Computation in Gene Networks

Asa Ben-Hur[1] and Hava T. Siegelmann[2]

[1] BioWulf Technologies
2030 Addison st. suite 102, Berkeley, CA 94704
[2] Lab for Inf. & Decision Systems
MIT Cambridge, MA 02139, USA
iehava@ie.technion.ac.il

Abstract. We present a biological computing paradigm that is based on genetic regulatory networks. Using a model of gene expression by piecewise linear differential equations we show that the evolution of protein concentrations in a cell can be considered as a process of computation. This is demonstrated by showing that this model can simulate memory bounded Turing machines. The simulation is robust with respect to perturbations of the system, an important property for both analog computers and biological systems.

1 Introduction

In recent years scientists have been looking for new paradigms for constructing computational devices. These include quantum computation [1], DNA computation [2], neural networks [3], neuromorphic engineering [4] and other analog VLSI devices. This paper describes a new paradigm based on genetic regulatory networks. The concept of a "genetic network" refers to the complex network of interactions between genes and gene products in a cell [5]. Since the 60's genetic regulatory systems are thought of as "circuits" or "networks" of interacting components [6], and were described in computer science terms: The genetic material is the "program" that guides protein production in a cell; protein levels determine the evolution of the network at subsequent times, and thus serve as its "memory". This analogy between computing and the process of gene expression was pointed out in various papers [7,8]. Bray suggests that protein based circuits are the device by which unicellular organisms react to their environment, instead of a nervous system [7]. However, until recently this was only a useful metaphor for describing gene networks. The papers [9,10] describe the successful fabrication of synthetic networks, i.e. *programming* of a gene network. In this paper we compare the power of this computational paradigm with the standard digital model of computation. In a related series of papers it is shown both theoretically and experimentally that chemical reactions can be used to implement Boolean logic and neural networks (see [11] and references therein).

Protein concentrations are continuous variables that evolve continuously in time. Moreover, biological systems do not have timing devices, so a description in terms of a *map* that simultaneously updates the system variables is inadequate. It is thus necessary to model gene networks by differential equations, and consider them as analog computers. The particular model of genetic networks we analyze here assumes switch-like behavior, so that protein concentrations are described by piecewise linear equations (see equation (1) below) [12,8], but we believe our results to hold for models that assume sigmoid response (see discussion). These equations were originally proposed as models of chemical oscillations in biological systems [13]. In this paper we make the analogy between gene networks and computational models complete by formulating an abstract computational device on the basis of these equations, and showing that this analog model can simulate a computation of a Turing machine [14]. The relation between digital models of computation and analog models is explored in a recent book [15], mainly from the perspective of neural networks. It is shown there that analog models are potentially stronger than digital ones, assuming an ideal noiseless environment. In this paper on the other hand we consider the possibility of noise and propose a design principle which makes the model equations robust. This comes at a price: we can only simulate memory bounded Turing machines. However, we argue that any physically realizable robust simulation has this drawback. On the subject of computation in a noisy environment see also [16,17]. We found that the gene network proposed in [9] follows this principle, and we quote from that paper their statement that "theoretical design of complex and practical gene networks is a realistic and achievable goal".

Computation with biological hardware is also the issue in the field of DNA computation [2]. As a particular example, the guided homologous recombination that takes place during gene rearrangement in ciliates was interpreted as a process of computation [18]. This process, and DNA computation in general, are symbolic, and describe computation at the molecular level, whereas gene networks are analog representations of the macroscopic evolution of protein levels in a cell.

2 Piecewise Linear ODEs for Gene Networks

In this section we present model equations for gene networks, and note a few of their dynamical properties [19]. The concentration of N proteins (or other biochemicals in a more general context) is given by N non-negative real variables $y_1, \dots, y_N$. Let $\theta_1, \dots, \theta_N$ be N threshold concentrations. The production rate of each protein is assumed to be constant until a threshold is crossed, when the production rate assumes a new value. This is expressed by the equations:

$$\frac{dy_i}{dt} = -k_i y_i + \tilde{\Lambda}_i(Y_1, \dots, Y_N) \ , \tag{1}$$

where Y_i is a Boolean variable associated with y_i, equal to 1 if $y_i \geq \theta_i$ and 0 otherwise; k_i is the degradation rate of protein i and $\tilde{\Lambda}_i$ is its production rate when gene i is "on". These equations are a special case of the model of Mestl et. al. [12] where each protein can have a number of thresholds, compared with just one threshold here (see also [20]). When there is just one threshold it is easy to associate Boolean values with the continuous variables. For simplicity we take $k_i = 1$, and define

$$x_i = y_i - \theta_i,$$

with the associated Boolean variables

$$X_i = \mathrm{sgn}(x_i),$$

where $\mathrm{sgn}(x) = 1$ for $x \geq 0$ and zero otherwise. Also denote $\Lambda_i(X_1, \ldots, X_N) = \tilde{\Lambda}_i(Y_1, \ldots, Y_N) - \theta_i$. Equation (1) now becomes:

$$\frac{dx_i}{dt} = -x_i + \Lambda_i(X_1, \ldots, X_N)\,, \tag{2}$$

and Λ_i is called the *truth table*. The set in $\mathbb{R}^N$ which corresponds to a particular value of a Boolean vector $X = (X_1, \ldots, X_N)$ is an *orthant* of $\mathbb{R}^N$. By abuse of notation, an orthant of $\mathbb{R}^N$ will be denoted by a Boolean vector X. The trajectories in an orthant are straight lines directed to a focal point $\Lambda(X) = (\Lambda_1(X), \ldots, \Lambda_N(X))$, as seen from equation (3) below. If the focal point $\Lambda(X)$ at a point $x = (x_1, \ldots, x_N)$ is in the same orthant as x, then the dynamics converges to the focal point. Otherwise it crosses the boundary to another orthant, where it is redirected to a different focal point. The sequence of orthants $X(1), X(2), \ldots$ that correspond to a trajectory $x(t)$ is called the *symbolic dynamics* of the vector field. In the next sections we will associate the symbolic dynamics of a a model Gene Network (GN) with a process of computation.

2.1 Dynamics

In an orthant all the Λ_i are constant, and the equations (2) are easily integrated. Starting from a point $x(0)$,

$$x_i(t) = \lambda_i + (x_i(0) - \lambda_i)e^{-t}\ , \tag{3}$$

where $\lambda_i = \Lambda_i(X_1(0), \ldots, X_N(0))$. The time t_i at which the hyper-plane $x_i = 0$ is crossed is given by

$$t_i = \ln \frac{\lambda_i - x_i(0)}{\lambda_i}$$

The switching time t_s is the time it takes to reach a new orthant:

$$t_s = \min_i t_i$$

We will consider networks with $\Lambda_i = \pm 1$ in all orthants. Thus when the focal point is in a different orthant than $x(0)$ we have that

$$t_s \leq \ln 2\ . \tag{4}$$

This gives a criterion for determining whether a GN is converging to a fixed point: if the time from the last switching time is larger than $\ln 2$, then the system is converging to the current focal point.

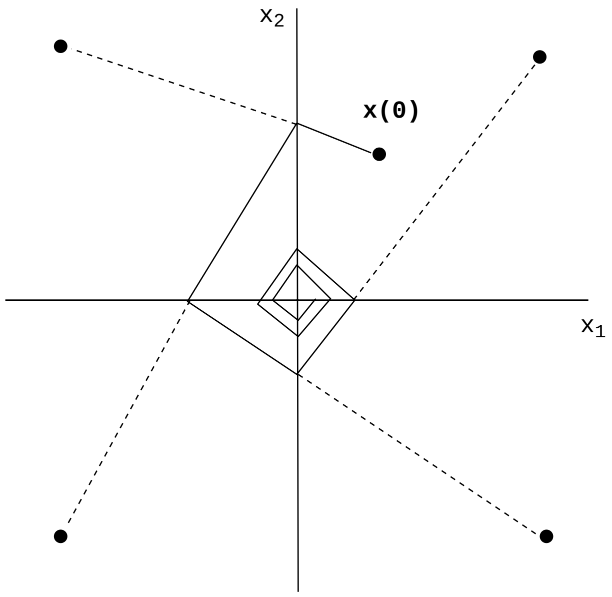

Fig. 1. A piecewise linear flow in two dimensions.

A network as in (2) is a continuous time version of a discrete network:

$$Z_i \mapsto \operatorname{sgn}(\Lambda_i(Z)) \ , \tag{5}$$

where $Z \in \{0,1\}^N$ is a vector of Boolean variables. This dynamics does not necessarily visit the same sequence of orthants as the corresponding continuous network.

We want to simulate a discrete dynamical system (Turing machine) with a continuous time GN. To bridge the gap, we first construct a discrete gene network of the form (5), whose corresponding continuous GN has the same symbolic dynamics. We will use truth tables with the following property:

Definition 1. *Two orthants are said to be* adjacent *if they differ in exactly one coordinate. A truth table Λ will be called* adjacent *if $\Lambda(X)$ is in the same orthant as X or in an adjacent orthant to X for all $x \in \mathbb{R}^N$. A network with an adjacent truth table will also be called adjacent.*

We note that in an adjacent network, all initial conditions in an orthant lead to the same adjacent orthant. Therefore all initial conditions of (2) in the same orthant have the same symbolic dynamics. An immediate result is the following lemma:

Lemma 1. *Let Λ be an adjacent truth table, then the symbolic dynamics of a continuous GN is the same as the dynamics of its discrete counterpart (5): $X(k) = Z(k),\ k = 0, 1, \ldots$, where $Z(k)$ is the kth iterate of (5) and $X(k)$ is the kth orthant visited by (2), for every initial condition of (2) corresponding to Z_0.*

In view of the above discussion, the focal points and the initial condition of an adjacent network can be taken as points in $\{-1,1\}^N$, and the problem of constructing a continuous network that simulates a discrete dynamics is reduced to the problem of constructing a discrete network that changes only one variable at each iteration.

When the truth table is not adjacent, a discrete network may show qualitatively different dynamics than its continuous counterpart: continuous high-dimensional GN's are "typically" chaotic [21,22]. However chaos is a dynamical behavior that is impossible in dynamical systems with a finite state space. The lack of sensitivity of the dynamics of an adjacent GN to the placement of the initial condition, and its equivalence to a discrete network leads to the following statement:

Corollary 1. *Adjacent networks are not chaotic.*

This implies a measure of stability to the dynamics of adjacent networks. Lack of sensitivity to perturbations of the system is captured by the following property of adjacent networks:

Claim. Let Λ be an adjacent truth table, and let $\tilde{\Lambda}$ be a truth table whose entries are $\Lambda_i(X) + \delta_i(X)$, where $\delta_i(X) \in [-c, c]$ for some $1 > c > 0$. Then $\tilde{\Lambda}$ is also adjacent, and the two networks have the same symbolic dynamics.

The robustness of adjacent networks is an important property for both analog computers and biological systems. The robustness of biological systems leads us to speculate that adjacency might be a principle underlying the robust behavior in the modeled systems. Adjacency can also be taken as a principle which can be used in the design of robust networks.

3 Preliminaries

In this section we give the definition of the Turing machine that will be simulated and provide the relevant concepts from complexity theory [14].

Definition 2. *A Turing machine is a tuple $M = (K, \Sigma, \Gamma, \delta, Q_1, Q_q)$. $K = \{Q_1, \ldots, Q_q\}$ is a finite set of* states*; $Q_1, Q_q \in K$ are the* initial/halting *states respectively; Σ is the* input alphabet*; $\Gamma = \Sigma \cup \{blank, \#\}$ is the* tape alphabet *which includes the* blank symbol *and the left end symbol, #, which marks the end of the tape; $\delta : K \times \Gamma \to K \times \Sigma \times \{L, R\}$ is the* transition function*, and L/R signify left/right movement of the read-write head. The transition function is such that it cannot proceed left of the left end marker, and does not erase it, and every tape square is blank until visited by the read-write head.*

At the beginning of a computation an input sequence is written on the tape starting from the square to the right of the left-end marker. The head is located at the leftmost symbol of the input string, and the finite-control is initialized at its start state Q_1. The computation then proceeds according to the transition function, until reaching the halting state. Without loss of generality we suppose that the input alphabet, Σ, is $\{0, 1\}$. We say that a Turing machine *accepts* an input word w if the computation on input w reaches the halting state with "1" written on the tape square immediately to the right of the left-end marker. If the machine halts with "0" at the first tape square, then the input is rejected by the machine. Other conventions of acceptance are also common. The language of a Turing machine M is the set $L(M)$ of strings accepted by M.

Languages can be classified by the computational resources required by a Turing machine which accepts them. The classification can be according to *time* or *space* (memory). The time complexity of a computation is the number of steps until halting and its space complexity is the number of tape squares used in the computation. The complexity classes P and PSPACE are defined to be the classes of languages that are accepted in polynomial time and polynomial space, respectively. More formally, a language L is in PSPACE if there exists a Turing machine M which accepts L, and there exists $c > 0$ such that on all inputs of length n M accesses $O(n^c)$ tape squares. To make a finer division one defines the class SPACE($s(n)$) of languages that are accepted in space $s(n)$.

3.1 On the Feasibility of Turing Machine Simulation

Turing machine simulations by differential equations appear in a number of papers: in [23] it was shown that an ODE in four dimensions can simulate a Turing machine, but the explicit form of the ODE is not provided. Finite automata and Turing machines are simulated in [24] by piecewise constant ODEs. Their method is related to the one used here.

A Turing machine has a countably infinite number of configurations. In order to simulate it by an ODE, an encoding of these configurations is required. These can essentially be encoded into a continuum in two ways:

- in a bounded set, and two configurations can have encodings that are arbitrarily close;
- in an unbounded set, keeping a minimal distance between encodings.

In the first possibility, arbitrarily small noise in the initial condition of the simulating system may lead to erroneous results, and is therefore unfeasible (this point is discussed in [25] in the context of simulating a Turing machine by a map in $\mathbb{R}^2$). The second option is also unfeasible since a physical realization must be finite in its extent. Thus simulation of an arbitrary Turing by a realizable analog machine is not possible, and simulation of resource bounded Turing machines is required. In this chapter we will show a robust simulation of memory bounded Turing machines by adjacent GN's.

4 Computing with GN's

In this section we formulate GN's as computational machines. The properties of adjacent networks suggest a natural interpretation of an orthant X as a robust representation of a discrete configuration. The symbolic dynamics of a network, i.e. the values $X(t)$ will be interpreted as a series of configurations of a discrete computational device.

Next we specify how it receives input and produces an output. A subset of the variables will contain the input as part of the initial condition of the network, and the rest of the variables will be initialized in some uniform way. Since we are considering $\Sigma = \{0, 1\}$, the input is encoded into the binary values $X(0)$. To specify a specific point in the orthant X we choose -1 to correspond to 0 and 1 to correspond to 1. Note that for adjacent networks *any* value of $x(0)$ corresponding to $X(0)$ leads to the same computation, so this choice is arbitrary.

In addition to input variables, another subset of the variables is used as output variables. For the purpose of language accepting a single output variable is sufficient. There are various ways of specifying halting. One may use a dynamical property, namely convergence to a fixed point, as a halting criterion. We noted that convergence to a fixed point is identified when after a time $\ln 2$ no switching has occurred (see equation (4)). Non-converging dynamics correspond to non-halting computations. While such a definition is natural from a dynamics point of view, it is not biologically plausible, since a cell evolves continuously, and convergence to a fixed point has the meaning of death. Another approach is to set aside a variable that will signify halting. Let this variable be X_N. It is initially set to 0, and when it changes to 1, the computation is complete, and the output may be read from the output variables. In this case non-halting computations are trajectories in which X_N never assumes the value 1. After X_N has reached the value 1, it may be set to 0, and a new computational cycle may begin. A formal definition of a GN as a computational machine is as follows.

Definition 3. *A genetic network is a tuple $G = (V, I, O, \Lambda, x_0, X_N)$, where V is the set of variables indexed by $\{1, \ldots, N\}$; $I \subseteq V$, $|I| = n$ and $O \subseteq V$, $|O| = m$ are the set of* input *and* output *variables respectively; $\Lambda : \{0, 1\}^N \to \{-1, 1\}^N$ is a truth table for a flow (2); $x_0 \in \{-1, 1\}^{N-n}$ is the initialization of variables in $V \setminus I$. X_N is the halting variable that is initialized to 0. A computation is halted the first time that $X_N = 1$.*

A GN G with n input variables and m output variables computes a partial mapping

$$f_G : \{0, 1\}^n \to \{0, 1\}^m,$$

which is the value of X_i for $i \in O$ when the halting state $X_N = 1$ is reached. If on input w the net does not reach a halting state then $f_G(w)$ is undefined.

We wish to characterize the languages accepted by GN's. For this purpose it is enough to consider networks with a single output variable, and say that a GN G *accepts* input $w \in \{0, 1\}^*$ if $f_G(w) = 1$. A fixed network has a constant number of input variables. Therefore it can only accept languages of the form

$$L_n = L \cap \{0, 1\}^n , \qquad (6)$$

where $L \subseteq \{0, 1\}^*$. To accept a language which contains strings of various lengths we consider a family $\{G_n\}_{n=1}^{\infty}$ of networks, and say that such a family accepts a language L if for all n, $L(G_n) = L_n$.

Before we define the complexity classes of GN's we need to introduce the additional concept of *uniformity*. A GN with N variables is specified by the entries of its truth table Λ. This table contains the 2^N focal points of the system. Thus the encoding of a general GN requires an exponential amount of information. In principle one can use this exponential amount of information to encode *every* language of the form $L \cap \{0,1\}^n$, and thus a series of networks exists for every language $L \subseteq \{0,1\}^*$. Turing machines on the other hand, accept only the subset of *recursive* languages [14]. A Turing machine is finitely specified, essentially by its transition function, whereas the encoding of an infinite series of networks is not necessarily finite. To obtain a series of networks that is equivalent to a Turing machine it is necessary to impose the constraint that truth tables of the series of networks should all be created by one finitely encoded machine. The device which computes the truth table must be simple in order to demonstrate that the computational power of the network is not a by-product of the computing machine that generates its transition function, but of the complexity of its time evolution. We will use a finite automaton with output [26], which is equivalent to a Turing machine which uses constant space. The initial condition of the series of networks needs also to be computable in a uniform way, in the same way as the truth table, since it can be considered as "advice", as in the model of advice Turing machines [14]. We now define:

Definition 4. *A family of networks* $\{G_n\}_{n=1}^{\infty}$ *is called* uniform *if there exist constant memory Turing machines* M_1, M_2 *that compute the truth table and initial condition as follows: on input* $X \in \{0,1\}^n$ M_1 *outputs* $\Lambda(X)$*;* M_2 *outputs* x_0 *for* G_n *on input* 1^n *..*

With the definition of uniformity we can define complexity classes. Given a function $s(n)$, we define the class of languages which are accepted by networks with less than $s(n)$ variables (not including the input variables):

$$GN(s(n)) = \{L \subseteq \{0,1\}^* \mid \text{there exists a uniform family of networks } \{G_n\}_{n=1}^{\infty}, \text{ s.t. } L(G_n) = L_n \text{ and } N \leq s(n) + n\}$$

The class with polynomial $s(n)$ will be denoted by PGN. The classes Adjacent-GN($s(n)$) and Adjacent-PGN of adjacent networks are similarly defined. Time constrained classes can also be defined.

5 The Computational Power of GN's

In this section we outline the equivalence between memory-bounded Turing machines and networks with adjacent truth tables. We begin with the following lemma:

Lemma 2. *Adjacent-GN*$(s(n)) \subseteq$ *SPACE*$(s(n))$*.*

Proof. Let $L \in$ Adjacent-GN($s(n)$). There exists a family of adjacent GN's, $\{G_n\}_{n=1}^{\infty}$ with truth tables $\Lambda^{(n)}$, such that $L(G_n) = L_n$, and constant memory

Turing machines M_1, M_2 that compute $\Lambda^{(n)}$ and x_0. Since $\Lambda^{(n)}$ is an adjacent truth table, and we are only interested in its symbolic dynamics, we can use the corresponding discrete network. The Turing machine M' for L will run M_1 to obtain the initial condition, and then run M_2 to generate the iterates of the discrete network. M' will halt when the network has reached a fixed point. The network G_n has $s(n)$ variables on inputs of length n, therefore the simulation requires at least that much memory. It is straightforward to verify that $O(s(n))$ space is also sufficient.

The Turing machine simulation in the next section shows:

Lemma 3. *Let M be a Turing machine working in space $s(n)$, then there exists a sequence of uniform networks $\{G_n\}_{n=1}^{\infty}$ with $s(n)$ variables such that $L(G_n) = L(M) \cap \{0,1\}^n$.*

We conclude:

Theorem 1. *Adjacent-GN($s(n)$)=SPACE($s(n)$).*

As a result of claim 2.1 we can state that adjacent networks compute robustly. This is unlike the case of dynamical systems which simulate arbitrary Turing machines, where arbitrarily small perturbations of a computation can corrupt the result of a computation (see e.g. [24,27,15]). The simulation of memory bounded machines is what makes the system robust. The above theorem gives only a lower bound on the computational power of the general class of GN's, i.e.:

Corollary 2. *SPACE($s(n)$) $\subseteq$ GN($s(n)$).*

One can obtain non-uniform computational classes in two ways, either by allowing advice to appear as part of the initial condition, or by using a weaker type of uniformity for the truth table. This way one can obtain an equivalence with a class of the type PSPACE/poly [14].

6 Turing Simulation by GN's

Since the discrete and continuous networks associated with an adjacent truth table have the same symbolic dynamics, it is enough to describe the dynamics of an adjacent discrete network. We show how to simulate a space bounded Turing machine by a discrete network whose variables represent the tape contents, state and head position. An adjacent map updates a single variable at a time. To simulate a general Turing machine by such a map each computational step is broken into a number of operations: updating the tape contents, moving the head, and updating the state; these steps are in turn broken into steps that can be performed by single bit updates.

Let M be a Turing machine that on inputs of length n uses space s. Without loss of generality we suppose that the alphabet of the Turing machine is $\Sigma = \{0,1\}$. To encode the three symbols $\{0, 1, \mathit{blank}\}$ by binary variables we use a pair of variables for each symbol. The first variable of the pair is zero iff the corresponding tape position is blank; "0" is encoded as "10" and "1" is encoded as "11". Note that the left end marker symbol need not be encoded since the variables of the network have numbers. We construct a network G with variables $Y_1, \dots, Y_s; B_1, \dots, B_s; P_1, \dots, P_s; Q_1, \dots, Q_q$ and auxiliary variables $B, Y, Q'_1, \dots, Q'_q, C_1, \dots, C_4$. The variables

$$Y_1, \dots, Y_s; B_1, \dots, B_s$$

store the contents of the tape: B_i indicates whether the square i of the tape is blank or not and Y_i is the binary value of a non-blank square. The input is encoded into the variables $Y_1, \dots, Y_n, B_1, \dots, B_n$. Since the Turing machine signifies acceptance of an input by the value of its left-most tape square, we take the output to be the value of Y_1. The position of the read-write head is indicated by the variables

$$P_1, \dots, P_s$$

If the head is at position i then $P_i = 1$ and the rest are zero. The state of the machine will be encoded in the variables

$$Q_1, \dots, Q_q$$

where Q_1 is the initial state and Q_q is the halting state of the machine. State i will be encoded by $Q_i = 1$ and the rest zero.

As mentioned above, a computation of the Turing machine is broken into a number of single bit updates. After updating variables related to the state or the symbol at the head position, information required to complete the computation step is altered. Therefore we need the following temporary variables

- Y,B - the current symbol
- $Q'_1, \dots, Q'_q$ - a copy of the current state variables.

A computation step of the Turing machine is simulated in four stages:

1. Update the auxiliary variables $Y, B, Q'_1, \dots, Q'_q$ with the information required for the current computation step;
2. Update the tape contents;
3. Move the head;
4. Update the state.

We keep track of the stage at which the simulation is at with a set of variables $C_1, \dots, C_4$ which evolve on a cycle which corresponds to the cycle of operations (1)-(4) above. Each state of the cycle is associated with an update of a single variable. After a variable is updated the cycle advances to its next state. However, since a variable update does not always change the value of a variable, e.g. the machine does not have to change the symbol at the head position, and since the GN needs to advance to a nearby orthant or else it enters into a fixed point each update is of the form:

```
If the variable pointed to by the cycle
   subnetwork needs updating - update it
else
   advance to the next state of the cycle
```

Suppose that the head of M is at position i and state j, and that in the current step Y_i is changed to value Y_i^+ which is a non-blank, the state is changed to state k, and the head is moved to position $i+1$. The sequence of variable updates with the corresponding cycle states is as follows:

cycle state	variable update		
0000	Y	$\leftarrow Y_i$	
0001	B	$\leftarrow B_i$	
0011	Q'_j	$\leftarrow 1$	
0111	Y_i	$\leftarrow Y_i^+$	update variable at head
0110	B_i	$\leftarrow 1$	Y_i^+ non-blank
1110	P_{i+1}	$\leftarrow 1$	new position of head
1111	P_i	$\leftarrow 0$	erase old position of head
1011	Q_k	$\leftarrow 1$	new state
1001	Q_j	$\leftarrow 0$	erase old state
1000	Q'_j	$\leftarrow 0$	prepare for next cycle

At the end of a cycle a new cycle is initiated.

On input $w = w_1 w_2 \ldots w_n$, $w_i \in \{0, 1\}$ the system is initialized as follows:

$$\begin{array}{lcl} Y_i & = & w_i,\ i = 1, \ldots, s \\ B_i & = & 1,\ i = 1, \ldots, s \\ P_1 & = & 1 \\ Q_1 & = & 1 \\ \multicolumn{3}{l}{\text{all other variables: } 0} \end{array}$$

This completes the definition of the network. The computation of the initial condition is trivial, and computing the truth table of this network is essentially a bit by bit computation of the next configuration of the simulated Turing machine, which can be carried out in constant space. The network we have defined has $O(s)$ variables, and each computation step is simulated by no more than 20 steps of the discrete network. □

Remark 1. It was pointed out that the Hopfield neural network is related to GN's [28]. Theorem 1 can be proved using complexity results about asymmetric and Hopfield networks found in [29,30]. However, in this case it is harder to define uniformity, and the direct approach taken here is simpler.

7 Language Recognition vs. Language Generation

The subclass of GN's with adjacent truth-tables has relatively simple dynamics whose attractors are fixed points or limit cycles. It is still unknown whether non-adjacent GN's are computationally stronger than adjacent GN's. General GN's can have chaotic dynamics, which are harder to simulate with Turing machines. We have considered GN's as *language recognizers*: where the input arrives at the beginning of a computation, and the decision to accept or reject is based on its state when a halting state is reached. In the context of language recognition, chaotic dynamics does not add computational power: given a chaotic system that accepts a language, there is a corresponding system that does not have a chaotic attractor for inputs on which the machine halts; such a system is obtained e.g., by defining the halting states as fixed points. However, one can also consider GN's as *language generators* by viewing the symbolic dynamics of these systems as generating strings of some language. In this case a single GN can generate strings of arbitrary length. But even in this case chaos is probably not very "useful", since the generative power of structurally stable chaotic systems is restricted to the simple class of regular languages [31]. More complex behavior can be found in dynamical systems at the onset of chaos (see [32,33]). Continuously changing the truth table can lead to a transition to chaotic behavior [34]. At the transition point complex symbolic dynamics can be expected, behavior which is not found in discrete Boolean networks.

8 Discussion

In this paper we formulated a computational interpretation of the dynamics of a switch-like ODE model of gene networks. We have shown that such an ODE can simulate memory bounded Turing machines. While in many cases such a model provides an adequate description, more realistic models assume sigmoidal response. In the neural network literature it is proven that sigmoidal networks with a sufficiently steep sigmoid can simulate the dynamics of switch-like networks [15,35]. This suggests that the results presented here carry over to sigmoidal networks as well.

The property of adjacency was introduced in order to reduce a continuous gene network to a discrete one. We consider it as more than a trick, but rather as a strategy for fault tolerant programming of gene networks. In fact, the genetic toggle switch constructed in [9] has this property. However, when it comes to non-synthetic networks, this may not be the case - nature might have other ways of programming them, which are not so transparent.

References

1. M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.
2. L. Kari. DNA computing: the arrival of biological mathematics. *The mathematical intelligencer*, 19(2):24–40, 1997.
3. J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City, 1991.
4. C. Mead. *Analog VLSI and Neural Systems.* Addison-Wesley, 1989.
5. H. Lodish, A. Berk, S.L. Zipursky, P. Matsudaira, D. Baltimore, and J. Darnell. *Molecular cell biology.* W.H. Freemand and Company, 4th edition, 2000.
6. S.A. Kauffman. Metabolic stability and epigenesis in randomly connected nets. *Journal of Theoretical Biology*, 22:437, 1969.
7. D. Bray. Protein molecules as computational elements in living cells. *Nature*, 376:307–312, July 1995.

8. H.H. McAdams and A. Arkin. Simulation of prokaryotic genetic circuits. *Annual Review of Biophysics and Biomolecular Structure*, 27:199–224, 1998.
9. T.S. Gardner, C.R. Cantor, and J.J. Collins. Construction of a genetic toggle switch in E. coli. *Nature*, 403:339–342, January 2000.
10. M.B. Elowitz and S. Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403:335–338, January 2000.
11. A. Arkin and J. Ross. Computational functions in biochemical reaction networks. *Biophysical Journal*, 67:560–578, 1994.
12. T. Mestl, E. Plahte, and S.W. Omholt. A mathematical framework for describing and analyzing gene regulatory networks. *Journal of Theoretical Biology*, 176:291–300, 1995.
13. L. Glass. Combinatorial and topological method in chemical kinetics. *Journal of Chemical Physics*, 63:1325–1335, 1975.
14. C. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, Mass., 1995.
15. H.T. Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing Limit.* Birkhauser, Boston, 1999.
16. H.T. Siegelmann, A. Roitershtein, and A. Ben-Hur. Noisy neural networks and generalizations. In *Proceedings of the Annual Conference on Neural Information Processing Systems 1999 (NIPS*99).* MIT Press, 2000.
17. W. Maass and P. Orponen. On the effect of analog noise in discrete time computation. *Neural Computation*, 10(5):1071–1095, 1998.
18. L.F. Landweber and L. Kari. The evolution of cellular computing: nature's solution to a computational problem. In *Proceedings of the 4th DIMACS meeting on DNA based computers*, pages 3–15, 1998.
19. L. Glass and J.S. Pasternack. Stable oscillations in mathematical models of biological control systems. *Journal of Mathematical Biology*, 6:207–223, 1978.
20. R.N. Tchuraev. A new method fo rthe analysis of the dynamics of the molecular genetic control systems. I. description of the method of generalized threshold models. *Journal of Theoretical Biology*, 151:71–87, 1991.
21. T. Mestl, R.J. Bagley, and L. Glass. Common chaos in arbitrarily complex feedback networks. *Physical Review Letters*, 79(4):653–656, 1997.
22. L. Glass and C. Hill. Ordered and disordered dynamics in random networks. *Europhysics Letters*, 41(6):599–604, 1998.
23. M.S. Branicky. Analog computation with continuous ODEs. In *Proceedings of the IEEE Workshop on Physics and Computation*, pages 265–274, Dallas, TX, 1994.
24. E. Asarin, O. Maler, and A. Pnueli. Reachability analysis of dynamical systems with piecewise-constant derivatives. *Theoretical Computer Science*, 138:35–66, 1995.
25. A. Saito and K. Kaneko. Geometry of undecidable systems. *Prog. Theor. Phys.*, 99:885–890, 1998.
26. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
27. P. Koiran and C. Moore. Closed-form analytic maps in one and two dimensions can simulate universal Turing machines. *Theoretical Computer Science*, 210:217–223, 1999.
28. J.E. Lewis and L. Glass. Nonlinear dynamics and symbolic dynamics of neural networks. *Neural Computation*, 4:621–642, 1992.
29. P. Orponen. The computational power of discrete hopfield nets with hidden units. *Neural Computation*, 8:403–415, 1996.

30. P. Orponen. Computing with truly asynchronous threshold logic networks. *Theoretical Computer Science*, 174:97–121, 1997.
31. C. Moore. Generalized one-sided shifts and maps of the interval. *Nonlinearity*, 4:727–745, 1991.
32. J.P. Crutchfield and K. Young. Computation at the onset of chaos. In W.H. Zurek, editor, *Complexity, Entropy and the Physics of Information*, pages 223–269, Redwood City, CA, 1990. Addison-Wesley.
33. C. Moore. Queues, stacks, and transcendentality at the transition to chaos. *Physica D*, 135:24–40, 2000.
34. R. Edwards, H.T. Siegelmann, K. Aziza, and L. Glass. Symbolic dynamics and computation in model gene networks. in preparation.
35. J. Sima and P. Orponen. A continuous-time hopfield net simulation of discrete neural networks. Technical Report 773, Academy of Sciences of the Czech Republic, 1999.