Periodic, Aperiodic, and Partly Periodic Clocks in Scientific Simulations

Clarence Lehman[1] **and Adrienne Keen**[2]
[1]University of Minnesota, 123 Snyder Hall, 1475 Gortner Avenue, Saint Paul, MN 55108, USA
[2]London School of Hygiene and Tropical Medicine, Keppel Street, London WC1E 7HT, UK

"Professor Einstein says that time differs from place to place...
If time is not true, what purpose have watchmakers?"
—Allen Moore, Dave Gibbons, 1986

Abstract— *In microscale simulations that forecast stochastic times for future events, most new events develop internally, either within a simulated entity or from interactions among entities in the simulation. Births, deaths, infections, migrations, and other events can arise internally in this way. However, some events arise exogenously, entirely outside the system, while others are triggered routinely by calendar times rather than by internal conditions. For example, in simulating the population of a region without its surrounding world, immigration of new individuals into the region would be exogenous, occurring at fixed or random intervals. For individuals in the simulation, the timing of medical checkups or other appointments could similarly occur at regular or irregular intervals, independently of other conditions in the simulation. Here we describe a way to implement clocks for such events, inspired by work on a large-scale epidemiological simulation program* [1]. *The clocks can tick deterministically or randomly following any probability distribution. Two forms of clocks, periodic and aperiodic, simulate natural processes such as oscillatory signals or radioactive decay. A third form, which we call partly periodic, does not typically occur in nature, but is devised to match empirical counts exactly. The design we describe is general and can be applied to any individual-based, agent-based, discrete event, or other microscale simulation model that stochastically schedules future events* [2].

Keywords: simulation clocks, microscale modeling, individual-based modeling, periodic events, aperiodic events, waiting-time paradox

1. Introduction

Microscale models simulate individuals directly rather than combining them into continuous fluids, probability distributions, or populations [3]. Individual-based and agent-based models are examples. Advances in computational power and methods now make microscale models competitive in simulating macroscale models defined as ordinary, partial, or integro-differential equations, as well as in simulating systems that cannot readily be formulated in macroscale terms. An efficient approach to microscale modeling schedules all events into the future rather than testing for events at each time step, and the existence of constant-time algorithms for managing schedules and groups of individuals [2] [4] [5] now allows hundreds of millions of individuals—from molecules to hayseeds to orca whales—to be tracked.

In this paper we describe our approach to "clocks" for events in large-scale simulations. We assume that a global timeline measures the flow of all events in the simulation. Any number of clocks may tick concurrently on this global timeline, with a new event scheduled at each tick of each clock. We explain the structure and dynamics of simple clocks, with algorithms for the clocks given in detail, and how collections of simple clocks may be combined into more elaborate custom clocks. The methods we describe may prove useful to scientists in ecology, epidemiology, economics, and other disciplines that employ individual-based, agent-based, discrete-event, or other forms of microscale modeling.

2. Events

Most events scheduled in microscale simulations arise endogenously from actions of individuals, conditions of individuals, or the environment of the system. For example, births in an ecological population model arise repeatedly as individual plants or animals reach appropriate age and condition. Infections in an epidemiological model arise when infectious individuals encounter susceptible individuals.

Yet some events are exogenous, not caused directly by actions within the system. Examples are seeds arriving on a simulated island from a mainland source, mutations triggered by external radioactive decay, and unpredictable fluctuations in the stock market. Other events are internal but more routine and temporal, occurring with little or no reference to the states of individuals or to interactions among individuals. Examples are scheduled checkups at a medical center, periodic events such as anniversaries, and random events like winning the lottery.

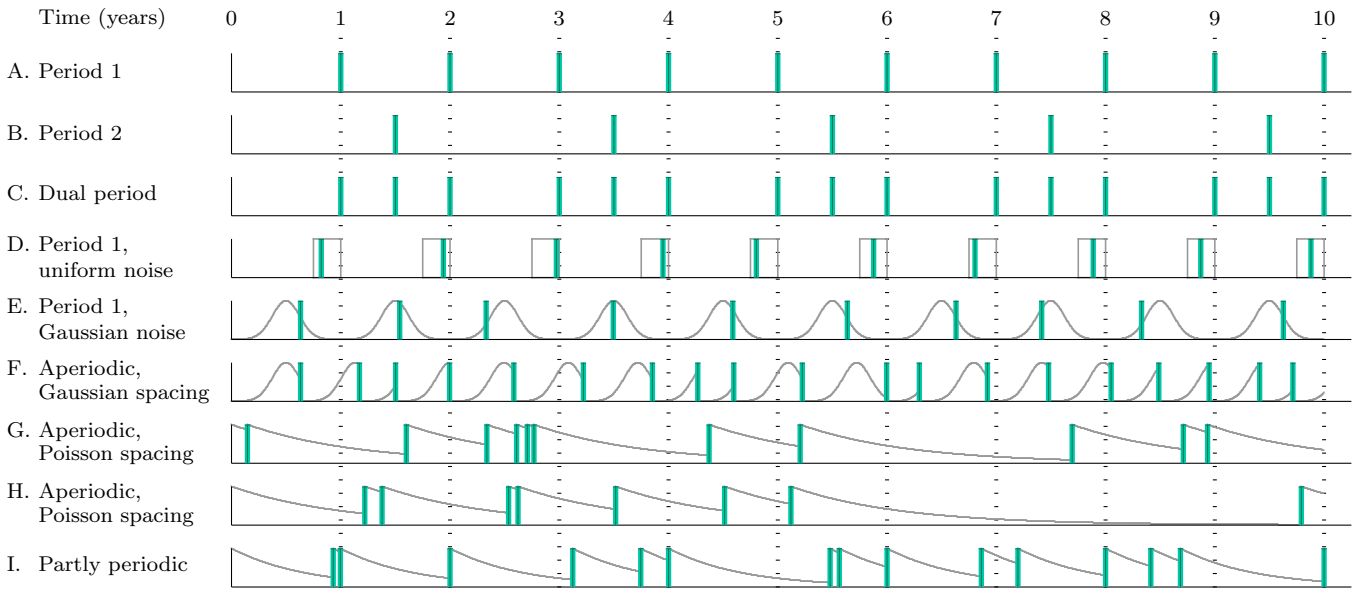

Figure 1. Illustration of clock patterns. *Clock ticks are shown with vertical bars on each graph for ten successive years.* **(A)** *Simple periodic clock with period 1 year.* **(B)** *Simple periodic clock with period 2, phase-shifted backwards 6 months.* **(C)** *The union of A and B above. Any combination of simpler clocks can form composite clocks.* **(D)** *Periodic clock with uniform random noise, with the tick anywhere in the last 3 months of the period. All points in that interval are equally likely, as outlined by the probability density function in gray.* **(E)** *Periodic clock with truncated Gaussian random noise, allowing the clock to tick anywhere within the period, but with the beginning and end of the period unlikely, as outlined by the probability density function in gray. Here* $\sigma = 1/8$ *year and* $\mu = 1/2,\ 3/2,\ 5/2, \ldots$ *years.* **(F)** *Aperiodic clock with the truncated Gaussian distribution of E. The probability evaluation restarts following each tick, making the aperiodic clock tick faster.* **(G)** *Simple aperiodic clock with mean time per tick of one year, where 11 ticks occurred in the time allotted to 10, due purely to random chance. A uniform random spacing of points (Poisson spacing) results from the exponential density function in gray.* **(H)** *Independent run of the same clock as in D, but where only 8 ticks occurred by random chance.* **(I)** *Partly periodic clock with 3 ticks distributed aperiodically every 2 years, the last of the 3 constrained to occur periodically at the end of the 2-year period.*

3. Periodic clocks

The simplest periodic clock ticks with complete regularity each time a fixed interval has elapsed. Starting at some initial time, t_0, it ticks at times $t_0+\varphi+\tau$, $t_0+\varphi+2\tau$, $t_0+\varphi+3\tau$, $\ldots$, $t_0+\varphi+n\tau$, and so forth ad infinitum, where τ is the period and φ is the phase.

Simple periodic clocks are established by *ClockPeriodic*, defined in the appendix, and provided with a clock number that identifies the data structure controlling the clock, an identification number denoting the event that is controlled by the clock, a period, and an optional phase shift. Such clocks can be established for any purpose—for example, to periodically examine conditions within the simulation like population growth rates, or to count out time intervals for a macroscale simulation of differential equations embedded within the microscale model.

The behavior of a simple periodic clock with period 1 is shown in Figure 1A. A similar simple periodic clock with period 2 is Figure 1B, shifted back from the end of its period by $1/2$ year ($\varphi = -0.5$). The union of these two clocks is Figure 2C. It ticks in the last three half-year intervals within the two-year period. Any collection of clocks can be combined into a single clock by assigning them the same identification number.

Figures 1D and 1E are periodic clocks with random fluctuations superimposed. Such fluctuations can arise from any probability distribution, which is supplied when the clock is started. Figure 1D is uniform random noise that allows a tick anywhere in the last quarter-period. That noise is supplied by the probability distribution shown in Figure 3A, where the cumulative probability P is 0 until $t = 9$ months, then rises linearly to 1 at $t = 12$ months. Figure 1E is similar, but with truncated Gaussian noise centered at the middle of the period. That noise is supplied by a cumulative probability distribution that rises sigmoidally from $P = 0$ at $t = 0$ to $P = 1$ at $t = 12$ months, as shown in Figure 3B. These cumulative distributions are represented by piecewise linear or higher-order approximations and passed to a random number generator that can work with arbitrary probability distributions [6].

Periodic example, birthday party clock. For an initial intuitive example, think about a "birthday party clock." A birthday begins regularly at midnight on a certain day of the year—a simple periodic clock. However, the celebration may be a few days early or a few days late. Nonetheless, the timing of the celebration does not change the timing of the birthday for subsequent years. That is, the celebration will always be synchronized with the calendar, not drifting

over time. This is therefore a periodic clock, with random noise for the timing of the celebration. (Note, however, that accounting for leap years considerably complicates the clock.)

Periodic example, macroscale simulation clock. Suppose a population is being simulated for which the microscale dynamics are not known, or for which there is not precise empirical data. Or suppose the dynamics are well known but the population is large enough that stochasticity tends to cancel out, so a microscale simulation of it is unnecessary. In such cases a macroscale population model may be embedded within a microscale simulation. For example, suppose a microscale model of the bacteria and individual parasitic ticks on populations of wild field mice is used to understand wildlife epidemiology. Also suppose that the behavior of the mouse population is not of interest, but is only necessary to provide background for the epidemiological part of the simulation. In that case the mouse population could be simulated with a known differential equation model embedded within the microscale model. A periodic clock can be used to count the time intervals of a macroscale differential equation solver—by Euler's method, Runge–Kutta, or other integration technique [7]. See Figure 2.

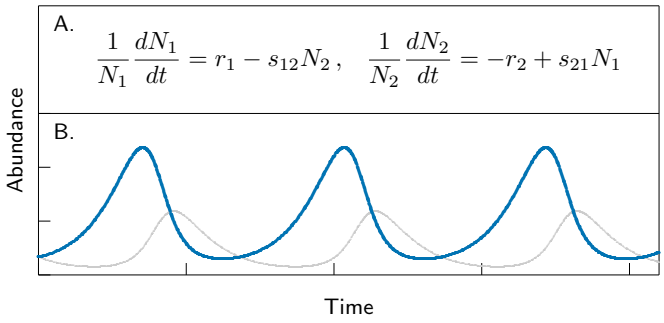

Figure 2. *A macroscale model embedded within a microscale model. Here a simple periodic clock supplies the time steps, Δt_i, to drive a differential equation simulator for population dynamics that do not need to be simulated by individuals within the microscale model.* **(A)** *The macroscale model to be embedded. This is an ecological Lotka–Volterra predator–prey model.* **(B)** *Simulated trace of the macroscale model, with $r_1{=}r_2{=}1$, $s_{12}{=}1$, and $s_{21}{=}2$. The abundance of prey $N_1(t)$, bold curve, can be used to drive individual births and deaths of a microscale prey species. The abundances of predator $N_2(t)$, light curve, are only to induce the bold curve and would not be needed further in the microscale simulation.*

4. Aperiodic clocks

The simplest aperiodic clock is the exact opposite of periodic. It ticks with complete irregularity, with all instants equally likely to see a tick (exponential delay), and with the probability of a tick in an instant defined by an average number of ticks per time unit. Starting at some initial time, t_0, it ticks at times $t_0+\tau_1$, $t_0+\tau_2$, $t_0+\tau_3$, $\ldots$, $t_0+\tau_4$, and so forth ad infinitum, where the random τ_i are uniformly distributed across time.

The behavior of a simple aperiodic clock ticking on average once per time unit is shown in Figures 1G and 1H. The first example ticks eleven times and the second ticks only eight, due solely to random variation. In the simplest aperiodic clock, the number of ticks per time unit follows a Poisson distribution and the time between ticks follows a corresponding exponential distribution [8]. In this case, the probability of exactly the expected number of ticks, 10, is only about 1 out of 8. (From the Poisson density function, $p(k) = k^\lambda \cdot e^{-\lambda}/k! = 10^{10} \cdot e^{-10}/10! \approx 0.125$.) Notice in Figures 1G and 1H how irregularly such a clock behaves.

The behavior of a related aperiodic clock appears in Figure 1F. It has the same probability distribution governing its ticks as the periodic clock in 1E above it. However, that probability distribution restarts at each tick. Notice how much more regular the ticks are in Figure 1F than in 1G and 1H, even though they are still aperiodic.

Aperiodic clocks are established by *ClockAperiodic*, defined in the appendix, and provided with a mean time between ticks or an optional probability distribution. Such clocks can simulate purely random processes such as radioactive decay, but also can approximate other events such as successive times of transmission in a population with infectious individuals.

Aperiodic example, hair appointment clock. Suppose that an individual is scheduled for a haircut every month, but that the haircut is occasionally delayed, due to negligence or other causes. If for some reason three months have elapsed between haircuts, most certainly is not necessary for the individual to have three haircuts in rapid succession to make up for the haircuts that were missed. The timing of the next haircut restarts at the time of the last, and occurs again at some average time in the future. That is an aperiodic clock.

5. Partly periodic clocks

Starting at some initial time, t_0, a partly periodic clock ticks at times $t_0+\tau$, $t_0+2\tau$, $t_0+3\tau$, $\ldots$, $t_0+n\tau$, and so forth, where τ is the period. In addition, however, it ticks aperiodically $k-1$ times in between each periodic tick, giving an average time between ticks of τ/k every period.

Figure 1I shows a partly periodic clock of period 2, with a mean time between ticks of $2/3$. One periodic tick and two aperiodic ticks occur in each period of the clock, with the spacing between all three ticks matching an exponential distribution. That exponential distribution decays more rapidly in Figure 1I than in 1F and 1H, because the mean time between ticks in 1F and 1H is 1 time unit, while in 1I it is only $2/3$ time unit.

Partly periodic example, immigration clock. Suppose a local population is being simulated and the number of immigrants each year is taken from known historical accounts. Suppose new individuals can arrive in the population at any random time of the year, but the number of immigrants each

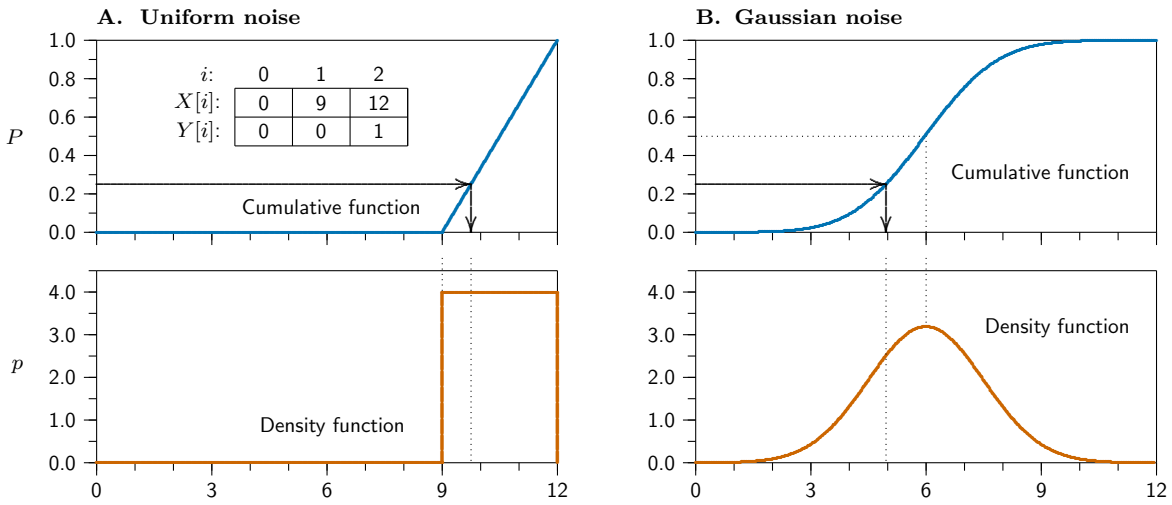

Figure 3. Random noise specifications. *Horizontal axes represent time, here marked in months. The vertical axis for cumulative functions represents the probability that the random variable is less than or equal to the corresponding value on the horizontal axis. The vertical axis for density functions, multiplied by the width of a small interval on the horizontal axis, represents the probability that a random number falls within that interval. Arrows demonstrate how random noise is generated, from a uniform random number located on the vertical axis, then followed to the right to the cumulative function, then down to the axis to pick a random number from the desired distribution* [6]. **(A)** *Uniform noise in the last 4 months of the year. Inset shows a tabular form of the cumulative distribution, in this case where three months on the horizontal axis,* $X[i] = \{0, 9, 12\}$*, correspond to three probabilities on the vertical axis,* $Y[i] = \{0, 0, 1\}$*.* **(B)** *Truncated Gaussian noise with mean 6 and variance* $9/4$*. The tabular form would be similar to that of Part A, but with several hundred small steps to approximate a continuous distribution, and optionally with nonlinear smoothing* [6].

year is fixed to exactly match historical records. This is not a periodic clock because immigrants can appear randomly throughout the year. However, it is not an aperiodic clock either, for in an aperiodic clock, the number of ticks in a given time interval cannot be specified exactly. This is a partly periodic clock. It is not intended to be an analog of a natural process, but rather to match empirical data, for which precise values are known at regular intervals.

6. Algorithms

Each clock is defined and tracked as one entry in an array $A[n]$ that stores the information associated with every individual in the simulation, as described in the appendix and elsewhere [2]. That array ordinarily records all future events for each individual, with the earliest of those events recorded in a global list of future events. Adding, deleting, and accessing events uses a constant amount of time regardless of how many events are in the global list [2].

However, for clocks, which act as "pseudo-individuals" in the simulation, data elements within the entry are used differently than they are used for individuals. Future times for each clock are recorded not explicitly but algorithmically, as described above and detailed in the appendix.

For periodic clocks, the algorithm records the period τ, the phase shift φ, and the number of ticks n that have occurred since the time t_0 that the clock started ticking. That allows the next time to be calculated as $t_0+(n+1)\tau+\varphi$. Multiplying the period by the number of elapsed ticks avoids cumulative drift due to rounding error, as would occur if the time of the previous tick were incremented by the period.

Aperiodic clocks are easier, since there is no starting time, phase shift, or past number of ticks to be tracked. These clocks have no memory of what has happened in the past. That makes their implementation simpler, even though understanding the dynamics of their behavior is more difficult (see discussion section below).

Partly periodic clocks are the most difficult. They require an entire sequence of k ticks to be remembered, so that all k can be rescaled precisely to fit within the period τ, with the last tick occurring at the last moment of the period. This could be accomplished with a quantity of memory proportional to k, to record the ticks in advance, but it can also be accomplished with a constant quantity instead. Deterministic random number generators have a state variable [9] that allows any subsequence of pseudo-random numbers to be regenerated. The algorithm for partly periodic clocks first runs through k ticks to determine how much total time $\mathcal{K}$ they would take. It then reruns the sequence one tick at a time, at each tick rescaling by $\tau/\mathcal{K}$.

All three kinds of clocks are proportional to n for speed and independent of n for memory. They are fully defined in the appendix, embodied in four algorithms.

1. *ClockPeriodic* Starts a periodic clock.
2. *ClockAperiodic* Starts an aperiodic clock.
3. *ClockPartlyPeriodic* Starts a partly periodic clock.
4. *ClockTick* Schedules another tick of any clock.

In addition, *T1*, *T2*, and *T3* are subroutines of *ClockTick* to implement the three kinds of clocks.

7. Discussion

Within the dynamics of periodic and aperiodic clocks, a curious observational paradox arises that affects measurements by individuals or agents within the simulation. It is helpful to understand this paradox when working with simulation clocks.

Suppose you have a simple periodic clock ticking precisely every ten simulated minutes and a matching simple aperiodic clock ticking on average at the same rate. Over the course of time both of these clocks tick equally often. Suppose each clock represents some service that individuals in the simulation occasionally wait for, such as catching a bus or being served at a medical emergency center. What will be the average interval between events, with an event occurring on average every ten simulated minutes, as measured by individuals within the simulation?

For a periodic clock, if a simulated individual arrives at a completely random time, independent of the ticking of the clock, that individual will observe an average of ten minutes from the time of the previous tick until the clock ticks again, since the clock is perfectly periodic. It would seem at first glance that it should be similar for an aperiodic clock—that an individual within the simulation would observe an average interval between ticks of ten minutes, since that is the average rate of the clock. But that is incorrect. For a completely random aperiodic clock ticking once every ten minutes on average, all individuals will observe an average time between ticks of not ten but of twenty minutes—half the speed of the actual clock! This effect also doubles the average time for each individual waiting for an aperiodic versus a periodic clock.

Such dynamics must be taken into account in auditing the performance of simulations, and even in designing real systems for use by living individuals. The inflated waiting time is not an illusion nor a property of computer simulation; it occurs in all aperiodic events, tangible or abstract. Upon first learning of this phenomenon, people are usually incredulous. Feller calls it the "waiting time paradox." He assures that although you may be shocked when you first encounter it, "after due reflection the difference becomes intuitively obvious." [8]

The resolution of the paradox lies in a "time line" that extends back to the past. If you picked a time at random on that line, you will have been more likely to have found yourself in a longer interval than a shorter one, simply because longer intervals contain a greater measure of time points than shorter ones. The intervals of a simple aperiodic clock vary widely, as illustrated in Figures 1G and 1H. When the exponential density function of 1G and 1H is integrated over all possible interval lengths and probabilities, the result of twice the average time between ticks emerges. With aperiodic clocks whose ticks are more aggregated than random, the observed interval between ticks can be arbitrarily long.

An understanding of this phenomenon and a review of how it applies in simulations is important so that aggregation of aperiodic events can be controlled in real systems to reduce actual waiting times there.

8. Conclusions

Many events arising within microscale simulations are simple enough to be handled by "clocks" of standard design. Three kinds of clocks—periodic, aperiodic, and partly periodic—cover a diversity of situations. All three require computing time proportional to the number of ticks and memory independent of the number of ticks.

9. Acknowledgements

We are grateful to Tendai Mugwagwa and Peter White for initial discussions leading to the design of these clocks, and to Todd Lehman, Shelby Williams, Katie Hoffman, and the anonymous reviewers for help with the presentation. This project was supported in part by a resident fellowship grant to C. Lehman from the UMN Institute on the Environment, by grants of computer time from the Minnesota Supercomputing Institute, and by doctoral research funding to A. Keen from the Modelling and Economics Unit at Public Health England, formerly Health Protection Agency, London.

References

[1] A. Keen, "Understanding tuberculosis dynamics in the United Kingdom using mathematical modelling," *Doctoral Thesis, London School of Hygiene and Tropical Medicine*, p. 493pp, 2013.

[2] C. Lehman, A. Keen, and R. Barnes, "Trading space for time: Constant-speed algorithms for managing future events in scientific simulations," *Proceedings, International Conference on Scientific Computing*, vol. CSC12, p. 8 pp, 2012.

[3] L. Gustafsson and M. Sternad, "Consistent micro, macro and state-based population modelling," *Mathematical Bioscience*, vol. 225, pp. 94–107, 2010.

[4] R. Brown, "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, pp. 1220–1227, 1988.

[5] A. Keen and C. Lehman, "Trading space for time: Constant-speed algorithms for grouping objects in scientific simulations," *Proceedings, International Conference on Scientific Computing*, vol. CSC12, pp. 146–151, 2012.

[6] C. Lehman and A. Keen, "Efficient pseudo-random numbers from any probability distribution," *Proceedings, International Conference on Modeling, Simulation, and Visualization Methods*, vol. MSV12, pp. 121–127, 2012.

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical recipes: The art of scientific computing, Third Edition," *Cambridge University Press, New York*, 2007.

[8] W. Feller, "An introduction to probability theory and its applications, Volume II, Second Edition," *John Wiley & Sons, New York*, 1971.

[9] D. E. Knuth, "The art of computer programming, volume 2: Seminumerical algorithms, third edition," *Addison-Wesley, Reading, MA*, 1997.

10. Appendix

To use the algorithms described in this paper, it is only necessary to understand the entry and exit conditions that appear at the beginning of each subroutine, not the code itself. Nonetheless, to allow complete evaluation of the algorithms, and to encourage further development of them, we present them as pseudo-code inspired by and simplified from the programming languages C, R, and Python. The algorithms are defined with sufficient precision that they can be run, tested, timed, modified, or translated to other languages. Familiarity with a few operators* and with the syntax of flow control (if, for, while, etc.), is sufficient to follow the algorithms. External routine *EventSchedule*(n, τ) schedules a future event for $A[n]$ at time τ [2], *Rand* generates uniform pseudo-random numbers $0 \leq \rho < 1$, *RandF* is an interface for other distributions, *RandSeed* retrieves the current state of the random sequence, *RandSeq*(q) resets the sequence to a new or previous state q, and *Cinverse* converts to any probability distribution [6]. These algorithms translated into operational C are available free from the authors upon request.

DATA ELEMENTS

Clock parameters are stored in array $A[n]$, which is also used for individuals in the simulation. The main simulation program keeps track of which values of n represent clocks, either by assigning them to fixed positions of the array or by placing them in a group [5] reserved for clocks. The elements of $A[n]$ must be large enough to represent both clocks and individuals.

Variables in $A[n]$ are assigned names beginning with V, as follows.

Vid	$\equiv A[n][0]$	Identification for this specific clock.
Vtype	$\equiv A[n][1]$	Type of clock (periodic, aperiodic, etc).
Vper	$\equiv A[n][2]$	Time units per tick or set of ticks.
Vbase	$\equiv A[n][3]$	Base time (periodic or aperiodic).
Vstep	$\equiv A[n][4]$	Number of steps presently beyond the base.
Vticks	$\equiv A[n][5]$	Number of ticks per period (partly periodic).
Vk	$\equiv A[n][6]$	Ticks thus far in the period (partly periodic).
Vscale	$\equiv A[n][7]$	Scaling factor for the period (partly periodic).
Vseed	$\equiv A[n][8]$	Random number seed (partly periodic).
Vx	$\equiv A[n][9]$	Cumulative distribution, 'x' variable.
Vy	$\equiv A[n][10]$	Cumulative distribution, 'y' variable.
Vnxy	$\equiv A[n][11]$	Number of elements in 'x' and 'y'.

CLOCK TICK

Routine *ClockTick* is called after each tick of a clock, typically from a dispatching routine in the main simulation program, to schedule the next tick. It is also called once a new clock is started, to schedule the first tick. **Upon entry, (1)** $A[n]$ records the state of the clock, as defined in the exit conditions of *ClockPeriodic*, *ClockAperiodic*, and *ClockPartlyPeriodic* below. In particular, **(2)** $Vtype$ defines the type of clock, 1=periodic, 2=aperiodic, 3=partly periodic. **On exit,** the next tick is scheduled.

ClockTick(n) **integer** n;
  choose from *Vtype*:
    case 1: *T1*(n); **return**; Periodic clock.
    case 2: *T2*(n); **return**; Aperiodic clock.
    case 3: *T3*(n); **return**; Partly periodic clock.
    other: *ExitMsg*(1); **return**; Improper clock type.

* *In the pseudo-code here, indentation defines the nested structure. Variables and function names are italicized and flow control and reserved words are bolded. Assignment is left to right, represented by '$\rightarrow$'. Individual parts of any compound assignments also operate left to right, so that '$a + 1 \rightarrow a \rightarrow b \rightarrow W[i][j]$' operates by first incrementing a and placing the results back in a, then in b, and then in the i, jth element of array W. Array indexing starts with 0. Any logical operators such as 'and' and 'or' are preemptive, terminating a chain of logical operations as soon as the result is known. Permanent global assignments are rendered '$\alpha \equiv \beta$'.*

Algorithm 1. PERIODIC CLOCK

Routine *ClockPeriodic* establishes a new periodic clock or alters an existing one. **Upon entry, (1)** $A[n]$ is an entry available for controlling the clock. **(2)** id contains the identification number for the clock. **(3)** $period$ is the period of the clock, greater than 0. **(4)** $phase$ is phase shift—the position of the periodic tick within the period. **(5)** x, y, and nxy optionally define a probability distribution for the noise, in the form of $RandF$. If x is null, no probability distribution is supplied and no noise is applied. **(6)** $Anull$ is an entry of $A[n]$ that has all variables zero. **(7)** t is the current time. **On exit, (1)** $A[n]$ is prepared so that clock ticks can be scheduled by a call to $ClockTick$. In particular, the following elements are set. **(2)** $Vtype$ is 1, meaning a periodic clock. **(3)** Vid carries the value of id on entry. **(4)** $Vperiod$ contains the value of $period$ on entry. **(5)** $Vbase$ contains the base time for the clock, equal to t on entry plus the value of $phase$ on entry. **(6)** Vx, Vy, and $Vnxy$ contain the values of x, y, and nxy on entry, respectively. These may be null if no distribution was supplied. **(7)** All other elements are zero.

ClockPeriodic(n, *id*, *period*, *phase*, $x[\,]$, $y[\,]$, *nxy*) **integer** n, *id*, *nxy*; **real** *period*, *phase*, $x[\,]$, $y[\,]$;

Anull $\rightarrow A[n]$; 1 $\rightarrow$ *Vtype*;	1. Prepare a new table entry.
id $\rightarrow$ *Vid*; *period* $\rightarrow$ *Vper*; $t +$ *phase* $\rightarrow$ *Vbase*;	2. Save entry parameters.
$x \rightarrow Vx$; $y \rightarrow Vy$; *nxy* $\rightarrow$ *Vnxy*;	3. Save any custom random function.

Routine *T1* is called each time a periodic clock has ticked, to schedule the next tick. It is also called immediately after *ClockPeriodic* has been called, to start the clock ticking. **Upon entry, (1)** $A[n]$ defines the clock structure. In particular, the following elements are relevant. **(2)** Vid contains the identification number for the clock. **(3)** $Vper$ contains the period. **(4)** $Vbase$ contains the base time for the clock, equal to the starting time plus the phase shift. **(5)** $Vstep$ contains the number of ticks thus far. **(6)** Vx, Vy, and $Vnxy$ optionally define a probability distribution for the noise, in the form of $RandF$. If Vx is null, no probability distribution is supplied and no noise is applied. **(7)** t is the current time. **On exit, (1)** The next tick of the clock is scheduled. **(2)** $Vstep$ is advanced by the 1.

T1(n) **integer** n;

if *Vx*: *RandF*(*Vx*, *Vy*, *Vnxy*) $\rightarrow w$, (w/*Vx*[*Vnxy* − 1])∗*Vper* $\rightarrow w$;	1. If random noise has been specified, select a tick from within that noise.
else *Vper* $\rightarrow w$;	2. Otherwise generate a precise tick.
Vbase + (*Vstep*∗*Vper*) + $w \rightarrow v$;	3. Compute the time of the tick.
Vstep + 1 $\rightarrow$ *Vstep*;	4. Advance the tick counter.
if $v < t$: $t \rightarrow v$;	5. Control any rounding error.
EventSchedule(n, v);	6. Schedule the next tick.

Algorithm 2. APERIODIC CLOCK

Routine *ClockAPeriodic* establishes a new aperiodic clock or alters an existing one. **Upon entry, (1)** $A[n]$ is an entry available for controlling the clock. **(2)** id contains the identification number for the clock. **(3)** $mean$ optionally defines the time between ticks in an exponential probability distribution, if a custom probability distribution is not supplied. **(4)** x, y, and nxy optionally define a custom probability distribution for the waiting time to the next tick, in the form of $RandF$. If x is null, no custom distribution is supplied and an exponential distribution is used instead. **(5)** $Anull$ is an entry of $A[n]$ that has all variables zero. **On exit, (1)** $A[n]$ is prepared so that clock ticks can be scheduled by a call to $ClockTick$. In particular, the following elements are set. **(2)** $Vtype$ is 2, meaning an aperiodic clock. **(3)** Vid carries the value of id on entry. **(4)** $Vper$ contains the value of $mean$ on entry. **(5)** Vx, Vy, and $Vnxy$ contain the values of x, y, and nxy on entry, respectively. These may be null if no custom distribution was supplied. **(6)** All other elements are zero.

ClockAperiodic(n, *id*, *mean*, $x[\,]$, $y[\,]$, *nxy*) **integer** n, *id*, *nxy*; **real** *mean*, $x[\,]$, $y[\,]$;

Anull $\rightarrow A[n]$; 2 $\rightarrow$ *Vtype*;	1. Prepare a new table entry.
id $\rightarrow$ *Vid*; *mean* $\rightarrow$ *Vper*;	2. Save entry parameters.
$x \rightarrow Vx$; $y \rightarrow Vy$; *nxy* $\rightarrow$ *Vnxy*;	3. Save any custom random function.

Routine *T2* is called each time an aperiodic clock has ticked, to schedule the next tick. It is also called immediately after *ClockAperiodic* has been called, to start the clock ticking. **Upon entry, (1)** $A[n]$ defines the clock structure. In particular, the following elements are relevant. **(2)** Vid contains the identification number for the clock. **(3)** $Vper$ contains the mean time between ticks in an exponential distribution if Vx, Vy, and $Vnxy$ do not specify a custom distribution. **(4)** Vx, Vy, and $Vnxy$ optionally define a custom probability distribution for the waiting time to the next tick, in the form of $RandF$. If Vx is null, no custom distribution is supplied and an exponential distribution is used instead. **(5)** t is the current time. **On exit,** The next tick of the clock is scheduled.

T2(n) **integer** n;

if *Vx*: *RandF*(*Vx*, *Vy*, *Vnxy*) $\rightarrow w$;	1. Generate the time until the next tick, from a custom distribution or an exponential.
else *Exponential*(*Vper*) $\rightarrow w$;	
EventSchedule(n, $t + w$);	2. Add current time to schedule the next tick.

Algorithm 3. PARTLY PERIODIC CLOCK

Routine *ClockPartlyPeriodic* establishes a new partly periodic clock or alters an existing one. **Upon entry, (1)** $A[n]$ is an entry available for controlling the clock. **(2)** id contains the identification number for the clock. **(3)** $period$ defines the period of the periodic part of the clock. **(4)** $ticks$ defines the number of ticks that will occur aperiodically during that period, spaced according to the dictates of the relevant probability distribution. **(5)** x, y, and nxy optionally define a custom probability distribution for the waiting time to the next tick, in the form of $RandF$. If x is null, then no custom distribution is supplied and an exponential distribution is used instead. **(6)** $Anull$ is an entry of $A[n]$ that has all variables zero. **(7)** t is the current time. **On exit, (1)** $A[n]$ is prepared so that clock ticks can be scheduled by a call to $ClockTick$. In particular, the following elements are set. **(2)** $Vtype$ is 3, meaning a partly periodic clock. **(3)** Vid carries the value of id on entry. **(4)** $Vper$ contains the value of $period$ on entry. **(5)** $Vticks$ contains the number of ticks to occur in that period. **(6)** $Vbase$ contains the value of t on entry. **(7)** Vx, Vy, and $Vnxy$ contain the values of x, y, and nxy on entry, respectively. **(8)** All other elements are zero.

ClockPartlyPeriodic(n, *id*, *period*, *ticks*, $x[\,]$, $y[\,]$, *nxy*) **integer** n, *id*, *nxy*, *ticks*; **real** *period*, $x[\,]$;

Anull $\rightarrow A[n]$; 3 $\rightarrow$ *Vtype*;	1. Prepare a new table entry.
id $\rightarrow$ *Vid*; *period* $\rightarrow$ *Vper*; *ticks* $\rightarrow$ *Vticks*; $t \rightarrow$ *Vbase*;	2. Save entry parameters.
$x \rightarrow$ *Vx*; $y \rightarrow$ *Vy*; *nxy* $\rightarrow$ *Vnxy*;	3. Save any custom random function.

Routine *T3* is called each time a partly periodic clock has ticked, to schedule the next tick. It is also called immediately after *ClockPartlyPeriodic* has been called, to start the clock ticking. **Upon entry, (1)** $A[n]$ defines the clock structure. In particular, the following elements are relevant. **(2)** Vid contains the identification number for the clock. **(3)** $Vper$ contains the $period$. **(4)** $Vticks$ contains the number of ticks to occur in that period. **(5)** $Vbase$ contains the starting time. **(6)** $Vstep$ contains the number of ticks thus far. **(7)** Vx, Vy, and $Vnxy$ optionally define a custom probability distribution for the waiting time to the next tick, in the form of $RandF$. If x is null, then no custom distribution is supplied and an exponential distribution is used instead. **(8)** t is the current time. **On exit, (1)** The next tick of the clock is scheduled. **(2)** $Vstep$ is advanced by the 1.

T3(n) **integer** n;

if *Vk* $= 0$:
  Vseed $\leftarrow$ *RandSeed*(); 0 $\rightarrow$ *Vscale*;
  loop $Vticks$ **times**:
  if *Vx*: *Vscale* + *RandF*(*Vx*, *Vy*, *Vnxy*) $\rightarrow$ *Vscale*;
  else *Vscale* + *Exponential*(1) $\rightarrow$ *Vscale*;
	1. At the beginning of an aperiodic section, run through the sequence to determine how much it has to be up or down-scaled to match the specific period.

Vk + 1 $\rightarrow$ *Vk*; **if** *Vk* $\geq$ *Vticks*: 0 $\rightarrow$ *Vk*;	2. Advance the subperiod counter.
RandSeed() $\rightarrow q$; *RandSeq*(*Vseed*); **if** *Vx*: *RandF*(*Vx*, *Vy*, *Vnxy*) $\rightarrow w$; **else** *Exponential*(1) $\rightarrow w$; *RandSeed*() $\rightarrow$ *Vseed*; *RandSeq*(q);	3. Step again through the previous random sequence to generate the next tick, without disturbing the current sequence.
Vbase + (*Vstep*$*$*Vper*) $\rightarrow$ *base*;	4. Locate the beginning of the period.
if *base* $<$ *t*: $t \rightarrow$ *base*;	5. Control any rounding error.
Vstep + 1 $\rightarrow$ *Vstep*;	6. Locate the next periodic point.
EventSchedule(n, $t + (w/$*Vscale*$)*$*Vper*);	7. Schedule the next tick.

RANDOM NUMBER INTERFACE

Routine *RandF* connects the internal routines represented here with the external random number routine *Cinverse* [6], which generates random numbers from any probability distribution. It uses two arrays to define the distribution, one for the x-axis and one for the y-axis. For example, the distribution of Figure 3A would have $x[0, 1, 2] = \{0, 9, 12\}$, $y[0, 1, 2] = \{0, 0, 1\}$, and $n = 3$, meaning the cumulative distribution remains 0 between $x = 0$ and $x = 9$, then rises linearly to 1 at $x = 12$. **Upon entry, (1)** x is an array of values in the set of random numbers to be generated. **(2)** y is an array of cumulative probabilities, each being the probability that a random value will be less than or equal to the corresponding value in x. **(3)** n is the number of entries in tables x and y. **On exit,** $RandF$ contains a random value drawn from the given distribution.

RandF(x, y, n) **integer** n; **real** $x[\,]$, $y[\,]$;
  return *Cinverse*(1, *Rand*(), $x[0]$, n, x, y, 0);