Simulating heterogeneous populations using Boolean models

Brian C. Ross[1,2], Mayla Boguslav[1,2], Holly Weeks[3], James Costello[1,2*]

[1]Computational Bioscience Program,

[2]Department of Pharmacology,

[3]Department of Biostatistics and Informatics,

University of Colorado Anschutz Medical Campus, Aurora, CO, USA

Abstract

Certain biological processes such as cancer development and immune activation are controlled by rare cellular events that are difficult to capture computationally through simulations of individual cells. Here we show that when cellular states are described using a Boolean network model, one can exactly simulate the dynamics of non-interacting, highly heterogeneous populations directly, without having to model the various subpopulations. This strategy captures even the rarest outcomes of the model with no sampling error. Our method can incorporate heterogeneity in both cell state and, by augmenting the model, the underlying rules of the network as well (i.e. mutations). We demonstrate our method by using it to simulate a heterogeneous population of Boolean networks modeling the T-cell receptor, spanning $\sim 10^{20}$ distinct cellular states and mutational profiles.

Computer models are widely used to predict behaviors of biological systems, generally by simulating a number of instances of that model and enumerating the range of observed outcomes. Comprehensive simulations of single cells may be realistic given recent progress in constructing elaborate cellular models, but simulation of an entire tissue is far more difficult owing to the vast number of cells, cell types, and their interactions.

As a step towards tissue-scale modeling of cells, we consider the problem of simulating large and heterogeneous populations of *non-interacting* cells. That is, we wish to model the wide variety of cellular states and dynamics experienced by a heterogeneous population of cells in a tissue living in isolation from each other. The output of the approach we propose will be the frequency with which certain events happen over time in a large population. In theory, this result could be obtained by averaging a large number of traditional single-cell simulations spanning the entire population, but in extremely heterogeneous populations it becomes unfeasible to simulate each distinct subpopulation, in which case the traditional recourse is to estimate the population statistics by Monte Carlo (random sampling) [1]. By design, the basic random sampling procedure captures typical outcomes of these simulations, and only rarely finds unusual occurrences. Yet some biological processes are determined by outliers [2], such as rare tumor cells [3–5] or immune clones [6]. We acknowledge that if something is known about the circumstances leading to a rare outcome, one can bias Monte Carlo to oversample that outcome and then correct for the biased sampling (a strategy known as importance sampling [7]), but oversampling inevitably introduces sampling biases.

Here we propose an alternative, exact method for simulating heterogeneous populations, which takes advantage of the observation that discrete models have a finite set of possible states. For these models, one can write the instantaneous state of some individual being modeled using a vector $\mathbf{b}_{(\alpha)}$ of all possible states, having a '1' in the state α of the individual and a 0 everywhere else. Assuming deterministic dynamics, the time evolution of this individual can then be written as a linear (matrix) operator F_b, and time evolution proceeds by repeated matrix multiplications of the state vector: $\mathbf{b}_{(\alpha)}(t) = F_b^{t-t_0}\mathbf{b}_{(\alpha)}(t_0)$ where t_0 and t are integer time steps. This approach is always possible in principle for discrete systems, even when it is too computationally expensive to be feasible in practice.

The potential usefulness of a linear representation is that the same equations that simulate an individual *automatically generalize to simulating arbitrary mixed populations* of individuals. The basic idea, as illustrated in Figure 1, is that a population-averaged vector $\langle \mathbf{b} \rangle = \sum_\alpha w_\alpha \mathbf{b}_{(\alpha)}$ evolves according to the same time-evolution operator F_b as a vector representing an individual, owing to the superposition property of linear systems. We will exploit this fact in deriving a time-evolution opera-

*E-mail: james.costello@ucdenver.edu

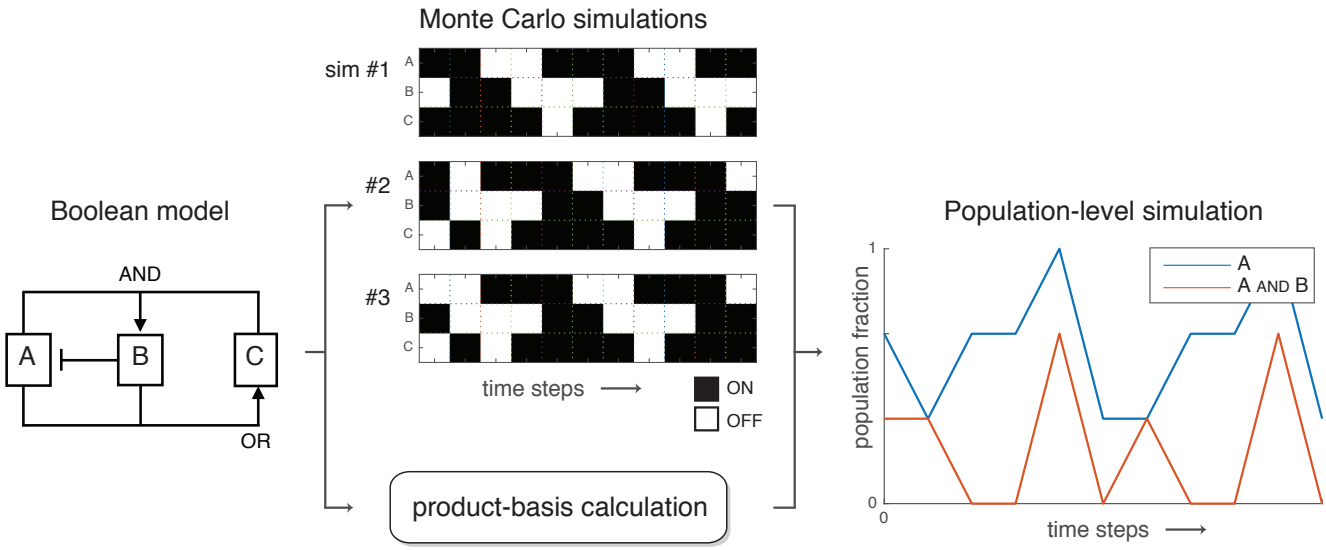

Figure 1. Schematic of a population-level simulation, showing how the population fractions showing activation of variable A and variable A AND B evolve over time when the individuals in the population have heterogeneous states. No information about the substructure of the population is lost in the averaging process when one takes into account higher-order correlations (such as A AND B).

tor using an algebra tailored for an individual, and then repurpose that operator to simulate mixed populations.

The obvious drawback of the linear method of time evolution is the typically huge number of states in a given system, causing both the state vector $\mathbf{b}$ (or $\langle\mathbf{b}\rangle$) and the time evolution operator F to be unfeasibly large. For example, Boolean networks are a class of simple models built entirely from ON/OFF variables, yet even these models have an exponential number of states (2^N for N Boolean variables in the model). In general a linear representation of the dynamics is typically only feasible if one looks at only a small subspace of the full linear space – i.e. considers only a small subset of the possible set of states of the model. However, this strategy is incompatible with our interest in simulating massively heterogeneous populations: a simulation involving n_s subpopulations necessarily involves at least n_s nonzero entries in both the state vector $\langle\mathbf{b}\rangle$ and in the time-evolution matrix F. Under this proportional scaling with heterogeneity, a linear representation of the system offers no computational improvement over simple enumerative simulation of each individual subpopulation.

Our proposed solution to the problem of proportional scaling with heterogeneity is to *change linear basis* from the state basis $\mathbf{b}$ to a new basis $\mathbf{x}$ in which the size of the *subspace of interest* scales independently of the heterogeneity of the mixed population. (The full spaces of $\mathbf{b}$ and $\mathbf{x}$ are necessarily of the same size.) Specifically, the goal of this paper is to introduce a linear basis for Boolean variables that we term a 'product basis', and give a prescription for calculating the time-evolution operator in this basis. Although our method only applies to Boolean network models, within the Boolean framework the method is very general, applying to deterministic Boolean models [8], as well as probabilistic [9] and continuous-time [10] Boolean models in the large-population limit. Note that although we assume a synchronous updating rule for all variables in the model, this is without loss of generality because an asynchronous network can be modeled as a probabilistic network with synchronous time steps [11].

By way of comparison, we note the significant advances that have been made in analyzing the possible long-term outcomes, or *attractors* (steady states and limit cycles), of Boolean models [8, 12–19]. Attractors have been found using network-reduction algorithms that find simple networks encoding the long-term behavior of more complex networks [12, 14, 20], methods that solve steady states as zeros of a polynomial equation [21], SAT methods [16, 17, 22], and binary decision diagrams [18, 19, 23] (the introduction of Ref. [13] gives a good review of these techniques). These techniques differ from our proposed method in that they focus on possible long-term behaviors, whereas ours gives an explicit population-averaged simulation of a defined starting population. While our method does have the ability to generate long-term dynamical equations that can be used to find attractors (see Appendix 2), our method differs in that it finds the attractors of a set of characteristics of interest, not the at-

tractors of the complete state.

We first derive the procedure for calculating a product-basis simulation to track transient and long term model behaviors, and give an example calculation illustrating the mathematics. Next, we demonstrate an application of our method by using it to simulate a large heterogeneous population using a published T-cell network [24].

Methods

Consider a Boolean network model consisting of N variables, which updates its state at each time step using a deterministic update rule. We denote the Boolean model variables using Roman letters, e.g. $i, j, \ldots$, and use Greek letters to refer to sets of Boolean model variables. For each possible subset of Boolean variables $\alpha = \{i, j, \ldots\}$, there is a unique state-basis variable b_α (which we sometimes write as $b_{ij\ldots}$) and a unique product-basis variable $x_\alpha = x_{ij\ldots}$. We consider the state-basis variables to be the components of a vector $\mathbf{b}$, and the product-basis variables to be components of vector $\mathbf{x}$. There are 2^N subsets of all N Boolean variables (including both θ and the empty subset denoted $\emptyset$), so both $\mathbf{b}$ and $\mathbf{x}$ are 2^N-length vectors.

Definition 1. *Consider a Boolean model composed of some set θ of Boolean variables, where B_i represents the state of Boolean variable $i \in \theta$ and follows the convention $B_i = 1$ for an* ON *variable and $B_i = 0$ for an* OFF *variable. If κ is the set of* ON *variables for some individual, then the values of the state-space and product-space variables describing that individual, indexed by Boolean subset α, are defined by:*

$$b_\alpha = \left(\prod_{i\in\alpha} B_i\right) \cdot \left(\prod_{i\in(\theta\setminus\alpha)} (1 - B_i)\right) = \begin{cases} 1 & \text{if } \alpha = \kappa \\ 0 & \text{otherwise} \end{cases}$$

$$x_\alpha = \prod_{i\in\alpha} B_i = \begin{cases} 1 & \text{if } \alpha \subseteq \kappa \\ 0 & \text{otherwise.} \end{cases}$$

By common convention the empty product $x_\emptyset$ equals 1. The fact that each x_α is a simple product of the values of its subscripted Boolean variables motivates the terminology 'product basis'. As shown in Appendix 1, a product-basis vector $\mathbf{x}$ is a fully equivalent representation of a mixed population to the state-basis vector $\mathbf{b}$.

Simulations of mixed populations

We build our simulations by selecting a set of product-basis variables of interest (generally a subset of the full state vector $\mathbf{x}$), and associating with each variable x_α an update rule f_α so that $x_\alpha(t+1) = f_\alpha(\mathbf{x}(t))$. (The exception is the case of continuous-time Boolean networks in which $f_\alpha = dx_\alpha/dt$, but we will treat those separately later.) We construct the simulation in two steps. The first step is to build the single-index update rules f_i (i.e. $\alpha = \{i\}$) for individual Boolean variable i, by enumeration of its input states. The second step is to build necessary higher-order update rules $f_{ij\ldots}$ as needed until the system of equations closes (i.e. until we have solved for each f_α corresponding to a variable x_α appearing in the right-hand side of some other f_β). As a necessary preliminary, we first show how to build the change-of-basis operator T that converts $\mathbf{b}$-basis vectors to $\mathbf{x}$-basis vectors.

Algorithm 1 (Constructing a change-of-basis matrix)**.** *Assume some set of n Boolean variables θ for which κ and α denote subsets. We can construct the change-of-basis matrix T_n that converts a 2^n-length $\mathbf{b}$-vector indexed by κ to a 2^n-length $\mathbf{x}$-vector indexed by α by assigning a 1 to each element $T_{n(\alpha\kappa)}$ for which $\alpha \subseteq \kappa$, and 0 to all other elements.*

Proof. Consider the state vector $\mathbf{b}_{(\kappa)}$ whose entries are all 0 except for a 1 in position κ, which describes an individual in state κ. The product-space representation $\mathbf{x}$ of this individual is found by multiplying $x_\alpha = \sum_\gamma T_{\alpha\gamma} b_\gamma = T_{\alpha\kappa}$: thus $\mathbf{x}$ is simply column κ of the change-of-basis matrix. Using Definition 1, the value of product-basis variable x_α, corresponding to $T_{\alpha\kappa}$, is 1 if $\alpha \subseteq \kappa$ and 0 otherwise. □

We can now give a procedure for calculating the single-variable update rules f_i in terms of its $N^{[i]}$ Boolean inputs $\theta^{[i]}$. To do so we will work in a reduced space where $\mathbf{b}^{[i]} = \{b_\rho | \rho \subseteq \theta^{[i]}\}$ and $\mathbf{x}^{[i]} = \{x_\rho | \rho \subseteq \theta^{[i]}\}$, as $N^{[i]}$ is generally small enough that we can explicitly write the change-of-basis operator $T_{N^{[i]}}$ in this space using Algorithm 1.

Algorithm 2 (Computing f_i)**.** *Define $\mathbf{k}^{[i]}$ as the vector such that $k_\alpha^{[i]}$ is 1 if Boolean input pattern $b_\alpha^{[i]}$ produces a 1 in output Boolean variable i, and 0 otherwise. Then $f_i = (\mathbf{k}^{[i]T} T_{N^{[i]}}^{-1})\mathbf{x}^{[i]}$, which is a linear equation in $\mathbf{x}^{[i]} \subseteq \mathbf{x}$.*

Proof. By definition $f_i = \mathbf{k}^{[i]T}\mathbf{b}^{[i]}$, as this expression reproduces the output rule. Using the fact that $T_{N^{[i]}}$ is invertible (proved in Appendix 1), we write $f_i = \mathbf{k}^{[i]T} T_{N^{[i]}}^{-1} T_{N^{[i]}} \mathbf{b}^{[i]}$ and identify $T_{N^{[i]}}\mathbf{b}^{[i]} = \mathbf{x}^{[i]}$, which proves the formula. □

Using the set of single-index f_i, one can compute the linear time-evolution function of any multi-index product-basis variable of interest $f_{ij\ldots}$ using the following method.

Algorithm 3 (Computing $f_{ij\ldots}$)**.** *We first compute f_α for $\alpha = \{i, j, \ldots\}$ as $f_\alpha \leftarrow f_i(\mathbf{x}) \cdot f_j(\mathbf{x}) \cdot \ldots$ (expressed in terms of $\mathbf{x}$-basis variables). Next we distribute each term inside the product, so that f_α is a weighted sum of products of $\mathbf{x}$-basis variables, and replace each nonlinear product of terms $x_\beta \cdot x_\gamma \cdot \ldots$ appearing inside f_α with the product-basis variable x_μ where $\mu = \beta \cup \gamma \cup \ldots$. This gives an expression for f_α that is linear in $\mathbf{x}$.*

Proof. First we show that $f_\alpha = f_{ij\ldots}$ equals the product $f_i \cdot f_j \cdot \ldots$.

$$\begin{aligned} f_{ij\ldots}(t) &= x_{ij\ldots}(t+1) \\ &= x_i(t+1) \cdot x_j(t+1) \cdot \ldots \\ &= f_i(t) \cdot f_j(t) \cdot \ldots \end{aligned}$$

Since this always holds true irrespective of the time step t, we simply calculate $f_{ij\ldots} = f_i \cdot f_j \cdot \ldots$.

The second step is to show that each $x_\beta \cdot x_\gamma \cdot \ldots$ equals $x_{\beta\cup\gamma\cup\ldots}$. Let $k, l, \ldots$ be the elements of $\mu = \beta \cup \gamma \cup \ldots$. Then $x_\beta \cdot x_\gamma \cdot \cdots = x_k^{p_k} \cdot x_l^{p_l} \cdot \ldots$ where $p_k, p_l, \cdots \geq 1$ are the respective number of times $k, l, \ldots$ appear in $\beta, \gamma, \ldots$. Since all x_i are Boolean variables, we have $x_i^p = x_i$ for any $p \geq 1$. Thus $x_\beta \cdot x_\gamma \cdot \cdots = x_k \cdot x_l \cdot \cdots = x_\mu$. □

Using Algorithms 2 and 3, we can give the full procedure for building a simulation that time-evolves any arbitrary set of product-basis variables of interest describing some individual modeled by the Boolean rules. We denote the set of variables of interest as Ω_0. Our algorithm constructs f_α for each $\alpha \in \Omega_0$, then additional f_β to time-evolve each x_β appearing in the right-hand side of the f_α, etc. until the equations form a closed system.

Algorithm 4 (Building a product-basis simulation)**.** *First we compute the full set of single-index f_i using Algorithm 2. Next we initialize the total set of required product-basis variables $\Omega \leftarrow \Omega_0$, and the set of product-basis variables whose dynamics we have solved $\Omega_S \leftarrow \emptyset$. Finally we iteratively solve f_α for each $\alpha \in \Omega \backslash \Omega_S$ using Algorithm 3, while updating $\Omega_S \leftarrow \Omega_S \cup \{\alpha\}$ and $\Omega \leftarrow \Omega \cup \{\mu\}$ for each variable x_μ appearing on the right-hand side of f_α. Iteration continues until $\Omega_S = \Omega$.*

Proof. We first note that Ω eventually converges to a finite set owing to the fact that the full variable space θ is a finite set, and that $\Omega \subseteq \theta$ only gains, never loses, elements of θ at each iterative step. Since a) $\Omega_S \subseteq \Omega$, b) Ω_S accumulates 1 term in $\Omega \backslash \Omega_S$ at each iterative step, and c) Ω converges, the algorithm always ends in a finite number of steps. When the algorithm ends the set of solved equations Ω_S contains the set of variables appearing in the right-hand side of those equations $\Omega_R \subseteq \Omega = \Omega_S$, so the resulting system of equations is closed. □

Writing our final closed system of linear equations as a square matrix F, we have a very simple update rule for simulations using the product-basis variables: $\mathbf{x}(t+1) = F \cdot \mathbf{x}(t)$.

Our final step is to generalize to a mixed-population simulation.

Definition 2. *Define a population-level state vector $\langle \mathbf{x} \rangle$ as a population-weighted linear combination of the state vectors of the subpopulations $\mathbf{x}_{(\alpha)}$:*

$$\langle \mathbf{x} \rangle (t) := \sum_\alpha w_\alpha \mathbf{x}_{(\alpha)}(t) \tag{0.1}$$

where the weighting factors w_α are proportional to the occurrence of subpopulations α in the overall mixed population.

Note that we are free to choose the normalization constant $n = \sum_\alpha w_\alpha$. In all our examples we will choose $n = 1$, giving w_α the interpretation as the population fraction of state α. Irrespective of the choice of n, this representation can be used to time-evolve mixed populations using our existing operators.

Claim 1. *Each element $\langle x_\alpha \rangle$ of vector $\langle \mathbf{x} \rangle$ is proportional to the overall occurrence of characteristic α in the overall mixed population. For $n = 1$, $\langle x_\alpha \rangle$ is the average occurrence of that trait in the mixed population.*

Proof. Equation 0.1 is the definition of a $\mathbf{w}$-weighted average of characteristics $\mathbf{x}$, and $\mathbf{w}$ is proportional to the population fraction. □

Claim 2. *Any linear time-evolution operator that properly time-evolves an individual in any arbitrary state $\mathbf{x}_\alpha$ also correctly time-evolves any arbitrary mixed population of individuals $\langle \mathbf{x} \rangle$.*

Proof. This follows from the fact that F commutes with the sum over subpopulations.

$$\begin{aligned} \langle \mathbf{x} \rangle (t) &= \sum_\alpha w_\alpha F^{t-t_0} \mathbf{x}_{(\alpha)}(t_0) \\ &= F^{t-t_0} \sum_\alpha w_\alpha \mathbf{x}_{(\alpha)}(t_0) \\ &= F^{t-t_0} \langle \mathbf{x} \rangle (t_0). \end{aligned} \tag{0.2}$$

□

Thus the time-evolution operator produced by Algorithm 4 correctly time-evolves the mean occurrence of the characteristics $\langle \mathbf{x} \rangle$ in any mixed population, *despite the fact that this operator was derived for an $\mathbf{x}$ describing an individual* (notably in assuming that each x_α is Boolean).

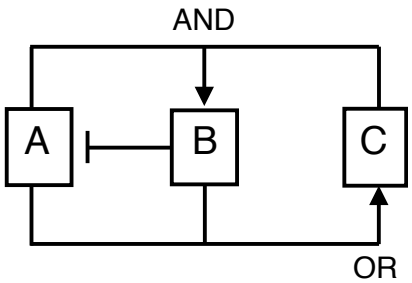

Figure 2. The 3-Boolean network used in Example 1. Arrows indicate the update rules for each variable: for example if either A or B is ON at time t then C will be ON at time $t+1$; otherwise C will be OFF.

Example 1: building equations

Suppose we wish to follow the time evolution of variable A in the model shown in Figure 2. The first step is to compute the change-of-basis matrices that will be used to compute the single-variable update rules f_A, f_B and f_C. Variable A takes input from the single variable B, so calculating f_A requires the change-of-basis matrix T_1. Ordering the elements by the subscripts $(\emptyset, B)$ respectively, and applying Algorithm 1, we obtain

$$T_1 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Variables B and C each take input from two variables. Ordering B's input states $(\emptyset, A, C, AC)$, and C's input states $(\emptyset, A, B, AB)$, we find that both f_B and f_C are computed using

$$T_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Next we build the single-index update functions. Variable A takes input only from variable B, so the possible patterns of active inputs are $(\emptyset, B)$, corresponding to the state-basis variables $(b_\emptyset^{[A]}, b_B^{[A]})$. The respective outputs are $(1,0) = \mathbf{k}^{[A]T}$ due to the NOT gate, from which we can immediately calculate f_A using Algorithm 2.

$$\begin{aligned} f_A &= \left(\mathbf{k}^{[A]T} T_1^{-1}\right) \begin{bmatrix} x_\emptyset \\ x_B \end{bmatrix} \\ &= x_\emptyset - x_B. \end{aligned} \tag{1.1}$$

In the same way we find that the input patterns $(\emptyset, b_A^{[B]}, b_C^{[B]}, b_{AC}^{[B]})$ to variable B lead to outputs $\mathbf{k}^{[B]T} = (0,0,0,1)$, and the input patterns $(\emptyset, b_A^{[B]}, b_B^{[B]}, b_{AB}^{[B]})$ to variable C lead to outputs $\mathbf{k}^{[C]T} = (0,1,1,1)$. From these and the 2-variable change-of-basis matrix T_2 we calculate

$$\begin{aligned} f_B &= x_{AC} \\ f_C &= x_A + x_B - x_{AB}. \end{aligned} \tag{1.2}$$

Having built the single-index update functions, we can now derive a linear system that tracks the time evolution of variable A. The immediate equation for this purpose is f_A which we already derived (Equation 1.1), but since it involves x_B and $x_\emptyset$ our simulation must also track those variables through time using Equation 1.2 along with

$$f_\emptyset = x_\emptyset. \tag{1.3}$$

Equation 1.2 requires that we track a new multi-index variable x_{AC}, requiring us to solve f_{AC} using Algorithm 3.

$$\begin{aligned} f_{AC} &= (1 - x_B) \cdot (x_A + x_B - x_{AB}) \\ &= x_A + x_B - x_{AB} - x_{AB} - x_B + x_{AB} \\ &= x_A - x_{AB} \end{aligned} \tag{1.4}$$

We continue the process of identifying new variables and solving for their update functions until the equations form a closed system.

$$\begin{aligned} f_{AB} &= (1 - x_B) \cdot (x_{AC}) \\ &= x_{AC} - x_{ABC} \end{aligned} \tag{1.5}$$

$$\begin{aligned} f_{ABC} &= (1 - x_B) \cdot (x_{AC}) \cdot (x_A + x_B - x_{AB}) \\ &= x_{AC} - x_{ABC} \end{aligned} \tag{1.6}$$

Equations 1.1-1.6 together with an initial state in $(x_A, x_B, x_{AC}, x_{AB}, x_{ABC})$ describe the time evolution of these quantities in a single Boolean network as a sequence of 0s and 1s in each variable. The final step is to reinterpret these equations as describing the dynamics of a mixed population of networks, formally by taking the mean of both sides of each equation.

$$f_{\langle A\rangle} = \langle x_\emptyset\rangle - \langle x_B\rangle \tag{1.1b}$$
$$f_{\langle B\rangle} = \langle x_{AC}\rangle \tag{1.2b}$$
$$f_{\langle \emptyset\rangle} = \langle x_\emptyset\rangle \tag{1.3b}$$
$$f_{\langle AC\rangle} = \langle x_A\rangle - \langle x_{AB}\rangle \tag{1.4b}$$
$$f_{\langle AB\rangle} = \langle x_{AC}\rangle - \langle x_{ABC}\rangle \tag{1.5b}$$
$$f_{\langle ABC\rangle} = \langle x_{AC}\rangle - \langle x_{ABC}\rangle \tag{1.6b}$$

The angle-brackets denote an average, and we have used the notation $\langle x_i(t+1)\rangle = f_{\langle i\rangle}$. Per Claim 2 the linear equations are unaffected by the averaging process, so the same equations used to derive the dynamics of a single instance of a model also describe the mean values of those same variables in a mixed population. Whereas the product variables take on binary values for an individual, the population-averaged variables are real-valued on the interval $[0, 1]$ (using our normalization): for example we would set $\langle x_A\rangle = 0.4$ if 40% of the population has gene A set ON.

Probabilistic and asynchronous Boolean networks Our method can be applied to probabilistic Boolean networks (PBNs) [9, 25], in which several state transitions are possible at each time step with different probabilities. As we will show, our algorithm works as given in the *large-population limit* in which time evolution of the average state $\langle \mathbf{x}\rangle$ becomes deterministic, despite the fact that each individual PBN is stochastic.

Applying our method to PBNs requires that we reinterpret the meaning of the time-evolution equations. For an individual we write:

$$p(x(t+1)) = F \cdot \mathbf{x}(t) \tag{1.7}$$

where $p(\cdot)$ denotes the likelihood of an outcome. With this new interpretation of the time-evolution operator F, our method still works, although we note here several changes to the logic. 1) In Algorithm 2 we generalize each $\mathbf{k}^{[i]}$ to be a vector of likelihoods that each respective input pattern produces a 1 in the output variable, so that $f_i = \mathbf{k}^{[i]T}\mathbf{b}^{[i]}$ as before. 2) The multiplication rule, now reading $p_{x_{ij\ldots}} = p(x_i)p(x_j)\ldots$, used by Algorithm 3 still holds, owing to the independence of outcomes $p(x_i)$, $p(x_j)$, etc. 3) We point out that although $p(\mathbf{x})$ is real-valued, the state vector of an individual $\mathbf{x}$ is still binary, so $x_\alpha^{p\geq 1} = x_\alpha$ as before (Algorithm 3).

Despite the fact that our algorithm produces valid product-basis equations of the form 1.7, the resulting linear system of equations does not form a closed system, simply because the left-hand side uses different variables than the right ($p(\mathbf{x})$ versus $\mathbf{x}$). Our method therefore cannot be used to simulate an individual instance of a PBN. However, by averaging both sides and taking the large-population limit so that $p(\mathbf{x}) \to \langle \mathbf{x}\rangle$, the system closes and reproduces Equation 0.2. Thus our system of equations accurately tracks the deterministic dynamics of arbitrary mixed populations of PBNs in the infinite-population limit, despite being unable to model stochastic individuals.

Large populations of asynchronous networks behave identically to large populations of PBNs [11] if we define a uniform time step: the likelihoods of the various possible updates over that time step give the state-transition weights in the corresponding synchronous PBN. If this time step is small enough, then the likelihood of two causally-connected asynchronous updates happening in the same step is small, and in this limit the local update rules for a PBN accurately model the asynchronous network. Therefore our analysis also applies to large populations of asynchronous networks for small time steps.

Continuous-time Boolean networks Probabilistic networks give one way to incorporate rate information into our model; another way is to work in continuous time using differential equations [10]: $f_A = dx_A/dt$. The differential form does require one change in our method: the rate of change of a higher-order variable is found by using the product rule of derivatives. Whereas under a discrete update $f_{ABC\ldots}$ is the product $f_A \cdot f_B \cdot f_C \cdot \ldots$, for the differential case we compute:

$$\begin{aligned} f_{ABC\ldots} &= \frac{d}{dt}\left(x_A x_B x_C \ldots\right) \\ &= x_B x_C \ldots f_A + x_A x_C \ldots f_B + \ldots . \end{aligned} \tag{1.8}$$

Also, under discrete updates the trivial function is $f_\emptyset = 1$ but with differential updates it is $f_\emptyset = 0$.

Results

Product-basis equations recapitulate Monte Carlo simulations We tested the product-basis method using 10^4 randomly generated 10-node deterministic networks, where each node's input combined the output of 1-4 randomly-selected nodes using randomly-generated logic rules. For each network, we ran 100 Monte Carlo individual simulations using a random ensemble of initial states (equal probability for all states), and compared the population average of certain state variables with a product-basis simulation using the same starting population distribution. In each case, the product-basis simulation reproduced the average of the Monte Carlo simulations. Next, we ran 10^4 tests of probabilistic networks

(PBNs), again using 100 Monte Carlo runs per test. In order to generate realistic PBNs, we augmented the original time-evolution functions f_i^0 with random rate parameters r_i, leading to the equations $f_i \leftarrow (1-r_i)x_i + r_i f_i^0$ for $0 \leq r_i \leq 1$. Again the equations of the product-basis method reproduced the results of Monte Carlo simulations, this time to within sampling error ($\sim 1/\sqrt{n_{runs}}$). Finally, we tested 2.5×10^3 continuous-time networks generated using the same rule (which are slower to check by Monte Carlo), and again found complete agreement with Monte Carlo.

Product-basis method can simulate highly heterogeneous populations beyond the scope of Monte Carlo To demonstrate the product-basis method on very heterogeneous populations, we applied it to the T-cell activation network described in Ref. [24] (see Figure 10 and Table 2 of that paper). The T-cell network is a deterministic, 40-node network with fifty-four edges, multiple feedback loops, and whose attractors include both steady states and limit cycles. For demonstration purposes, we show a traditional Monte Carlo simulation of an *individual* T-cell network in Figure 3A, obtained by choosing an initial Boolean state and straightforwardly applying the model rules over each successive time step. This particular simulation shows transient (nonrepeating) behavior for the first 10 time steps, leading to a limit cycle with a repeating period of 6 time steps.

Next, we performed a population-level simulation using the product-basis method. We first provided a target set of variables to follow in time, which the product-basis algorithm used to generate a set of time-evolution equations involving those variables (along with other variables that were added automatically to close the system of equations). We chose to track three variables: cyclic-AMP response element (CRE) mediated gene activation, the AP1 (Activating protein 1) transcription factor, and their co-occurrence which we label CRE AND AP1. The co-occurrence variable was not redundant: although CRE AND AP1 is simply the product CRE $\times$ AP1 for an individual (because the variables are Boolean), this is *not* true at the population level. (For example if the levels of CRE and AP1 are both 0.5, then the level of CRE AND AP1 could equal 0.25 if CRE and AP1 states are uncorrelated in the population, 0 if they are perfectly anticorrelated, 0.5 if they are perfectly correlated, or any other value in between.)

We first generated the product-basis time-evolution equations for our three variables of interest. Next, we set the model variables to an initial state representing a mixed population, and used the time-evolution equations to track the population-level average of each of the three variables for 50 time steps (Figure 3B). We stress that this is an exact result, with no sampling error. The starting population we considered was a uniform mixture of all possible 2^{40} initial states of the Boolean network, but because 7 of these variables are not upstream of either CRE or AP1, we consider our effective initial population size to be $2^{33} \approx 10^{10}$. At this level of diversity, it would require extensive computational resources to reproduce our exact result by exhaustive enumeration over the initial states.

Next, we demonstrated the ability of the product-basis method to analyze mutations in the network by including the full set of possible gene knock-outs in the T-cell activation network. We did this by adding a set of 'wild-type' variables to the network, one for each original variable in the system, and included the wild-type variables in the update rules using an AND operation. For example, an update rule reading $[A \leftarrow B \text{ OR } C]$ became $[A \leftarrow (B \text{ OR } C) \text{ AND } A^{\mathrm{WT}}]$. Note that the presence of mutations effectively doubles the size of the network and thus vastly increases the heterogeneity, which is determined by both the number of activation states of the original variables and the number of mutational profiles, in total spanning the order of 10^{20} different subpopulations. Enumeration over the initial states is impossible at this level of diversity, and the traditional solution is a sparse sampling method such as Monte Carlo, although that loses the ability to resolve very rare subpopulations. Despite this heterogeneity, our product-basis method was able to produce the exact time-evolution equations for CRE AND AP1. We chose an initial population that contained each possible combination of knock-out mutations at a 0.01% mutation rate per variable (roughly the highest possible rate of homozygous knockouts given a 1%-per-gene-locus mutation rate [26]) superimposed upon a uniform mixture of each activation state (adjusted so that mutated genes always began OFF). From this initial state, we again followed the exact time course of the CRE AND AP1 population fraction, and compared it to our original wild-type result (Figure 3C). Notably, a small fraction of the population reached a steady state showing both CRE activation and AP1 expression. We also validated the result (to within statistical error) using Monte Carlo, although as shown in the figure, Monte Carlo was only useful for comparing the transient behavior, not the rare persistent subpopulations which fell far below its sampling resolution.

Finally, we examined the mutations leading to CRE AND AP1 coexpression. We hypothesized that this was due to knockout of the cCbl gene in the recurrent core of this network, and tested this hypothesis by generating the time course of the three-way co-occurrence of CRE AND AP1 AND WT-cCbl, where WT denotes the respective wild-type variable. This final time series dropped to zero at steady state, indicating that mutations in this variable are necessary for persistent CRE AND AP1 coexpression (see Fig. 3D), and that no other set of knockout muta-

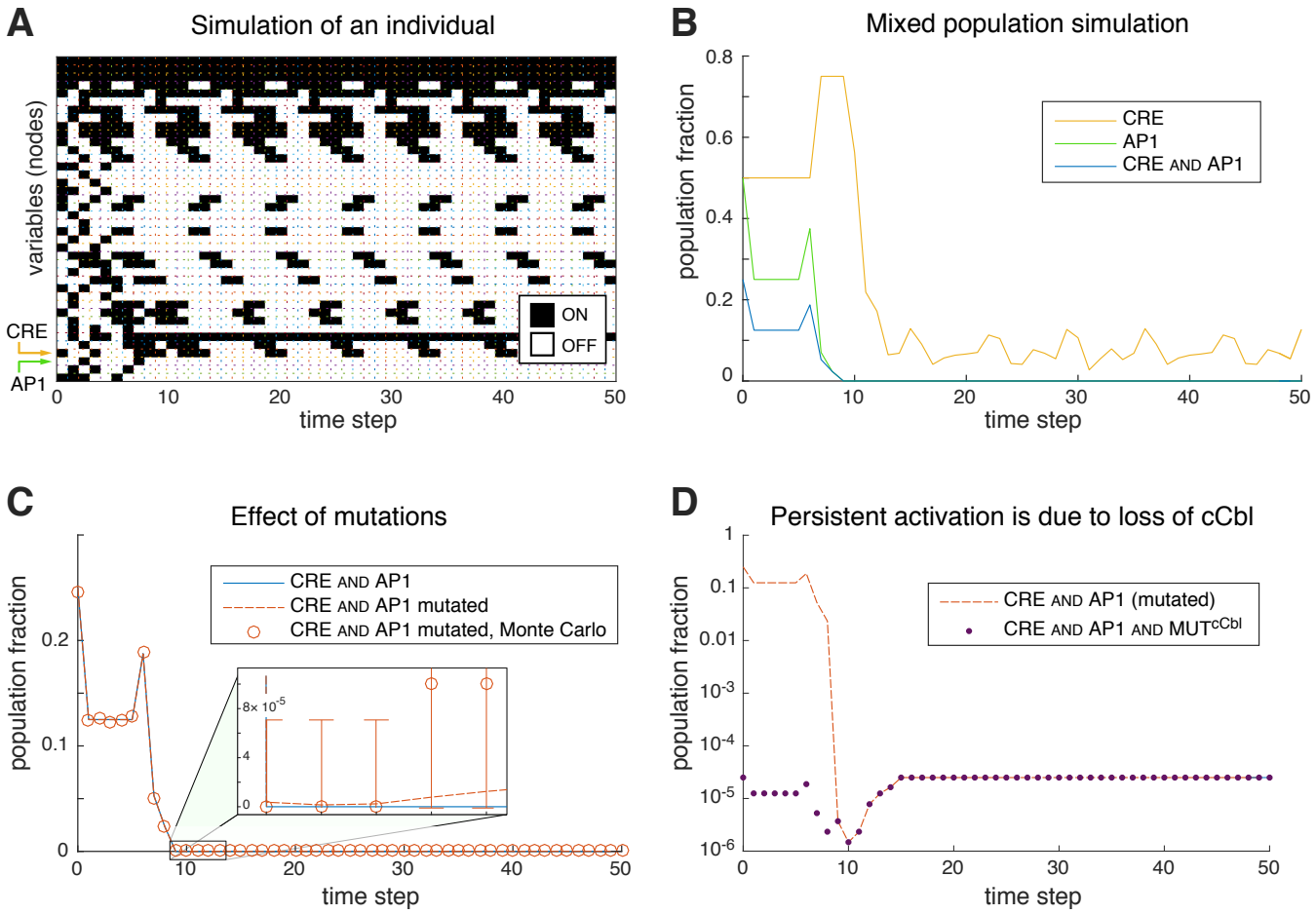

Figure 3. A) Time evolution of one instance of the T-cell network [24], starting from a random initial state. White/black rectangles signify OFF/ON Boolean states. **B)** Time evolution of the population fraction having activated CRE elements and/or expressing the transcription factor AP1 in a heterogeneous population of T-cell networks, computed using a product-basis calculation. The heterogeneous population begins at $t = 0$ as a uniform mixture of all possible $2^{33} \approx 10^{10}$ initial states of the upstream portion of the model. **C)** The effect of a 10^{-4} knock-out mutation rate per gene in the heterogeneous population. Monte Carlo, but not the product-basis calculation, required this high rate of mutations in order to detect persistent coactivation of CRE and AP1. **D)** The co-occurrence of CRE activation and AP1 expression in mutated networks shown on a log-10 scale (dotted red line), compared with the amount of this coexpression coincident with mutated cCbl (purple dots). The mutated fraction was computed by subtracting the time series of CRE AND AP1 AND WT-cCbl from the time series of CRE AND AP1.

tions could recapitulate this phenotype.

Our results from the T-cell network demonstrate several important aspects of our method. First, we are able to simulate extremely heterogeneous populations, involving far more subpopulations than could be analyzed individually. Second, although our method only deals with heterogeneity in the states of the Boolean variables, we can still simulate a genetically-heterogeneous population by augmenting the Boolean network with mutation variables. Third, we can exactly model subpopulations that are present at very low levels, which are difficult to resolve by random sampling (see the error bars in Figure 3B). For example, the contribution of each triple-mutant was factored in even though a given triple-mutant was present in only 10^{-12} of the population. While one might artificially raise the Monte Carlo mutation rate to oversample the mutations [7], this has the disadvantage of overweighting the effect of multiple mutants, even though realistic evolutionary paths take one or very few mutational steps at a time [27]. In contrast, our exact result is dominated

by the evolutionarily-accessible subpopulations that are closest to wild-type.

MATLAB code The code used to generate these results is named `tCellActivationEx.m`, and is available for download at `https://github.com/CostelloLab/ProductBasis`. The equation-generating process for Figures 3B and 3C took $\sim$ 3 and $\sim$ 300 seconds respectively using our code (written in MATLAB R2015b 8.6.0.267246, running on a 2.6GHZ Intel core i7 Mac with OS 10.9.5). The Monte Carlo comparison in Figure 3C ($n_{runs} = 10^4$) took $\sim$ 140 seconds.

Discussion

Our product-basis method allows the direct simulation of highly heterogeneous populations, including the transient processes that are ignored by Boolean-attractor analyses. Our approach can be used to follow single variables of the system over time, as well as the correlations between these variables that are both necessary and sufficient to fully describe the dynamics of the population. We also showed that our method, when applied to a network augmented by mutation nodes, can effectively explore heterogeneity in the network rules in addition to heterogeneity in network state. In each of our simulations, all subpopulations are exactly accounted for in the output time series, no matter how rare. The only requirement for use of our approach is that the underlying model be built using Boolean variables.

The key to our method is to write the time-evolution equations as a linear system, but in a different basis than the usual state space basis. Our variables have several advantages over state space variables. First, descriptors of a mixed population naturally use words that correspond more closely to our variables than to individual states. For example, we might specify that half the population starts with both genes A and B on, which implies that $x_{AB} = 0.5$, but is agnostic about the state of the rest of the network. Another advantage is that our equations often close using relatively few of our product variables for any mixed population, whereas the number of equations required in the state space basis scales with the heterogeneity of the population: the simulations we showed in Figure 3 would require all 2^{40} state space variables. Thus our choice of variables is better for modeling very heterogeneous populations. Finally, our basis allows for a variable-factorization scheme which can help to simplify the calculation if we only care about the long-term behavior.

We acknowledge that our method can become intractable for complex networks due to the fact that construction of these simulations is potentially an exponential problem. Fully-connected Boolean networks with random logic rules will probably always be challenging, but we believe it should be possible to improve our performance on certain network motifs such as downstream feedback loops that give our method difficulty. Our proposed basis is only one of very many obtained through shear-type transformations similar to our T matrices, and it may well be that other choices of basis perform better on certain types of network. Future work will explore this possibility. Additionally, the equation-reduction method for finding attractors (Appendix 2) is somewhat of an art, and our future work will aim to improve this part of the calculation for typical network models (although the attractor analysis is also known to be NP-hard [28]).

The product-basis method can be applied to any system involving heterogeneous populations, as long as the individuals in a population can be modeled using Boolean logic. Heterogeneity plays a major role in such varied systems as healthy and cancerous tissues, evolution at the organism scale, and the social dynamics of unique individuals [29]. In all of these cases, rare and unexpected dynamics are difficult to capture by simulations of individuals, while pure attractor analyses may miss important aspects of the dynamics. We have demonstrated that the methodology outlined here can help to capture these important but elusive events.

Appendices

Appendix 1: product variables form a complete and independent basis spanning the state space

Claim 3. *The change-of-basis matrix carrying* $\mathbf{b}$ *variables to* $\mathbf{x}$ *variables is an invertible transformation.*

Proof. Our proof first requires us to explicitly write the change-of-basis matrix T. We can do this using Algorithm 1 once we have chosen a particular ordering of the entries of both the $\mathbf{b}$ and $\mathbf{x}$ vectors. Here we will choose to order the entries of both $\mathbf{b}$ and $\mathbf{x}$ by binary order of the subscripts on their respective state variables. In other words, the index is given by the binary number having 1s in the positions indicated by the subscripts of each state variable: the state variable written as $b_\alpha = b_{ij\ldots}$ is the mth entry in the state-space vector $\mathbf{b}$ and $x_\beta = x_{ij\ldots}$ is the mth entry in the product-space vector $\mathbf{x}$, where $m = 2^{u-1} + 2^{v-1} + \cdots + 1$ if we take $1 \leq \{i, j, \ldots\} \leq N$. This choice of ordering fixes a particular representation of the change-of-basis matrix. For example, for the case of three Boolean variables the change-of-basis matrix is:

$$T_3 = \begin{array}{c} \\ 1 \\ x_A \\ x_B \\ x_{AB} \\ x_C \\ x_{AC} \\ x_{BC} \\ x_{ABC} \end{array} \begin{array}{c} \begin{array}{cccccccc} b_\emptyset & b_A & b_B & b_{AB} & b_C & b_{AC} & b_{BC} & b_{ABC} \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Our objective is to show that for arbitrarily-sized systems, the change-of-basis matrix is upper-triangular, with ones on the diagonal.

Lemma 1. *The diagonal entry* $x_{(m)}$ *equals 1.*

Proof. The vector component $x_{(m)}$ lying along the diagonal T_{mm} is the value of product variable having the same set of indices as $b_{ij\ldots}$, namely $x_{ij\ldots}$. Since $i, j, \ldots$ are the set of ON variables, $x_{ij\ldots}$ evaluates to $B_i \cdot B_j \cdot \cdots = 1 \cdot 1 \cdot \cdots = 1$. □

Lemma 2. *All entries below the diagonal* ($x_{(p)}$ *for* $p > m$) *equal 0.*

Proof. Assume we have some $x_{(p)} \neq 0$ for $p \neq m$, corresponding to some product-space variable $x_{kl\ldots}$ which is a product of Boolean variables $B_k \cdot B_l \cdot \ldots$ which must each be nonzero in order for $x_{(p)} = x_{kl\ldots}$ to be nonzero. Since $\{B_i, B_j, \ldots\}$ is the set of ON Boolean variables, we have that $P = \{k, l, \ldots\}$ is a proper subset of $M = \{i, j, \ldots\}$; note that $P \neq M$ because $p \neq m$ and each index position corresponds to a unique set of Boolean variables (because binary representations of numbers are unique). Thus $m = \sum_{r \in M} 2^{r-1} + 1 = (\sum_{r \in P} 2^{r-1} + 1) + \sum_{r \in (M \setminus P)} 2^{r-1}$ where $M \setminus P$ is nonempty, implying that $m > p$. Therefore, if $p > m$ then the premise cannot hold, and $x_{(p)} = 0$. □

Repeating this logic over all indices m, we conclude that each diagonal entry of the change-of-basis matrix is 1, and the entire lower triangle is 0. Since the change-of-basis matrix is triangular the eigenvalues are the diagonal entries, which are all 1. Therefore the determinant is 1, so the change-of-basis transformation is non-degenerate (and volume-preserving) and therefore invertible. □

Appendix 2: equation-reduction for long-time behavior

A mixed-population simulation may or may not be practical, depending on whether the system of linear equations closes with a manageable number of variables. One way to simplify the equations is to focus on the long-time behavior of the system, after quickly-decaying eigenmodes have dropped near zero and time evolution effectively occurs in the smaller subspace of slowly-decaying or persistent eigenmodes. In the limit where we focus on the infinite-time behavior, this subspace is simply the attractors of the system (steady states or limit cycles). Our general strategy will be to 1) identify fast-decaying modes as they appear in F, set them to zero and use the resulting expression as a constraint that can be used 2) eliminate variables from later steps of the calculation. In order to be useful for simplifying calculations, these steps must be taken before the linear system closes – in other words, when our partially-computed update function $F_{partial}$ is still a rectangular matrix.

Our strategy for finding decaying modes in $F_{partial}$ relies on a QR decomposition, which can be performed on rectangular matrices. In the case of a deterministic network, all eigenvalues have modulus 0 or 1 implying that all decaying modes are strict linear dependencies. (*Proof*: The state space and thus by Appendix 1 the product-basis space is finite, so every eigenmode must either decay to zero in finite time implying that $\lambda = 0$, or else repeat in Δt time steps so that $|\lambda| = 1$ and $\arg(\lambda) = 2\pi k/\Delta t$ for integer k.) The set of linearly-dependent rows in F is found by identifying (numerically) zero entries along the diagonal of the R matrix. The dependencies themselves are then calculated by solving the matrix equation $N = CD$ for polynomial coefficients C, where N and D are the submatrices of F containing its independent/de-

pendent rows. For PBNs, the complication is that decaying eigenvalues of the full system are generally nonzero, so the decaying modes are not simple linear dependencies. Note that each decaying eigenvalue is part of the spectrum of $F_\square$ (the square component of $F_{partial}$), although the reverse is not true. Our implementation uses the spectrum of $F_\square$ as a set of candidate eigenvalues, and for each sufficiently-small candidate λ (based on a user-defined threshold) finds the linearly dependent subspace using a QR decomposition of $F_{partial} - \lambda I_{partial}$, where I is rectangular with ones on the diagonal and zeros elsewhere.

Each constraint takes the form of a polynomial in $\mathbf{f}$ that effectively becomes zero after a certain number of time steps have passed, implying that the same polynomial in $\mathbf{x}$ becomes zero one time step later. The general form of a constraint is thus:

$$C = c_\alpha x_\alpha + c_\beta x_\beta + \ldots \underset{t \geq t_c}{=} 0$$

There are several ways in which these constraint polynomials could help simplify the calculation, but we must be careful that 1) application of a constraint should not undo the work of a prior constraint, and 2) there should be a way of resolving how two constraints attempt to remove the same variable. Here we give two constraint algorithms that help to simplify long-time analyses in the cases of probabilistic and deterministic Boolean models respectively.

Algorithm 5 (Equation-reduction method 1)**.** *Given constraint C, we choose the 'lowest' variable x_ϕ (which may be $x_\emptyset = 1$) based on a binary representation of its indices phi as in Appendix 1, and eliminate that variable from all current and future time-evolution equations in F by simple substitution: $x_\phi \to -(c_\alpha/c_\phi)x_\alpha - (c_\beta/c_\phi)x_\beta - \ldots$. Constraining a time-evolution equation valid from time t_e using this formula yields a different time-evolution equation valid for $t \geq \max(t_e, t_c)$.*

Proof. The substitution is justified for $t \geq tc$, so the resulting time-evolution equation is accurate after time t_c. A given constraint either eliminates a variable (if the constraint polynomial contains only one term), or else replaces a variable exclusively with 'higher'-ordered variables. Each replacement with higher-ordered variables increases the total number of indices in each constrained term by at least 1, and since the maximum number of indices is N the process of applying constraints eventually ends. Each constraint application permanently eliminates one variable from the finite 2^N-sized variable space, so the process of constraining the equations eventually ends, which is only possible if there are no more fast-decaying modes. Thus this process eventually eliminates the fast-decaying eigenspace, while preserving the slowly-decaying/persistent eigenspace (because the constrained f equations are valid after a certain time $t_{c_{max}}$). □

We use reduction method 1 for PBNs and asynchronous networks as the second method does not apply to them. This method is relatively inefficient since it only eliminates one variable per constraint. There is also some indeterminism: if two constraints apply to a given term in an equation, we simply choose one of the two constraints to apply. We give a simple demonstration of reduction-method 1 (on a deterministic network) by using it to find the attractors of Example 1.

Example 1, continued

Eqs. 1.1-1.6 in Example 1 contain a single linear dependency: $f_{AB} = f_{ABC}$. Since the network is deterministic, this dependency guarantees that after the first time step x_{AB} will equal x_{ABC}, so we write $x_{AB} \underset{t \geq 2}{=} x_{ABC}$. We use this fact to eliminate x_{AB}, giving a new set of steady state equations:

$$\begin{aligned} f_A &= 1 - x_B \\ f_B &= x_{AC} \\ f_\emptyset &= 1. \\ f_{AC} &\underset{t \geq 2}{=} x_A - x_{ABC} \\ f_{ABC} &= x_{AC} - x_{ABC} \end{aligned} \tag{1.4'}$$

Our new set of equations has the same non-zero eigenspace as the original set (1.1-1.6), except Eq. 1.4′ is only valid from the second time step onwards. However, the new equations lack the null eigenspace because we removed the only linear dependency. The 5 eigenvalues all have phases that are multiples of $2\pi/5$, indicating that the sole attractor is a limit cycle with a period of 5 time steps. We acknowledge that in this case the constraint helps find the attractors but does not simplify the calculation, since it only appears after the full F has been calculated.

Algorithm 6 (Equation-reduction method 2)**.** *Given the constraint polynomial C, we select a variable x_ϕ whose indices ϕ have the property $\gamma \backslash \phi \neq \emptyset$ for every other term x_γ. Constraining a time-evolution equation of a deterministic model valid from time t_e involves constraining every variable x_ρ in that equation that is factorized by x_ϕ, in the sense that $\phi \backslash \rho = \emptyset$. Application of the constraint takes $x_\rho \to -(c_\alpha/c_\phi)x_{\alpha \cup \rho} - (c_\beta/c_\phi)x_{\beta \cup \rho} - \ldots$, yielding an equation that is valid for $t \geq \max(t_e, t_c)$.*

Proof. For time $t \geq \max(t_e, t_c)$ both the original time-evolution equation and the constraint equation are valid, implying that $x_\phi = -(c_\alpha/c_\phi)x_\alpha - (c_\beta/c_\phi)x_\beta - \ldots$. For deterministic models we have $x_\rho = x_\phi \cdot x_\rho$, since $\phi \subseteq \rho$ and indices can be freely duplicated in deterministic models. Substituting the two formulas gives the constraint rule. Since each constraint application either eliminates x_ϕ or replaces it with variables containing more indices, the arguments of constraint method 1 apply, showing that the method eventually eliminates the fast-decaying eigenspace while leaving correct time-evolution equations for the slower-decaying modes. □

Equation-reduction method 2 is considerably more aggressive than method 1, because each constraint eliminates not only one or more variables involved in the constraint, but also any variables that are factorized by them: thus a dependency involving a low-order variable with few indices can exclude a significant fraction of the variable space. We demonstrate this method 2 using a second example.

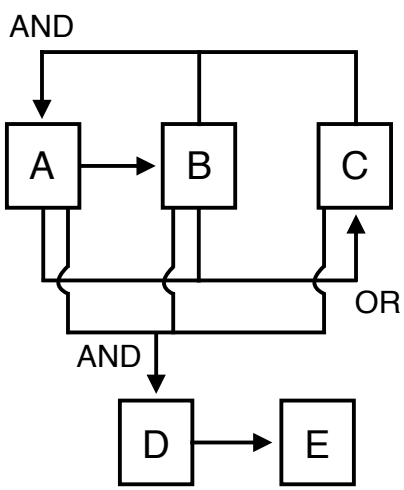

Figure 4. The network used in Example 2.

Example 2: applying constraints

Suppose we want to find the long-time behavior of Boolean variables A, C, and E in the network of Figure 4. After two iterations of solving for time-evolution functions we have:

$$f_A = x_{BC} \tag{2.1}$$
$$f_C = x_A + x_B - x_{AB} \tag{2.2}$$
$$f_E = x_D \tag{2.3}$$

$$f_B = x_A \tag{2.4}$$
$$f_D = x_{ABC} \tag{2.5}$$
$$f_{AB} = x_{ABC} \tag{2.6}$$
$$f_{BC} = x_A \tag{2.7}$$

At this point there are two linear dependencies: $f_{AB} = f_D$ and $f_{BC} = f_B$, implying that

$$x_{AB} \underset{t\geq 2}{=} x_D \tag{C1}$$
$$x_D \underset{t\geq 2}{=} x_{AB} \tag{C2}$$
$$x_B \underset{t\geq 2}{=} x_{BC}. \tag{C3}$$

Since we are only interested in the long-time behavior, we will use these constraints to simplify our equations. For example, if we were to retain x_{ABC} it would be affected by the relationships involving x_{AB}, x_B, and x_{BC}, and it would not be possible to enforce all of these by substitution because there is only one B-index on x_{ABC}. But we can simultaneously enforce all constraints by *multiplying* them:

$$\begin{aligned} x_{ABC} &= x_{AB} \cdot x_B \cdot x_{BC} \cdot x_{ABC} \\ &\underset{t\geq 2}{=} x_D \cdot x_{BC} \cdot x_B \cdot x_{ABC} \\ &= x_{ABCD}. \end{aligned}$$

More generally, the first constraint attaches an AB index to every variable containing a D, and a D index to each variable with AB indices, and the second constraint adds a C index to every variable with a B index.

Constraining our system and eliminating disused variables gives us

$$f_A = x_{BC}$$
$$f_C \underset{t\geq 2}{=} x_A + x_{BC} - x_{ABCD} \tag{2.2$'$}$$
$$f_E \underset{t\geq 2}{=} x_{ABCD} \tag{2.3$'$}$$
$$f_{BC} = x_A.$$

Our new equations require us to solve for another variable (while applying the constraints):

$$f_{ABCD} \underset{t\geq 2}{=} x_{ABCD}. \tag{2.8}$$

The system is now closed (5 equations involving 5 variables), so if our goal is to produce a simulation (valid from time step 2 onwards) then we are done. However, if our objective is to find the attractors, then we must remove another dependency that was unmasked by the last equation: $f_C \underset{t\geq 2}{=} f_{BC} + f_A - f_E$, which implies three useful constraints.

$$x_A \underset{t\geq 3}{=} x_C + x_E - x_{BC} \tag{C4}$$

$$x_C \underset{t\geq 3}{=} x_A - x_E + x_{BC} \tag{C5}$$

$$x_E \underset{t\geq 3}{=} x_A - x_C + x_{BC} \tag{C6}$$

Each constraint reduces the size of the variable space: for example, the first eliminates all variables containing index A with no C, E, or BC. We did not solve for x_{BC} because doing so would not eliminate any variables, because the x_C term does not attach any new indices.

After applying the new constraints we obtain:

$$f_{BC} \underset{t\geq 3}{=} x_{AC} + x_{AE} - x_{ABCD} \tag{2.7$'$}$$

$$f_{ABCD} \underset{t\geq 2}{=} x_{ABCD}$$

$$f_{AC} = x_{BC} \tag{2.9}$$

$$f_{AE} \underset{t\geq 2}{=} x_{ABCD} \tag{2.10}$$

$$f_{CE} \underset{t\geq 2}{=} x_{ABCD} \tag{2.11}$$

$$f_{BCE} \underset{t\geq 2}{=} x_{ABCD} \tag{2.12}$$

This produces new dependencies: $f_{AE} \underset{t\geq 2}{=} f_{ABCD}$ and $f_{CE} \underset{t\geq 2}{=} f_{ABCD}$, implying that

$$x_{AE} \underset{t\geq 3}{=} x_{ABCD} \tag{C7}$$

$$x_{ABCD} \underset{t\geq 3}{=} x_{AE} \tag{C8}$$

$$x_{CE} \underset{t\geq 3}{=} x_{ABCD}. \tag{C9}$$

With these constraints our example ends with the following closed system of dynamical equations:

$$f_{BC} \underset{t\geq 3}{=} x_{AC} \tag{2.7$''$}$$

$$f_{ABCDE} \underset{t\geq 3}{=} x_{ABCDE} \tag{2.13}$$

$$f_{AC} = x_{BC}.$$

The attractor is always reached at or before time step 3. The constraint equations (C1-C9) map our variables of interest (x_A, x_C, and x_E) to linear combinations of variables in the final system: for example, x_A is mapped by constraint C4 to $x_{AC} + x_{AE} - x_{ABC}$, then mapped using constraints C1, C7, and C8 to x_{AC}, whose dynamics are given by the final time-evolution equations. The eigenvalues of this final system are $(-1, 1, 1)$, implying a steady state along with a period-2 cycle.

When using equation-reduction method 2, the order in which we calculate the time-evolution of new variables and remove dependencies greatly influences the complexity of the calculation. One general rule we have found is that constraints on the lowest-index variables are most helpful for deterministic networks, because they factorize the largest part of the variable space. Following this rule of thumb, our implementation solves the time-evolution equations for only those variables having the fewest indices between each search for new dependencies. Additionally, our code solves for low-index factors of new variables even if they were not directly involved in earlier equations in hopes that they will produce helpful constraints. In most tests, this prioritization method speeds up the calculation.

Equation-reduction method 2 is susceptible to numerical problems that fall into two categories: 1) polynomial coefficients can become very large, and 2) gradual erosion of numerical precision can blur the distinction between zero and nonzero terms. In addition our code monitors both sources of error; in the latter case by tracking the floating-point error in each step of the calculation. When either type of calculation error exceeds defined bounds, our method exits with an error message.

The reason polynomial coefficients can become large is that multiplication by constraints can produce several identical terms that add together. For example, the constraint $x_1 = x_2 + x_3$ takes $x_{123} \to 2x_{123}$. Typically the variable being multiplied would be found to be zero later in the calculation (so having a coefficient of 2 is not an error in the math), but if many such constraints are applied and the coefficient grows exponentially then the numerics will eventually fail. We have included a filter that can usually identify these outlier coefficients and remove them using the rule that all individual (not population-averaged) product variables are either 0 or 1 at all time

steps. For example, if $f_4 = 2x_{123}$ then we can reason that if f_4 is either 0 or 1, then x_{123} cannot be 1 and therefore we have a new constraint $x_{123} = 0$ which simplifies $f_4 \to 0$. In general, we identify these removable terms using the heuristic that if a large coefficient multiplying a combination of integer-weighted variables is greater than the sum of all other coefficients plus one, then that combination of variables (which is an integer for a homogeneous population) must be zero.

Our equation-reduction method 2 was tested by running 10^4 comparisons between Monte Carlo and our dynamical equations using a random amount of equation-reduction (which determines the earliest time point at which the simulation is valid) for each test. We then compared the predictions of our exact simulations to the population dynamics as predicted by Monte Carlo. In each case the product-basis simulation exactly reproduced the average of the Monte Carlo simulations. These networks were small, but still provided a stringent test of the numerics, as they could theoretically involve linear systems of equations up to size $(2^{10})^2$. During this test, our program also generated additional networks whose numerical error overstepped a threshold and were thus omitted from the Monte Carlo comparison; overall 18.5% of tests generated were discarded.

Appendix 3: algorithm and code

Pseudocode is given in Algorithm 1. The full code is available at: `https://github.com/CostelloLab/ProductBasis`.

Algorithm 1 build closed system of equations F

Initialize set of unsolved variables with variables of interest: $X \leftarrow \{x_1, x_2, ..., x_n\}$
Initialize set of variable update rules: $F \leftarrow \emptyset$
Initialize set of constraints: $C \leftarrow \emptyset$
Initialize simulation start time: $t_{sim} \leftarrow 1$
Initialize set of equation start times of F: $T_F \leftarrow \emptyset$
Initialize set of constraint start times: $T_C \leftarrow \emptyset$

while X is not empty **do**

    % Reduce equations if necessary
    if $size(F) > equation_reduction_threshold$ **then**
        $F_\square \leftarrow$ square-matrix component of F (i.e. only terms in solved variables for each f_i)
        $\lambda \leftarrow$ set of eigenvalues of $F_\square^T$
        for each $\lambda_i \in \lambda$ **do**
            $G \leftarrow F - \lambda_i[I\ 0]$
            $R \leftarrow$ upper-triangular matrix from $QR(G)$
            $V_j \leftarrow$ set of dependent variables j found from $|R_{jj}| < \epsilon$
            $D \leftarrow$ set of linear-dependencies found by $V_j = G(\overline{V_j})/G(V_j)$
            if $V_j \neq \emptyset$ **then**
                $t_{sim} \leftarrow \min(t_{sim}, \min(T_F(x_k \in D)) + \max(1, \text{ceiling}(\log \epsilon_{decay} / \log |\lambda_i|)))$
                break
            end if
        end for
        $D \leftarrow$ sort $\{d_i$ for which $n_{terms}(d_i) \leq 2 \min n_{terms}(d)\}$ by number of terms
        for each $d_i \in D$ **do**
            NewConstraints(d_i, t_{sim})
        end for
        Sort C by number of terms
    end if

    % Add new equations
    $X \leftarrow X \cup$ lowest-index factors of X
    Sort X by number of terms
    for each $x_i \in X$ **do**
        $f_i \leftarrow 1$
        for each Boolean factor x_b of x_i **do**
            if f is a discrete update rule **then**
                $f_i \leftarrow f_i \cdot f_b$
            else
                $f_i \leftarrow f_i + f_b \cdot \prod_{x_j \neq x_i \in X} x_j$
            end if
        end for
        $F \leftarrow F \cup$ Constrain$(f_i, 1)$
    end for

end while

```
function NewConstraints(d, t_c)
    if d ← RemoveOversizeCoefficients(d/minCoef(d)) adds no new constraint then
        for each x_j ∈ d that does not factor another x_{k≠j} ∈ d do
            RHS_j ← solve (d = 0) for x_j
            if {x_j} ≠ Constrain({RHS_j}, t_c)[1] then
                for each prior constraint (x_p = RHS_p) do
                    if x_p is factorized by x_j and x_p · RHS_j = RHS_p then
                        C ← C \ {c_p}
                    end if
                end for
                C ← RemoveOversizeCoefficients(C ∪ {x_j = RHS_j})
                for each x_k = multiples of x_j do
                    X_new ← new terms in Constrain(x_k)
                    X ← X ∪ X_new
                    A ← {a_l = coefficients of x_k in F}
                    F ← F + A · (RHS_k − x_k)
                    (t_{f_l} for all l such that a_l ≠ 0) ← t_c
                end for
            end if
        end for
    end if
end function

function Constrain(poly, t_c)
    while ∃ unconstrained terms in poly do
        for each constraint x_i = RHS_i do
            % Multiply all factored variables by the constraint
            for each unconstrained variable x_j with coefficient a_j in poly do
                if x_j is factorized by x_i then
                    poly_new ← x_j · RHS_i
                    if x_j ⊄ poly_new then
                        poly ← (poly − a_j · x_j) ∪ poly_new
                        t_{f_i} ← t_c
                    end if
                end if
            end for
            poly ← RemoveOversizeCoefficients(poly)
        end for
    end while
    return Polys
end function

% Check for overlarge coefficients
function RemoveOversizeCoefficients(poly)
    while ∃ unchecked coefficients in poly_i do
        for each coefficient a_i in poly_i do
            n_j = round(a_j/a_i)
            if |a_i| > Σ_j |a_j − n_j a_i| then
                NewConstraints(n_j, t_sim)
                a_j ← a_j − n_j a_i for all j
                break
            end if
        end for
    end while
    return Polys
end function
```

Acknowledgments

We would like to thank Sharon Lutz for giving useful mathematical advice, and Federico Stefanini and Julio Saez-Rodriguez for helpful comments on the manuscript. This work is supported by the Boettcher Foundation (J.C.C.) and NIH grant 2T15LM009451 (B.C.R.).

References

[1] David J Earl and Michael W Deem. "Monte Carlo simulations". *Molecular modeling of proteins* (2008), pp. 25–36.

[2] Akshata R Udyavar et al. "Novel hybrid phenotype revealed in Small Cell Lung Cancer by a transcription factor network model that can explain tumor heterogeneity". *Cancer Research* (2016), canres–1467.

[3] Li Ding et al. "Clonal evolution in relapsed acute myeloid leukaemia revealed by whole-genome sequencing". *Nature* 481.7382 (2012), pp. 506–510.

[4] Ash A Alizadeh et al. "Toward understanding and exploiting tumor heterogeneity". *Nature medicine* 21.8 (2015), pp. 846–853.

[5] Brett E Johnson et al. "Mutational analysis reveals the origin and therapy-driven evolution of recurrent glioma". *Science* 343.6167 (2014), pp. 189–193.

[6] Kenneth Murphy and Casey Weaver. *Janeway's immunobiology*. Garland Science, 2016.

[7] Jun S Liu. *Monte Carlo strategies in scientific computing*. Springer Science & Business Media, 2008.

[8] István Albert et al. "Boolean network simulations for life scientists". *Source code for biology and medicine* 3.1 (2008), p. 16.

[9] Ilya Shmulevich et al. "Probabilistic Boolean networks: a rule-based uncertainty model for gene regulatory networks". *Bioinformatics* 18.2 (2002), pp. 261–274.

[10] Roberto Serra, Marco Villani, and Anna Salvemini. "Continuous genetic networks". *Parallel computing* 27.5 (2001), pp. 663–683.

[11] Ilya Shmulevich and John D Aitchison. "Deterministic and stochastic models of genetic regulatory networks". *Methods in enzymology* 467 (2009), pp. 335–356.

[12] Jorge GT Zañudo and Réka Albert. "An effective network reduction approach to find the dynamical repertoire of discrete dynamic networks". *Chaos: An Interdisciplinary Journal of Nonlinear Science* 23.2 (2013), p. 025111.

[13] Alan Veliz-Cuba et al. "Steady state analysis of Boolean molecular network models via model reduction and computational algebra". *BMC bioinformatics* 15.1 (2014), p. 1.

[14] Aurélien Naldi et al. "A reduction of logical regulatory graphs preserving essential dynamical properties". *International Conference on Computational Methods in Systems Biology*. Springer. 2009, pp. 266–280.

[15] Stefan Bornholdt. "Boolean network models of cellular regulation: prospects and limitations". *Journal of the Royal Society Interface* 5.Suppl 1 (2008), S85–S94.

[16] Takeyuki Tamura and Tatsuya Akutsu. "Detecting a singleton attractor in a Boolean network utilizing SAT algorithms". *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 92.2 (2009), pp. 493–501.

[17] Elena Dubrova and Maxim Teslenko. "A SAT-based algorithm for finding attractors in synchronous boolean networks". *IEEE/ACM transactions on computational biology and bioinformatics* 8.5 (2011), pp. 1393–1399.

[18] Desheng Zheng et al. "An efficient algorithm for computing attractors of synchronous and asynchronous Boolean networks". *PloS one* 8.4 (2013), e60593.

[19] Abhishek Garg et al. "Synchronous versus asynchronous modeling of gene regulatory networks". *Bioinformatics* 24.17 (2008), pp. 1917–1925.

[20] Alan Veliz-Cuba. "Reduction of Boolean network models". *Journal of theoretical biology* 289 (2011), pp. 167–172.

[21] Alan Veliz-Cuba, Abdul Salam Jarrah, and Reinhard Laubenbacher. "Polynomial algebra of discrete models in systems biology". *Bioinformatics* 26.13 (2010), pp. 1637–1643.

[22] Vincent Devloo, Pierre Hansen, and Martine Labbé. "Identification of all steady states in large networks by logical analysis". *Bulletin of mathematical biology* 65.6 (2003), pp. 1025–1051.

[23] Aurélien Naldi, Denis Thieffry, and Claudine Chaouiya. "Decision diagrams for the representation and analysis of logical models of genetic networks". *International Conference on Computational Methods in Systems Biology*. Springer. 2007, pp. 233–247.

[24] Steffen Klamt et al. "A methodology for the structural and functional analysis of signaling and regulatory networks". *BMC bioinformatics* 7.1 (2006), p. 56.

[25] Panuwat Trairatphisan et al. "Recent development and biomedical applications of probabilistic Boolean networks". *Cell communication and signaling* 11.1 (2013), p. 46.

[26] Bert Vogelstein et al. "Cancer genome landscapes". *Science* 339.6127 (2013), pp. 1546–1558.

[27] Michael Lynch. "Rate, molecular spectrum, and consequences of human mutation". *Proceedings of the National Academy of Sciences* 107.3 (2010), pp. 961–968.

[28] Qianchuan Zhao. "A remark on "Scalar equations for synchronous Boolean networks with biological Applications" by C. Farrow, J. Heidel, J. Maloney, and J. Rogers". *IEEE Transactions on Neural Networks* 16.6 (2005), pp. 1715–1716.

[29] Jurek Kolasa and C David Rollo. "Introduction: the heterogeneity of heterogeneity: a glossary". *Ecological heterogeneity*. Springer, 1991, pp. 1–23.