Diffusing Ideas

Software, noise and building mathematical toys

Robert J. Hardwick and C. M. Gomez-Perales

April 10, 2023

Introduction

Diffusing Ideas is a book of research exploration and software development which we have written for the interest of mathematically-inclined programmers and computational scientists. It's the output of many interrelated projects over several years which have sought to generalise the computational mathematics of simulating, statistically inferring, manipulating and automatically controlling stochastic phenomena as far as possible.

The book accompanies a lot of new open-source scientific software written predominantly in Go [1], and with some Python [2] to top it off. A major motivation for creating these new tools is to prepare a foundation of code from which to develop new and more complex applications. We also hope that the resulting framework will enable anyone to explore and study new phenomena effectively, regardless of their scientific background.

The need to properly test all this software has also provided a wonderful excuse to study and play with an extensive range of mathematical toy models. we've chosen these models based on a fairly broad background of interests, but also to illustrate the remarkable cross-disciplinary applicability of stochastic processes. However, we've often found that mathematical formalities can obscure the computations that a programmer must implement. So, while we've tried to be as ambitious as possible with the level of technical sophistication in these models, we've also tried to write the expressions in a computer-friendly way where feasible[1] and have added code snippets throughout.

A quick note on the code: any software that we describe in this book (including the software which compiles the book itself [3]) will always be shared under a MIT License [4] in a public Git repository.[2] Forking these repositories and submitting pull requests for new features or applications is strongly encouraged too, though we apologise in advance if we don't follow these up very quickly as all of this work has to be conducted independently in free time, outside of work hours.

No quest would be complete without a guide, so we think this introduction should end with a list of the key milestones in the book; comprising its four major parts. These parts each correspond to answering one of the following interdependent research questions:

Part 1. How do we simulate a general set of stochastic phenomena?

Part 2. How do we then learn/identify the answer to **Part 1** from real-world data?

Part 3. How do we simulate a general set of control policies to interact with the answer to **Part 1**?

Part 4. How do we then optimise the answer to **Part 3** to achieve a specified control objective?
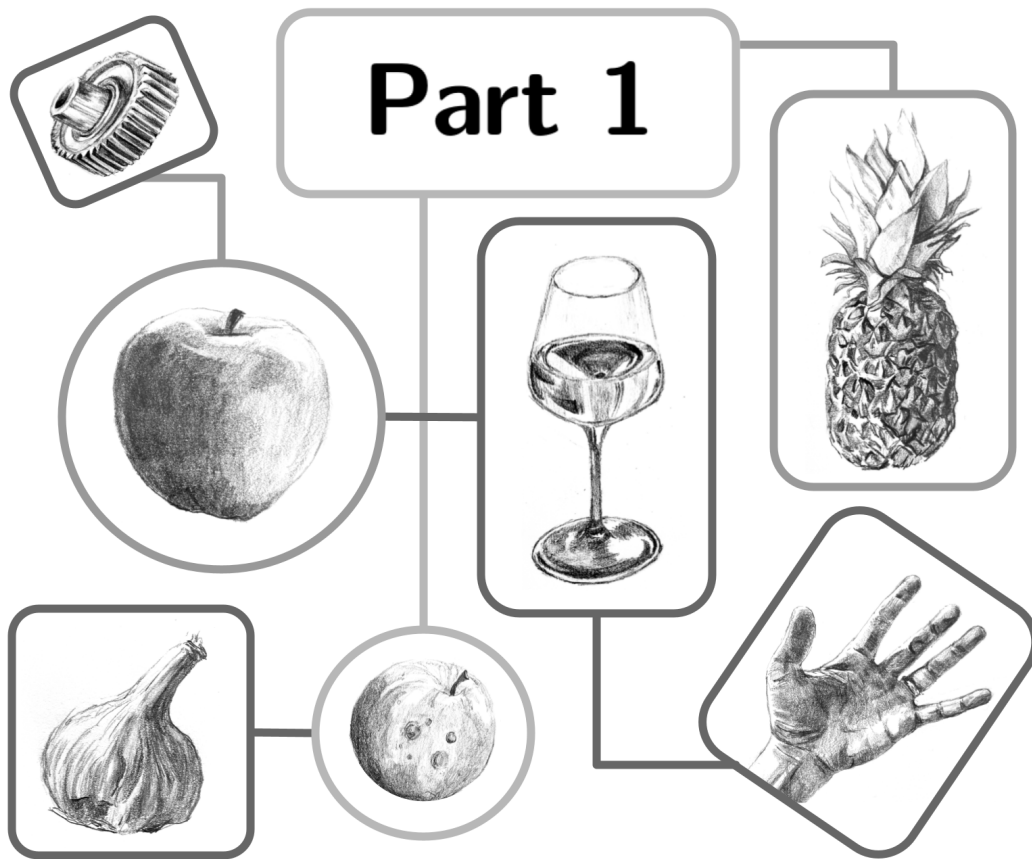
---

[1]For example, we'll typically be thinking more in terms of 'matrices' and less about 'operators'.

[2]The repositories will always be somewhere on this list: https://github.com/umbralcalc?tab=repositories.

Table of contents

Part 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 $X' \to X$

$$X_{\mathsf{t}+1}^i = F_{\mathsf{t}+1}^i(X', z, \mathsf{t}) \,, \tag{1.1}$$

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); z is a vector of arbitrary size which contains the 'hidden' other parameters that are necessary to iterate the process; and $F_{\mathsf{t}+1}^i(X', z, \mathsf{t})$ as the latest element of an arbitrary matrix-valued function. As we shall discuss shortly, $F_{\mathsf{t}+1}^i(X', z, \mathsf{t})$ may represent not just operations on deterministic variables, but also on stochastic ones. There is also no requirement for the function to be continuous.
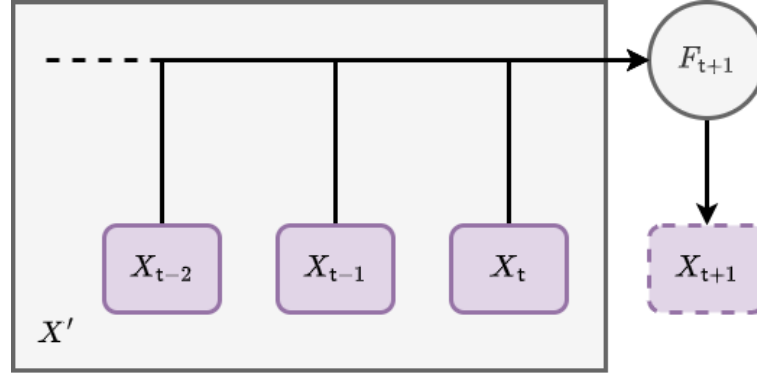
Figure 1.1: Graph representation of Eq. (1.1).

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' as an entire matrix input into a function which 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 something like this.

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

// iterate the state history forward in time by one step
func IterationFormula(
    stateHistory StateHistory,
    otherParams  OtherParams,
    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

---

[1] Or memory at least within some window.

returns the specific change in continuous time that corresponds to the step $\mathsf{t} - 1 \to \mathsf{t}$, we will always be able to compute the total elapsed time through the relation

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

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

$$F_{\mathsf{t}+1}^{i}(X', z, \mathsf{t}) = D_{\mathsf{t}+1}^{i}(X', z, \mathsf{t}) + S_{\mathsf{t}+1}^{i}(X', z, \mathsf{t}) . \tag{1.3}$$

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

$$S_{\mathsf{t}+1}^{i}(X', z, \mathsf{t}) = W_{\mathsf{t}+1}^{i} - W_{\mathsf{t}}^{i} . \tag{1.4}$$

One draws the increments $W_{\mathsf{t}+1}^{i} - W_{\mathsf{t}}^{i}$ 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 $P_{\mathsf{t}+1}^{i}(x)$ of the increments $x^{i} = W_{\mathsf{t}+1}^{i} - W_{\mathsf{t}}^{i}$ is

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

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',z,t) function we defined
// for the Wiener Process in the main text above
func SFunctionWienerProcess(
    stateHistory StateHistory,
    otherParams OtherParams,
    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_{t}^i to the Wiener process like so

$$S_{\mathsf{t}+1}^i(X', z, \mathsf{t}) = X_{\mathsf{t}}^i(W_{\mathsf{t}+1}^i - W_{\mathsf{t}}^i). \tag{1.6}$$

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' [5, 6, 7]. 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

$$S_{\mathsf{t}+1}^0(X', z, \mathsf{t}) = g[W_{\mathsf{t}+1}^0, t(\mathsf{t}+1)] - g[W_{\mathsf{t}}^0, t(\mathsf{t})] \tag{1.7}$$

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

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 $S_{\mathsf{t}+1}^i(X', z, \mathsf{t}) = (X_{\mathsf{t}+1}^i + X_{\mathsf{t}}^i)(W_{\mathsf{t}+1}^i - W_{\mathsf{t}}^i)/2$ for Eq. (1.6).

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 code would probably look something like this.

```
// 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. [8], 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

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

where $S^0_{\mathsf{t}+1}(X', z, \mathsf{t}) = [B_H]_{\mathsf{t}+1} - [B_H]_\mathsf{t}$. 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 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

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

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 [9] — 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

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

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 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

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

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',z,t) term
func SFunctionInhomogeneousPoissonProcess(
    stateHistory StateHistory,
    otherParams OtherParams,
    timeStepNumber int,
) StateVector {
    // get this from otherParams
    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 [10] 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

$$S_{\mathsf{t}+1}^0(X', z, \mathsf{t}) = \Lambda(\mathsf{t}+1) \tag{1.13}$$

$$S_{\mathsf{t}+1}^1(X', z, \mathsf{t}) = [N_{S_{\mathsf{t}+1}^0}]_{\mathsf{t}+1}^i - [N_{S_{\mathsf{t}}^0}]_{\mathsf{t}}^i. \tag{1.14}$$

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 $[N_\lambda]_{\mathsf{t}}^i$ which are replaced by independent samples $[J_\lambda]_{\mathsf{t}}^i$ from another probability distribution, i.e.,

$$S_{\mathsf{t}+1}^i(X', z, \mathsf{t}) = [J_\lambda]_{\mathsf{t}+1}^i - [J_\lambda]_{\mathsf{t}}^i. \tag{1.15}$$

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 $[J_\lambda]_{\mathsf{t}}^i$ 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

$$S_{\mathsf{t}+1}^0(X', z, \mathsf{t}) = \mathcal{I}_{\mathsf{t}+1}(X', z, \mathsf{t}) \tag{1.16}$$

$$S_{\mathsf{t}+1}^1(X', z, \mathsf{t}) = [N_{S_{\mathsf{t}+1}^0}]_{\mathsf{t}+1}^i - [N_{S_{\mathsf{t}}^0}]_{\mathsf{t}}^i, \tag{1.17}$$

where the stochastic rate $\mathcal{I}_{\mathsf{t}+1}(X', z, \mathsf{t})$ now depends on the history explicitly. Amongst other potential inputs we can see, e.g., Hawkes processes [11] as an example of above by substituting

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

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',z,t) term
func SFunctionHawkesProcess(
    stateHistory StateHistory,
    otherParams OtherParams,
    timeStepNumber int,
) StateVector {
    // notice how the noise is also Markovian here too
```

```
14    // get this from otherParams
15    var constantBackgroundRate float64
16    sFunctionValue := make(StateVector, 2)
17    timeIncrement := ExpDistributedTimeIncrementFunction(
18        smallTimeInterval,
19    )
20    // loop over past state vectors and sum them to get the
21    // self-exciting stochastic event rate for the Hawkes process
22    // and put this in state vector index 0
23    sFunctionValue[0] = constantBackgroundRate
24    for lagIndex, stateVector := range stateHistory {
25        currentTime := ElapsedTimeFunction(timeStepNumber)
26        laggedTime := ElapsedTimeFunction(timeStepNumber-lagIndex)
27        sFunctionValue[0] += ExcitingKernel(
28            currentTime,
29            laggedTime,
30        ) * stateVector[1]
31    }
32    // use the stochastic rate to compute the probability of the
33    // next event occurring with the same rejection method as for the
34    // time-inhomogeneous Poisson process above
35    prob := sFunctionValue[0] /
36        (sFunctionValue[0] + 1.0/timeIncrement)
37    sFunctionValue[1] := 0.0
38    if rand.Float64() < prob {
39        sFunctionValue[1] = 1.0
40    }
41    return sFunctionValue
42 }
```

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

$$S_{\mathsf{t}+1}^0(X', z, \mathsf{t}) = W_{\mathsf{t}+1}^0 - W_{\mathsf{t}}^0 \tag{1.19}$$

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

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 $P_{(t+1)t'}(x|x')$ is the conditional probability that the matrix row at time $(t+1)$ will be $x = X_{t+1}$ given that the row at time t' was $x' = X_{t'}$. 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 $1/t$ 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

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

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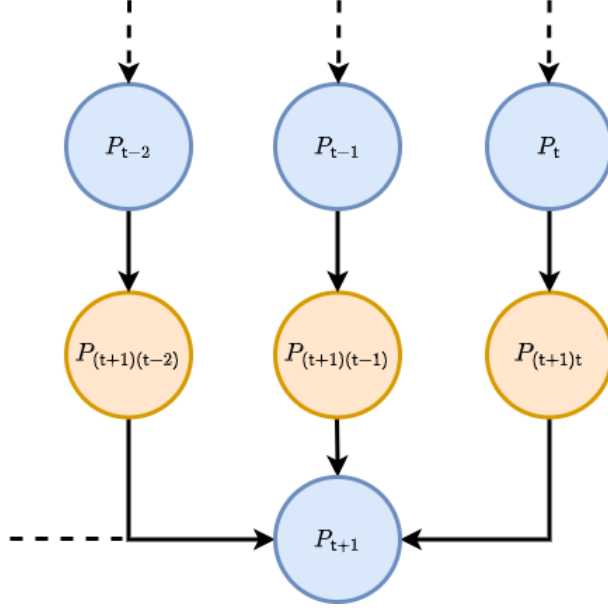
Figure 1.2: Graph representation of Eq. (1.21).

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

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

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 [12, 13] for each point in time to approximate the overall equation like this

$$P_{\mathsf{t}+1}(x) = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} P_{\mathsf{t}'}(x) - \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{i=1}^{d} \frac{\partial}{\partial x^i} \left[\alpha_{(\mathsf{t}+1)\mathsf{t}'}^{i}(x) P_{\mathsf{t}'}(x) \right]$$

$$+ \frac{1}{2\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial}{\partial x^i} \frac{\partial}{\partial x^j} \left[\beta_{(\mathsf{t}+1)\mathsf{t}'}^{ij}(x) P_{\mathsf{t}'}(x) \right] + \dots , \tag{1.25}$$

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

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

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

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

$$J_{(\mathsf{t}+1)\mathsf{t}'}^{i}(x) = \alpha_{(\mathsf{t}+1)\mathsf{t}'}^{i}(x) P_{\mathsf{t}'}(x) - \frac{1}{2} \sum_{j=1}^{d} \frac{\partial}{\partial x^j} \left[\beta_{(\mathsf{t}+1)\mathsf{t}'}^{ij}(x) P_{\mathsf{t}'}(x) \right] + \dots . \tag{1.29}$$

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

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

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

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

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

Note 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 $J_{(\mathsf{t}+1)\mathsf{t}'}(x) = 0$.
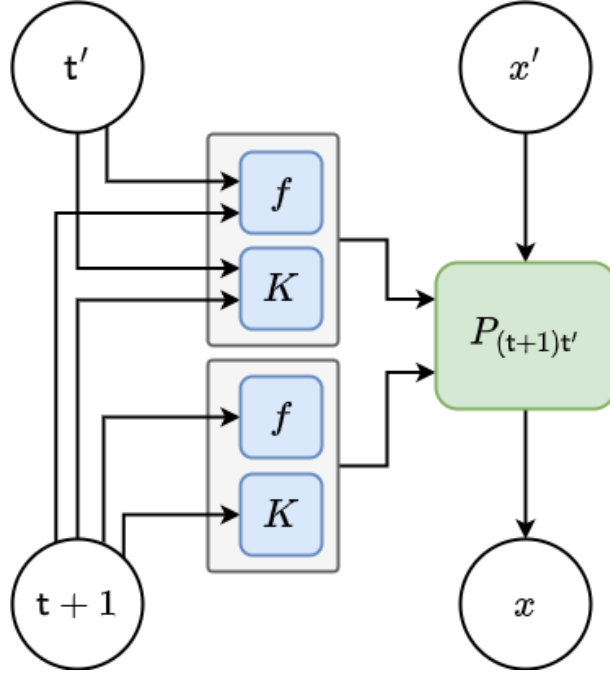

Figure 1.3: Graph representation of the generative process implied by Eq. (1.34).

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

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

$$+ \sum_{i=1}^{d} \sum_{j=1}^{d} \big[x - f(\mathsf{t}+1)\big]^{i} [K^{-1}(\theta, \mathsf{t}+1, \mathsf{t}')]^{ij} \big[x' - f(\mathsf{t}')\big]^{j} \Bigg\}, \tag{1.34}$$

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 [14] — 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

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

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

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

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

$$P_{\mathsf{t}+1}^i = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{j=1}^{d} P_{\mathsf{t}'}^j P_{(\mathsf{t}+1)\mathsf{t}'}^{ij} = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{j=1}^{d} \left[P_{\mathsf{t}'}^j A_{(\mathsf{t}+1)\mathsf{t}'}^{ij} + (1 - P_{\mathsf{t}'}^j) B_{(\mathsf{t}+1)\mathsf{t}'}^{ij} \right],$$
(1.39)

---

[3]This is very similar to how a 'Dirac comb' works in signal processing [15].

where $P_{t'}^i$ 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 $P_{t'}^j$ was either 'on' or 'off', respectively.

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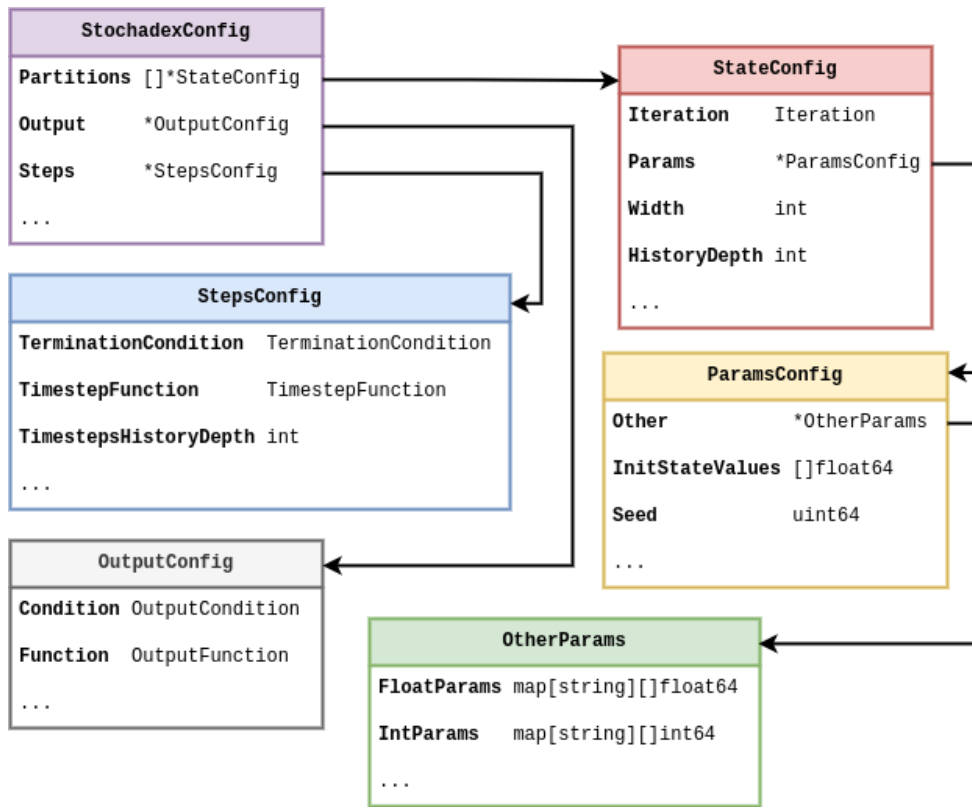
Figure 1.4: A relational summary of the core data types in the stochadex.

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 [16], StoSpa [17] and FLAME GPU [18] — have all approached this problem in different ways, and with different software architectures.

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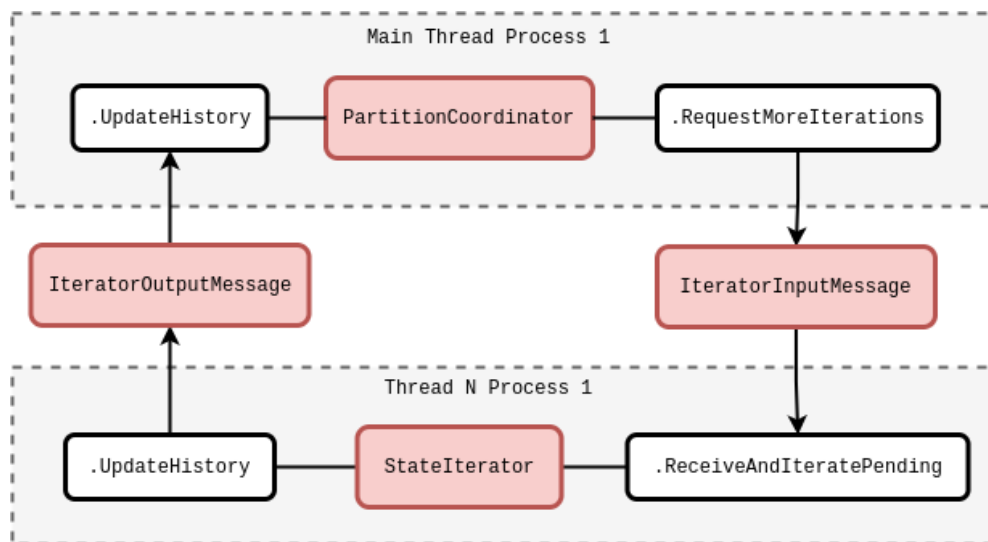
Figure 1.5: A loop of the stochadex simulation algorithm with two state partitions.

Simulating a financial market

Concept. The idea here is to use the Q-Hawkes processes and the Bouchaud work to come up with some interesting simulations of financial markets.

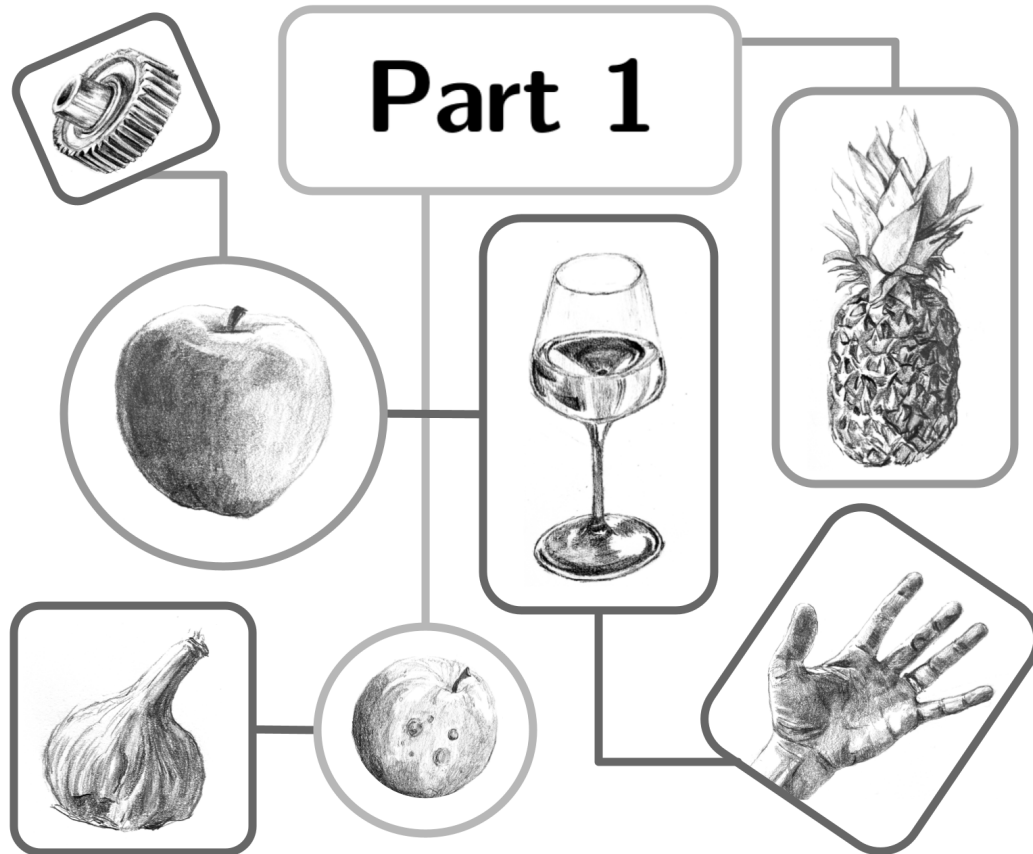## 2.1 Introducing Q-Hawkes processes

Quantum jumps on generic networks

Concept. The idea is to follow this sort of thing here to simulate the Lindblad equation over an arbitrary network of entangled states.

3.1 The Lindblad equation

Part 2. How do we then learn/identify the answer to Part 1 from real-world data?

Empirical dynamical emulators

Concept. The idea here is to leverage our probabilistic formalism to be able to empirically emulate a wide variety stochastic phenomena within a generalised framework.

Part 1

5

Inferring dynamical 2D maps
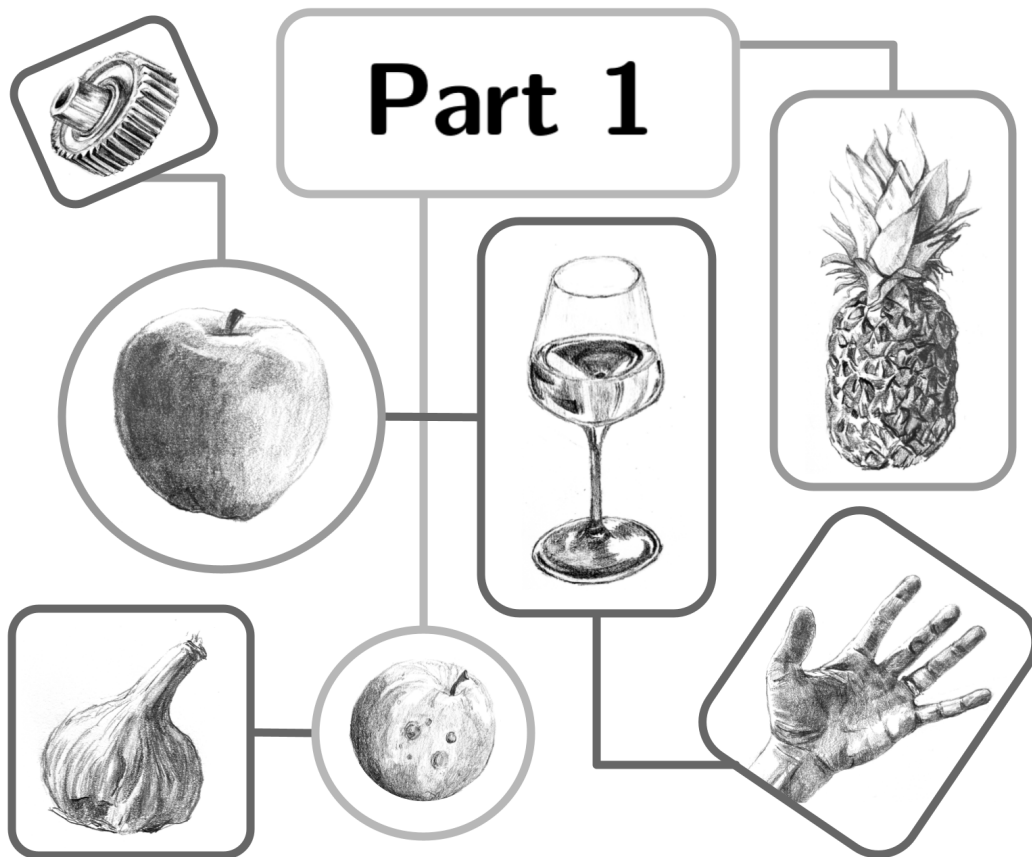
Concept. The idea here is

5.1   Adapting the stochadex formalism

Part 1

Learning from ants on curved surfaces

Concept. The idea here is

6.1 Diffusive limits for ant interactions

Part 1

A world of hydrodynamic ensembles

Concept. The idea here is

7.1 The Boltzmann/Navier-Stokes equations

Part 1

Generalised statistical inference tools

Concept. The idea here is to extend the stochadex with tools for very generalised statistical inference (ABC algorithms and the like) that will work in nearly every situation. Probably need to exploit the phase space analogy of the formalism.

8.1 Likelihood-free methods

Part 3. How do we simulate a general set of control policies to interact with the answer to Part 1?

Part 1
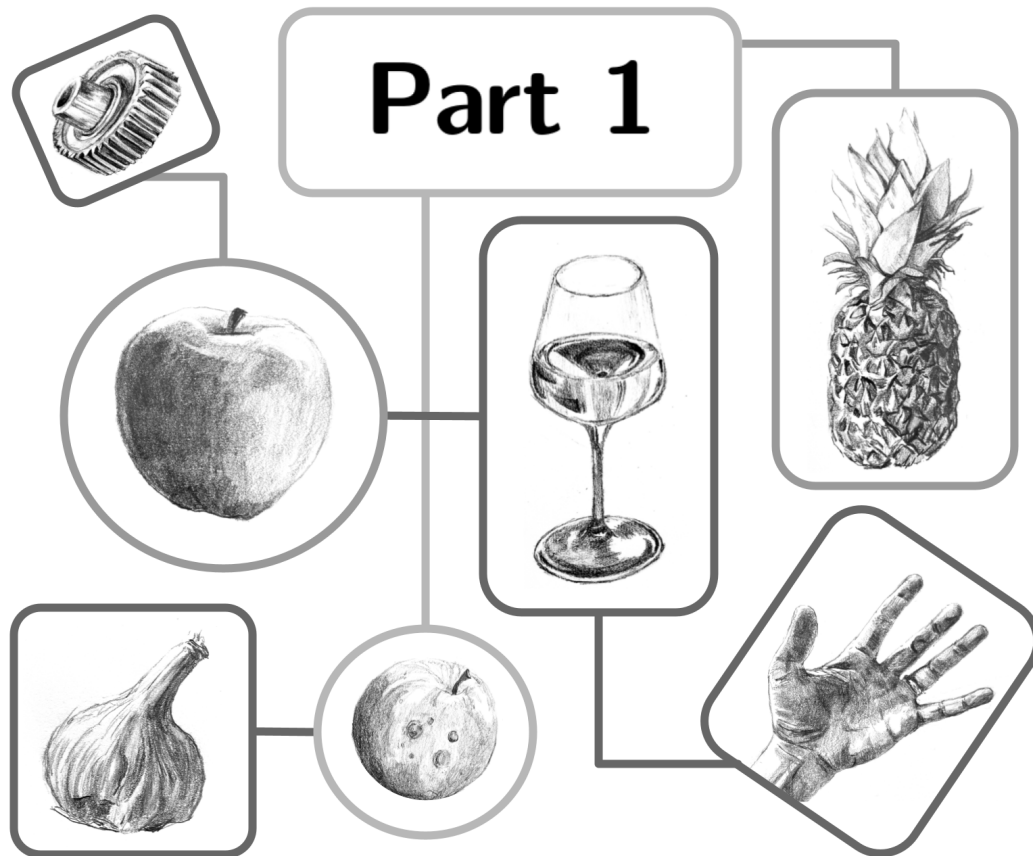
Interacting with systems in general

Concept. To design and build a generalised concept of interacting with stochastic processes of any kind. The mathematical formalism and software that we introduce here will serve as a common language and interface for any simulation studies into manipulating real world phenomena, and should enable the learning of control algorithms. We will call this software 'dexetera', since it originates as an extension to the stochadex. For the mathematically-inclined, this chapter will cover how dexetera is structured in theory by developing some useful extensions to the stochadex formalism and illustrating with some simple examples. For the programmers, the public Git repository for the code described in this chapter can be found here: https://github.com/umbralcalc/dexetera.

9.1 Formalising general interactions

Let's start by considering how we might adapt the mathematical formalism we have been using so far to be able to take actions which can manipulate the state at each timestep. Using the mathematical notation that we inherited from the stochadex, we may extend the formula for updating the state history matrix $X' \to X$ to include a vector-valued 'state measurement' function M_{t} and two layers of possible interactions which are facilitated by a new vector-valued 'parametric action' function G_{t} and a new vector-valued 'state action' function H_{t}. During a timestep over which these actions are applied, the stochadex state update formula can be extended to look like this system of equations

$$\mathcal{S}_{\mathsf{t}}^i = M_{\mathsf{t}}^i(X', Z_{\mathsf{t}}, \mathsf{t}) \tag{9.1}$$

$$Z_{\mathsf{t}+1}^i = G_{\mathsf{t}+1}^i(Z_{\mathsf{t}}, \mathcal{S}_{\mathsf{t}}, \mathcal{A}_{\mathsf{t}}, \mathsf{t}) \tag{9.2}$$

$$X_{\mathsf{t}+1}^i = H_{\mathsf{t}+1}^i[F_{\mathsf{t}}(X', Z_{\mathsf{t}+1}, \mathsf{t}), S_{\mathsf{t}}, \mathcal{A}_{\mathsf{t}}, \mathsf{t}], \tag{9.3}$$

where, other than the new functions, we have introduced; the concept of a 'measured state' $\mathcal{S}_{\mathsf{t}}$ of the system (some vector of the that doesn't have to share the same length as X_{t}) and the actions performed $\mathcal{A}_{\mathsf{t}}$ on the system (some vector of parameters which define what actions are taken) at timestep t.

In Eqs. (9.2) and (9.3), notice that we have replaced the constant vector of parameters z (as in the stochadex formalism) for a time-dependent vector Z_t of parameters that can be updated by G_t at any (but not necessarily every) timestep. From the perspective of the whole matrix X update step, the measurement of M_t proceeded by the actions of both G_t and H_t all combine to become technically equivalent to applying a formal composition of functions like this

$$X^i_{\mathsf{t}+1} = H^i_{\mathsf{t}+1}(F_\mathsf{t}(X', G^i_{\mathsf{t}+1}(Z_\mathsf{t}, M_\mathsf{t}(X', Z_\mathsf{t}, \mathsf{t}), \mathcal{A}_\mathsf{t}, \mathsf{t}), \mathsf{t}), M_\mathsf{t}(X', Z_\mathsf{t}, \mathsf{t}), \mathcal{A}_\mathsf{t}, \mathsf{t}) \tag{9.4}$$

$$= \mathcal{F}^i_{\mathsf{t}+1}(X', Z_\mathsf{t}, \mathcal{A}_\mathsf{t}, \mathsf{t}), \tag{9.5}$$

which looks like an absolute mess! However, it illustrates how $\mathcal{F}$, which refers to a modified version of the F function, contains our actions but no other new parameters need be specified; only function operations. Hence, while we have provided two distinct ways one might encode actions to manipulate a stochastic phenomenon, we shall often just refer to them together as 'taking an action' $\mathcal{A}_\mathsf{t}$ at timestep t — because the parameters which define either type of action at this timestep should all be stored within $\mathcal{A}_\mathsf{t}$ anyway. It is, however, important not to forget the full mathematical formulation when performing calculations.

In Go, the code for the new iteration formula given by Eq. (9.5), which includes taking actions in the same timestep, would look something like this.

```go
type ActionsVector       []float64
type MeasuredStateVector  []float64

// iterate the state history forward in time by one step
// with actions in the process
func IterationFormulaWithActions(
    stateHistory StateHistory,
    otherParams  OtherParams,
    actions ActionsVector,
    timeStepNumber int,
) StateVector {
    // measuring the state
    var measuredState MeasuredStateVector
    measuredState = MeasureTheState(stateHistory, otherParams, timeStepNumber)
    // taking a parametric action
    newestOtherParams := TakeParametricAction(
        otherParams,
        measuredState,
        actions,
        timeStepNumber,
    )
    newestStateVector := IterationFormula(
        stateHistory,
        otherParams,
        timeStepNumber,
    )
    // taking a state action
    newestStateVector = TakeStateAction(
        newestStateVector,
        measuredState,
        actions,
        timeStepNumber,
    )
    return newestStateVector
}
```

10

Angling for freshwater fish

Concept. The idea here is

10.1 A large-scale Lotka-Volterra model

Inspired by the empirical dynamical modelling approach to sockeye salmon in Ref. [19], but also desiring a generative model which has some link to the classic causal models promoted by mathematical ecology; the goal here is to create and calibrate a stochastic model which predicts the fish counts, weights, lengths and ages for each species in each area based on the past system states. To do this, we will combine some well-known models from mathematical ecology with supervised learning.

The one-step master equation for the proposed stochastic simulation is given implictly by

$$\frac{\mathrm{d}}{\mathrm{d}t}P(\ldots,n_i,\ldots,t) = \sum_{\forall i}\mathcal{T}_i^+(\ldots,n_i-1,\ldots,\mathsf{f},t)P(\ldots,n_i-1,\ldots,t) \tag{10.1}$$

$$+ \sum_{\forall i}\mathcal{T}_i^-(\ldots,n_i+1,\ldots,\mathsf{f},t)P(\ldots,n_i+1,\ldots,t) \tag{10.2}$$

$$- \sum_{\forall i}\left[\mathcal{T}_i^+(\ldots,n_i,\ldots,\mathsf{f},t)+\mathcal{T}_i^-(\ldots,n_i,\ldots,\mathsf{f},t)\right]P(\ldots,n_i,\ldots,t)\,, \tag{10.3}$$

where the time t is defined in units of years and $\mathcal{T}_i^+$ and $\mathcal{T}_i^-$ are the transition coefficients for the i-th species, which depend not only on the counts for all species $n_1, n_2, \ldots$, but also (in principle) on a larger feature space f generated by the available data up to time t.

The famous Lotka-Volterra system, with some modficiations for fishing and a larger set of species, would suggest transition coefficients of the form

$$\mathcal{T}_i^+(\ldots, n_i, \ldots, \mathsf{f}, t) = \mathcal{T}_i^+(\ldots, n_i, \ldots) = \Lambda_i(n_i) + n_i \alpha_i \sum_{\forall i' \text{ prey}} n_{i'} \tag{10.4}$$

$$\mathcal{T}_i^-(\ldots, n_i, \ldots, \mathsf{f}, t) = \mathcal{T}_i^-(\ldots, n_i, \ldots) = n_i \mu_i + n_i \gamma_i + n_i \beta_i \sum_{\forall i' \text{ pred}} n_{i'} \,, \tag{10.5}$$

where: $\Lambda_i(n_i) = \tilde{\Lambda}_i n_i e^{-\lambda_i(n_i-1)}$ is the density-dependent birth rate; μ_i is the species death rate; α_i is the increase in the baseline birth rate per fish caused by the increase in prey population; β_i is the rate per fish of predation of the species; and γ_i accounts for the rate of recreational fishing per fish of the species. To approach the present data-driven simulation problem, we're going to generalise this model by training $\mathcal{T}_i^+(\ldots, n_i, \ldots, \mathsf{f}, t)$ and $\mathcal{T}_i^-(\ldots, n_i, \ldots, \mathsf{f}, t)$ directly from the data and generated features.

Look into the likelihood from, e.g., an electrofishing survey such as in Ref. [20]...

$$\mathsf{Likelihood} = \sum_{\text{data}} \mathrm{NB}\big[\mathsf{data}; w_{i,\mathsf{survey}} \langle n_i(t_{\mathsf{data}}) \rangle, k_{i,\mathsf{survey}} \big] \,, \tag{10.6}$$

Part 1

Managing a rugby match

Concept. Building a toy model simulation of a rugby match whose outcome can be manipulated through correctly-timed player substitutions and game management decisions. The dexetera state manipulation framework we have built around the stochadex can meet these requirements, and a dashboard can be created for user interaction. All this combines together to make a simple dashboard game, which we call: 'trywizard'. For the mathematically-inclined, this chapter will motivate the construction of a specific modelling framework for rugby match simulation. For the programmers, the public Git repository for the code described in this chapter can be found here: https://github.com/umbralcalc/trywizard.

11.1 Designing the event simulation engine

Since the basic state manipulation framework and simulation engine will run using dexetera, the mathematical novelties in this project are all in the design of the rugby match model itself. And, as ever, we're not especially keen on spending a lot of time doing detailed data analysis to come up with the most realistic values for the parameters that are dreamed up here. Even though this would also be interesting.[1]

Let's begin by specifying an appropriate state space to live in when simulating a rugby match. It is important at this level that events are defined in quite broadly applicable terms, as it will define the state space available to our stochastic sampler and hence the simulated game will never be allowed to exist outside of it. It seems reasonable to characterise a rugby union match by the following set of states: Penalty, Free Kick (the punitive states); Penalty Goal, Drop Goal, Try (the scoring states); Kick Phase, Run Phase, Knock-on, Scrum, Lineout, Maul and Ruck (the general play states). Using this set of states, in Fig. 11.1 we have summarised our approach to match state transitions into a single event graph. In order to capture the fully detailed range of events that are possible in a real-world match, we've needed to be a little imaginative in how we define the

---

[1]One could do this data analysis, for instance, by scraping player-level performance data from one of the excellent websites that collect live commentary data such as rugbypass.com or espn.co.uk/rugby.

kinds of state transitions which occur. In other words; it's fair to say that our simplified model here represents just a subset of states that a real rugby match could exist in.

Figure 11.1: Simplified event graph of a rugby union match.

In addition to occupying some state in the event graph, the state of a rugby match must also include a binary 'possession' element which encodes which team has the ball at any moment. We should also include the 2-dimensional pitch location of the ball as an element of the match state in order to get a better sense of how likely some state transitions are, e.g., when playing on the edge of the pitch near the touchline it's clearly more likely that a Run Phase is going to result in a Lineout than if the state is currently in the centre of the pitch. To add even more detail, in the next section we will elaborate further by introducing states for each playing position on each side, which encode the playing abilities of each player, their fatigue status and their substitution status. The latter of these keeps track of whether or not the player has been injured or yellow/red-carded during the course of a game and should also enable management strategies to become more nuanced.

Since a rugby match exists in continuous time, it is natural to choose a continuous-time event-based simulation model for our game engine. As we have discussed in previous chapters already, this means we will be characterising transition probabilities of the event graph in Fig. 11.1 by ratios of event rates in time. Recalling our notation in previous chapters, if we consider the current state vector of the match X_t, we can start by assigning each transition $\mathsf{a} \to \mathsf{b}$ on the event graph an associated expected rate of occurance $\lambda_{\mathsf{a} \to \mathsf{b}}$ which is defined in units of continuous time, e.g., seconds. In addition to the transitions displayed on the graph, we can add a 'possession change

transition'; where the possession of the ball in play moves to the opposing team. This transition may occur while the match is also in any of the white-coloured states on the graph (apart from Knock-on which determines a possession change immediately through a Scrum) and let's assign this a state, parameter and timestep-dependent expected rate of occurance $\lambda_{\text{pos}}(X_t, z, t)$.

Based on our dicussion above, an appropriate encoding for the overall game state at timestep index t could be a state vector X_t whose elements are

$$X_t^0 = \begin{cases} 0 & \text{Match State} = \text{Penalty} \\ 1 & \text{Match State} = \text{Free Kick} \\ \dots \end{cases} \tag{11.1}$$

$$X_t^1 = \begin{cases} 0 & \text{Possession} = \text{Home Team} \\ 1 & \text{Possession} = \text{Away Team} \end{cases}. \tag{11.2}$$

But how does this overall game state connect to the event rates? The probabilistic answer is quite straightforward. If the probability of the match state being $X_t^0 = a$ at timestep t is written as $P_t^0(a)$, then the probability of $X_{t+1}^0 = b$ in the following timestep is

$$P_{t+1}^0(b) = \frac{\frac{1}{\tau} P_t^0(b) + \sum_{\forall a \neq b} \lambda_{a \to b} \mathcal{T}_{a \to b}(X_t, z, t) P_t^0(a)}{\left[\frac{1}{\tau} + \sum_{\forall a \neq b} \lambda_{a \to b} \mathcal{T}_{a \to b}(X_t, z, t)\right]}, \tag{11.3}$$

where $\forall a \neq b$ in the summation indicates that all the available transitions from a to b, where $a \neq b$, should be summed over and $\mathcal{T}_{a \to b}(X_t, z, t)$ is a time, parameter and state-dependent transition probability that is determined by the playing tactics of each team as well as the general likelihoods of gameplay which are expected from a real rugby match. Note that in the expression above, we have also defined τ as a timescale short enough such that no transition is likely to occur during that interval. An equivalent to Eq. (11.3) should also apply to the possession change transition rate, i.e., the probability that the Home Team has possession P_t^1 at time t evolves according to

$$P_{t+1}^1 = \frac{\frac{1}{\tau} P_t^1 + \lambda_{\text{pos}}(X_t, z, t)(1 - P_t^1)}{\left[\frac{1}{\tau} + \lambda_{\text{pos}}(X_t, z, t)\right]}. \tag{11.4}$$

Before we move on to other details, it's quite important to recognise that because our process is defined in continuous time, the possession change rate may well vary continuously (this will be especially true when we talk about, e.g., player fatigue). Hence, Eq. (11.3) is only an *approximation* of the true underlying dynamics that we are trying to simulate — and this approximation will only be accurate if τ is small. The reader may recall that we discussed this same issue from the point of view of simulating time-inhomogeneous Poisson processes with the rejection method when we were building the stochadex in an earlier chapter.

While these match state transitions and possession changes are taking place, we also need to come up with a model for how the ball location L_t changes during the course of a game, and as a function of the current game state. Note that, because the ball location is a part of the overall game state, it will be included as information contained within some elements of X_t as well. To make this explicit, we can simply set $X_t^2 = L_t^{\text{lon}}$ and $X_t^3 = L_t^{\text{lat}}$ — where L_t^{lon} denotes the longitudinal component (lengthwise along the pitch) and L_t^{lat} denotes the lateral component (widthwise across the pitch). If we associate every state on the event graph with a single change in spatial location of the ball on the pitch, we then need to construct a process which makes 'jumps' in 2-dimensional

space each time a state transition occurs. To keep things simple and intuitive, we will say that movements of the ball are only allowed to occur during either a Run Phase or a Kick Phase.

In the case of a Run Phase, let's choose the longitudinal component of the ball location $L_{\mathsf{t}}^{\mathrm{lon}}$ to be updated by the difference between samples drawn from two exponential distributions (one associated to each team). Hence, the probability density $P_{\mathsf{t}}(\ell)$ of $L_{\mathsf{t}}^{\mathrm{lon}} = \ell$ at timestep t evolves according to

$$P_{\mathsf{t}+1}(\ell) = \int_0^\infty \mathrm{d}\ell' \, \mathsf{ExponentialPDF}(\ell + \ell'; a_{\mathrm{run}}) \mathsf{ExponentialPDF}(\ell'; d_{\mathrm{run}}) \,, \qquad (11.5)$$

where a_{run} and d_{run} are the exponential distribution scale parameters for an attacking and defending player, respectively, and we have chosen positive values of ℓ to be aligned with the forward direction for the attacking team. We shall elaborate on where a_{run} and d_{run} come from when we discuss associating events for player abilities in due course. If we now consider lateral component of the ball location $L_{\mathsf{t}}^{\mathrm{lat}}$ during a Run Phase; it makes sense that this wouldn't be affected much by either team within the scope of detail in this first version of our model. Hence, the probability density $P_{\mathsf{t}}(w)$ of $L_{\mathsf{t}}^{\mathrm{lat}} = w$ at timestep t can just be updated like so

$$P_{\mathsf{t}+1}(w) = \mathsf{NormalPDF}(w; L_{\mathsf{t}}^{\mathrm{lat}}, \sigma_{\mathrm{run}}^2) \,, \qquad (11.6)$$

where σ_{run} is the typical jump in lateral motion (the standard deviation parameter of the normal distribution).

Turning our attention to the Kick Phase; the longitudinal component is only realistically controlled by the attacking team. There's also an angular component to this motion which is associated to the direction that the kicker is orientating their kick. Therefore, the probability density of $P_{\mathsf{t}}(\ell)$ with $L_{\mathsf{t}}^{\mathrm{lon}} = \ell \cos\theta$ and $L_{\mathsf{t}}^{\mathrm{lat}} = \ell \sin\theta$ at timestep t will evolve according to

$$P_{\mathsf{t}+1}(\ell) = \mathsf{ExponentialPDF}(\ell; a_{\mathrm{kick}}) \,, \qquad (11.7)$$

where a_{kick} this time is the exponential scale parameter for the attacking team player who is kicking and θ is the angle they have chosen to kick at (where straight ahead is defined as $\theta = 0$). Similarly, the lateral component for a Kick Phase is only going to be controlled by the accuracy of the kicker with the angle and longitudinal distance already applied. Therefore, the corresponding update to the probability density $P_{\mathsf{t}}(w)$ where $L_{\mathsf{t}}^{\mathrm{lon}} = w \sin\theta$ and $L_{\mathsf{t}}^{\mathrm{lat}} = w \cos\theta$ at timestep t will look like

$$P_{\mathsf{t}+1}(w) = \mathsf{NormalPDF}(w; L_{\mathsf{t}}^{\mathrm{lat}} + \ell \sin\theta, \sigma_{\mathrm{kick}}^2) \,, \qquad (11.8)$$

with a standard deviation σ_{kick}, and its interpretation now being the accuracy of the kicker.

11.2   Associating events to player states and abilities

In the last section we introduced a continuous-time event-based simulation model for a rugby union match. In this section we are going to add more detail into this model by inventing how to associate specific player states and abilities to the event rates of the simulation. Before continuing, we want to reiterate that this model is entirely made up and, while we hope it illustrates some interesting mathematical modelling ideas in the context of rugby, there's no particular reason why a statistical inference with a reliable dataset should prefer our model to others which may exist.

In Fig. 11.2 we began by separating playing positions on the rugby field into their usual descriptions and then associating each player type with a short list of simplified attributes. Our model is then to associate a player with an 'possession attacking' and 'possession defending' ability which corresponds to each of their attributes. For example, a Front Row Forward will have 8 abilities associated to them: 2 for each of their Scrum, Lineout, Ruck and Maul attributes.
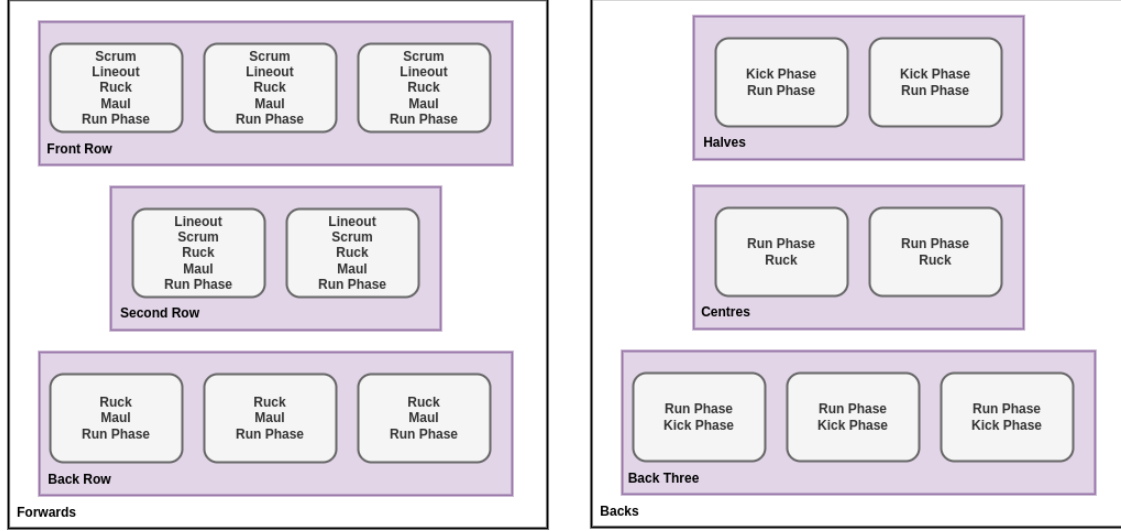

Figure 11.2: Associated playing abilities for each position type.

Let's now create a vector-valued function $a_{\mathrm{pos}}(X_{\mathsf{t}}, z)$ which returns all of the posession attacking attributes (which are all stored in z) that are associated to match state X_{t}^0 and an analogous one $d_{\mathrm{pos}}(X_{\mathsf{t}}, z)$ for the possession defending attributes. The dependencies of these functions on the ball possession state X_{t}^1 comes from the fact that when, e.g., the Home Team has possession of the ball it will be their possession attacking attributes that are returned by $a_{\mathrm{pos}}(X_{\mathsf{t}}, z)$ and the Away Team's possession defending attributes that are returned by $d_{\mathrm{pos}}(X_{\mathsf{t}}, z)$.

In order to model the effect of player fatigue over the course of a match, we can include some vectors of player fatigue values f and times at which each player started playing t_{start} into a formula for the decline of each attribute. We can then assign these values to be part of the overall match state like so

$$X_{\mathsf{t}}^j = a_{\mathrm{pos}}^j(X_{\mathsf{t}}, z)e^{-f^j[t(\mathsf{t}) - t_{\mathrm{start}}^j]} \tag{11.9}$$

$$X_{\mathsf{t}}^k = d_{\mathrm{pos}}^k(X_{\mathsf{t}}, z)e^{-f^k[t(\mathsf{t}) - t_{\mathrm{start}}^k]}, \tag{11.10}$$

where the j and k each index the range of indices in the overall match state which correspond to the attacking and defending team player states, respectively.

So how does each player affect the events of a match? In our model, we would argue that players should be able to directly influence the possession change rate $\lambda_{\mathrm{pos}}(X_{\mathsf{t}}, z, \mathsf{t})$ through a balance of

attacking and defensive attributes in the following relation

$$\lambda_{\text{pos}}(X_{\text{t}}, z, \text{t}) = \frac{\lambda_{\text{pos}}^* \sum_{\forall k} X_{\text{t}}^j}{\sum_{\forall k} X_{\text{t}}^j + \sum_{\forall j} X_{\text{t}}^k} \,, \tag{11.11}$$

where λ_{pos}^* is the maximum rate that is physically possible and the $\forall j$ and $\forall k$ in the summations indicate summing over all attacking and defending player attributes, respectively. In addition to this possession change influence, players who have Run Phase and Kick Phase attributes may affect the gain in distance (and accuracy in the case of Kick Phase) that each state translates to on the pitch.

Let's first describe how we intend the Run Phase to work. Every time the match state transitions into a Run Phase, an individual player on the attacking side is chosen at random (uniformly across the team[2]) to be the the nominal 'attacker'. At the same time, an individual player on the defending side is chosen at random (again, uniformly across the team) to be the the nominal 'defender'. Once these players have been chosen (and hence the a_{run} and d_{run} parameters have been determined), the longitudinal motion update we described in Eq. (11.5) can then be applied. Note that the a_{run} and d_{run} parameters should also receive a fatigue decrement depending on the time that each player has remained on the pitch, much like the exponential factors we applied in Eqs. (11.9) and (11.10).

Finally, we turn or attention to the mechanics of a Kick Phase. For tactical kicking in the middle of play, this operates in a similar way as a Run Phase; one of the players on the attacking side (the side in possession of the ball) is chosen at random (uniformly) and their a_{kick} and σ_{kick} attributes are used in Eqs. (11.7) and (11.8) to update the position of the ball. In the specific case of Goal kicking from a Penalty, the designated place kicker on each side also uses their same Kick Phase attributes to determine the success/failure of the kick at the posts.

11.3   Deciding on managerial actions

So how does managing a rugby match map to measured states and taking actions? To begin with, it makes sense to claim that the measurement of the total match state is perfect in this instance, and hence $\mathcal{S}_{\text{t}} = X_{\text{t}}$. We then have to decide between what sorts of managerial actions are parametric in nature (and hence affect z) and which ones directly act on the state of the system itself.

Our model structure would suggest that the only way in which a manager can influence the state of a match is through modifying the parameters which go into the $a_{\text{pos}}(X_{\text{t}}, z)$, $d_{\text{pos}}(X_{\text{t}}, z)$ or $\mathcal{T}_{\text{a}\to\text{b}}(X_{\text{t}}, z, \text{t})$ functions. Additionally, we would argue that the parameters a_{run}, d_{run}, a_{kick} and σ_{kick} in Eqs. (11.5), (11.7) and (11.8) can also be determined by the manager to some extent through player selection/substitutions.

11.4   Writing the game itself

- Show which stochadex/dexetera methods were called and how they were used to simulate the game.

- Give a summary of how the dashboard backend works (diagram would help) and how this connects up to the streamlit frontend via protobuf messages.

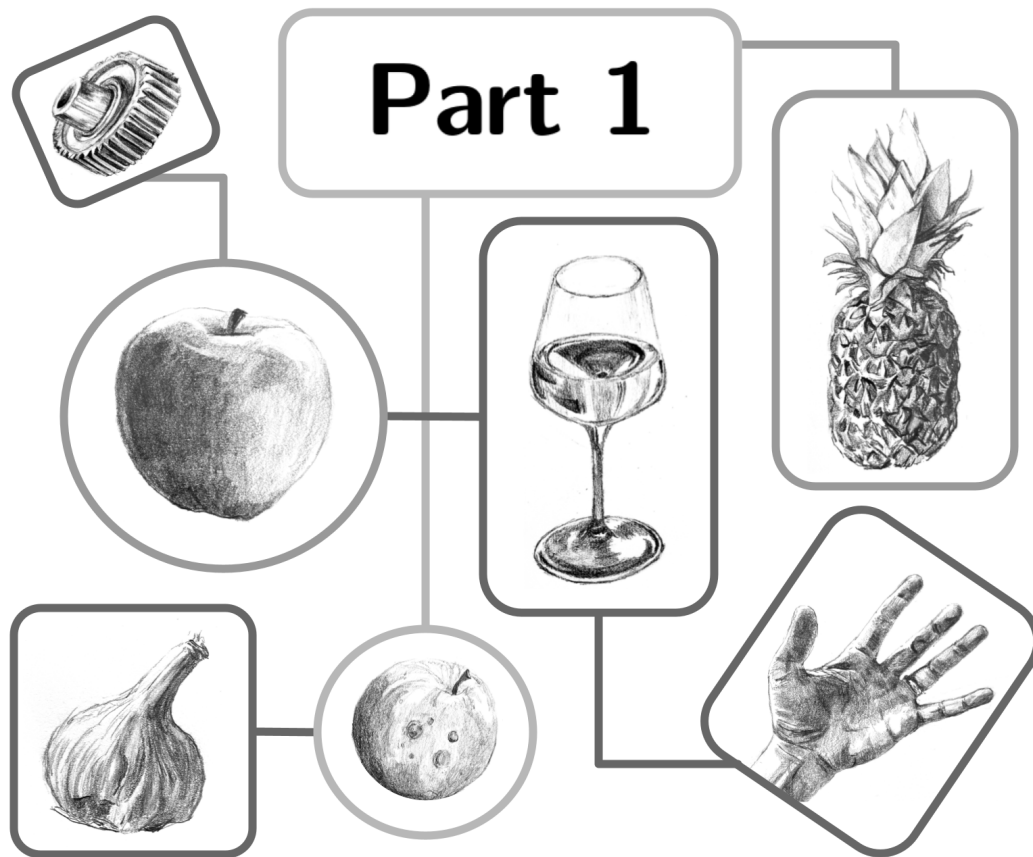[2]This uniform sampling could be refined later to associate sampling probabilities with game state and player roles.

Part 1

Influencing house prices

Concept. The idea here is

Part 1

Part 4. How do we then optimise the answer to Part 3 to achieve a specified control objective?
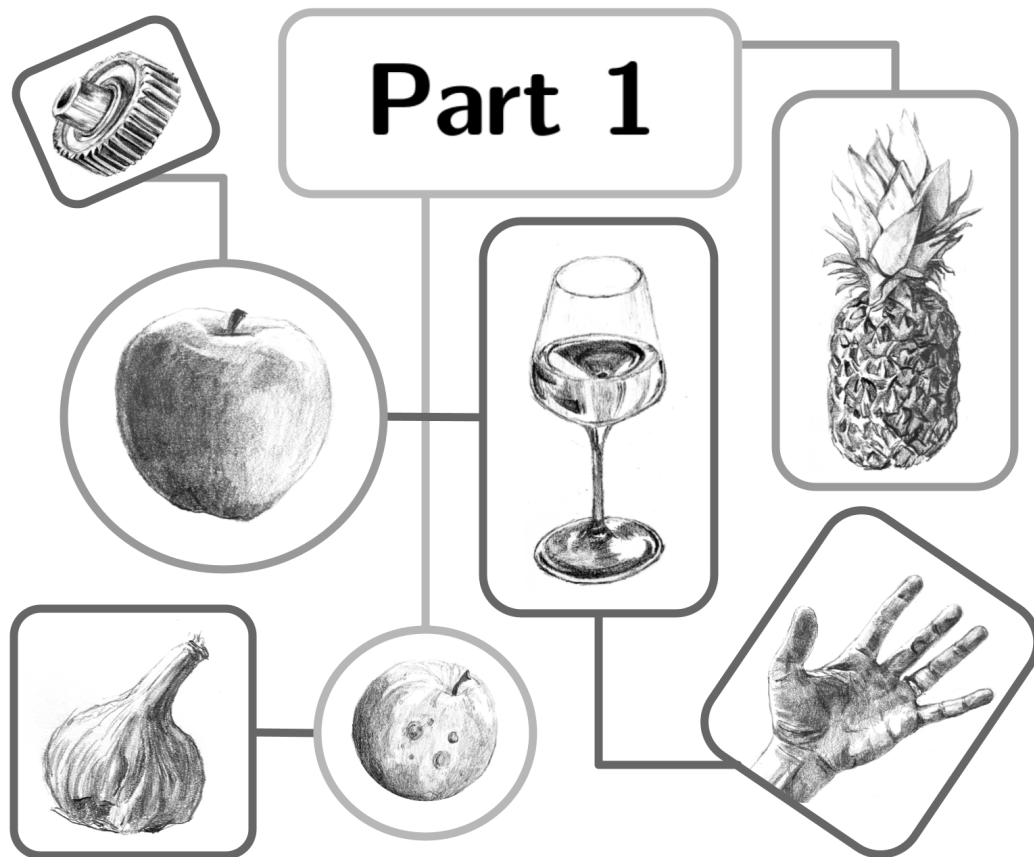
# Optimising actions for control objectives

Concept. The idea here is

13.1 States, actions and formulating rewards

We observe the state at timestep t with $\mathcal{S}_\mathsf{t} = M_\mathsf{t}(X', Z_\mathsf{t}, \mathsf{t})$. In a Markov Decision Process (MDP), based on this observation alone, we would then take actions $X_{\mathsf{t}+1} = \mathcal{F}_{\mathsf{t}+1}(X', Z_\mathsf{t}, \mathcal{A}_\mathsf{t}, \mathsf{t})$, for which we would later attribute reward $\mathcal{R}_\mathsf{t}$.

Part 1

Resource allocation for epidemics

Concept. The idea here is to limit the spread of some abstract epidemic through the correct time-dependent resource allocation.

Part 1

15
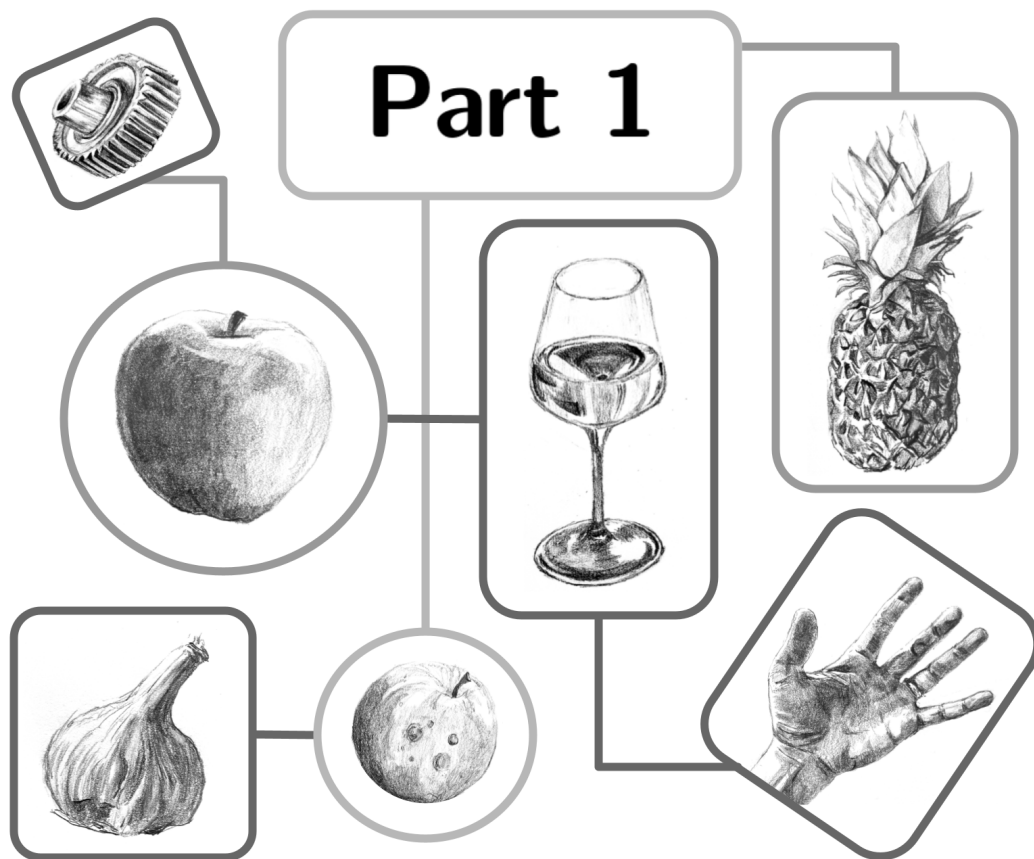
Quantum system control

Concept. The idea here is to follow stuff along these lines here.

Bibliography

[1] "The Go Programming Language." https://go.dev/.

[2] "The Python Programming Language." https://www.python.org/.

[3] "The Diffusing Ideas GitHub Repository."
    https://github.com/umbralcalc/diffusing-ideas.

[4] "Open Source Initiative: MIT License." https://opensource.org/licenses/MIT.

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

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

[7] 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.

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

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

[10] 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.

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

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

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

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

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

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

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

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

[19] H. Ye, R. J. Beamish, S. M. Glaser, S. C. Grant, C.-h. Hsieh, L. J. Richards et al., *Equation-free mechanistic ecosystem forecasting using empirical dynamic modeling*, *Proceedings of the National Academy of Sciences* **112** (2015) E1569–E1576.

[20] "Electrofishing to assess a river's health." `https://environmentagency.blog.gov.uk/2015/10/29/electrofishing-to-assess-a-rivers-health/`.