Diffusing Ideas

Software, noise and building mathematical toys

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

August 25, 2023

Shared by the authors under an MIT License.
The code to compile this book is open source and can be found in this repository:
https://github.com/umbralcalc/diffusing-ideas.

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 TypeScript [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 pseudocode diagrams 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

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 $X' \to X$

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

where: i is an index for the dimensions of the 'state' space; $\mathtt{t}$ 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 $F^i_{\mathtt{t}+1}(X', z, \mathtt{t})$ as the latest element of an arbitrary matrix-valued function. As we shall discuss shortly, $F^i_{\mathtt{t}+1}(X', z, \mathtt{t})$ may represent not just operations on deterministic variables, but also on stochastic ones. There is also no requirement for the function to be continuous.

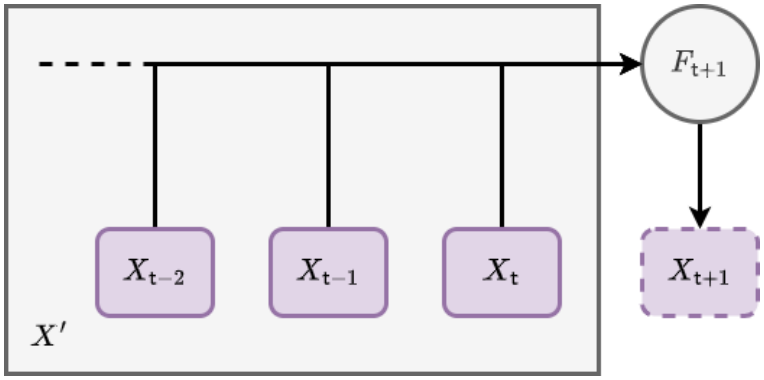

Figure 1.1: Graph representation of Eq. (1.1).

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 pseudocode you could easily write something with the same idea in it, and it would probably look something like the method diagram in Fig. 1.2.

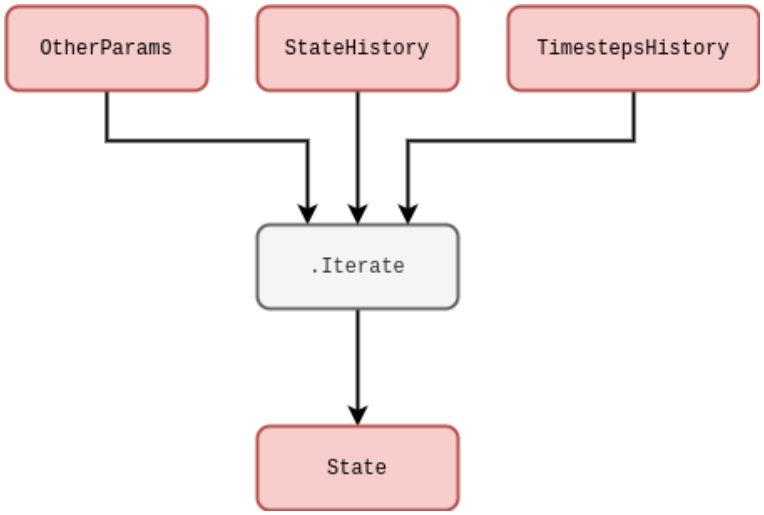

Figure 1.2: Pseudocode representation of Eq. (1.1).

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

[1] Or memory at least within some window.

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

For more complex physical models and integrators, the distinct notions of 'numerical timestep' and 'total elapsed continuous time' will crop up quite frequently. Hence, before moving on further details, it will be important to define the total elapsed time variable $t(\mathtt{t})$ 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 $\mathtt{t}-1 \to \mathtt{t}$, we will always be able to compute the total elapsed time through the relation

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

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 $\delta t(\mathtt{t})$ function and summing over it we can enable this flexibility in the computation. In case the summation notation is no fun for programmers; we're simply adding up all of the differences in time to get a total. We've illustrated this in Fig. 1.3 for more clarity.

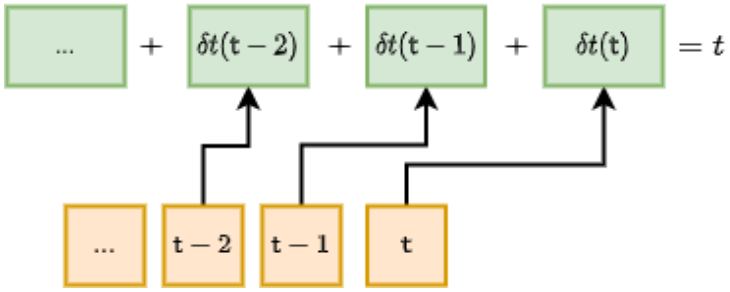

Figure 1.3: Illustration of Eq. (1.2).

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

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

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

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

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 pseudocode, the Wiener process is schematically represented by Fig. 1.4.

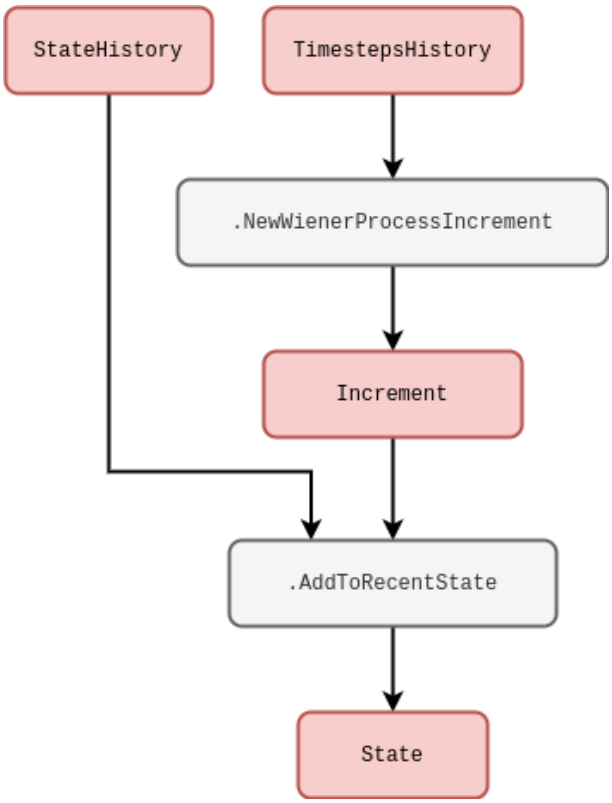

Figure 1.4: Schematic of code for a Wiener process.

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

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

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' [5, 6, 7]. The

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

pseudocode for any of these should hoepfully be fairly straightforward to deduce based on the lines we've already written above.

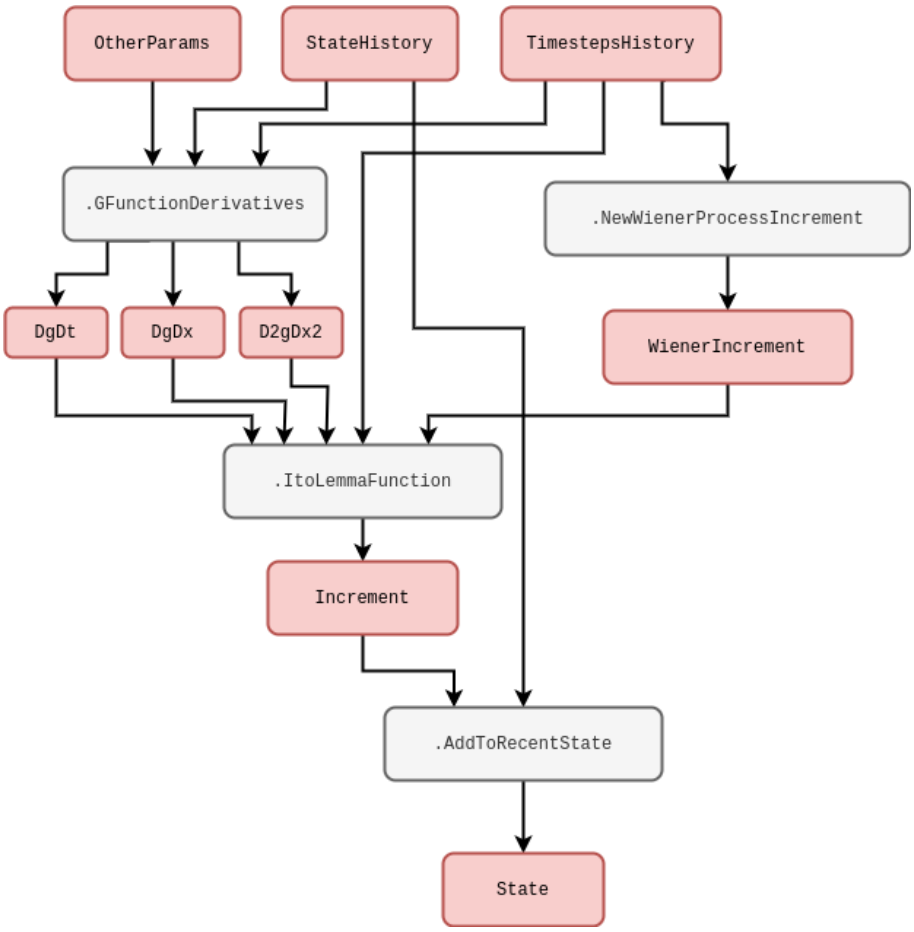

Figure 1.5: Schematic of code for Eq. (1.8).

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

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

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

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

can still be made a bit more meaningful to the programmer in this way. The pseudocode in general would look something like Fig. 1.5.

Let's now look at a more complicated type of noise. For example, we might consider sampling from a *fractional Brownian motion* process $[B_H]_{\mathtt{t}}$, 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

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

where $S^0_{\mathtt{t}+1}(X', z, \mathtt{t}) = [B_H]_{\mathtt{t}+1} - [B_H]_{\mathtt{t}}$. 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 computational form of this integral is illustrated in Fig. 1.6 to try and disentangle some of the mathematics as a program.

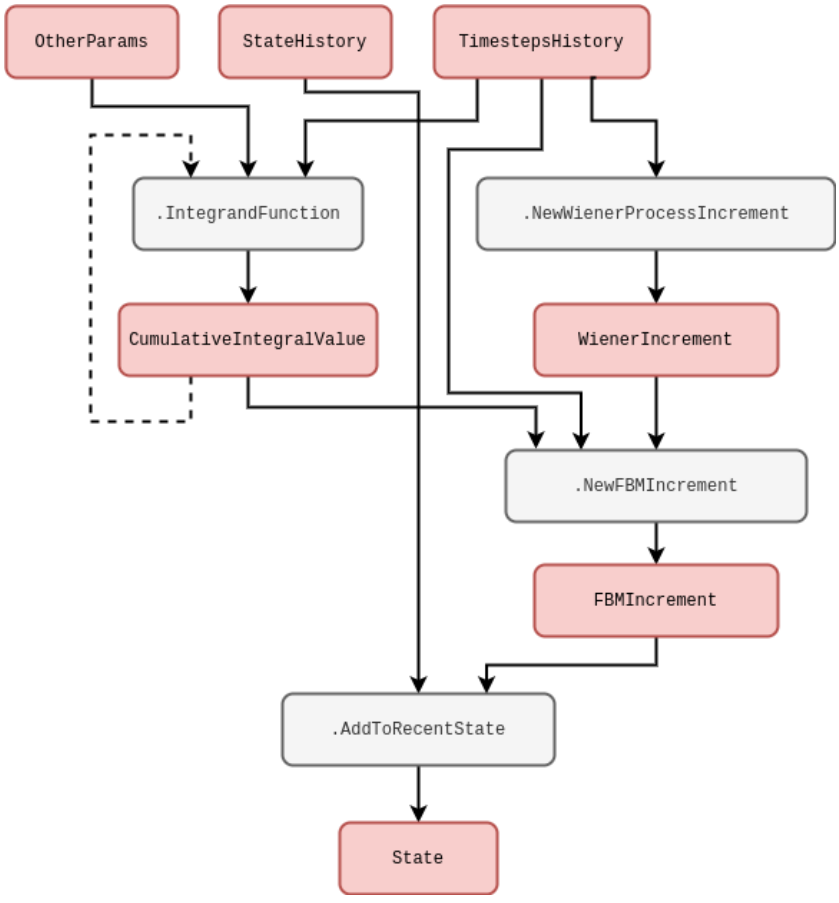

Figure 1.6: Schematic of code for Eq. (1.9).

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

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

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

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

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*. Fig. 1.7 may also help to understand this concept from the programmer's perspective.

There are a few extensions to the simple Poisson process that introduce additional stochastic processes. *Cox (doubly-stochastic) processes*, for instance, are basically where we replace the time-dependent rate $\lambda(\mathsf{t})$ with independent samples from some other stochastic process $\Lambda(\mathsf{t})$. For example, a Neyman-Scott process [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

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

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

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

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

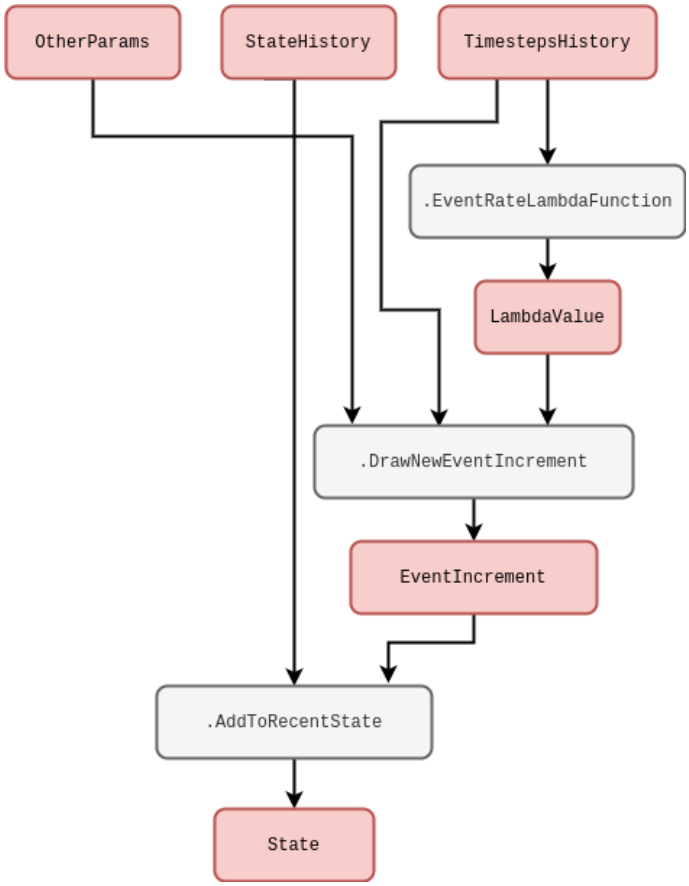

Figure 1.7: Schematic of code for an inhomogeneous Poisson process.

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 pseudocode we wrote previously would be sufficient to simulate this process with one tweak: add into the `DrawNewEventIncrement` function the calling of a function which generates the $[J_\lambda]^i_{\mathtt{t}}$ samples and output these if the event occurs.

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

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

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

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

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

where γ is the 'exciting kernel' and μ is some constant background rate. In order to simulate a Hawkes process using our formalism, the pseudocode would be something like Fig. 1.8.

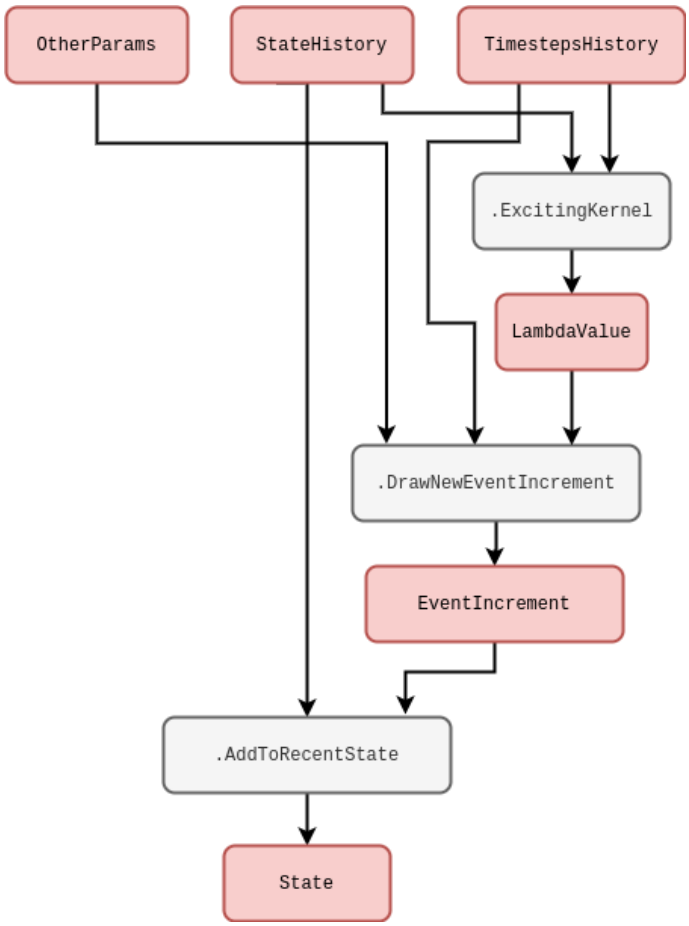

Figure 1.8: Schematic of code for a Hawkes process.

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

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

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

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

1.2 Software design

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

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

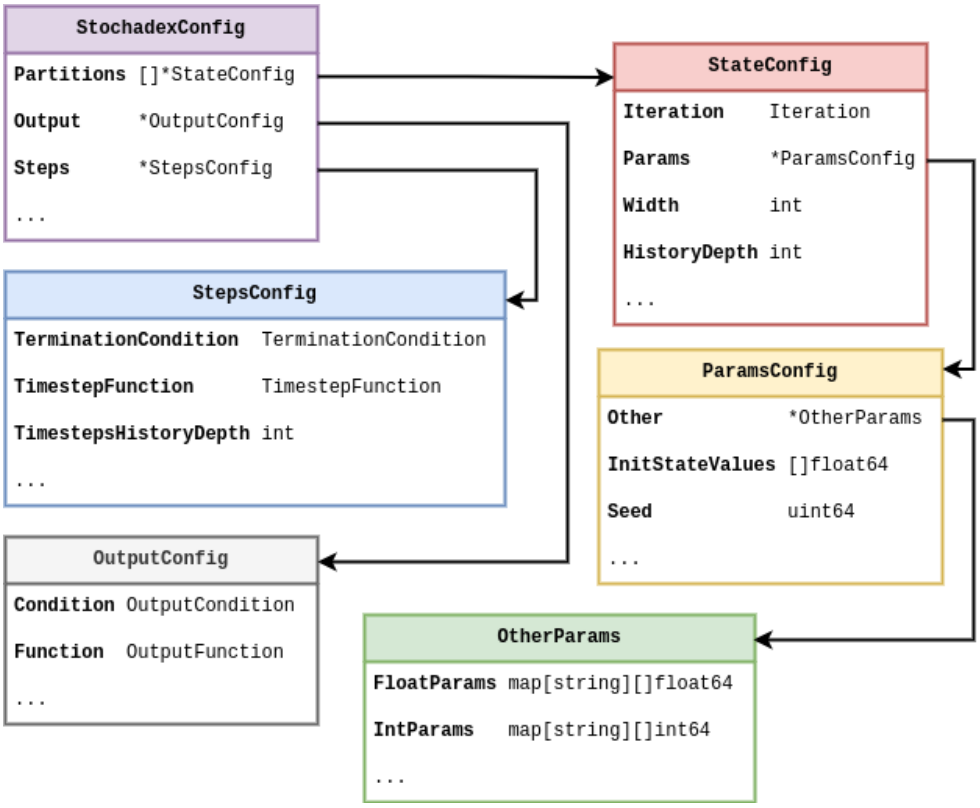

Figure 1.9: A relational summary of the configuration data types in the stochadex.

Once the user has written the code to create these functions for the stochadex, we want to

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

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

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

It's clear that in order to simulate Eq. (1.1), we need an interative algorithm which reapplies a user-specified function to the continually-updated history. But let's now return to the point we made earlier about how the performance of such an algorithm will depend on the size of the state history matrix X. The key bit of the algorithm design that isn't so straightforward is: how do we sucessfully split this state history up into separate partitions in memory while still enabling them to communicate effectively with each other? Other generalised simulation frameworks — such as SimPy [12], StoSpa [13] and FLAME GPU [14] — have all approached this problem in different ways, and with different software architectures.

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

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

2. The `StateIterator` in each goroutine executes the iteration and stores the resulting state update in a variable.

3. Once all of the iterations have been completed, the `PartitionCoordinator` then requests each goroutine to update its relevant partition of the state history by sending another `IteratorInputMessage` to each.

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

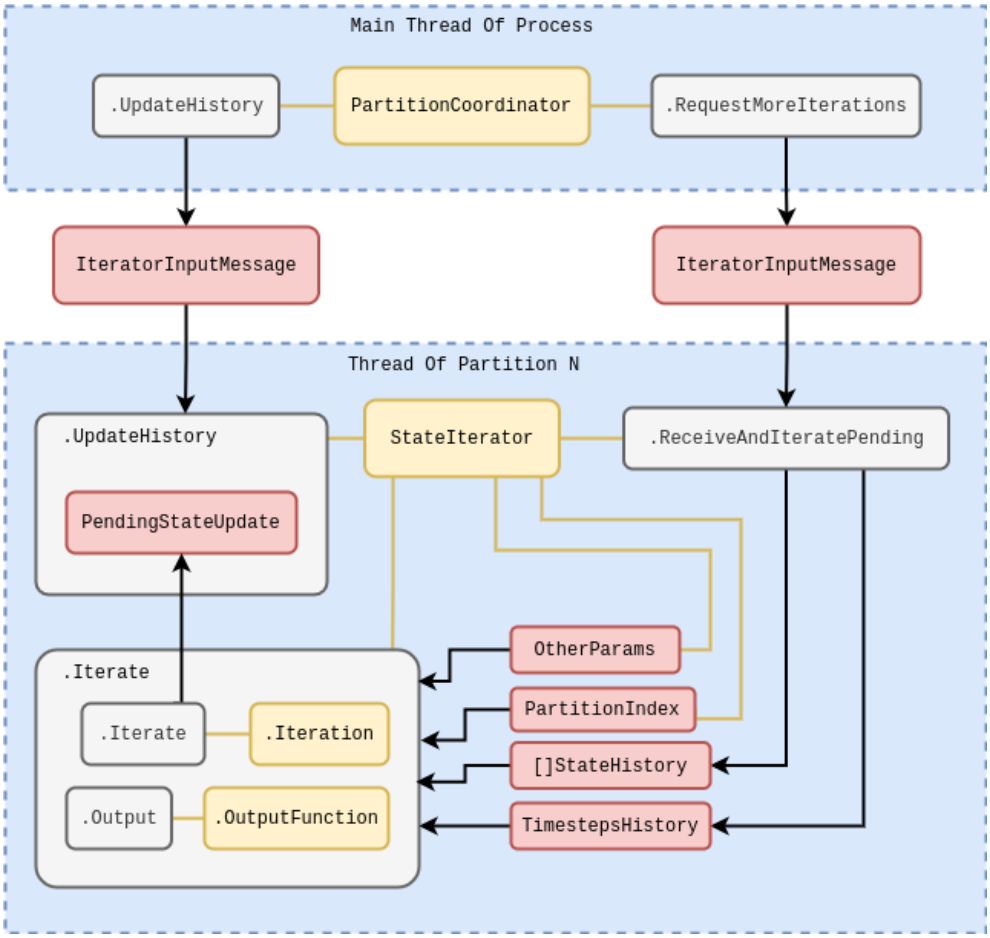

Figure 1.10: Schematic for a step of the stochadex simulation algorithm.

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

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

In order to debug the simulation code and gain a more intuitive understanding of the outputs from a model as it is being developed, we have also written a lightweight frontend dashboard

React [15] app in TypeScript to visualise any stochadex simulation as it is running. This dashboard can be launched by passing config at runtime to the main stochadex executable, and we have illustrated how all this fits together in a flowchart shown in Fig. 1.11.

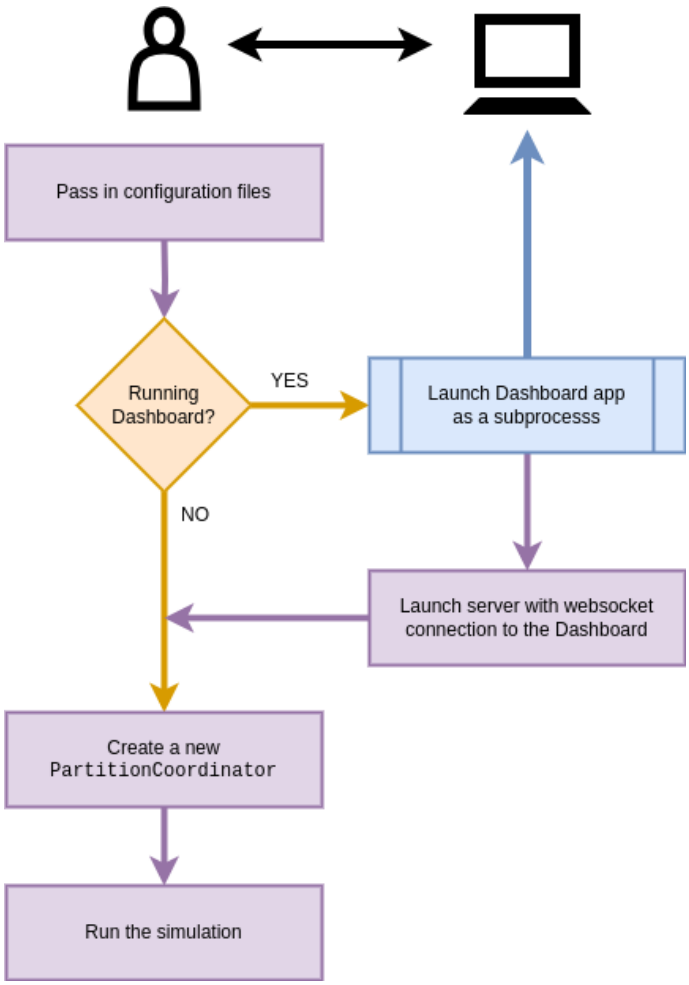

Figure 1.11: A diagram of the main stochadex binary executable.

2

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

3

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

4

Empirical probabilistic filtering

Concept. To extend the formalism that we developed in previous chapters to enable the empirical emulation of real-world data via a probabilistic filter. This technique should enable a researcher to model complex dynamical trends in the data very well; at the cost of making the abstract interpretation of the model less immediately comprehensible than the statistical inference models in some proceeding chapters. As our generalised framework applies to a wide variety stochastic phenomena, our filtering algorithm will be applicable to a great breadth of data modeling problems as well. We will also explore some examples which illustrate how the filter should be applied in practice and then follow this up with how the code is designed and implemented as part of a new software package called the 'learnadex'. For the mathematically-inclined, this chapter will take a detailed look at how our formalism can be extended to focus on probabilistic filters and their optimisation using real-world data. For the programmers, the software described in this chapter lives in the public Git repository: https://github.com/umbralcalc/learnadex.

4.1 Probabilistic formalism

The key distinction between the methods that we will develop in this chapter and the ones in the proceeding chapters is in their utility when faced with the problem of attempting to model real-world data. In the proceeding chapter, we shall describe some powerful techniques that can be used most effectively when the researcher is aware of the family of models that generated the data. In the present chapter, we will go into the details of how a more 'empirical' approach can be derived for dynamical process modeling in a probabilistic framework which locally adapts the model to the data through time.

While we think that it's worth going into some mathematical detail to give a better sense of where our formalism comes from; we want to emphasise that the framework we discuss here is not new to the technical literature at all. Our overall framework draws on influences from Empirical Dynamical Modeling (EDM) [16], some classic nonparametric local regression techniques — such as LOWESS/Savitzky-Golay filtering [17] — and also Gaussian processes [18] as well. The novelties here, instead, lie more in the specifics of how we combine some of these ideas together when

referencing the stochadex formalism, and how this manifests in designing more generally-applicable software for the user.

Before we are able to develop this empirical filtering algorithm, we need to return to the stochadex formalism that we introduced in the first chapter of this book. As we discussed at that point; this formalism is appropriate for sampling from nearly every stochastic phenomenon that one can think of. However, when trying robustly assess how far a model is from accurately describing a set of real-world data, trying to use only generated samples of the model process can be diffcult. Instead, in this section, we are going to extend this formalism to look at how probability theory can help with this data comparison problem in a systematic way.

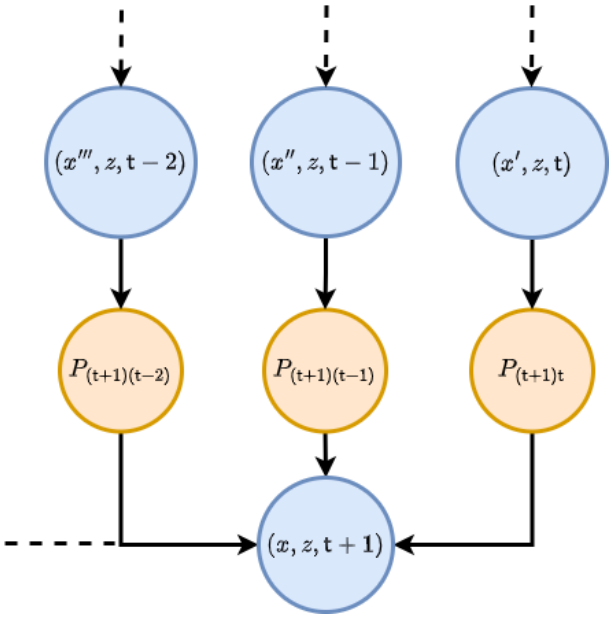

Figure 4.1: Graph representation of Eq. (4.1).

So, how do we begin? In the first chapter, we defined the general stochastic process with the formula $X^i_{\mathrm{t}+1} = F^i_{\mathrm{t}+1}(X', z, \mathrm{t})$. This equation also has an implicit *master equation* associated to it that fully describes the time evolution of the *probability density function* $P_{\mathrm{t}+1}(x|z)$ of the next matrix row being $x = X_{\mathrm{t}+1}$ given that the parameters of the process are z. This can be written as

$$P_{\mathrm{t}+1}(x|z) = \frac{1}{\mathrm{t}} \sum_{\mathrm{t}'=0}^{\mathrm{t}} \int_{\omega_{\mathrm{t}'}} \mathrm{d}x' P_{\mathrm{t}'}(x'|z) P_{(\mathrm{t}+1)\mathrm{t}'}(x|x', z)\,, \tag{4.1}$$

where at the moment we are assuming the state space is continuous in each dimension and $P_{(\mathrm{t}+1)\mathrm{t}'}(x|x', z)$ is the conditional probability that the matrix row at time $(\mathrm{t}+1)$ will be $x = X_{\mathrm{t}+1}$ given that the row at time t' was $x' = X_{\mathrm{t}'}$ and the parameters of the process were z. 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. 4.1.

The factor of $1/\mathsf{t}$ in Eq. (4.1) 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

$$\frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \to \frac{1}{t(\mathsf{t})}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \delta t(\mathsf{t}')\,. \tag{4.2}$$

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.

If we wanted to compute the mean of the distribution $M_{\mathsf{t}+1}(z)$ in Eq. (4.1), it would be straightforward to just multiply both sides of the expression by x and integrate over $\mathrm{d}x$ in the $\omega_{\mathsf{t}+1}$ domain. However, there is another similar expression for the mean that we can derive under certain conditions which will be valuable to us when developing the empirical filter. If the probability distribution is *stationary* — meaning that $P_{\mathsf{t}'}(x|z) = P_{\mathsf{t}''}(x|z)$ for all t' and t'' — it's possible to derive[1]

$$M_{\mathsf{t}+1}(z) = \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, x\, P_{\mathsf{t}+1}(x|z) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}x' \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, x'\, P_{\mathsf{t}'}(x'|z) P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)\,. \tag{4.3}$$

The standard covariance matrix elements can also be computed in a similar fashion

$$\begin{aligned} C^{ij}_{\mathsf{t}+1}(z) &= \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, [x - M_{\mathsf{t}+1}(z)]^i [x - M_{\mathsf{t}+1}(z)]^j P_{\mathsf{t}+1}(x|z) \\ &= \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}x' \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, [x' - M_{\mathsf{t}+1}(z)]^i [x' - M_{\mathsf{t}+1}(z)]^j P_{\mathsf{t}'}(x'|z) P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)\,. \end{aligned} \tag{4.4}$$

While they look quite abstract, Eqs. (4.3) and (4.4) express the core idea behind how our probabilistic filter will function. By assuming a stationary distribution, we gain the ability to directly estimate the statistics of the probability distribution $P_{\mathsf{t}+1}(x|z)$ from past samples it may have in empirical data; which are represented here by $P_{\mathsf{t}'}(x'|z)$.

In order to study higher-order out-of-time-order correlations, we can also consider using the statistical moments computed from joint distributions like these

$$P_{(\mathsf{t}+1)\mathsf{t}'}(x, x'|z) = P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z) P_{\mathsf{t}'}(x'|z) \tag{4.5}$$

$$P_{(\mathsf{t}+1)\mathsf{t}'\mathsf{t}''}(x, x', x''|z) = P_{(\mathsf{t}+1)\mathsf{t}'\mathsf{t}''}(x|x', x'', z) P_{\mathsf{t}'\mathsf{t}''}(x'|x'', z) P_{\mathsf{t}''}(x''|z)\,. \tag{4.6}$$

For example, Eq. (4.5) would apply if we wanted to retrieve the out-of-time-order pairwise correlations between x at timestep $\mathsf{t}+1$ and x' at timestep t'. Using this pairwise relationship we can also derive an equation for the out-of-time-order covariance matrix elements

$$C^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(z) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}x' \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, [x - M_{\mathsf{t}+1}(z)]^i [x' - M_{\mathsf{t}'}(z)]^j P_{(\mathsf{t}+1)\mathsf{t}'}(x, x'|z)\,. \tag{4.7}$$

[1] To see that this is true, first note that the joint distribution $P_{(\mathsf{t}+1)\mathsf{t}'}(x, x'|z) = P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z) P_{\mathsf{t}'}(x'|z)$. Secondly, note that joint distributions always allow variable swaps trivially like this $P_{(\mathsf{t}+1)\mathsf{t}'}(x, x'|z) = P_{\mathsf{t}'(\mathsf{t}+1)}(x', x|z)$. Then, lastly, note that stationarity of $P_{\mathsf{t}+1}(x|z) = P_{\mathsf{t}'}(x|z)$ means

$$\int \mathrm{d}x \int \mathrm{d}x'\, x\, P_{(\mathsf{t}+1)\mathsf{t}'}(x, x'|z) = \int \mathrm{d}x \int \mathrm{d}x'\, x\, P_{\mathsf{t}'(\mathsf{t}+1)}(x, x'|z) = \int \mathrm{d}x \int \mathrm{d}x'\, x\, P_{(\mathsf{t}+1)\mathsf{t}'}(x', x|z)\,,$$

where we've used the trivial variable swap to get to the last equality, and the domain references ω in the integrals are implicitly defined.

What other processes can be described by Eq. (4.1)? For Markovian phenomena, the equation no longer depends on timesteps older than the immediately previous one, hence the expression reduces to just

$$P_{\mathsf{t}+1}(x|z) = \int_{\omega_{\mathsf{t}}} \mathrm{d}x' P_{\mathsf{t}}(x'|z) P_{(\mathsf{t}+1)\mathsf{t}}(x|x', z)\,. \tag{4.8}$$

An analog of Eq. (4.1) 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

$$P_{\mathsf{t}+1}(x|z) = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{\omega_{\mathsf{t}'}} P_{\mathsf{t}'}(x'|z) P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)\,, \tag{4.9}$$

where we note that the P's in the expression above all now refer to *probability mass functions*. In the even-simpler case where x is just a vector of binary 'on' or 'off' states, Eq. (4.9) reduces to

$$\begin{aligned} P^i_{\mathsf{t}+1}(z) &= \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{j=1}^{d} P^j_{\mathsf{t}'}(z) P^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(z) \\ &= \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{j=1}^{d} \left\{ P^j_{\mathsf{t}'}(z) A^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(z) + [1 - P^j_{\mathsf{t}'}(z)] B^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(z) \right\}, \end{aligned} \tag{4.10}$$

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

In this section, we looked into how the mathematical formalism used in the stochadex could be extended with probability theory. Now that we have more of a sense of how this formalism works, we are ready to move on to designing the algorithm for our filter. So let's go!

4.2 Generalised filtering algorithm

The empirical probabilistic filtering algorithm that we're going to describe in this section will depend on the stationarity of $P_{\mathsf{t}+1}(x|z) = P_{\mathsf{t}'}(x|z)$ such that, e.g., Eq. (4.3) is applicable. But before we think about the various kinds of conditional probability we could use in the filter, we need to connect values of x in our filter to the data from which it will be learning.

If the mean is a sufficient statistic for the distribution which describes the data, a choice of, e.g., Exponential, Poisson or Binomial distribution could be used where the mean is estimated directly from the time series using Eq. (4.3), given a conditional probability $P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)$. Extending this idea further to include distributions which also require a variance to be known, e.g., the Normal, Gamma or Negative Binomial distributions could be used where the variance (and/or covariance) could be estimated using Eq. (4.4). These are just a few simple examples of distributions that can link the estimated statistics from Eqs. (4.3) and (4.4) to a time series dataset. However, the algorithmic framework is very general to whatever choice of 'data linking' distribution that a researcher might need.

We should probably make what we've just said a little more mathematically concrete. Let's define a data vector $y = Y_{\mathsf{t}+1}$ at time $\mathsf{t}+1$, which shares the same length as x, and represents

a specific observation of x at this point in a real dataset. At this point in time, we can define $P_{t+1}[y{=}Y_{t+1}; M_{t+1}(z), C_{t+1}(z), \dots]$ as representing the likelihood of $y = Y_{t+1}$ given the estimated statistics from Eqs. (4.3) and (4.4) (and maybe higher-orders). Note that in order to do this, we need to identify the x' and $\mathtt{t}'$ values that are used to estimate, e.g., $M_{t+1}(z)$ with the past data values which are observed in the dataset time series itself. Now that we have this likelihood, we can immediately evaluate an objective function (a cumulative log-likelihood) that we might seek to optimise over for a given dataset

$$\ln \mathcal{L}_{t+1}(Y|z) = \ln P_{t+1}[y{=}Y_{t+1}; M_{t+1}(z), C_{t+1}(z), \dots] + \ln P_{t}[y{=}Y_{t}; M_{t}(z), C_{t}(z), \dots] + \dots, \quad (4.11)$$

where the summation continues until all of the past measurements $Y_{t+1}, Y_t, \dots$ which exist as rows in the data matrix Y have been taken into account.

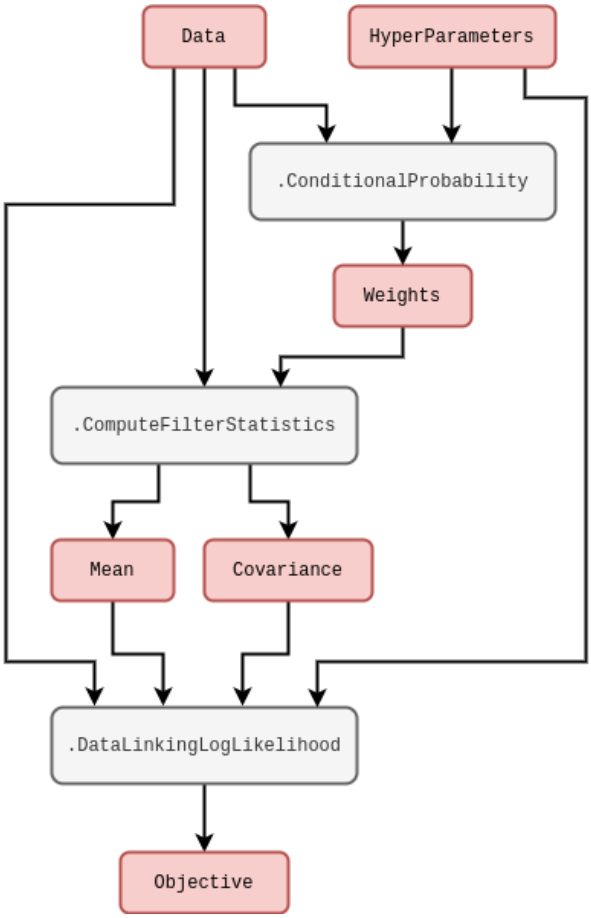

Figure 4.2: Code schematic of the probability filter optimisation.

By optimising Eq. (4.11) with respect to all of the hyperparameters z, we are effectively adopting what is known as an 'empirical Bayes' approach [18, 19]. We might think of the overall generalised filtering algorithm as the code schematic in Fig. 4.2.

In order to specify what $P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)$ is, it's quite natural to define a set of hyperparameters for the elements of z. For example; one clear, and generally applicable, option for the conditional probability is that of a Gaussian process

$$P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z) = \frac{\exp\left\{-\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}[x - f(\mathsf{t}+1)]^i \left[K^{-1}(\theta, \mathsf{t}+1, \mathsf{t}')\right]^{ij}[x' - f(\mathsf{t}')]^j\right\}}{|2\pi K(\theta, \mathsf{t}+1, \mathsf{t}')|^{\frac{1}{2}}}, \quad (4.12)$$

where K is a covariance matrix; f is a function in time and θ represents a set of hyperparameters which are both typically learned by optimisation of the corresponding likelihood function of the data; and $|A|$ denotes computing the determinant of matrix A. In the expression above, both f and θ represent sets of parameters which can exist in z.

In the case of Eq. (4.12) above, the hyperparameters that would be optimised will not just include θ; but also a set of values for f at different timesteps. This would essentially equate our optimised filter to be the equivalent to the maximum a posteriori (MAP) of the Gaussian process in this instance. The only difference here is that by passing in sufficient statistics of this process to some data-linking distribution, we enable many different kinds of data to be described by the same underlying conditional probability-based filter. A Gaussian process also only represents one choice in the unlimited number of possible filters we are able to try in the very simple framework we're describing.

As another form of flexibility, one might also wish to try adapting the data-linking distributions to include an intercept term and linear coefficients for the statistics which are passed to it. These could hence be treated as additional hyperparameters and optimised jointly with the others if there is sufficient constraining power in the data.

The optimisation approach that one chooses to use for obtaining the best hyperparameters in the conditional probability of Eq. (4.11) will depend on a few factors. For example, if the number of hyperparameters is relatively low, but their gradients are difficult to calculate exactly; then a gradient-free optimiser (such as the Nelder-Mead [20] method or something like a particle swarm [21, 22]) would likely be the most effective choice. On the other hand, when the number of hyperparameters ends up being relatively large, it's usually quite desriable to utilise the gradients in algorithms like vanilla Stochastic Gradient Descent [23] (SGD) or Adam [24].

If the gradients of Eq. (4.11) are needed, we can always factorise each derivative with respect to hyperparameter z^i in the following way through the chain rule

$$\begin{aligned}\frac{\partial}{\partial z^i}\ln\mathcal{L}_{\mathsf{t}+1}(Y|z) = &\frac{\partial M_{\mathsf{t}+1}}{\partial z^i}\frac{\partial}{\partial M_{\mathsf{t}+1}}\ln P_{\mathsf{t}+1}[y{=}Y_{\mathsf{t}+1}; M_{\mathsf{t}+1}(z), C_{\mathsf{t}+1}(z), \dots] \\ &+ \frac{\partial C_{\mathsf{t}+1}}{\partial z^i}\frac{\partial}{\partial C_{\mathsf{t}+1}}\ln P_{\mathsf{t}+1}[y{=}Y_{\mathsf{t}+1}; M_{\mathsf{t}+1}(z), C_{\mathsf{t}+1}(z), \dots] + \dots . \end{aligned} \quad (4.13)$$

By factoring derivatives in this manner, the computation can be separated into two parts: the derivatives with respect to $M_{\mathsf{t}+1}$ and $C_{\mathsf{t}+1}$, which are typically quite straightforward; and the derivatives with respect to z elements, which typically need a more involved calculation depending on the model. Incidentally, this separation also neatly lends itself to abstracting gradient calculations as having a simpler, general purpose component that can be built directly into a library of data models and a more complex, model-specific component that the user must specify.

4.3 Software design

Let's now take a step back from the specifics of the probabilistic filtering algorithm to introduce our new sofware package for this part of the book: the 'learnadex'. At its core, the learnadex algorithm adapts the stochadex iteration engine to iterate through streams of data in order to accumulate a global objective function value with respect to that data. The user may then choose which optimisation algorithm (or write their own) to use in order to leverage this objective for learning a better representation of the data. Readers with some familiarity of machine learning concepts will note that this design should enable either 'offline' or 'online' learning algorithms to work.

Using the iteration engine of the stochadex makes neat use of software which has already been designed and tested in earlier chapters of this book. However, in order to fully achieve this, a few minor extensions to the typing structures and code abstractions are necessary; as we show in Fig. 4.3. To start with, we separate out 'learning' from the kind of optimiser in the overall config so as to enable multiple optimisation algorithms to be used for the same learning problem. The hyperparameters that define that optimisation problem domain can be determined by the user with an extension to the `OtherParams` object so that it includes some optional Boolean masks over the parameters, i.e., `OtherParams.FloatParamMask` and `OtherParams.IntParamMask`. These masks are used to extract the parameters of interest, which can then be flattened and formatted to fit into any generic optimisation algorithm.

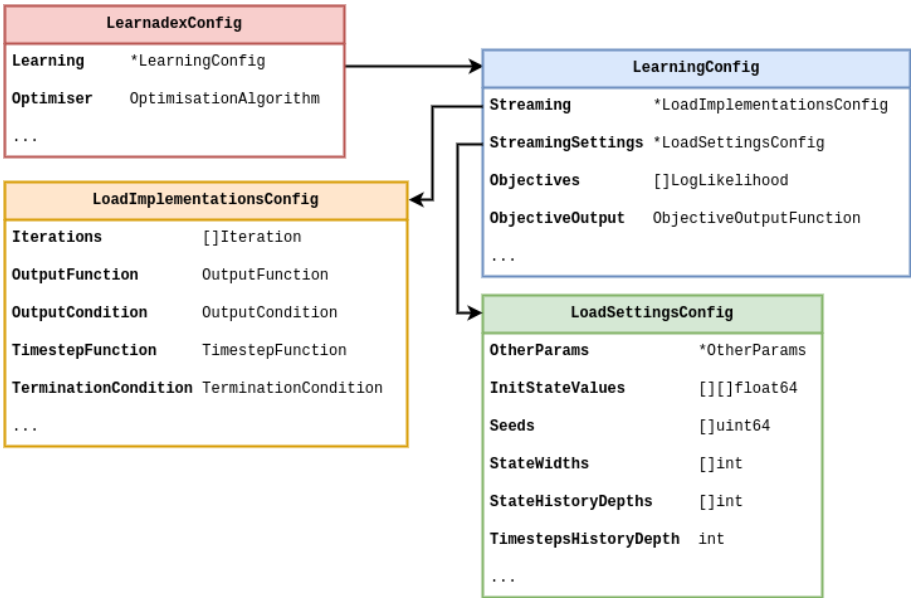

Figure 4.3: A relational summary of the core data types in the learnadex.

On the learning side; in order to define a specific objective for each data iterator to compute while the data streams through it, we have abstracted a 'log-likelihood' type. Similarly, each iterator also gets a data streamer configuration which defines where the data is streaming from — e.g., from a file on disk, from a local database instance or maybe via a network socket — and also some inherited abstractions from the stochadex which define the time stepping function and when the

data stream ends. In Fig. 4.4 below, we provide a schematic of the method calls of (and within) each data iterator.

- describe the method calls diagram in more detail — in particular, point out how it can replace the `Iteration.Iterate` method which is called when the `StateIterator` is asked for another iteration from the `PartitionCoordinator` of the stochadex
- then talk about the optimiser! starting with non-gradient-based: the two packages that are supported out of the box are gonum and eaopt (still need to do gago — see here: github.com/maxhalford/eaopt)
- then talk about the output - talk about the possibilities for output and what the default setting to json logs is for
- could also be written to, e.g., a locally-hosted database server and the best-suited would be a NoSQL document database, e.g., MongoDB [25], but building something bespoke and simpler is more aligned with the use-case here and with the principles of this book
- describe the need for log exploration and visualisation and then introduce logsplorer - a REST API for querying the json logs (with basic filtering and selection capabilities but could be extended to more advanced options) and optionally also launches a visualisation React app written in Typescript
- note how this could be scaled to cloud services easily and remotely queried through the logsplorer API and visualised

As with the software we wrote for the stochadex, the learnadex main binary executable leverages templating to enable full configurability of all the implementations and settings of Fig. 4.3 through passing configs at runtime. Users can alternatively use the learnadex as a library for import, if they desire more control over the code execution.

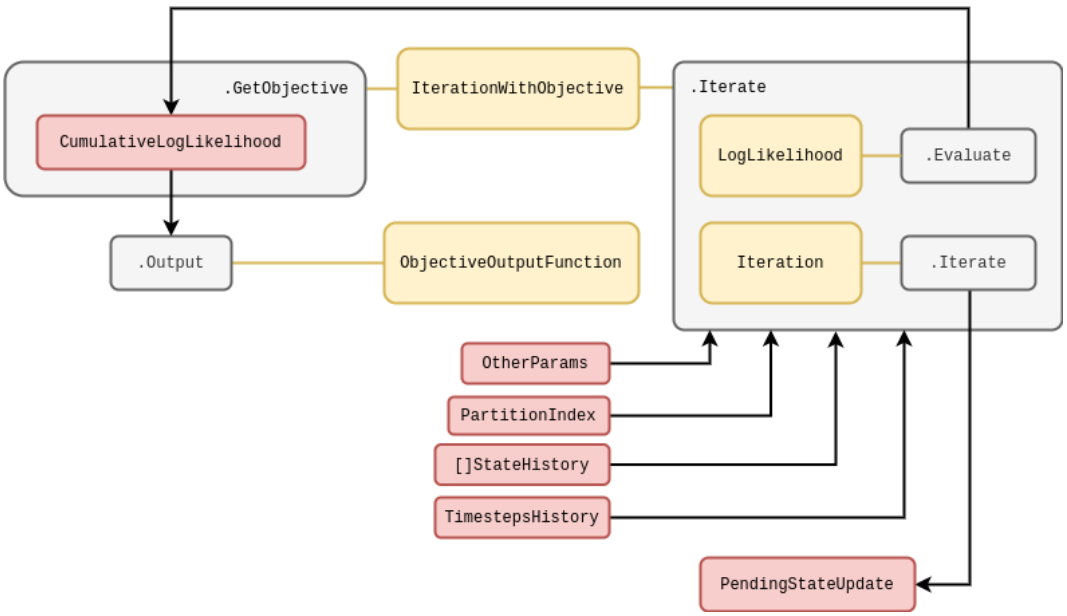

Figure 4.4: A schematic of an iteration with an objective function evaluation.

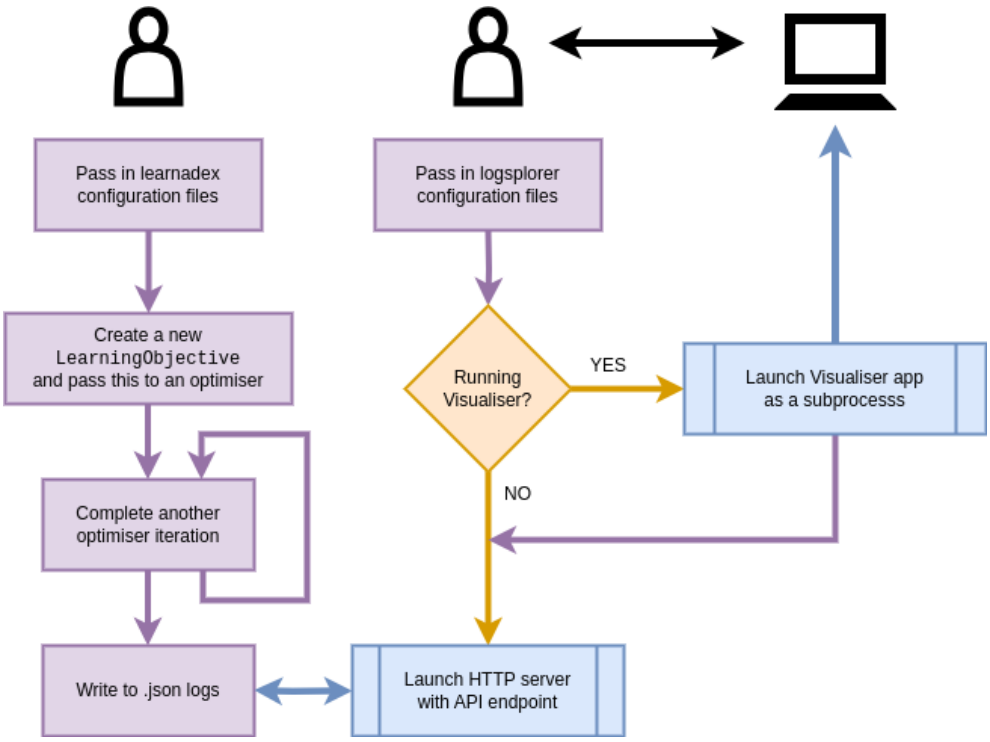

Figure 4.5: A diagram of the main learnadex and logsplorer executables.

5

Generalised MAP inference

Concept. To largely generalise the procedure of statistical inference for any model using an algorithm which builds from techniques we developed in the previous chapter. When we say 'statistical inference' here; we specifically mean computing the maximum a posteriori (MAP) estimate for any arbitrary stochastic model which has been defined in the stochadex simulator. In order for our algorithm to evaluate the MAP of a model, we show that the user must specify the prior distribution over model parameters, and the model must itself be defined within the stochadex. In this chapter, we will discuss some concepts which are commonplace within the field of Bayesian inference and provide a few simple examples of how our algorithm might work in various instances. For the mathematically-inclined, this chapter will give a very brief exposition for Bayesian statistical inference metholodology — in particular, how it relates to the evaluation of MAP estimates. For the programmers, the software described in this chapter lives in the public Git repository: https://github.com/umbralcalc/learnadex.

5.1 Inference methodology

In Bayesian inference, one applies Bayes' rule to the problem of statistically inferring a model from some dataset. This typically involves the following formula for a posterior distribution

$$\mathcal{P}_{\mathtt{t}+1}(z|Y) \propto \mathcal{L}_{\mathtt{t}+1}(Y|z)\mathcal{P}(z)\,. \tag{5.1}$$

In the formula above, one relates the prior probability distribution over a parameter set $\mathcal{P}(z)$ and the likelihood $\mathcal{L}_{\mathtt{t}+1}(Y|z)$ of some data matrix Y up to timestep $\mathtt{t}+1$ given the parameters z of a model to the posterior probability distribution of parameters given the data $\mathcal{P}_{\mathtt{t}+1}(z|Y)$ up to a proportionality constant. All this may sound a bit technical in statistical language, so it can also be helpful to summarise what the formula above states verbally as follows: the initial (prior) state of knowledge about the parameters z we want to learn can be updated by some likelihood function of the data to give a new state of knowledge about the values for z (the 'posterior' probability).

From the point of view of statistical inference, if we seek to maximise $\mathcal{P}_{\mathtt{t}+1}(z|Y)$ — or its logarithm — in Eq. (5.1) with respect to z, we will obtain what is known as a maximum posteriori

(MAP) estimate of the parameters. In fact, we have already encountered this metholodology in the previous chapter when discussing the algorithm which obtains the best fit parameters for the empirical probability filter. In this case; while it appears that we optimised the log-likelihood directly as our objective function, one can easily show that this is also technically equivalent obtaining a MAP estimate where one chooses a specfic prior $\mathcal{P}(z) \propto 1$ (typically known as a 'flat prior').

So we need to now specify in a little more detail how we might go about a practical calculation of the posterior with some arbitrary stochastic process model that has been defined in the stochadex. In order to make the comparison to a real dataset, any stochadex model of interest will always need to be able to generate observations which can be directly compared to the data. To formalise this a little; a stochadex model could be represented as a map from z to a set of stochastic outputs $\mathsf{Y}_{\mathsf{t}+1}(z), \mathsf{Y}_{\mathsf{t}}(z), \dots$ that are directly comparable to the rows in the real data matrix Y. The values in Y may only represent a noisy or partial measurement of the latent state of the simulation x, so a more complete picture can be provided by the following probabilistic relation

$$P_{\mathsf{t}+1}(\mathsf{y}|z) = \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x \, P_{\mathsf{t}+1}(\mathsf{y}|x) P_{\mathsf{t}+1}(x|z) \,, \tag{5.2}$$

where, in practical terms, the measurement probability $P_{\mathsf{t}+1}(\mathsf{y}|x)$ of $\mathsf{y} = \mathsf{Y}_{\mathsf{t}+1}$ given $x = X_{\mathsf{t}+1}$ can be represented by sampling from another stochastic process which takes the state of the stochadex simulation as input. Given that we have this capability to compare like-for-like between the data and the simulation; the first problem is to figure out how this comparison between two sequences of vectors can be done in a way which ensures the the statistics of the posterior are ultimately respected.

Introduce the fact that state is needed as well as z... Also introduce Likelihood-free methods and ABC-like method here https://link.springer.com/article/10.1007/s11222-022-10092-4

We may generally assert that

$$\mathcal{P}_{\mathsf{t}+1}(x, z|Y) = \prod_{\mathsf{t}'=0}^{\mathsf{t}+1} \left[\int_{\varpi_{\mathsf{t}'}} \mathrm{d}\mathsf{y}' \, \mathcal{P}_{\mathsf{t}'}(\mathsf{y}'|Y) \right] P_{\mathsf{t}+1}(x, z|\mathsf{Y}) \,. \tag{5.3}$$

As we demonstrated in the previous chapter, it's possible for us to also optimise a probability distribution $\mathcal{P}_{\mathsf{t}+1}(\mathsf{y}|Y) = P_{\mathsf{t}+1}(\mathsf{y}; M_{\mathsf{t}+1}, C_{\mathsf{t}+1}, \dots)$ for each step in time to match the statistics of the measurements in Y as well as possible, given some statistics $M_{\mathsf{t}+1}$ and $C_{\mathsf{t}+1}$. We do not necessarily need to obtain these statistics from the probability filter estimation method, but could instead try to fit them via some other objective function. Either way, this represents a lossy *compression* of the data we want to fit the simulation to, and so the best possible fit is desirable; regardless of overfitting.

Let's consider a few concrete examples of $P_{\mathsf{t}+1}(\mathsf{y}; M_{\mathsf{t}+1}, C_{\mathsf{t}+1}, \dots)$. If the data measurements were well-described by a multivariate normal distribution, then one would use terms like

$$P_{\mathsf{t}+1}(\mathsf{y}; M_{\mathsf{t}+1}, C_{\mathsf{t}+1}, \dots) = \mathsf{MultivariateNormalPDF}(\mathsf{y}; M_{\mathsf{t}+1}, C_{\mathsf{t}+1}) \,, \tag{5.4}$$

where $M_{\mathsf{t}+1}$ and $C_{\mathsf{t}+1}$ would be estimated from the data measurements in Y. Similarly, if the data measurements were instead better described by a Poisson distribution, we might disregard the need for a covariance matrix statistic $C_{\mathsf{t}+1}$ and instead use

$$P_{\mathsf{t}+1}(\mathsf{y}; M_{\mathsf{t}+1}, C_{\mathsf{t}+1}, \dots) = \mathsf{PoissonPMF}(\mathsf{y}; M_{\mathsf{t}+1}) \,. \tag{5.5}$$

Once again, here, we would need to determine $M_{\mathsf{t}+1}$ from Y. The more statistically-inclined readers may notice that the probability mass function here would require the integrals in Eq. (5.3) to be replaced with summations over the relevant domains.

Eq. (5.3) demonstrates how one can construct a statistically meaningful way to compare the sequence of real data measurements $Y_{\mathsf{t}+1}, Y_{\mathsf{t}}, \dots$ to their modelled equivalents $\mathsf{Y}_{\mathsf{t}+1}(z), \mathsf{Y}_{\mathsf{t}}(z), \dots$. But we still haven't shown how to compute $P_{\mathsf{t}+1}(x, z|\mathsf{Y})$ for a given simulation, and this can be the most challenging part. To begin with, we can apply Bayes' rule and the chaining of conditional probability to find

$$P_{\mathsf{t}+1}(x, z|\mathsf{Y}) \propto P_{\mathsf{t}+1}(\mathsf{y}|x) P_{\mathsf{t}+1}(x|z, \mathsf{Y}') P_{\mathsf{t}}(z|\mathsf{Y}')\,, \tag{5.6}$$

where, to keep the expression simpler, we are leveraging our notation in previous chapters to use Y' as indicating all of the past rows of simulation measurements up to Y_{t}.

The relationship between $P_{\mathsf{t}+1}(\mathsf{y}|x, \mathsf{Y}')$ and previous timesteps can be directly inferred from the probabilistic iteration formula that we introduced in the previous chapter. The expression is

$$P_{\mathsf{t}+1}(x|z, \mathsf{Y}') = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}x'\, P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)\, P_{\mathsf{t}'}(x'|z, \mathsf{Y}')\,. \tag{5.7}$$

So we can map probabilities of x throughout time and so learned information about the state of the system can be applied from previous values, given z. But is there a similar relationship we might consider for $P_{\mathsf{t}+1}(z|\mathsf{Y})$? Yes indeed, the marginalisation

$$P_{\mathsf{t}+1}(z|\mathsf{Y}) = \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, P_{\mathsf{t}+1}(x, z|\mathsf{Y}) \propto \left[\int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, P_{\mathsf{t}+1}(\mathsf{y}|x) P_{\mathsf{t}+1}(x|z, \mathsf{Y}') \right] P_{\mathsf{t}}(z|\mathsf{Y}')\,, \tag{5.8}$$

shows how the z updates can occur in an iterative fashion.

Got up to here...

We now have an objective function to optimise values of z with respect to in Eq. (5.3) but, in practice, this optimisation problem typically has several layers of difficulty to it. Since the model has been defined by its stochastically generated samples $\mathsf{Y}_{\mathsf{t}+1}(z), \mathsf{Y}_{\mathsf{t}}(z), \dots$, the objective function will manifestly be stochastic too. Another layer of difficulty is that gradients of the objective function are not immediately computable and so navigation around the optimisation domain could be difficult, especially in high-dimensional problems. To solve this issue in a way which maintains the generality of our approach, it turns out that we can rely on another application of our probability filter!

Let's now consider a process which represents the progress made by an optimiser towards the optimum value, whose state is defined by the parameters z. If we assume that all the elements of z are continuous[1] in their domain υ, one representation of this process we might consider is

$$\mathcal{P}^{n+1}_{\mathsf{t}+1}(x|z, Y_{\mathsf{t}}, Y_{\mathsf{t}-1}, \dots) = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}x'\, P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)\, \mathcal{P}^{n+1}_{\mathsf{t}'}(x'|z, Y_{\mathsf{t}'}, Y_{\mathsf{t}'-1}, \dots) \tag{5.9}$$

$$\mathcal{P}^{n+1}_{\mathsf{t}+1}(\mathsf{Y}_{\mathsf{t}+1}|z, Y_{\mathsf{t}}, Y_{\mathsf{t}-1}, \dots) = \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, P_{\mathsf{t}+1}(\mathsf{Y}_{\mathsf{t}+1}|x) \mathcal{P}^{n+1}_{\mathsf{t}+1}(x|z, Y_{\mathsf{t}}, Y_{\mathsf{t}-1}, \dots) \tag{5.10}$$

$$\mathcal{P}^{n+1}_{\mathsf{t}+1}(z|\mathsf{Y}_{\mathsf{t}+1}, Y_{\mathsf{t}+1}) = \frac{1}{n} \sum_{n'=0}^{n} \int_{\upsilon_{n'}} \mathrm{d}z'\, P^{(n+1)n'}(z|z')\, \mathcal{P}^{n'}_{\mathsf{t}+1}(z'|\mathsf{Y}_{\mathsf{t}+1}, Y_{\mathsf{t}+1})\,, \tag{5.11}$$

[1] We can modify this expression to work for discrete variables too.

where n is the iteration number of the optimisation, and hence $\mathcal{P}_{t+1}^{n+1}$ corresponds to a different candidate posterior distribution to one at any of the previous steps $\mathcal{P}_{t+1}^{n'}$. Eq. (5.11) can represent the convergence of a probabilistic optimisation algorithm — represented in particular by $P^{(n+1)n'}(z|z')$ — towards the true posterior distribution of z. As such an algorithm converges, we can recompute (and hence iteratively improve) the MAP estimate with respect to each iteration of the posterior.

Readers with some machine learning experience may be familiar with the classic exploration vs exploitation tradeoffs. It's clear that these tradeoffs will manifest in our case here when trying to strike a balance between iterating the posterior distribution in Eq. (5.11) and optimizing the current posterior with respect to z to compute the MAP.

Readers may also have recognized that Eq. (5.11) has the same structure as the generalised probabilistic description we introduced previously. This structure enables us to reuse all of the exposition we provided for the probabilistic filter and highlights how the filter itself can be used in the algorithm to optimise the posterior.

If we now synthesize both of these observations together, we can see how a stochastic variant of the well-known Expectation-Maximisation Algorithm [26, 27, 18] naturally emerges.

- consider a tunable conditional probability that is a Gaussian with mean and variance estimated directly from the history (don't optimise it like in Bayesian optimisation!) with a tunable exponential timestep kernel (timestep being the optimiser step here) and the resulting function can be optimised (or draw monte-carlo samples from) in an EM algorithm approach. The resulting Gaussian function could also exploit gradients for SGD.
- Illustrate the algorithm structure and describe the maths for it step-by-step as well: 1. choose random sampling for $P^{(n+1)n'}(z|z')$ 2. alternate between using the probability filter for $P^{(n+1)n'}(z|z')$ to sample and optimisation of the current posterior iteration with respect to z
- Reference the contrast to ABC methods here, which involve approximating the data likelihood with a simple proximity function with a tolerance ϵ.
- Also talk about the BOLFI method which does indeed use the full Bayesian optimisation as it goes.
- Programming steps for this chapter: need to write a new OptimisationAlgorithm which implements this algorithm — the rest should just be config

6

Inferring 2D spatial dynamics

Concept. The idea here is... For the mathematically-inclined, this chapter will... For the programmers, the software described in this chapter lives in the public Git repository: https://github.com/umbralcalc/learnadex.

6.1 Adapting the probabilistic formalism

Let's by returning to the probabilistic filter formalism that we introduced earlier and noting that the covariance matrix estimate given by

$$C^{ij}_{\mathtt{t}+1}(z) = \int_{\omega_{\mathtt{t}+1}} \mathrm{d}x\,[x - M_{\mathtt{t}+1}(z)]^i [x - M_{\mathtt{t}+1}(z)]^j P_{\mathtt{t}+1}(x|z)\,, \tag{6.1}$$

represents a matrix that could get very large, depending on the problem. For example; if we encoded the state of a 2-dimensional spatial field of values into the elements $X^i_{\mathtt{t}}$, the number of elements in the covariance matrix $C^{ij}_{\mathtt{t}+1}(z)$ would scale as $4N^2$ — where N here is the number of spatial points we wanted to encode.

One solution to this scaling problem is to exploit the fact that, in many spatial processes, the proximity of points can strongly determine how correlated they are. Hence, for pairwise distances further than some threshold, the covariance matrix elements should tend towards 0. If we were to place points along the diagonal of $C^{ij}_{\mathtt{t}+1}(z)$ in order of how close they are to each other, this threshold would then be represented as a *banded matrix*. We have illustrated such a matrix in Fig. 6.1 in which the 'bandwidth' is defined as the number of diagonals one needs to traverse from the main diagonal before encountering a diagonal of 0s.

The extra detail that's also needed here is to consider how we encode a 2-dimensional spatial process into our state vector, and how the elements of the resulting state vector might be correlated to one another depending on their spatial proximity. If we start with a Markovian Gaussian random field, we can derive the Matérn kernel over these spatial coordinates in order to correlate the state vectors in such a way.

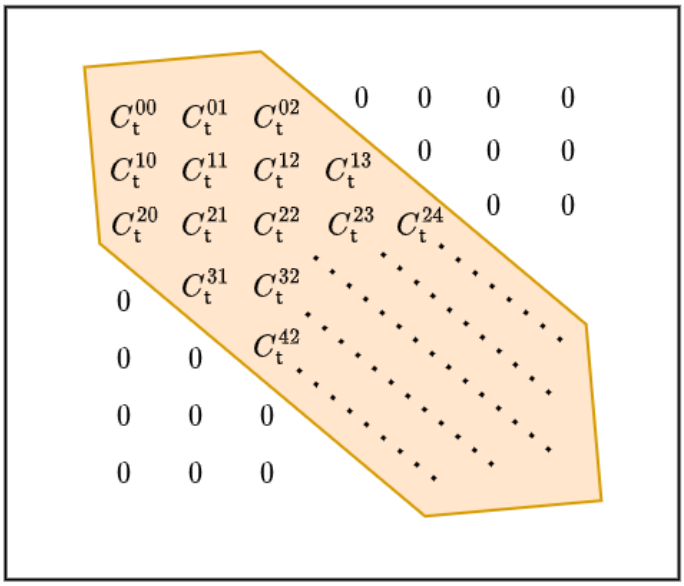

Figure 6.1: An illustration of a banded covariance matrix with a bandwidth of 2.

7

Learning from ants on curved surfaces

Concept. The idea here is

7.1 Diffusive limits for ant interactions

8

A world of hydrodynamic ensembles

Concept. The idea here is...

8.1 The Boltzmann/Navier-Stokes equations

Consider the full phase space (so add in velocity components to each state point in the 2d plane).

Part 3

9

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 two layers of possible interactions which are facilitated by a new vector-valued 'parametric action' function $G_{\mathtt{t}}$ and a new vector-valued 'state action' function $H_{\mathtt{t}}$. In doing so we shall be defining the domain of an acting entity in the stochastic process environment — which we shall hereafter refer to as simply the 'agent'.

During a timestep over which actions are performed by the agent, the stochadex state update formula can be extended to look like this system of equations

$$Z^i_{\mathtt{t}+1} = G^i_{\mathtt{t}+1}(Z_{\mathtt{t}}, \mathcal{A}_{\mathtt{t}+1}) \tag{9.1}$$

$$X^i_{\mathtt{t}+1} = H^i_{\mathtt{t}+1}[F_{\mathtt{t}}(X', Z_{\mathtt{t}+1}, \mathtt{t}), \mathcal{A}_{\mathtt{t}+1}]\,, \tag{9.2}$$

where we have also introduced the concept of the 'actions' performed $\mathcal{A}_{\mathtt{t}+1}$ on the system; some vector of parameters which define what actions are taken at timestep $\mathtt{t}+1$.

In Eqs. (9.1) and (9.2), notice that we have replaced the constant vector of parameters z (as in the stochadex formalism) for a time-dependent vector $Z_{\mathtt{t}}$ of parameters that can be updated by $G_{\mathtt{t}}$

at any (but not necessarily every) timestep. From the perspective of the whole matrix X update step; $G_{\mathtt{t}}$ and $H_{\mathtt{t}}$ combine to become technically equivalent to applying this formal composition of functions

$$X^i_{\mathtt{t}+1} = H^i_{\mathtt{t}+1}(F_{\mathtt{t}}(X', G^i_{\mathtt{t}+1}(Z_{\mathtt{t}}, \mathcal{A}_{\mathtt{t}+1}), \mathtt{t}), \mathcal{A}_{\mathtt{t}+1}) = \mathcal{F}^i_{\mathtt{t}+1}(X', Z_{\mathtt{t}}, \mathcal{A}_{\mathtt{t}+1}, \mathtt{t})\,, \tag{9.3}$$

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' $\mathcal{A}_{\mathtt{t}+1}$ at timestep $\mathtt{t}+1$ — because the parameters which define either type of action at this timestep should all be stored within $\mathcal{A}_{\mathtt{t}+1}$ anyway. It is, however, important not to forget the full mathematical formulation when performing calculations.

The code for the new iteration formula given by Eq. (9.3), which includes taking actions in the same timestep, would look something like Fig. 9.1.

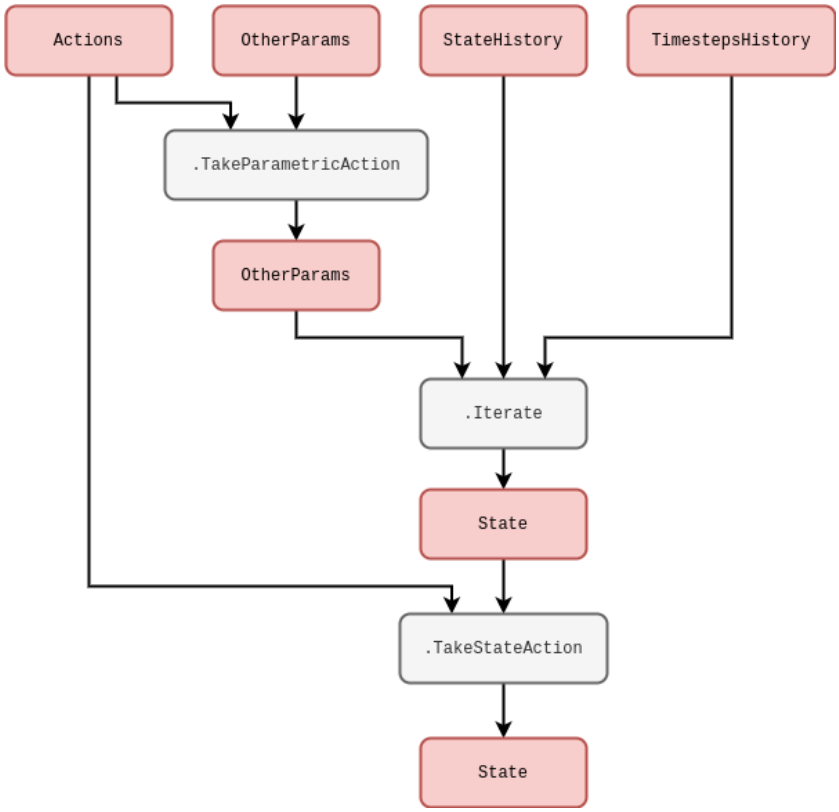

Figure 9.1: Code schematic of Eq. (9.3).

10

Angling for freshwater fish

Concept. The idea here is

10.1 A large-scale Lotka-Volterra model

Inspired by the empirical dynamical modeling approach to sockeye salmon in Ref. [28], 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. [29]...

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

11

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 modeling 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); Kickoff, 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

[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.

define the kinds of state transitions which occur.[2] For example, kickoffs, 22m dropouts and goal line dropouts are all modelled here as a Kickoff state but with different initial ball locations on the pitch (we'll get to how ball location changes later on).

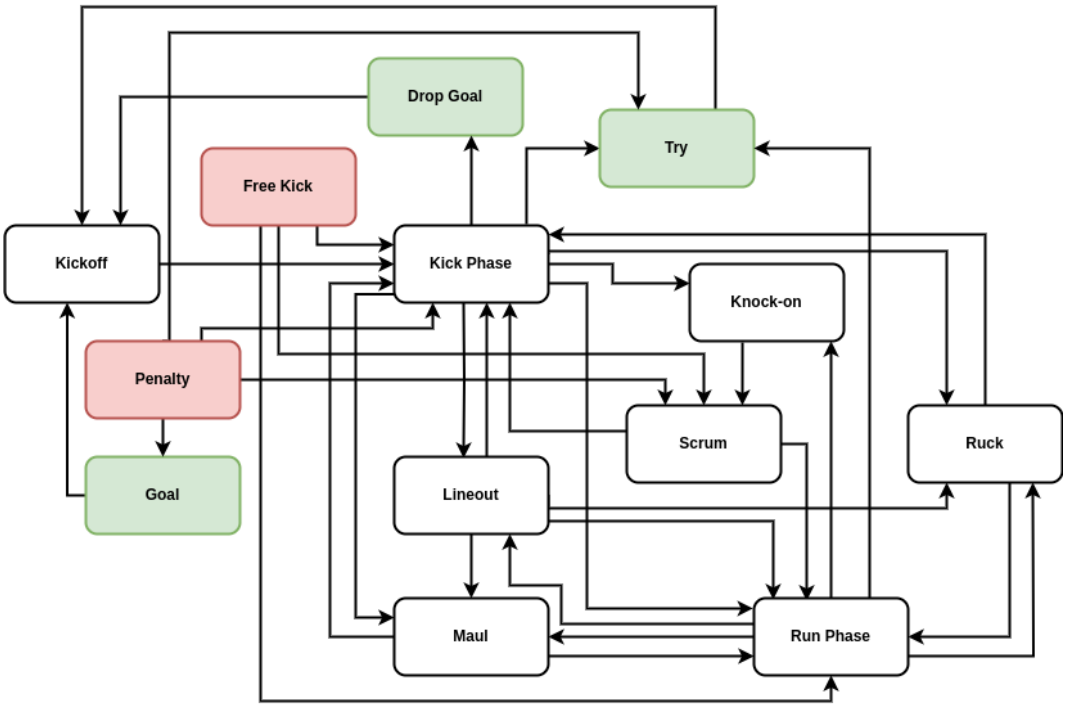

Figure 11.1: Simplified event graph of a rugby union match.

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 introduce states for each player.

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 $a \to b$ on the event graph an associated expected rate of occurance $\lambda_{a \to b}$ 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

[2] It's also fair to say that our simplified model here represents just a subset of states that a real rugby match could exist in.

may occur while the match is also in most of the white-coloured states on the graph apart from Knock-on (which determines a possession change immediately through a Scrum) or a Kickoff (which directly proceeds a Kick Phase from which the ball may change possession). Let's assign possession changes with a state, parameter and timestep-dependent expected rate of occurance $\lambda_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t})$.

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

$$X^0_{\mathsf{t}} = \begin{cases} 0 & \text{Match State} = \textsf{Penalty} \\ 1 & \text{Match State} = \textsf{Free Kick} \\ \dots & \end{cases} \tag{11.1}$$

$$X^1_{\mathsf{t}} = \begin{cases} 0 & \text{Possession} = \textsf{Home Team} \\ 1 & \text{Possession} = \textsf{Away Team} \end{cases} . \tag{11.2}$$

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 $X^0_{\mathsf{t}} = \mathsf{a}$ at timestep t is written as $P^0_{\mathsf{t}}(\mathsf{a})$, then the probability of $X^0_{\mathsf{t}+1} = \mathsf{b}$ in the following timestep is

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

where $\forall \mathsf{a} \neq \mathsf{b}$ in the summation indicates that all the available transitions from a to b, where $\mathsf{a} \neq \mathsf{b}$, should be summed over and $\mathcal{T}_{\mathsf{a}\to\mathsf{b}}(X_{\mathsf{t}}, z, \mathsf{t})$ 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^1_{t} at time t evolves according to

$$P^1_{\mathsf{t}+1} = \frac{\frac{1}{\tau} P^1_{\mathsf{t}} + \lambda_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t})(1 - P^1_{\mathsf{t}})}{\left[\frac{1}{\tau} + \lambda_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t})\right]} . \tag{11.4}$$

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 $X^2_{\mathsf{t}} = L^{\text{lon}}_{\mathsf{t}}$ and $X^3_{\mathsf{t}} = L^{\text{lat}}_{\mathsf{t}}$ — where $L^{\text{lon}}_{\mathsf{t}}$ denotes the longitudinal component (lengthwise along the pitch) and $L^{\text{lat}}_{\mathsf{t}}$ 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 most cases this restriction makes sense given the real-world game patterns, but perhaps the only clear exception is the Penalty $\longrightarrow$ Goal transition; which is easier to think of as a kind of 'Kick Phase transition' anyway.

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 $P_{\mathsf{t}+1}(\ell)$ of $L_{\mathsf{t}+1}^{\mathrm{lon}} - L_{\mathsf{t}}^{\mathrm{lon}} = \ell$, evolves according to

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

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 $P_{\mathsf{t}+1}(w)$ of $L_{\mathsf{t}+1}^{\mathrm{lat}} - L_{\mathsf{t}}^{\mathrm{lat}} = w$ can just be updated like so

$$P_{\mathsf{t}+1}(w) = \mathsf{NormalPDF}(w; 0, \sigma_{\mathrm{run}}^2) \,, \tag{11.6}$$

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 and lateral components are only realistically controlled by the attacking team — specifically, by the player who is currently the kicker. Referring back to the state transitions which precede a Kick Phase in Fig. 11.1, we note that there are several types of kick which can dictate what the mechanics of the process should look like. To keep things simple, we can cluster these types of event into the following categories

1. Kicks at the goalposts for points:
   - Penalty $\longrightarrow$ Goal
   - Kick Phase $\longrightarrow$ Drop Goal
2. Kicks in the general field of play where the ball does not leave the field:
   - Kick Phase $\longrightarrow$ Try
   - Kick Phase $\longrightarrow$ Run Phase
   - Kick Phase $\longrightarrow$ Knock-on
   - Kick Phase $\longrightarrow$ Ruck
   - Kick Phase $\longrightarrow$ Maul
3. Kicks to the touchline where the ball leaves the field:
   - Kick Phase $\longrightarrow$ Lineout

To model case 1. above, the simplest option would be to associate the attempt at goal with a goal success probability for the kicker p_{goal} which, in the simple first version of our model, will not depend on the location on the pitch from which the kick is taken. We will, however, restrict kickers to only be allowed an attempt at goal if they are within their opposing team's half — this is not strictly a rule in rugby, but it simplifies the automation of the decision logic quite nicely for now.

In case 2 above, we can think of two main tactical options that a team might be employing. The first of these is kicking a further longitudinal distance in the field of play in order to gain territory, but lose possession, and the second is to kick to regain possession but with a shorter longitudinal distance. When kicking to gain territory, we'll just assign a uniform probability to the lateral position update of the ball location for simplicity and for the probability density $P_{\mathsf{t}+1}(\ell)$ of $L^{\text{lon}}_{\mathsf{t}+1} - L^{\text{lon}}_{\mathsf{t}} = \ell$, we'll use

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

where a_{kick} is the kicking player's longitudinal scale parameter. When kicking to regain possession, we will use another exponential distribution with another constant scale parameter ℓ_{typ} for the typical distance gained by this type of kick (unassociated to either team's abilities) and also assign a 'regain possession' probability p_{reg} which is associated to the abilities of the players chasing the kick (on the kicker's team).

Lastly, to model case 3. above, the event has determined that the ball will leave the field of play and so the remaining unknowns that need to be determined are: which side of the field this occurred (we'll just choose the side closest to touch when the ball was last in play), the longitudinal distance of the kick and whether or not the ball bounced before leaving the field. For simplicity, let's determine the last of these through another kind of kick accuracy probability p_{kick} associated to the kicker. This just leaves the longitudinal distance that the kick achieved along the touchline; in this case we'll just assign the probability density $P_{\mathsf{t}+1}(\ell)$ of $L^{\text{lon}}_{\mathsf{t}+1} - L^{\text{lon}}_{\mathsf{t}} = \ell$ to that of Eq. (11.7).

Generally, the proceeding Lineout will be taken with the opposition team (to the kicker) in possession at the point where the ball left the field. However, there are two notable exceptions to this rule. The first is if the ball does not bounce before leaving the field of play and the player is outside of their team's 22m line — the Lineout then must occur where the ball was last kicked from with the opposition team in possession. The second exception is if the ball bounces into touch when the player who kicked it was inside their team's half of the pitch — the Lineout then occurs where the ball left the field but is taken with the kicker's team in possession.

Before moving on to player states and abilities it's important to note that, in addition to the other state variables we have discussed above, the score of the match obviously needs to be recorded in the overall match state X_{t} as well. To be explicit, we'll say that $X^4_{\mathsf{t}} = \mathsf{Home\ Team\ Score}$ and $X^5_{\mathsf{t}} = \mathsf{Away\ Team\ Score}$, recalling that kicks at goal from a Penalty Goal or Drop Goal are worth 3 points, a Try is worth 5 points and if this is proceeded by a conversion (another kick at goal taken laterally inline with where the Try was scored across the pitch) then this is worth an additional 2 points.

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 modeling 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 have begun 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. Note that our model associates a player with an 'possession attacking' and 'possession defending' ability which corresponds to each of their possession attributes that are indicated by the diagram. For example, a Front Row Forward will have 10 possession abilities associated to them: 2 for each of their Scrum, Lineout, Ruck, Maul and Run attributes.

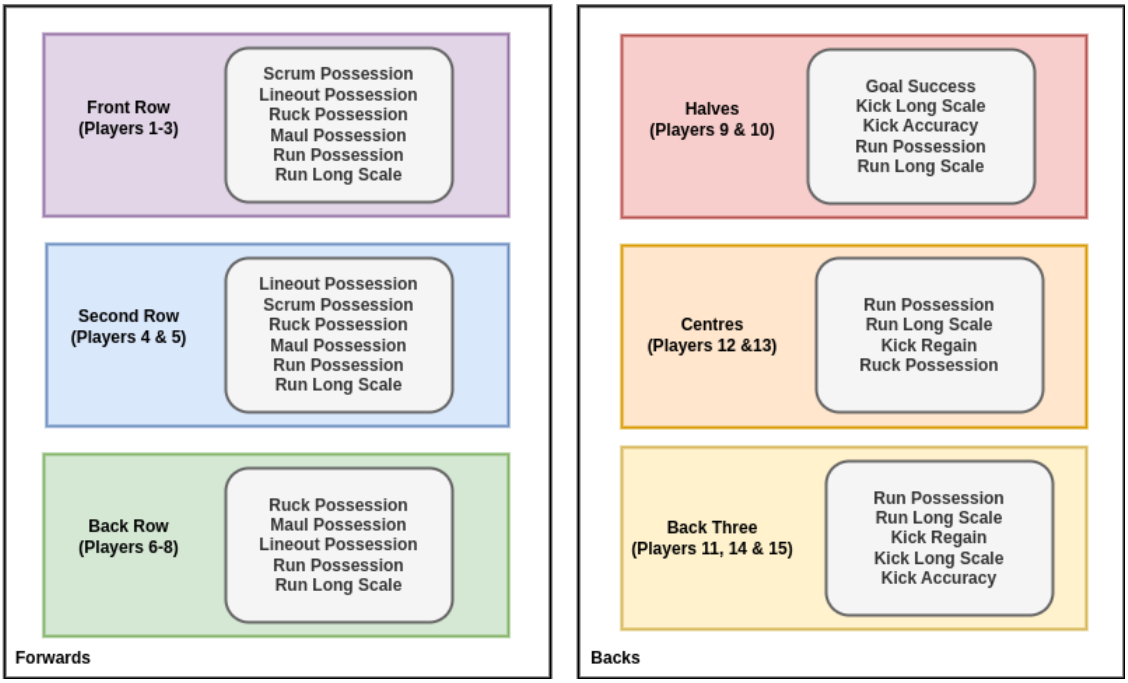

Figure 11.2: Associated playing abilities for each position type.

Let's now say that z contains all of these parameters for all of the players on both sides, and also whether or not each player is actively playing or on the bench. With this information, and the knowledge of which team is in possession from X_t^1, it should be simple to create a vector-valued function $a_{\text{pos}}(X_t, z)$ which returns all of the possession attacking attributes that are associated to match state X_t^0 and an analogous one $d_{\text{pos}}(X_t, z)$ 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 $a_{\text{pos}}(X_t, z)$ and the Away Team's possession defending attributes that are returned by $d_{\text{pos}}(X_t, z)$.

In order to model the effect of player fatigue over the course of a match, we can add some vectors of player fatigue values f and times at which each player started playing t_{start} into the collection of parameters that are contained within z. These new parameters can then be used to define a

formula for the decline of each attribute over the course of a match. Let's redefine these declining values as

$$a^i_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t}) = a^i_{\text{pos}}(X_{\mathsf{t}}, z)e^{-f^i[t(\mathsf{t})-t^i_{\text{start}}]} \tag{11.8}$$

$$d^i_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t}) = d^i_{\text{pos}}(X_{\mathsf{t}}, z)e^{-f^i[t(\mathsf{t})-t^i_{\text{start}}]}\,. \tag{11.9}$$

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 $\lambda_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t})$ through a balance of attacking and defensive attributes in the following relation

$$\lambda_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t}) = \frac{\lambda^*_{\text{pos}} \sum_{\forall i} d^i_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t})}{\sum_{\forall i} a^i_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t}) + \sum_{\forall i} d^i_{\text{pos}}(X_{\mathsf{t}}, z, \mathsf{t})}\,, \tag{11.10}$$

where λ^*_{pos} is the maximum rate that is physically possible and the $\forall i$ in the summations indicates summing over all attacking or defending player attributes of the vector in each instance. In addition to this possession change influence, players who have Run Phase and Kick Phase longitudinal scale attributes may affect the gain in distance 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 (uniformally across the team[3]) to be the the nominal 'attacker'. At the same time, an individual player on the defending side is chosen at random (again, uniformally 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. (11.8) and (11.9).

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 with kicking parameters is chosen at random (uniformally) and their a_{kick} or p_{kick} attributes are used in the relevant equations. If a kick stays within the field of play for the attacking team to attempt to regain possession, an additional 'chaser' player on the attacking team who possesses a kick regain p_{reg} probability is randomly chosen. In the specific case of Goal kicking from a Penalty, the designated place kicker on each side uses their p_{goal} attribute 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 taking actions with our formalism? We have to figure out what sorts of managerial actions $\mathcal{A}_{\mathsf{t}+1}$ are parametric in nature (affecting z) and if any directly act on the state of the system itself (affecting $X_{\mathsf{t}+1}$). Let's jump straight to the answer to the second question; it doesn't seem like there is any situation where the overall match state X_{t} itself is directly changed by a managerial action, so we will be using only parametric actions in this chapter.

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 are used by the $a_{\text{pos}}(X_{\mathsf{t}}, z)$, $d_{\text{pos}}(X_{\mathsf{t}}, z)$ or

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

$\mathcal{T}_{\mathsf{a}\to\mathsf{b}}(X_\mathsf{t}, z, \mathsf{t})$ functions. In the case of the possession attacking and defending attributes $a_{\text{pos}}(X_\mathsf{t}, z)$ and $d_{\text{pos}}(X_\mathsf{t}, z)$, a parametric action that the manager can perform would be to modify which players are actively playing through substitutions. To indicate that this underlying data may change, we can promote z to its actionable, time-dependent version Z_t such that the functions now become $a_{\text{pos}}(X_\mathsf{t}, Z_\mathsf{t})$ and $d_{\text{pos}}(X_\mathsf{t}, Z_\mathsf{t})$. In order to map substitutions/initial squad selections to the vector $\mathcal{A}_{\mathsf{t}+1}$, we can define the first set of 15 indices $\mathcal{A}^i_{\mathsf{t}+1}$ (where $i = 0, \ldots, 14$) as the player IDs chosen to be on the pitch for either the Home Team or the Away Team, depending on which side the manager is in charge of. If Z_t contains all of these IDs and the positions that they are allowed to play (as well as whether they are currently playing or not), then when the manager wishes to make a substitution, all that is needed is a change to $\mathcal{A}^i_{\mathsf{t}+1}$ and the parametric action function $G_{\mathsf{t}+1}(Z_\mathsf{t}, \mathcal{A}_{\mathsf{t}+1})$ can be defined to make a change $Z_\mathsf{t} \to Z_{\mathsf{t}+1}$ in response to this.

But what about $\mathcal{T}_{\mathsf{a}\to\mathsf{b}}(X_\mathsf{t}, Z_\mathsf{t}, \mathsf{t})$? What kinds of managerial actions can change the state transition probabilities through parameters? Given that these transition probabilities mostly arise from the tactics of each team, if the tactics of either team were changed throughout the match due to managerial decisions, these actions could be mapped to $\mathcal{T}_{\mathsf{a}\to\mathsf{b}}(X_\mathsf{t}, Z_\mathsf{t}, \mathsf{t})$. In order for these actions to have a clear influence on the game, however, we need to specify how team tactics get translated into transition probabilities. To keep things as simple as possible, we're going to specify only two tactical 'axes' on which a manager has to decide a position during each moment of the match.

The first, and perhaps more obvious, tactical decision axis to dynamically manipulate is the ratio between Kick Phase and Run Phase that the team chooses when it has possession of the ball, depending on what part of the pitch they are playing in. The second axis maps how aggressively a team pursues scoring tries over any other points (even when they may be on offer from a Penalty Goal). This latter axis also only really matters for the team in possession of the ball, so we aren't planning to map out any defensive tactics in our model for now. Since both of these actions can be mapped to a single axis each, these ratios ρ (each defined between $1 \geq \rho \geq 0$) can populate the last two indices of the actions vector: $\mathcal{A}^{15}_{\mathsf{t}+1}$ and $\mathcal{A}^{16}_{\mathsf{t}+1}$. When either of them has been changed, this can be mapped to $Z_{\mathsf{t}+1}$ using the parametric action function $G_{\mathsf{t}+1}(Z_\mathsf{t}, \mathcal{A}_{\mathsf{t}+1})$ and then $Z_{\mathsf{t}+1}$, in turn, can change the corresponding ratios between transition probabilities $\mathcal{T}_{\mathsf{a}\to\mathsf{b}}(X_{\mathsf{t}+1}, Z_{\mathsf{t}+1}, \mathsf{t}+1)$ when the attacking team is making these in-play decisions in the proceeding timestep.

Before moving on to the next section, there's a quick point to make about how managerial actions can affect ball locations on the pitch. In addition to the changes that we have discussed above, let's recall that the parameters a_{run}, d_{run}, a_{kick}, p_{kick}, p_{reg} and p_{goal} can all be indirectly determined by the manager to some extent through player selection/substitutions. So, while tactical managerial actions can change the ratios of state transitions themselves (at least while in possession of the ball), the players which are chosen in the first place (or by substitution) can have quite a significant influence on the outcome of a match.

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.

12

Influencing house prices

Concept. The idea here is

Part 4

13

Optimising actions for control objectives

Concept. The idea here is

13.1 States, actions and attributing rewards

Rewrite the beginning of this section to talk about:

- dynamic ensemble prediction and updating - online learning of the (x, z) posterior/MAP for generalised simulations by comparing previous future predictions to the current state and updating the surrogate distributions
- once this pattern is working, the methodology can be adapted directly to incorporate a policy/action-generating function as introduced further below

Up to this point, we have only considered actions which were either scheduled up front through some fixed process or through user interaction via a game interface. In order to start creating algorithms to act on the system state for us, we now need to develop a formalism which 'closes the loop' by feeding information back from the stochastic process to another decision-making process. Note that in most cases, the state of real-world phenomena cannot be measured perfectly. So in order to enable any agent trained on simulated phenomena to potentially act in the real world, we will need to model this measurement process as part of the information retrieval step.

Let's now define the concept of an 'environment state' $\mathcal{S}_{\mathsf{t}+1}$ at timestep $\mathsf{t}+1$; this is a new vector that doesn't have to share the same length as the measured state vector $X_{\mathsf{t}+1}$. We will then say generally that this environment state is 'observed' by the agent using the following observation function

$$\mathcal{S}^i_{\mathsf{t}+1} = O^i_{\mathsf{t}+1}(X', \mathcal{Z}_{\mathsf{t}+1}, \mathsf{t})\,, \tag{13.1}$$

where we have also introduced a new vector $\mathcal{Z}_{\mathtt{t}+1}$ which we will use to store all of the relevant parameters to the agent[1] at timestep $\mathtt{t}+1$.

If we are now given the conditional probability that an action vector element $\mathcal{A}_{\mathtt{t}+1} = a$ is chosen given that state vector $\mathcal{S}_{\mathtt{t}+1} = s$ has been measured $\pi(a, s) = p(a|s)$, we can use this to draw new actions for the agent with a newly defined action-generating function

$$\mathcal{A}^i_{\mathtt{t}+1} = \Pi^i_{\mathtt{t}+1}(\mathcal{S}_{\mathtt{t}+1}, \mathcal{Z}_{\mathtt{t}+1})\,. \tag{13.2}$$

From this point on we'll call $\pi(a, s)$ the 'policy' adoped by the agent. A Markov Decision Process (MDP) defines an algorithm in which the agent uses a single state measurement vector and its given policy π to draw actions $\mathcal{A}_{\mathtt{t}+1}$ at timestep $\mathtt{t}+1$. It then performs these actions in its environment, which we have previously formalised through defining the iteration $X_{\mathtt{t}+1} = \mathcal{F}_{\mathtt{t}+1}(X', Z_{\mathtt{t}}, \mathcal{A}_{\mathtt{t}+1}, \mathtt{t})$.

In order to assess the quality of an agent's actions, we might later attribute a reward value $\mathcal{R}_{\mathtt{t}}$ for actions that were taken at timestep $\mathtt{t}$. Using a series of these rewards, a return value R can also be computed using a future discount factor γ like so

$$R = \sum_{\mathtt{t}=0}^{\infty} \gamma^{\mathtt{t}} \mathcal{R}_{\mathtt{t}}\,. \tag{13.3}$$

A state-value function V_π is defined as the expectation (under policy π) of return R, given state vector $\mathcal{S}_{\mathtt{t}} = s$, i.e.,

$$V_\pi(s) = \mathrm{E}_\pi(R|s)\,. \tag{13.4}$$

Similarly, an action-value function Q_π is defined as the expectation (again, under policy π) of return R, given state vector $\mathcal{S}_{\mathtt{t}} = s$ and action vector $\mathcal{A}_{\mathtt{t}} = a$, i.e.,

$$Q_\pi(s, a) = \mathrm{E}_\pi(R|s, a)\,. \tag{13.5}$$

Follow-up this bit with the model-based approach that we're going to take in this book.

- Talk through value and policy learning - in this book we will be doing the value learning with our generalised stochastic model and then the policy learning bit is more nuanced.
- The value learning can be facilitated in software using a predictive model which is able to roll forecast rewards forward in time in a Monte Carlo fashion up to a window from a certain point given an input prior distribution of policies.
- This input prior distribution of policies can itself be optimised by maximising expected utility in a Bayesian design framework!

[1] This vector is intended to include parameters for measurement, policy specification and ultimately the learning algorithm as well.

14

Resource allocation for epidemics

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

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 TypeScript Programming Language." https://www.typescriptlang.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] "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] "The React Library." https://react.dev/.

[16] G. Sugihara and R. M. May, *Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series*, *Nature* **344** (1990) 734–741.

[17] A. Savitzky and M. J. Golay, *Smoothing and differentiation of data by simplified least squares procedures.*, *Analytical chemistry* **36** (1964) 1627–1639.

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

[19] D. J. MacKay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[20] J. A. Nelder and R. Mead, *A simplex method for function minimization*, *The computer journal* **7** (1965) 308–313.

[21] J. Kennedy and R. Eberhart, *Particle swarm optimization*, in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4, pp. 1942–1948, IEEE, 1995.

[22] Y. Shi and R. Eberhart, *A modified particle swarm optimizer*, in *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*, pp. 69–73, IEEE, 1998.

[23] H. Robbins and S. Monro, *A stochastic approximation method*, *The annals of mathematical statistics* (1951) 400–407.

[24] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, *arXiv preprint arXiv:1412.6980* (2014) .

[25] "The MongoDB Webpage." https://www.mongodb.com/.

[26] H. O. Hartley, *Maximum likelihood estimation from incomplete data*, *Biometrics* **14** (1958) 174–194.

[27] A. P. Dempster, N. M. Laird and D. B. Rubin, *Maximum likelihood from incomplete data via the em algorithm*, *Journal of the royal statistical society: series B (methodological)* **39** (1977) 1–22.

[28] 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.

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