juliacon

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

Guilherme Augusto Zagatti[1], Samuel A. Isaacson[3], Christopher Rackauckas[4], 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

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 Doobs method, Gillespies 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 reveals 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. 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. Such methods are hard to scale for large populations where numerous events 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 common in biochemistry, which often requires the simulation of chemical reactions in large systems.

Previously, most of the focus has been on either univariate processes with exotic intensities or large systems with conditionally constant intensities, but not on both. As such, there was no readily available general-purpose software for simulating compound point processes in large systems with time-dependent rates. We contribute a simulation algorithm `Coevolve` to `JumpProcesses.jl` for efficiently simulating such processes. The implemented algorithm improves the 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 do not change between jumps, and avoids the unnecessary re-computation of randomly generated numbers, as well as the computation of the intensity rate when its lower bound is available. The extension of `JumpProcesses.jl` significantly expands the type of models that can be simulated with the library, including 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 to incorporate compound point processes with time-dependent rates in a wide variety of applications.

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 understanding 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 unidimensional support. They are exhaustively described in [1]. The likelihood of any evolutionary point process is fully characterized by its conditional intensity and mark distribu-

tion :

$$\lambda^*(t) \equiv \lambda(t \mid H_{t^-}) = \frac{\Pr(t \mid H_{t^-})}{1 - \Pr(t \mid H_{t^-})} \text{ and } f^*(k \mid t_n)$$

Where $H_{t^-} = \{(t_n, k_n) \mid 0 \leq t_n \leq t\}$ denotes the internal history of the process up to but not including t, and the superscript $*$ denotes the conditoning of a function on the internal history. We can interpret the conditonal 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^-}.

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. We know that the distance between any two points in a homogeneous process is distributed according to the exponential distribution — see Theorem 7.2 [8]. Given the homogeneous process with intensity λ, then the distance Δt between two points is distributed according to $\Delta t \sim \exp(1/\lambda)$. 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 $T = -ln(V) / \lambda \sim \exp(1)$. 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 [8]. 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 [1] can be transformed to a homogeneous Poisson process with unit rate via the following formula:

$$\int_{t_{n-1}}^{t_n} \lambda^*(u)\, du = \Delta\tilde{t}_n \,,\, \Delta\tilde{t}_n \sim \exp(1) \tag{1}$$

The idea is to draw realizations from the unit rate Poisson process and solve Equation 1 above to obtain the desired point process. In Algorithm 1, we adapt Algorithm 7.4 [1]. The advantage of using the inverse method is that it is easier to couple the point process with differential equations and solve the system with the same numerical integrator.
Whenever the conditional intensity is constant between two points, Equation 1 can be solved analytically. Let $\lambda^*(t) = \lambda_{n-1}, \forall t_{n-1} \leq t < t_n$, then

$$\int_{t_{n-1}}^{t_n} \lambda^*(u)\, du = \Delta\tilde{t}_{n-1} \iff$$
$$\lambda_{n-1}(t_n - t_{n-1}) = \Delta\tilde{t}_n \iff$$
$$t_n = t_{n-1} + \frac{\Delta\tilde{t}_n}{\lambda_{n-1}}$$

Which is equivalent to drawing the next realization time from the re-scaled exponential distribution $\Delta t_n \sim \exp(1/\lambda_{n-1})$. 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 1 often requires a numerical solution. To get around a similar obstacle in the context of the piecewise deterministic Markov process, Veltz [19] proposes a change of variable trick that sees the root finding problem transformed into an initial value problem.
We map the evolutionary point process to a piecewise deterministic Markov process as following. Let $\varphi^*_t(\cdot)$ define a flow describing the deterministic evolution of the conditional intensity function over $t \geq 0$, $\mathbf{1}(\cdot)$ be the indicator function, and $\lambda^*_n \equiv \lambda^*(t_n)$, then the conditional intensity function can be written as a jump process:

$$\lambda^*(t) = \sum_{n \geq 1} \varphi^*_{t - t_{n-1}}(\lambda^*_{n-1}) \mathbf{1}(t_{n-1} \leq t < t_n)$$

According to Meiss [13], if $\varphi^*_t(\lambda^*)$ is a flow, then it is a solution to the initial value problem:

$$\varphi^*_0(\lambda^*_n) = \lambda^*_n \,,\, \frac{d}{dt}\varphi^*_t(\lambda^*_n) = f(\varphi^*_t(\lambda^*_n))$$

Given the definition of the conditional intensity we have that:

$$\Pr(t_n - t_{n-1} \mid \lambda^*_{n-1}) = \exp\left(-\int_0^t \varphi_u(\lambda^*_{n-1}) du\right)$$

Therefore, we have a piecewise deterministic Markov process that satisfies the conditions of Theorem 3.1 [19]. In this case, we can find t_n by solving the following initial value problem:

$$\begin{cases} \nu^*(0) = \lambda^*(t_{n-1}) \,,\, \dfrac{d}{du}\nu^*(u) = \dfrac{f(\nu^*(u))}{\nu^*(u)} \\ s(0) = t_{n-1} \,,\, \dfrac{d}{du}s(u) = \dfrac{1}{\nu^*(u)} \end{cases} \tag{2}$$

Up to $\tilde{t} \sim \exp(1)$. Veltz denotes this method *CHV*. In Algorithm 1, we can implement the CHV method by solving Equation 2 instead of Equation 1. The algorithmic complexity is then determined by the ODE solver.
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. A naive implementation of Line 9 scales with the number of marks $O(K)$ 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 $O(K)$.
Finally, Algorithm 1 is not guaranteed to terminate in finite time since one might need to sample many points before $t_n > T$. The

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

sampling rate can be especially high when simulating the process in a large population with self-exciting encounters. In biochemistry, Salis and Kaznessis [15] 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 1, 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 time of the next event t_n by solving Equation 1
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 [10]. The classical algorithm from Ogata [14] 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 [5, 4]. 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 molecule interaction boils down 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 [8].

Alternatively, in biochemistry, *thinning* methods are known as *rejection* algorithms. Than *et al.* [17, 18] 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 intensities are used for thinning. A similar procedure is also described in [16], 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 [18]. To implement the algorithm, we define three functions, $\bar{M}^*(t) = \bar{M}(t \mid H_t)$, $\underline{M}^*(t) = \underline{M}(t \mid H_t)$ and $L^*(t) = L(t \mid H_t)$, that characterize the local boundedness condition such that:

$$\lambda^*(t+u) \leq \bar{M}^*(t) \text{ and } \lambda^*(t+u) \geq \underline{M}^*(t), \forall\, 0 \leq u \leq L^*(t)$$

The tighter the bound on $\bar{M}^*(t)$, 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 $\bar{M}^*(t)$ and the duration of the local interval $L^*(t)$. On the other hand, the infimum $\underline{M}^*(t)$ can be used to avoid the evaluation of $\lambda^*(t+u)$ in Line 10 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 $\bar{M}^*(t) = \underline{M}^*(t) = \lambda_{n-1}$ and $L^*(t) = \infty$. We have that for any $u \sim \exp(1 \,/\, \bar{M}^*(t)) = \exp(1/\lambda_{n-1})$ and $v \sim U[0, 1]$, $u < L^*(t) = \infty$ and $v < \lambda^*(t+u) \,/\, \bar{M}^*(t) = 1$. Therefore, we advance the internal history for every iteration of Algorithm 2. In this case, the bound $\bar{M}^*(t)$ is as tight as possible, and this method becomes the same as the *inverse* method of Subsection 4.1.

While *thinning* algorithms solve the issue by computing the inverse of the integrated conditional intensity, the issue of termination can be aggravated by the fact that we are now required to sample from a process with a rate higher than 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 11 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 $H_{T^-} \leftarrow \{\}$
3: set $n \leftarrow 0, t \leftarrow 0$
4: **while** true **do**
5: $t \leftarrow \mathrm{NextTimeViaThinning}([t, T), H_{T^-}, \lambda^*)$
6: **if** $t \geq T$ **then**
7: **break**
8: **end if**
9: $n \leftarrow n + 1$
10: $t_n \leftarrow t$
11: update f^* and draw the mark $k_n \sim f^*(k \mid t_n)$
12: update the history $H_{T^-} \leftarrow H_{T^-} \cup (t_n, k_n)$
13: **end while**
14: **return** H_{T^-}
15: **end procedure**

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

1: **procedure** NextTimeViaThinning($[t, T)$, λ^*, H_t,)
2: **while** $t < T$ **do**
3: update λ^*
4: find $\bar{M}^*(t)$, $\underline{M}^*(t)$ and $L^*(t)$ which satisfies Equation 4.2
5: draw $u \sim \exp(1 \;/\; \bar{M}^*(t))$ and $v \sim U[0,1]$
6: **if** $u > L^*(t)$ **then**
7: $t \leftarrow t + L^*(t)$
8: **next**
9: **end if**
10: **if** $(v > \underline{M}^*(t))$ and $(v > \lambda^*\,(t+u) \;/\; \bar{M}^*(t))$ **then**
11: $t \leftarrow t + u$
12: **next**
13: **end if**
14: $t \leftarrow t + u$
15: **break**
16: **end while**
17: **return** t
18: **end procedure**

4.3 Queueing methods

As an alternative to his *direct* method — in this text referred as the *thinning* method —, Gillespie introduced the *first reaction* method in his seminal work on simulation algorithms [5]. The *first reaction* method separately simulates the next reaction time for each reaction channel. 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 *thinning* method to simulate a set of inter-dependent point processes, making use of the *superposition theorem* — Theorem 3.3 [8] — 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 [7] simulation. In both cases, the phenomena are modelled as point processes.

We prefer to call this class of methods *queueing* methods since most efficiency gains come from maintaining a priority queue of the next event times. Algorithm 4 presents a method for sampling a superposed point process by keeping the strike time of each process in a priority queue Q. The priority queue is initially constructed in $O(K)$ steps in Lines 4 to 7 of Algorithm 4. But, in contrast to *thinning* methods, updates to the conditional intensity depend only on the size of the neighbourhood of k. That is, the processes k' whose conditional intensity depends on the history of k. If the graph is sparse, then updates will be faster than with *thinning*. Another advantage of *queueing* is that, since marks are determined according to their priority in the queue, it is possible to simulate point processes with a finite space of marks as 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.

Algorithm 4 The *queueing* method for simulating a marked evolutionary point process over a fixed duration of time $[0, T)$.

1: **procedure** QueueingMethod($[0, T)$, λ_k^*, f_k^*,)
2: initialize the history $H_{T^-} \leftarrow \{\}$
3: set $n \leftarrow 0, t \leftarrow 0$
4: **for** k=1,K **do**
5: $t \leftarrow \text{NextTimeViaThinning}([0,T), H_{T^-}, \lambda_k^*(\cdot))$
6: push t_k, k to Q
7: **end for**
8: **while** $t < T$ **do**
9: pop t, k from Q
10: **if** $t \geq T$ **then**
11: **break**
12: **end if**
13: $n \leftarrow n + 1$
14: $t_n \leftarrow t$
15: update f^* and draw the mark $k_n \sim f^*\,(k \mid t_n)$
16: update the history $H_{T^-} \leftarrow H_{T^-} \cup (t_n, k_n)$
17: **for** $k' \in \{k\} \cup \text{Neighborhood}(k)$ **do**
18: $t_{k'} \leftarrow \text{NextTimeViaThinning}([t,T), H_{T^-}, \lambda_{k'}^*(\cdot))$
19: push $t_{k'}, k'$ to Q
20: **end for**
21: **end while**
22: **return** H_{T^-}
23: **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.

Since *inverse* methods solve a differential equation to determine the next jump time, they require a continuous numerical integrator. This facility is provided by `OrdinaryDiffEq.jl`, which easily interoperates with `JumpProcesses.jl` as it also belongs to the SciML organization. All point processes to be solved via the *inverse* method must be initialized as a `VariableRateJump`. `JumpProcesses.jl` builds a continuous callback following the algorithm in [15] and passes the problem to the `OrdinaryDiffEq.jl` solver.

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 eleven different aggregators, seven of which implements a variation of the *thinning* method and four 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 K interdependent processes. How-

ever, this can be easily conciliated 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 mark distribution becomes the categorical distribution weighted by the intensity of each process. Conditional on the selected process, the corresponding `affect!(integrator)` is invoked. Thus, the mark distribution can be re-written as $a_n \sim f^*(a|k_n, t_n)f^*(k|t_n)$. Moreover, since the intensity between jumps is constant, Algorithm 3 short-circuits to quickly return $t \sim \exp(1/\bar{M}) = \exp(1/\lambda_n)$ as discussed in Subsection 4.2.

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 [5] 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 [16]. `SortingDirect` keeps the intensity rate in a loosely sorted array following [12]. 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 $\bar{M}$ equals to the maximum rate. However, the implementation differs. It thins with $\bar{M} = \lambda_n$, 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 queue is used in `RSSACR`. Both of these algorithms follow from [17, 18].

Next, we consider the *queuing* aggregators. Starting with aggregators that only support `ConstantRateJump` we have, `FRM`, `FRMFW` and `NRM`. `FRM` and `FRMFW` follows the *first reaction* method in [5]. To compute the next jump, both algorithms compute the time to next event for each process and selects the process with minimum time. This is equivalent to assuming a complete dependency graph in Algorithm 4. For large systems, they can be less efficient than `NRM`. The latter implementation is sourced from [3] and follows Algorithm 4 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 only aggregator that allows `VariableRateJump` to open the door for the efficient simulation of processes with time-dependent intensity rates. Our implementation takes inspiration from [2]. It improves it in several fronts. First, we take advantage of its modularity 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 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 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].

6. Benchmarks

This section compares the algorithms described in Section 5 implemented in `JumpProcesses.jl`. Since `Coevolve` is a new aggregator and the only one to support `VariableRateJump` we proceed in three steps. First, we test the correctness of `Coevolve` by conducting some statistical analysis. Second, we compare all the existing aggregators with the jump benchmarks in `SciMLBenchmarks.jl`. `Coevolve` should attain similar performance to `NRM`. Finally, we propose a new benchmark for compound, self-exciting point processes to evaluate the performance of the `Coevolve` aggregator against the alternative which uses an *inverse* method. We also use this new benchmark to compare across another python library for simulating Hawkes processes.

To simulate a process intended for a discrete solver with `JumpProcesses.jl`, we define the 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 a discrete stepper. The code for simulating the homogeneous Poisson process with `Coevolve` is reproduced in Listing 1.

Listing 1: Simulation of the homogeneous Poisson process.

```julia
using JumpProcesses
rate(u, p, t) = p[1]
affect!(integrator) = integrator.u[1] += 1
jump = ConstantRateJump(rate, affect!)
u, tspan, p = [0.], (0., 200.), (0.25,)
dprob = DiscreteProblem(u, tspan, p)
jprob = JumpProblem(dprob, Coevolve(), jump;
  dep_graph=[[1]])
sol = solve(jprob, SSAStepper())
```

The simulation of a Hawkes process 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 — shown below —, but we keep the code here as close as possible to its usual definition for illustration purposes.

Listing 2: Simulation of the Hawkes process.

```julia
using JumpProcesses
h = Float64[]
rate(u, p, t) = p[1] +
  p[2]*sum(exp.([-p[3]*(t-_t) for _t in h]))
lrate(u, p, t) = p[1]
urate = rate
```

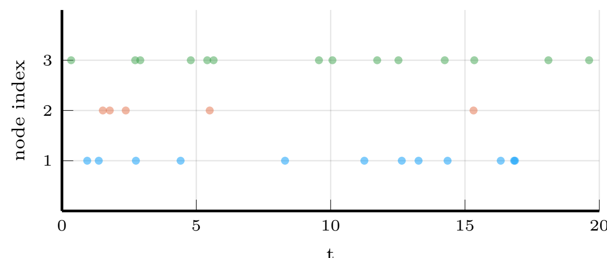
```
rateinterval(u, p, t) = 1/(2*urate(u,p,t))
affect!(integrator) = (push!(h, integrator.t);
  integrator.u[1] += 1)
jump = VariableRateJump(rate, affect!; lrate,
  urate, rateinterval)
u, tspan, p = [0.], (0., 200.), (0.25, 0.5, 2.0)
dprob = DiscreteProblem(u, tspan, p)
jprob = JumpProblem(dprob, Coevolve(), jump;
  dep_graph=[[1]])
sol = solve(jprob, SSAStepper())
```

To assess the correctness of the `Coevolve` aggregator, we add it to the `JumpProcesses.jl` test suite. Some tests in the suite check whether the aggregators are able to obtain empirical statistics close to the expected one 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.

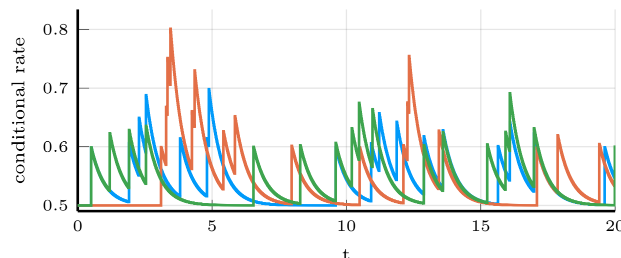
In addition to that, the correctness of the implemented algorithm can be visually assessed using a QQ-plot. As discussed in Subsection 4.1, every simple point process can be transformed to a Poisson process with unit rate. This implies that the interval between points for any such transformed process should match the exponential distribution. Therefore, the correctness of the `Coevolve` aggregator can be assessed as following. First, transform the simulated intervals with the appropriate compensator. Compute the empirical quantiles of the transformed intervals. Plot the empirical quantiles with the corresponding quantiles of the exponential distribution. If the simulator produces correct trajectories, this plot known as QQ-plot should depict the points aligned around the 45-degree line. We produce QQ-plots for the homogeneous Poisson process as well as the compound Hawkes process — see Section 3.6 [9] for a definition — to attest the correctness of `Coevolve`. Figure 1 (d) depicts the QQ-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.

Next, we assess the speed of the different aggregators. `SciMLBenchmarks.jl` includes four jump benchmarks for point processes over the real line: a 1-dimensional continuous time random walk approximation of a diffusion model (Diffusion), the multi-state model from Appendix A.6 [11] (Multi-state), a simple negative feedback gene expression model (Gene I) and the negative feedback gene expression from [6] (Gene II). The intensity rate of all the modelled processes in these benchmarks are constant between jumps. 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. We also note that the performance of `Coevolve` matches `NRM` except for the last problem. Further investigation is needed to understand this difference in performance since the former should reduce to the latter when no `VariableRateJump` is used.

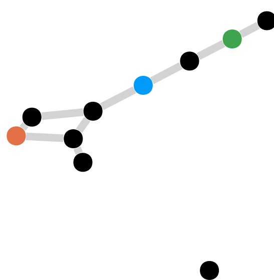
Finally, we add a new benchmark to the benchmark suite which simulates the compound Hawkes process for an increasing number



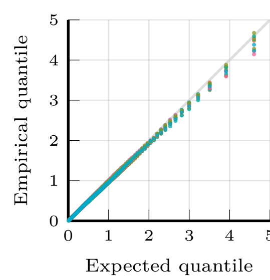
(a)



(b)



(c)



(d)

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) QQ-plot of transformed inter-event time for 250 simulations colored by node.

Benchmark	Diffusion	Multi-state	Gene I	Gene II
`Direct`	4.85 s	0.12 s	0.17 ms	0.44 s
`FRM`	15.88 s	0.22 s	0.24 ms	0.62 s
`SortingDirect`	1.17 s	0.12 s	0.21 ms	0.39 s
`NRM`	0.87 s	0.27 s	0.44 ms	0.75 s
`DirectCR`	0.43 s	0.20 s	0.38 ms	0.79 s
`RSSA`	1.91 s	0.11 s	0.35 ms	0.59 s
`RSSACR`	0.37 s	0.16 s	0.73 ms	0.83 s
`Coevolve`	0.77 s	0.28 s	0.40 ms	0.96 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 [11] (Multi-state), a simple negative feedback gene expression model (Gene I) and the negative feedback gene expression from [6] (Gene II).

of self-exciting processes. Let a graph with V nodes, then 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

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

This process is known as self-exciting, because the occurence 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.

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

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 [9]. 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.

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

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

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

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.

We assess the benchmark in five different settings. First, we run the *inverse* method and `Coevolve` using the brute force implementation of the intensity rate which loops through the whole history of past events. Second, we implement a recursive algorithm for computing the intensity rate which we use to run both methods. We also run the benchmark against `PiecewiseDeterministicMarkovProcesses.jl` [2] which is developed by the same author who proposed the *CHV* algorithm discussed in Subsection 4.1. Finally, we run the benchmark using the Python library Tick [3]. This library implements a version of the thinning method for simulating the process and implements a recursive algorithm for computing the intensity rate.

Table 2 shows that `Coevolve` is orders of magnitude faster than the *inverse* method for any system size. As shown in Algorithm 4, 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. Instead, the *inverse* method is unable to complete within the allocated time for larger systems because it is required to find an ever larger number of roots of an ever larger system of differential equations.

The recursive implementation of the intensity rate also brings considerable performance boost placing `Coevolve` as one of the fastest algorithms. 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. Finally, *CHV* is slower for smaller networks, but slightly faster than `Coevolve` for larger models. The fact that *CHV* is not well-integrated with the SciML organization might pose a challenge for some users.

7. Conclusion

This paper demonstrates that `JumpProcesses.jl` is a fast, general-purpose library for simulating evolutionary point processes. With the addition of the `Coevolve` aggregator, 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 enhacements from other aggregators in `JumpProcesses.jl`. However, there are still a number of ways forward. First, the aggregator cannot simulate processes whose rate depends on variables from differential equations. This extension would allow two-way interoperability with the rest of the SciML family. Next, we should consider whether it is possible to reduce the number of iterations during thinning. Processes get updated once its dependencies change. Once we know a process will not fire before one of its dependencies, we should be able to stop the iteration and wait for the update or continue from where the iteration stopped if the dependency gets delayed. This might become important once we allow rates that depend on differential equations. Third, given the performance of the *CHV* algorithm in our benchmarks, we should consider adding it to `JumpProcesses.jl` as another aggregator such that it can benefit with tighter integration with the SciML organization. Fourth, `JumpProcesses.jl` would benefit from further development in inexact methods. At the moment, support is limited to processes with constant rates between jumps and 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 [9]. 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

[2] `https://github.com/rveltz/PiecewiseDeterministicMarkovProcesses.jl`
[3] `https://github.com/X-DataInitiative/tick`

	Brute force				**Recursive**							
V	***Inverse***		`Coevolve`		***Inverse***		`Coevolve`		***CHV***		`Tick`	
	n	time	n	time	n	time	n	time	n	time	n	time
1	50	143.4 μs	50	2.6 μs	50	104.3 μs	50	**4.1 μs**	50	150.0 μs	50	28.3 μs
10	50	16.0 ms	50	342.0 μs	50	9.8 ms	50	**96.0 μs**	50	472.2 μs	50	134.6 μs
20	50	106.8 ms	50	2.5 ms	50	47.0 ms	50	**356.1 μs**	50	713.3 μs	50	933.5 μs
30	29	344.8 ms	50	5.7 ms	50	158.0 ms	50	**649.2 μs**	50	1.2 ms	50	3.0 ms
40	6	1.9 s	50	13.6 ms	9	1.2 s	50	**1.2 ms**	50	1.4 ms	50	7.3 ms
50	3	3.6 s	50	26.0 ms	5	2.4 s	50	**1.8 ms**	50	1.8 ms	50	14.3 ms
60	2	6.9 s	50	45.6 ms	3	4.3 s	50	2.5 ms	50	**2.5 ms**	50	28.9 ms
70	2	9.5 s	50	72.3 ms	2	6.6 s	50	3.5 ms	50	**2.9 ms**	50	56.9 ms
80	1	15.4 s	50	118.7 ms	1	10.9 s	50	4.3 ms	50	**3.5 ms**	50	97.8 ms
90	1	26.0 s	50	146.3 ms	1	17.4 s	50	5.3 ms	50	**4.4 ms**	50	160.8 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` and `Tick` is a Python library. Fastest time is **bolded**, second fastest underlined.

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. References

[1] D. J. Daley and D. 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. Exact stochastic simulation of coupled chemical reactions. 81(25). doi:10.1021/j100540a008.

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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