

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

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

<sup>1</sup>Institute of Data Science, National University of Singapore, Singapore

<sup>2</sup>School of Computing, National University of Singapore, Singapore

<sup>3</sup>Department of Mathematics and Statistics, Boston University

<sup>4</sup>Computer Science and AI Laboratory (CSAIL), Massachusetts Institute of Technology

<sup>5</sup>Department 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 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 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 via thinning. `JumpProcesses.jl` 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<sup>1</sup>. This exactness provides unbiased 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.*  $\tau$ -leap methods [5], 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 compound point processes in large systems with time-dependent rates. To enable the efficient simulation of such processes, we contribute a new simulation algorithm together with its implementation as the `Coevolve` aggregator in `JumpProcesses.jl`, a core sub-library of the popular `DifferentialEquations.jl` library [18]. 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 described in [3] from where the new `JumpProcesses.jl` aggregator borrows its name. The addition of `Coevolve` dramatically boosts the computational performance of the library in simulating processes with intensities that have an explicit dependence on time and/or other continuous variables, significantly expanding the type of models that can be efficiently simulated. Widely-used point processes with such intensities include compound inhomogeneous Poisson 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` 9.7<sup>2</sup>.

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

<sup>1</sup>Some 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.

<sup>2</sup>All examples and benchmarks in this paper use version 9.9 of the library

73 building blocks for all other simulation methods. Then, we identify  
 74 and discuss three types of exact simulation methods. In the sec-  
 75 ond part of this paper, we describe the algorithms implemented in  
 76 `JumpProcesses.jl` and how they relate to the literature. We  
 77 highlight our contribution `Coevolve`, investigate the correctness  
 78 of our implementation and provide performance benchmarks to  
 79 demonstrate its value. The paper concludes by discussing potential  
 80 improvements.

## 81 2. The temporal point process

82 The TPP is a stochastic collection of marked points over a one-  
 83 dimensional support. They are exhaustively described in [2]. The  
 84 likelihood of any TPP is fully characterized by its conditional in-  
 85 tensity,

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

86 and conditional mark distribution,  $f^*(k|t)$  — see Chapter 7 [2].  
 87 Here  $H_{t-} = \{(t_n, k_n) \mid 0 \leq t_n < t\}$  denotes the internal  
 88 history of the process up to but not including  $t$ , the superscript  $*$   
 89 denotes the conditioning of any function on  $H_{t-}$ , and  $p^*(t)$  is the  
 90 density function corresponding to the probability of an event taking  
 91 place at time  $t$  given  $H_{t-}$ . We can interpret the conditional in-  
 92 tensity as the likelihood of observing a point in the next infinitesimal  
 93 unit of time, given that no point has occurred since the last observed  
 94 point in  $H_{t-}$ . Lastly, the mark distribution denotes the density func-  
 95 tion corresponding to the probability of observing mark  $k$  given the  
 96 occurrence of an event at time  $t$  and internal history  $H_{t-}$ .

## 97 3. The homogeneous process

98 A homogeneous process can be simulated using properties of the  
 99 Poisson process, which allow us to describe two equivalent sam-  
 100 pling procedures. The first procedure consists of drawing success-  
 101 ive inter-arrival times. The distance between any two points in  
 102 a homogeneous process is distributed according to the exponen-  
 103 tial distribution — see Theorem 7.2 [10]. Given the homogeneous  
 104 process with intensity  $\lambda$ , then the distance  $\Delta t$  between two points  
 105 is distributed according to  $\Delta t \sim \exp(\lambda)$ . Draws from the ex-  
 106 ponential distribution can be performed by drawing from a uni-  
 107 form distribution in the interval  $[0, 1]$ . If  $V \sim U[0, 1]$ , then  
 108  $T = -\ln(V)/\lambda \sim \exp(1)$ . (Note, however, in Julia the opti-  
 109 mized Ziggurat-based method used in the `randexp` stdlib func-  
 110 tion is generally faster than this *inverse* method for sampling a  
 111 unit exponential random variable.) When a point process is homo-  
 112 geneous, the *inverse* method of Subsection 4.1 reduces to this ap-  
 113 proach. Thus, we defer the presentation of this Algorithm to the  
 114 next section.

115 The second procedure uses the fact that Poisson processes can be  
 116 represented as a mixed binomial process with a Poisson mixing dis-  
 117 tribution — see Proposition 3.5 [10]. In particular, the total number  
 118 of points of a Poisson homogeneous process in  $[0, T)$  is distributed  
 119 according to  $\mathcal{N}(T) \sim \text{Poisson}(\lambda T)$  and the location of each  
 120 point within the region is independently distributed according to  
 121 the uniform distribution  $t_n \sim U[0, T]$ .

## 122 4. Exact simulation methods

### 123 4.1 Inverse methods

124 The *inverse* method leverages Theorem 7.4.I [2] which states that  
 125 every simple point process<sup>3</sup> can be transformed to a homogeneous  
 126 Poisson process with unit rate via the compensator. Let  $t_n$  be the  
 127 time in which the  $n$ -th chronologically sorted event took place and  
 128  $t_0 \equiv 0$ , we define the compensator as:

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

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

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

132 The idea is to draw realizations from the unit rate Exponential pro-  
 133 cess and solve Equation 4.2 for  $t_n$  to determine the next event/firing  
 134 time. We illustrate this in Algorithm 1 where we adapt Algorithm  
 135 7.4 [2].

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

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

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

144 The main drawback of the *inverse* method is that the root finding  
 145 problem defined in Equation 4.2 often requires a numerical solu-  
 146 tion. To get around a similar obstacle in the context of PDMPs,  
 147 Veltz [24] proposes a change of variables in time that recasts the  
 148 root finding problem into an initial value problem. He denotes his  
 149 method *CHV*.

150 PDMPs are composed of two parts: the jump process and the  
 151 piecewise ODE that changes stochastically at jump times — see  
 152 Lemaire *et al.* [12] for a formal definition. Therefore, it is easy to  
 153 employ *CHV* in our case by setting the ODE part to zero through-  
 154 out time. Adapting from Veltz [24], we can determine the model  
 155 jump time  $t_n$  after sampling  $\Delta \tilde{t}_n \sim \exp(1)$  by solving the fol-  
 156 lowing initial value problem until  $\Delta \tilde{t}_n$ .

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

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

160 Alternatively, when the intensity function is differentiable between  
 161 jumps we can go even further by recasting the jump problem as a

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

162 PDMP. Let  $\lambda_n^* \equiv \lambda^*(t_n)$ , then the flow  $\varphi_{t-t_n}(\lambda_n^*)$  maps the initial  
 163 value of the conditional intensity at time  $t_n$  to its value at time  
 164  $t$ . In other words, the flow describes the deterministic evolution of  
 165 the conditional intensity function over time. Next, denote  $\mathbf{1}(\cdot)$  as  
 166 the indicator function, then the conditional intensity function can  
 167 be re-written as a jump process:

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

168 According to Meiss [16], if  $\varphi_t(\cdot)$  is a flow, then it is a solution to  
 169 the initial value problem:

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

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

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

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

174 Equations 4.5 and 4.7 define a PDMP satisfying the conditions of  
 175 Theorem 3.1 [24]. In this case, we find  $t_n$  by solving the following  
 176 initial value problem from 0 to  $\Delta \tilde{t}_n \sim \exp(1)$ .

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

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

181 In Algorithm 1, we can implement the CHV method by solving  
 182 either Equation 4.4 or Equation 4.8 instead of Equation 4.2. We  
 183 denote the first specification as *CHV simple* and the second as  
 184 *CHV full*. Note that *CHV full* requires that the conditional inten-  
 185 sity be piecewise differentiable. The algorithmic complexity is then  
 186 determined by the ODE solver and no root-finding is required. In  
 187 Section 6.2, we will show that there are substantial differences in  
 188 performance between them with the full specification being faster.  
 189 Another concern with Algorithm 1 is updating and drawing from  
 190 the conditional mark distribution in Line 8, and updating the condi-  
 191 tional intensity in Line 9. Assume a process with  $K$  number of  
 192 marks. A naive implementation of Line 9 scales with the number  
 193 of marks as  $O(K)$  since  $\lambda^*$  is usually constructed as the sum of  $K$   
 194 independent processes, each of which requires updating the condi-  
 195 tional intensity rate. Likewise, drawing from the mark distribution  
 196 in Line 8 usually involves drawing from a categorical distribution  
 197 whose naive implementations also scales with the number of marks  
 198 as  $O(K)$ .

199 Finally, Algorithm 1 is not guaranteed to terminate in finite time  
 200 since one might need to sample many points before  $t_n > T$ . The

sampling rate can be especially high when simulating the process  
 in a large population with self-exciting encounters. In biochemistry,  
 Salis and Kaznessis [20] partition a large system of chemical reac-  
 tions into two: fast and slow reactions. While they approximate the  
 fast reactions with a Gaussian process, the slow reactions are solved  
 using a variation of the inverse method. They obtain an equivalent  
 expression for the rate of slow reactions as in Equation 4.2, which  
 is integrated with the Euler method.

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

---

```

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
    
```

---

## 4.2 Thinning methods

*Thinning* methods are popular approaches 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 [2], this pro-  
 cedure 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 [2].

*Thinning* algorithms have been proposed in different forms [2].  
 Shedler-Lewis [13] first suggested a thinning routine that simulated  
 processes with bounded intensity over a fixed interval. Ogata's re-  
 finement [17] suggests a procedure for evolving the simulation via  
 local boundary conditions and fixed partitions of the real line. As  
 long as the intensity conditioned on the simulated history remains  
 locally bounded, it is possible to simulate subsequent points indef-  
 initely.

In biochemistry, the *thinning* method was popularized by Gille-  
 spie [7, 6]. For this reason, this method is also called the *Gille-  
 spie* method. Gillespie himself called it the *direct* method or the  
*stochastic simulation algorithm*. Gillespie introduced *thinning*  
 in the context of simulating chemical reactions of well-stirred sys-  
 tems. He developed a stochastic model for molecule interactions  
 from physics principles without any references to the point process  
 theory developed in this section. His model of chemical interactions  
 is equivalent to a marked Poisson process with constant conditional  
 intensity between jumps. The model consists of distinct populations  
 of molecular species that interact through several reaction channels.  
 A chemical reaction consists of a Poisson process that transforms  
 a set of molecules of some type into a set of molecules of another  
 type. What Gillespie calls the master equation can be deduced from  
 the *superposition theorem* — Theorem 3.3 [10].

In biochemistry, *thinning* methods are known as *rejection* algo-  
 rithms. Than *et al.* [22, 23] proposed the *rejection-based algo-  
 rithm with composition-rejection search*, yet another more so-  
 phisticated variation of the *thinning* method. In this case, the pro-

245 cedure groups similar processes together. For each group, an upper- 297  
 246 and lower-bound conditional intensity is used for thinning. A similar 298  
 247 procedure is also described in [21], in which the authors refer 299  
 248 to their algorithm as *kinetic Monte Carlo*. 300

249 Algorithm 2 presents a *thinning* algorithm, which is a modified 301  
 250 version of Algorithm 7.5.IV [2]. To implement the algorithm, we 302  
 251 define three functions,  $\bar{B}^*(t) = \bar{B}(t | H_t)$ ,  $\underline{B}^*(t) = \underline{B}(t | H_t)$   
 252 and  $L^*(t) = L(t | H_t)$ , that characterize the local boundedness  
 253 condition such that:

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

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

254 The tighter the bound  $\bar{B}^*(\cdot)$  on  $\lambda^*(\cdot)$ , the lower the number of  
 255 discarded samples. Since looser bounds lead to less efficient algo-  
 256 rithms, the art, when simulating via *thinning*, is to find the optimal  
 257 balance between the local supremum of the conditional intensity  
 258  $\bar{B}^*(\cdot)$  and the duration of the local interval  $L^*(t)$ . On the other  
 259 hand, the infimum  $\underline{B}^*(\cdot)$  can be used to avoid the evaluation of  
 260  $\lambda^*(\cdot)$  in Line 12 of Algorithm 3 which often can be expensive.

261 In Line 6 of Algorithm 2, since the candidate interval  $u$  is itself the  
 262 random inter-event interval from a TPP with conditional intensity  
 263  $\bar{B}^*(\cdot)$ , we are back to simulating a TPP via the inverse method.  
 264 Therefore, the wrong choice of  $\bar{B}^*(\cdot)$  could in fact deteriorate the  
 265 performance of the simulation. In many applications, the bound  
 266  $\bar{B}^*(\cdot)$  is constant over  $[0, L^*(t)]$  which simplifies the simulation  
 267 since then  $u \sim \exp(\bar{B}^*(t))$ . Alternatively, Bierkens *et al.* [1] uses  
 268 a Taylor approximation of  $\lambda^*(t)$  to obtain an upper-bound which  
 269 is a linear function of  $t^4$ .

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

278 We can draw more connections between *thinning* and *inversion*.  
 279 Lemaire *et al.* [12] propose a version of the *thinning* algorithm  
 280 for PDMPs which does not use a local interval for rejection —  
 281 equivalent to  $L^*(t) = \infty$ . They propose an optimal upper-bound  
 282  $\bar{B}^*(t)$  as a piecewise constant function partitioned in such a way  
 283 that it envelopes the intensity function as strictly as possible. The  
 284 efficiency of their algorithm depends on the assumption that the  
 285 stochastic process determined by  $\bar{B}^*(t)$  can be efficiently inverted.  
 286 They show that under certain conditions the stochastic process deter-  
 287 mined by  $\bar{B}^*(t)$  converges in distribution to the target condi-  
 288 tional intensity as the partitions of the optimal boundary converge  
 289 to zero. These results suggest that the efficiency of *thinning* com-  
 290 pared to *inversion* most likely depends on the rejection rate ob-  
 291 tained by the former and the number of steps required by the ODE  
 292 solver for the latter.

293 While *thinning* algorithms avoid the issue of directly computing  
 294 the inverse of the integrated conditional intensity, they increase the  
 295 number of time steps needed in the sampling algorithm as we are  
 296 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  $\lambda^*$ 
4:     find  $\bar{B}^*(t), \underline{B}^*(t)$  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  $(v \leq \underline{B}^*(t+u))$  or  $(v \leq \lambda^*(t+u) / \bar{B}^*(t+u))$  then
13:     $t \leftarrow t + u$ 
14:  return  $t$ 
15: end if
16:   $t \leftarrow t + u$ 
17: end while
18: return  $t$ 
19: end procedure

```

---

### 4.3 Queuing methods for multivariate processes

302 As an alternative to his *direct* method — in this text referred as the  
 303 constant rate *thinning* method —, Gillespie introduced the *first*  
 304 *reaction* method in his seminal work on simulation algorithms [7].  
 305 The *first reaction* method separately simulates the next reaction  
 306 time for each reaction channel — *i.e.* for each mark. It then selects  
 307 the smallest time as the time of the next event, followed by updat-  
 308 ing the conditional intensity of all channels accordingly. This is a  
 309 variation of the constant rate *thinning* method to simulate a set of  
 310 inter-dependent point processes, making use of the *superposition*  
 311 *theorem* — Theorem 3.3 [10] — in the inverse direction.  
 312 Gibson and Bruck [4] improved the *first reaction* method with the  
 313 *next reaction* method. They innovate on three fronts. First, they  
 314

<sup>4</sup>Their 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>.

315 keep a priority queue to quickly retrieve the next event. Second,  
 316 they keep a dependency graph to quickly locate all conditional in-  
 317 tensity rates that need to be updated after an event is fired. Third,  
 318 they re-use previously sampled reaction times to update unused re-  
 319 action times. This minimizes random number generation, which  
 320 can be costly. Priority queues and dependency graphs have also  
 321 been used in the context of social media [3] and epidemics [9] sim-  
 322 ulation. In both cases, the phenomena are modelled as point pro-  
 323 cesses.

324 We prefer to call this class of methods *queued thinning* methods  
 325 since most efficiency gains come from maintaining a priority queue  
 326 of the next event times. Up to this point we assumed that we were  
 327 sampling from a global process with a mark distribution that could  
 328 generate any mark  $k$  given an event at time  $t$ . With queuing, it is  
 329 possible to simulate point processes with a finite space of marks  
 330 as  $M$  interdependent point processes — see Definition 6.4.1 [2]  
 331 of multivariate point processes — doing away with the need to draw  
 332 from the mark distribution at every event occurrence. Alternatively,  
 333 it is possible to split the global process into  $M$  interdependent pro-  
 334 cesses each of which has its own mark distribution.

335 Algorithm 5, presents a method for sampling a superposed point  
 336 process consisting of  $M$  processes by keeping the strike time of  
 337 each process in a priority queue  $Q$ . The priority queue is initially  
 338 constructed in  $O(M)$  steps in Lines 4 to 7 of Algorithm 5. In con-  
 339 trast to *thinning* methods, updates to the conditional intensity de-  
 340 pend only on the size of the neighborhood of  $i$ . That is, processes  $j$   
 341 whose conditional intensity depends on the history of  $i$ . If the graph  
 342 is sparse, then updates will be faster than with *thinning*.

343 A source of inefficiency in some implementations of *queued thin-*  
 344 *ning* algorithms such as [3] is the fact that one goes through mul-  
 345 tiple rejection cycles at time  $t$  before accepting a time candidate  
 346  $t < t_i$  for process  $i$ . This requires looking ahead in the future. In  
 347 addition to that, if process  $j$ , which  $i$  depends on, takes place be-  
 348 fore  $t_i$ , then we need to repeat the whole thinning process to obtain  
 349 a new time candidate for  $i$ .

350 In Algorithm 5, we take a different approach which performs thin-  
 351 ning in synchrony with the main loop, avoiding look ahead and  
 352 wasted rejections. Our main contribution is to modify the main loop  
 353 of previous thinning algorithms to allow at most one event proposal  
 354 for each sub-process for each time step. The proposed candidates  
 355 are always added to the priority queue  $Q$  because we need to stop at  
 356 each proposed time. When the candidate is pre-rejected, we update  
 357 the bounds and make a new proposal. Alternatively, if the candidate  
 358 time has not been pre-rejected, we draw the acceptance threshold  
 359 and compute the intensity rate to make a decision. If the candi-  
 360 date is accepted, we trigger a new round of thinning. Otherwise,  
 361 we update the bounds and make a new proposal. Overall, we avoid  
 362 unnecessary updates. Additionally, thinning is now synced with the  
 363 main loop, which allows the coupling of this simulator with other  
 364 algorithms that step chronologically through time. These include  
 365 ordinary differential equation solvers, enabling us to simulate jump  
 366 processes with rates given by a differential equation. This is the first  
 367 *queued thinning* synced algorithm we are aware of.

368 Since Algorithm 5 can be mapped to a *non-queued thinning* algo-  
 369 rithm — see [3] —, it can simulate any point process on the real  
 370 line with a non-negative, left-continuous, history-adapted and lo-  
 371 cally bounded intensity rate as per Proposition 7.5.1 [2].

## 372 5. Implementation

373 `JumpProcesses.jl` is a Julia library for simulating jump —  
 374 or point — processes which is part of Julia’s SciML organiza-  
 375 tion. Jumps are handled via callbacks that are checked at the

---

**Algorithm 4** Generates the next candidate time for *queued thin-*  
*ning*.

---

```

1: procedure QUEUETIME( $t, \lambda^*, H_t$ )
2:   update  $\lambda^*$ 
3:   find  $\bar{B}^*, \underline{B}^*$  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  $t, \bar{B}^*, \underline{B}^*,$  accepted
13: end procedure

```

---



---

**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:      $(t_i, \bar{B}_i^*, \underline{B}_i^*, a_i) \leftarrow$  QueueTime( $0, H_{T-}, \lambda_i^*(\cdot)$ )
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  $v \sim U[0, \bar{B}_i^*]$ 
15:    if  $a_i$  and  $(v > \underline{B}_i^*)$  and  $(v > \lambda^*(t))$  then
16:       $a_i \leftarrow$  false
17:    end if
18:    if  $a_i$  then
19:       $n \leftarrow n + 1$ 
20:       $t_n \leftarrow t$ 
21:      update  $f^*$  and draw the mark  $k_n \sim f_i^*(k | t_n)$ 
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:         $(t_j, \bar{B}_j^*, \underline{B}_j^*, a_j) \leftarrow$  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:       $(t_i, \bar{B}_i^*, \underline{B}_i^*, a_i) \leftarrow$  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

```

---

376 end of each time-step of some time evolution algorithm, *e.g.* an  
 377 ODE solver from `OrdinaryDiffEq.jl`, a stochastic differen-  
 378 tial equation solver from `StochasticDiffEq.jl`, or the pure-  
 379 jump process `SSAStepper` provided by `JumpProcesses.jl`.  
 380 In simple terms, callbacks are functions that can be arbitrarily  
 381 called at each step of the main loop of a time-stepping method.  
 382 Our discussion in Section 4 identified three exact methods for  
 383 simulating point processes. In all the cases, we identified two

384 mathematical constructs required for simulation: the intensity  
 385 rate and the mark distribution. In `JumpProcesses.jl`, these  
 386 can be mapped to user defined functions `rate(u, p, t)` and  
 387 `affect!(integrator)`. The former takes the current state of  
 388 the system, `u`, user provided parameters, `p`, and the current time, `t`,  
 389 and returns the value of the intensity function at this time. The latter  
 390 takes the solver `integrator` object, which stores all solution  
 391 information, and updates it, including the state `integrator.u`,  
 392 for whatever changes should occur when the jump it encodes fires  
 393 at the time `integrator.t`. The library provides APIs for defining  
 394 processes based on the nature of the intensity rate and the  
 395 intended simulation algorithm. Processes simulated using exact  
 396 sampling methods can choose between `ConstantRateJump` and  
 397 `VariableRateJump`. While the former expects the rate between  
 398 jumps to be constant, the latter allows for time-dependent rates.  
 399 The library also provides the `MassActionJump` API to define  
 400 large systems of point processes that can be expressed as mass ac-  
 401 tion type reaction equations. Finally, `RegularJump` is intended  
 402 for tau-leaping methods.

403 The *inverse* method as described around Equation 4.2 uses  
 404 root finding to calculate the next jump time. Jumps to be  
 405 simulated via the *inverse* method must be initialized as a  
 406 `VariableRateJump`. `JumpProcesses.jl` builds a continuous  
 407 callback following the algorithm in [20] and passes the problem  
 408 to an `OrdinaryDiffEq.jl` integrator, which easily inter-  
 409 operates with `JumpProcesses.jl` (both libraries are part of  
 410 the *SciML* organization, and by design built to easily compose).  
 411 `JumpProcesses.jl` does not yet support the CHV ODE based  
 412 approach.

413 Alternatively, *thinning* methods can be simulated via discrete  
 414 steps. In the context of the library, any method that uses thinning  
 415 via a discrete callback is called an *aggregator*. There are twelve  
 416 different aggregators which we discuss below and are summarized  
 417 in Table 4 in the Annex. Aggregator’s handle sampling the next  
 418 jump time and type, which is then read via the callback by the user-  
 419 selected time-stepper.

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

427 As all the constant rate *thinning* aggregators only support  
 428 `ConstantRateJumps` and `MassActionJumps`, *i.e.* the intensi-  
 429 ty between jumps is constant, Algorithm 3 short-circuits to  
 430 quickly return  $t \sim \exp(\bar{B}) = \exp(\lambda_n)$  as discussed in Subsec-  
 431 tion 4.2. Next, the mark distribution becomes the categorical distri-  
 432 bution weighted by the intensity of each process. That is, given  
 433 an event at time  $t_n$ , we have that the probability of drawing pro-  
 434 cess  $i$  out of  $M$  sub-processes is  $\lambda_i^*(t_n)/\lambda^*(t_n)$ . Conditional on  
 435 sub-process  $i$ , the corresponding `affect!(integrator)` is in-  
 436 voked, that is,  $k_n \sim f_i^*(k | t_n)$ . So all sub-processes could poten-  
 437 tially be marked, but note users need to handle any additional sam-  
 438 pling related to such marks within their provided `affect!` func-  
 439 tion. Where most implementations differ is on updating the mark  
 440 distribution in Line 11 of Algorithm 2 and the conditional intensity  
 441 rate in Line 3 of Algorithm 3.

442 `Direct` and `DirectFW` follow the *direct* method in [7] which  
 443 re-evaluates all intensities after every iteration scaling at  $O(K)$ . It  
 444 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 de-  
 pendency graph. `SortingDirect` draws from the ground pro-  
 cess, but keeps the intensity rate in a loosely sorted array follow-  
 ing [15]. `RDirect` is a rejection-based direct method which as-  
 signs 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 homoge-  
 neous Poisson process with the maximum rate, then randomly se-  
 lects a candidate process and confirms the candidate only if its rate  
 is above a random proportion of the maximum rate. `DirectCR` —  
 from [21] — is a composition-rejection method that groups sub-  
 processes with similar rates using a priority table. Each group is as-  
 signed the sum of all the rates within it. We apply a routine equiv-  
 alent 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 [22] — follows Algorithm 2 very closely in the case where  
 the bounds are constant between jumps. `RSSACR` — from [23] —  
 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.  
 Finally, we have what we call the *queued thinning* ag-  
 gregators. Starting with aggregators that only support  
`ConstantRateJumps` we have, `FRM`, `FRMFW` and `NRM`.  
`FRM` and `FRMFW` 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 *queued thinning* method sourced  
 from [4]. `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 em-  
 phasis on implementing processes commonly studied in statisti-  
 cs such as self-exciting point processes characterized by time-  
 varying and history-dependent intensity rates. Our latest ag-  
 gregator, `Coevolve`, which is an implementation of Algo-  
 rithm 5, addresses this gap. This is the first aggregator that sup-  
 ports `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 Al-  
 gorithm 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 ob-  
 tain the next candidate time for the first round of iteration of the  
*thinning* procedure in Algorithm 3. This saves one round of sam-  
 pling from the exponential distribution, which translates into a  
 faster algorithm. Second, it adapts to processes with constant intensi-  
 ty 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.

```

529 using JumpProcesses
530 rate(u, p, t) = p[1]
531 affect!(integrator) = (integrator.u[1] += 1;
532   nothing)
533
534 jump = ConstantRateJump(rate, affect!)
535 u, tspan, p = [0.], (0., 200.), (0.25,)
536 dprob = DiscreteProblem(u, tspan, p)
537 jprob = JumpProblem(dprob, Direct(), jump;
538   dep_graph=[[1]])
539 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.

```

550 using JumpProcesses
551 h = Float64[]
552 rate(u, p, t) = p[1] +
553   p[2]*sum(exp(-p[3]*(t-_t)) for _t in h; init=0)
554 lrate(u, p, t) = p[1]
555 urate = rate
556 rateinterval(u, p, t) = 1/(2*urate(u,p,t))
557 affect!(integrator) = (push!(h, integrator.t);
558   integrator.u[1] += 1; nothing)
559 jump = VariableRateJump(rate, affect!; lrate,
560   urate, rateinterval)
561 u, tspan, p = [0.], (0., 200.), (0.25, 0.5, 2.0)
562 dprob = DiscreteProblem(u, tspan, p)
563 jprob = JumpProblem(dprob, Coevolve(), jump;
564   dep_graph=[[1]])
565 sol = solve(jprob, SSAStepper())
566

```

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

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

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

Then the transformed simulated interval is given by:

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

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

### 6.2 Benchmarks

We conduct a set of benchmarks to assess the performance of the `JumpProcesses.jl` aggregators described in Section 5. All benchmarks are available in `SciMLBenchmarks.jl`<sup>5</sup>. All were run in `BuildKite`<sup>6</sup> via the continuous integration facilities provided by the package maintainers. We have added two benchmark suites to assess the performance of the new aggregators under settings that could not be simulated with previous aggregators.

<sup>5</sup><https://github.com/SciML/SciMLBenchmarks.jl/tree/7d356203ea107d7343af1ce41d94b64847095d4a/benchmarks/Jumps> and <https://github.com/SciML/SciMLBenchmarks.jl/tree/7d356203ea107d7343af1ce41d94b64847095d4a/benchmarks/HybridJumps>

<sup>6</sup><https://buildkite.com/julialang/scimlbenchmarks-dot-jl/builds/1849#018c3980-5247-42ab-a7fe-3145209b26c5>

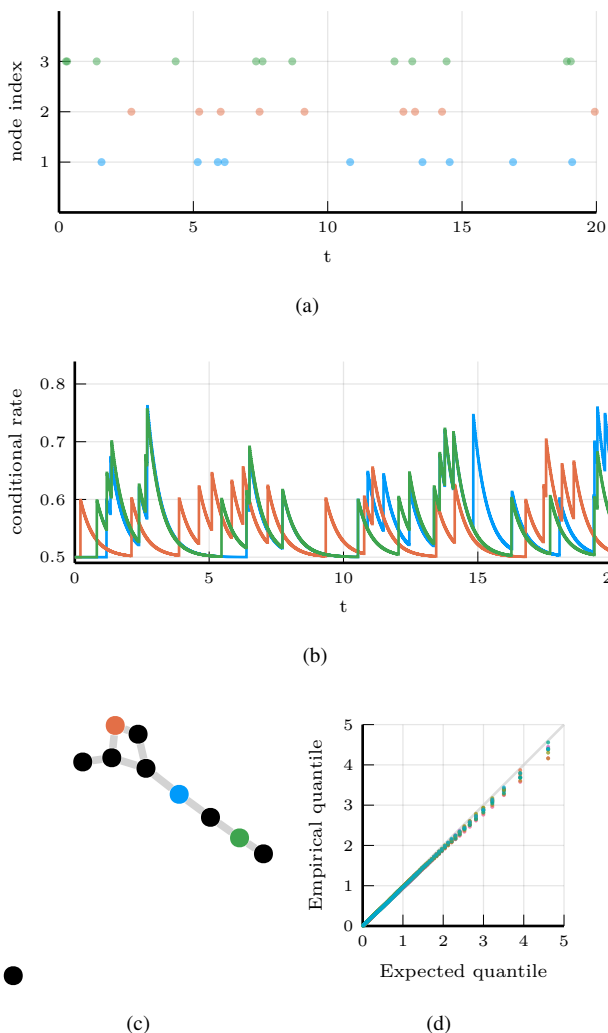


Fig. 1: Simulations of 10-nodes compound Hawkes process with parameters  $\lambda = 0.5, \alpha = 0.1, \beta = 2.0$  for 200 units of time. (a) and (b) sampled trajectory and intensity rate for a single simulation for the three selected nodes in (c) for the first 20 units of time. (c) underlying 10-nodes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

	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>	<b>0.24 ms</b>	<b>0.53 s</b>
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	<b>0.10 s</b>	0.43 ms	0.66 s
RSSACR	<b>0.46 s</b>	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 [14] (Multi-state), a simple negative feedback gene expression model (Gene I) and the negative feedback gene expression from [8] (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.

of queuing methods suggesting that there is a penalty when using such methods for jump processes whose rates are constant between jumps. We also note that the performance of `Coevolve` lag that of `NRM` despite the fact that `Coevolve` should take the same number of steps as `NRM` when no `VariableRateJump` is used. The reason behind this discrepancy is likely due to implementation differences, but left for future investigation.

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

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

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

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

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 [11] which also provides additional discussion and results on the Hawkes process. We derive the compound case here. Let  $t_{N_i} = \max\{t_{n_j} < t \mid j \in E_i\}$  and  $\phi_i^*(t)$  below.

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

First, we assess the speed of the aggregators against jump processes whose rates are constant between jumps. There are four such benchmarks: a 1-dimensional continuous time random walk approximation of a diffusion model (Diffusion), the multi-state model from Appendix A.6 [14] (Multi-state), a simple negative feedback gene expression model (Gene I) and the negative feedback gene expression from [8] (Gene II). We simulate a single trajectory for each aggregator to visually check that they produce similar trajectories for a given model. The Diffusion, Multi-state, Gene I and Gene II benchmarks are then simulated 50, 100, 2000 and 200 times, respectively. Check the source code for further implementation details.

Benchmark results are listed in Table 1. The table shows that no single aggregator dominates suggesting they should be selected according to the task at hand. However, `FRM`, `NRM`, `Coevolve` never dominate any benchmark. In common, they all belong to the family



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

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

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

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

664 We assess the benchmark in eight different settings. First, we run  
665 the *inverse* method. *Coevolve* and *CHV simple* using the brute  
666 force formula of the intensity rate which loops through the whole  
667 history of past events — Equation 6.3. Second, we simulate the  
668 same three methods with the recursive formula — Equation 6.6.  
669 Next, we run the benchmark against *CHV full*. All *CHV* spec-  
670 ifications are implemented with *PiecewiseDeterministic-*  
671 *MarkovProcesses.jl*<sup>7</sup> which is developed by Veltz, the au-  
672 thor of the *CHV* algorithm discussed in Subsection 4.1. Finally,  
673 we run the benchmark using the Python library *Tick*<sup>8</sup>. This library  
674 implements a version of the thinning method for simulating the  
675 Hawkes process and implements a recursive algorithm for comput-  
676 ing the intensity rate.

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

682 The recursive implementation of the intensity rate brings a consid-  
683 erable boost to the simulations, placing *Coevolve* as one of the  
684 fastest algorithms. As shown in Algorithm 5, every sampled point  
685 in *Coevolve* requires a number of expected updates equal to the  
686 expected degree of the dependency graph. Therefore, it is able to  
687 complete non-exploding simulations efficiently.

688 The Python library *Tick* remains competitive for smaller prob-  
689 lems, but gets considerably slower for bigger ones. Also, it is only  
690 specialized to the Hawkes process. Another drawback is that the  
691 library wraps the actual C++ implementation. In contrast, *Jump-*  
692 *Processes.jl* can simulate many other point processes with a  
693 relatively simple user-interface provided by the Julia language.

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

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

<sup>7</sup><https://github.com/rveltz/PiecewiseDeterministicMarkovProcesses.jl>

<sup>8</sup><https://github.com/X-DataInitiative/tick>

708 Finally, we introduce a new benchmark which is intended to assess  
709 the performance of algorithms capable of simulating the stochastic  
710 model of hippocampal synaptic plasticity with geometrical read-  
711 out of enzyme dynamics proposed in [19]. For short, we denote it  
712 as the synapse model. We chose to benchmark this model as it is  
713 representative of a complex biochemical model. It couples a jump  
714 problem containing 98 jumps affecting 49 discrete variables with  
715 a stiff, ordinary differential equation problem containing 34 con-  
716 tinuous variables. Continuous variables affect jump rates while the  
717 discrete variables affect the continuous problem. There are 3 stages  
718 to the simulation: pre-synaptic evolution, glutamate release, and  
719 post-synaptic evolution. Among the algorithms considered, only  
720 the *inverse* method implemented in *JumpProcesses.jl*, *Coe-*  
721 *volve* and *CHV* are theoretically able to simulate the synapse  
722 model. However, in practice, only the last two complete at least one  
723 benchmark run. The original synapse problem was described as a  
724 PDMP, so we do not make the distinction between *CHV simple*  
725 and *full* in this benchmark.

Benchmark results are displayed in Table 3. We observe that *CHV*  
is the fastest algorithm completing the synapse evolution in about  
half of the time it takes *Coevolve* with less than half of the allo-  
cations. Further investigation reveals that the thinning procedure in  
*Coevolve* reaches an average of 70 percent over all jumps which  
then leads to 2 to 3 times more function evaluations and Jaco-  
bians created compared to *CHV*. Our implementation adds stop-  
ping times via a call to `register_next_jump_time!` even for  
rejected jumps — we do not know a jump will be rejected until  
evaluated. This then leads the ODE solver to step to those times and  
make additional function evaluations and Jacobians. Lemaire *et*  
*al.* [12] performs a similar benchmark in which they compare the  
Hodgkin-Huxley model against different thinning conditions and  
against an ODE approximation. They find that a thinned algorithm  
with optimal boundary conditions can run significantly faster than  
the ODE approximation. Thus, there could be plenty of room to  
improve the performance of *Coevolve* in our setting.

A disadvantage of *CHV* compared with *Coevolve* is that it sup-  
ports limited saving options by design. To save at pre-specified  
times would require using the fact that solutions are piecewise con-  
stant to determine solutions at times in-between jumps — and for  
coupled ODE-jump problems would require root-finding to deter-  
mine when  $s(u) = s_n$  for each desired saving time  $s_n$  in Equa-  
tion 4.8. The alternative proposed in [24] is to introduce an artificial  
jump to the model such as the homogeneous Poisson process with  
unit rate to sample the system at regular intervals. Alternatively,  
*Coevolve* allows saving at any arbitrary point. A common work-  
flow in simulating jump processes, particularly when interested in  
calculating statistics over time, is to pre-specify a precise set of  
times at which to save a simulation. In theory, this reduces mem-  
ory pressure, particularly for systems with large numbers of jumps,  
and can give increased computational performance relative to sav-  
ing the state at the occurrence of every jump. However, in the case  
of the synapse model, the number of candidate time rejections far  
surpasses the number of jumps. Therefore, reducing the number of  
saving points — *e.g.* only saving at start and end of the simulation  
— does not significantly reduce allocations or running time. Given  
these considerations, we decided to save after every jump and at  
regular pre-specified intervals that occur at the same frequency as  
the artificial saving jump used by *CHV*.

Another parameter that can affect the precision and speed of the  
synapse benchmark is the ODE solver. The author of *Piece-*  
*wiseDeterministicMarkovProcesses.jl* discuss some

of these issues in Discourse<sup>9</sup>. Some ODE solvers can be faster and more precise. Due to time constraints, we have not investigated this matter. The synapse benchmark uses the AutoTsit5(Rosenbrock23()) solver in both Coevolve and CHV. Further investigation of this matter is left to future research.

## 7. Conclusion

This paper demonstrates that JumpProcesses.jl is a fast, general-purpose library for simulating 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 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 [11]. It would be important to adapt these algorithms for general, compound branching processes to cater for a larger number of settings.

Finally, JumpProcesses.jl also includes algorithms for jumps over two-dimensional spaces. It might be worth conducting a similar comparative exercise to identify algorithms in statistics for 2-

and  $N$ -dimensional processes that could also be added to JumpProcess.jl as it has the potential to become the go-to library for general point process simulation.

## 8. Acknowledgements

This project has been made possible in part by grant number 2021-237457 from the Chan Zuckerberg Initiative DAF, an advised fund of Silicon Valley Community Foundation. SAI was also partially supported by NSF-DMS 1902854 and 2325185.

## 9. References

- [1] Joris Bierkens, Paul Fearnhead, and Gareth Roberts. The Zig-Zag process and super-efficient sampling for Bayesian analysis of big data. 47(3). doi:10.1214/18-AOS1715.
- [2] Daryl J. Daley and David Vere-Jones. *An Introduction to the Theory of Point Processes: Volume I: Elementary Theory and Methods*. Probability and Its Applications, An Introduction to the Theory of Point Processes. Springer-Verlag, 2 edition. doi:10.1007/b97277.
- [3] 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.
- [4] 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.
- [5] Daniel T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. 115(4). doi:10.1063/1.1378322.
- [6] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. 81(25). doi:10.1021/j100540a008.
- [7] 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.
- [8] 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.
- [9] Petter Holme. Fast and principled simulations of the SIR model on temporal networks. 16(2). doi:10.1371/journal.pone.0246961.
- [10] Günter Last and Mathew Penrose. *Lectures on the Poisson Process*. Cambridge University Press, 1st edition edition.
- [11] Patrick J. Laub, Young Lee, and Thomas Taimre. *The Elements of Hawkes Processes*. Springer International Publishing. doi:10.1007/978-3-030-84639-8.
- [12] Vincent Lemaire, Michèle Thieullen, and Nicolas Thomas. Exact Simulation of the Jump Times of a Class of Piecewise Deterministic Markov Processes. 75(3). doi:10.1007/s10915-017-0607-4.
- [13] P. a. W Lewis and G. S. Shedler. Simulation of nonhomogeneous poisson processes by thinning. 26(3). doi:10.1002/nav.3800260304.
- [14] 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.

<sup>9</sup><https://discourse.julialang.org/t/help-me-beat-lsoda/88236>

	V	Brute Force			Recursive				
		<i>Inverse</i>	Coevolve	<i>CHV simple</i>	<i>Inverse</i>	Coevolve	<i>CHV simple</i>	<i>CHV full</i>	<i>Tick</i>
Time	1	74.1 $\mu$ s	<b>4.8 <math>\mu</math>s</b>	203.1 $\mu$ s	76.6 $\mu$ s	<u>5.0 <math>\mu</math>s</u>	201.5 $\mu$ s	197.9 $\mu$ s	30.7 $\mu$ s
	10	10.0 ms	205.2 $\mu$ s	5.1 ms	3.8 ms	<b>73.5 <math>\mu</math>s</b>	471.6 $\mu$ s	607.3 $\mu$ s	<u>175.0 <math>\mu</math>s</u>
	20	89.6 ms	1.5 ms	48.9 ms	16.2 ms	<b>265.8 <math>\mu</math>s</b>	964.1 $\mu$ s	<u>902.3 <math>\mu</math>s</u>	1.2 ms
	30	274.2 ms	3.3 ms	129.5 ms	45.7 ms	<b>502.9 <math>\mu</math>s</b>	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>	<b>913.5 <math>\mu</math>s</b>	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=31</i>	2.4 s <i>n=9</i>	<b>1.5 ms</b>	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=15</i>	4.1 s <i>n=5</i>	<b>2.2 ms</b>	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=8</i>	6.8 s <i>n=3</i>	<u>3.0 ms</u>	6.9 ms	<b>3.0 ms</b>	89.5 ms
	80	15.5 s <i>n=1</i>	138.8 ms	3.3 s <i>n=5</i>	10.6 s <i>n=2</i>	<u>4.0 ms</u>	9.1 ms	<b>3.2 ms</b>	147.1 ms
	90	27.86 s <i>n=1</i>	139.7 ms	5.6 s <i>n=4</i>	16.0 s <i>n=1</i>	<u>5.3 ms</u>	11.8 ms	<b>3.9 ms</b>	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
<i>Inverse</i>	-	-
Coevolve	<u>4.9 s</u>	<u>94.0 Mb</u>
<i>CHV</i>	<b>2.7 s</b>	<b>43.5 Mb</b>

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.

884 [15] James M. McCollum, Gregory D. Peterson, Chris D. Cox,  
885 Michael L. Simpson, and Nagiza F. Samatova. The sort-  
886 ing direct method for stochastic simulation of biochemi-  
887 cal systems with varying reaction execution behavior. 30(1).  
888 doi:10.1016/j.compbiolchem.2005.10.007.

889 [16] James Meiss. *Differential Dynamical Systems, Re-*  
890 *vised Edition*. Mathematical Modeling and Computa-  
891 tion. Society for Industrial and Applied Mathematics.  
892 doi:10.1137/1.9781611974645.

893 [17] Y. Ogata. On Lewis’ simulation method for point processes.  
894 27(1). doi:10.1109/TIT.1981.1056305.

895 [18] Christopher Rackauckas and Qing Nie. `DifferentialEqua-`  
896 `tions.jl` A Performant and Feature-Rich Ecosystem for Solv-  
897 ing Differential Equations in Julia. 5(1). doi:10.5334/jors.151.

898 [19] Yuri E. Rodrigues, Cezar M. Tigaret, H el ene Marie, Cian  
899 ODonnell, and Romain Veltz. A stochastic model of hip-

900 pocampal synaptic plasticity with geometrical readout of en-  
901 zyme dynamics. doi:10.1101/2021.03.30.437703.

902 [20] Howard Salis and Yiannis Kaznessis. Accurate hybrid  
903 stochastic simulation of a system of coupled chemical or bio-  
904 chemical reactions. 122(5). doi:10.1063/1.1835951.

905 [21] Alexander Slepoy, Aidan P. Thompson, and Steven J. Plimp-  
906 ton. A constant-time kinetic Monte Carlo algorithm for sim-  
907 ulation of large biochemical reaction networks. 128(20).  
908 doi:10.1063/1.2919546.

909 [22] Vo Hong Thanh, Corrado Priami, and Roberto Zunino.  
910 Efficient rejection-based simulation of biochemical  
911 reactions with stochastic noise and delays. 141(13).  
912 doi:10.1063/1.4896985.

913 [23] Vo Hong Thanh, Roberto Zunino, and Corrado Pri-  
914 ami. Efficient Constant-Time Complexity Algorithm for  
915 Stochastic Simulation of Large Reaction Networks. 14(3).  
916 doi:10.1109/TCBB.2016.2530066.

917 [24] Romain Veltz. A new twist for the simulation  
918 of hybrid systems using the true jump method.  
919 doi:10.48550/arXiv.1504.06873. arxiv:1504.06873.

920 **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		[15]
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		[21]
RSSA	Rejection-based stochastic simulation algorithm	Processes are assigned lower- and upper-bounds. Sample on upper-bounds.	ground	graph	x	x		[22]
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		[23]
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		[4]
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.