

University of Massachusetts Amherst

From the Selected Works of Hava Siegelmann

March, 2004

Computation in Gene Networks

Hava Siegelmann, *University of Massachusetts - Amherst*
Asa Ben-Hur



Available at: https://works.bepress.com/hava_siegelmann/7/

Computation in gene networks

Asa Ben-Hur

Department of Biochemistry, Stanford University, Stanford, California 94305-5307

Hava T. Siegelmann

Department of Computer Science, University of Massachusetts at Amherst, Amherst, Massachusetts 01003

(Received 4 August 2003; accepted 23 October 2003; published online 31 December 2003)

Genetic regulatory networks have the complex task of controlling all aspects of life. Using a model of gene expression by piecewise linear differential equations we show that this process 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. Robustness is achieved using a condition that ensures that the model equations, that are generally chaotic, follow a predictable dynamics. © 2004 American Institute of Physics. [DOI: 10.1063/1.1633371]

Since the 1960s genetic regulatory systems 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,” etc. This is not only a useful metaphor for describing gene networks: using a model of gene expression by piecewise linear differential equations we formulate a digital computational device, thus making precise the analogy between gene networks and computational models. The simulation of the digital computational device by a set of differential equations shown here is robust with respect to perturbations, an important property for both analog computers and biological systems.

I. 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 computational 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 1960s 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 Ref. 11 and references therein).

In this paper we aim to show that the dynamics of a model gene network is complex enough to be equivalent in its expressive power to the dynamics of a Turing machine, which is one of many equivalent abstractions of a digital computer. But first, we must fix a model for gene networks. There is a large variety of models that are used to describe gene networks¹² ranging from Boolean networks.

However, when discussing dynamics we should keep in mind that 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 reasonable to suggest that differential equations are perhaps the most expressive way of modeling their complex dynamics. This modeling choice makes gene networks analog computers. The particular model of genetic networks we analyze here assumes switchlike behavior, so that protein concentrations are described by piecewise linear equations [see Eq. (1) below],^{8,13} 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.¹⁴ 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.¹⁵ The relation between digital models of computation and analog models is explored in a recent book,¹⁶ 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. While the equations are in general chaotic,^{17,18} our design principle makes them nonchaotic. On the subject

of computation in a noisy environment see also Refs. 19 and 20. We found that the gene network proposed in Ref. 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.²¹ 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.

II. PIECEWISE LINEAR ODES FOR GENE NETWORKS

In this section we present model equations for gene networks, and note a few of their dynamical properties.²² 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

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

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.*¹³ where each protein can have a number of thresholds, compared with just one threshold here (see also Ref. 23). 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

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

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

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

and Λ_i is called the *truth table*. The set in \mathbb{R}^N which corresponds to a particular value of a Boolean vector $X = (X_1, \dots, X_N)$ 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 $\Lambda(X) = (\Lambda_1(X), \dots, \Lambda_N(X))$, as seen from Eq. (3). If the focal point $\Lambda(X)$ at a point $x = (x_1, \dots, x_N)$ is in the same orthant as x , then the dynamics converge to the focal point. Otherwise it crosses the boundary to another orthant, where it is redirected to a different focal point (see Fig. 1). The sequence of orthants $X(1), X(2), \dots$ that correspond to a tra-

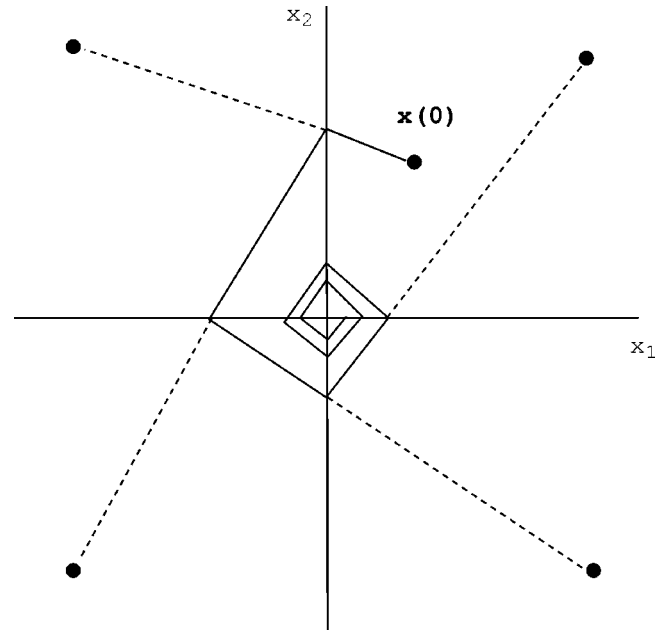


FIG. 1. A piecewise linear flow in two dimensions.

jectory $x(t)$ is called the *symbolic dynamics* of the vector field. In the next sections we will associate the symbolic dynamics of a model gene network (GN) with a process of computation.

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

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

where $\lambda_i = \Lambda_i(X_1(0), \dots, X_N(0))$. By setting $x_i(t) = 0$ in Eq. (3) we obtain the time t_i at which the hyperplane $x_i = 0$ is crossed,

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

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

$$t_s \leq \ln 2. \quad (4)$$

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.

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

$$Z_i \mapsto \text{sgn}(\Lambda_i(Z)), \quad (5)$$

where $Z \in \{0, 1\}^N$ is a vector of Boolean variables. This dynamic 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): $X(k) = Z(k)$, $k = 0, 1, \dots$, where $Z(k)$ is the k th iterate of (5) and $X(k)$ is the k th 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 GNs are “typically” chaotic.^{17,18} 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 1: 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.

III. PRELIMINARIES

In this section we give the definition of the Turing machine that will be simulated and provide the relevant concepts from complexity theory.¹⁵

Definition 2: A Turing machine is a tuple $M = (K, \Sigma, \Gamma, \delta, Q_1, Q_q)$. $K = \{Q_1, \dots, Q_q\}$ is a finite set of *states*; $Q_1, Q_q \in K$ are the *initial/halting* states, respectively; Σ is the *input alphabet*; $\Gamma = \Sigma \cup \{\text{blank}, \#\}$ is the *tape alphabet* which includes the *blank symbol* and the *left end symbol*, $\#$, which marks the end of the tape; $\delta: K \times \Gamma \rightarrow K \times \Sigma \times \{L, R\}$ 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 $\text{SPACE}(s(n))$ of languages that are accepted in space $s(n)$.

On the feasibility of Turing machine simulation: Turing machine simulations by differential equations appear in a number of papers: in Ref. 24 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 Ref. 25 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:

- (1) In a bounded set, and two configurations can have encodings that are arbitrarily close;
- (2) 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

Ref. 26 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 paper we will show a robust simulation of memory bounded Turing machines by adjacent GNs.

IV. COMPUTING WITH GNS

In this section we formulate GNs as computational machines. The properties of adjacent networks suggest a natural interpretation of an orthant X as a 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 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 Eq. (4)]. Nonconverging 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 nonhalting 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 $\{1, \dots, N\}$; $I \subseteq V$, $|I| = n$ and $O \subseteq V$, $|O| = m$ are the set of *input* and *output* variables, respectively; $\Lambda: \{0,1\}^N \rightarrow \{-1,1\}^N$ 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

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

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 GNs. 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

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

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 GNs 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.¹⁵ 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,²⁷ 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.¹⁵ We now define the following.

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 $X \in \{0,1\}$ the machine M_1 outputs the truth table, $\Lambda(X)$; the machine M_2 outputs the initial condition x_0 for network 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),

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

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.

V. THE COMPUTATIONAL POWER OF GNS

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 GNs, $\{G_n\}_{n=1}^{\infty}$ 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 the following.

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 as follows.

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

As a result of claim 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., Refs. 16, 25, and 28). 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 GNs.

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

One can obtain nonuniform 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.¹⁵

VI. TURING SIMULATION BY GNS

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 con-

tents, 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 $\{0, 1, \text{blank}\}$ 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 nonblank 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 leftmost 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 nonblank, 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^+ nonblank
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 $w = w_1 w_2 \dots w_n$, $w_i \in \{0,1\}$ the system is initialized as follows:

$$Y_i = w_i, \quad i = 1, \dots, s,$$

$$B_i = 1, \quad i = 1, \dots, s,$$

$$P_1 = 1,$$

$$Q_1 = 1,$$

all other variables 0.

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 GNs.²⁹ Theorem 1 can be proved using complexity results about asymmetric and Hopfield networks found in Refs. 30 and 31. However, in this case it is harder to define uniformity, and the direct approach taken here is simpler.

VII. LANGUAGE RECOGNITION VS LANGUAGE GENERATION

The subclass of GNs with adjacent truth tables has relatively simple dynamics whose attractors are fixed points or limit cycles. It is still unknown whether nonadjacent GNs are computationally stronger than adjacent GNs. General GNs can have chaotic dynamics, which are harder to simulate with Turing machines. We have considered GNs 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 GNs 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.³² More complex behavior can be found in dynamical systems at the onset of chaos (see Refs. 33 and 34). Continuously changing the truth table can lead to a transition to chaotic behavior.³⁵ At the transition point complex symbolic dynamics can be expected, behavior which is not found in discrete Boolean networks.

VIII. DISCUSSION

In this paper we formulated a computational interpretation of the dynamics of a switchlike ODE model of gene networks. We have shown that families of such ODEs with an increasing number of variables (“genes”) can simulate memory bounded Turing machines. The computer science analogy can help in understanding the apparent discrepancy between the number of genes predicted in humans vs lower eukaryotes such as *C. elegans* (humans have only twice as many genes as *C. elegans*), and the apparent difference in complexity between these organisms: more complex regulatory networks in the higher eukaryotes allow them to access a larger fraction of their state space, i.e., make more use of their available “memory” (see Ref. 36).

Our model of gene regulation considered only transcript abundance and ignored other possible variables such as post-translational modifications, transcript half-life, and other variables that may encode information about the state of the system. These variables represent additional memory available to the system, so the overall analogy remains.

While in many cases a switchlike ODE provides an adequate description, more realistic models assume sigmoidal response. In the neural network literature it is proved that sigmoidal networks with a sufficiently steep sigmoid can simulate the dynamics of switchlike networks.^{16,37} 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 Ref. 9 has this property. However, when it comes to nonsynthetic networks, nature does not adhere to this principle—some genes affect the transcription of a large number of genes.

- ¹M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, 2000).
- ²L. Kari, "DNA computing: The arrival of biological mathematics," *Math. Intell.* **19**, 24–40 (1997).
- ³J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation* (Addison-Wesley, Redwood City, CA, 1991).
- ⁴C. Mead, *Analog VLSI and Neural Systems* (Addison-Wesley, New York, 1989).
- ⁵H. Lodish, A. Berk, S. L. Zipursky, P. Matsudaira, D. Baltimore, and J. Darnell, *Molecular Cell Biology* (W. H. Freeman, New York, 2000).
- ⁶S. A. Kauffman, "Metabolic stability and epigenesis in randomly connected nets," *J. Theor. Biol.* **22**, 437–467 (1969).
- ⁷D. Bray, "Protein molecules as computational elements in living cells," *Nature (London)* **376**, 307–312 (1995).
- ⁸H. H. McAdams and A. Arkin, "Simulation of prokaryotic genetic circuits," *Annu. Rev. Biophys. Biomol. Struct.* **27**, 199–224 (1998).
- ⁹T. S. Gardner, C. R. Cantor, and J. J. Collins, "Construction of a genetic toggle switch in *E. coli*," *Nature (London)* **403**, 339–342 (2000).
- ¹⁰M. B. Elowitz and S. Leibler, "A synthetic oscillatory network of transcriptional regulators," *Nature (London)* **403**, 335–338 (2000).
- ¹¹A. Arkin and J. Ross, "Computational functions in biochemical reaction networks," *Biophys. J.* **67**, 560–578 (1994).
- ¹²H. De Jong, "Modeling and simulation of genetic regulatory systems: A literature review," *J. Comput. Biol.* **9**, 67–103 (2002).
- ¹³T. Mestl, E. Plahte, and S. W. Omholt, "A mathematical framework for describing and analyzing gene regulatory networks," *J. Theor. Biol.* **176**, 291–300 (1995).
- ¹⁴L. Glass, "Combinatorial and topological method in chemical kinetics," *J. Chem. Phys.* **63**, 1325–1335 (1975).
- ¹⁵C. Papadimitriou, *Computational Complexity* (Addison-Wesley, Reading, MA, 1995).
- ¹⁶H. T. Siegelmann, *Neural Networks and Analog Computation: Beyond the Turing Limit* (Birkhauser, Boston, 1999).
- ¹⁷T. Mestl, R. J. Bagley, and L. Glass, "Common chaos in arbitrarily complex feedback networks," *Phys. Rev. Lett.* **79**, 653–656 (1997).
- ¹⁸L. Glass and C. Hill, "Ordered and disordered dynamics in random networks," *Europhys. Lett.* **41**, 599–604 (1998).
- ¹⁹H. T. Siegelmann, A. Roitershtein, and A. Ben-Hur, "Noisy neural networks and generalizations," *Proceedings of the Annual Conference on Neural Information Processing Systems 1999 (NIPS*99)* (MIT Press, Cambridge, MA, 2000).
- ²⁰W. Maass and P. Orponen, "On the effect of analog noise in discrete time computation," *Neural Comput.* **10**, 1071–1095 (1998).
- ²¹L. F. Landweber and L. Kari, "The evolution of cellular computing: Nature's solution to a computational problem," *Proceedings of the 4th DIMACS Meeting on DNA Based Computers*, 1998, pp. 3–15.
- ²²L. Glass and J. S. Pasternack, "Stable oscillations in mathematical models of biological control systems," *J. Math. Biol.* **6**, 207–223 (1978).
- ²³R. N. Tchuraev, "A new method for the analysis of the dynamics of the molecular genetic control systems. I. Description of the method of generalized threshold models," *J. Theor. Biol.* **151**, 71–87 (1991).
- ²⁴M. S. Branicky, "Analog computation with continuous ODEs," *Proceedings of the IEEE Workshop on Physics and Computation*, Dallas, TX, 1994, pp. 265–274.
- ²⁵E. Asarin, O. Maler, and A. Pnueli, "Reachability analysis of dynamical systems with piecewise-constant derivatives," *Theor. Comput. Sci.* **138**, 35–66 (1995).
- ²⁶A. Saito and K. Kaneko, "Geometry of undecidable systems," *Prog. Theor. Phys.* **99**, 885–890 (1998).
- ²⁷J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, New York, 1979).
- ²⁸P. Koiran and C. Moore, "Closed-form analytic maps in one and two dimensions can simulate universal Turing machines," *Theor. Comput. Sci.* **210**, 217–223 (1999).
- ²⁹J. E. Lewis and L. Glass, "Nonlinear dynamics and symbolic dynamics of neural networks," *Neural Comput.* **4**, 621–642 (1992).
- ³⁰P. Orponen, "The computational power of discrete Hopfield nets with hidden units," *Neural Comput.* **8**, 403–415 (1996).
- ³¹P. Orponen, "Computing with truly asynchronous threshold logic networks," *Theor. Comput. Sci.* **174**, 97–121 (1997).
- ³²C. Moore, "Generalized one-sided shifts and maps of the interval," *Nonlinearity* **4**, 727–745 (1991).
- ³³J. P. Crutchfield and K. Young, "Computation at the onset of chaos," in *Complexity, Entropy and the Physics of Information*, edited by W. H. Zurek (Addison-Wesley, Redwood, CA, 1990).
- ³⁴C. Moore, "Queues, stacks, and transcendental at the transition to chaos," *Physica D* **135**, 24–40 (2000).
- ³⁵R. Edwards, H. T. Siegelmann, K. Aziza, and L. Glass, "Symbolic dynamics and computation in model gene networks," *Chaos* **11**, 160 (2001).
- ³⁶J. S. Mattick, "Non-coding RNAs: The architects of eukaryotic complexity," *EMBO J.* **2**, 986–991 (2001).
- ³⁷J. 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.