juliacon

Extending JumpProcess.jl for fast point process simulation with time-varying intensities

Guilherme Augusto Zagatti[1], Samuel A. Isaacson[3], Christopher Rackauckas[4], Vasily Ilin[5], See-Kiong Ng[1, 2], and Stéphane Bressan[1, 2]

[1]Institute of Data Science, National University of Singapore, Singapore
[2]School of Computing, National University of Singapore, Singapore
[3]Department of Mathematics and Statistics, Boston University
[4]Computer Science and AI Laboratory (CSAIL), Massachusetts Institute of Technology
[5]Department of Mathematics, University of Washington

ABSTRACT

Point processes model the occurrence of a countable number of random points over some support. They can model diverse phenomena, such as chemical reactions, stock market transactions and social interactions. We show that `JumpProcesses.jl` is a fast, general-purpose library for simulating point processes. `JumpProcesses.jl` was first developed for simulating jump processes via stochastic simulation algorithms (SSAs) (including Doob's method, Gillespie's methods, and Kinetic Monte Carlo methods). Historically, jump processes have been developed in the context of dynamical systems to describe dynamics with discrete jumps. In contrast, the development of point processes has been more focused on describing the occurrence of random events. In this paper, we bridge the gap between the treatment of point and jump process simulation. The algorithms previously included in `JumpProcesses.jl` can be mapped to three general methods developed in statistics for simulating evolutionary point processes. Our comparative exercise revealed that the library initially lacked an efficient algorithm for simulating processes with variable intensity rates. We, therefore, extended `JumpProcesses.jl` with a new simulation algorithm, `Coevolve`, that enables the rapid simulation of processes with locally-bounded variable intensity rates. It is now possible to efficiently simulate any point process on the real line with a non-negative, left-continuous, history-adapted and locally bounded intensity rate coupled or not with differential equations. This extension significantly improves the computational performance of `JumpProcesses.jl` when simulating such processes, enabling it to become one of the few readily available, fast, general-purpose libraries for simulating evolutionary point processes.

1. Introduction

Methods for simulating the trajectory of evolutionary point processes can be split into exact and inexact methods. Exact methods describe the realization of each point in the process chronologically. This exactness avoids bias from numerical approximations, but such methods can suffer from reduced performance when simulating systems with large populations (where numerous events can fire within a short period since every single point needs to be accounted for). Inexact methods trade accuracy for speed by simulating the total number of events in successive intervals. They are popular in biochemical applications, e.g. τ-leap methods [4], which often require the simulation of chemical reactions in systems with large molecular populations.

Previously, point process simulation library development focused primarily on univariate processes with exotic intensities, or large systems with conditionally constant intensities, but not on both. As such, there was no widely used general-purpose software for efficiently simulating compound point processes in large systems with time-dependent rates. To enable the efficient simulation of such processes, we contribute the `Coevolve` aggregator to `JumpProcesses.jl`, a core component of the popular `DifferentialEquations.jl` library [17]. The implemented algorithm improves the COEVOLVE algorithm described in [2] from where it borrows its name. Among other improvements, our algorithm supports any process with locally bounded conditional intensity rates, adapts to intensity rates that can change between jumps, can be coupled with differential equations, and avoids both the unnecessary re-computation of randomly generated numbers and the computation of the intensity rate when its lower bound is available. This extension of `JumpProcesses.jl` dramatically boosts the computational performance of the library in simulating processes with intensities that have an explicit dependence on time and/or other continuous variables, significantly expanding the type of models that can be efficiently simulated. Widely-used point processes with such intensities include compound inhomogeneous Poisson, Hawkes, and stress-release processes — all described in [1]. Since `JumpProcesses.jl` is a member of Julia's SciML organization, it also becomes easier, and more feasible, to incorporate compound point processes with explicit time-dependent rates into a wide variety of applications and higher-level analyses. With our new additions we bump `JumpProcesses.jl` to version 9.7[1].

In this paper, we bridge the gap between simulation methods developed in statistics and biochemistry, which led us to the development of `Coevolve`. First, we briefly introduce evolutionary point processes. Next, since all simulation methods require a basic under-

[1]All examples and benchmarks in this paper use this version of the library

standing of simulation methods for the Poisson homogeneous process, we first describe such methods. Then, we identify and discuss three general, exact methods. In the second part of this paper, we describe the algorithms in `JumpProcesses.jl` and how they relate to the literature. We highlight our contribution `Coevolve`, investigate the correctness of our implementation and provide performance benchmarks to demonstrate its value. The paper concludes by discussing potential improvements.

2. The evolutionary point process

The evolutionary point process is a stochastic collection of marked points over a one-dimensional support. They are exhaustively described in [1]. The likelihood of any evolutionary point process is fully characterized by its conditional intensity,

$$\lambda^*(t) \equiv \lambda(t \mid H_{t^-}) = \frac{p^*(t)}{1 - \int_{t^-}^{t_n} p^*(u)\, du}, \tag{2.1}$$

and conditional mark distribution, $f^*(k|t)$ — see Chapter 7 [1]. Here $H_{t^-} = \{(t_n, k_n) \mid 0 \leq t_n \leq t\}$ denotes the internal history of the process up to but not including t, the superscript $*$ denotes the conditioning of any function on H_{t^-}, and $p^*(t)$ is the density function corresponding to the probability of an event taking place at time t given H_{t^-}. We can interpret the conditional intensity as the likelihood of observing a point in the next infinitesimal unit of time, given that no point has occurred since the last observed point in H_{t^-}. Lastly, the mark distribution denotes the density function corresponding to the probability of observing mark k given the occurrence of an event at time t and internal history H_{t^-}.

3. The homogeneous process

A homogeneous process can be simulated using properties of the Poisson process, which allow us to describe two equivalent sampling procedures. The first procedure consists of drawing successive inter-arrival times. The distance between any two points in a homogeneous process is distributed according to the exponential distribution — see Theorem 7.2 [9]. Given the homogeneous process with intensity λ, then the distance Δt between two points is distributed according to $\Delta t \sim \exp(\lambda)$. Draws from the exponential distribution can be performed by drawing from a uniform distribution in the interval $[0, 1]$. If $V \sim U[0, 1]$, then $T = -\ln(V)/\lambda \sim \exp(1)$. (Note, however, in Julia the optimized Ziggurat-based method used in the `randexp` stdlib function is generally faster than this *inverse* method for sampling a unit exponential random variable.) When a point process is homogeneous, the *inverse* method of Subsection 4.1 reduces to this approach. Thus, we defer the presentation of this Algorithm to the next section.

The second procedure uses the fact that Poisson processes can be represented as a mixed binomial process with a Poisson mixing distribution — see Proposition 3.5 [9]. In particular, the total number of points of a Poisson homogeneous process in $[0, T)$ is distributed according to $\mathcal{N}(T) \sim \text{Poisson}(\lambda T)$ and the location of each point within the region is distributed according to the uniform distribution $t_n \sim U[0, T]$.

4. Exact simulation methods

4.1 Inverse methods

The *inverse* method leverages Theorem 7.4.I [1] which states that every simple point process[2] can be transformed to a homogeneous Poisson process with unit rate via the compensator. Let t_n be the time in which the n-th chronologically sorted event took place and $t_0 \equiv 0$, we define the compensator as:

$$\Lambda^*(t_n) \equiv \tilde{t}_n \equiv \int_0^{t_n} \lambda^*(u) du \tag{4.1}$$

The transformed data $\tilde{t}_n$ forms a homogeneous Poisson process with unit rate. Now, if this is the case, then the transformed interval is distributed according to the exponential distribution.

$$\Delta\tilde{t}_n \equiv \tilde{t}_n - \tilde{t}_{n-1} \sim \exp(1) \tag{4.2}$$

The idea is to draw realizations from the unit rate Exponential process and solve Equation 4.2 for t_n to determine the next event/firing time. We illustrate this in Algorithm 1 where we adapt Algorithm 7.4 [1].

Whenever the conditional intensity is constant between two points, Equation 4.2 can be solved analytically. Let $\lambda^*(t) = \lambda_{n-1}, \forall t_{n-1} \leq t < t_n$, then

$$\begin{aligned} &\int_{t_{n-1}}^{t_n} \lambda^*(u)\, du = \Delta\tilde{t}_n \iff \\ &\lambda_{n-1}(t_n - t_{n-1}) = \Delta\tilde{t}_n \iff \\ &t_n = t_{n-1} + \frac{\Delta\tilde{t}_n}{\lambda_{n-1}}. \end{aligned} \tag{4.3}$$

Which is equivalent to drawing the next realization time from the re-scaled exponential distribution $\Delta t_n \sim \exp(\lambda_{n-1})$. As we will see in Subsection 2, this implies that the *inverse* and *thinning* methods are the same whenever the conditional intensity is constant between jumps.

The main drawback of the *inverse* method is that the root finding problem defined in Equation 4.2 often requires a numerical solution. To get around a similar obstacle in the context of the piecewise deterministic Markov process, Veltz [23] proposes a change of variables in time that recasts the root finding problem into an initial value problem. He denotes his method ***CHV***.

Piecewise deterministic Markov processes are composed of two parts: the jump process and the piecewise ODE that changes stochastically at jump times — see Lemaire *et al.* [11] for a formal definition. Therefore, it is easy to employ ***CHV*** in our case by setting the ODE part to zero throughout time. Adapting from Veltz [23], we can determine the model jump time t_n after sampling $\Delta\tilde{t}_n \sim \exp(1)$ by solving the following initial value problem until $\Delta\tilde{t}_n$.

$$t(0) = t_{n-1}\ ,\ \frac{dt}{d\tilde{t}} = \frac{1}{\lambda^*(t)} \tag{4.4}$$

Looking back at Equation 4.1, we note that it is a one-to-one mapping between t and $\tilde{t}$ which makes it completely natural to write $t(\Delta\tilde{t}_n) \equiv \Lambda^{*-1}(\tilde{t}_{n-1} + \Delta\tilde{t}_n)$.

[2] A simple point process is a process in which the probability of observing more than one point in the same location is zero.

Alternatively, when the intensity function is differentiable between jumps we can go even further by recasting the jump problem as a piecewise deterministic Markov process. Let $\lambda_n^* \equiv \lambda^*(t_n)$, then the flow $\varphi_{t-t_n}(\lambda_n^*)$ maps the initial value of the conditional intensity at time t_n to its value at time t. In other words, the flow describes the deterministic evolution of the conditional intensity function over time. Next, denote $\mathbf{1}(\cdot)$ as the indicator function, then the conditional intensity function can be re-written as a jump process:

$$\lambda^*(t) = \sum_{n \geq 1} \varphi_{t-t_{n-1}}(\lambda_{n-1}) \mathbf{1}(t_{n-1} \leq t < t_n). \tag{4.5}$$

According to Meiss [15], if $\varphi_t(\cdot)$ is a flow, then it is a solution to the initial value problem:

$$\varphi_0(\lambda_n^*) = \lambda_n^* \, , \, \frac{d}{dt}\varphi_{t-t_n}(\lambda_n^*) = g(\varphi_{t-t_n}(\lambda_n^*)) \tag{4.6}$$

where $g : \mathbb{R}^+ \to \mathbb{R}$ is the vector field of λ^* such that $d\lambda^*/dt = g(\lambda^*)$.

Based on Equation 2.1, we find that the probability of observing an interval longer than s given internal history H_{t^-} is equivalent to:

$$\begin{aligned} \Pr(t_n - t_{n-1} > s \mid H_{t^-}) &= 1 - \int_{t_{n-1}}^{t_{n-1}+s} p^*(u)du = \\ &= \exp\left(-\int_{t_{n-1}}^{t_{n-1}+s} \lambda^*(u)du\right) = \\ &= \exp\left(-\int_{t_{n-1}}^{t_{n-1}+s} \varphi_{u-t_{n-1}}(\lambda_{n-1}^*)du\right) \end{aligned} \tag{4.7}$$

Equations 4.5 and 4.7 define a piecewise deterministic Markov process satisfying the conditions of Theorem 3.1 [23]. In this case, we find t_n by solving the following initial value problem from 0 to $\Delta\tilde{t}_n \sim \exp(1)$.

$$\begin{cases} \lambda^*(t(0)) = \lambda^*(t_{n-1}) \, , \, \dfrac{d\lambda^*}{d\tilde{t}} = \dfrac{g(\lambda^*(t))}{\lambda^*(t)} \\ t(0) = t_{n-1} \, , \, \dfrac{dt}{d\tilde{t}} = \dfrac{1}{\lambda^*(t)}. \end{cases} \tag{4.8}$$

This problem specifies how the conditional intensity and model time evolve with respect to the transformed time. The solution to Equation 4.2 is then given by $(t_n = t(\Delta\tilde{t}_n), \lambda^*(t(\Delta\tilde{t}_n)) = \lambda^*(t_n))$.

In Algorithm 1, we can implement the CHV method by solving either Equation 4.4 or Equation 4.8 instead of Equation 4.2. We denote the first specification as ***CHV simple*** and the second as ***CHV full***. Note that ***CHV full*** requires that the conditional intensity be piecewise differentiable. The algorithmic complexity is then determined by the ODE solver and no root-finding is required. In Section 6.2, we will show that there are substantial differences in performance between them with the full specification being faster.

Another concern with Algorithm 1 is updating and drawing from the conditional mark distribution in Line 8, and updating the conditional intensity in Line 9. Assume a process with K number of marks. A naive implementation of Line 9 scales with the number of marks as $O(K)$ since λ^* is usually constructed as the sum of K independent processes, each of which requires updating the conditional intensity rate. Likewise, drawing from the mark distribution in Line 8 usually involves drawing from a categorical distribution whose naive implementations also scales with the number of marks as $O(K)$.

Finally, Algorithm 1 is not guaranteed to terminate in finite time since one might need to sample many points before $t_n > T$. The sampling rate can be especially high when simulating the process in a large population with self-exciting encounters. In biochemistry, Salis and Kaznessis [19] partition a large system of chemical reactions into two: fast and slow reactions. While they approximate the fast reactions with a Gaussian process, the slow reactions are solved using a variation of the inverse method. They obtain an equivalent expression for the rate of slow reactions as in Equation 4.2, which is integrated with the Euler method.

Algorithm 1 The *inverse* method for simulating a marked evolutionary point process over a fixed duration of time $[0, T)$.

1: **procedure** INVERSEMETHOD($[0, T)$, λ^*, f^*,)
2: initialize the history $H_{T^-} \leftarrow \{\}$
3: set $n \leftarrow 0, t \leftarrow 0$
4: **while** $t < T$ **do**
5: $n \leftarrow n + 1$
6: draw $\Delta\tilde{t}_n \sim \exp(1)$
7: find the next event time t_n by solving Equation 4.2 or 4.8
8: update f^* and draw the mark $k_n \sim f^*(k \mid t_n)$
9: update the history $H_{T^-} \leftarrow H_{T^-} \cup (t_n, k_n)$ and λ^*
10: **end while**
11: **return** H_{T^-}
12: **end procedure**

4.2 Thinning methods

Thinning methods are one of the most popular methods for simulating point processes. The main idea is to successively sample a homogeneous process, then thin the obtained points with the conditional intensity of the original process. As stated in Proposition 7.5.I [1], this procedure simulates the target process by construction. The advantage of *thinning* over *inverse* methods is that the former only requires the evaluation of the conditional intensity function while the latter requires computing the inverse of its integrated form [1].

Thinning algorithms have been proposed in different forms [1]. The Shedler-Lewis algorithm can simulate processes with bounded intensity [12]. The classical algorithm from Ogata [16] overcomes this limitation and only requires the local boundedness of the conditional intensity. The advantage of Ogata's algorithm and its variations is that it can simulate processes with potentially unbounded intensity, such as self-exciting ones. As long as the intensity conditioned on the simulated history remains locally bounded, it is possible to simulate subsequent points indefinitely.

In biochemistry, the *thinning* method was popularized by Gillespie [6, 5]. For this reason, this method is also called the ***Gillespie*** method. Gillespie himself called it the ***direct*** method or the ***stochastic simulation algorithm***. Gillespie introduced the *thinning* method in the context of simulating chemical reactions of well-stirred systems. He developed a stochastic model for molecule interactions from physics principles without any references to the point process theory developed in this section. His model of chemical interactions is equivalent to a marked Poisson process with constant conditional intensity between jumps. The model consists of distinct populations of molecular species that interact through several reaction channels. A chemical reaction consists of a Poisson process that transforms a set of molecules of some type into a set

of molecules of another type. What Gillespie calls the master equation can be deduced from the *superposition theorem* — Theorem 3.3 [9].

Alternatively, in biochemistry, *thinning* methods are known as *rejection* algorithms. Than *et al.* [21, 22] proposed the *rejection-based algorithm with composition-rejection search*, yet another more sophisticated variation of the *thinning* method. In this case, the procedure groups similar processes together. For each group, an upper- and lower-bound conditional intensity is used for thinning. A similar procedure is also described in [20], in which the authors refer to their algorithm as *kinetic Monte Carlo*.

In Algorithm 2, we modify Algorithm 7.5.IV [1] to incorporate the idea of a lower bound for the conditional intensity from [22]. To implement the algorithm, we define three functions, $\bar{B}^*(t) = \bar{B}(t \mid H_t)$, $\underline{B}^*(t) = \underline{B}(t \mid H_t)$ and $L^*(t) = L(t \mid H_t)$, that characterize the local boundedness condition such that:

$$\lambda^*(t+u) \leq \bar{B}^*(t) \text{ and } \lambda^*(t+u) \geq \underline{B}^*(t), \quad \forall\, 0 \leq u \leq L^*(t). \tag{4.9}$$

The tighter the bound on $\bar{B}^*(t)$, the lower the number of samples discarded. Since looser bounds lead to less efficient algorithms, the art, when simulating via *thinning*, is to find the optimal balance between the local supremum of the conditional intensity $\bar{B}^*(t)$ and the duration of the local interval $L^*(t)$. On the other hand, the infimum $\underline{B}^*(t)$ can be used to avoid the evaluation of $\lambda^*(t+u)$ in Line 5 of Algorithm 3 which often can be expensive.

When the conditional intensity is constant between jumps such that $\lambda^*(t) = \lambda_{n-1}, \forall t_{n-1} \leq t < t_n$, let $\bar{B}^*(t) = \underline{B}^*(t) = \lambda_{n-1}$ and $L^*(t) = \infty$. We have that for any $u \sim \exp(1 \,/\, \bar{B}^*(t)) = \exp(\lambda_{n-1})$ and $v \sim U[0,1]$, $u < L^*(t) = \infty$ and $v < \lambda^*(t+u) \,/\, \bar{B}^*(t) = 1$. Therefore, we advance the internal history for every iteration of Algorithm 2. In this case, the bound $\bar{B}^*(t)$ is as tight as possible, and this method becomes the same as the *inverse* method of Subsection 4.1.

We can draw many more connections between the *thinning* and *inverse* methods. Lemaire *et al.* [11] propose a version of the *thinning* algorithm for Piecewise Deterministic Markov Processes which does not use the local interval L^* for rejection — this is equivalent to $L^*(t) = \infty$ —, and does not assume the upper bound $\bar{B}^*(t)$ is constant over $L^*(t)$. The efficiency of their algorithm depends on the assumption that the stochastic process determined by $\bar{B}^*(t)$ can be efficiently inverted such that candidate times can be efficiently obtained using Equation 4.1. They propose an optimal bound as a piecewise constant function partitioned in such a way that it envelopes the intensity function as strictly as possible. They then show that under certain conditions the stochastic process determined by $\bar{B}^*(t)$ converges in distribution to the target conditional intensity as the partitions of the optimal boundary converge to zero. Although their simulation approach does not exactly match ours, it suggests some properties between the *thinning* and the *inverse* method that we could investigate in the future. Among other things, the efficiency of *thinning* compared to *inversion* most likely depends on the rejection rate obtained by the former and the number of steps required by the ODE solver for the latter.

While *thinning* algorithms avoid the issue of directly computing the inverse of the integrated conditional intensity, they increase the number of time steps needed in the sampling algorithm as we are now sampling from a process with an increased intensity relative to the original process. Moreover, like the *inverse* method, *thinning* algorithms can also face issues related with drawing from the conditional mark distribution — Line 11 of Algorithm 2 —, and updating the conditional intensity — Line 3 of Algorithm 3 — and the mark distribution — Line 12 of Algorithm 2.

Algorithm 2 The *thinning* method for simulating a marked evolutionary point process over a fixed duration of time $[0, T)$.

```
1:  procedure ThinningMethod([0, T), λ*, f*,)
2:      initialize the history H_{T⁻} ← {}
3:      set n ← 0, t ← 0
4:      while true do
5:          t ← TimeViaThinning([t, T), H_{T⁻}, λ*)
6:          if t ≥ T then
7:              break
8:          end if
9:          n ← n + 1
10:         t_n ← t
11:         update f* and draw the mark k_n ~ f*(k | t_n)
12:         update the history H_{T⁻} ← H_{T⁻} ∪ (t_n, k_n)
13:     end while
14:     return H_{T⁻}
15: end procedure
```

Algorithm 3 Generates the next event time via *thinning*.

```
1:  procedure TimeViaThinning([t, T), λ*, H_t,)
2:      while t < T do
3:          update λ*
4:          find B̄*(t), B̲*(t) and L*(t) which satisfy Eq. 4.9
5:          draw u ~ exp(B̄*(t)) and v ~ U[0, B̄*(t)]
6:          if u > L*(t) then
7:              t ← t + L*(t)
8:              next
9:          end if
10:         if (v > B̲*(t)) and (v > λ*(t + u)) then
11:             t ← t + u
12:             next
13:         end if
14:         t ← t + u
15:         break
16:     end while
17:     return t
18: end procedure
```

4.3 Queuing methods

As an alternative to his *direct* method — in this text referred as the constant rate *thinning* method —, Gillespie introduced the *first reaction* method in his seminal work on simulation algorithms [6]. The *first reaction* method separately simulates the next reaction time for each reaction channel — *i.e.* for each mark. It then selects the smallest time as the time of the next event, followed by updating the conditional intensity of all channels accordingly. This is a variation of the constant rate *thinning* method to simulate a set of inter-dependent point processes, making use of the *superposition theorem* — Theorem 3.3 [9] — in the inverse direction.

Gibson and Bruck [3] improved the *first reaction* method with the *next reaction* method. They innovate on three fronts. First, they keep a priority queue to quickly retrieve the next event. Second, they keep a dependency graph to quickly locate all conditional intensity rates that need to be updated after an event is fired. Third,

they re-use previously sampled reaction times to update unused reaction times. This minimizes random number generation, which can be costly. Priority queues and dependency graphs have also been used in the context of social media [2] and epidemics [8] simulation. In both cases, the phenomena are modelled as point processes.

We prefer to call this class of methods *queuing* methods since most efficiency gains come from maintaining a priority queue of the next event times. Up to this point we assumed that we were sampling from a global process with a mark distribution that could generate any mark k given an event at time t. With queuing, it is possible to simulate point processes with a finite space of marks as M interdependent point processes — see Definition 6.4.1 [1] of multivariate point processes — doing away with the need to draw from the mark distribution at every event occurrence. Alternatively, it is possible to split the global process into M interdependent processes each one of which with its own mark distribution.

Our contribution, Algorithm 5, presents a method for sampling a superposed point process consisting of M processes by keeping the strike time of each process in a priority queue Q. The priority queue is initially constructed in $O(M)$ steps in Lines 4 to 7 of Algorithm 5. In contrast to *thinning* methods, updates to the conditional intensity depend only on the size of the neighborhood of i. That is, processes j whose conditional intensity depends on the history of i. If the graph is sparse, then updates will be faster than with *thinning*.

A source of inefficiency in some implementations of queuing algorithms is the fact that one might need to go through multiple rejection cycles before accepting a time candidate t_i for process i. This might require looking ahead in the future. In addition to that, if process j, which i depends on, takes place before i, then we need to repeat the whole thinning process to obtain a new time candidate for i. We thus propose Algorithm 5 which is a queuing algorithm that performs thinning in synchrony with the main loop, thus avoiding look ahead and wasted rejections. Since thinning is now synced with the main loop, it is possible to couple this simulator with other algorithms that step chronologically through time. These include ordinary differential equation solvers, enabling us to simulate jump processes with rates given by a differential equation. This is the first synced thinning algorithm we are aware of.

Algorithm 4 Generates the next candidate time for *queuing*.

1: **procedure** QUEUETIME(t, λ^*, H_t,)
2: update λ^*
3: find $\bar{B}^*(t)$, $\underline{B}^*(t)$ and $L^*(t)$ which satisfy Eq. 4.9
4: draw $u \sim \exp(\bar{B}^*(t))$
5: **if** $u > L^*(t)$ **then**
6: accepted $\leftarrow$ false
7: **else**
8: accepted $\leftarrow$ true
9: **end if**
10: $t \leftarrow t + u$
11: **return** $t, \bar{B}^*(t), \underline{B}^*,$ accepted
12: **end procedure**

Algorithm 5 The *queuing* method for simulating a marked evolutionary point process over a fixed duration of time $[0, T)$.

1: **procedure** QUEUINGMETHOD($[0, T)$, $\{\lambda_k^*\}$, $\{f_k^*\}$,)
2: initialize the history $H_{T^-} \leftarrow \{\}$
3: set $n \leftarrow 0, t \leftarrow 0$
4: **for** i=1,M **do**
5: $(t_i, \bar{B}_i^*, \underline{B}_i^*, a_i) \leftarrow \text{QueueTime}(0, H_{T^-}, \lambda_i^*(\cdot))$
6: push $(t_i, \bar{B}_i^*, \underline{B}_i^*, a_i, i)$ to Q
7: **end for**
8: **while** $t < T$ **do**
9: first $(t_i, i, \bar{B}_i^*, \underline{B}_i^*, a_i, i)$ from Q
10: $t \leftarrow t_i$
11: **if** $t \geq T$ **then**
12: **break**
13: **end if**
14: draw $v \sim U[0, \bar{B}_i^*]$
15: **if** $(v > \underline{B}_i^*)$ and $(v > \lambda^*(t))$ **then**
16: $a_i \leftarrow$ false
17: **end if**
18: **if** a_i **then**
19: $n \leftarrow n + 1$
20: $t_n \leftarrow t$
21: update f^* and draw the mark $k_n \sim f_i^*(k \mid t_n)$
22: update the history $H_{T^-} \leftarrow H_{T^-} \cup (t_n, k_n)$
23: **for** $j \in \{i\} \cup \text{Neighborhood}(i)$ **do**
24: $(t_j, \bar{B}_j^*, \underline{B}_j^*, a_j) \leftarrow \text{QueueTime}(0, H_{T^-}, \lambda_j^*(\cdot))$
25: update $(t_j, \bar{B}_j^*, \underline{B}_j^*, a_j, j)$ in Q
26: **end for**
27: **else**
28: $(t_i, \bar{B}_i^*, \underline{B}_i^*, a_i) \leftarrow \text{QueueTime}(0, H_{T^-}, \lambda_i^*(\cdot))$
29: update $(t_i, \bar{B}_i^*, \underline{B}_i^*, a_i, i)$ in Q
30: **end if**
31: **end while**
32: **return** H_{T^-}
33: **end procedure**

5. Implementation

`JumpProcesses.jl` is a Julia library for simulating jump — or point — processes which is part of Julia's SciML organization. Our discussion in Section 4 identified three exact methods for simulating point processes. In all the cases, we identified two mathematical constructs required for simulation: the intensity rate and the mark distribution. In `JumpProcesses.jl`, these can be mapped to user defined functions `rate(u, p, t)` and `affect!(integrator)`. The library provides APIs for defining processes based on the nature of the intensity rate and the intended simulation algorithm. Processes intended for exact methods can choose between `ConstantRateJump` and `VariableRateJump`. While the former expects the rate between jumps to be constant, the latter allows for time-dependent rates. The library also provides the `MassActionJump` API to define large systems of point processes that can be expressed as reaction equations. Finally, `RegularJump` are intended for inexact methods.

The *inverse* method as described around Equation 4.2 uses root find to find the next jump time. Jumps to be simulated via the *inverse* method must be initialized as a `VariableRateJump`. `JumpProcesses.jl` builds a continuous callback following the algorithm in [19] and passes the problem to an `OrdinaryDiffEq.jl` integrator, which easily interoperates with `JumpProcesses.jl` (both libraries are part of the *SciML* organization, and by design built to easily compose). `JumpProcesses.jl` does not yet support the CHV ODE based approach.

Alternatively, *thinning* and *queuing* methods can be simulated via discrete steps. In the context of the library, any method that uses a discrete callback is called an *aggregator*. There are twelve differ-

ent aggregators, seven of which implement a variation of the *thinning* method and five of which a variation of the *queuing* method. We start with the *thinning* aggregators, none of which support `VariableRateJump`. Algorithm 2 assumes that there is a single process. In reality, all the implementations assume a finite multivariate point process with M interdependent processes. This can be easily conciliated, as we do now, using Definition 6.4.1 [1] which states the equivalence of such process with a point process with a finite space of marks. As all the *thinning* aggregators only deal with `ConstantRateJump`, the intensity between jumps is constant, Algorithm 3 short-circuits to quickly return $t \sim \exp(\bar{B}) = \exp(\lambda_n)$ as discussed in Subsection 4.2. Next, the mark distribution becomes the categorical distribution weighted by the intensity of each process. That is, given an event at time t_n, we have that the probability of drawing process i out of M sub-processes is $\lambda_i^*(t_n)/\lambda^*(t_n)$. Conditional on sub-process i, the corresponding `affect!(integrator)` is invoked, that is, $k_n \sim f_i^*(k \mid t_n)$. Here we use a notation analogous to Section 4.3.

Where most implementations differ is on updating the mark distribution in Line 11 of Algorithm 2 and the conditional intensity rate in Line 3 of Algorithm 3. `Direct` and `DirectFW` follows the *direct* method in [6] which re-evaluates all intensities after every iteration scaling at $O(K)$. When drawing the process to fire, it executes a search in an array that stores the cumulative sum of rates. `DirectCR`, `SortingDirect` and `RDirect` only re-evaluate the intensities of the processes that are affected by the realized process. This operation is executed efficiently by keeping a vector of dependencies. These three algorithms differ in how they select the process. `DirectCR` keeps the intensity rates in a priority table, it is implemented after [20]. `SortingDirect` keeps the intensity rate in a loosely sorted array following [14]. In both cases, the idea is to use a randomly generated number between zero and one to guide the search for the next jump. With the intensity rates sorted, more frequent processes should be selected faster than less frequent ones. Overall, this should increase the speed of the simulation. `RDirect` keeps track of the maximum rate of the system, it implements an algorithm equivalent to *thinning* with $\bar{B}$ equal to the maximum rate. However, the implementation differs. It thins with $\bar{B} = \lambda_n$, then randomly selects a candidate process and confirms the candidate only if its rate is above a random proportion of the maximum rate. Finally, `RSSA` and `RSSACR` group processes with similar rates in bounded brackets. The upper bounds are used for *thinning*. For each round of *thinning*, a sampled candidate process is considered for selection. In `RSSA`, the candidate process is selected similarly to `Direct`, while a priority table is used in `RSSACR`. Both of these algorithms follow from [21, 22].

Next, we consider the *queuing* aggregators. Starting with aggregators that only support `ConstantRateJump`s we have, `FRM`, `FRMFW` and `NRM`. `FRM` and `FRMFW` follow the *first reaction* method in [6]. To compute the next jump, both algorithms compute the time to the next event for each process and select the process with minimum time. This is equivalent to assuming a complete dependency graph in Algorithm 5. For large systems, they can be less efficient than `NRM`. The latter implementation is sourced from [3] and follows Algorithm 5 very closely.

Previously, we attempted to bridge the gap between the treatment of point process simulation in statistics and biochemistry. Despite the many commonalities, most of the algorithms implemented in `JumpProcesses.jl` are derived from the biochemistry literature. There has been less emphasis on implementing processes commonly studied in statistics such as self-exciting point processes characterized by time-varying and history-dependent intensity rates. This is addressed by our latest aggregator, `Coevolve`. This is the first aggregator that supports `VariableRateJump`s, facilitating substantially more performant simulation of processes with time-dependent intensity rates in `JumpProcesses.jl` and `DifferentialEquations.jl` compared to the current *inverse* method-based approach that relies on ODE integration and continuous events.

The implementation of this aggregator takes inspiration from [2], and improves the method in several ways. First, we take advantage of the modularity and composability of Julia to design an API that accepts any intensity rate, not only the Hawkes'. Second, we avoid the re-computation of unused random numbers. When updating processes that have not yet fired, we can transform the unused time of constant rate processes to obtain the next candidate time for the first round of iteration of the *thinning* procedure in Algorithm 3. This saves one round of sampling from the exponential distribution, which translates into a faster algorithm. Third, we allow the user to supply a lower bound rate which can short-circuit the loop in Algorithm 3, saving yet another round of sampling. Fourth, it adapts to processes with constant intensity between jumps which reduces the loop in Algorithm 3 to the equivalent implemented in `NRM`. Finally, since `Coevolve` can be mapped to a *thinning* algorithm — see [2] —, it can simulate any point process on the real line with a non-negative, left-continuous, history-adapted and locally bounded intensity rate as per Proposition 7.5.I [1].

`Coevolve` syncs with the main execution loop which means that it can be easily coupled with differential equations modeled with `OrdinaryDiffEq.jl`. For instance, It is possible to model processes whose rates are given by a differential equation. This is a departure from the algorithm described in [2] which translates not only into a faster, but also more flexible simulator. This difference in implementation follows our discussion on the relationship between the main execution loop and the thinning loop in Section 4.3.

6. Empirical evaluation

This section conducts some empirical evaluation of the `JumpProcesses.jl` aggregators described in Section 5. First, since `Coevolve` is a new aggregator, we test its correctness by conducting statistical analysis. Second, we conduct the jump benchmarks available in `SciMLBenchmarks.jl`. We have added new benchmarks that assess the performance of the new aggregators under settings that could not be simulated with previous aggregators.

6.1 Statistical analysis of `Coevolve`

To simulate a process intended for a discrete solver with `JumpProcesses.jl`, we define a discrete problem, initialize the jumps and define the jump problem which takes the aggregator as an argument. The jump problem can then be solved with the discrete stepper provided by `JumpProcesses.jl`, `SSAStepper`. The code for simulating the homogeneous Poisson process with `Direct` is reproduced in Listing 1.

Listing 1: Simulation of the homogeneous Poisson process.

```
using JumpProcesses
rate(u, p, t) = p[1]
affect!(integrator) = (integrator.u[1] += 1;
  nothing)
jump = ConstantRateJump(rate, affect!)
u, tspan, p = [0.], (0., 200.), (0.25,)
dprob = DiscreteProblem(u, tspan, p)
jprob = JumpProblem(dprob, Direct(), jump;
```

```
    dep_graph=[[1]])
sol = solve(jprob, SSAStepper())
```

The simulation of a Hawkes process — see Subsection 6.2 for a definition — requires a `VariableRateJump` along with the rate bounds and the interval for which the rates are valid. Also, since the Hawkes process is history dependent, we close the `rate` and `affect!` function with a vector containing the history of events. The code for simulating the Hawkes process is reproduced in Listing 2. Note that it is possible to simplify the computation of the rate — see Subsection 6.2 —, but we keep the code here as close as possible to its usual definition for illustration purposes.

Listing 2: Simulation of the Hawkes process.

```
using JumpProcesses
h = Float64[]
rate(u, p, t) = p[1] +
  p[2]*sum(exp(-p[3]*(t-_t)) for _t in h; init=0)
lrate(u, p, t) = p[1]
urate = rate
rateinterval(u, p, t) = 1/(2*urate(u,p,t))
affect!(integrator) = (push!(h, integrator.t);
  integrator.u[1] += 1; nothing)
jump = VariableRateJump(rate, affect!; lrate,
  urate, rateinterval)
u, tspan, p = [0.], (0., 200.), (0.25, 0.5, 2.0)
dprob = DiscreteProblem(u, tspan, p)
jprob = JumpProblem(dprob, Coevolve(), jump;
  dep_graph=[[1]])
sol = solve(jprob, SSAStepper())
```

To assess the correctness of `Coevolve`, we add it to the `JumpProcesses.jl` test suite. Some tests check whether the aggregators are able to obtain empirical statistics close to the expected in a number of simple biochemistry models such as linear reactions, DNA repression, reversible binding and extinction. The test suite was missing a unit test for self-exciting process. Thus, we have added a test for the univariate Hawkes model that checks whether algorithms that accept `VariableRateJump` are able to produce an empirical distribution of trajectories whose first two moments of the observed rate are close to the expected ones.

In addition to that, the correctness of the implemented algorithm can be visually assessed using a Q-Q plot. As discussed in Subsection 4.1, every simple point process can be transformed to a Poisson process with unit rate. This implies that the interval between points for any such transformed process should match the exponential distribution. Therefore, the correctness of any aggregator can be assessed as following. First, transform the simulated intervals with the appropriate compensator. Let t_{n_i} be the time in which the n-th event of sub-process i took place and $t_{0_i} \equiv 0$, the compensator for sub-process i is given by the following:

$$\Lambda_i^*(t_{n_i}) \equiv \Lambda_{n_i}^* \equiv \int_0^{t_{n_i}} \lambda_i^*(u)du \tag{6.1}$$

Then the transformed simulated interval is given by:

$$\Delta\Lambda_{n_i} \equiv \Lambda_{n_i}^* - \Lambda_{(n-1)_i}^* \tag{6.2}$$

Compute the empirical quantiles of the transformed intervals. That is, the q-th quantile is the interval $\Delta\Lambda_q$ that divides the sorted intervals in two sets, those below and above $\Delta\Lambda_q$ such that q-percent of the elements are below it. Plot the empirical quantiles with the corresponding quantiles of the exponential distribution. If the simulator produces correct trajectories, this plot known as Q-Q plot should depict the points aligned around the 45-degree line. We produce Q-Q plots for the homogeneous Poisson process as well as the compound Hawkes process — see Subsection 6.2 for a definition — to attest the correctness of `Coevolve`. Figure 1 (d) depicts the Q-Q plot for a ten-node compound Hawkes process with parameters $\lambda = 0.5, \alpha = 0.1, \beta = 2.0$ simulated 250 times for 200 units of time. Figure 1 also depicts the trajectory, the conditional intensity and the network structure of a single simulation for three random nodes in panels (a), (b) and (c) respectively. We obtained similar Q-Q plots for the other algorithms that benchmarked the Multivariate Hawkes process below.

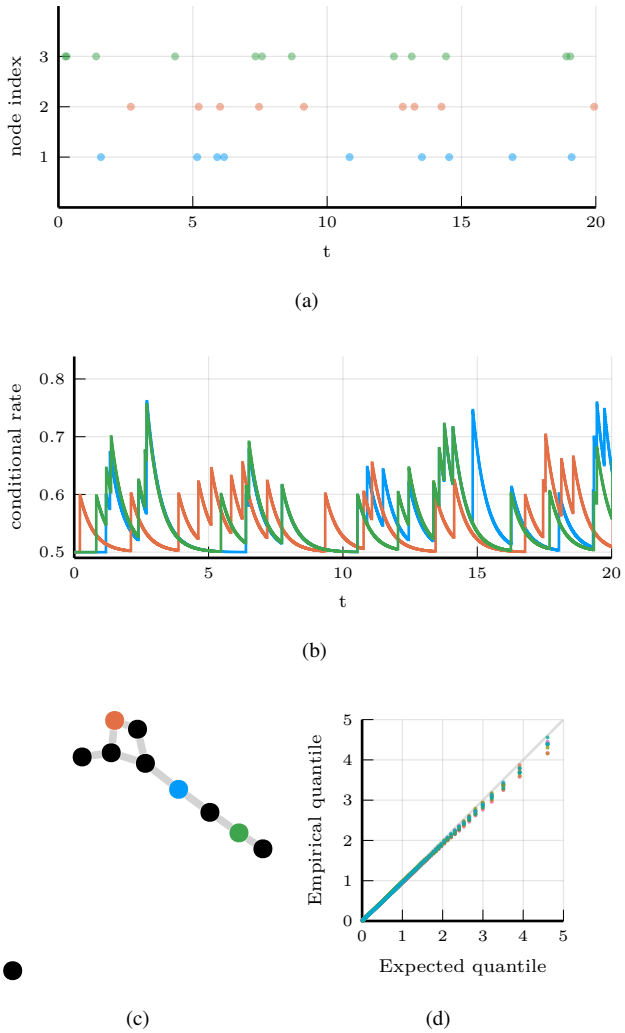

Fig. 1: Simulations of 10-nodes compound Hawkes process with parameters $\lambda = 0.5, \alpha = 0.1, \beta = 2.0$ for 200 units of time. (a) and (b) sampled trajectory and intensity rate for a single simulation for the three selected nodes in (c) for the first 20 units of time. (c) underlying 10-nodes network with three random nodes selected. (d) Q-Q plot of transformed inter-event time for 250 simulations colored by node.

6.2 Benchmarks

We conduct a set of benchmarks to assess the performance of the `JumpProcesses.jl` aggregators described in Section 5. All

	Diffusion	Multi-state	Gene I	Gene II
`Direct`	7.18 s	0.16 s	0.24 ms	0.59 s
`FRM`	15.04 s	0.25 s	0.29 ms	0.78 s
`SortingDirect`	1.08 s	0.11 s	**0.23 ms**	**0.50 s**
`NRM`	0.75 s	0.25 s	0.39 ms	0.89 s
`DirectCR`	0.51 s	0.21 s	0.47 ms	1.00 s
`RSSA`	1.42 s	**0.10 s**	0.43 ms	0.65 s
`RSSACR`	**0.46 s**	0.16 s	0.91 ms	1.07 s
`Coevolve`	0.90 s	0.36 s	0.59 ms	1.33 s

Table 1. : Median execution time. A 1-dimensional continuous time random walk approximation of a diffusion model (Diffusion), the multi-state model from Appendix A.6 [13] (Multi-state), a simple negative feedback gene expression model (Gene I) and the negative feedback gene expression from [7] (Gene II). Fastest time is **bold**, second fastest underlined. Benchmark source code and dependencies are available in `SciMLBenchmarks.jl`, see first paragraph of Section 6.2 for source references.

benchmarks are available in `SciMLBenchmarks.jl`[3]. All were run in BuildKite[4] via the continuous integration facilities provided by the package maintainers. We have added two benchmark suites to assess the performance of the new aggregators under settings that could not be simulated with previous aggregators.

First, we assess the speed of the aggregators against jump processes whose rates are constant between jumps. There are four such benchmarks: a 1-dimensional continuous time random walk approximation of a diffusion model (Diffusion), the multi-state model from Appendix A.6 [13] (Multi-state), a simple negative feedback gene expression model (Gene I) and the negative feedback gene expression from [7] (Gene II). We simulate a single trajectory for each aggregator to visually check that they produce similar trajectories for a given model. The Diffusion, Multi-state, Gene I and Gene II benchmarks are then simulated 50, 100, 2000 and 200 times, respectively. Check the source code for further implementation details.

Benchmark results are listed in Table 1. The table shows that no single aggregator dominates suggesting they should be selected according to the task at hand. However, `FRM`, `NRM`, `Coevolve` never dominate any benchmark. In common, they all belong to the family of queuing methods suggesting that there is a penalty when using such methods for jump processes whose rates are constant between jumps. We also note that the performance of `Coevolve` lag that of `NRM` despite the fact that `Coevolve` should take the same number of steps as `NRM` when no `VariableRateJump` is used. The reason behind this discrepancy is likely due to implementation differences, but left for future investigation.

Second, we add a new benchmark which simulates the compound Hawkes process for an increasing number processes. Consider a graph with V nodes. The compound Hawkes process is characterized by V point processes such that the conditional intensity rate of node i connected to a set of nodes E_i in the graph is given by

$$\lambda_i^*(t) = \lambda + \sum_{j \in E_i} \sum_{t_{n_j} < t} \alpha \exp\left[-\beta(t - t_{n_j})\right]. \tag{6.3}$$

This process is known as self-exciting, because the occurrence of an event j at t_{n_j} will increase the conditional intensity of all the processes connected to it by α. The excited intensity then decreases at a rate proportional to β.

$$\begin{aligned}\frac{d\lambda_i^*(t)}{dt} &= -\beta \sum_{j \in E_i} \sum_{t_{n_j} < t} \alpha \exp\left[-\beta(t - t_{n_j})\right] \\ &= -\beta\left(\lambda_i^*(t) - \lambda\right)\end{aligned} \tag{6.4}$$

The conditional intensity of this process has a recursive formulation which can significantly speed the simulation. The recursive formulation for the univariate case is derived in [10] which also provides additional discussion and results on the Hawkes process. We derive the compound case here. Let $t_{N_i} = \max\{t_{n_j} < t \mid j \in E_i\}$ and $\phi_i^*(t)$ below.

$$\begin{aligned}\phi_i^*(t) &= \sum_{j \in E_i} \sum_{t_{n_j} < t} \alpha \exp\left[-\beta(t - t_{N_i} + t_{N_i} - t_{n_j})\right] \\ &= \exp\left[-\beta(t - t_{N_i})\right] \sum_{j \in E_i} \sum_{t_{n_j} \le t_{N_i}} \alpha \exp\left[-\beta(t_{N_i} - t_{n_j})\right] \\ &= \exp\left[-\beta(t - t_{N_i})\right]\left(\alpha + \phi_i^*(t_{N_i})\right)\end{aligned} \tag{6.5}$$

Then the conditional intensity can be re-written in terms of $\phi_i^*(t_{N_i})$.

$$\lambda_i^*(t) = \lambda + \phi_i^*(t) = \lambda + \exp\left[-\beta(t - t_{N_i})\right]\left(\alpha + \phi_i^*(t_{N_i})\right) \tag{6.6}$$

A random graph is sampled from the Erdős-Rényi model. This model assumes the probability of an edge between two nodes is independent of other edges, which we fix at 0.2. Note that this setup implies an increasing expected node degree.

We fix the Hawkes parameters at $\lambda = 0.5, \alpha = 0.1, \beta = 5.0$ ensuring the process does not explode and simulate models in the range from 1 to 95 nodes for 25 units of time. We simulate 50 trajectories with a limit of ten seconds to complete execution. For this benchmark, we save the state of the system exactly after each jump.

We assess the benchmark in eight different settings. First, we run the *inverse* method, `Coevolve` and *CHV simple* using the brute force formula of the intensity rate which loops through the whole history of past events — Equation 6.3. Second, we simulate the same three methods with the recursive formula — Equation 6.6. Next, we run the benchmark against *CHV full*. All *CHV* specifications are implemented with `PiecewiseDeterministicMarkovProcesses.jl` [5] which is developed by Veltz, the author of the *CHV* algorithm discussed in Subsection 4.1. Finally, we run the benchmark using the Python library Tick[6]. This library implements a version of the thinning method for simulating the Hawkes process and implements a recursive algorithm for computing the intensity rate.

Table 2 shows that the *Inverse* method which relies on root finding is the most inefficient of all methods for any system size. For large system size this method is unable to complete all 50 simulation runs because it needs to find an ever larger number of roots of an ever larger system of differential equations.

The recursive implementation of the intensity rate brings a considerable boost to the simulations, placing `Coevolve` as one of the

[3] `https://github.com/SciML/SciMLBenchmarks.jl/tree/3bf650c1aae7b10e49cbd10e8f626d2a517f3e79/benchmarks/Jumps`

[4] `https://buildkite.com/julialang/scimlbenchmarks-dot-jl/builds/1326#01898802-ba51-4cd5-a31f-6c9b937b6146`

[5] `https://github.com/rveltz/PiecewiseDeterministicMarkovProcesses.jl`

[6] `https://github.com/X-DataInitiative/tick`

fastest algorithms. As shown in Algorithm 5, every sampled point in `Coevolve` requires a number of expected updates equal to the expected degree of the dependency graph. Therefore, it is able to complete non-exploding simulations efficiently.

The Python library `Tick` remains competitive for smaller problems, but gets considerably slower for bigger ones. Also, it is only specialized to the Hawkes process. Another drawback is that the library wraps the actual `C++` implementation. In contrast, `JumpProcesses.jl` can simulate many other point processes with a relatively simple user-interface provided by the Julia language.

There is substantial difference between the performance of recursive *CHV simple* and *CHV full*. The former does not make use of the derivative of the intensity function in Equation 6.4 which is more efficient to compute than the recursive rate in Equation 6.6.

On the one hand, `Coevolve` clearly dominates *CHV simple*. On the other hand, *CHV full* is slower for smaller networks, but slightly faster than `Coevolve` for larger models. This change in relative performance occurs due to the rate of rejection in `Coevolve` increasing in model size for this particular model. We compute the rejection rate as one minus the ratio between the number of jumps and the number of calls to the upper bound. A system with a single node sees a rejection rate of around 8 percent which rapidly increases to 80 percent when the system reaches 20 nodes and plateaus at around 95 percent with 95 nodes.

Finally, we introduce a new benchmark which is intended to assess the performance of algorithms capable of simulating the stochastic model of hippocampal synaptic plasticity with geometrical readout of enzyme dynamics proposed in [18]. For short, we denote it as the synapse model. We chose to benchmark this model as it is representative of a complex biochemical model. It couples a jump problem containing 98 jumps affecting 49 discrete variables with a stiff, ordinary differential equation problem containing 34 continuous variables. Continuous variables affect jump rates while the discrete variables affect the continuous problem. There are 3 stages to the simulation: pre-synaptic evolution, glutamate release, and post-synaptic evolution. Among the algorithms considered, only the *inverse* method implemented in `JumpProcesses.jl`, `Coevolve` and *CHV* are theoretically able to simulate the synapse model. However, in practice, only the last two complete at least one benchmark run. The original synapse problem was described as a piecewise deterministic Markov process, so we do not make the distinction between *CHV simple* and *full* in this benchmark.

Benchmark results are displayed in Table 3. We observe that *CHV* is the fastest algorithm completing the synapse evolution in about half of the time it takes `Coevolve` with less than half of the allocations. Further investigation reveals that the thinning procedure in `Coevolve` reaches an average of 70 percent over all jumps which then leads to 2 to 3 times more function evaluations and Jacobians created compared to *CHV*. Our implementation adds stopping times via a call to `register_next_jump_time!` even for rejected jumps — we do not know a jump will be rejected until evaluated. This then leads the ODE solver to step to those times and make additional function evaluations and Jacobians. Lemaire *et al.* [11] performs a similar benchmark in which they compare the Hodgkin-Huxley model against different thinning conditions and against an ODE approximation. They find that a thinned algorithm with optimal boundary conditions can run significantly faster than the ODE approximation. Thus, there could be plenty of room to improve the performance of `Coevolve` in our setting.

A disadvantage of *CHV* compared with `Coevolve` is that it supports limited saving options by design. To save at pre-specified times would require using the fact that solutions are piecewise constant to determine solutions at times in-between jumps — and for coupled ODE-jump problems would require root-finding to determine when $s(u) = s_n$ for each desired saving time s_n in Equation 4.8. The alternative proposed in [23] is to introduce an artificial jump to the model such as the homogeneous Poisson process with unit rate to sample the system at regular intervals. Alternatively, `Coevolve` allows saving at any arbitrary point. A common workflow in simulating jump processes, particularly when interested in calculating statistics over time, is to pre-specify a precise set of times at which to save a simulation. In theory, this reduces memory pressure, particularly for systems with large numbers of jumps, and can give increased computational performance relative to saving the state at the occurrence of every jump. However, in the case of the synapse model, the number of candidate time rejections far surpasses the number of jumps. Therefore, reducing the number of saving points — *e.g.* only saving at start and end of the simulation — does not significantly reduce allocations or running time. Given these considerations, we decided to save after every jump and at regular pre-specified intervals that occur at the same frequency as the artificial saving jump used by *CHV*.

Another parameter that can affect the precision and speed of the synapse benchmark is the ODE solver. The author of `PiecewiseDeterministicMarkovProcesses.jl` discuss some of these issues in Discourse[7]. Some ODE solvers can be faster and more precise. Due to time constraints, we have not investigated this matter. The synapse benchmark uses the `AutoTsit5(Rosenbrock23())` solver in both `Coevolve` and *CHV*. Further investigation of this matter is left to future research.

7. Conclusion

This paper demonstrates that `JumpProcesses.jl` is a fast, general-purpose library for simulating evolutionary point processes. With the addition of `Coevolve`, any point process on the real line with a non-negative, left-continuous, history-adapted and locally bounded intensity rate can be simulated with this library. The objective of this paper was to bridge the gap between the treatment of point process simulation in statistics and biochemistry. We demonstrated that many of the algorithms developed in biochemistry which served as the basis for the `JumpProcesses.jl` aggregators can be mapped to three general methods developed in statistics for simulating evolutionary point processes. We showed that the existing aggregators mainly differ in how they update and sample from the intensity rate and mark distribution. As we performed this exercise, we noticed the lack of an efficient aggregator for variable intensity rates in `JumpProcesses.jl`, a gap which `Coevolve` is meant to fill.

`Coevolve` borrows many enhancements from other aggregators in `JumpProcesses.jl`. However, there are still a number of ways forward. First, given the performance of the *CHV* algorithm in our benchmarks, we should consider adding it to `JumpProcesses.jl` as another aggregator so that it can benefit from tighter integration with the SciML organization and libraries. The saving behavior of *CHV* might pose a challenge when bringing this algorithm to the library. We could leverage the connection between *inverse* and *thinning* methods illustrated in Subsection 4.2 to attempt to develop a version of this algorithm that can evolve in synchrony with model time. Second, the new aggregator depends on the user providing bounds on the jump rates as well as the duration of their validity. In practice, it can be difficult to determine

[7] `https://discourse.julialang.org/t/help-me-beat-lsoda/88236`

		Brute Force			Recursive				
	V	***Inverse***	`Coevolve`	***CHV simple***	***Inverse***	`Coevolve`	***CHV simple***	***CHV full***	***Tick***
Time	**1**	113.7 μs	**4.8 μs**	174.2 μ s	112.1 μs	5.1 μs	175.6 μs	173.1 μs	31.4 μs
	10	17.5 ms	211.8 μs	4.8 ms	11.0 ms	**76.1 μs**	432.4 μs	579.0 μs	179.0 μs
	20	139.1 ms	1.5 ms	50.7 ms	59.3 ms	**282.9 μs**	924.7 μs	884.4 μs	1.2 ms
	30	415.3 ms *n=25*	3.3 ms	133.0 ms	200.0 ms	**516.9 μs**	1.7 ms	1.3 ms	3.7 ms
	40	2.2 s *n=5*	8.2 ms	342.0 ms *n=30*	1.6 s *n=7*	**1.0 ms**	2.5 ms	1.6 ms	9.2 ms
	50	5.1 s *n=2*	16.9 ms	722.0 ms *n=14*	3.4 s *n=3*	**1.6 ms**	3.7 ms	2.0 ms	21.2 ms
	60	8.5 s *n=2*	37.7 ms	1.3 s *n=8*	6.2 s *n=2*	**2.3 ms**	5.1 ms	2.5 ms	45.0 ms
	70	14.2 s *n=1*	59.5 ms	2.1 s *n=5*	10.9 s *n=1*	3.3 ms	6.8 ms	**3.0 ms**	87.5 ms
	80	22.2 s *n=1*	88.3 ms	3.3 s *n=3*	15.2 s *n=1*	4.2 ms	9.0 ms	**3.3 ms**	142.2 ms
	90	35.8 s *n=1*	139.7 ms	6.2 s *n=2*	24.6 s *n=1*	5.5 ms	11.9 ms	**3.8 ms**	241.9 ms

Table 2. : Median execution time for the compound Hawkes process, V is the number of nodes and n is the total number of successful executions under ten seconds. Brute force refers to the implementation of the intensity rate looping through the whole history of past events. Recursive refers to a recursive implementation that only requires looking at the previous state of each node. *Inverse* and `Coevolve` are algorithms from `JumpProcesses.jl`, *CHV* is an algorithm from `PiecewiseDeterministicMarkovProcesses.jl`. See Subsection 4.1 for the distinction between *CHV simple* and *CHV full*. `Tick` is a Python library. All simulations were run 50 times except when stated otherwise under the running time. Fastest time is **bold**, second fastest underlined. Benchmark source code and dependencies are available in `SciMLBenchmarks.jl`, see first paragraph of Section 6.2 for source references.

	Time	**Allocation**
Inverse	-	-
`Coevolve`	4.9 s	95.2 Mb
CHV	**2.4 s**	**43.8 Mb**

Table 3. : Median execution time and memory allocation. All simulations were run 50 times, a dash indicates that no runs were successful. Fastest time is **bold**, second fastest underlined. Benchmark source code and dependencies are available in `SciMLBenchmarks.jl`, see first paragraph of Section 6.2 for source references.

these bounds a priori, particularly for models with many ODE variables. Moreover, determining such bounds from an analytical solution or the underlying ODEs does not guarantee their holding for the numerically computed solution (which is obtained via an ODE discretization), and so modifications may be needed in practice. A possible improvement would be for `JumpProcesses.jl` to determine these bounds automatically taking into account the derivative of the rates. Deriving efficient bounds require not only knowledge of the problem and a good amount of analytical work, but also knowledge about the numerical integrator. At best, the algorithm can perform significantly slower if a suboptimal bound or interval is used, at worst it can return incorrect results if a bound is incorrect — *i.e.* it can be violated inside the calculated interval of validity. Third, `JumpProcesses.jl` would benefit from further development in inexact methods. At the moment, support is limited to processes with constant rates between jumps and the only solver available `SimpleTauLeaping` does not support marks. Inexact methods should allow for the simulation of longer periods of time when only an event count per time interval is required. Hawkes processes can be expressed as a branching process. There are simulation algorithms that already take advantage of this structure to leap through time [10]. It would be important to adapt these algorithms for general, compound branching processes to cater for a larger number of settings. Finally, `JumpProcesses.jl` also includes algorithms for jumps over two-dimensional spaces. It might be worth conducting a similar comparative exercise to identify algorithms in statistics for 2- and N-dimensional processes that could also be added to `JumpProcess.jl` as it has the potential to become the go-to library for general point process simulation.

8. Acknowledgements

This project has been made possible in part by grant number 2021-237457 from the Chan Zuckerberg Initiative DAF, an advised fund of Silicon Valley Community Foundation. SAI was also partially supported by NSF-DMS 1902854.

9. References

[1] Daryl J. Daley and David Vere-Jones. *An Introduction to the Theory of Point Processes: Volume I: Elementary Theory and Methods*. Probability and Its Applications, An Introduction to the Theory of Point Processes. Springer-Verlag, 2 edition. doi:10.1007/b97277.

[2] Mehrdad Farajtabar, Yichen Wang, Manuel Gomez-Rodriguez, Shuang Li, Hongyuan Zha, and Le Song. COEVOLVE: A joint point process model for information diffusion and network evolution. 18(1). doi:10.5555/3122009.3122050.

[3] Michael A. Gibson and Jehoshua Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. 104(9). doi:10.1021/jp993732q.

[4] Daniel T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. 115(4). doi:10.1063/1.1378322.

[5] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. 81(25). doi:10.1021/j100540a008.

[6] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. 22(4). doi:10.1016/0021-9991(76)90041-3.

[7] Abhishekh Gupta and Pedro Mendes. An Overview of Network-Based and -Free Approaches for Stochastic Simulation of Biochemical Systems. 6(1). doi:10.3390/computation6010009.

[8] Petter Holme. Fast and principled simulations of the SIR model on temporal networks. 16(2). doi:10.1371/journal.pone.0246961.

[9] Günter Last and Mathew Penrose. *Lectures on the Poisson Process*. Cambridge University Press, 1st edition edition.

[10] Patrick J. Laub, Young Lee, and Thomas Taimre. *The Elements of Hawkes Processes*. Springer International Publishing. doi:10.1007/978-3-030-84639-8.

[11] Vincent Lemaire, Michèle Thieullen, and Nicolas Thomas. Exact Simulation of the Jump Times of a Class of Piecewise Deterministic Markov Processes. 75(3). doi:10.1007/s10915-017-0607-4.

[12] P. A. W. Lewis and G. S. Shedler. Simulation of Nonhomogeneous Poisson Processes with Log Linear Rate Function. 63(3). doi:10.2307/2335727. jstor:2335727.

[13] Luca Marchetti, Corrado Priami, and Vo Hong Thanh. *Simulation Algorithms for Computational Systems Biology*. Texts in Theoretical Computer Science. An EATCS Series. Springer International Publishing. doi:10.1007/978-3-319-63113-4.

[14] James M. McCollum, Gregory D. Peterson, Chris D. Cox, Michael L. Simpson, and Nagiza F. Samatova. The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior. 30(1). doi:10.1016/j.compbiolchem.2005.10.007.

[15] James Meiss. *Differential Dynamical Systems, Revised Edition*. Mathematical Modeling and Computation. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611974645.

[16] Y. Ogata. On Lewis' simulation method for point processes. 27(1). doi:10.1109/TIT.1981.1056305.

[17] Christopher Rackauckas and Qing Nie. DifferentialEquations.jl A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. 5(1). doi:10.5334/jors.151.

[18] Yuri E. Rodrigues, Cezar M. Tigaret, Hélène Marie, Cian ODonnell, and Romain Veltz. A stochastic model of hippocampal synaptic plasticity with geometrical readout of enzyme dynamics. doi:10.1101/2021.03.30.437703.

[19] Howard Salis and Yiannis Kaznessis. Accurate hybrid stochastic simulation of a system of coupled chemical or biochemical reactions. 122(5). doi:10.1063/1.1835951.

[20] Alexander Slepoy, Aidan P. Thompson, and Steven J. Plimpton. A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks. 128(20). doi:10.1063/1.2919546.

[21] Vo Hong Thanh, Corrado Priami, and Roberto Zunino. Efficient rejection-based simulation of biochemical reactions with stochastic noise and delays. 141(13). doi:10.1063/1.4896985.

[22] Vo Hong Thanh, Roberto Zunino, and Corrado Priami. Efficient Constant-Time Complexity Algorithm for Stochastic Simulation of Large Reaction Networks. 14(3). doi:10.1109/TCBB.2016.2530066.

[23] Romain Veltz. A new twist for the simulation of hybrid systems using the true jump method. doi:10.48550/arXiv.1504.06873. arxiv:1504.06873.