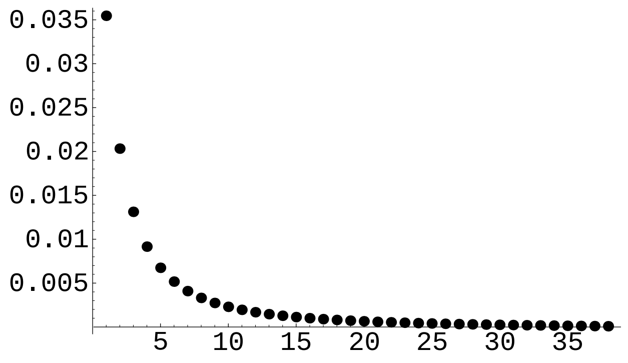

Figure 5.3: Bound for the relative error committed on the first step of the cycle, for the integer values of B between 3 and 40, choosing $\alpha_{c4} = 10^{-7}$.

Let us recall that the error in the first step can be bounded by

$$\begin{aligned}
down\_error(1) &= \left|\frac{2}{1+e^{-\frac{2\alpha_{c1}e^{\alpha_{c3}+\alpha_{c2}A_2}(e^{\alpha_{c4}}-1)}{(1+e^{\alpha_{c3}+\alpha_{c2}A_2})(1+e^{\alpha_{c3}+\alpha_{c2}A_2+\alpha_{c4}})}-\alpha_{c5}}} - 1 - \frac{1}{B}\right| \\
&= \left|\frac{2}{1+e^{-\frac{2\alpha_{c1}\alpha_{c4}e^{\alpha_{c3}+\alpha_{c2}A_2}(\frac{e^{\alpha_{c4}}-1}{\alpha_{c4}})}{(1+e^{\alpha_{c3}+\alpha_{c2}A_2})(1+e^{\alpha_{c3}+\alpha_{c2}A_2+\alpha_{c4}})}-\alpha_{c5}}} - 1 - \frac{1}{B}\right| \\
&\xrightarrow[\alpha_{c4}\to 0]{} \left|\frac{2}{1+e^{-\frac{2\alpha_{c1}\alpha_{c4}e^{\alpha_{c3}+\alpha_{c2}A_2}}{(1+e^{\alpha_{c3}+\alpha_{c2}A_2})^2}-\alpha_{c5}}} - 1 - \frac{1}{B}\right| \\
&= \left|\frac{2}{1+e^{-\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c3}+\alpha_{c2}A_2)-\alpha_{c5}}} - 1 - \frac{1}{B}\right| \\
&= \left|\frac{2}{1+e^{-\frac{2}{B}}} - 1 - \frac{1}{B}\right| \\
&\xrightarrow[B\to +\infty]{} \left|\frac{2}{1+e^{0}} - 1 - 0\right| = 0
\end{aligned}$$

So, concluding, by choosing B sufficiently large and α_{c4} sufficiently small, we can control the errors during the whole cycle, except for the last step. In figure 5.4 we can see the bound of the error given by the above expression, which occurs when α_{c4} tends to 0. As expected, the bounds are not very different from figure 5.3, only tighter.

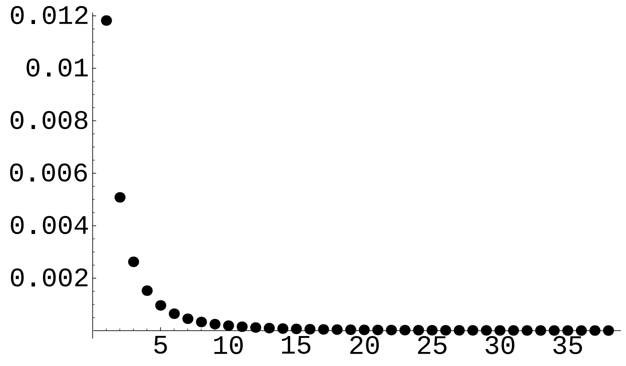

Figure 5.4: $\frac{2}{1+e^{-\frac{2}{B}}} - 1 - \frac{1}{B}$ for the first integer values of B. This is, for each B, the best bound on the errors we may obtain.

5.4 The errors sent by the flip-flops

Let us see now what happens with the errors in the signals sent by the flip-flops. Let us suppose now, that the functioning of the restless counters is exact except for the errors caused by the flip-flops, in order to prove that these errors are controllable. During the first half of the cycle, the value ideally sent by the flip-flop is A_2, and this value is approximately 1. Let γ be the difference between these two values.

$$|A_2 - F| = \gamma \Leftrightarrow |(\alpha_{c2}A_2 + \alpha_{c3}) - (\alpha_{c2}F + \alpha_{c3})| = |\alpha_{c2}|\gamma$$
$$\Rightarrow |\varrho'(\alpha_{c2}A_2 + \alpha_{c3}) - \varrho'(\alpha_{c2}F + \alpha_{c3})| \approx |\varrho''(\alpha_{c2}A_2 + \alpha_{c3})\alpha_{c2}|\gamma.$$

Given the fact that $\forall x \in \mathbb{R}\ |\varrho''(x) < \frac{1}{5}|$ (as we can see in figure 5.5), then:

$$|\varrho'(\alpha_{c2}A_2 + \alpha_{c3}) - \varrho'(\alpha_{c2}F + \alpha_{c3})| < \frac{|\alpha_{c2}|\gamma}{5}$$

and thus

$$|\frac{2}{B} - (\alpha_{c1}\alpha_{c4}\varrho'(\alpha_{c2}A_2 + \alpha_{c3}) + \alpha_{c5})| < \frac{|\alpha_{c1}\alpha_{c2}\alpha_{c4}|\gamma}{5}$$

and the error in each step is bounded by $error(y) < \frac{|\alpha_{c1}\alpha_{c2}\alpha_{c4}|\gamma y}{10}$.

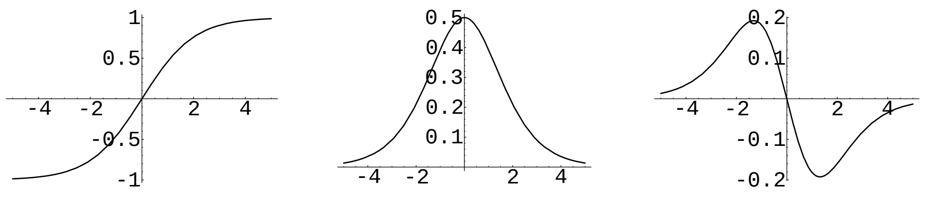

Figure 5.5: Functions ϱ, ϱ', and ϱ''.

As we saw in section 4.1, the convergence for the fixed points of the flip-flops is linear, and

$\frac{\gamma_{n+1}}{\gamma_n} \underrightarrow{n \to \infty} \frac{\partial \varrho(\alpha_{f2}x+\alpha_{f3})}{\partial x}(A_2) = \frac{\alpha_{f2}}{1+Cosh(\alpha_{f2}A_2+\alpha_{f3})}$.

The error on the first step is bounded by

$$error(y) < \frac{|\alpha_{c1}\alpha_{c2}\alpha_{c4}|\gamma y}{10}$$

and the error on step n is smaller than

$$\frac{|\alpha_{c1}\alpha_{c2}\alpha_{c4}|(\frac{\alpha_{f2}}{1+Cosh(\alpha_{f2}A_2+\alpha_{f3})})^{(n-1)}\gamma y}{10}.$$

Replacing A_2 by A_1 in the previous expression we obtain a bound for the errors in the second half of the cycle. Now we will show that these conditions are enough to assure that these errors will not compromise the functioning of our network. Recalling our discussion from section 4.1 (see page 29) we have chosen values of α_{f2} and α_{f3} such that

$$\frac{\alpha_{f2}}{1+Cosh(\alpha_{f2}A_2+\alpha_{f3})}, \frac{\alpha_{f2}}{1+Cosh(\alpha_{f2}A_1+\alpha_{f3})} < 1.$$

For $\alpha_{f2} = 30$ and $\alpha_{f3} = 1/2$, we have $\frac{\alpha_{f2}}{1+Cosh(\alpha_{f2}A_2+\alpha_{f3})} \approx 3.4 \times 10^{-12}$. Clearly, from the fact that the errors caused by the flip-flops diminish with $(\frac{\alpha_{f2}}{1+Cosh(\alpha_{f2}A_2+\alpha_{f3})})^{n-1}$, there is no problem at all with the first half of the cycle, as long as we control the error on the first step, which is always possible, because γ can be made arbitrarily small.

The error committed in n steps on the descending phase can be determined by remarking that

$$y\prod_{i=0}^{n-1}(\frac{1}{B} - \epsilon\gamma k^i) < update_n(y) < y\prod_{i=0}^{n-1}(\frac{1}{B} + \epsilon\gamma k^i)$$

where $k = \frac{\alpha_{f2}}{1+Cosh(\alpha_{f2}A_2+\alpha_{f3})}$, $\epsilon = \frac{|\alpha_{c1}\alpha_{c2}\alpha_{c4}|}{10}$ and γ is the difference between A_2 and the value received on the first time step. The relative error

$$\begin{aligned}\frac{|\frac{y}{B^n} - update_n(y)|}{\frac{y}{B^n}} &= |1 - \frac{B^n update_n(y)}{y}| \\ &< |1 - \prod_{i=0}^{n-1}(1+\epsilon\gamma B k^i)| \\ &= |1 - (1 + O(\prod_{i=0}^{n-1}(\epsilon\gamma B k^i)))| \\ &= O((\epsilon\gamma B)^n k^{\frac{n^2-1}{2}}).\end{aligned}$$

The real problem at hand is what happens at the turning point, when the rc neuron receives the input to reverse the sense. The value stored on the rc neuron is approximately B^{-n} and the error in this first step will be approximately multiplied by B^n. Should the error in this first step be constant (even if it were a constant as small as desired, as if it were γ), and by allowing n sufficiently large we could have γB^n too large to keep the errors within controlled bounds.

This is the main reason for the reluctancy by some scientists in accepting the results of [KS96]. It is not clear from the discussion in [KS96] and [Sie99] how this problem can be overcome. It is only affirmed that the errors committed by the flip-flops can be made arbitrarily small. If we could only assure this property, for any given network that we had designed to have the errors of the flip-flops bounded by γ, then during the network's computation, as the periods of the rc neurons' cycles increase the values of these neurons achieve values so small that they become sensitive to errors smaller than γ; and for every natural n, we would be able to design a neural network capable of simulating a Turing machine correctly as long as the periods of the neurons didn't exceed n, but we would not be capable of building a really universal network.

Fortunately, the error in the update of the rc neuron is proportional to the value stored, y, and for that reason we will be capable of limiting the error, for n arbitrarily large. The adaptation of the previous error analysis is straightforward, but we state the obvious, for completeness. The error on step n after the turning point is bounded by

$$\frac{|\alpha_{c1}\alpha_{c2}\alpha_{c4}|(\frac{\alpha_{f2}}{1+Cosh(\alpha_{f2}A_1+\alpha_{f3})})^{(n-1)}\gamma y}{10};$$

the bounds for the final result are

$$y\prod_{i=0}^{n-1}(B-\epsilon\gamma k^i) < update_n(y) < y\prod_{i=0}^{n-1}(B+\epsilon\gamma k^i)$$

and the relative error can be controlled because

$$\begin{aligned}\frac{|yB^n - update_n(y)|}{yB^n} &= |1-\frac{update_n(y)}{B^n y}|\\ &< |1-\prod_{i=0}^{n-1}(1+\frac{\epsilon\gamma k^i}{B})|\\ &= |1-(1+O(\prod_{i=0}^{n-1}(\frac{\epsilon\gamma k^i}{B})))|\\ &= O((\frac{\epsilon\gamma}{B})^n k^{\frac{n^2-1}{2}}).\end{aligned}$$

5.5 Combination of errors

Finally, we must show that the conjunction of the two distinct types of errors considered is still controllable, and that no problems arise from the repetition of the process. The total error considering the approximations and the flip-flops is bounded by the sum of both errors added to their product, therefore can also be as small as desired. As for the repetition of the process, the restless counters are implemented in such a way that they remain at 0 for two time steps after each day. This means that the rc neuron continues to update with input A_1 for two time units longer. We have already seen that the system works fine until the final step. Now let's see what goes on in these 3 time units.

The two additional time steps are in fact used as a way of correcting the errors, and that is what allows us to maintain the errors controlled *ad aeternum*. For values close to 1, the update of the rc neuron, when receiving A_1 will no longer cause the value of the neuron to be multiplied by B, but instead will draw that value to a fixed point of the equation

$\varrho[\alpha_{c1}\varrho(\alpha_{c2}A_1+\alpha_{c3}+\alpha_{c4}x)-\alpha_{c1}\varrho(\alpha_{c2}A_1+\alpha_{c3})+\alpha_{c5}x]=x$. An unavoidable (but harmless) drawback caused by this feature is the error in the step before.

The error in the last step can be properly bounded, although it cannot be made arbitrarily small. The correction of errors made in the extra steps allows us to correct even larger errors than those committed. In figure 5.6 we can see the results of performing a cycle of the rc neuron with errors from the flip-flops supposed to be always in the same direction. The upper line of losangles is a line with the constant value that we used to start the iteration. The line with the stars is the result of the cycle, assuming that no errors were committed by the flip-flops, for different period lengths. The starts and the triangles are the result of assuming always an error of 1% on the flip-flops, in the squares in such a way that causes an increase in the overall computation, and in the case of the triangles, we assume the errors in the opposite way. We can also see that the interval 0.9 to 1.01 maps into a subinterval of itself (with the two extra steps included). Therefore the repetition of these steps is possible, without losing control of the errors committed over time.

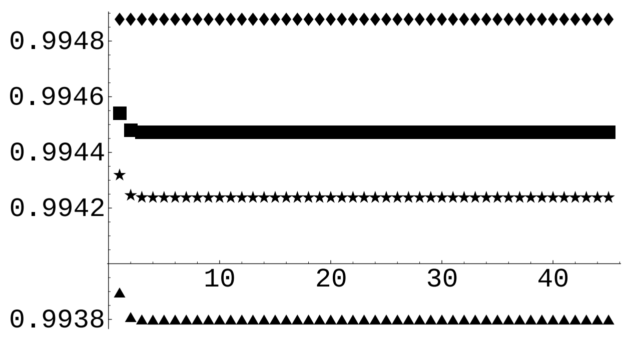

Figure 5.6: Result of the errors sent by the flip-flops in the value stored by the rc neurons.

5.6 The main theorem

The proof of the correctness of the simulation is inductive on the serial number of the day (that is, the time of M_0 alarming). As the network $\mathcal{N}$ consists of three parts: finite automaton (FA), flip flops (FF) and restless counters (RC), for each part we assume a "well behaved input" in day d and prove a "well behaved output" for the same day. Because on the first day inputs to all parts are well-behaved, the correctness follows inductively.

Theorem 5.6.1 The following statements are equivalent:

(1) On each day, the finite control sends O(1) signals (intentionally non-zero) to the flip flops. Each signal has an error bounded by $\mu < 0.01$ and the sum of errors in the signals of the finite control during the day is bounded by a constant $\beta < 0.1$.

(2) On each day, the signals sent by the flip flops have an arbitrarily small error γ, (as a function of μ and the constants of the flip flops) and the sum of the total error of all signals during the day is bounded by an arbitrarily small constant δ.

(3) On each day, a restless counter with a value y acquires a total multiplicative error $\zeta < 0.01$. That is, the ratio of the actual value with the ideal value will always be between 0.99 and 1.01.

Remarks:

The bounded number of signals sent by the finite control to the flip-flops can be determined easily. In the worst hypothesis possible, a flip-flop neuron receives 10 input signals from the finite control. From chapter 3, we know that the period of the counters is only lengthened, at most, twice a day, and their alarm time is also delayed, at most, twice a day. As we have also seen in chapter 3, the lengthening of the period of the adder clocks is trickier then the auxiliary clocks. In particular, they need to receive 4 inputs from the finite control. This situation in which the FF receives 10 signals occurs when the network simulates an increase in an adder. At the end of the cycle, when A_0 and A_00 alarm together, if an adder is to be increased, then its flip-flop neuron receives 2 inputs to cause its delay, and 4 for each lengthen operation.

The feature that the total error of all signals sent by the flip-flop over the day is arbitrarily small is essential to the correct functioning of the network. Otherwise, an accumulation of errors from the flip flops would follow into the rc neurons and cause the lost of the stored values of the counters.

The fact that the multiplicative error of the neuron is always so small will allow us to retrieve the value stored in its activation without danger of confusion. It is also what allows us, together with the error-correcting effect of these devices, to maintain a bound of error that allows us to keep the multiplicative errors controlled independently of how many days the computation lasts.

Another imortant thing is the fact that $\mu < 0.01$ independently of the day. This means that the network doesn't start loosing accuracy with time.

Proof:

$1 \Rightarrow 2$:

We assume that the finite control sends Start-Inc$_i$ and Start-Dec$_i$ to ff$_i$, that these two neurons are never active simultaneously, and that the error on the values of the neurons is always smaller than μ. We will show in this first part of the proof that if the errors of the inputs to the flip-flops are bounded in this way, then the errors committed by the flip-flops will be bounded by 0.01 as well.

The update equation of each flip-flop is

$$\text{ff}_i(t+1) = \varrho(\alpha_{f1}(\text{Start-Inc}_i(t) - \text{Start-Dec}_i(t)) + \alpha_{f2}\text{ff}_i(t) + \alpha_{f3}).$$

We will analyze two different situations separately: in the first situation the flip-flop changes its value in order to change the direction of the restless counter, in the second situation the flip-flop is simply converging to one of its attractors.

<u>1st Case</u>

For the first case we will assume that the flip-flop was converging to A_2 and that Start-Inc$_i$ became active. The opposite case is completely similar. Then

$$\begin{cases} 1-\mu < \text{Start-Inc}_i(t) < 1 \\ -\mu < \text{Start-Dec}_i(t) < \mu \end{cases} \Rightarrow 1-2\mu < \text{Start-Inc}_i(t) - \text{Start-Dec}_i(t) < 1+\mu$$

$\Rightarrow (1+\mu)\alpha_{f1} < \alpha_{f1}(\text{Start-Inc}_i(t) - \text{Start-Dec}_i(t)) < (1-2\mu)\alpha_{f1}$

$\Rightarrow (1+\mu)\alpha_{f1} + \alpha_{f2}(A_2-\mu) < \alpha_{f1}(\text{Start-Inc}_i(t) - \text{Start-Dec}_i(t)) + \alpha_{f2}\text{ff}_i(t) < (1-2\mu)\alpha_{f1} + \alpha_{f2}(A_2+\mu)$

$\Rightarrow (1+\mu)\alpha_{f1} + \alpha_{f2}(A_2-\mu) + \alpha_{f3} < \alpha_{f1}(\text{Start-Inc}_i(t) - \text{Start-Dec}_i(t)) + \alpha_{f2}\text{ff}_i(t) + \alpha_{f3} < (1-2\mu)\alpha_{f1} + \alpha_{f2}(A_2+\mu) + \alpha_{f3}$

$\Rightarrow$ [4] $\varrho((1+\mu)\alpha_{f1} + \alpha_{f2}(A_2-\mu) + \alpha_{f3}) < \varrho(\alpha_{f1}(\text{Start-Inc}_i(t) - \text{Start-Dec}_i(t)) + \alpha_{f2}\text{ff}_i(t) + \alpha_{f3}) < \varrho((1-2\mu)\alpha_{f1} + \alpha_{f2}(A_2+\mu) + \alpha_{f3})$

$\Rightarrow \varrho((1+\mu)\alpha_{f1} + \alpha_{f2}(A_2-\mu) + \alpha_{f3}) < \text{ff}_i(t+1) < \varrho((1-2\mu)\alpha_{f1} + \alpha_{f2}(A_2+\mu) + \alpha_{f3})$.

If we choose constants α_{f1}, α_{f2}, and α_{f3} such that:

$$\begin{cases} \varrho(\alpha_{f1} + \alpha_{f2}A_2 + \alpha_{f3}) = A_1 \\ \varrho(-\alpha_{f1} + \alpha_{f2}A_1 + \alpha_{f3}) = A_2 \end{cases}$$

the difference between the value obtained and the ideal value A_1 can be bounded by $|\varrho(\alpha_{f1} + \alpha_{f2}A_2 + \alpha_{f3}) - \varrho(\alpha_{f1} + \alpha_{f2}A_2 + \alpha_{f3} + \mu(\alpha_{f1} - \alpha_{f2}))|$ and $|\varrho(\alpha_{f1} + \alpha_{f2}A_2 + \alpha_{f3}) - \varrho(\alpha_{f1} + \alpha_{f2}A_2 + \alpha_{f3} + \mu(\alpha_{f2} - 2\alpha_{f1}))|$ and therefore can be approximately bounded by $|\varrho'(\alpha_{f1} + \alpha_{f2}A_2 + \alpha_{f3})\mu(\alpha_{f2} - 2\alpha_{f1})|$.

If we choose α_{f3} to be 0, and start iterating the values of α_{f2}, then for any fixed value of α_{f2} greater than 2 there is only one value α_{f1} that verifies each of the above equations, and it is the same for both. After some simple manipulations we obtain

$$\alpha_{f1} = \varrho^{-1}(A_1) - \alpha_{f2}A_2 = ln(\tfrac{1+A_1}{1-A_1}) - \alpha_{f2}A_2$$

As $\alpha_{f2} \to \infty$, $\alpha_{f1} \to -2\alpha_{f2}$, and asymptotically we have

[4] Because ϱ is monotonic, i.e., $x < y \Rightarrow \varrho(x) < \varrho(y)$

$$\begin{aligned}\varrho'(\alpha_{f1}+\alpha_{f2}A_2+\alpha_{f3})\mu(\alpha_{f2}-2\alpha_{f1}) &= \tfrac{2e^{-\alpha_{f1}-\alpha_{f2}A_2}}{(1+e^{-\alpha_{f1}-\alpha_{f2}A_2})^2}\mu(\alpha_{f2}-2\alpha_{f1}) \\ &\approx \tfrac{2e^{\alpha_{f2}(2-A_2)}}{(1+e^{\alpha_{f2}(2-A_2)})^2}\mu(5\alpha_{f2}) \\ &\approx \tfrac{10\mu\alpha_{f2}}{1+e^{\alpha_{f2}(2-A_2)}}.\end{aligned}$$

And if we see what happens with consecutive integer values of α_{f2} we notice that the ratio of the errors tends to $\frac{1}{e}$.

$$\begin{aligned}\frac{\frac{10\mu(\alpha_{f2}+1)}{1+e^{(\alpha_{f2}+1)(2-A_2)}}}{\frac{10\mu\alpha_{f2}}{1+e^{\alpha_{f2}(2-A_2)}}} &= \frac{\frac{(\alpha_{f2}+1)}{1+e^{(\alpha_{f2}+1)(2-A_2)}}}{\frac{\alpha_{f2}}{1+e^{\alpha_{f2}(2-A_2)}}} \\ &\xrightarrow[\alpha_{f2}\to\infty]{} \frac{\frac{1}{e^{(\alpha_{f2}+1)(2-A_2)}}}{\frac{1}{e^{\alpha_{f2}(2-A_2)}}} \\ &= \frac{e^{\alpha_{f2}(2-A_2)}}{e^{(\alpha_{f2}+1)(2-A_2)}} \\ &= e^{A2-2} \\ &\xrightarrow[\alpha_{f2}\to\infty]{} e^{-1}\end{aligned}$$

We conclude then that the error commited by the flip-flops can be controlled by simply increasing the value of α_{f2}. Moreover, empirically we can see that by choosing $\alpha_{f2} = n$, and admitting $\mu = 0.01$ we assure that the error is smaller than $(\frac{2}{5})^n$. In order to achieve an error of γ all we have to do is choose $\alpha_{f2} > log_{\frac{2}{5}}(\gamma)$.

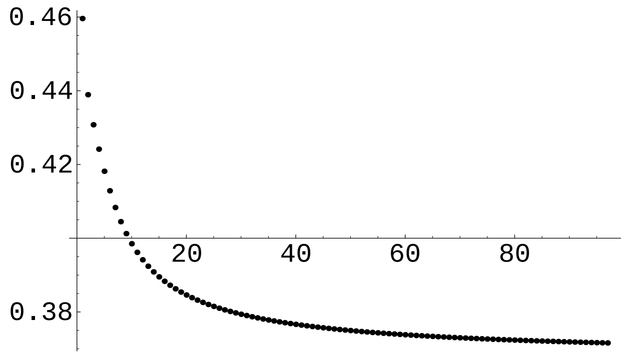

Figure 5.7: Ratios of the errors committed between consecutive integer values of α_{f2}, admitting $\mu = 0.01$.

It is also possible to show, using similar reasons, that the error committed when the flip-flop was converging to A_1 and changes direction can also be bounded by γ arbitrarily small.

The case when the flip-flop is converging is when both Start-Inc$_i(t)$ and Start-Dec$_i(t)$ are close to 0. Then by a simple error analysis we have:

$|\text{Start-Inc}_i(t) - \text{Start-Dec}_i(t)| < \frac{2}{100} \Rightarrow |\alpha_{f1}(\text{Start-Inc}_i(t) - \text{Start-Dec}_i(t))| < \frac{2|\alpha_{f1}|}{100}$.

<u>2^{nd} Case</u>

For the values of the constants considered in the previous case, we have the following results: if there were no errors coming from the finite control, i.e., if $\text{Start-Inc}_i(t) = \text{Start-Dec}_i(t) = 0$, then the fixed points A_1 and A_2 would be linear attractors for the update function of the flip-flops. It would be then possible, using the result of proposition 4.1.2 to show that around the fixed points there was a region with the following property: applying the update equation to a point in that region would reduce the distance to the fixed point by at least a half, since $\varrho'(x) < \frac{1}{2}, \forall x$. That way, the sum of all errors committed by the flip-flop during the cycle could be bound by arbitrarily small $\delta = \sum_{i=0}^{\infty} \frac{\gamma}{2^i} = 2\gamma$. The question lies now on knowing whether the small errors from the finite control jeopardize such behavior. And the answer is: depends on the values of α_{f2}. In figures 5.8 and 5.9 we can see the graphics of the ratios of the differences and the perturbation caused by the errors. In figure 5.8, when $\alpha_{f2} = 4$, there is a region where the relative errors are no longer controlled. In figure 5.9, we can see the difference when $\alpha_{f2} = 25$. The larger α_{f2} is, the smaller will be the ratio and the broader will be the region where that ratio is smaller than $\frac{1}{2}$.

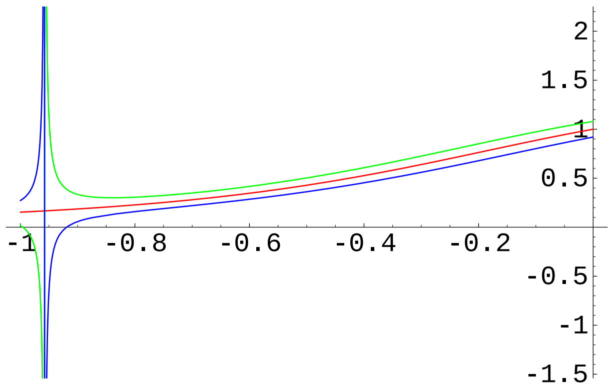

Figure 5.8: Plot of the ratios of the errors: $\frac{\varrho(4x)-A_1}{x-A_1}$, in red. The perturbations $\frac{\varrho(-0.02\alpha_{f1}4x)-A_1}{x-A_1}$and $\frac{\varrho(0.02\alpha_{f1}4x)-A_1}{x-A_1}$ are shown in green and blue, respectively.

We thus conclude that given inputs with controlled errors to the flip-flops, their outputs will have controlled errors.

$2 \Rightarrow 3$:

Resulting from the previous error analysis.

$3 \Rightarrow 1$:

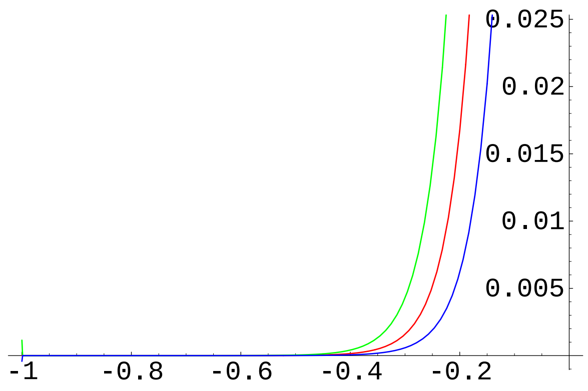

Figure 5.9: Same as figure 5.7, but now for $\alpha_{f2} = 25$.

Because the finite control is feedforward, and since each restless counter alarms, at most, two times in each day of the rc neuron implementing the alarm clock A_0(recall the proof of proposition 3.2.1), the finite control will output (intentionally) nonzero signals only a finite and bounded number of times a day. Let us recall from section 4.4, that the update equation for the finite control is

$$x_i(t+1) = \varrho(\alpha_a(\sum_{i=1}^{N} a_{ij}x_j + \theta_i)).$$

By adjusting the constants α_a, θ_i in our implementation of the finite control, we can make these neurons have arbitrarily small errors when they change their values. During the quiescent period, we can bound the errors caused by the restless counters being nonzero by some constant c times the sum of the values of all the restless counters at every time of the day. By choosing the weights appropriately, we can, in fact, make c as small as desired.

$\square$

The input to the neural network is encoded in the initial value of the rc-neuron that simulates the morning counter for the alarm clock A_1. In the end of the computation, the output is stored as the phase difference between two rc-neurons. The simplest way to read this output is to have a special neuron, activated only when the restless counter machine would halt, that "ensures" that these 2 rc-neurons will continue with regular cycle, inhibiting the respective *Start-Inc* and *Start-Dec* neurons; and halting the network once these two neurons become active.

The proof of theorem 5.6.1 may be adapted to any *sigmoidal-like* function, i.e., any function that satisfies the properties in section 4.1. Therefore we conclude that recurrent neural networks using any sigmoidal-like function ϱ are Turing universal. In particular, the commonly used nets following the update rule $x(t+1) = \rho(x(t))$ where $\rho : [-1,1]^n \to [-1,1]^n$ has the form

$$\rho_i(x) = tanh(\sum_{j=1}^{n} \alpha_{ij} x_j - \theta_i)$$

, for constants α_{ij} and θ_i.

Chapter 6

Conclusions

We conclude from the previous chapters that Turing universality is a general property of many recurrent neural networks and we make a brief analysis of the consequences of that fact, as well as some quick remarks concerning the final result of this thesis.

The equivalence between ARNNs and Turing machines has many implications regarding the decidability, or more generally, the complexity of questions about sigmoidal neural networks.

One of the consequences of this result (theorem 5.6.1) is that there is no computable limit on the running time of a network, although there are clearly particular networks with well-specified limits. As a consequence, in the presence of noise, the errors affect the result of the computation for long enough computations, no matter how small we constraint those errors to be.

Another consequence is that it is not possible, with all generality, to determine whether a network ever converges or enters a detectable oscillatory state, or even whether a given neuron ever gets close to 0 or 1. This is in contrast with some particular recurrent networks, such as the Hopfield network, where convergence is assured.

The contents of this thesis fulfills its main objectives, but is far from being a well finished work. The proofs could most certainly be simplified and stated more elegantly; the examples given could be better chosen and explained, and many passages could and should be treated with more care, whereas in some others there may be annoying repetitions.

We have tried, to the limits of our efforts, to make this thesis self-contained. The price to pay is an initial chapter (chapter 2), much larger than necessary. The same applies for our will to give examples for all the positive results that we have dealt with: this thesis could have been written in a definitely more concise way, although certainly a less clear one.

Finally, we would like to stress the fact that in this thesis knowledge from several branches of mathematics are combined in an unusual way: in particular, we have focused our attention on the error analysis of a not very trivial dynamical system. We believe that in the future, a closer interaction between Computer Science, Numerical Analysis, Dynamical Systems and other areas from Mathematics and Physics will become more and more important, not to say essential in the development of Computer Science.

Index

List of Figures

Bibliography

[Alv99] Carlos J. Alves. *Fundamentos de Análise Numérica.* Associação dos Estudantes do Instituto Superior Técnico - Secção de Folhas, 1999.

[Arb87] Michael A. Arbib. *Brains, Machines, and Mathematics , Second Edition.* Springer-Verlag, 1987.

[Atk88] Kendall A. Atkinson. *An Introduction to Numerical Analysis, Second Edition.* John Wiley & Sons, 1988.

[Dav82] Martin Davis. *Computability and Unsolvability, Second Edition.* Dover, 1982.

[FB94] Robert W. Floyd and Richard Beigel. *The Language of Machines: an introduction to computability and formal languages.* Computer Science Press, 1994.

[Gar95] Max H. Garzon. *Models of Massive Parallelism.* Springer, 1995.

[HS87] Ralph Hartley and Harold H. Szu. A comparison of the computational power of neural network models. *Proceedings, IEEE First International Conference on Neural Networks*, 3:17–22, 1987.

[HU01] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation, Second Edition.* Addison-Wesley, 2001.

[Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–42, 1956.

[Koi93] Pascal Koiran. *Puissance de calcul des réseaux de neurones artificiels.* PhD thesis, École Normale Supérieure de Lyon, Lyon,France, 1993.

[KS96] Joe Killian and Hava T. Siegelmann. The dynamic universality of sigmoidal neural networks. *Information and Computation*, 128(1):48–56, 1996.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

[MK99] Cristopher Moore and Pascal Koiran. Closed-form analytic maps in one and two dimensions can simulate universal turing machines. *Theoretical Computer Science*, (210):217–223, 1999.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, (5):115–133, 1943.

[MS98] Wolfgang Maass and Eduardo D. Sontag. Analog neural nets with gaussian or other common noise distribution cannot recognize arbitrary regular languages. *Neural Computation*, 1998.

[Par94] Ian Parberry. *Circuit Complexity and Neural Networks.* MIT Press, 1994.

[Pol87] Jordan B. Pollack. *On Connectionist Models of Natural Language Processing.* PhD thesis, University of Illinois, Urbana,Illinois,USA, 1987.

[Rog96] Yurii Rogozhin. Small universal turing machines. *Theoretical Computer Science*, 168:215–240, 1996.

[Sha56] Claude E. Shannon. A universal turing machine with two internal states. *Automata Studies*, pages 157–166, 1956.

[Sie99] Hava T. Siegelmann. *Neural Networks and Analog Computation - Beyond the Turing Limit.* Birkhäuser, 1999.

[SR98] Hava T. Siegelmann and Alexander Roitershtein. Noisy analog neural networks and definite languages: stochastic kernels approach. *Technical Report, Technion, Haifa, Israel*, 1998.

[SS91] Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–81, 1991.

[SS95] Hava T. Siegelmann and Eduardo D. Sontag. On computational power of neural networks. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

[WM93] David H. Wolpert and Bruce J. McLellan. A computationally universal field computer which is purely linear. *TR 93-06-09*, 1993. Santa Fé Institute.