

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

Guilherme Augusto Zagatti¹, Samuel A. Isaacson³, Christopher Rackauckas⁴, Vasily Ilin⁵, See-Kiong Ng^{1,2}, and Stéphane Bressan^{1,2}

¹Institute of Data Science, National University of Singapore, Singapore

²School of Computing, National University of Singapore, Singapore

³Department of Mathematics and Statistics, Boston University

⁴Computer Science and AI Laboratory (CSAIL), Massachusetts Institute of Technology

⁵Department of Mathematics, University of Washington

1 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 the `JumpProcesses.jl` library, which was first developed for simulating jump processes via stochastic simulation algorithms (SSAs) — including Doob’s method, Gillespie’s methods, and Kinetic Monte Carlo methods — also provides performant methods [**G: for**] sampling temporal point processes (TPPs). 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 TPPs. Our comparative exercise reveals that the library lacked an efficient algorithm for simulating processes with variable intensity rates. We develop [**G:a new simulation algorithm Coevolve. This is**] the first thinning algorithm to step in sync with model time reducing the number of time proposal rejections and allowing for new possibilities such as simulating variable-rate jump processes coupled with differential equations [**G: We implement the new algorithm in**] `JumpProcesses.jl` [**G:, which**] can now simulate any point process on the real line with a non-negative, left-continuous, history-adapted and locally bounded intensity rate efficiently, enabling the library to become one of the few readily available, fast and general-purpose options for simulating TPPs.

1. Introduction

Methods for simulating the trajectory of temporal point processes (TPPs) can be split into exact and inexact methods. Exact methods generate statistically exact realizations of each point in the process chronologically ¹. This exactness provides [**G: unbiased correct**]

¹Some exact methods might not be completely exact since they rely on root finding approximation methods. However, we follow convention and denote all such methods as exact methods.

samples, but can suffer from reduced performance when simulating systems 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 [9], which often require the simulation of chemical reactions in systems with large molecular populations. Previously, the development of point process simulation libraries 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 [**G: multivariate TPPs in large systems with time-dependent rates. To enable the efficient simulation of such systems, we contribute a new simulation algorithm for multivariate TPPs. Our new method is a type of thinning algorithm that thins in sync with time. This allows the coupling of large multivariate TPPs with other algorithms that step chronologically through time such as differential equation solvers. Our new algorithm improves the COEVOLVE algorithm from [5]. COEVOLVE itself can be seen as an improvement from the next reaction method of [6]. We can trace the idea of synced thinning back to Section 7.5 [3] where it is discussed, but no algorithmic implementation of such idea existed until now. COEVOLVE [5] did not entertain jump processes that belong to systems of differential equations.**]

[**G: Our new algorithm is implemented as the Coevolve aggregator in JumpProcesses.jl, a core sub-library of the popular DifferentialEquations.jl library [22]. It is named after COEVOLVE [5] as our new algorithm supersedes the original one. The new aggregator**] 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 by it. Widely-used point processes with such intensities include compound inhomogeneous Poisson process, Hawkes processes, stress-release processes and piecewise deterministic Markov processes (PDMPs).

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. Our new additions are available as of `JumpProcesses.jl` 9.7².

This paper starts by bridging the gap between simulation methods developed in statistics and biochemistry, which led us to the development of `Coevolve`. We briefly introduce TPPs and simulation methods for the homogeneous Poisson process, which serve as building blocks for all other simulation methods. Then, we identify and discuss three types of exact simulation methods. In the second part of this paper, we describe the algorithms implemented 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 temporal point process

The TPP is a stochastic collection of marked points over a one-dimensional support. They are exhaustively described in [3]. The likelihood of any TPP is fully characterized by its conditional intensity,

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

and conditional mark distribution, $f^*(k \mid t)$ — see Chapter 7 [3]. **[G: A mark is any random attribute associated with a point.]** Here $H_{t-} = \{(t_n, k_n) \mid 0 \leq t_n < t\}$ denotes the history of the process up to but not including t . **[G: In other words, the history is a sequence of tuples with the timestamp and mark of each event.]** 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 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 [14]. 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 [14]. 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 independently distributed according to the uniform distribution **[G: $t_n \sim U[0, T]$]**.

4. Exact simulation methods

4.1 Inverse methods

The *inverse* method leverages Theorem 7.4.I [3] which states that every simple point process³ 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:

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

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. **[G:**

$$\Delta \tilde{t}_n \equiv \tilde{t}_n - \tilde{t}_{n-1} = \int_{t_{n-1}}^{t_n} \lambda^*(u) du \sim \exp(1) \quad (4.2)$$

] 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 [3].

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

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

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 PDMPs, Veltz [30] proposes a change of variables in time that recasts the root finding problem into an initial value problem. He denotes his method *CHV*.

PDMPs are composed of two parts: the jump process and the piecewise ODE that changes stochastically at jump times — see Lemaire *et al.* [16] for a formal definition.

[G: It is easy to employ *CHV* in our case by setting the ODE part to zero throughout time. By re-arranging Equation 4.1 and Equation 4.2, we note that it is a one-to-one mapping between t and \tilde{t} which allow us to obtain $t(\Delta \tilde{t}_n) = \Lambda^{*-1}(\tilde{t}_{n-1} + \Delta \tilde{t}_n)$ which describes the law of motion for a PDMP. Adapting from

²All examples and benchmarks in this paper use version 9.9 of the library

³A simple point process is a process in which the probability of observing more than one point in the same location is zero.

167 Veltz [30], we can determine the model jump time t_n after sam- 201
 168 pling $\Delta\tilde{t}_n \sim \exp(1)$ by solving the following initial value prob- 202
 169 lem until $\Delta\tilde{t}_n$, which we denote *CHV simple*. 203

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

170]
 171 Alternatively, when the intensity function is differentiable between 204
 172 jumps we can go even further [G: by recasting the jump problem 205
 173 as a PDMP]. Let $\lambda_n^* \equiv \lambda^*(t_n)$, then the flow $\varphi_{t-t_n}(\lambda_n^*)$ maps 206
 174 the initial value of the conditional intensity at time t_n to its value 207
 175 at time t . In other words, the flow describes the deterministic evo- 208
 176 lution of the conditional intensity function over time. Next, denote 209
 177 $\mathbf{1}(\cdot)$ as the indicator function, then the conditional intensity func- 210
 178 tion can be re-written as a jump process: 211

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

179 According to Meiss [20], if $\varphi_t(\cdot)$ is a flow, then it is a solution to 212
 180 the initial value problem: 213

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

181 where $g : \mathbb{R}^+ \rightarrow \mathbb{R}$ is the vector field of λ^* such that $d\lambda^*/dt = 214$
 182 $g(\lambda^*)$. 215

183 Based on Equation 2.1, we find that the probability of observing an 216
 184 interval longer than s given history H_{t-} is equivalent to: 217

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

185 Equations 4.5 and 4.7 define a PDMP satisfying the conditions of 218
 186 Theorem 3.1 [30]. In this case, we find t_n by solving the following 219
 187 initial value problem from 0 to $\Delta\tilde{t}_n \sim \exp(1)$. [G: 220

$$\begin{cases} \varphi_0(\lambda_{n-1}^*) = \lambda^*(t_{n-1}), \frac{d}{d\tilde{t}} \varphi_{\Delta\tilde{t}}(\lambda_{n-1}^*) = \frac{g(\lambda^*(t))}{\lambda^*(t)} \\ t(0) = t_{n-1}, \frac{dt}{d\tilde{t}}(\Delta\tilde{t}) = \frac{1}{\lambda^*(t)}. \end{cases} \quad (4.8)$$

188] This problem specifies how the conditional intensity and model 221
 189 time evolve with respect to the transformed time. The solution to 222
 190 Equation 4.2 is then given by $(t_n = t(\Delta\tilde{t}_n), \lambda^*(t(\Delta\tilde{t}_n)) = 223$
 191 $\lambda^*(t_n))$. [G: We denote this problem *CHV full*.] 224

192 In Algorithm 1, we can implement the CHV method by solving ei- 225
 193 ther Equation 4.4 or Equation 4.8 instead of Equation 4.2. Note that 226
 194 *CHV full* requires that the conditional intensity be piecewise dif- 227
 195 ferentiable. The algorithmic complexity is then determined by the 228
 196 ODE solver and no root-finding is required. In Section 6.2, we will 229
 197 show that there are substantial differences in performance between 230
 198 them with the full specification being faster. 231

199 Another concern with Algorithm 1 is updating and drawing from 232
 200 the conditional mark distribution in Line 8 [G: , and updating the 233

conditional intensity in Line 9]. Assume a process with K num- 204
 205 ber of marks. [G: A naive implementation of Line 9 scales with the 206
 207 number of marks as $O(K)$ since λ^* is usually constructed as the 208
 208 sum of K independent processes, each of which requires updating 209
 209 the conditional intensity rate. Likewise, Naively] drawing from the 210
 210 mark distribution in Line 8 usually involves drawing from a cat- 211
 211 egorical distribution whose implementations also scales with the 212
 212 number of marks as $O(K)$. 213

214 Finally, Algorithm 1 is not guaranteed to terminate in finite time 215
 215 since one might need to sample many points before $t_n > T$. The 216
 216 sampling rate can be especially high when simulating the process 217
 217 in a large population with self-exciting encounters. In biochemistry, 218
 218 Salis and Kaznessis [24] partition a large system of chemical reac- 219
 219 tions into two: fast and slow reactions. While they approximate the 220
 220 fast reactions with a Gaussian process, the slow reactions are solved 221
 221 using a variation of the inverse method. They obtain an equivalent 222
 222 expression for the rate of slow reactions as in Equation 4.2, which 223
 223 is integrated with the Euler method. 224

Algorithm 1 The *inverse* method for simulating a marked TPP over 225
 a fixed duration of time $[0, T)$. 226

```

1: procedure INVERSEMETHOD( $[0, T), \lambda^*, f^*$ )
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  $\lambda^*$ 
10:  end while
11:  return  $H_{T-}$ 
12: end procedure

```

219 4.2 Thinning methods

220 *Thinning* methods are popular approaches for simulating point 221
 221 processes. The main idea is to successively sample a homogeneous 222
 222 process, then thin the obtained points with the conditional intensity 223
 223 of the original process. As stated in Proposition 7.5.I [3], this pro- 224
 224 cedure simulates the target process by construction. The advantage 225
 225 of *thinning over inverse* methods is that the former only requires 226
 226 the evaluation of the conditional intensity function while the latter 227
 227 requires computing the inverse of its integrated form [3]. 228

228 *Thinning* algorithms have been proposed in different forms [3]. 229
 229 Shedler-Lewis [17] first suggested a thinning routine that simulated 230
 230 processes with bounded intensity over a fixed interval. Ogata's re- 231
 231 finement [21] suggests a procedure for evolving the simulation via 232
 232 local boundary conditions and fixed partitions of the real line. As 233
 233 long as the intensity conditioned on the simulated history remains 234
 234 locally bounded, it is possible to simulate subsequent points indef- 235
 235 initely. 236

236 In biochemistry, the *thinning* method was popularized by Gille- 237
 237 spie [7, 8]. For this reason, this method is also called the *Gille- 238*
 238 spie method. Gillespie himself called it the *direct* method or the 239
 239 *stochastic simulation algorithm*. Gillespie introduced *thinning* 240
 240 in the context of simulating chemical reactions of well-stirred sys- 241
 241 tems. He developed a stochastic model for molecule interactions 242
 242 from physics principles without any references to the point process 243
 243 theory developed in this section. His model of chemical interactions 244
 244 is equivalent to a marked Poisson process with constant conditional

245 intensity between jumps. The model consists of distinct populations
 246 of molecular species that interact through several reaction channels.
 247 A chemical reaction consists of a Poisson process that transforms
 248 a set of molecules of some type into a set of molecules of another
 249 type. What Gillespie calls the master equation can be deduced from
 250 the *superposition theorem* — Theorem 3.3 [14].

251 In biochemistry, *thinning* methods are known as *rejection* algo-
 252 rithms. Than *et al.* [28, 29] proposed the *rejection-based algo-*
 253 *algorithm with composition-rejection search*, yet another more so-
 254 phisticated variation of the *thinning* method. In this case, the pro-
 255 cedure groups similar processes together. For each group, an upper-
 256 and lower-bound conditional intensity is used for thinning. A simi-
 257 lar procedure is also described in [27], in which the authors refer
 258 to their algorithm as *kinetic Monte Carlo*.

259 Algorithm 2 presents a *thinning* algorithm, which is a modified
 260 version of Algorithm 7.5.IV [3]. To implement the algorithm, we
 261 define three functions, $\bar{B}^*(t) = \bar{B}(t | H_t)$, $B^*(t) = B(t | H_t)$
 262 and $L^*(t) = L(t | H_t)$, that characterize the local boundedness
 263 condition such that:

$$\lambda^*(t+u) \leq \bar{B}^*(t+u) \text{ and } \lambda^*(t+u) \geq B^*(t+u),$$

$$\forall 0 \leq u \leq L^*(t). \quad (4.9)$$

264 The tighter the bound $\bar{B}^*(\cdot)$ on $\lambda^*(\cdot)$, the lower the number of
 265 discarded samples. Since looser bounds lead to less efficient algo-
 266 rithms, the art, when simulating via *thinning*, is to find the optimal
 267 balance between the local supremum of the conditional intensity
 268 $\bar{B}^*(\cdot)$ and the duration of the local interval $L^*(t)$. On the other
 269 hand, the infimum $B^*(\cdot)$ can be used to avoid the evaluation of
 270 $\lambda^*(\cdot)$ in Line 13 of Algorithm 3 which often can be expensive.

271 In Line 6 of Algorithm 2, since the candidate interval u is itself the
 272 random inter-event interval from a TPP with conditional intensity
 273 $\bar{B}^*(\cdot)$, we are back to simulating a TPP via the inverse method.
 274 Therefore, the wrong choice of $\bar{B}^*(\cdot)$ could in fact deteriorate the
 275 performance of the simulation. In many applications, the bound
 276 $\bar{B}^*(\cdot)$ is constant over $[0, L^*(t)]$ which simplifies the simulation
 277 since then $u \sim \exp(\bar{B}^*(t))$. Alternatively, Bierkens *et al.* [2] uses
 278 a Taylor approximation of $\lambda^*(t)$ to obtain an upper-bound which
 279 is a linear function of t^4 .

280 When the conditional intensity is constant between jumps such that
 281 $\lambda^*(t) = \lambda_{n-1}, \forall t_{n-1} \leq t < t_n$, let $\bar{B}^*(t) = B^*(t) = \lambda_{n-1}$
 282 and $L^*(t) = \infty$. We have that for any $u \sim \exp(1 / \bar{B}^*(t)) =$
 283 $\exp(\lambda_{n-1})$ and $v \sim U[0, 1]$, $u < L^*(t) = \infty$ and $v <$
 284 $\lambda^*(t+u) / \bar{B}^*(t) = 1$. Therefore, we advance the history
 285 for every iteration of Algorithm 2. In this case, the bound $\bar{B}^*(t)$
 286 is as tight as possible, and this method becomes equivalent to the
 287 *inverse* method of Subsection 4.1.

288 We can draw more connections between *thinning* and *inversion*.
 289 Lemaire *et al.* [16] propose a version of the *thinning* algorithm
 290 for PDMPs which does not use a local interval for rejection —
 291 equivalent to $L^*(t) = \infty$. They propose an optimal upper-bound
 292 $\bar{B}^*(t)$ as a piecewise constant function partitioned in such a way
 293 that it envelopes the intensity function as strictly as possible. The
 294 efficiency of their algorithm depends on the assumption that the
 295 stochastic process determined by $\bar{B}^*(t)$ can be efficiently inverted.
 296 They show that under certain conditions the stochastic process de-
 297 termined by $\bar{B}^*(t)$ converges in distribution to the target condi-

tional intensity as the partitions of the optimal boundary converge
 to zero. These results suggest that the efficiency of *thinning* com-
 pared to *inversion* most likely depends on the rejection rate ob-
 tained 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, *thin-*
ning 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 TPP
 over a fixed duration of time $[0, T)$.

```

1: procedure THINNINGMETHOD( $[0, T)$ ,  $\lambda^*$ ,  $f^*$ )
2:   initialize the history  $H_{T^-} \leftarrow \{\}$ 
3:   set  $n \leftarrow 0, t \leftarrow 0$ 
4:   while true do
5:      $t \leftarrow \text{TIMEVIATHINNING}([t, T), H_{T^-}, \lambda^*)$ 
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  $k_n \sim f^*(k | t_n)$ 
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 TIMEVIATHINNING( $[t, T)$ ,  $\lambda^*$ ,  $H_t$ )
2:   while  $t < T$  do
3:     update  $\lambda^*$  [G: with new  $H_t$ ]
4:     find  $\bar{B}^*(t)$ ,  $B^*(t)$  and  $L^*(t)$  which satisfy Eq. 4.9
5:     draw candidate interval  $u$  such that
6:        $P(u > s) = \exp(-\int_0^s \bar{B}^*(t+s) ds)$ 
7:     draw acceptance threshold  $v \sim U[0, 1]$ 
8:     if  $u > L^*(t)$  then
9:        $t \leftarrow t + L^*(t)$ 
10:    next
11:    end if
12:    if [G: ( $v \leq B^*(t+u) / \bar{B}^*(t+u)$ )]
13:      or ( $v \leq \lambda^*(t+u) / \bar{B}^*(t+u)$ ) then
14:         $t \leftarrow t + u$ 
15:        [G: return #break]
16:    end if
17:     $t \leftarrow t + u$ 
18:  end while
19:  return  $t$ 
20: end procedure

```

⁴Their implementation of the Zig-Zag process, a type of PMDP for Markov Chain Monte Carlo, is available as a Julia package at <https://github.com/mschauer/ZigZagBoomerang.jl>.

4.3 Queuing methods for multivariate processes

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 [7]. 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 [14] — in the inverse direction.

Gibson and Bruck [6] 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 [5] and epidemics [13] simulation. In both cases, the phenomena are modelled as point processes.

We prefer to call this class of methods *queued thinning* 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 [3] 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 of which has its own mark distribution.

Algorithm 5 presents our [G: new] method for sampling a superposed point process consisting of M processes by keeping the strike time of each process in a priority queue Q . [G: Thus, it is an example of *queued thinning*.] 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 *queued thinning* algorithms such as [5] is the fact that one goes through multiple rejection cycles at time t before accepting a time candidate $t < t_i$ for process i . This requires looking ahead in the future. In addition to that, if process j , which i depends on, takes place before t_i , then we need to repeat the whole thinning process to obtain a new time candidate for i .

In Algorithm 5, we take a different approach which performs thinning in synchrony with the main loop, avoiding look ahead and wasted rejections. Our main contribution is to modify the main loop of previous thinning algorithms to allow at most one event proposal for each sub-process for each time step. The proposed candidates are always added to the priority queue Q because we need to stop at each proposed time. When the candidate is pre-rejected, we update the bounds and make a new proposal. Alternatively, if the candidate time has not been pre-rejected, we draw the acceptance threshold and compute the intensity rate to make a decision. If the candidate is accepted, we trigger a new round of thinning. Otherwise, we update the bounds and make a new proposal. Overall, we avoid unnecessary updates. Additionally, thinning is now synced with the

main loop, which allows the coupling of 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 *queued thinning* synced algorithm we are aware of. Since Algorithm 5 can be mapped to a *non-queued thinning* algorithm — see [5] —, 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 [3].

Algorithm 4 Generates the next candidate time for *queued thinning*.

```

1: procedure QUEUETIME( $t, \lambda^*, H_t$ )
2:   update  $\lambda^*$ 
3:   find  $\bar{B}^*, B^*$  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:      $u \leftarrow L^*(t)$ 
8:   else
9:     accepted  $\leftarrow$  true
10:  end if
11:   $t \leftarrow t + u$ 
12:  return  $t, \bar{B}^*, B^*$ , accepted
13: end procedure

```

5. Implementation

JumpProcesses.jl is a Julia library for simulating jump — or point — processes which is part of Julia’s SciML organization. In the Julia ecosystem, there are other libraries that can sample certain TPPs including Hawkes.jl⁵, HawkesProcesses.jl⁶, NetworkHawkesProcesses.jl⁷, PointProcessInference.jl [26]⁸, GeoStats.jl [12]⁹, PiecewiseDeterministicMarkovProcesses.jl [30]¹⁰, and PointProcesses.jl [4]¹¹. Apart from PiecewiseDeterministicMarkovProcesses.jl, these other libraries can only sample the Poisson and/or the Hawkes processes. PointProcesses.jl also offers a formalized interface that other packages can implement to leverage its TPP modelling functionality. While JumpProcesses.jl can be used to directly simulate TPPs, in its documentation we also show how it can be wrapped to conform to this interface¹². [G: Outside of Julia, there are many packages to simulate TPPs. A non-exhaustive list include the Python libraries Tick [1], PoPPy [31], hawkesbook [15], and the R library PtProcess [11].]

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

⁵<https://github.com/em1234321/Hawkes.jl>

⁶<https://github.com/dm13450/HawkesProcesses.jl>

⁷<https://github.com/cswaney/NetworkHawkesProcesses.jl>

⁸<https://github.com/mschauer/PointProcessInference.jl>

⁹<https://github.com/JuliaEarth/GeoStats.jl>

¹⁰<https://github.com/rveltz/PiecewiseDeterministicMarkovProcesses.jl>

¹¹<https://github.com/gdalle/PointProcesses.jl>

¹²https://docs.sciml.ai/JumpProcesses/stable/application/s/advanced_point_process

Algorithm 5 The *queued thinning* method for simulating a marked TPP 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:      $(t_i, \bar{B}_i^*, \underline{B}_i^*, a_i) \leftarrow \text{QUEUE TIME}(0, H_{T^-}, \lambda_i^*(\cdot))$ 
6:     push  $(i, t_i, \bar{B}_i^*, \underline{B}_i^*, a_i)$  to  $Q$ 
7:   end for
8:   while  $t < T$  do
9:     first  $(i, t_i, \bar{B}_i^*, \underline{B}_i^*, a_i)$  from  $Q$ 
10:     $t \leftarrow t_i$ 
11:    if  $t \geq T$  then
12:      break
13:    end if
14:    draw  $[G: v \sim U[0, 1]]$ 
15:    if  $a_i$  and  $[G: (v > \underline{B}_i^*/\bar{B}_i^*) \text{ and } (v > \lambda^*(t)/\bar{B}_i^*)]$  then
16:       $a_i \leftarrow \text{false}$ 
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  $k_n \sim f_i^*(k | t_n)$ 
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:         $(t_j, \bar{B}_j^*, \underline{B}_j^*, a_j) \leftarrow \text{QueueTime}(t, \lambda_j^*, H_{T^-})$ 
25:        update  $(j, t_j, \bar{B}_j^*, \underline{B}_j^*, a_j)$  in  $Q$ 
26:      end for
27:    else
28:       $(t_i, \bar{B}_i^*, \underline{B}_i^*, a_i) \leftarrow \text{QueueTime}(t, \lambda_i^*, H_{T^-})$ 
29:      update  $(i, t_i, \bar{B}_i^*, \underline{B}_i^*, a_i)$  in  $Q$ 
30:    end if
31:  end while
32:  return  $H_{T^-}$ 
33: end procedure

```

lem to an OrdinaryDiffEq.jl integrator, which easily inter-operates with JumpProcesses.jl (both libraries are part of the SciML organization, and by design built to easily compose). JumpProcesses.jl does not yet support [G:the any of the] CHV approaches.

5.2 Thinning implementation

Alternatively, *thinning* methods can be simulated via discrete steps. [G: In JumpProcesses.jl, simulation approaches that take discrete steps are handled via discrete callbacks that are checked at the end of each time-step of some time evolution algorithm, e.g. an ODE solver from OrdinaryDiffEq.jl, a stochastic differential equation solver from StochasticDiffEq.jl, or the pure-jump process SSASStepper provided by JumpProcesses.jl. In simple terms, discrete callbacks involve two functions. Condition functions are checked at each step of the main loop of a time-stepping method to see if the callback should be executed, and if it should, an associated affect function is called.] In the context of the library, any method that uses thinning via a discrete callback is called an *aggregator*. There are twelve different aggregators which we discuss below and are summarized in Table 4 in the Annex. [G: At the moment, it is not necessarily the case that one method supersedes the other. There are cases in which a particular method might be faster than others.]

We start with constant rate *thinning* aggregators for marked TPPs. Algorithm 2 assumes that there is a single process. In reality, all the implementations first assume a finite multivariate point process with M interdependent sub-processes. This can be easily conciliated, as we do now, using Definition 6.4.1 [3] which states the equivalence of such process with a point process with a finite space of marks.

As all the constant rate *thinning* aggregators only support ConstantRateJumps and MassActionJumps, i.e. 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, $k_n \sim f_i^*(k | t_n)$. So all sub-processes could potentially be marked, but note users need to handle any additional sampling related to such marks within their provided *affect!* function. 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 [G:and-DirectFW] follow the *direct* method in [7] which re-evaluates all intensities after every iteration scaling at $O(K)$. It draws the next-time from the ground process whose rate is the sum of all sub-processes' rates. It selects the mark by executing a search in an array that stores the cumulative sum of rates.

SortingDirect, RDirect, DirectCR are improvements over the Direct method. They only re-evaluate the intensities of the processes that are affected by the realized process based on a dependency graph. SortingDirect draws from the ground process, but keeps the intensity rate in a loosely sorted array following [19]. RDirect is a rejection-based direct method which assigns the maximum rate of the system as the bound to all processes. The implementation slight differs from Algorithm 2. Since all sub-process have the same rate it draws the next time from a homogeneous Poisson process with the maximum rate, then randomly se-

rate and the mark distribution. In JumpProcesses.jl, these can be mapped to user defined functions `rate(u, p, t)` and `affect!(integrator)`. The former takes the current state of the system, `u`, user provided parameters, `p`, and the current time, `t`, and returns the value of the intensity function at this time. The latter takes the solver `integrator` object, which stores all solution information, and updates it, including the state `integrator.u`, for whatever changes should occur when the jump it encodes fires at the time `integrator.t`. The library provides APIs for defining processes based on the nature of the intensity rate and the intended simulation algorithm. Processes simulated using exact sampling 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 mass action type reaction equations. Finally, `RegularJump` is intended for [G:tau7]-leaping methods.

5.1 Inverse implementation

The *inverse* method as described around Equation 4.2 uses root finding to calculate 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 [24] and passes the prob-

lects a candidate process and confirms the candidate only if its rate is above a random proportion of the maximum rate. `DirectCR` — from [27] — is a composition-rejection method that groups sub-processes with similar rates using a priority table. Each group is assigned the sum of all the rates within it. We apply a routine equivalent to `Direct` to select the time in which the next group fires. Given a group, we then select which process fires. `RSSA` and `RSSACR` place processes in bounded brackets. `RSSA` — from [28] — follows Algorithm 2 very closely in the case where the bounds are constant between jumps. `RSSACR` — from [29] — groups sub-processes with similar rates like `DirectCR`, but then places each group within a bounded bracket. It then samples the next group to fire similar to `RSSA`. Given the group, it selects a candidate to fire and performs a thinning routine to accept or reject.

5.3 Queued thinning implementation

Finally, we have what we call the *queued thinning* aggregators. Starting with aggregators that only support `ConstantRateJumps` we have, `FRM[G; -FRMF]` and `NRM`. `FRM[G; -and FRMF]` follow the *first reaction* method in [7]. 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, these methods are inefficient compared to `NRM` which is a `[G:queued-thinningqueued thinning]` method sourced from [6]. `NRM` gains efficiency by using an indexed priority queue to store and determine next event times, and by using dependency graphs to only update intensities that would need to be recalculated after a given event.

Most of the algorithms implemented in `JumpProcesses.jl` come 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. Our latest aggregator, `Coevolve`, which is an implementation of Algorithm 5, addresses this gap. This is the first aggregator that supports `VariableRateJumps`. Compared with the current *inverse* method-based approach that relies on ODE integration, the new aggregator substantially improves the performance of simulations with time-dependent intensity rates and/or coupled with differential equations from `DifferentialEquations.jl`. `Coevolve` also employs several enhancements compared to Algorithm 5. First, 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. Second, it adapts to processes with constant intensity between jumps which reduces the loop in Algorithm 3 to the equivalent implemented in `NRM` for `ConstantRateJumps` and `MassActionJumps`.

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`. On the one hand, we can think of the stepper as the routine that determines how the numerical solver advances time. On the other hand, the aggregator is the algorithm for sampling the path of a jump process. The aggregator provides stopping times to the stepper.

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 a self-exciting process. Thus, we have added a test for the univariate Hawkes model that checks whether

609 algorithms that accept `VariableRateJump` are able to produce
 610 an empirical distribution of trajectories whose first two moments of
 611 the observed rate are close to the expected ones.

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

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

622 Then the transformed simulated interval is given by:

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

623 Compute the empirical quantiles of the transformed intervals. That
 624 is, the q -th quantile is the interval $\Delta\Lambda_q$ that divides the sorted
 625 intervals in two sets, those below and above $\Delta\Lambda_q$ such that q -percent
 626 of the elements are below it. Plot the empirical quantiles with the
 627 corresponding quantiles of the exponential distribution. If the simulator
 628 produces correct trajectories, this plot known as Q-Q plot should
 629 depict the points aligned around the 45-degree line. We produce Q-Q
 630 plots for the homogeneous Poisson process as well as the compound
 631 Hawkes process — see Subsection 6.2 for a definition — to attest the
 632 correctness of `CoevoLve`. Figure 1 (d) depicts the Q-Q plot for a
 633 ten-node compound Hawkes process with parameters $\lambda = 0.5, \alpha = 0.1, \beta = 2.0$
 634 simulated 250 times for 200 units of time. Figure 1 also depicts the
 635 trajectory, the conditional intensity and the network structure of a
 636 single simulation for three random nodes in panels (a), (b) and (c)
 637 respectively. We obtained similar Q-Q plots for the other algorithms
 638 that benchmarked the Multivariate Hawkes process below.
 639

6.2 Benchmarks

641 We conduct a set of benchmarks to assess the performance of the
 642 `JumpProcesses.jl` aggregators described in Section 5. All
 643 benchmarks are available in `SciMLBenchmarks.jl`¹³. All were
 644 run in `BuildKite`¹⁴ via the continuous integration facilities provided
 645 by the package maintainers. We have added two benchmark suites
 646 to assess the performance of the new aggregators under settings that
 647 could not be simulated with previous aggregators.

648 First, we assess the speed of the aggregators against jump processes
 649 whose rates are constant between jumps. There are four such
 650 benchmarks: a 1-dimensional continuous time random walk approx-
 651 imation of a diffusion model (Diffusion), the multi-state model
 652 from Appendix A.6 [18] (Multi-state), a simple negative feedback
 653 gene expression model (Gene I) and the negative feedback gene
 654 expression from [10] (Gene II). We simulate a single trajectory for

¹³<https://github.com/SciML/SciMLBenchmarks.jl/tree/7d356203ea107d7343af1ce41d94b64847095d4a/benchmarks/Jumps>
 and <https://github.com/SciML/SciMLBenchmarks.jl/tree/7d356203ea107d7343af1ce41d94b64847095d4a/benchmarks/HybridJumps>

¹⁴<https://buildkite.com/julialang/scimlbenchmarks-dot-jl/builds/1849#018c3980-5247-42ab-a7fe-3145209b26c5>

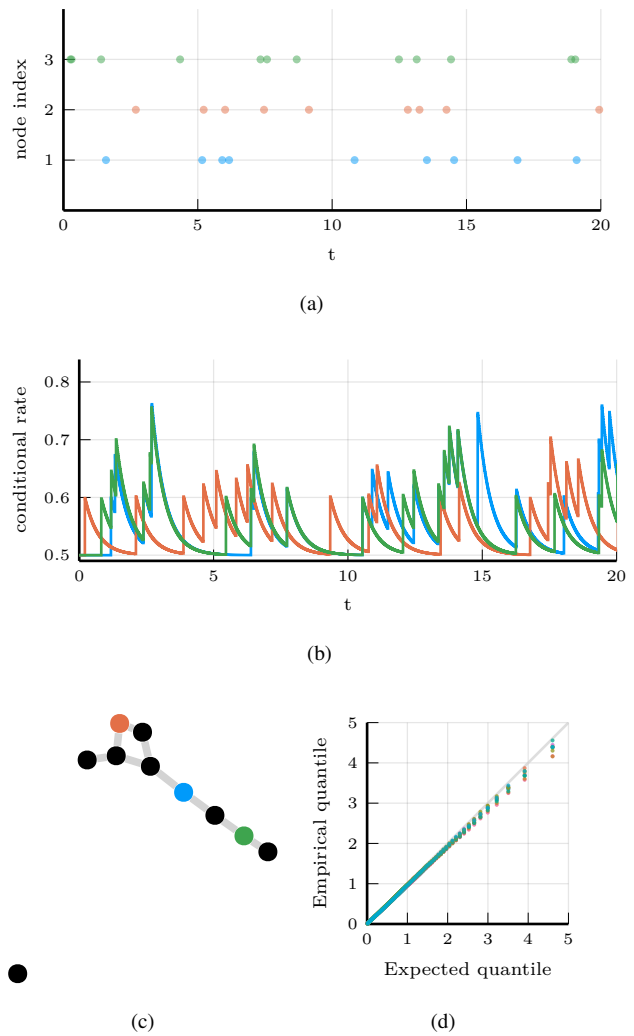


Fig. 1: Simulations of 10-nodes compound Hawkes process with parameters $\lambda = 0.5, \alpha = 0.1, \beta = 2.0$ for 200 units of time. (a) and (b) sampled trajectory and intensity rate for a single simulation for the three selected nodes in (c) for the first 20 units of time. (c) underlying 10-nodes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

655 each aggregator to visually check that they produce similar trajec-
 656 tories for a given model. The Diffusion, Multi-state, Gene I and
 657 Gene II benchmarks are then simulated 50, 100, 2000 and 200
 658 times, respectively. Check the source code for further implementa-
 659 tion details.

660 Benchmark results are listed in Table 1. The table shows that no
 661 single aggregator dominates suggesting they should be selected ac-
 662 cording to the task at hand. However, `FRM`, `NRM`, `CoevoLve` never
 663 dominate any benchmark. In common, they all belong to the fam-
 664 ily of queuing methods. **[G: The fact that these are not the fastest
 665 methods for the constant rate benchmarks shows that improvements
 666 to thinning algorithms bring substantial performance gains which
 667 could potentially be explored in queued thinning algorithms. A par-
 668 ticular issue with queuing methods is the cost of updating the un-
 669 derlying indexed priority queue data structure which stores the next
 670 event times. A table-based data-structure would be expected to be**

	Diffusion	Multi-state	Gene I	Gene II
Direct	7.14 s	0.16 s	<u>0.24 ms</u>	<u>0.59 s</u>
FRM	15.76 s	0.25 s	0.29 ms	0.77 s
SortingDirect	1.06 s	<u>0.11 s</u>	0.24 ms	0.53 s
NRM	0.76 s	0.25 s	0.39 ms	0.90 s
DirectCR	<u>0.50 s</u>	0.22 s	0.49 ms	1.09 s
RSSA	1.42 s	0.10 s	0.43 ms	0.66 s
RSSACR	0.46 s	0.15 s	0.91 ms	1.06 s
Coevolve	0.88 s	0.34 s	0.53 ms	1.29 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 [18] (Multi-state), a simple negative feedback gene expression model (Gene I) and the negative feedback gene expression from [10] (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.

671 **more competitive such as proposed in [25].** We also note that the
 672 performance of `Coevolve` lags that of `NRM` despite the fact that
 673 `Coevolve` should take the same number of steps as `NRM` when
 674 no `VariableRateJump` is used. The reason behind this discrepan-
 675 cy is likely due to implementation differences, but left for future
 676 investigation.

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

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

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

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

686 The conditional intensity of this process has a recursive formulation
 687 which can significantly speed the simulation. The recursive formu-
 688 lation for the univariate case is derived in [15] which also provides
 689 additional discussion and results on the Hawkes process. We derive
 690 the compound case here. Let $t_{N_i} = \max\{t_{n_j} < t \mid j \in E_i\}$
 691 and $\phi_i^*(t)$ below.

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

692 Then the conditional intensity can be re-written in terms of
 693 $\phi_i^*(t_{N_i})$.

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

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 with the graph size.

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 `PiecewiseDeterministic-MarkovProcesses.jl` 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`¹⁵. 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 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, `Jump-Processes.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 read-out of enzyme dynamics proposed in [23]. 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

¹⁵<https://github.com/X-DataInitiative/tick>

754 problem containing 98 jumps affecting 49 discrete variables with
 755 a stiff, ordinary differential equation problem containing 34 con-
 756 tinuous variables. Continuous variables affect jump rates while the
 757 discrete variables affect the continuous problem. There are 3 stages
 758 to the simulation: pre-synaptic evolution, glutamate release, and
 759 post-synaptic evolution. Among the algorithms considered, only
 760 the *inverse* method implemented in `JumpProcesses.jl`, `Coevolve`
 761 and `CHV` are theoretically able to simulate the synapse
 762 model. However, in practice, only the last two complete at least one
 763 benchmark run. The original synapse problem was described as a
 764 PDMP, so we do not make the distinction between *CHV simple*
 765 and *full* in this benchmark.

766 Benchmark results are displayed in Table 3. We observe that *CHV*
 767 is the fastest algorithm completing the synapse evolution in about
 768 half of the time it takes `Coevolve` with less than half of the allo-
 769 cations. Further investigation reveals that the thinning procedure in
 770 `Coevolve` reaches an average of 70 percent over all jumps which
 771 then leads to 2 to 3 times more function evaluations and Jaco-
 772 bians created compared to *CHV*. Our implementation adds stop-
 773 ping times via a call to `register_next_jump_time!` even for
 774 rejected jumps — we do not know a jump will be rejected until
 775 evaluated. This then leads the ODE solver to step to those times and
 776 make additional function evaluations and Jacobians. Lemaire *et*
 777 *al.* [16] performs a similar benchmark in which they compare the
 778 Hodgkin-Huxley model against different thinning conditions and
 779 against an ODE approximation. They find that a thinned algorithm
 780 with optimal boundary conditions can run significantly faster than
 781 the ODE approximation. Thus, there could be plenty of room to
 782 improve the performance of `Coevolve` in our setting.

783 A disadvantage of *CHV* compared with `Coevolve` is that it sup-
 784 ports limited saving options by design. To save at pre-specified
 785 times would require using the fact that solutions are piecewise con-
 786 stant to determine solutions at times in-between jumps — and for
 787 coupled ODE-jump problems would require root-finding to deter-
 788 mine when $s(u) = s_n$ for each desired saving time s_n in Equa-
 789 tion 4.8. The alternative proposed in [30] is to introduce an artificial
 790 jump to the model such as the homogeneous Poisson process with
 791 unit rate to sample the system at regular intervals. Alternatively,
 792 `Coevolve` allows saving at any arbitrary point. A common work-
 793 flow in simulating jump processes, particularly when interested in
 794 calculating statistics over time, is to pre-specify a precise set of
 795 times at which to save a simulation. In theory, this reduces mem-
 796 ory pressure, particularly for systems with large numbers of jumps,
 797 and can give increased computational performance relative to sav-
 798 ing the state at the occurrence of every jump. However, in the case
 799 of the synapse model, the number of candidate time rejections far
 800 surpasses the number of jumps. Therefore, reducing the number of
 801 saving points — *e.g.* only saving at start and end of the simulation
 802 — does not significantly reduce allocations or running time. Given
 803 these considerations, we decided to save after every jump and at
 804 regular pre-specified intervals that occur at the same frequency as
 805 the artificial saving jump used by *CHV*.

806 Another parameter that can affect the precision and speed of the
 807 synapse benchmark is the ODE solver. The author of `Piece-`
 808 `wiseDeterministicMarkovProcesses.jl` discuss some
 809 of these issues in Discourse¹⁶. Some ODE solvers can be faster and
 810 more precise. Due to time constraints, we have not investigated this
 811 matter. The synapse benchmark uses the `AutoTsit5` (Rosen-

brock23()) solver in both `Coevolve` and *CHV*. Further inves-
 tigation of this matter is left to future research.

7. Conclusion

This paper demonstrates that `JumpProcesses.jl` is a fast, general-purpose library for simulating TPPs. 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 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 TPPs. 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, a gap which `Coevolve` is meant to fill.

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.

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 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. The approach of `ZigZagBoomerang.jl` [2] that combines Taylor approximation of the conditional intensity with automatic differentiation could be explored. 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 [15]. 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.

¹⁶<https://discourse.julialang.org/t/help-me-beat-lsoda/88236>

	V	Brute Force			Recursive				
		Inverse	Coevolve	CHV simple	Inverse	Coevolve	CHV simple	CHV full	Tick
Time	1	74.1 μ s	4.8 μs	203.1 μ s	76.6 μ s	<u>5.0 μs</u>	201.5 μ s	197.9 μ s	30.7 μ s
	10	10.0 ms	205.2 μ s	5.1 ms	3.8 ms	73.5 μs	471.6 μ s	607.3 μ s	<u>175.0 μs</u>
	20	89.6 ms	1.5 ms	48.9 ms	16.2 ms	265.8 μs	964.1 μ s	<u>902.3 μs</u>	1.2 ms
	30	274.2 ms	3.3 ms	129.5 ms	45.7 ms	502.9 μs	1.6 ms	<u>1.3 ms</u>	3.7 ms
	40	1.9 s <i>n=37</i>	8.4 ms	320.8 ms <i>n=31</i>	1.2 s <i>n=9</i>	913.5 μs	2.5 ms	<u>1.6 ms</u>	9.3 ms
	50	3.6 s <i>n=7</i>	16.8 ms	681.0 ms <i>n=15</i>	2.4 s <i>n=5</i>	1.5 ms	3.6 ms	<u>2.0 ms</u>	21.7 ms
	60	6.7 s <i>n=3</i>	37.9 ms	1.3 s <i>n=8</i>	4.1 s <i>n=3</i>	2.2 ms	5.1 ms	<u>2.6 ms</u>	46.9 ms
	70	10.6 s <i>n=2</i>	58.5 ms	2.2 s <i>n=5</i>	6.8 s <i>n=2</i>	<u>3.0 ms</u>	6.9 ms	3.0 ms	89.5 ms
	80	15.5 s <i>n=1</i>	138.8 ms	3.3 s <i>n=4</i>	10.6 s <i>n=1</i>	<u>4.0 ms</u>	9.1 ms	3.2 ms	147.1 ms
	90	27.86 s <i>n=1</i>	139.7 ms	5.6 s <i>n=2</i>	16.0 s <i>n=1</i>	<u>5.3 ms</u>	11.8 ms	3.9 ms	233.4 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	<u>4.9 s</u>	<u>94.0 Mb</u>
CHV	2.7 s	43.5 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.

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 and 2325185.

9. References

- [1] Emmanuel Bacry, Martin Bompain, Stéphane Gaïffas, and Soren Poulsen. Tick: A Python library for statistical learning, with a particular emphasis on time-dependent modelling, March 2018. doi:10.48550/arXiv.1707.03003. 1707.03003.
- [2] Joris Bierkens, Paul Fearnhead, and Gareth Roberts. The Zig-Zag process and super-efficient sampling for Bayesian analysis of big data. *The Annals of Statistics*, 47(3), June 2019. doi:10.1214/18-AOS1715.
- [3] 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, New York, 2 edition, 2003. doi:10.1007/b97277.
- [4] Guillaume Dalle. PointProcesses.jl. Zenodo, January 2024. doi:10.5281/zenodo.10477603.
- [5] 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. *The Journal of Machine Learning Research*, 18(1), January 2017.
- [6] Michael A. Gibson and Jehoshua Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *The Journal of Physical Chemistry A*, 104(9), March 2000. doi:10.1021/jp993732q.
- [7] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4), December 1976. doi:10.1016/0021-9991(76)90041-3.
- [8] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25), December 1977. doi:10.1021/j100540a008.
- [9] Daniel T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 115(4), July 2001. doi:10.1063/1.1378322.

- 911 [10] Abhishekh Gupta and Pedro Mendes. An Overview of
912 Network-Based and -Free Approaches for Stochastic Simu-
913 lation of Biochemical Systems. *Computation*, 6(1), March
914 2018. doi:10.3390/computation6010009.
- 915 [11] David Harte. PtProcess : An R Package for Modelling Marked
916 Point Processes Indexed by Time. *Journal of Statistical
917 Software*, 35(8), 2010. doi:10.18637/jss.v035.i08.
- 918 [12] Júlio Hoffmann, Fredrik Ekre, Martijn Visser, Anshul
919 Singhvi, Durand D'souza, M. A. Siddique, Morten Piibeleht,
920 Tony Kelman, and Zlatan Vasović. JuliaEarth/GeoStats.jl:
921 V0.11.7. Zenodo, June 2020. doi:10.5281/zenodo.3875233.
- 922 [13] Petter Holme. Fast and principled simulations of the SIR
923 model on temporal networks. *PLOS ONE*, 16(2), February
924 2021. doi:10.1371/journal.pone.0246961.
- 925 [14] Günter Last and Mathew Penrose. *Lectures on the Pois-
926 son Process*. Cambridge University Press, 1st edition edi-
927 tion, October 2017.
- 928 [15] Patrick J. Laub, Young Lee, and Thomas Taimre. *The Ele-
929 ments of Hawkes Processes*. Springer International Pub-
930 lishing, 2021. doi:10.1007/978-3-030-84639-8.
- 931 [16] Vincent Lemaire, Michèle Thiullen, and Nicolas Thomas.
932 Exact Simulation of the Jump Times of a Class of Piece-
933 wise Deterministic Markov Processes. *Journal of Scien-
934 tific Computing*, 75(3), June 2018. doi:10.1007/s10915-017-
935 0607-4.
- 936 [17] P. A. W. Lewis and G. S. Shedler. Simulation of
937 nonhomogeneous poisson processes by thinning.
938 *Naval Research Logistics Quarterly*, 26(3), 1979.
939 doi:10.1002/nav.3800260304.
- 940 [18] Luca Marchetti, Corrado Priami, and Vo Hong Thanh. *Sim-
941 ulation Algorithms for Computational Systems Biol-
942 ogy*. Texts in Theoretical Computer Science. An EATCS
943 Series. Springer International Publishing, Cham, 2017.
944 doi:10.1007/978-3-319-63113-4.
- 945 [19] James M. McCollum, Gregory D. Peterson, Chris D. Cox,
946 Michael L. Simpson, and Nagiza F. Samatova. The sort-
947 ing direct method for stochastic simulation of biochemical
948 systems with varying reaction execution behavior. *Compu-
949 tational Biology and Chemistry*, 30(1), February 2006.
950 doi:10.1016/j.compbiolchem.2005.10.007.
- 951 [20] James Meiss. *Differential Dynamical Systems, Revised
952 Edition*. Mathematical Modeling and Computation. Soci-
953 ety for Industrial and Applied Mathematics, January 2017.
954 doi:10.1137/1.9781611974645.
- 955 [21] Y. Ogata. On Lewis' simulation method for point processes.
956 *IEEE Transactions on Information Theory*, 27(1), Janu-
957 ary 1981. doi:10.1109/TIT.1981.1056305.
- 958 [22] Christopher Rackauckas and Qing Nie. DifferentialEqua-
959 tions.jl – A Performant and Feature-Rich Ecosystem for
960 Solving Differential Equations in Julia. 5(1), May 2017.
961 doi:10.5334/jors.151.
- 962 [23] Yuri E. Rodrigues, Cezar M. Tigaret, Hélène Marie,
963 Cian O'Donnell, and Romain Veltz. A stochastic
964 model of hippocampal synaptic plasticity with geo-
965 metrical readout of enzyme dynamics, March 2021.
966 doi:10.1101/2021.03.30.437703.
- 967 [24] Howard Salis and Yiannis Kaznessis. Accurate hybrid
968 stochastic simulation of a system of coupled chemical or
969 biochemical reactions. *The Journal of Chemical Physics*,
970 122(5), February 2005. doi:10.1063/1.1835951.
- 971 [25] Kevin R. Sanft and Hans G. Othmer. Constant-complexity
972 Stochastic Simulation Algorithm with Optimal Binning.
973 *The Journal of Chemical Physics*, 143(7), August 2015.
974 doi:10.1063/1.4928635. 1503.05832.
- 975 [26] Moritz Schauer, Frank van der Meulen, and Shota Gugushvili.
976 Mschauer/PointProcessInference.jl: V0.2.2. Zenodo, March
977 2020. doi:10.5281/zenodo.3716127.
- 978 [27] Alexander Slepoy, Aidan P. Thompson, and Steven J.
979 Plimpton. A constant-time kinetic Monte Carlo algorithm
980 for simulation of large biochemical reaction networks.
981 *The Journal of Chemical Physics*, 128(20), May 2008.
982 doi:10.1063/1.2919546.
- 983 [28] Vo Hong Thanh, Corrado Priami, and Roberto Zunino. Ef-
984 ficient rejection-based simulation of biochemical reactions
985 with stochastic noise and delays. *The Journal of Chemical
986 Physics*, 141(13), October 2014. doi:10.1063/1.4896985.
- 987 [29] Vo Hong Thanh, Roberto Zunino, and Corrado Priami. Ef-
988 ficient Constant-Time Complexity Algorithm for Stochastic
989 Simulation of Large Reaction Networks. *IEEE/ACM Trans-
990 actions on Computational Biology and Bioinformatics*,
991 14(3), May 2017. doi:10.1109/TCBB.2016.2530066.
- 992 [30] Romain Veltz. A new twist for the simulation of hy-
993 brid systems using the true jump method, April 2015.
994 doi:10.48550/arXiv.1504.06873. 1504.06873.
- 995 [31] Hongteng Xu. PoPPy: A Point Process Toolbox Based
996 on PyTorch, October 2019. doi:10.48550/arXiv.1810.10122.
997 1810.10122.

998 **Annex**

Aggregator	Name	Description	Sample from	Update	Jump types			Source
					MA	Con.	Var.	
Direct	Direct	Rates kept in a non-sorted array. Sample on ground process.	ground	all	x	x		[7]
DirectFW	Direct with FunctionWrapper	Same as Direct, but wraps rate functions with FunctionWrapper for type stability and better performance in system with many jumps.	ground	all	x	x		[7]
SortingDirect	Sorting direct	Rates kept in a loosely sorted array. Sample on ground process.	ground	graph	x	x		[19]
RDirect	Rejection-based direct	Sample next time using the maximum rate of the system, then randomly selects a candidate and confirms the jump only if its rate is above a random proportion of the maximum rate.	ground	graph	x	x		ours*
DirectCR	Direct with composition-rejection search	Rates in group with similar rates using a priority table. Group rates are the sum of rates in group.	ground	graph	x	x		[27]
RSSA	Rejection-based stochastic simulation algorithm	Processes are assigned lower- and upper-bounds. Sample on upper-bounds.	ground	graph	x	x		[28]
RSSACR	Rejection-based stochastic simulation algorithm with composition-rejection search	Rates in group with similar rates using a priority table. Groups and processes are assigned lower- and upper-bounds. Sample on group upper-bounds.	ground	graph	x	x		[29]
FRM	First reaction method	Selects the minimum time from all samples.	sub	all	x	x		[7]
FRMFW	First reaction method with FunctionWrapper	Same as FRM, but wraps rate functions with FunctionWrapper for type stability and better performance in systems with many jumps.	sub	all	x	x		[7]
NRM	Next reaction method	Keeps a priority queue of times. Next event is the earliest in queue.	sub	graph	x	x		[6]
Coevolve	Coevolve	Synced with model time. Keeps a priority queue of candidate times. Next stop time is the earliest in the queue.	sub	graph	x	x	x	ours

Table 4. : JumpProcesses.jl aggregators. *Sample from* indicates whether the algorithm samples the ground process (or some composition of it), or each sub-process separately. *Update* indicates whether the algorithm updates all rates, or only those affected by the realization of a process via a dependency graph. *Jump types* indicates whether aggregators support `MassActionJump` (MA), `ConstantRateJump` (Con.), or `VariableRateJump` (Var.). In *source*, *ours** indicates that the algorithm was developed by the maintainers of the library prior to this paper.