1

# Generalising simulation engines

**Concept.** To design and build a generalised simulation engine that is able to generate samples from practically any real-world stochastic processes that a researcher could encounter. With such a thing pre-built and self-contained, it can become the basis upon which to build generalised software solutions for a lot of different problems. For the mathematically-inclined, this chapter will require the introduction of a new formalism which we shall refer back to throughout the book. For the programmers, the software which is designed and described in this chapter can be found in the public Git respository here: https://github.com/umbralcalc/stochadex.

## 1.1 Computational formalism

Before diving into the design of software we need to mathematically define the general computational approach that we're going to take. Because the language of stochastic processes is primarily mathematics, we'd argue this step is essential in enabling a really general description. From experience, it seems reasonable to start by writing down the following formula which describes iterating some arbitrary process forward in time (by one finite step) and adding a new row each to some matrix $X_{0:\mathsf{t}} \rightarrow X_{0:\mathsf{t}+1}$

$$X^i_{\mathsf{t}+1} = F^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) \,, \tag{1.1}$$

where: $i$ is an index for the dimensions of the 'state' space; $\mathsf{t}$ is the current time index for either a discrete-time process or some discrete approximation to a continuous-time process; $X_{0:\mathsf{t}+1}$ is the next version of $X_{0:\mathsf{t}}$ after one timestep (and hence one new row has been added); $z$ is a vector of arbitrary size which contains the 'hidden' other parameters that are necessary to iterate the process; and $F^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t})$ as the latest element of an arbitrary matrix-valued function.

Throughout the book, the notation $A_{\mathsf{b}:\mathsf{c}}$ will always refer to a slice of rows from index $\mathsf{b}$ to $\mathsf{c}$ in a matrix (or row vector) $A$. As we shall discuss shortly, $F^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t})$ may represent not just operations on deterministic variables, but also on stochastic ones. There is also no requirement for the function to be continuous.

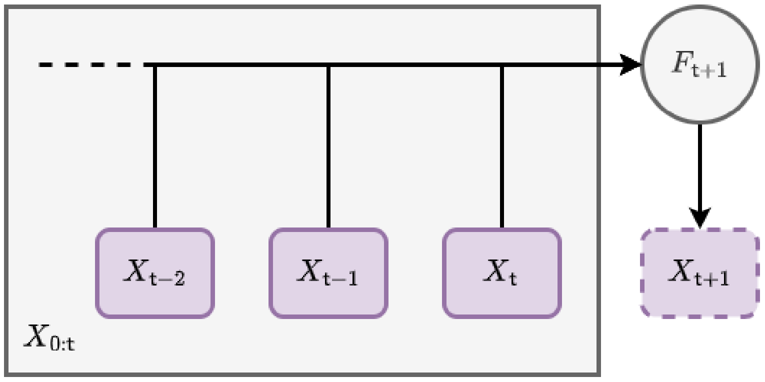

Figure 1.1: Graph representation of Eq. (1.1).

The basic computational idea here is illustrated in Fig. 1.1; we iterate the matrix $X$ forward in time by a row, and use its previous version $X_{0:t}$ as an entire matrix input into a function which populates the elements of its latest rows. In pseudocode you could easily write something with the same idea in it, and it would probably look something like the method diagram in Fig. 1.2.

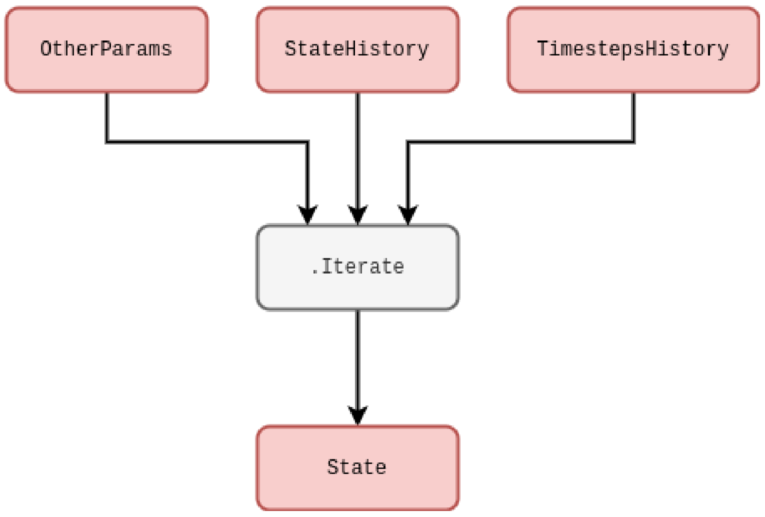

Figure 1.2: Pseudocode representation of Eq. (1.1).

Pretty simple! But why go to all this trouble of storing matrix inputs for previous values of the same process? It's true that this is mostly redundant for *Markovian* phenomena, i.e., processes where their only memory of their history is the most recent value they took. However, for a large class of stochastic processes a full memory[1] of past values is essential to consistently construct the sample paths moving forward. This is true in particular for *non-Markovian* phenomena, where the

[1] Or memory at least within some window.

latest values don't just depend on the immediately previous ones but can depend on values which occured much earlier in the process as well.

For more complex physical models and integrators, the distinct notions of 'numerical timestep' and 'total elapsed continuous time' will crop up quite frequently. Hence, before moving on further details, it will be important to define the total elapsed time variable $t(\mathsf{t})$ for processes which are defined in continuous time. Assuming that we have already defined some function $\delta t(\mathsf{t})$ which returns the specific change in continuous time that corresponds to the step $\mathsf{t}-1 \rightarrow \mathsf{t}$, we will always be able to compute the total elapsed time through the relation

$$t(\mathsf{t}) = \sum_{\mathsf{t}'=0}^{\mathsf{t}} \delta t(\mathsf{t}') \,. \tag{1.2}$$

It's important to remember that our steps in continuous time may not be constant, so by defining the $\delta t(\mathsf{t})$ function and summing over it we can enable this flexibility in the computation. In case the summation notation is no fun for programmers; we're simply adding up all of the differences in time to get a total.

## 1.2 Example phenomena

So, now that we've mathematically defined a really general notion of iterating the stochastic process forward in time, it makes sense to discuss some simple examples. For instance, it is frequently possible to split $F$ up into deteministic (denoted $D$) and stochastic (denoted $S$) matrix-valued functions like so

$$F^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = D^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) + S^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) \,. \tag{1.3}$$

In the case of stochastic processes with continuous sample paths, it's also nearly always the case with mathematical models of real-world systems that the deterministic part will at least contain the term $D^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = X^i_{\mathsf{t}}$ because the overall system is described by some stochastic differential equation. This is not a really requirement in our general formalism, however.

What about the stochastic term? For example, if we wanted to consider a *Wiener process noise*, we can define $W^i_{\mathsf{t}}$ is a sample from a Wiener process for each of the state dimensions indexed by $i$ and our formalism becomes

$$S^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = W^i_{\mathsf{t}+1} - W^i_{\mathsf{t}} \,. \tag{1.4}$$

One draws the increments $W^i_{\mathsf{t}+1} - W^i_{\mathsf{t}}$ from a normal distribution with a mean of 0 and a variance equal to the length of continuous time that the step corresponded to $\delta t(\mathsf{t}+1)$, i.e., the probability density $P_{\mathsf{t}+1}(x^i)$ of the increments $x^i = W^i_{\mathsf{t}+1} - W^i_{\mathsf{t}}$ is

$$P_{\mathsf{t}+1}(x^i) = \mathsf{NormalPDF}[x^i; 0, \delta t(\mathsf{t}+1)] \,. \tag{1.5}$$

Note that for state spaces with dimensions $> 1$, we could also allow for non-trivial cross-correlations between the noises in each dimension. In pseudocode, the Wiener process is schematically represented by Fig. 1.3.

In another example, to model *geometric Brownian motion noise* we would simply have to multiply $X^i_{\mathsf{t}}$ to the Wiener process like so

$$S^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = X^i_{\mathsf{t}}(W^i_{\mathsf{t}+1} - W^i_{\mathsf{t}}) \,. \tag{1.6}$$

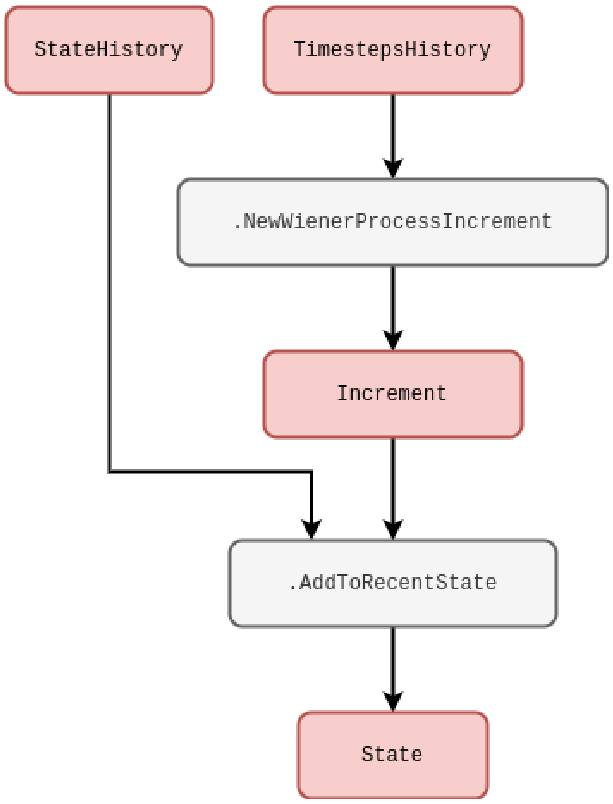

Figure 1.3: Schematic of code for a Wiener process.

Here we have implicitly adopted the Itô interpretation to describe this stochastic integration. Given a carefully-defined integration scheme other interpretations of the noise would also be possible with our formalism too, e.g., Stratonovich[2] or others within the more general '$\alpha$-family' [1, 2, 3]. The pseudocode for any of these should hoepfully be fairly straightforward to deduce based on the lines we've already written above.

We can imagine even more general processes that are still Markovian. One example of these in a single-dimension state space would be to define the noise through some general function of the Wiener process like so

$$S^0_{\mathtt{t}+1}(X_{0:\mathtt{t}}, z, \mathtt{t}) = g[W^0_{\mathtt{t}+1}, t(\mathtt{t}+1)] - g[W^0_{\mathtt{t}}, t(\mathtt{t})] \tag{1.7}$$

$$= \left[\frac{\partial g}{\partial t} + \frac{1}{2}\frac{\partial^2 g}{\partial x^2}\right]\delta t(\mathtt{t}+1) + \frac{\partial g}{\partial x}(W^0_{\mathtt{t}+1} - W^0_{\mathtt{t}}), \tag{1.8}$$

where $g(x,t)$ is some continuous function of its arguments which has been expanded out with Itô's Lemma on the second line. Note also that the computations in Eq. (1.8) could be performed with numerical derivatives in principle, even if the function were extremely complicated. This is unlikely

[2]Which would implictly give $S^i_{\mathtt{t}+1}(X_{0:\mathtt{t}}, z, \mathtt{t}) = (X^i_{\mathtt{t}+1} + X^i_{\mathtt{t}})(W^i_{\mathtt{t}+1} - W^i_{\mathtt{t}})/2$ for Eq. (1.6).

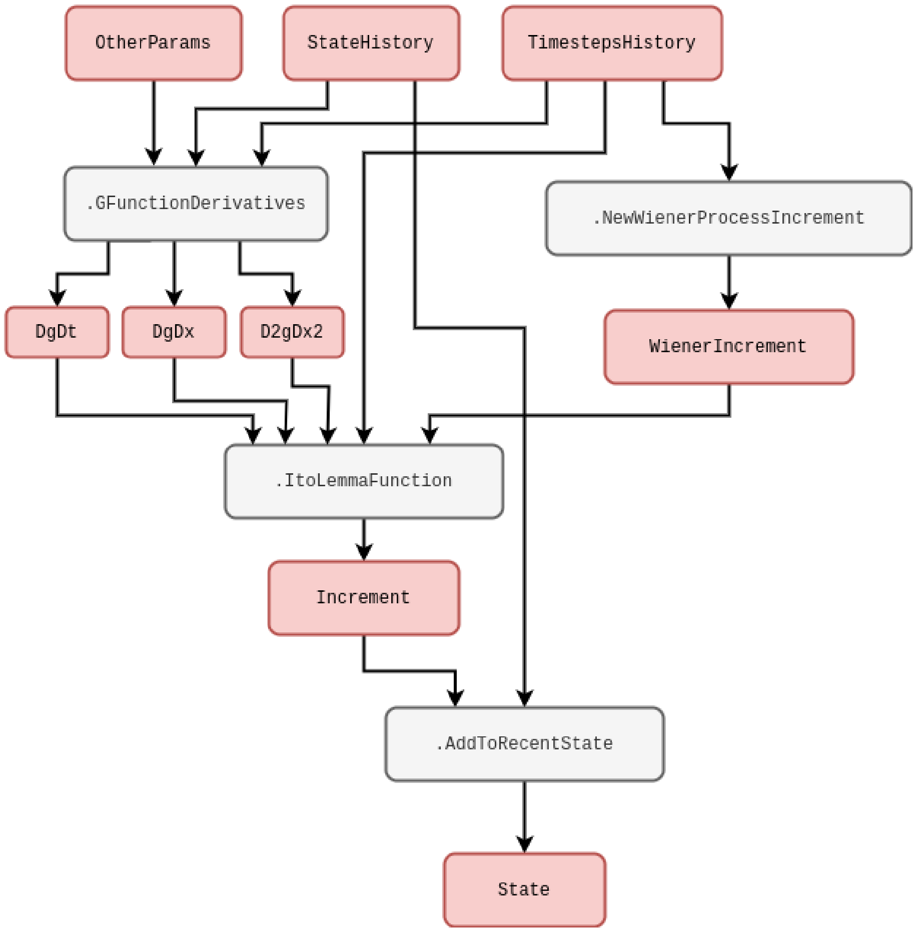

Figure 1.4: Schematic of code for Eq. (1.8).

to be the best way to describe the process of interest, however, the mathematical expressions above can still be made a bit more meaningful to the programmer in this way. The pseudocode in general would look something like Fig. 1.4.

Let's now look at a more complicated type of noise. For example, we might consider sampling from a *fractional Brownian motion* process $[B_H]_{\mathsf{t}}$, where $H$ is known as the 'Hurst exponent'. Following Ref. [4], we can simulate this process in one of our state space dimensions by modifying the standard Wiener process by a fairly complicated integral factor which looks like this

$$S^0_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = \frac{(W^0_{\mathsf{t}+1} - W^0_{\mathsf{t}})}{\delta t(\mathsf{t})} \int_{t(\mathsf{t})}^{t(\mathsf{t}+1)} \mathrm{d}t' \frac{(t'-t)^{H-\frac{1}{2}}}{\Gamma(H+\frac{1}{2})} {}_2F_1\left(H-\frac{1}{2}; \frac{1}{2} - H; H + \frac{1}{2}; 1 - \frac{t'}{t}\right), \quad (1.9)$$

where $S^0_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = [B_H]_{\mathsf{t}+1} - [B_H]_{\mathsf{t}}$. The integral in Eq. (1.9) can be approximated using an appropriate numerical procedure (like the trapezium rule, for instance). In the expression above, we have used the symbols ${}_2F_1$ and $\Gamma$ to denote the ordinary hypergeometric and gamma functions, respectively. A computational form of this integral is illustrated in Fig. 1.5 to try and disentangle some of the mathematics as a program.

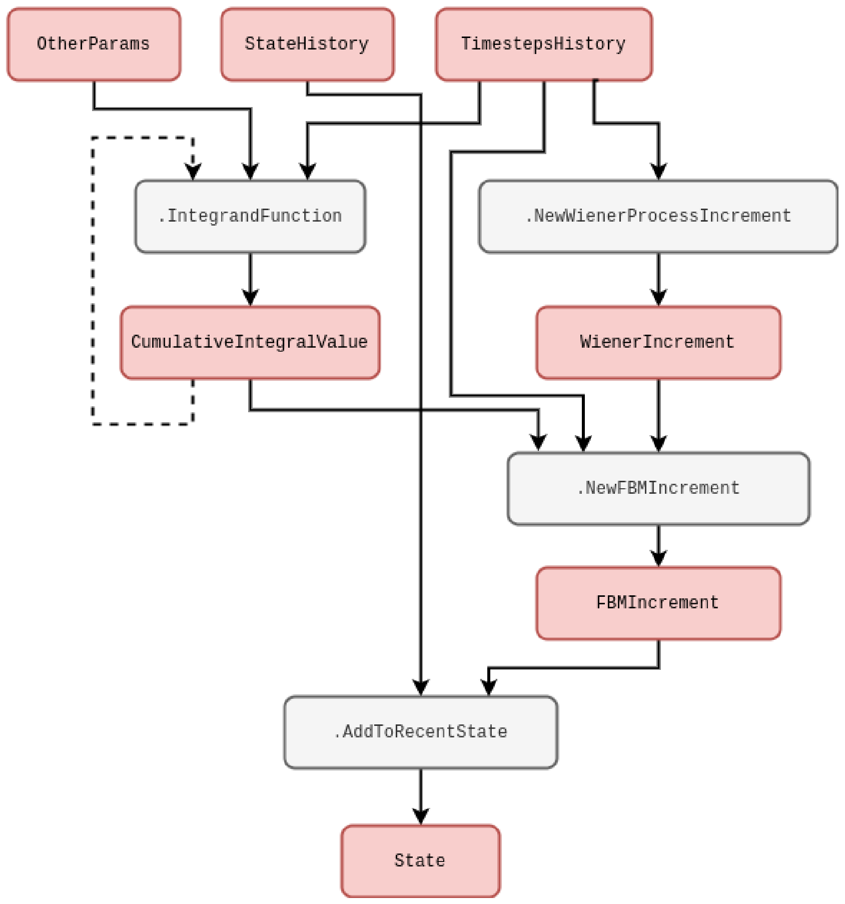

Figure 1.5: Schematic of code for Eq. (1.9).

So far we have mostly been discussing noises with continuous sample paths, but we can easily adapt our computation to discontinuous sample paths as well. For instance, *Poisson process noises* would generally take the form

$$S^i_{\mathtt{t}+1}(X_{0:\mathtt{t}}, z, \mathtt{t}) = [N_\lambda]^i_{\mathtt{t}+1} - [N_\lambda]^i_{\mathtt{t}}\,, \tag{1.10}$$

where $[N_\lambda]^i_{\mathtt{t}}$ is a sample from a Poisson process with rate $\lambda$. One can think of this process as counting the number of events which have occured up to the given interval of time, where the intervals between each succesive event are exponentially distributed with mean $1/\lambda$. Such a simple counting process could be simulated exactly by explicitly setting a newly-drawn exponential variate to the next continuous time jump $\delta t(\mathtt{t}+1)$ and iterating the counter. Other exact methods exist to handle more complicated processes involving more than one type of 'event', such as the Gillespie algorithm [5] — though these techniques are not always be applicable in every situation.

Is using step size variation always possible? If we consider a *time-inhomogeneous Poisson process noise*, which would generally take the form

$$S^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = [N_{\lambda(\mathsf{t}+1)}]^i_{\mathsf{t}+1} - [N_{\lambda(\mathsf{t})}]^i_{\mathsf{t}}\,, \tag{1.11}$$

the rate $\lambda(\mathsf{t})$ has become a deterministically-varying function in time. In this instance, it likely not be accurate to simulate this process by drawing exponential intervals with a mean of $1/\lambda(\mathsf{t})$ because this mean could have changed by the end of the interval which was drawn. An alternative approach (which is more generally capable of simulating jump processes but is an approximation) first uses a small time interval $\tau$ such that the most likely thing to happen in this period is nothing, and then the probability of the event occuring is simply given by

$$p(\mathsf{event}) = \frac{\lambda(\mathsf{t})}{\lambda(\mathsf{t}) + \frac{1}{\tau}}\,. \tag{1.12}$$

This idea can be applied to phenomena with an arbitrary number of events and works well as a generalised approach to event-based simulation, though its main limitation is worth remembering; in order to make the approximation good, $\tau$ often must be quite small and hence our simulator must churn through a lot of steps. From now on we'll refer to this well-known technique as the *rejection method*. Fig. 1.6 may also help to understand this concept from the programmer's perspective.

There are a few extensions to the simple Poisson process that introduce additional stochastic processes. *Cox (doubly-stochastic) processes*, for instance, are basically where we replace the time-dependent rate $\lambda(\mathsf{t})$ with independent samples from some other stochastic process $\Lambda(\mathsf{t})$. For example, a Neyman-Scott process [6] can be mapped as a special case of this because it uses a Poisson process on top of another Poisson process to create maps of spatially-distributed points. In our formalism, a two-state implementation of the Cox process noise would look like

$$S^0_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = \Lambda(\mathsf{t}+1) \tag{1.13}$$

$$S^1_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = [N_{S^0_{\mathsf{t}+1}}]^i_{\mathsf{t}+1} - [N_{S^0_{\mathsf{t}}}]^i_{\mathsf{t}}\,. \tag{1.14}$$

This process could be simulated using the pseudocode we wrote for the time-inhomogeneous Poisson process previously — where we would just replace `EventRateLambdaFunction` with a method that generates the stochastic rate $\Lambda(\mathsf{t})$.

Another extension is *compound Poisson process noise*, where it's the count values $[N_\lambda]^i_{\mathsf{t}}$ which are replaced by independent samples $[J_\lambda]^i_{\mathsf{t}}$ from another probability distribution, i.e.,

$$S^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = [J_\lambda]^i_{\mathsf{t}+1} - [J_\lambda]^i_{\mathsf{t}}\,. \tag{1.15}$$

Note that the rejection method of Eq. (1.12) can be employed effectively to simulate any of these extensions as long as a sufficiently small $\tau$ is chosen. Once again, the pseudocode we wrote previously would be sufficient to simulate this process with one tweak: add into the `DrawNewEventIncrement` method the calling of a function which generates the $[J_\lambda]^i_{\mathsf{t}}$ samples and output these if the event occurs.

All of the examples we have discussed so far are Markovian. Given that we have explicitly constructed the formalism to handle non-Markovian phenomena as well, it would be worthwhile going some examples of this kind of process too. *Self-exciting process noises* would generally take the form

$$S^0_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = \mathcal{I}_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) \tag{1.16}$$

$$S^1_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = [N_{S^0_{\mathsf{t}+1}}]^i_{\mathsf{t}+1} - [N_{S^0_{\mathsf{t}}}]^i_{\mathsf{t}}\,, \tag{1.17}$$

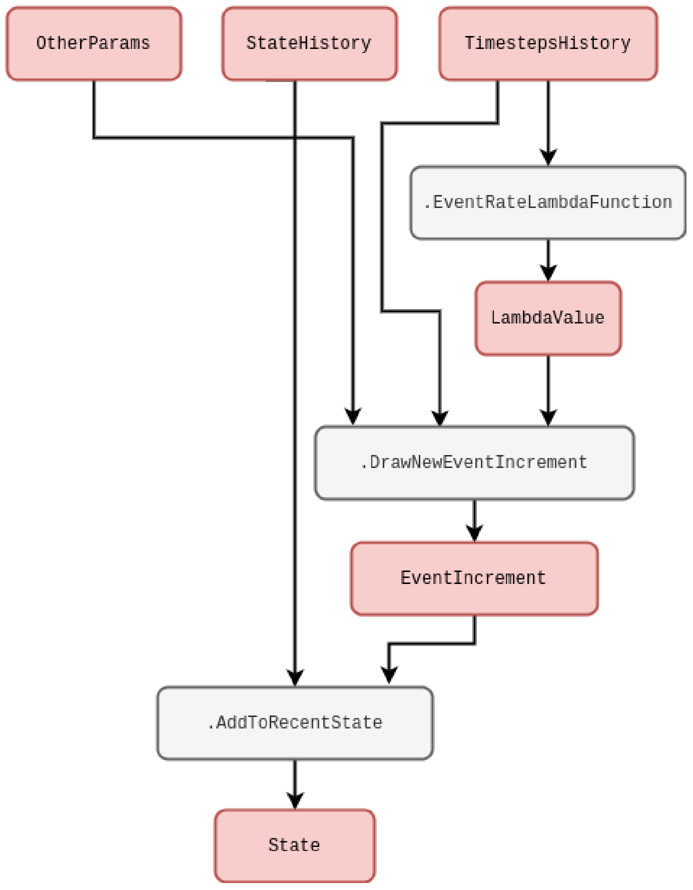

Figure 1.6: Schematic of code for an inhomogeneous Poisson process.

where the stochastic rate $\mathcal{I}_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t})$ now depends on the history explicitly. Amongst other potential inputs we can see, e.g., Hawkes processes [7] as an example of above by substituting

$$\mathcal{I}_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = \mu + \sum_{\mathsf{t}'=0}^{\mathsf{t}} \phi[t(\mathsf{t}) - t(\mathsf{t}')] S^1_{\mathsf{t}'} \,, \tag{1.18}$$

where $\phi$ is the 'exciting kernel' and $\mu$ is some constant background rate. In order to simulate a Hawkes process using our formalism, the pseudocode would be something like Fig. 1.7.

Note that this idea of integration kernels could also be applied back to our Wiener process. For example, another type of non-Markovian phenomenon that frequently arises across physical and life systems integrates the Wiener process history like so

$$S^0_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = W^0_{\mathsf{t}+1} - W^0_{\mathsf{t}} \tag{1.19}$$

$$S^1_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}) = u \sum_{\mathsf{t}'=0}^{\mathsf{t}} e^{-u[t(\mathsf{t}) - t(\mathsf{t}')]} S^0_{\mathsf{t}'} \,, \tag{1.20}$$

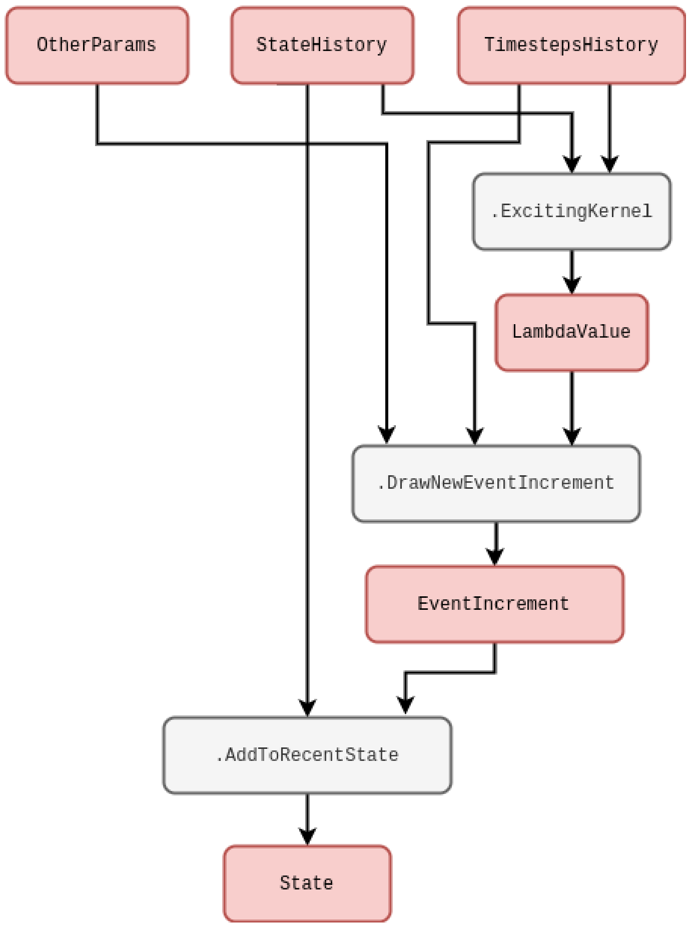

Figure 1.7: Schematic of code for a Hawkes process.

where $u$ is inversely proportional to the length of memory in continuous time.

## 1.3 Software design

So we've proposed a computational formalism and then studied it in more detail to demonstrate that it can cope with a variety of different stochastic phenomena. Now we're ready to summarise what we want the stochadex software package to be able to do. But what's so complicated about Eq. (1.1)? Can't we just implement an iterative algorithm with a single function? It's true that the fundamental concept is very straightforward, but as we'll discuss in due course; the stochadex needs to have a lot of configurable features so that it's applicable in different situations. Ideally, the stochadex sampler should be designed to try and maintain a balance between performance and flexibility of utilisation.

If we begin with the obvious first set of criteria; we want to be able to freely configure the iteration function $F$ of Eq. (1.1) and the timestep function $t$ of Eq. (1.2) so that any process we want can be described. The point at which a simulation stops can also depend on some algorithm termination condition which the user should be able to specify up-front.

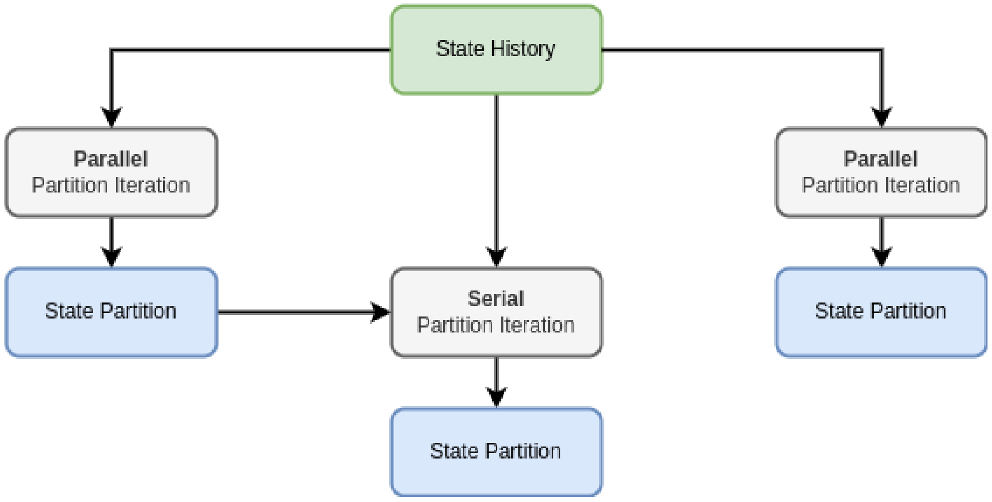

Figure 1.8: A schematic illustrating the difference between parallel and serial partitions in for a single step of the simulation.

Once the user has written the code to create these functions for the stochadex, we want to then be able to recall them in future only with configuration files while maintaining the possibility of changing their simulation run parameters. This flexibility should facilitate our uses for the simulation later in the book, and from this perspective it also makes sense that the parameters should include the random seed and initial state value.

The state history matrix $X$ should be configurable in terms of its number of rows — what we'll call the 'state width' — and its number of columns — what we'll call the 'state history depth'. If we were to keep increasing the state width up to millions of elements or more, it's likely that on most machines the algorithm performance would grind to a halt when trying to iterate over the resulting $X$ within a single thread. Hence, before the algorithm or its performance in any more detail, we can pre-empt the requirement that $X$ should represented in computer memory by a set of partitioned matrices which are all capable of communicating to one-another downstream. In this paradigm, we'd like the user to be able to configure which state partitions are able to communicate with each other without having to write any new code.

Within each parallel partition of the state history, the stochadex also enables further partitioning of state (and hence its corresponding update function) into serial blocks in memory. This enables the user to define much simpler (and hence more resuable) iteration functions to use in configuring future projects. During a simulation step, the key difference between parallel partitions and the serial partitions within each is that the former can only have shared access to the entire state history up to the last state values for the whole simulation. The latter, however, can also have access to the state values produced most recently by any partitions before it in the serial configuration. In Fig. 1.8 we have illustrated this difference between parallel and serial partitions in a simple schematic.

For convenience, it seems sensible to also make the outputs from stochadex runs configurable.

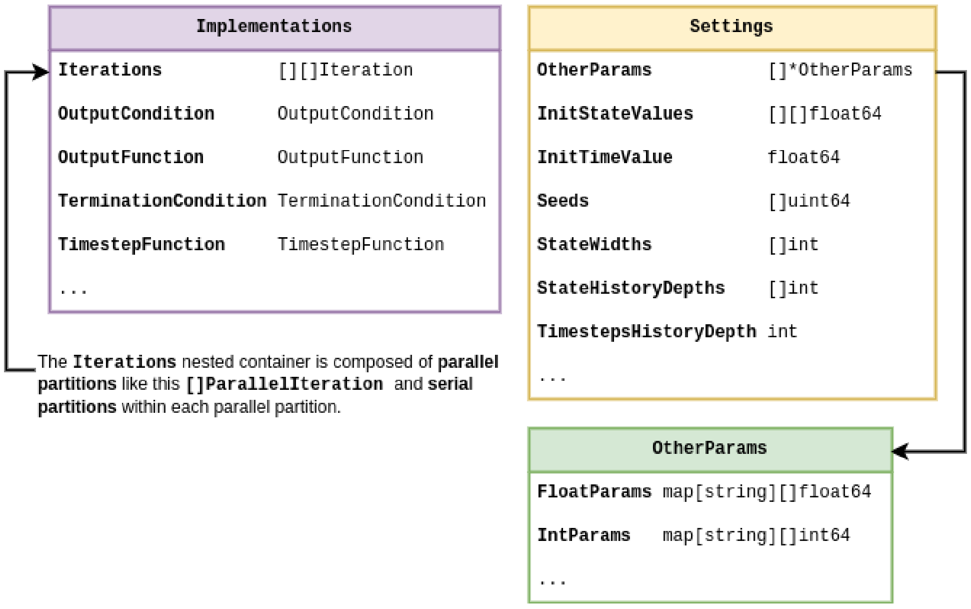
Implementations
Iterations [][]Iteration
OutputCondition OutputCondition
OutputFunction OutputFunction
TerminationCondition TerminationCondition
TimestepFunction TimestepFunction
...
The Iterations nested container is composed of parallel partitions like this []ParallelIteration and serial partitions within each parallel partition.
Settings
OtherParams []*OtherParams
InitStateValues [][]float64
InitTimeValue float64
Seeds []uint64
StateWidths []int
StateHistoryDepths []int
TimestepsHistoryDepth int
...
OtherParams
FloatParams map[string][]float64
IntParams map[string][]int64
...

Figure 1.9: A relational summary of the configuration data types in the stochadex.

A user should be able to change the form of output that they want through, e.g., some specified function of $X$ at the time of outputting data. The times that the stochadex should output this data can also be decided by some user-specified condition so that the frequency of output is fully configurable as well. This flexibility can be useful when the user only requires a limited number of state snapshots at particular times.

In summary, we've put together a schematic of configuration data types and their relationships in Fig. 1.9. In this diagram there is some indication of the data type that we propose to store each piece information in (in Go syntax), and the diagram as a whole should serve as a useful guide to the basic structure of configuration files for the stochadex.

It's clear that in order to simulate Eq. (1.1), we need an interative algorithm which reapplies a user-specified function to the continually-updated history. But let's now return to the point we made earlier about how the performance of such an algorithm will depend on the size of the state history matrix $X$. The key bit of the algorithm design that isn't so straightforward is: how do we sucessfully split this state history up into separate partitions in memory while still enabling them to communicate effectively with each other? Other generalised simulation frameworks — such as SimPy [8], StoSpa [9] and FLAME GPU [10] — have all approached this problem in different ways, and with different software architectures.

In Fig. 1.10 we've illustrated what a loop involving separate state partitions looks like in the stochadex simulator. Each partition is handled by concurrently running execution threads of the same process, while a separate process may be used to handle the outputs from the algorithm. As the diagram shows, the main sequence of each loop iteration follows the pattern:

1. The `PartitionCoordinator` requests more iterations from each state partition by sending an `IteratorInputMessage` to a concurrently running goroutine.

2. The `StateIterator` in each goroutine executes the iteration and stores the resulting state update in a variable.
3. Once all of the iterations have been completed, the `PartitionCoordinator` then requests each goroutine to update its relevant partition of the state history by sending another `IteratorInputMessage` to each.

This pattern ensures that no partition has access to values in the state history which are out of sync with its current state in time, and hence prevents anachronisms from occuring in the overall simulation state iteration.

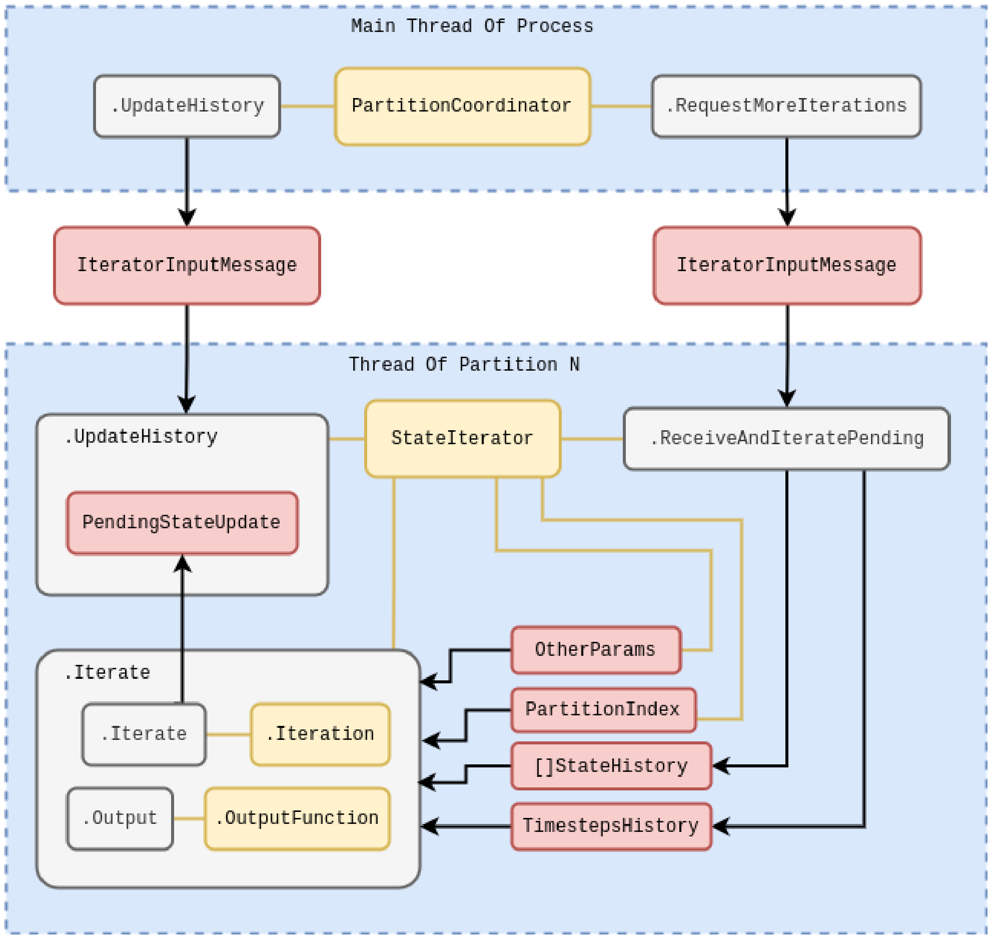

Figure 1.10: Schematic for a step of the stochadex simulation algorithm.

It's also worth noting that while Fig. 1.10 illustrates only a single process; it's obviously true that we may run many of these whole diagrams at once to parallelise generating independent realisations of the simulation, if necessary.

As we stated at the beginning of this chapter: the full implementation of the stochadex can be found on GitHub by following this link: https://github.com/umbralcalc/stochadex. Users can build the main binary executable of this repository and determine what configuration of the stochadex they would like to have through config at runtime (one can infer these configurations from Fig. 1.9). As Go is a statically typed language, this level of flexibility has been achieved using code templating and generation proceeding runtime build and execution via `go run` 'under-the-hood'. Users who find this particular execution pattern undesirable can also use all of the stochadex types, tools and methods as part of a standard Go package import.

In order to debug the simulation code and gain a more intuitive understanding of the outputs from a model as it is being developed, we have also written a lightweight frontend dashboard React [11] app in TypeScript to visualise any stochadex simulation as it is running. This dashboard can be launched by passing config at runtime to the main stochadex executable, and we have illustrated how all this fits together in a flowchart shown in Fig. 1.11.

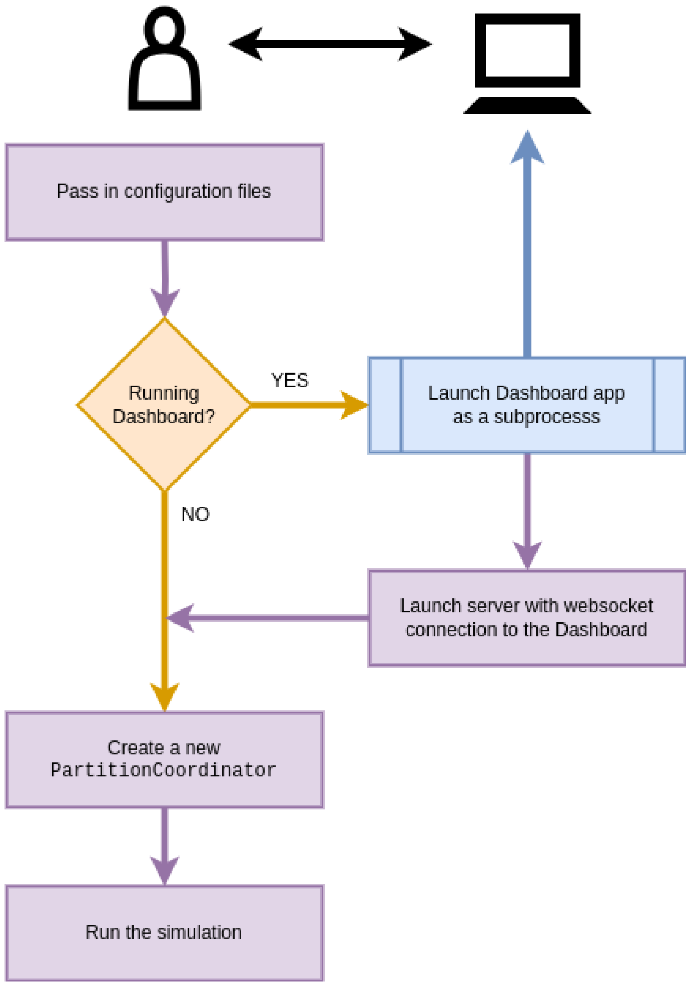
Pass in configuration files
Running Dashboard?
YES
NO
Launch Dashboard app as a subprocesss
Launch server with websocket connection to the Dashboard
Create a new PartitionCoordinator
Run the simulation

Figure 1.11: A diagram of the main stochadex binary executable.

# Bibliography

[1] N. G. Van Kampen, *Stochastic processes in physics and chemistry.* Elsevier, 1992, vol. 1.

[2] H. Risken, "Fokker-planck equation," in *The Fokker-Planck Equation.* Springer, 1996, pp. 63–95.

[3] L. Rogers and D. Williams, *Diffusions, Markov Processes and Martingales 2: Ito Calculus.* Cambridge University Press, 04 2000, vol. 1, pp. xiv+480.

[4] L. Decreusefond *et al.*, "Stochastic analysis of the fractional brownian motion," *Potential analysis*, vol. 10, no. 2, pp. 177–214, 1999.

[5] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *The journal of physical chemistry*, vol. 81, no. 25, pp. 2340–2361, 1977.

[6] J. Neyman and E. L. Scott, "Statistical approach to problems of cosmology," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 20, no. 1, pp. 1–29, 1958.

[7] A. G. Hawkes, "Spectra of some self-exciting and mutually exciting point processes," *Biometrika*, vol. 58, no. 1, pp. 83–90, 1971.

[8] "SimPy: a process-based discrete-event simulation framework," https://gitlab.com/team-simpy/simpy/, accessed: 2023-02-13.

[9] "StoSpa: A C++ package for running stochastic simulations to generate sample paths for reaction-diffusion master equation," https://github.com/BartoszBartmanski/StoSpa, accessed: 2023-02-13.

[10] "FLAME GPU: A GPU accelerated agent-based simulation library for domain independent complex systems simulation," https://github.com/FLAMEGPU/FLAMEGPU2/, accessed: 2023-02-13.

[11] "The React Library," https://react.dev/, accessed: 2023-08-17.