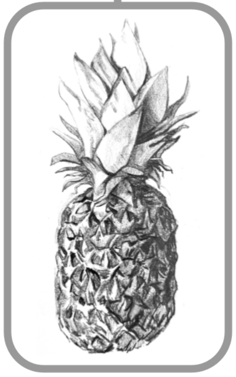
Part 1



1

Building a generalised simulator

Concept. To design and build a generalised simulation engine that is able to generate samples from a ‘Pokédex’ of possible stochastic processes that a researcher might encounter. A ‘Pokédex’ here is just a fanciful description for a very general class of multidimensional stochastic processes that pop up everywhere in taming the mathematical wilds of real-world phenomena, and which also leads to a name for the software itself: the ‘stochadex’. 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 public Git repository for the code that is described in this chapter can be found here: https://github.com/umbralcalc/stochadex.

1.1 Computational formalism

Before we dive into software design of the stochadex, 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 $X' \rightarrow X$

$$X^i_{\mathsf{t}+1} = F^i_{\mathsf{t}+1}(X', \mathsf{t})\,, \tag{1.1}$$

where: i is an index for the dimensions of the ‘state’ space; t is the current time index for either a discrete-time process or some discrete approximation to a continuous-time process; X is the next version of X' after one timestep (and hence one new row has been added); and $F^i_{\mathsf{t}+1}(X', \mathsf{t})$ as the latest element of an arbitrary matrix-valued function. As we shall discuss shortly, $F^i_{\mathsf{t}+1}(X', \mathsf{t})$ may represent not just operations on deterministic variables, but also on stochastic ones. There is also no requirement for the function to be continuous.

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' as an entire matrix input into a function which

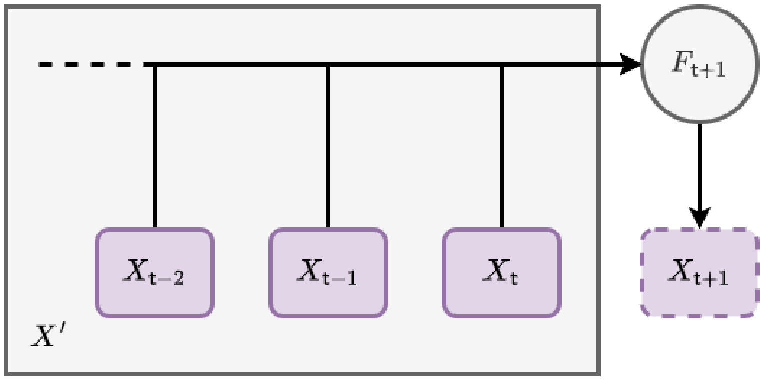


Figure 1.1: Graph representation of Eq. (1.1).

populates the elements of its latest rows. In Go you could easily write something with the same idea in it, and the code would probably look like this.

```go
type StateVector  []float64
type StateHistory []StateVector

// iterate the state history forward in time by one step
func IterationFormula(
    stateHistory StateHistory,
    timeStepNumber int,
) StateVector {
    for t, stateVector := range stateHistory {
        for i, stateElement := range stateVector {
            // do something
        }
    }
    return newestStateVector
}
```

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 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 $t(\mathtt{t})$ for processes which are defined in continuous time. Assuming that we have already defined some function $\delta t(\mathtt{t})$ which returns the specific change in continuous time that corresponds to the step $\mathtt{t}-1 \rightarrow \mathtt{t}$, we will always

[1] Or memory at least within some window.

be able to compute the total elapsed time through the relation

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

This seems a lot of effort, no? Well 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; in Go we're implicitly doing this.

```go
// get the next increment from this step number forward in
// some unit of time, e.g., seconds
func TimeIncrementFunction(timeStepNumber int) float64 {
    // compute the next increment
    return nextTimeIncrement
}

// compute the total time elapsed up to the input step number in
// some unit of time, e.g., seconds
func ElapsedTimeFunction(timeStepNumber int) float64 {
    totalElapsedTime := 0.0
    for t := 0; t < timeStepNumber; t++ {
        totalElapsedTime += TimeIncrementFunction(t)
    }
    return totalElapsedTime
}
```

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

$$F^i_{\mathsf{t}+1}(X', \mathsf{t}) = D^i_{\mathsf{t}+1}(X', \mathsf{t}) + S^i_{\mathsf{t}+1}(X', \mathsf{t}) \, . \tag{1.3}$$

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 $D^i_{\mathsf{t}+1}(X', \mathsf{t}) = X^i_{\mathsf{t}}$ 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_{t} is a sample from a Wiener process for each of the state dimensions indexed by i and our formalism becomes

$$S^i_{\mathsf{t}+1}(X', \mathsf{t}) = W^i_{\mathsf{t}+1} - W^i_{\mathsf{t}} \, . \tag{1.4}$$

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 $P^i_{\mathsf{t}+1}(x)$ of the increments $x^i = W^i_{\mathsf{t}+1} - W^i_{\mathsf{t}}$ is

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

Note that for state spaces with dimensions > 1, we could also allow for non-trivial cross-correlations between the noises in each dimension. In Go, the Wiener process can be represented within our formalism like this.

```go
import (
    "math"
    "math/rand"
)

// generate a new Wiener process increment for a state element
func NewWienerProcessIncrement(timeStepNumber int) float64 {
    // use the time increment function we defined earlier
    timeIncrement := TimeIncrementFunction(timeStepNumber)
    // multiply by the square-root here as
    // it is proportional to the standard deviation
    value := math.Sqrt(timeIncrement) * rand.NormFloat64()
    return value
}

// returns the state vector from the S(X',t) function we defined
// for the Wiener Process in the main text above
func SFunctionWienerProcess(
    stateHistory StateHistory,
    timeStepNumber int,
) StateVector {
    // we don't care about the state history so the noise = Markovian
    sFunctionValue := make(StateVector, 0)
    for _ = range stateVector {
        increment := NewWienerProcessIncrement(timeStepNumber)
        sFunctionValue = append(sFunctionValue, increment)
    }
    return sFunctionValue
}
```

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

$$S^i_{\mathsf{t}+1}(X',\mathsf{t}) = X^i_{\mathsf{t}}(W^i_{\mathsf{t}+1} - W^i_{\mathsf{t}})\,. \tag{1.6}$$

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 'α-family' [1, 2, 3]. The Go code 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

$$S^0_{\mathsf{t}+1}(X',\mathsf{t}) = g[W^0_{\mathsf{t}+1}, t(\mathsf{t}+1)] - g[W^0_{\mathsf{t}}, t(\mathsf{t})] \tag{1.7}$$

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

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 to be the best way to describe the process of interest, however, the mathematical expressions above

[2]Which would implictly give $S^i_{\mathsf{t}+1}(X',\mathsf{t}) = (X^i_{\mathsf{t}+1} + X^i_{\mathsf{t}})(W^i_{\mathsf{t}+1} - W^i_{\mathsf{t}})/2$ for Eq. (1.6).

can still be made a bit more meaningful to the programmer in this way. The code would probably look something like this.

```go
// some function
func G(wienerProcessSample float64, timeStepNumber int) float64 {
    // return something
}

// discretely represents the dg/dt expression in the equation above
func DgDt(
    newWienerProcessSample float64,
    previousWienerProcessSample float64,
    timeStepNumber int,
) float64 {
    return (G(newWienerProcessSample, timeStepNumber) -
        G(previousWienerProcessSample, timeStepNumber)) /
        TimeIncrementFunction(timeStepNumber)
}

// discretely represents the dg/dx expression in the equation above
func DgDx(
    newWienerProcessSample float64,
    previousWienerProcessSample float64,
    timeStepNumber int,
) float64 {
    return (G(newWienerProcessSample, timeStepNumber) -
        G(previousWienerProcessSample, timeStepNumber)) /
        (newWienerProcessSample - previousWienerProcessSample)
}

// discretely represents the d^2g/dx^2 expression in the equation above
func D2gDx2(
    // newDgDx and previousDgDx could be passed in here
    newWienerProcessSample float64,
    previousWienerProcessSample float64,
    timeStepNumber int,
) float64 {
    // newDgDx and previousDgDx are the result of applying the function
    // for dg/dx defined above on two different timesteps
    return (newDgDx - previousDgDx) /
        (newWienerProcessSample - previousWienerProcessSample)
}
```

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

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

where $S^0_{\mathsf{t}+1}(X',\mathsf{t}) = [B_H]_{\mathsf{t}+1} - [B_H]_{\mathsf{t}}$. 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 Γ to denote the ordinary hypergeometric and gamma functions, respectively. A discretised form of this integral is written below in Go to try and disentangle some of the mathematics as a program.

```go
import "scientificgo.org/special"

// computes the integral term in the fractional Brownian motion process
// defined above using a scientific library
func FractionalBrownianMotionIntegral(
  currentTime float64,
  nextTime float64,
  hurstExponent float64,
  numberOfIntegrationSteps int,
) float64 {
  integralStepSize := (nextTime - currentTime)/float64(numberOfIntegrationSteps)
  a := []float64{hurstExponent - 0.5, 0.5 - hurstExponent}
  b := []float64{hurstExponent + 0.5}
  integralValue := 0.0
  // implements the trapezium rule in a loop over the steps
  // between the current and the next point in time
  for t := 0; t < numberOfIntegrationSteps; t++ {
    t1 := currentTime + float64(t)*integralStepSize
    t2 := t1 + integralStepSize
    functionValue1 := (math.Pow(t1-currentTime, hurstExponent-0.5) /
        math.Gamma(hurstExponent+0.5)) *
        special.HypPFQ(a, b, 1.0-t1/currentTime)
    functionValue2 := (math.Pow(t2-currentTime, hurstExponent-0.5) /
        math.Gamma(hurstExponent+0.5)) *
        special.HypPFQ(a, b, 1.0-t2/currentTime)
    integralValue += 0.5 * (functionValue1 + functionValue2) * integralStepSize
  }
  return integralValue
}
```

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

$$S^i_{\mathsf{t}+1}(X',\mathsf{t}) = [N_\lambda]^i_{\mathsf{t}+1} - [N_\lambda]^i_{\mathsf{t}}\,, \tag{1.10}$$

where $[N_\lambda]^i_{\mathsf{t}}$ is a sample from a Poisson process with rate λ. 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 $\delta t(\mathsf{t}+1)$ 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

$$S^i_{\mathsf{t}+1}(X',\mathsf{t}) = [N_{\lambda(\mathsf{t}+1)}]^i_{\mathsf{t}+1} - [N_{\lambda(\mathsf{t})}]^i_{\mathsf{t}}\,, \tag{1.11}$$

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 τ 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, τ 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*. The following chunk of Go code may also help to understand this concept from the programmer's perspective.

```go
// generate new event rates as a function of timestep
func EventRateLambdaFunction(timeStepNumber int) float64 {
    // return a new rate
}

// get the next exponentially-distributed time increment
func ExpDistributedTimeIncrementFunction(
    smallTimeInterval float64,
) float64 {
    nextTimeIncrement := smallTimeInterval * rand.ExpFloat64()
    return nextTimeIncrement
}

// returns the time-inhomogeneous Poisson process S(X',t) term
func SFunctionInhomogeneousPoissonProcess(
    stateHistory StateHistory,
    timeStepNumber int,
) StateVector {
    // notice how the noise is also Markovian here too
    var smallTimeInterval float64
    sFunctionValue := make(StateVector, 0)
    for element = range stateVector {
        timeIncrement := ExpDistributedTimeIncrementFunction(
            smallTimeInterval,
        )
        // specify an arbitrary function of time for the event rate here
        eventRateLambda := EventRateLambdaFunction(timeStepNumber)
        prob := eventRateLambda /
            (eventRateLambda + 1.0/timeIncrement)
        // note that the difference between steps for a Poisson
        // process can only ever be 0 or 1
        element := 0.0
        if rand.Float64() < prob {
            element = 1.0
        }
        sFunctionValue = append(sFunctionValue, element)
    }
    return sFunctionValue
}
```

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

$$S^0_{\mathsf{t}+1}(X',\mathsf{t}) = \Lambda(\mathsf{t}+1) \tag{1.13}$$
$$S^1_{\mathsf{t}+1}(X',\mathsf{t}) = [N_{S^0_{\mathsf{t}+1}}]^i_{\mathsf{t}+1} - [N_{S^0_{\mathsf{t}}}]^i_{\mathsf{t}}\,. \tag{1.14}$$

This process could be simulated using the Go code we wrote for the time-inhomogeneous Poisson process above — where we would just replace `EventRateLambdaFunction` with a function 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.,

$$S^i_{\mathsf{t}+1}(X',\mathsf{t}) = [J_\lambda]^i_{\mathsf{t}+1} - [J_\lambda]^i_{\mathsf{t}}\,. \tag{1.15}$$

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 τ is chosen. Once again, the Go code we wrote above would be sufficient to simulate this process with one tweak: replace the allocation of the `element` variable to `element = 1.0` with the output of a function which generates the $[J_\lambda]^i_{\mathsf{t}}$ samples.

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

$$S^0_{\mathsf{t}+1}(X',\mathsf{t}) = \mathcal{I}_{\mathsf{t}+1}(X',\mathsf{t}) \tag{1.16}$$
$$S^1_{\mathsf{t}+1}(X',\mathsf{t}) = [N_{S^0_{\mathsf{t}+1}}]^i_{\mathsf{t}+1} - [N_{S^0_{\mathsf{t}}}]^i_{\mathsf{t}}\,, \tag{1.17}$$

where the stochastic rate $\mathcal{I}_{\mathsf{t}+1}(X',\mathsf{t})$ 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

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

where γ is the 'exciting kernel' and μ is some constant background rate. In order to simulate a Hawkes process using our formalism, the Go code would look like this.

```go
func ExcitingKernel(
    currentTime float64, somePreviousTime float64,
) float64 {
  // returns some number
}

// returns the Hawkes process S(X',t) term
func SFunctionHawkesProcess(
    stateHistory StateHistory,
    timeStepNumber int,
) StateVector {
    // notice how the noise is also Markovian here too
    var constantBackgroundRate float64
    sFunctionValue := make(StateVector, 2)
    timeIncrement := ExpDistributedTimeIncrementFunction(
        smallTimeInterval,
    )
    // loop over past state vectors and sum them to get the
    // self-exciting stochastic event rate for the Hawkes process
    // and put this in state vector index 0
    sFunctionValue[0] = constantBackgroundRate
    for lagIndex, stateVector := range stateHistory {
        currentTime := ElapsedTimeFunction(timeStepNumber)
        laggedTime := ElapsedTimeFunction(timeStepNumber-lagIndex)
        sFunctionValue[0] += ExcitingKernel(
            currentTime,
            laggedTime,
        ) * stateVector[1]
    }
    // use the stochastic rate to compute the probability of the
    // next event occurring with the same rejection method as for the
    // time-inhomogeneous Poisson process above
    prob := sFunctionValue[0] /
        (sFunctionValue[0] + 1.0/timeIncrement)
    sFunctionValue[1] := 0.0
    if rand.Float64() < prob {
        sFunctionValue[1] = 1.0
    }
    return sFunctionValue
}
```

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

$$S^0_{\mathsf{t}+1}(X', \mathsf{t}) = W^0_{\mathsf{t}+1} - W^0_{\mathsf{t}} \tag{1.19}$$

$$S^1_{\mathsf{t}+1}(X', \mathsf{t}) = \frac{1}{T}\sum_{\mathsf{t}'=0}^{\mathsf{t}} e^{-\frac{t(\mathsf{t})-t(\mathsf{t}')}{T}} S^0_{\mathsf{t}'} \,, \tag{1.20}$$

where T is some decay coefficient which quantifies the length of memory in continuous time.

So we've introduced the basic elements of our computational formalism and demonstrated how flexible the approach can be in simulating just about any stochastic phenomenon imaginable. Before progressing to algorithm design, it will be helpful to discuss some useful concepts that should enable us analyse the system later on in the book.

1.2 Useful probabilistic concepts

The general stochastic process that we defined with Eq. (1.1) also has an implicit *master equation* associated to it which fully describes the time evolution of the *probability density function* $P_{\mathsf{t}+1}(x)$ of the most recent matrix row $x = X_{\mathsf{t}+1}$ at time t. This can be written as

$$P_{\mathsf{t}+1}(x) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}x' P_{\mathsf{t}'}(x') P_{(\mathsf{t}+1)\mathsf{t}'}(x|x') \,, \tag{1.21}$$

where at the moment we are assuming the state space is continuous in each dimension and $P_{(\mathsf{t}+1)\mathsf{t}'}(x|x')$ is the conditional probability that the matrix row at time $(\mathsf{t}+1)$ will be $x = X_{\mathsf{t}+1}$

given that the row at time t' was $x' = X_{\mathsf{t}'}$. This is a very general equation which should almost always apply to any continuous stochastic phenomenon we want to study in due course. To try and understand what this equation is saying we find it's helpful to think of an iterative relationship between probabilities; each of which is connected by their relative conditional probabilities. This kind of thinking is also illustrated in Fig. 1.2.

The factor of $1/\mathsf{t}$ in Eq. (1.21) is a normalisation factor — this just normalises the sum of all probabilities to 1 given that there is a sum over t'. Note that, if the process is defined over continuous time, we would need to replace

$$\frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \rightarrow \frac{1}{t(\mathsf{t})}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \delta t(\mathsf{t}')\,. \tag{1.22}$$

But what is ω_{t}? You can think of this as just the domain of possible x' inputs into the integral which will depend on the specific stochastic process we are looking at.

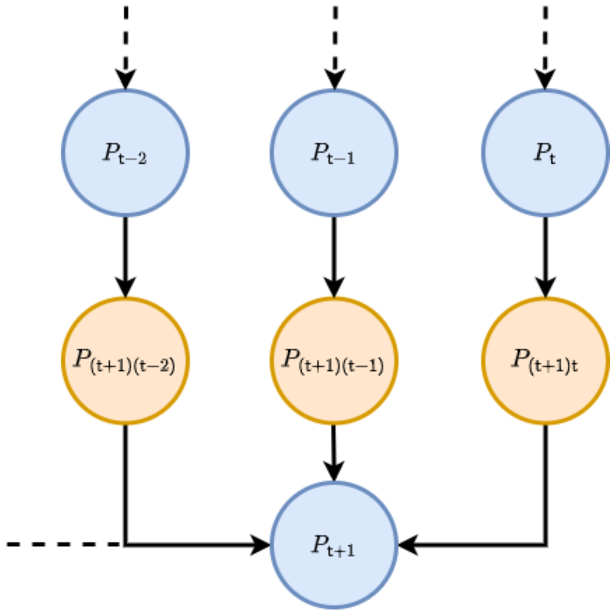

$P_{\mathsf{t}-2}$
$P_{\mathsf{t}-1}$
P_{t}
$P_{(\mathsf{t}+1)(\mathsf{t}-2)}$
$P_{(\mathsf{t}+1)(\mathsf{t}-1)}$
$P_{(\mathsf{t}+1)\mathsf{t}}$
$P_{\mathsf{t}+1}$

Figure 1.2: Graph representation of Eq. (1.21).

What if we wanted the joint distribution of both rows $P_{(\mathsf{t}+1)\mathsf{t}'}(x, x')$? One way to obtain this would be to extend Eq. (1.21) such that both matrix rows are marginalised over separately like so

$$P_{(\mathsf{t}+1)\mathsf{t}'}(x, x') = \frac{1}{(\mathsf{t}'-1)\mathsf{t}}\sum_{\mathsf{t}''=0}^{\mathsf{t}}\sum_{\mathsf{t}'''=0}^{\mathsf{t}'-1}\int_{\omega_{\mathsf{t}''}} \mathrm{d}x'' \int_{\omega_{\mathsf{t}'''}} \mathrm{d}x''' P_{\mathsf{t}''\mathsf{t}'''}(x'', x''')P_{(\mathsf{t}+1)\mathsf{t}''}(x|x'')P_{\mathsf{t}'\mathsf{t}'''}(x'|x''')\,. \tag{1.23}$$

Given Eqs. (1.21) and (1.23) it's also possible to work out what the conditional probabilities would look like using the simple relation

$$P_{(\mathsf{t}+1)\mathsf{t}'}(x|x') = \frac{P_{(\mathsf{t}+1)\mathsf{t}'}(x,x')}{P_{\mathsf{t}'}(x')}\,. \tag{1.24}$$

The implicit notation in Eq. (1.21) can hide some staggering complexity. To analyse the system in more detail, we can also do a kind of Kramers-Moyal expansion [8, 9] for each point in time to approximate the overall equation like this

$$\begin{aligned} P_{\mathsf{t}+1}(x) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} P_{\mathsf{t}'}(x) - \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\sum_{i=1}^{d}\frac{\partial}{\partial x^i}\left[\alpha^i_{(\mathsf{t}+1)\mathsf{t}'}(x)P_{\mathsf{t}'}(x)\right] \\ + \frac{1}{2\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\sum_{i=1}^{d}\sum_{j=1}^{d}\frac{\partial}{\partial x^i}\frac{\partial}{\partial x^j}\left[\beta^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(x)P_{\mathsf{t}'}(x)\right] + \dots\,, \end{aligned} \tag{1.25}$$

in which we have assumed that the state space is d-dimensional. In this expansion, we also needed to define these new integrals

$$\alpha^i_{(\mathsf{t}+1)\mathsf{t}'}(x) = \int_{\omega_{\mathsf{t}'}} \mathrm{d}x'(x'-x)^i P_{(\mathsf{t}+1)\mathsf{t}'}(x'|x) \tag{1.26}$$

$$\beta^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(x) = \int_{\omega_{\mathsf{t}'}} \mathrm{d}x'(x'-x)^i(x'-x)^j P_{(\mathsf{t}+1)\mathsf{t}'}(x'|x)\,. \tag{1.27}$$

So the matrix notation of Eq. (1.21) can indeed hide a very complicated calculation. Truncating the expansion at second-order, Eq. (1.25) tells us that there can be first and second derivatives contributing to the flow of probability to each element of the row $x = X_{\mathsf{t}+1}$ which depend on every element of the matrix X'. The probability does indeed *flow*, in fact. We can define a quantity known as the 'probability current' $J_{(\mathsf{t}+1)\mathsf{t}'}(x)$ from t' to $(\mathsf{t}+1)$ which illustrates this through the following continuity relation

$$P_{\mathsf{t}+1}(x) - \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} P_{\mathsf{t}'}(x) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\left[P_{\mathsf{t}+1}(x) - P_{\mathsf{t}'}(x)\right] = -\frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} J_{(\mathsf{t}+1)\mathsf{t}'}(x)\,. \tag{1.28}$$

By inspection of Eq. (1.25) we can therefore also deduce that

$$J^i_{(\mathsf{t}+1)\mathsf{t}'}(x) = \alpha^i_{(\mathsf{t}+1)\mathsf{t}'}(x)P_{\mathsf{t}'}(x) - \frac{1}{2}\sum_{j=1}^{d}\frac{\partial}{\partial x^j}\left[\beta^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(x)P_{\mathsf{t}'}(x)\right] + \dots\,. \tag{1.29}$$

What would happen if we assumed that α and β were just arbitrary time-dependent functions? For example, let's make the following assumptions

$$\alpha^i_{(\mathsf{t}+1)\mathsf{t}'}(x) = \mu^i(\mathsf{t}') - x^i \tag{1.30}$$

$$\beta^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(x) = 2\Sigma^{ij}(\theta,\mathsf{t}')\,, \tag{1.31}$$

where $\mu(\mathsf{t}')$ is an arbitrary vector-valued function of the timestep and $\Sigma(\theta,\mathsf{t}')$ is an arbitrary matrix (often known as the 'diffusion tensor') which depends on both the timestep and a set of hyperparameters θ. If we now also assume stationarity of $P_{\mathsf{t}'}(x) = P_{\mathsf{t}''}(x)$ for any t' and t'' such that

$$P_{\mathsf{t}+1}(x) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} P_{\mathsf{t}'}(x)\,, \tag{1.32}$$

we can solve Eq. (1.25) to obtain the following stationary solution

$$P_{\mathsf{t}'}(x) = \mathsf{MultivariateNormalPDF}[x; \mu(\mathsf{t}'), \Sigma(\theta, \mathsf{t}')] \,. \tag{1.33}$$

I actually hid a little bit of the detail in that last step; the solution also required the identification that the flow of probability between timesteps vanishes uniquely for each and every t' such that $J_{(\mathsf{t}+1)\mathsf{t}'}(x) = 0$.

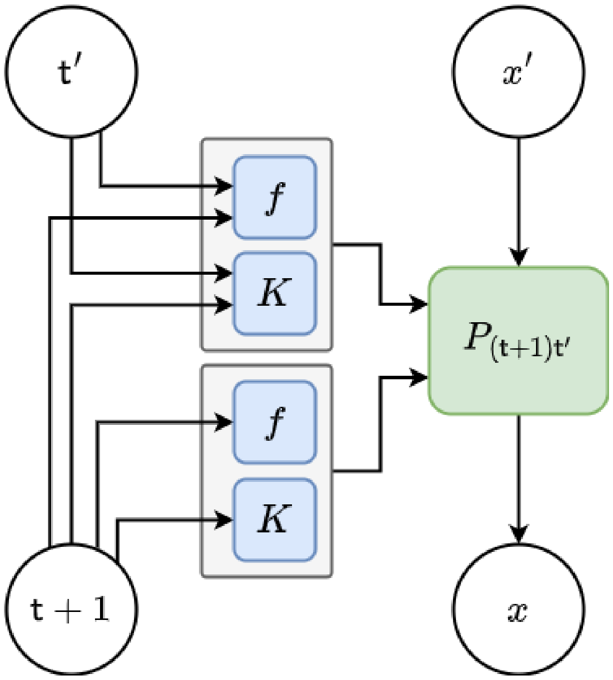


Figure 1.3: Graph representation of the generative process implied by Eq. (1.34).

It's possible to take this derivation a bit further by expanding Eq. (1.23) in a similar fashion, truncating it to second-order, assuming only time-dependent terms and then solving it in the stationary limit. By plugging this solution (and its corresponding marginal distribution equivalent) into Eq. (1.24), it's possible to get something that looks like this conditional distribution

$$\begin{aligned} P_{(\mathsf{t}+1)\mathsf{t}'}(x|x') \propto \exp\bigg\{ &-\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} \big[x - f(\mathsf{t}+1)\big]^i [K^{-1}(\theta, \mathsf{t}+1, \mathsf{t}+1)]^{ij} \big[x - f(\mathsf{t}+1)\big]^j \\ &+ \sum_{i=1}^{d}\sum_{j=1}^{d} \big[x - f(\mathsf{t}+1)\big]^i [K^{-1}(\theta, \mathsf{t}+1, \mathsf{t}')]^{ij} \big[x' - f(\mathsf{t}')\big]^j \bigg\} \,, \end{aligned} \tag{1.34}$$

where $K(\theta, \mathsf{t}+1, \mathsf{t}')$ is some arbitrary covariance matrix that encodes how the correlation structure varies with the between compared states at two different timesteps and $K^{-1}(\theta, \mathsf{t}+1, \mathsf{t}')$ denotes taking its inverse. Eq. (1.34) may look a bit familiar to some readers who like using Gaussian processes from the machine learning literature [10] — this version implies a *generative* model for a future x value (which we've illustrated in Fig 1.3), in contrast to the more standard equation used to *infer* values of f. These are two sides of the same coin though.

What other processes can be described by Eq. (1.21)? For Markovian phenomena, the equation no longer depends on timesteps older than the immediately previous one, hence the expression reduces to just

$$P_{\mathsf{t}+1}(x) = \int_{\omega_{\mathsf{t}}} \mathrm{d}x' P_{\mathsf{t}}(x') P_{(\mathsf{t}+1)\mathsf{t}}(x|x') \,. \tag{1.35}$$

It's also easy to show that Eq. (1.25) naturally simplifies into the more usually applied Kramers-Moyal expansion when considering a Markovian process — you just remove the sum over t' and the $1/\mathsf{t}$ normalisation factor.

Note that an analog of Eq. (1.21) exists for discrete state spaces as well. We just need to replace the integral with a sum and the schematic would look something like this

$$P_{\mathsf{t}+1}(x) = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{\omega_{\mathsf{t}'}} P_{\mathsf{t}'}(x') P_{(\mathsf{t}+1)\mathsf{t}'}(x|x') \,, \tag{1.36}$$

where we note that the P's in the expression above all now refer to *probability mass functions*. Because the state space is now discrete, we cannot immediately intuit an approximative expansion from this expression.

Ok, as a brief mathematical aside; if one is really determined to use a similar approach to the one we derived above, it's quite straightforward to rewrite it in terms of continuous-valued characteristic functions like so

$$\varphi_{\mathsf{t}+1}(s) = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\ell_{\mathsf{t}'}} \mathrm{d}s' \mathcal{C}(s') \varphi_{\mathsf{t}'}(s') \varphi_{(\mathsf{t}+1)\mathsf{t}'}(s|s') \tag{1.37}$$

$$\mathcal{C}(s') = \frac{1}{(2\pi)^d} \sum_{\omega_{\mathsf{t}'}} e^{-i(s' \cdot x')} \,, \tag{1.38}$$

where $\ell_{\mathsf{t}'}$ defines all the continuous values that the vector s' can possibly have at time t'. In the expression above, $\mathcal{C}(s')$ acts like is a kind of comb[3] to map the continuous frequency domain of s' onto the discrete state space of x'. Note also that $\mathcal{C}(s')$ uses the imaginary number i and, to be visually tidier, the dot product notation $a \cdot b$ just means the sum of vector elements: $a \cdot b = \sum_{\forall k} a^k b^k$. In principle, one can perform an approximative expansion on Eq. (1.37) like we did for continuous state spaces. This isn't always the most practical way of analysing the system though.

We have one more important example to discuss and then we can cap off this analysis subsection. In the even-simpler case where x is just a vector of binary 'on' or 'off' states, Eq. (1.36) reduces to

$$P^i_{\mathsf{t}+1} = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{j=1}^{d} P^j_{\mathsf{t}'} P^{ij}_{(\mathsf{t}+1)\mathsf{t}'} = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{j=1}^{d} \left[P^j_{\mathsf{t}'} A^{ij}_{(\mathsf{t}+1)\mathsf{t}'} + (1 - P^j_{\mathsf{t}'}) B^{ij}_{(\mathsf{t}+1)\mathsf{t}'} \right] , \tag{1.39}$$

where $P^i_{\mathsf{t}'}$ now represents the probability that element $x^i = 1$ (is 'on') at time t'. The matrices A and B are defined as conditional probabilities where the previous state in time $P^j_{\mathsf{t}'}$ was either 'on' or 'off', respectively.

[3]This is very similar to how a 'Dirac comb' works in signal processing [11].

1.3 Software design

So we've proposed a computational formalism and done a bit of analysis on it 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.

Once someone 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.

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

In summary, we've put together a schematic of data types and their relationships in Fig. 1.4. 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 [12], StoSpa [13] and FLAME GPU [14] — have all approached this problem in different ways, and with different software architectures.

In Fig. 1.5 we've illustrated what a loop involving two separate state partitions looks like in the

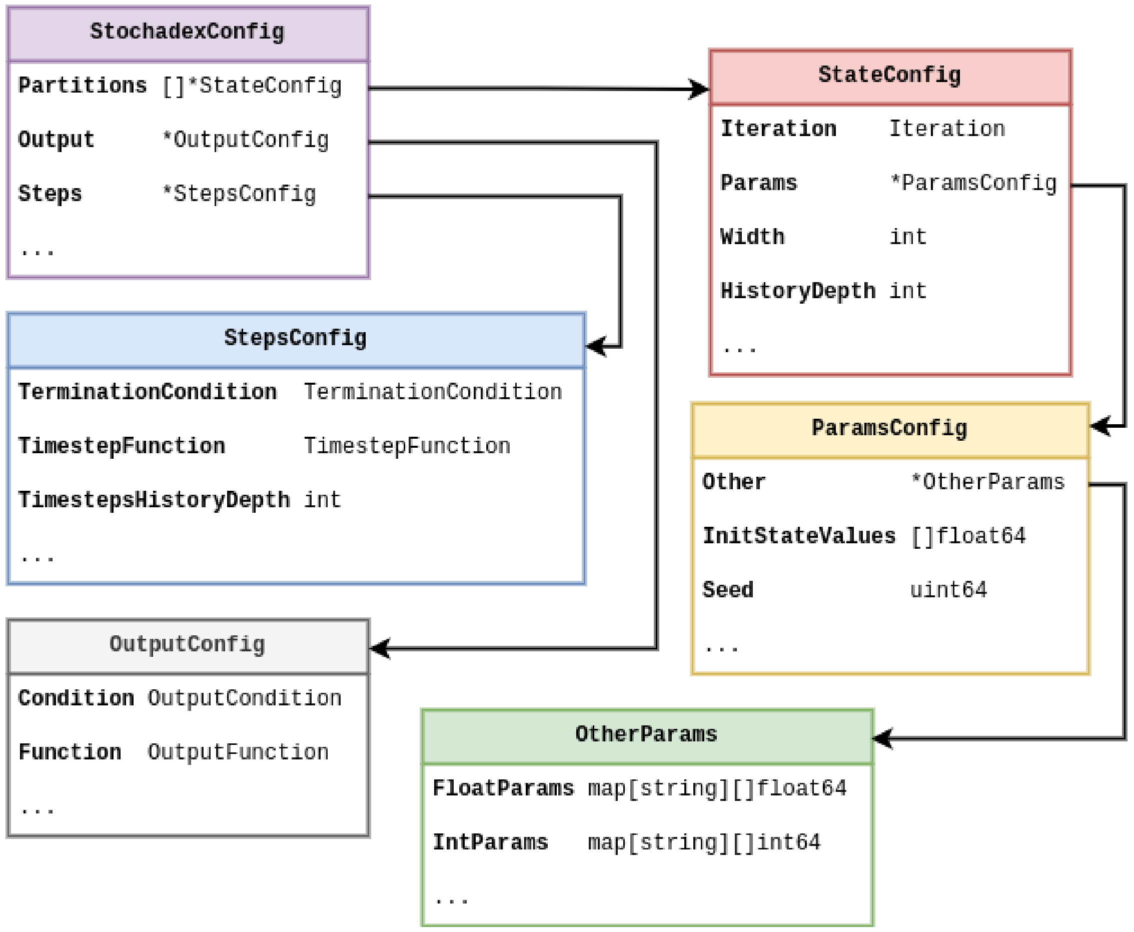


Figure 1.4: A relational summary of the core data types in the stochadex.

stochadex simulator. Each parition is handled by concurrently running execution threads of the same process, while a separate process is used to handle the outputs from the algorithm. While this diagram illustrates only a single use for multiple processes, it's obviously true that we may run many of these whole diagrams at once on a multicore machine to generate a batch of independent ensembles if necessary.

1.4 Implementation details

Now that the design of the algorithm and its basic data types have been outlined, we can finally turn to the implementation details. As we mentioned in the introduction, most of the core software in this book has been written in Go — the main reason for this is mostly because it is a performant language that we enjoy developing with. More objectively, the key feature of Go that made it attractive for building the stochadex are its lightweight concurrency primitives called 'goroutines' [15] which made

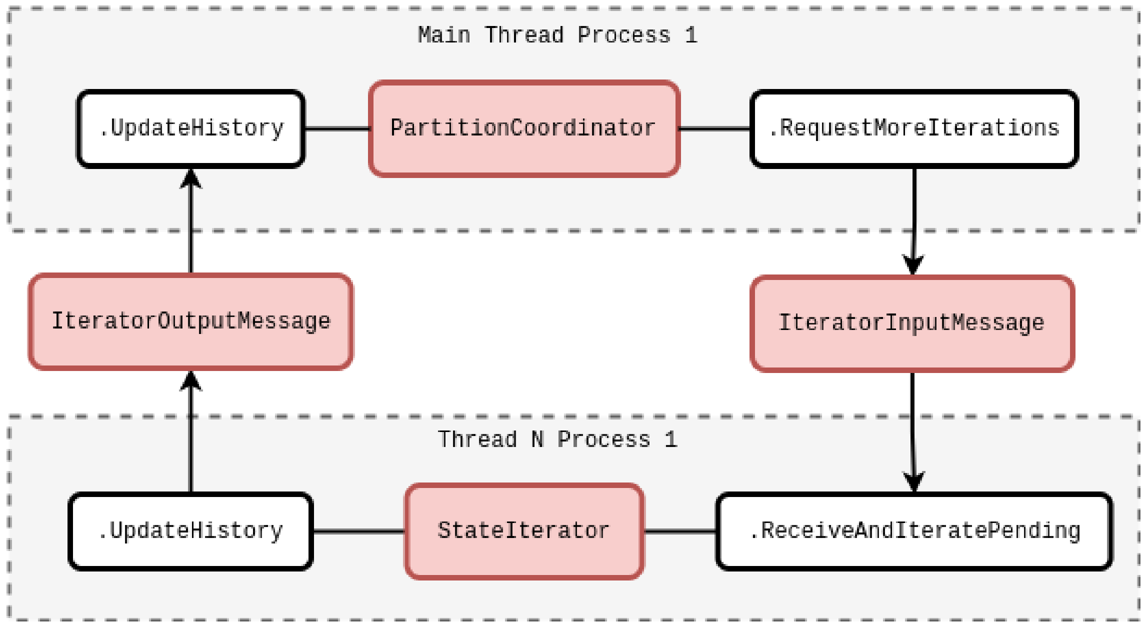


Figure 1.5: A loop of the stochadex simulation algorithm with two state partitions.

it easy and quick to piece together a scalable pipeline without excessive head-scratching. Ideal for projects in your limited free time!

Remember to reference the godoc documentation for the code here.

Bibliography

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

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

[3] L. Rogers and D. Williams, *Diffusions, Markov Processes and Martingales 2: Ito Calculus*, vol. 1, pp. xiv+480. Cambridge University Press, 04, 2000. 10.1017/CBO9781107590120.

[4] L. Decreusefond et al., *Stochastic analysis of the fractional brownian motion*, *Potential analysis* **10** (1999) 177–214.

[5] D. T. Gillespie, *Exact stochastic simulation of coupled chemical reactions*, *The journal of physical chemistry* **81** (1977) 2340–2361.

[6] J. Neyman and E. L. Scott, *Statistical approach to problems of cosmology*, *Journal of the Royal Statistical Society: Series B (Methodological)* **20** (1958) 1–29.

[7] A. G. Hawkes, *Spectra of some self-exciting and mutually exciting point processes*, *Biometrika* **58** (1971) 83–90.

[8] H. A. Kramers, *Brownian motion in a field of force and the diffusion model of chemical reactions*, *Physica* **7** (1940) 284–304.

[9] J. Moyal, *Stochastic processes and statistical physics*, *Journal of the Royal Statistical Society. Series B (Methodological)* **11** (1949) 150–210.

[10] K. P. Murphy, *Machine learning: a probabilistic perspective.* MIT press, 2012.

[11] D. Brandwood, *Fourier transforms in radar and signal processing.* Artech House, 2012.

[12] "SimPy: a process-based discrete-event simulation framework." `https://gitlab.com/team-simpy/simpy/`.

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

[14] "FLAME GPU: A GPU accelerated agent-based simulation library for domain independent complex systems simulation." `https://github.com/FLAMEGPU/FLAMEGPU2/`.

[15] "A Tour of Go: Goroutines." `https://go.dev/tour/concurrency/1`.