BiPSim: a flexible and generic stochastic simulator for cell processes - Supplementary Information

December 9, 2019

Contents

1 Introduction

This document walks through the choices in design that we made while developing BiPSim (the code is documented using Doxygen, technical details are therefore best found in the Doxygen-generated manual). It highlights the central classes in BiPSim's architecture and how classes interact.

The first section focuses on the base components of the simulator, reactants and reactions. It starts with a global overview of all rectants and reactants. A second subsection describes the same element again, but goes further into hypotheses and critical design elements. Appendices are added to describe elements that have been important in the simulator development but did not fit naturally in the main document (testing strategies, utility classes, etc.).

The second section focuses on the implementation of Gillespie's Stochastic Simulation Algorithm (SSA). It starts with an overview of the SSA, insisting the trade-off between two of its components: selecting the next reaction to perform and updating reaction rates. The second subsection presents various strategies to select the next reaction implemented in BiPSim: the direct method, the binary search and the composition-rejection method. The third subsection presents efficient strategies to update reaction rates as implemented in BiPSim, which play an essential role in the final performance of the simulator. Appendices provide additional information about the implementation and some perspectives.

2 Implementation of reactions and reactants

3 Implementation of Gillespie's Stochatic Simulation Algorithm

3.1 Introduction

Gillespie's algorithm was designed to simulate chemical reaction networks, even in conditions where there are only a few molecules. In this section we briefly present how it was conceived and implemented.

3.1.1 Principles underlying Gillespie's algorithm

We suppose we are given a set of chemical reactions that we can write, in general as

$$a_1A_1 + a_2A_2 + ... + a_rA_r \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} b_1B_1 + .. + b_pB_p$$

the simplest reaction being a complexation reaction of the form

$$A + B \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} C$$

Assumption 3.1 (Well-mixed medium)**.** The first important hypothesis is that the molecules of every chemical species are distributed uniformly across space at all times. Then the probability that a molecule of species A encounters a molecule of species B is proportional to n_An_B where n_A is the number of molecules of species A and n_B the number of molecules of species B. Note that the probability that a molecule of species A encounters another molecule of species A is proportional to $n_A(n_A - 1)$.

Assumption 3.2 (Low-order reactions)**.** The second hypothesis is that reactions represent low-order phenomena, *i.e.* they result uniquely of the concomitant encounter of all species involved (as opposed to a sequence of events implying one or the other species).

Definition 3.3 (Propensity)**.** The propensity of a reaction is the rate at which it occurs. In the simplest case

$$A + B \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} C$$

the propensity of the complexation reaction is $k_1n_An_B$ and the propensity of the decomplexation is $k_{-1}n_C$. In a more general reaction

$$a_1A_1 + a_2A_2 + ... + a_rA_r \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} b_1B_1 + .. + b_pB_p$$

verifying the two assumptions above, the propensity of the forward reaction is

$$r = k_1 \prod_{i=0}^{r} n_{A_i}(n_{A_i} - 1)...(n_{A_i} - a_i + 1)$$

Property 3.4 (Exponential distribution of reaction timing.)**.** *Under the two assumptions above, it can be shown that the timing of reactions are distributed exponentially, where the parameter of the exponential distribution is the propensity of the reactions. Because the propensity depends on the number of molecules, the probability density evolves over time.*

Assumption 3.5 (Memorylessness of reaction network.)**.** We suppose that in the system of reactions considered, the occurrences of reactions are determined only based on their propensity. Occurrences of reactions are not independent in the sense that a reaction may change the propensity of another reaction (if they have chemicals in common). However, a reaction may not facilitate another reaction in any other way than changing its propensity.

Property 3.6 (Next reaction in a network.)**.** *Under the current assumptions, at any time, the next reaction timing is determined by the first reaction that fires. Because the next reaction timings of all reactions are given by exponential distributions, the next reaction timing is the minimum of all these exponential drawings. Mathematically the minimum of exponentially distributed random variables is still exponentially distributed. More precisely, suppose there are N reactions and let r_1, ..., r_N be their propensities, then the next reaction timing follows an exponential distribution with parameter $\sum_{i=0..N} r_i$ and the probability that the next reaction is reaction k is $r_k / \sum_{i=0..N} r_i$.*

3.1.2 The Stochastic Simulaton Algorithm (SSA) and its variants

Gillespie et al. (2013) propose a good review of the original algorithm and some of its variants. The original SSA algorithm that can be summarized as follows:

- STEP 1: Update propensity functions.
- STEP 2: Select reaction to perform and next reaction time, perform reaction.

Most of the effort is spent on studying how to optimize the second step, but we will see that the two steps are actually interrelated. Drawing statistics have to follow the properties described above. In a sense, the SSA and its variants are said to be (statistically) exact.

3.1.3 Aim of this section

Our aim was to try various methods to optimize the second step of the SSA for a large biological system. We reviewed the literature and implemented the main alternatives and compared them on a benchmark inspired by our research project. We provide some perspectives that could be worked on to improve the algorithms or the way they interact with the updating step.

3.2 Selecting reaction to perform

We start with the second step of the SSA, selecting a reaction given some propensities. Most algorithms presented here are reviewed in Gillespie et al. (2013).

Formally, the problem is quite simple: we are given a set of N real values $\{r_1, r_2, ..., r_N\}$ and we draw index i with probability $p_i = r_i / \sum r_i$. Mathematically, this is a multinomial distribution and thus could be drawn as such by any standard random number library.

We will start by stating a standard technique to perform mulitnomial drawings. We will then introduce methods that increase the speed of the drawing by using *a priori* knowledge.

3.2.1 Direct method

Principle The first method that was used historically is staightforward and sometimes referred to as *biased wheel*. Schematically speaking, you could imagine a wheel similar to "wheel of fortune", except the size allowed to each index on the wheel is proportional to its propensity value, so that large value have a larger probability to be drawn when the wheel is spinned (Fig. 1).

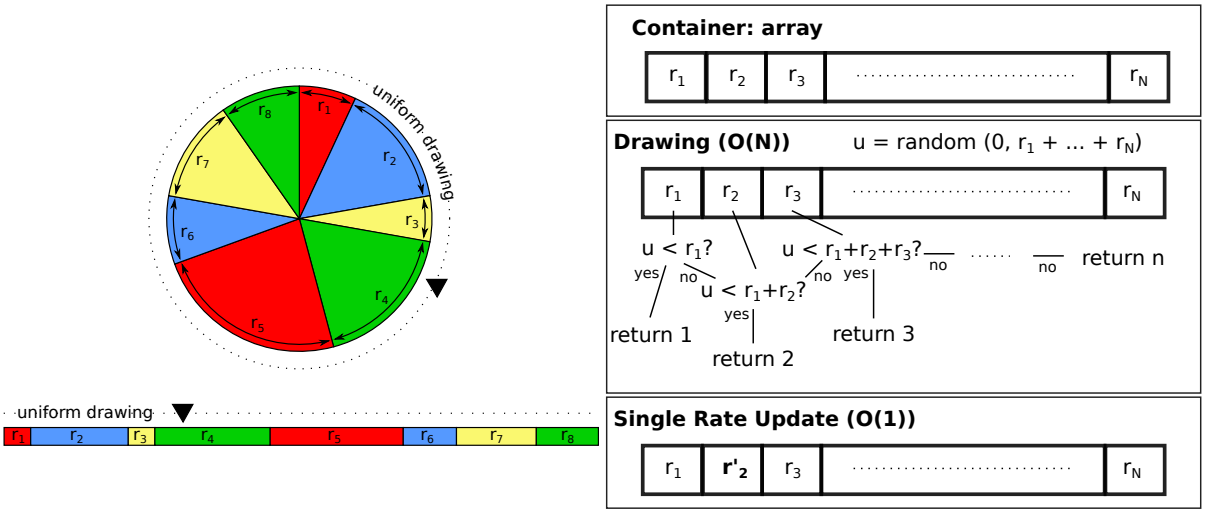
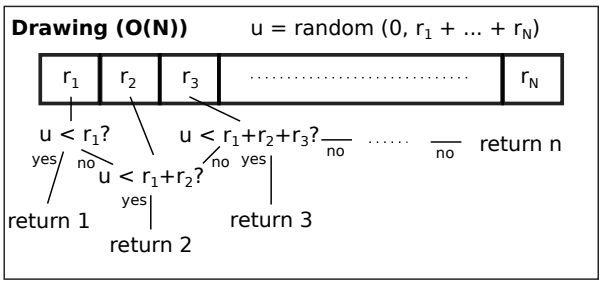


Figure 1: (Left) Illustration of drawing along a biased wheel and its equivalent representation as a segment. (Right) Container and algorithms used to maintain the structure.

Generally the wheel is seen as a segment subdivided into N subsegments of length r_1, ..., r_N. A value u is drawn on this $[0, \sum r_i)$ segment. We proceed iteratively to find to which subsegment u belongs. If $u < r_1$, it belongs to subsegment 1. If $r_1 \leq u < r_1 + r_2$, it belongs to subsegment 2, etc.

Sketch of algorithm and complexity Worst case of the drawing (Fig. 2) occurs when u is in the last subsegment, so the loop has to be iterated N times, yielding $O(N)$ complexity.



Data: Array r of size N, R = sum (r).
Result: Index drawn according to multinomial drawing.

```
u = uniform ([0, R));
index = 1;
cum_sum = r[1];
while u ≥ cum_sum do
    index = index + 1;
    cum_sum = cum_sum + r[index];
end
return index
```

Figure 2: Direct drawing method

3.2.2 Next reaction method

The next reaction method is based on the direct method. It is used in systems where it is known that propensities are not uniform. The point is to accelerate the direct method by sorting (at least rougly) propensities within the vector. This does not change drawing statistics but accelerates finding where a drawing is located on the wheel. For example, say a propensity takes up 50% of total propensity and is located at the beginning of the vector. Then each random drawing $u < 0.5 \sum r_i$ will be instantly found to fall into the first sector of the biased wheel. In a non-sorted vector, we would have to loop through several small sectors before finding the big one.

Because the next reaction method is only a twist of the direct method, we do not go into furter details now. We will also omit it in complexity analysis, as it has the same worst-case complexity as the direct method. We will only reference it again in the Experiment section.

3.2.3 Binary tree

Principle In this approach, we organize propensities inside a tree. Propensities are placed in the leaves of the tree. Nodes are then assembled iteratively 2 by 2 to compute the sum of all propensities (Fig. 3).

The idea is that with the structure *in place*, finding the index that has been drawn is quicker. Similarly to the standard drawing, a value u is drawn on the $[0, R = \sum r_i)$ segment. We start from the root node and need to find on which side of the tree u lies. The two children nodes summarize how much weight there is on each side of the tree, say w_{left} and w_{right} respectively. If $u < w_{\text{left}}$, we descend to the left child node and proceed the same way until we reach a leaf. If $u \geq w_{\text{left}}$, we descend to the right child and we proceed iteratively with $u = u - w_{\text{left}}$.

This procedure is actually very similar to the biased wheel method, except we perform

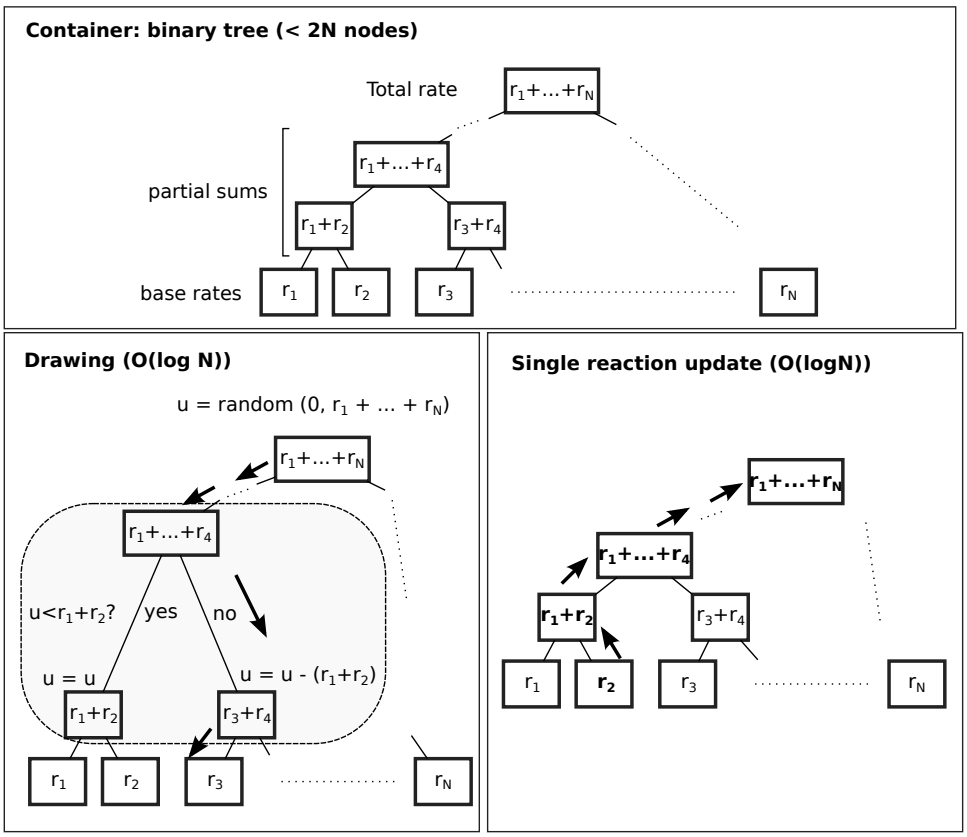


Figure 3: Binary tree containing propensities. Propensity values are found at the leaves of the tree. Intermediate nodes represent partial sums of nodes below, root holds the sum of all propensities.

some kind of progressive zooming in on the subsegments delimited by the propensity values (Fig. 3).

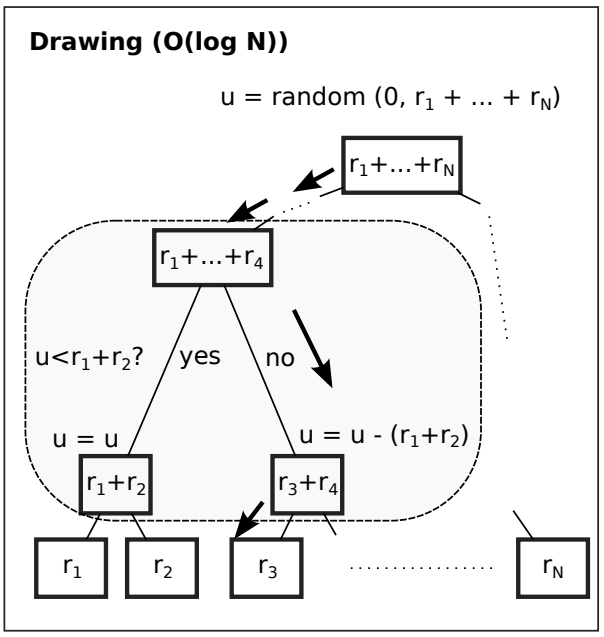
Sketch of algorithm and complexity Complexity for performing the drawing (Fig. 4) is given by the depth of the tree, $\lceil \log_2 N \rceil$, which is $O(\log N)$.

Complexity for updating the tree (Fig. 5) is also given by depth of the tree, $O(\log N)$. Note that we need not update every node in the tree, only the parents of the updated leaf up to the root node.

3.2.4 Hybrid method

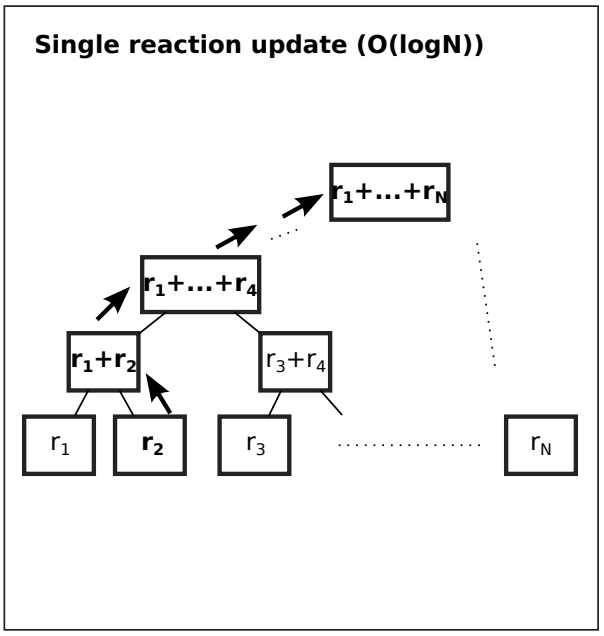
Principle In this approach, we organize propensities into groups and use a different drawing method: *rejection-base drawing*. This approach has been presented in Slepoy et al. (2008).

Rejection-based drawing The aim is to perform a multinomial drawing, similar to what is done by a biased wheel or a binary tree. The problem with the latter methods



Data: Binary tree with propensities at its leaves.
Result: Index drawn according to multinomial drawing.
u = uniform ([0, tree.root.value));
node = tree.root;
while *node is not a leaf* **do**
  if *u < node.left_child.value* **then**
    node = node.left_child;
  else
    node = node.right_child;
    u = u - node.left_child.value;
  end
end
return *node.index*

Figure 4: Binary tree: drawing method.



Data: Binary tree with propensities at its leaves. Index i_update of propensity to update, new propensity value p_update.
Result: Updated binary tree.
node = tree.leaf [i_update];
node.value = p_update;
while *node is not root* **do**
  node = node.parent;
  node.value =
    node.left_child.value +
    node.right_child.value;
end

Figure 5: Binary tree: update method.

is that we need to iterate through some structure before finding the right value. With rejection-based drawing, we attempt to *jump* to the right solution. Let $\{r_i\}$ be the propensities of the N reactions in the system and $R = \sum_{1 \leq i \leq N} r_i$.

1. We choose a value r_M such that $\forall i, r_M \geq r_i$.

2. Until a good candidate is found.
    a) We draw a random number i between 1 and N (with replacement).
    b) We draw a random number u on the $[0, r_M]$ segment. If $u > r_i$, we reject i, else we keep it.

A careful proof shows that the probability of drawing index i is equal to r_i/R (Serebrinsky, 2011). Note that the choice of r_M is critical for the efficiency of the method (Fig. 6A). Formally, the probability to accept a candidate is $\sum_i \mathbb{P}(\text{draw } i)\, \mathbb{P}(\text{accept } i) = \sum_i 1/N \times r_i/r_M = R/(Nr_M)$. If applied naively, the number of candidates to loop through is a geometric law with parameter $R/(Nr_M)$. The expected number of candidates is thus Nr_M/R. For a uniform distribution, this value can be 1, but in general, it yields bad results (Fig. 6B).

Group method The idea behind the algorithm is to improve the acceptation probability by placing propensities in *groups*:

1. We draw a group index by using a classical method (biased wheel or binary tree).
2. We draw a propensity inside the group by using the rejection-based method.

Slepoy et al. (2008) propose placing propensities into binary groups. They choose a base rate b. Groups are of the form $(0, b]$, $(b, 2b]$, $(2b, 4b]$, *etc.* $(0, b]$ contains all propensities between 0 and b, and so on. When applying the rejection method to any of these groups (except $(0, b]$), the acceptation probability is $\geq 1/2$ (Fig. 6C). Note that the number of groups K does not generally depend on N, it only depends on the highest propensity value. In general, it remains relatively small.

Suppose the structure is already in place, *i.e.* propensities are placed in the right group and the total propensity for each group is known. Step 1 is at most $O(K)$, which is independent of N. Step 2 requires less than 2 candidates on average, so it is $O(1)$. This results in $O(1)$ globally, making it significantly more efficient than the two previous methods. However, we will see that its implementation is also trickier in order to preserve this theoretical complexity.

Sketch of algorithm and complexity Because of the group structure, the loop in Figure 8 is $O(1)$ (see above). The first multinomial drawing is at most $O(K)$, so the complexity is globally $O(K)$. Because K, the number of groups, does not naturally scale with N, the number of reactions, the complexity is overall $O(1)$, as N is the real variable of interest here.

At first sight, updating the group structure is also $O(1)$ (Fig. 9). However, the parts about removing or inserting a reaction into a group must be carefully implemented in order to achieve that result.

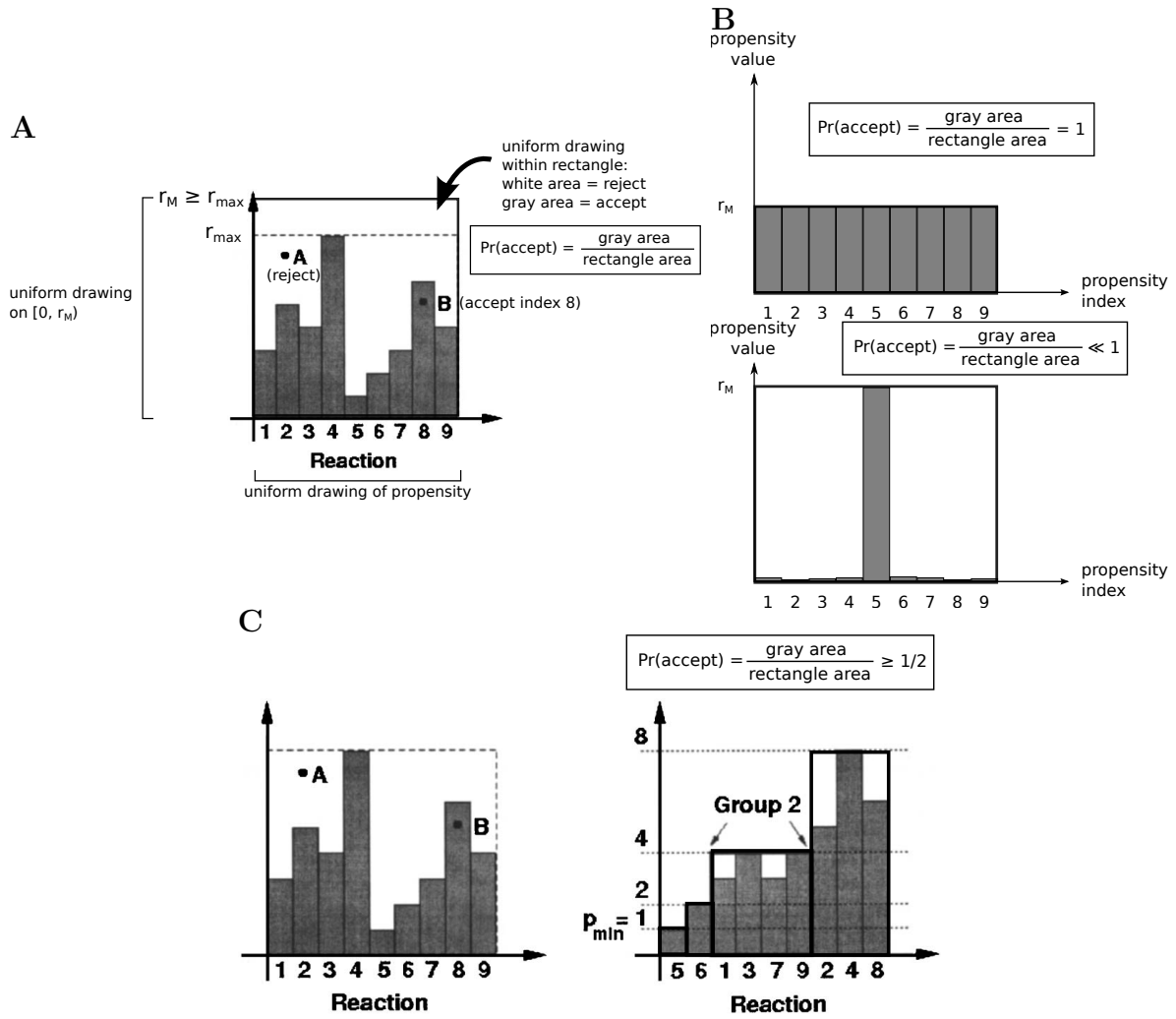


Figure 6: Rejection based drawing (adapted from Slepoy et al. (2008)). (A) Geometric illustration of rejection principle. Drawing occurs in a 2D space, with propensities aligned along the x axis and their value given by gray bars along the y axis. A drawing is accepted if it falls into the gray domain. Note that the probability to draw a propensity is proportional to its value, as an accepted drawing will be distributed uniformly across the gray domain. (B) Examples displaying efficiency of the technique (maximal for uniform propensities, minimal when some are very high and most are very low). (C) Sorting propensities into groups whose limits are powers of 2 ensures a minimal 1/2 acceptation probability *within a given group*.

3.2.5 Summary

Table 1 summarizes the worst case complexity of four methods presented. Note that the update complexity was derived in the case where onle one propensity needed to be updated. To obtain the overall complexity, we need to take into account the number of

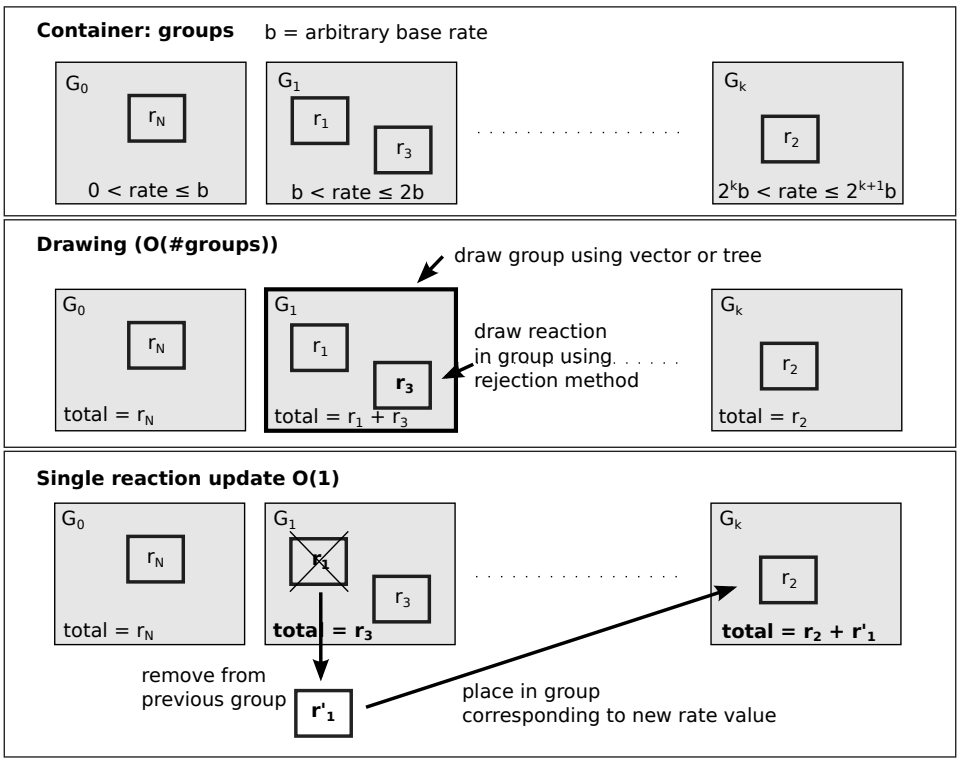
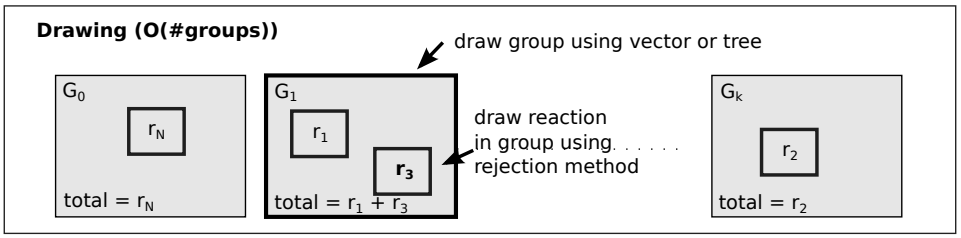


Figure 7: Hybrid method using group structure and rejection algorithm.



Data: $K+1$ groups, group k containing propensities whose value falls in the interval $(0, b]$ if k=0, $(2^{k-1}b, 2^k b]$ if k > 0. Propensities are stored as a couple containing their value and original index.

Result: Index drawn according to multinomial drawing.

`// drawing using a direct method like binary tree or biased wheel`

group = groups [multinomial (group[0].total_propensity, ...,
group[K].total_propensity)];

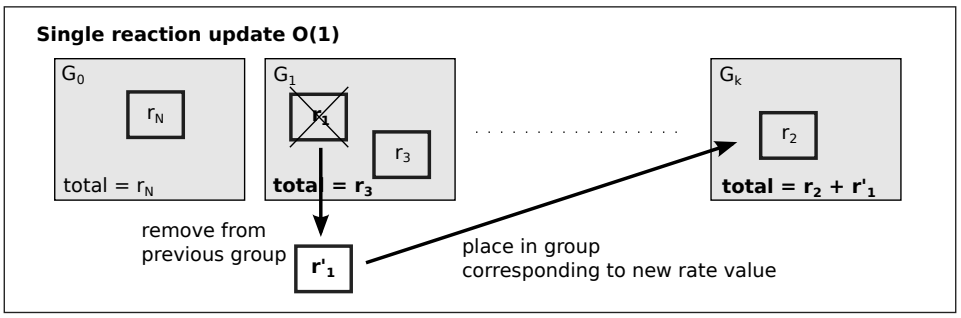
repeat

| candidate = group.propensities [uniform (1, group.number_propensities)];

until *candidate.value > uniform (0, group.max_propensity)*;

return *candidate.index*

Figure 8: Hybrid method: drawing method.



Data: $K+1$ groups, group k containing propensities whose value falls in the interval $(0, b]$ if k=0, $(2^{k-1}b, 2^k b]$ if k > 0. Propensities are stored as a couple containing their value and original index. Index i_update of propensity to update, new propensity value p_update.
Result: Updated group structure.
Function *group_index (propensity)*
  if *propensity* $\geq b$ **then**
    return $\lceil \log_2(\textit{reaction.propensity}/b) \rceil$
  else
    return *0*
  end
propensity = propensity corresponding to index i_update;
previous_group = groups [group_index (propensity.value)];
Remove propensity from previous_group and update group's total propensity;
new_group = groups [group_index (p_update)];
propensity.value = p_update;
Add propensity to new_group and update group's total propensity;

Figure 9: Hybrid method: update method.

reactions U whose propensity needs to be updated. A naive analysis indicates that the binary tree could be less efficient than the direct method depending on U and N (Tab. 1). In the next section, we will analyze how U can influence the efficiency of each method.

Method	Drawing Complexity	Update Complexity (one propensity)	Total Complexity
Direct Method	$O(N)$	$O(1)$	$O(N)$?
Binary Tree	$O(\log N)$	$O(\log N)$	$O(U \log N)$?
Hybrid Method	$O(1)$	$O(1)$	$O(U)$?

Table 1: Comparison of worst-case complexities of methods presented here. N is the number of reactions in the system, $U \leq N$ the number of reactions whose propensity needs to be updated. The last column is a projection based on the first two columns, real total complexities are given in the next section.

3.3 Perspectives

Because the hybrid method leads to $O(1)$ complexity for drawing reactions, it is not possible to do substantially better on that part of the algorithm. Emphasis should be placed on winning time for updating reactions.

Selection reaction: tau-leaping The first idea to explore is abandon the exact framework and only update reactions once in a while by introducing time steps for updates. The problem with this method is that a reactant can run out during the time step and still be consumed, leading to negative chemical population. This has to be avoided at all costs. Automatical time step adjustments, termed **tau-leaping**, have been proposed to resolve this issue (Cao et al., 2005). However, these methods tend to be less efficient if there is constantly a reactant whose concentration is low, which would be the case in a whole-cell simulation scenario. It could be interesting to derive an algorithm that is able to classify reactions into sensitive (if they involve low-concentration reactants) and unsensitive. It would then assess each pool its own time step, with sensitive reacion rates being updated more often.

Propensity updating: factorizing reactions A situation were the algorithms we have used perform very poorly is when there is a reactant involved in nearly all reactions (hub in the reactant-reaction network). This means virtually every reaction will change the concentration of this reactant, and every reaction propensity will need to be updated, leading to poor performance (because structures underlying algorithms need to be completely rebuilt, as seen previously in the document). As propensities are of multiplicative nature, this is actually not necessary. Say all propensites in the system are of the form $[A] \times ...$, where $[A]$ is the concentration of the ubiquitous reactant. Then we could store $[A]$ on one side, and the remaining part of the propensites elsewhere. Updating would be easier, only $[A]$ would need to be updated. Drawing would only necessitate the second part of the propensity (renormalizing by $[A]$ does not change the drawing probabilities). This works even if not all reactions depend on A by pooling reactions that depend on a same reactant together. Pooling has to be done carefully to ensure the implementation remains efficient and statistically correct.

Implementation: parallel computing Using parallel computing can enhance performance. For example, in the case where updates are not a problem and propensities are updated constantly, generating random numbers takes up a large amount of time in the simulation (at least 25%). A first step would be using a node only for the purpose of computing and storing random numbers. Second, if updtating the system is a problem, this is a task that can be naturally parallelized. Concentration of reactants do not change during update, so there is a limited danger of data corruption by assigning updates to different nodes.

3.4 Implementation details

Implementation details fall into two categories. Optimization concerns ensure that implementations yield the claimed theoretical complexity. Numerical concerns address rounding problems that occur quite frequently in real-word situations.

When optimization is addressed, structures and algorithms are proposed that yield the asymptotic complexities presented in this document. Only rarely do we insist on optimizations that will only improve the multiplicative or additive constants of the algorithms. The reader is free to use its own adapted/further optimized structures.

Numerical issues are dangerous as algorithms ignoring them may lead to plausible, yet statistically flawed, results. They *must* be addressed at all costs. We propose simple ways to treat them. We do not doubt that more rigorous treatments exist.

3.4.1 Numerical concern: updating total propensity on the fly

We start with numerical issues, as all methods are impacted by rounding problems. For illustrative purposes, we focus on the computation of the total of a set of real values.

Context Suppose we have a vector v of n real values. The task is to maintain the total value $T := \sum_{1 \leq i \leq n} v[i]$. Computing T anew every time a value changes is too expensive. Suppose we want to set $v[i]$ to a new value new_v_i. Theoretically, the new value of T will be $T - v[i] + new\_v\_i$.

Numerical issue: absorption On a computer, such operations will *always* lead to rounding problems. In other words, we must be aware that T will always be different from the real total. Question is: how different? Here we need to get a little technical and talk about the problem of *absorption*. Rougly speaking, a 64 bit `double` has a precision of 15 to 17 digits, the rest can be ignored. Take some number a. The next `double` a machine can represent is roughly $a + 10^{-15}a$. If we add b such that $b < 10^{-16}a$, rounding leads to $a + b = a$: *a absorbs b*. Now imagine $b \simeq 10^{-n}a$ where $1 \leq n \leq 15$. b is no longer absorbed by a, but its original precision will be lost in operations such as $b + a - a$. The first operation treated is $b + a$, where only $\simeq 16 - n$ digits of b that fall in the meaningful region of a are really kept (Fig. 10). When a is removed, only these digits are restored: b is left with only $16 - n$ meaningful digits. To track absorption, we need to know how the *largest* number treated in our operations compares to the current total. Let M be that number and T the total introduced in the context paragraph. The number of meaningful digits of T is $\simeq 16 - \lceil \log_{10} M - \log_{10} T \rceil$.

Solutions One possible solution is to switch to higher precision. 128 bit `quadruple` offer 33 to 36 meaningful digits, spanning a larger range of magnitudes. But even with larger precision, it may be necessary to track the current precision of T. This can be achieved by recording the largest number M used to compute T and use $d = 16 - \lceil \log_{10} M - \log_{10} T \rceil$ (replace 16 with 34 for `quadruple`) as a proxy for the number of meaningful digits. Because of rounding problems affecting meaningful digits, this is

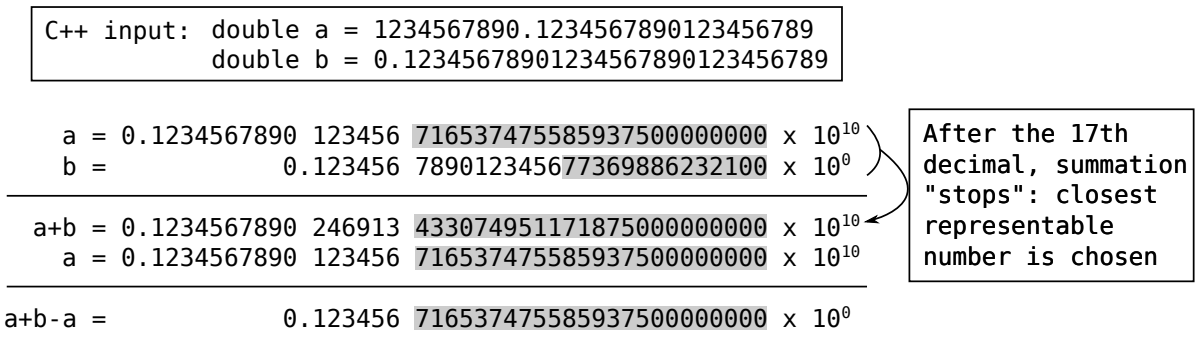


Figure 10: Illustration of a rounding problem due to absorption. Gray areas show the decimals the computer has no real control of. They are chosen according to closest representation available. When summing terms, the sum can only be exact up to 17 decimals at most, so a part of the decimals of the smaller term of the sum are ignored and cannot be recovered.

rather an optimistic estimate. When d falls below a predefined limit (somewhere between 5 and 10), T has to be recomputed from scratch. This operation can also be dangerous in various ways (if v has a lot of small values and a large value, compensation of large numbers that have opposite sign). In practice, these dangers do not naturally arise in our case, but they can be treated by sorting the numbers by magnitude and carefully considering the order in which they are added up.

3.4.2 Direct Method

The only concern with this method is to update the total propensity properly (Sec. 3.4.1).

3.4.3 Binary Tree

Updating intermediate nodes at most once This optimization is pretty straightforward. The tree is updated starting from the highest depth. We suppose the tree is built so that all leaves are at the same depth. We use an update queue that initially contains all nodes whose propensities are outdated.

We follow this simple rule: every time a node is updated, its parent is added to the update queue. Each node holds an `outdated` flag. If a node is marked `outdated`, we know it is already in the queue and we do not add it a second time. The flag is removed when the node is updated.

By applying this procedure, nodes are added layer by layer. As mentionned above, we start with leaves at the highest depth. As we progress through leaves, there parents are added at their succession and so on, until root is finally reached. By using the flag system, each node is queued and updated at most once.

Note that the update queue will at most contain all nodes of the tree, *i.e.* $\simeq 2N$ nodes. To avoid memory reallocation, it can (should) be represented by an array of size $2N$ and two pointers to the start and end of the queue.

This procedure can be adapted if the leaves are not all at the same depth.

Numerical concern: rounding problems Because of rounding problems in the summation, it is possible (but extremely rare) that a node carrying a zero rate will be selected. This must be avoided at all cost because a reaction that is theoretically impossible will be performed, yielding unknown results. Such an event must be avoided or at least simply tested. Because this happens extremely rarely we did not try to avoid but simply perform a test. If a zero leaf is selected we know the drawing was supposed to fall in the area and simply take the next nonzero leaf.

3.4.4 Hybrid Method

Numerical concerns Because the total propensities of groups are stored and drawn according to a classical multinomial drawing method, it must be made sure that the total is updated properly, as stated in 3.4.1.

Efficient inserting/removal in groups

Principle In order to achieve $O(1)$ complexity, base operations of the algorithm have to be performed with care. Most importantly, no loop should be allowed in the algorithm. For example, if propensities were stored in a list and we needed to remove a propensity at some position in the list, we would need to loop through the list to find it, which could lead to $O(N)$ worst-case complexity. These are the points were loops must be avoided:

- Update: when looking for a propensity of index i, its group and location within group must be instantly found.
- Uniform drawing within group: any propensity must be accessed instantly (propensities have to be stored in an array structure).

Example of implementation Figure 11 shows an example of a design achieving optimal complexity using tokens to represent propensities. The costliest step is to determine the new group where to place the token. A logarithmic-like function has to be used, but if groups are delimited by powers of 2, more efficient low level functions can be used, such as `frexp` in C.

It is easy to design a similar structure without actual tokens, but in a language like C++, all these classes can be inlined and do not actually exist in the compiled code. If the design is careful enough, it can be both human-readable by using classes representing clear abstractions AND efficient.

Choice of base rate This is the trickiest part of the algorithm. The choice can be left to the user, but it is also possible to imagine a structure that adapts dynamically. We did not have time to put much thought into it, but some ideas would be

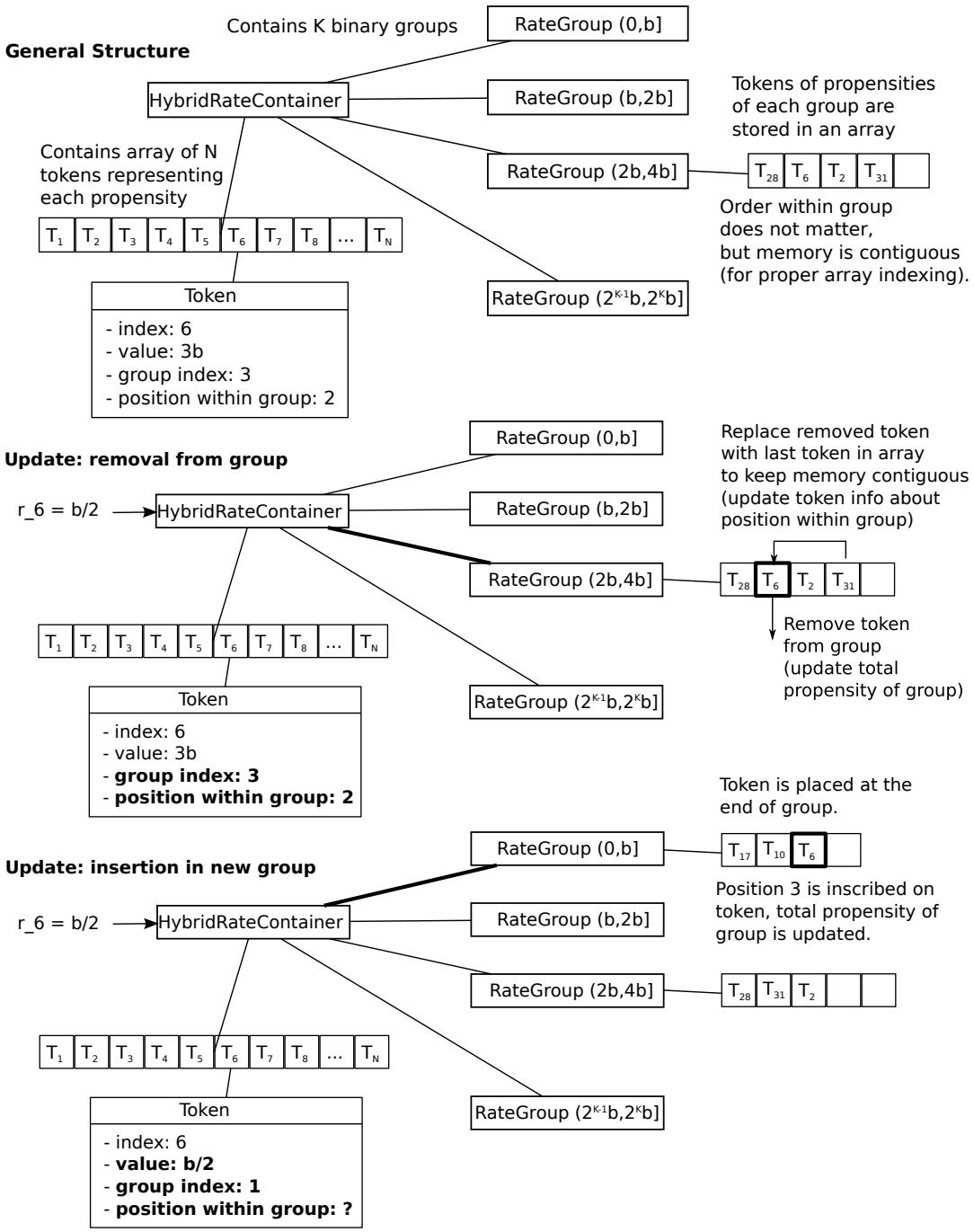


Figure 11: Example of structure warranting optimal complexity. Propensities are stored in their `RateGroup` as `Token`s containing additional information, enabling $O(1)$ operations.

- Create a base group at some arbirary rate. Each time a propensity (or too many propensities) falls into that group, subdivide the group into new groups, thus lowering the base group rate.

- Replace the base group with any container adapted to multinomial drawing, possible a hybrid drawing structure. If the value drawn falls into that group, the drawing method of the structure chosen is applied by reusing the same value to avoid additionnal random generator costs.

In both cases, the question is: at what point is it worth creating subgroups/substructures? This question has to be carefully analyzed before a choice of implementation is actually made.

References

Yang Cao, Daniel T. Gillespie, and Linda R. Petzold. Avoiding negative populations in explicit Poisson tau-leaping. *J Chem Phys*, 123(5):054104, 2005. URL http://scitation.aip.org/content/aip/journal/jcp/123/5/10.1063/1.1992473.

Daniel T. Gillespie, Andreas Hellander, and Linda R. Petzold. Perspective: Stochastic algorithms for chemical kinetics. *J Chem Phys*, 138 (17), May 2013. ISSN 0021-9606. doi: 10.1063/1.4801941. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3656953/.

Santiago A. Serebrinsky. Physical time scale in kinetic Monte Carlo simulations of continuous-time Markov chains. *PhysRev E*, 83(3), March 2011. ISSN 1539-3755, 1550-2376. doi: 10.1103/PhysRevE.83.037701. URL http://link.aps.org/doi/10.1103/PhysRevE.83.037701.

Alexander Slepoy, Aidan P. Thompson, and Steven J. Plimpton. A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks. *J Chem Phys*, 128(20):205101, 2008. ISSN 00219606. doi: 10.1063/1.2919546. URL http://scitation.aip.org/content/aip/journal/jcp/128/20/10.1063/1.2919546.