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 the development of cancer are controlled by events that are very rare at the single-cell level, and which are therefore difficult to capture computationally through simulations of individual cells. Here we demonstrate a method for exactly simulating the dynamics of large populations directly, without having to model the individuals in the population. Our method can thus resolve even the rarest outcomes of the model. The only restriction is that the underlying model use only Boolean variables. The populations we consider are composed of non-interacting individuals that span an arbitrary mixture of states, but are described by the same underlying model, although by adding 'mutation variables' one can also simulate heterogeneity in model rules. Boolean models are thus ideal for exploring the dynamics of highly heterogeneous populations in which very rare subpopulations may play important roles.

Computer models are widely used to predict typical behaviors of biological systems, generally by simulating a number of instances of that model and enumerating the range of observed outcomes. In the most common random-sampling procedure called Monte Carlo [1], many individual instances of that model are initialized to different random starting states (determined by some predefined distribution) and then evolved over discrete time steps by applying the rules of the model. These simulations can be run very efficiently, allowing the evaluation of complex models, including whole-cell simulations [2]. 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 the rare outliers [3]. For example, tumor initiation, progression, and drug resistance require either rare alterations to arise or selection of low frequency, pre-existing populations to clonally expand and eventually dominate the population in the long run [4–6]. In situations where 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]); however in practice the bias is usually limited to simple parameters such as a mutation rate, which can help narrow the search but may be insufficient to find an outcome that results only from very particular circumstances. Furthermore, oversampling can introduce sampling biases: for example, simulations using an unrealistically high mutation rate may discover extraordinary outcomes by virtue of having huge mutational loads, but such a mutational profile may be neither biologically-realistic nor evolutionarily-accessible.

The shortcomings of Monte Carlo have led to interest in exact, analytical solutions to biological models that are not limited by finite sampling; such exact results require very simple models. In particular, significant advances have been made in analyzing Boolean networks [8–16], a class of models built entirely from ON/OFF variables. With these models, the focus has been on extracting the possible long-term outcomes, or *attractors*. An attractor may be a stable state (steady state) or else a repeating pattern of states (limit cycle). Attractors have been found using network-reduction algorithms that find simple networks encoding the long-term behavior of more complex networks [9, 11, 17], methods that solve steady states as zeros of a polynomial equation [18], SAT methods [13, 14, 19], and binary decision diagrams [15, 16, 20]. See the introduction of Ref. [10] for a review of these techniques.

In this paper we show that Boolean networks can be used in another type of exact analysis: namely, direct simulation of population statistics, without having to simulate the individuals in that population. The simulations

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

are exact, so they capture every subpopulation of the model and every event that occurs, no matter how rare. The output of our method is a time series of the frequency with which the characteristic(s) of interest appear in the population. The feasibility of building these simulations depends on the size and topology of the network (for complex or highly-recurrent networks it can be an exponential problem), though we can often simplify the difficult cases using an equation-reduction method that ignores the first few time steps of the simulation. In the limit where we focus only on the long-term behavior, the output describes the attractor dynamics of the characteristics of interest, thus relating our method to existing Boolean analyses.

Our method applies to deterministic [8], probabilistic [21], and continuous-time [22] Boolean networks. Here, we only analyze synchronous Boolean network models (i.e. networks whose variables all update together in discrete time steps), but this is without loss of generality because an asynchronous network can be modeled as a probabilistic network with synchronous time steps [23]. One major benefit to our method is its simplicity, as it follows from only two rules: 1) work in a linear basis whose variables are products of the Boolean state variables, and 2) ignore quickly-decaying modes if we are looking at late-time behavior. Because of rule 1, we refer to our approach as the *product-basis* method.

Results

Validation 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. First, we ran 100 Monte Carlo individual simulations of each network, using a random ensemble of initial states (equal probability for all states). Next, we used our *product-basis* method to generate exact time-evolution equations using a random amount of equation-reduction (which determines the earliest time point at which the simulation is valid). 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.

Next, we tested probabilistic networks (PBNs). Our main test consisted of full-time simulations (i.e. without equation-reduction) as those had lower numerical error and allowed for a tighter Monte Carlo comparison. 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 $0 \leq r_i \leq 1$. We generated 10^4 PBNs using this technique, calculated the time-evolution equations for each, and again found that our method's equations reproduced the results of Monte Carlo simulations to within statistical error. 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.

Demonstration: T-cell network To demonstrate the *product-basis* method, we used 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 with 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 1A, 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 our *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 the 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 perfect correlated, or any other value on the interval $[0, 0.5]$.)

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 1B). We stress that this is an exact result, with no sampling error. The start-

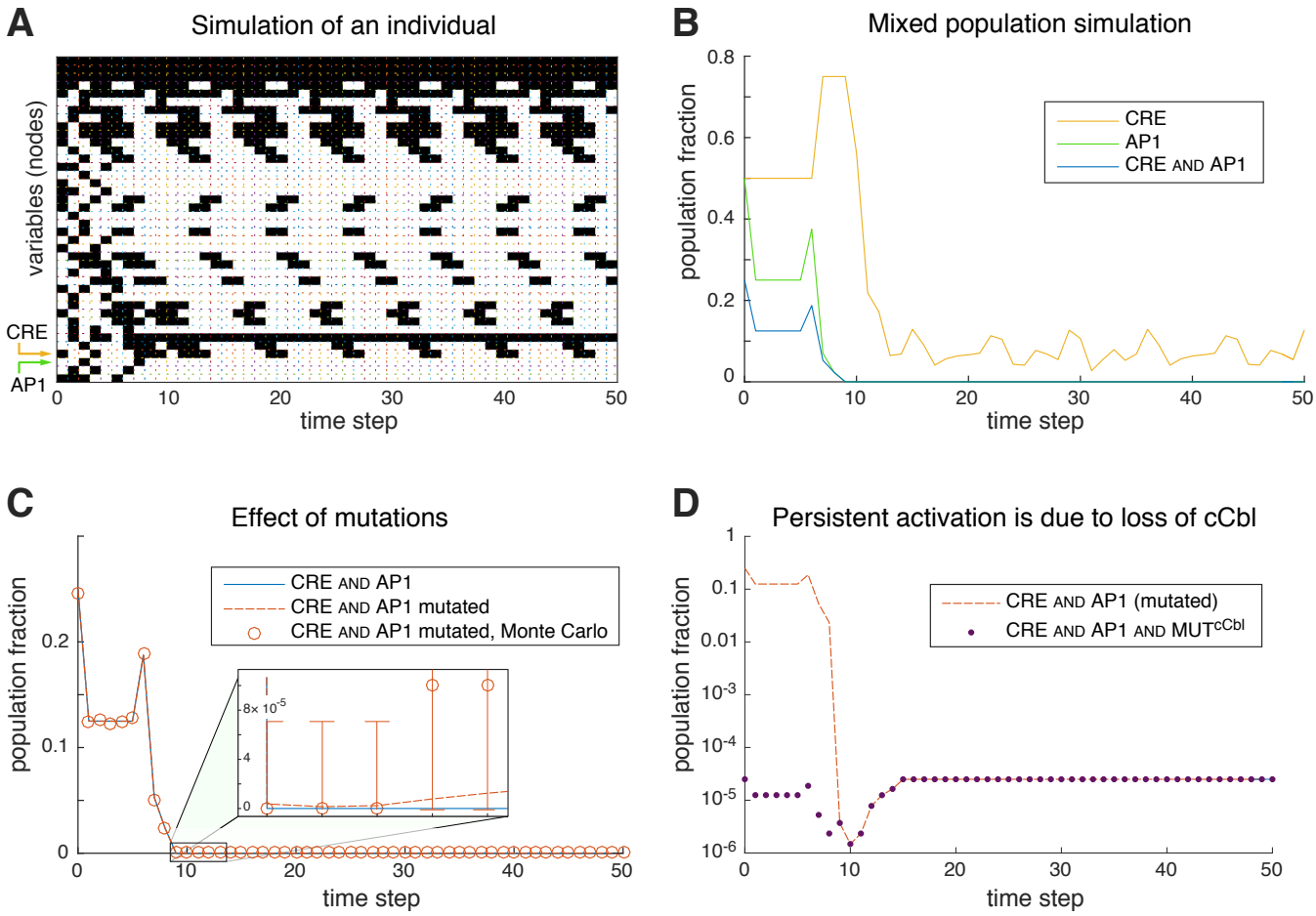

Figure 1. A) Time evolution of one instance of the T-cell network [24], starting from a random initial state. White/black rectangles signify OFF/ON Boolean states. **B)** Time evolution of the population fraction having activated CRE elements and/or expressing the transcription factor AP1 in a heterogeneous 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.

ing 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 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 recourse 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 [25]) superimposed upon a uniform mixture of each activation state (adjusted so that non-wild-type 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 1C). 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 subpopulation which fell far below its sampling resolution.

Finally, we used our method to infer 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. 1D).

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 1B). 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 [26]. In contrast, our exact result is dominated by the evolutionarily-accessible subpopulations that are closest to wild-type.

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 1B and 1C took ~ 3 and ~ 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 1C ($n_{runs} = 10^4$) took ~ 140 seconds. Note that Monte Carlo error is proportional to $1/\sqrt{n_{runs}}$.

Methods

Simulations of mixed populations

The principle behind our method is to write the time evolution of each variable in our model using a *linear equation*. Doing so guarantees that the dynamical equations we derive for a single cell also fully describe the dynamics of a mixed population of cells, owing to the superposition property of linear systems.

The key to writing linear equations is to introduce a new variable to represent each nonlinear term in a naïve update rule, acknowledging that we will have to solve for the dynamics of each new variable as well. In our case, each nonlinear term is a product of Boolean variables, so the update rule for its respective introduced variable will be a product of the constituent Boolean update rules. We demonstrate this procedure using Example 1.

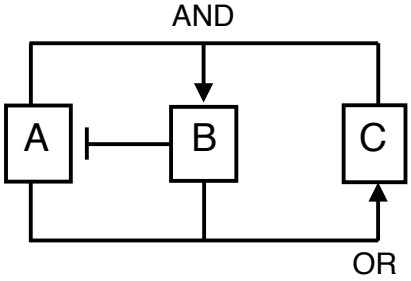

Figure 2. The 3-Boolean network used in Example 1. Arrows indicate how each variable updates based on the values of its inputs at the previous time step. For example if either A or B is ON at time t then C will be ON at time $t+1$; otherwise C will be OFF.

Example 1: building equations

Suppose we want to track the time evolution of vari-

able A in the network shown in Figure 2. Since this network evolves by discrete time steps, we write $x_A(t+1) = f_A(x_B)$ where f_A performs the NOT operation. A linear equation implementing the NOT gate is:

$$f_A(x_B) = 1 - x_B. \tag{1.1}$$

Evidently, in order to follow x_A over all time we must also track the state of its input variable B over time. B implements an AND gate which is a *nonlinear* operation: $f_B = x_A \cdot x_C$. To make the equation linear, we introduce $x_{AC} = x_A \cdot x_C$, which is 1 if and only if both A and C are ON, and write f_B in terms of this new variable.

$$f_B = x_{AC} \tag{1.2}$$

We still need to calculate f_{AC} for our new variable x_{AC}, which is simply the product $f_A \cdot f_C$. (Proof: $f_{AC} = x_{AC}(t+1) = x_A(t+1) \cdot x_C(t+1) = f_A \cdot f_C$.) f_C implements an OR gate whose linear equation involves yet another product variable x_{AB}.

$$\begin{aligned} f_{AC} &= f_A \cdot f_C \\ &= (1 - x_B) \cdot (x_A + x_B - x_{AB}) \\ &= x_A - x_{AB} \end{aligned} \tag{1.3}$$

The formula for f_{AC} made use of the fact that any Boolean value squared equals itself: for example $x_B \cdot x_B = x_B$, and $x_B \cdot x_{AB} = x_B \cdot x_A \cdot x_B = x_{AB}$.

The process of replacing product terms with new variables, and then solving for the time evolution of those new variables, continues until the equations form a closed system: each variable's time evolution is in terms of other variables in our system.

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

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

This gives us a closed linear system. To avoid the constant term we can rewrite Eq. (1.1) as $f_A = x_\emptyset - x_B$, where $x_\emptyset = 1$ updates according to:

$$f_\emptyset = x_\emptyset. \tag{1.6}$$

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.

$$f_{\langle A \rangle} = 1 - \langle x_B \rangle \tag{1.1b}$$

$$f_{\langle B \rangle} = \langle x_{AC} \rangle \tag{1.2b}$$

$$f_{\langle AC \rangle} = \langle x_A \rangle - \langle x_{AB} \rangle \tag{1.3b}$$

$$f_{\langle AB \rangle} = \langle x_{AC} \rangle - \langle x_{ABC} \rangle \tag{1.4b}$$

$$f_{\langle ABC \rangle} = \langle x_{AC} \rangle - \langle x_{ABC} \rangle \tag{1.5b}$$

Here angle-brackets denote an average, and we have used the notation $\langle f_i \rangle = \langle x_i(t+1) \rangle = f_{\langle i \rangle}$. The linear equations are unaffected by the averaging process, showing that 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. The only difference is that the equations were derived using binary variables, whereas the population-averaged variables are real-valued on the interval $[0, 1]$: for example we would set $\langle x_A \rangle = 0.4$ if 40% of the population has gene A set ON.

Any state or mixture of states can be written as a linear combination of product-basis variables $\{x\}$, because these variables form a complete basis spanning the state space (see Appendix 1 for a proof; also Ref. [18] proves a similar result for a slightly different Boolean algebra). Since each time-evolution function f is a sum over all states causing a '1' in the output variable when written in the state basis, it follows that each f is also a linear combination of our x variables. Therefore our procedure for modeling a mixed population always works in principle, even if some networks require too many equations for this method to be practical.

We can extend mixed-population modeling to probabilistic [21] and continuous-time [22] Boolean networks. Probabilistic Boolean networks require no changes in the algorithm; the only difference is that the polynomial coefficients in our equations may not be integers. For example, if the NOT gate in Fig. 2 is leaky with A turning ON with a probability of 0.9/0.4 depending on whether B was OFF/ON at the last time step, then the transition rule for gene A becomes $f_A = 0.9 - 0.5x_B$. As before, a linear equation can always be written, because a) a linear equation can still be written in the state space basis, and b) our x variables are just a different basis covering the

state space (see Appendix 1). Probabilistic networks give one way to incorporate rate information into our model; another way is to work in continuous time using differential equations: $f_A = dx_A/dt$. 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 $f_A \cdot f_B \cdot f_C \cdots$, for the differential case we compute:

$$\begin{aligned} f_{ABC\ldots} &= \frac{d}{dt}\left(x_A x_B x_C \ldots\right) \\ &= x_A x_B \ldots f_C + x_A x_C \ldots f_B + \ldots . \end{aligned} \tag{1.7}$$

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

Long-term behaviors

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 n. In the worst case, the entire 2^N-sized variable space could be involved. By counting subscripts we know that there are $n = 2^N$ product variables associated with an N-Boolean network, which is expected because our x-variables are simply a change of basis from the state space (see Appendix 1). Therefore the problem has potentially exponential complexity.

One way to make progress when the full closed system of equations is unmanageable is to focus on the attractors (steady states or limit cycles). The attractors are governed by a linear space whose size is determined by the number of attractor states, which for biological networks is usually much smaller than the full equation space. Mathematically, this means that our linear equations form a very degenerate system: if there are only n^* attractor states, then there are only n^* non-zero eigenvalues and n^* linearly independent equations. So for a 50-node network with a single steady state attractor we might have $n = 2^{50} \approx 10^{15}$ equations in the worst case, but $n^* = 1$, which is a vastly smaller linear system. To only find the structure of the final-state space we select n^* linearly independent variables, substitute them for the other variables in the time-evolution functions, and do an eigenvalue analysis on the much-smaller $n^* \times n^*$ system. A continuation of Example 1 gives a simple demonstration of this procedure.

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:

$$\begin{aligned} f_A &= 1 - x_B \\ f_B &= x_{AC} \\ f_{AC} &\underset{t\geq 2}{=} x_A - x_{ABC} \\ f_{ABC} &= x_{AC} - x_{ABC} \\ f_\emptyset &= 1. \end{aligned} \tag{1.3$'$}$$

Our new set of equations has the same non-zero eigenspace as the original set (1.1-1.6), except Eq. 1.3′ is only valid from the second time step onwards. However, the new equations lack the null eigenspace because we removed the only linear dependency. States lying in the null eigenspace by definition decay and therefore correspond to transients in the time evolution, whereas eigenvectors whose eigenvalues have magnitude 1 do not decay and are part of the final attractor states. 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. The states are: $(100) \to (101) \to (111) \to (011) \to (001) \to \ldots$ at which point the sequence repeats.

Constraints are traditionally enforced by substitution; however, there are two possible issues with substituting constraint expressions in our case. First, there is no guarantee that when applying two constraints the second will not undo the work of the first, for example by reintroducing an index on a variable that was eliminated by the first constraint. The second issue is that two dependencies might constrain overlapping indices on the same variable. We have developed the following two constraint algorithms for dealing with these issues, designed for probabilistic and deterministic models respectively.

Equation-reduction 1 The most straightforward way to simplify the equations is to use a variable-substitution strategy that prevents a variable from reappearing in a calculation after it has been eliminated. Our strategy solves each dependency for its 'lowest' variable based on a binary representation of its indices and eliminates that variable from the rest of the calculation by simple substitution. Each constraint thus replaces its substituted variable exclusively with 'higher'-ordered variables, thus guaranteeing that the application of constraints will never cause an infinite loop. We retain each constraint in memory and reapply it any time the substituted variable

reappears in another time-evolution equation. Note that equation-reduction method 1 may only apply one constraint if two constraints apply to a given variable. This reduction method is the only one that can be used by PBNs or asynchronous networks.

Equation-reduction 2 There will be no time savings in the calculation if we only eliminate a variable after we have already computed its time-evolution function. Our second equation-reduction method simplifies a calculation while solving for a model's long-term behavior, by taking advantage of the fact that each linear dependency involving Boolean variables constrains not only those variables, but also any variables that are *factorized* by them (i.e. they have all of the dependent variable's subscripts). Thus a dependency involving a low-order variable with few indices can exclude a significant fraction of the variable space. We find these constraints concurrently with the process of adding equations, and thereby potentially avoid having to evaluate a significant fraction of our variable space. This is the default reduction method for deterministic networks.

For example, in the network of Figure 3, variable x_{ABC} is subject to constraints from both x_{AB} and x_{BC}, and substituting one of these constraints could potentially eliminate the subscript B that the other constraint requires. We avoid both these problems by *multiplying* constraints rather than substituting them, using the fact that we can freely duplicate (or remove duplicates of) constrained indices on Boolean variables (because a Boolean raised to any positive power equals itself). This process forces the removal of all variables containing certain indices and lacking others, of which there is always at least one.

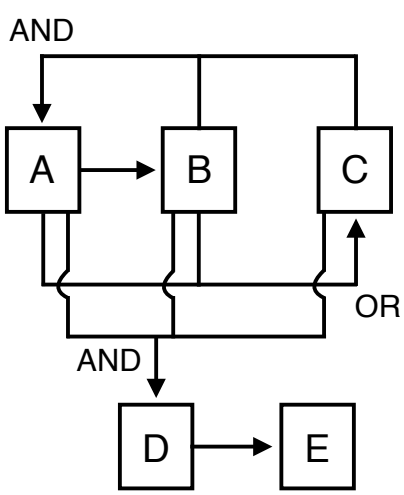

Figure 3. The network used in Example 2.

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 3. 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 our method enforces 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.

Note that multiplication of variables becomes a *union* operation on the variable indices in the *product basis*.

Each dependency produces at least one constraint that permanently eliminates at least one of the dependent variables, and often many other variables as well. *Proof*: if the dependency contains only one term, then that variable is zero, eliminating it and all variables it factorizes. If there is more than one term in the dependency, then each lowest-index variable (which may be $x_\emptyset = 1$) accumulates at least one index from another variable in the dependency. Therefore each lowest-index variable is always eliminated. *Corollary*: the calculation eventually terminates because there are a finite number of variables, and each new linear dependency removes at least one further variable.

Applying the equation-reduction method to each dependency in our linear system $F = \{f_i\}$ is guaranteed to eliminate the entire null space of F, simply because variables will be removed from the system until there are no more dependencies. On the other hand, our equation-reduction method does not affect the non-null eigenspace involving the variables of interest, because the constraints C map those variables to variables in the final equations: $X_{final} = C \cdot X_{interest}$. Therefore the long-time behavior is accurately modeled by $F_{interest} = C^T F_{final} C$, which therefore contains all the persistent eigenmodes.

In a deterministic Boolean network all eigenvalues of F have modulus either 0 or 1, owing to the fact that the state space (and therefore by Appendix 1 our *product-basis* space) is finite. Therefore every eigenmode must either decay to zero in finite time (implying that $\lambda = 0$), or else repeat in Δt time steps ($|\lambda| = 1$ and $\arg(\lambda) = 2\pi k/\Delta t$ for integer k).

Probabilistic and asynchronous Boolean networks Our method can be used to model large populations of probabilistic Boolean networks (PBNs) [21, 27], in which several state transitions are possible at each time step, and the various transitions may have different probabilities. The time evolution of an individual PBN is stochastic, so from the beginning we work with the population-averaged $f_{\langle i\rangle}$, $f_{\langle ij\rangle}$, etc., which can be thought of as the unconditional probabilities of an individual PBN evolving by $x_i \leftarrow 1$, $x_{ij} \leftarrow 1$, etc. over a time step. In the population-averaged variables, the time evolution of each product variable $f_{\langle ij\rangle}$ is again a product of single-variable time-evolution functions $f_{\langle i\rangle} \cdot f_{\langle j\rangle}$, because the constituent single-variable functions (f_i, f_j) are drawn independently for each individual in the population. Thus we can build time-evolution equations for PBNs that are valid for all time using the same equation-multiplying process we used for deterministic networks. Since PBN equations are solved using population-averaged variables that are not Boolean (so $f_{\langle i\rangle} \neq f^2_{\langle i\rangle}$) we are only able to constrain these equations using equation-reduction method 1: a constraint on a given averaged variable $x_{\langle i\rangle}$ cannot be used to factorize higher-order averaged variables such as $x_{\langle ij\rangle}$, thus reduction method 2 cannot work.

The full system of linear time-evolution equations F of a PBN contains eigenmodes having real-valued eigenvalues whose modulus is on the interval $[0, 1]$. (A modulus larger than 1 would represent a mode growing without bound, which is impossible viewed from the state-space basis because the population fraction in each state b_i is restricted to the interval $[0, 1]$). Therefore, unlike a deterministic network, a PBN can have slowly-decaying modes with eigenvalues between 0 and 1. For PBNs we generalize our equation-reduction method to identify decaying modes before all f_i have been solved (i.e. at a stage where we have only computed some $F_{partial}$ which is in general a rectangular matrix). These modes m have the property $m \cdot F = \lambda\,[m\ 0]$. We discard these modes after they have become sufficiently small, defined by $e^{-\lambda(t-\max_m(t_{f_m}))} < \epsilon$ where t_{f_m} are the 'starting times' of the involved equations and ϵ is a user-defined threshold.

Large populations of asynchronous networks behave identically to large populations of PBNs [23] if we define a uniform time step: the likelihoods of the various possible updates 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. The conversion of an asynchronous updating scheme to a PBN lowers some of the eigenvalues that would otherwise be of magnitude 1 in a synchronous version of the network, and therefore our equation-reduction method can identify more degeneracies in asynchronous deterministic networks than in their synchronous counterparts, partially compensating for the fact that each degeneracy allows for the removal of only one variable.

Calculational notes In order to be useful for simplifying calculations, the two equation-reduction methods must identify linear dependencies in our update rules before the linear system closes – in other words, when our partially-computed update function $F_{partial}$ is still a rectangular matrix. Our implementation of the reduction method finds these dependencies by using QR decomposition to find a set of dependent variables, then solves the update rules for these variables in terms of the independent variables using matrix division. 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 candidate λ 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.

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.

The *product-basis* method, particularly the equation-reduction component, 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. Our implementation of the *product-basis* method is usually able to prevent (1) from becoming a problem in deterministic networks, due to an extra filter in the calculation (see below). 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 for deterministic networks 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.

Discussion

Our *product-basis* method allows the direct simulation of highly heterogeneous populations, including the transient processes that are ignored by Boolean-attractor analyses. We can also use equation-reduction to find the late-time behavior at lower computational cost, which in the limit gives the long-term attractor dynamics *of the target variables*. This is a slightly different output than that produced by most existing Boolean analytic methods which generate the attractors of the entire network state. 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 other variables. 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 1 would require all 2^{40} state space variables. Thus our choice of variables is better for modeling very heterogeneous populations. Finally, our basis allows some variables to factorize others, allowing us to vastly simplify the calculation in many cases where we only care about the long-term behavior.

We acknowledge that our method can become intractable for complex networks due to the fact that the construction of these simulations is potentially an exponential problem. A full simulation can require up to 2^N equations to model; even the attractor analysis is known to be NP-hard [28]. Large size, complex logic rules, and certain feedback loops seem to cause the equation set to be large. These are fundamental limitations. However, the equation-reduction method for finding attractors is somewhat of an art, and our future work will aim to improve this part of the calculation for typical network models. Notably, our current implementation of the two equation-reduction methods often aborts due to numerical error exceeding bounds; improving the numerical stability will be a priority for future work.

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: correlation variables form a complete and independent basis

For N Boolean variables there are 2^N variables in the state space basis (labeled b): this is just the number of states of the system. Likewise there are 2^N variables in the product space basis (labeled x) because there are 2^N combinations of subscripts on these variables: each of N subscripts may be present or absent. Therefore the two spaces have the same number of variables, but this does not prove that the *product-basis* spans the entire Boolean space. In order to show that the x-variables form a complete basis in b-space, we imagine explicitly writing the transformation matrix from b-space to x-space. For example, for the case of three Boolean variables this matrix is:

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

The transformation matrix is upper-triangular, with ones on the diagonal. The reason is that each x-variable is turned on by the b-variable having the same indices (hence the ones along the diagonal), and by any b-variables containing additional indices implying a higher position in the matrix (since the indices are arranged in binary order of their subscripts); however an x-variable is never turned on by a b-variable that is *missing* one of its indices, which is why the lower triangular block is empty. Since the matrix is triangular the eigenvalues are the ones on the diagonal, so the determinant is one, the transformation is non-degenerate (and volume-preserving) and therefore the x-space spans the b-space.

Appendix 2: 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 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] Jonathan R Karr et al. "A whole-cell computational model predicts phenotype from genotype". *Cell* 150.2 (2012), pp. 389–401.

[3] 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.

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

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

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

[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] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

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

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

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

[19] 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.

[20] 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.

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

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

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

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

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

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

[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.