Worlds of Observation

Building more realistic environments for machine learning

Robert J. Hardwick

January 11, 2024

Edited by C. M. Gomez-Perales

Introduction

In designing automated control algorithms of practical importance to the real world it's common to find that only partial observations of the system state are possible. You need only to think of the measurement uncertainties in any scientific experiment, the latent demand behind orders in a financial market, the unknown reservoirs of infection for a disease pathogen or even the limits to complete supply chain component observability in recognising just how commonly we find ourselves in this situation. When data is our only guide, this obscurity can make the learning of algorithms to control these systems an extreme — if not frequently impossible — challenge, without further insight provided by a more domain-specific model.

Worlds of Observation is a book about building more realistic training environments for machine learning algorithms to control these 'noisy' systems in the real world. Model-free reinforcement learning is a popular and very powerful approach to generating such algorithms [1], especially when there is plenty of data and the system is fully observable. However, this book will not use spend much time thinking about the model-free approach. We will instead be designing and building learning environments with a more model-based approach to control in mind. These environments are intended to replicate situations where the data isn't always so complete and useful, and will provide the necessary tools to cope with these tricky scenarios. Those readers who are data scientists, research engineers, statistical programmers or computational scientists may find our mathematically descriptive, yet practically-minded, approach in this book quite interesting and maybe a little different to the usual perspectives.

As can be expected from a book about algorithms, this text accompanies a lot of new open-source scientific software. Most of this code is written in various combinations of Go [2], C++ [3], Python [4] and TypeScript [5]. 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 study new phenomena and explore complex control problems effectively, regardless of their scientific background.

The need to properly test all this software has also provided a wonderful excuse to categorise some realistic, commonly-faced problems into a set of archetype simulation environments. These practical categories have been created based on various bits of domain knowledge and experience with real-world systems, but are not at all intended to form a complete set. We hope that our range of archetypes nicely illustrates the cross-disciplinary applicability of our algorithmic framework and gives some appealing variety to the reader as well. To explore these simulated systems further, we encourage readers to take a look at WorldsOOp (https://github.com/worldsoop/) — an open source software ecosystem inspired directly by this book.

To achieve some measure of rigour in our algorithmic approach, an important part of this book is the mathematical framework that it introduces and uses throughout. It seems silly to us that

mathematical formalities can obscure the practical computations that a programmer is being asked to implement when reading an equation. So, while we've tried to be as ambitious as possible with the level of technical detail in this book, we've also attempted to write many of the mathematical expressions in a more computer-friendly way where feasible,[1] in contrast with the more conventional formal descriptions. To help with this goal of explainability, we also make use of quite a lot of illustrations and diagrams.

A quick note on the code. All of the software that we describe in this book (including the software which compiles the book itself: https://github.com/umbralcalc/worlds-of-observation/) is shared under a MIT License [6] and can be found in one of these public Git repositories:

- Chapters 1-4: https://github.com/umbralcalc/
- Chapters 5-10: https://github.com/worldsoop/

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.

The book is split into two main parts: **Part 1**, which details the theoretical background and design of code; and **Part 2**, which explores some applications to realistic archetypes. We hope you, the reader, really enjoy reading through this book and using the all of the code that was built while writing it. We're very grateful to have been able to make use of all the amazing software written and maintained by the open source community which would otherwise have made this project impossible to achieve.

To cite this book in any work, please use the following BibTeX:

```
@book{worlds-of-observation-WIP,
  title   = {Worlds of Observation: Building more realistic environments for machine
     learning},
  author  = {Hardwick, Robert J},
  year    = {WIP},
  publisher = {umbralcalculations},
  url     = {https://umbralcalc.github.io/worlds-of-observation/book.pdf},
}
```

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

Table of contents

Part 1

1

Generalising simulation engines

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

1.1 Computational formalism

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

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

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; $X_{0:\mathtt{t}+1}$ is the next version of $X_{0:\mathtt{t}}$ after one timestep (and hence one new row has been added); z is a vector of arbitrary size which contains the 'hidden' other parameters that are necessary to iterate the process; and $F^i_{\mathtt{t}+1}(X_{0:\mathtt{t}}, z, \mathtt{t})$ as the latest element of an arbitrary matrix-valued function.

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

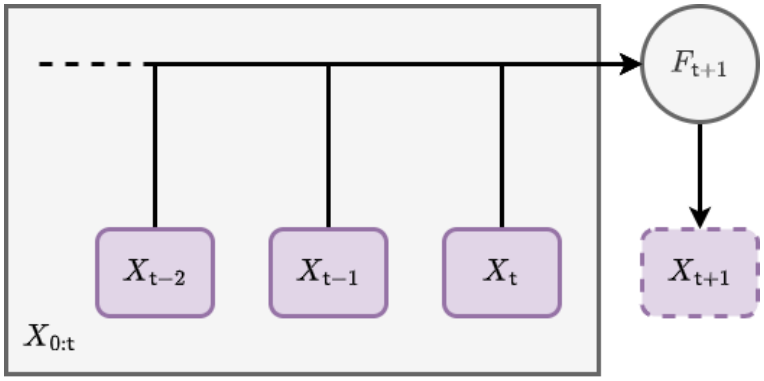

Figure 1.1: Graph representation of Eq. (1.1).

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 $X_{0:t}$ as an entire matrix input into a function which populates the elements of its latest rows. In pseudocode you could easily write something with the same idea in it, and it would probably look something like the method diagram in Fig. 1.2.

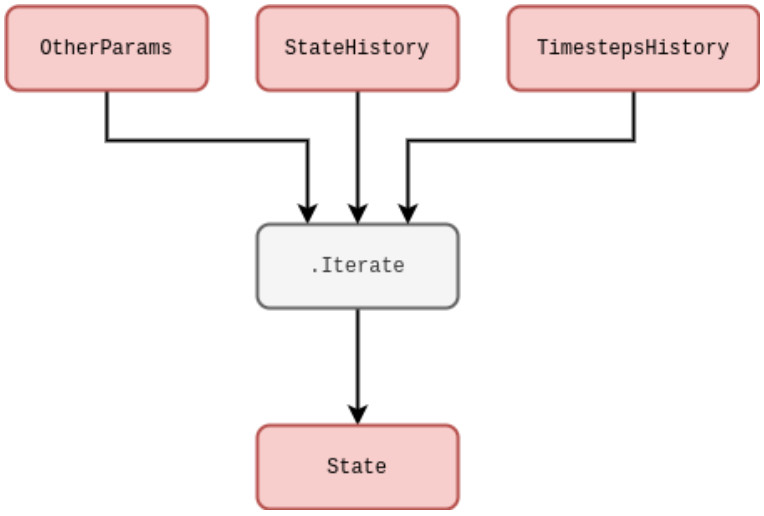

Figure 1.2: Pseudocode representation of Eq. (1.1).

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

[1] Or memory at least within some window.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

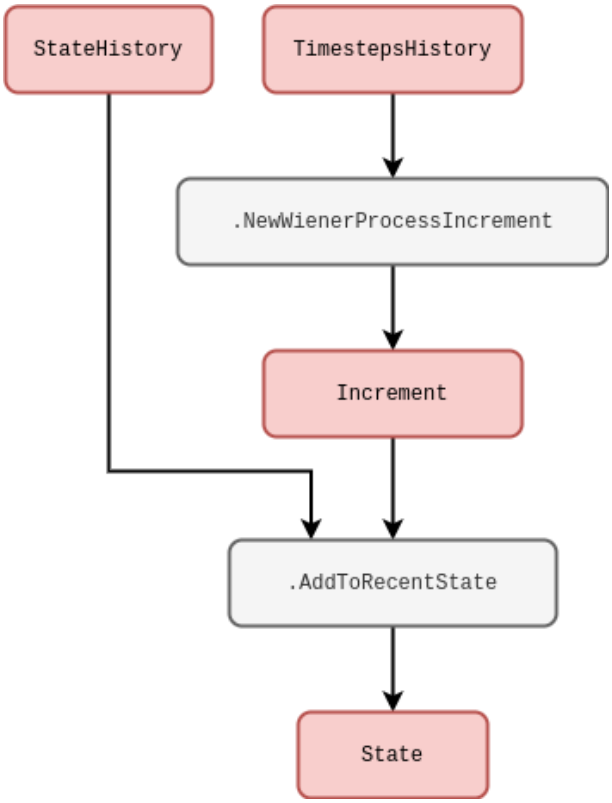

Figure 1.3: Schematic of code for a Wiener process.

our formalism too, e.g., Stratonovich[2] or others within the more general 'α-family' [7, 8, 9]. The pseudocode for any of these should hoepfully be fairly straightforward to deduce based on the lines we've already written above.

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

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

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

where $g(x,t)$ is some continuous function of its arguments which has been expanded out with Itô's Lemma on the second line. Note also that the computations in Eq. (1.8) could be performed with numerical derivatives in principle, even if the function were extremely complicated. This is unlikely to be the best way to describe the process of interest, however, the mathematical expressions above can still be made a bit more meaningful to the programmer in this way. The pseudocode in general would look something like Fig. 1.4.

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

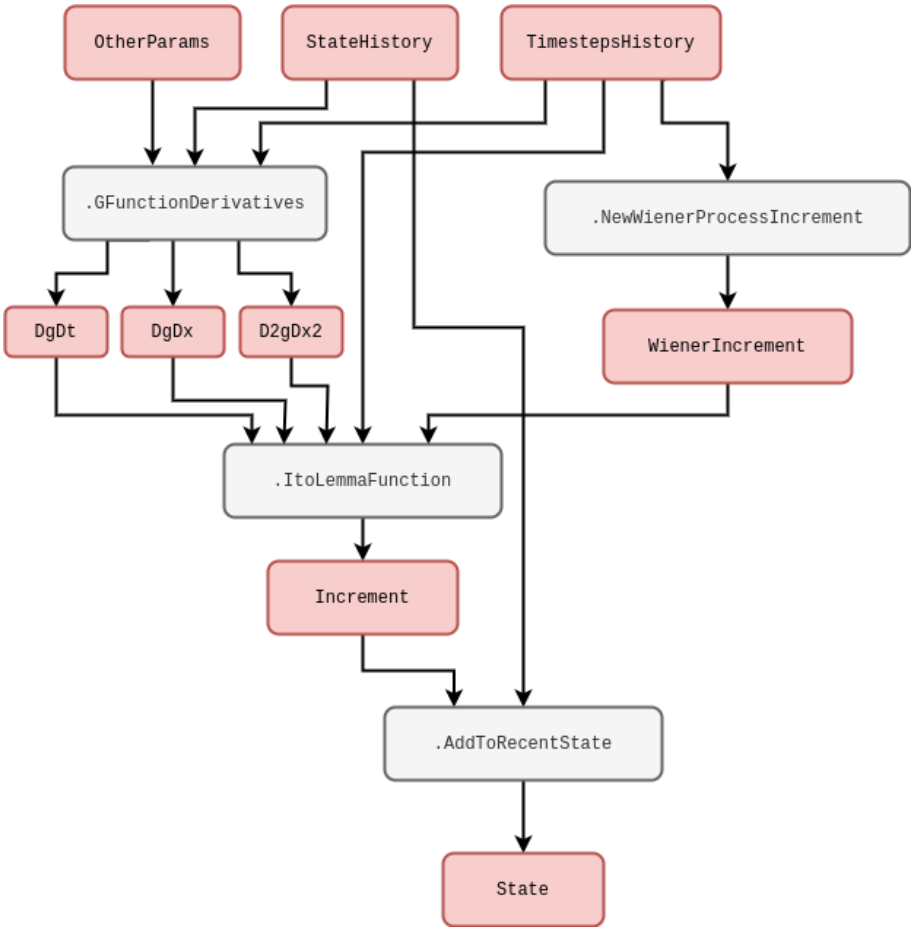

Figure 1.4: Schematic of code for Eq. (1.8).

Let's now look at a more complicated type of noise. For example, we might consider sampling from a *fractional Brownian motion* process $[B_H]_{\mathsf{t}}$, where H is known as the 'Hurst exponent'. Following Ref. [10], we can simulate this process in one of our state space dimensions by modifying the standard Wiener process by a fairly complicated integral factor which looks like this

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

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

So far we have mostly been discussing noises with continuous sample paths, but we can easily

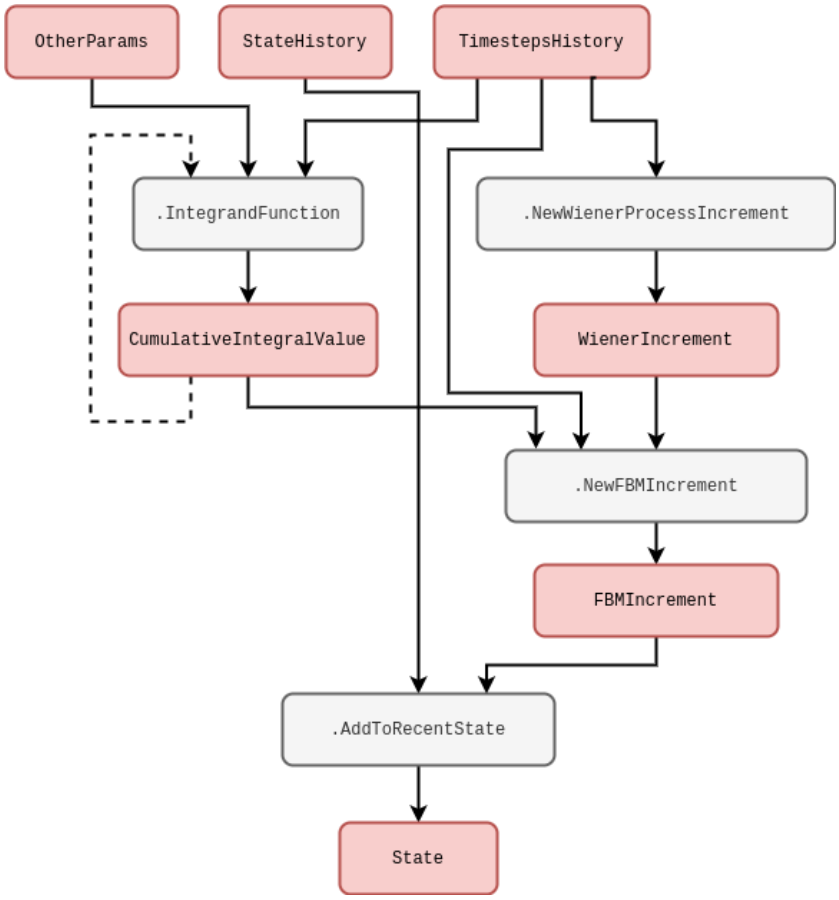

Figure 1.5: Schematic of code for Eq. (1.9).

adapt our computation to discontinuous sample paths as well. For instance, *Poisson process noises* would generally take the form

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

where $[N_\lambda]^i_t$ 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 $\delta t(t+1)$ 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 [11] — 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

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

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. 1.6 may also help to understand this concept from the programmer's perspective.

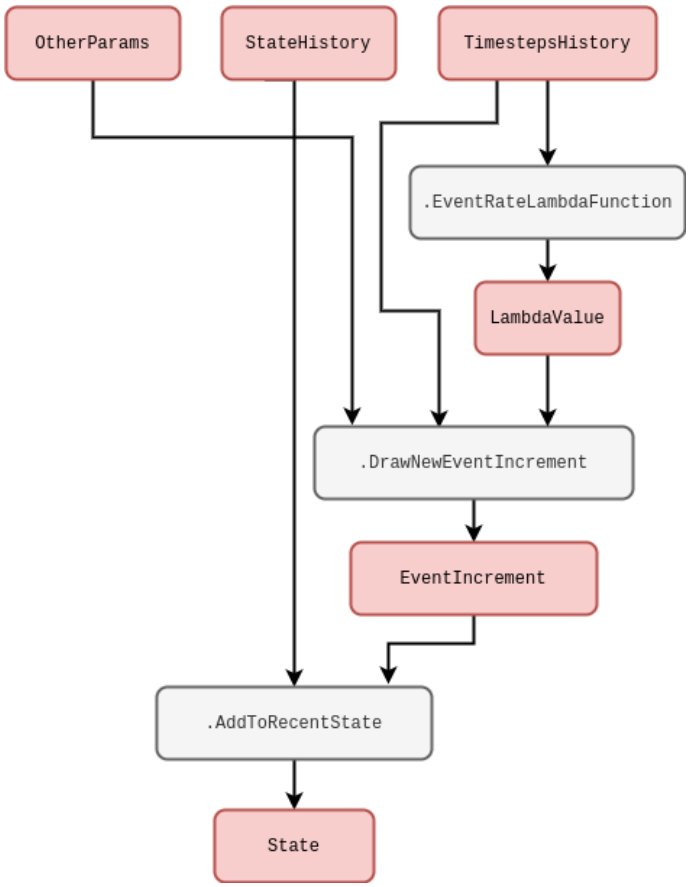

Figure 1.6: Schematic of code for an inhomogeneous Poisson process.

There are a few extensions to the simple Poisson process that introduce additional stochas-

tic processes. *Cox (doubly-stochastic) processes*, for instance, are basically where we replace the time-dependent rate $\lambda(\mathtt{t})$ with independent samples from some other stochastic process $\Lambda(\mathtt{t})$. For example, a Neyman-Scott process [12] 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

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

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

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 $\Lambda(\mathtt{t})$.

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

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

Note that the rejection method of Eq. (1.12) can be employed effectively to simulate any of these extensions as long as a sufficiently small τ is chosen. Once again, the 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

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

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

where the stochastic rate $\mathcal{I}_{\mathtt{t}+1}(X_{0:\mathtt{t}}, z, \mathtt{t})$ now depends on the history explicitly. Amongst other potential inputs we can see, e.g., Hawkes processes [13] as an example of above by substituting

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

where ϕ 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. 1.7.

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

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

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

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

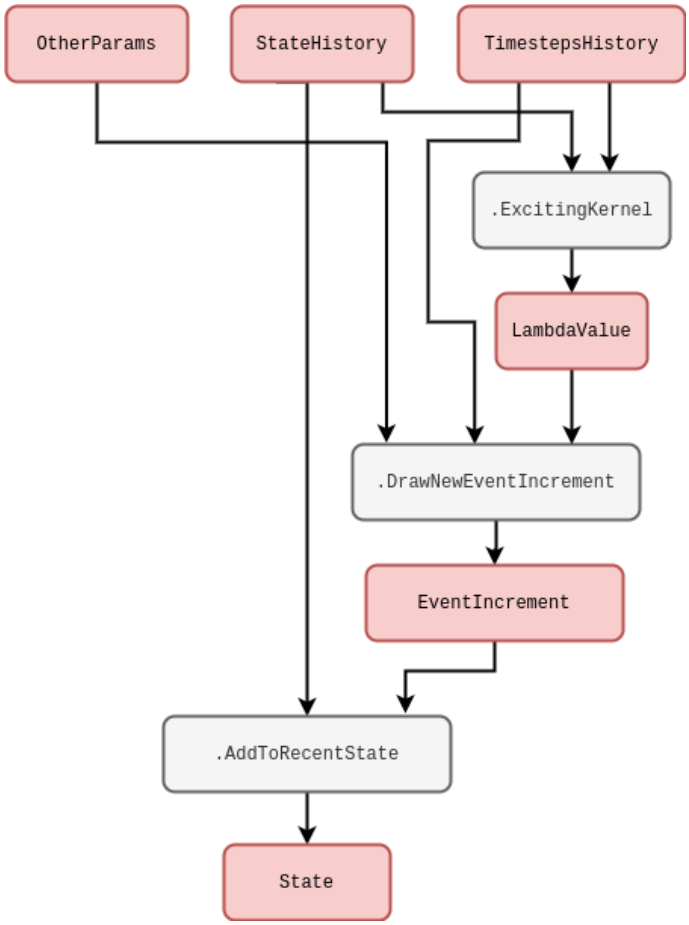

Figure 1.7: Schematic of code for a Hawkes process.

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.

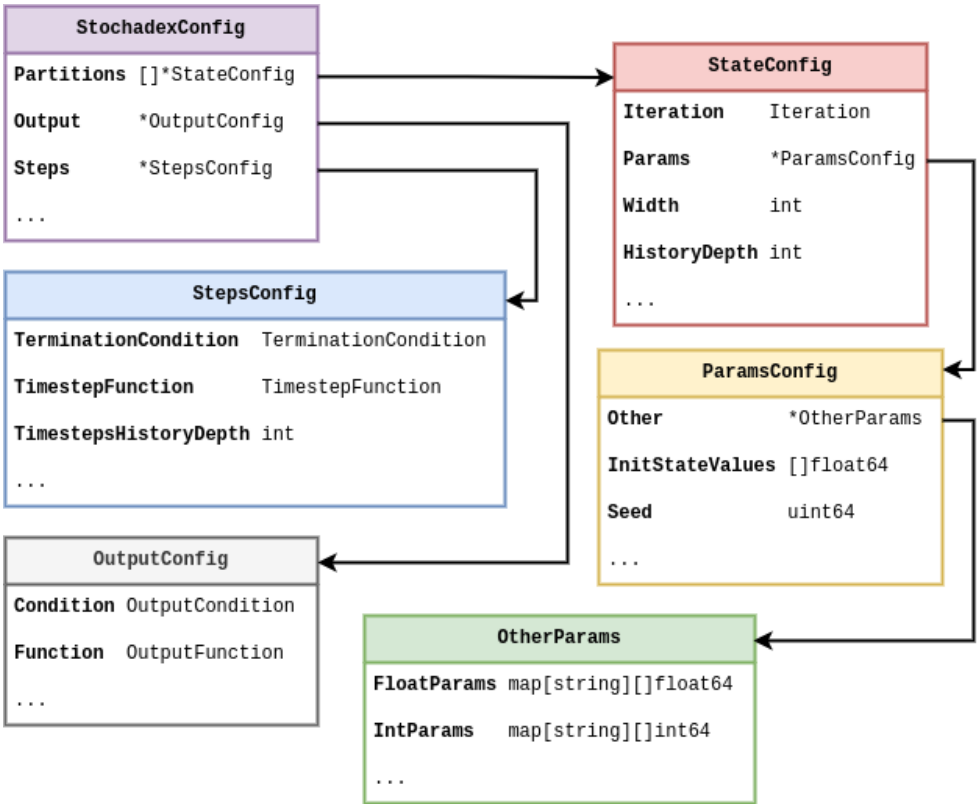

Figure 1.8: A relational summary of the configuration data types in the stochadex.

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. 1.8. 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 [14], StoSpa [15] and FLAME GPU [16] — have all approached this problem in different ways, and with different software architectures.

In Fig. 1.9 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.

It's also worth noting that while Fig. 1.9 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. 1.8). 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 [17] 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. 1.10.

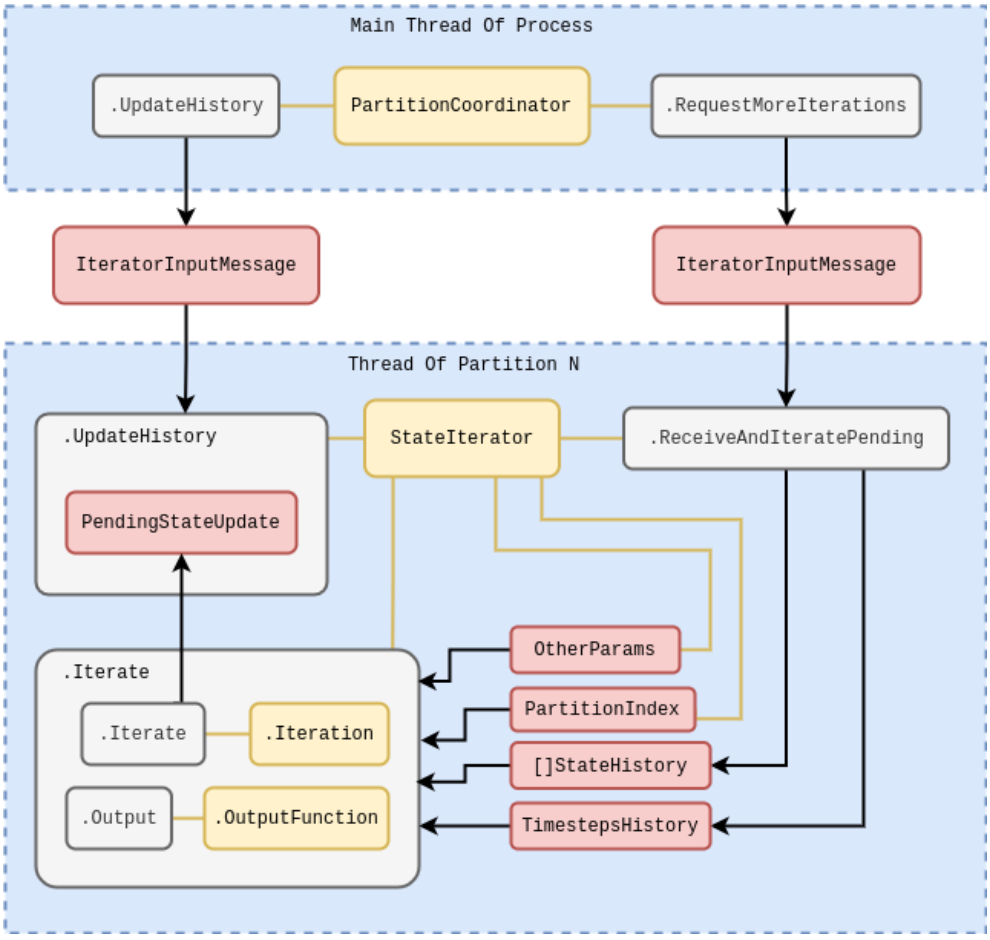

Figure 1.9: Schematic for a step of the stochadex simulation algorithm.

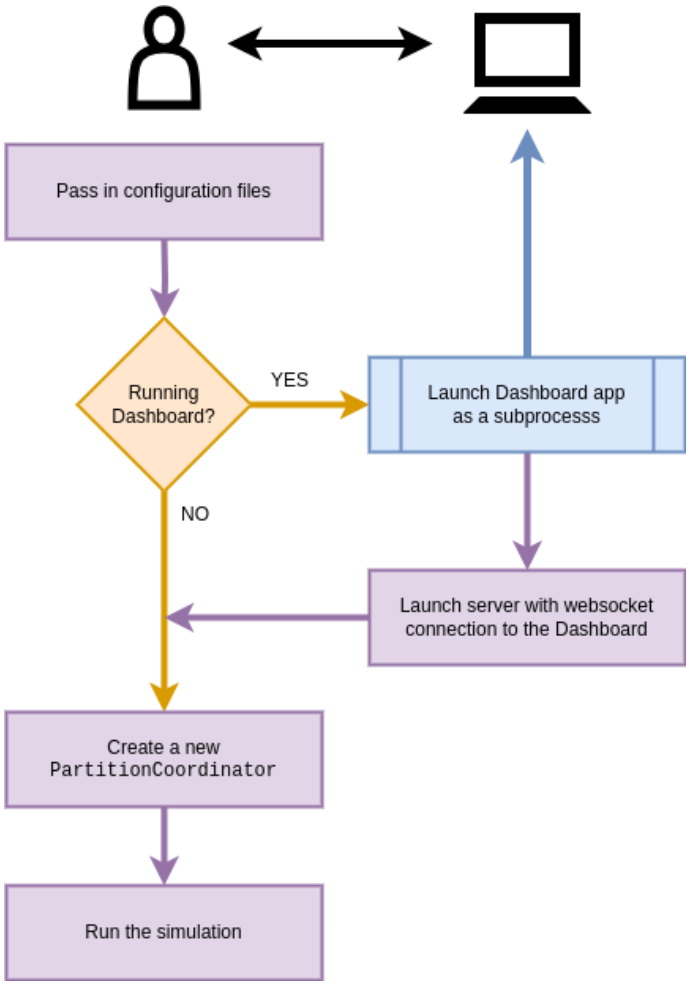

Figure 1.10: A diagram of the main stochadex binary executable.

2

Probabilistic learning methods

Concept. To extend the formalism that we developed in the previous chapter to describe the time evolution of state probabilities. Having introduced the basic concepts, we then use this formalism to motivate some important methods for probabilistic learning. In particular, we will see how Gaussian processes regression and 'empirical probabilistic reweighting' work, as well as discuss how some phenomena permit a convenient description via 'mean-field' equations or regression of distribution moments. For the mathematically-inclined, this chapter will take a detailed look at how our formalism can be extended to focus on the time evolution of probabilities, with a view to probabilistic learning later on. For the programmers, this chapter will mostly focus on the algorithm concepts, but all of the relevant software lives in this public Git repository: https://github.com/umbralcalc/learnadex.

2.1 Probabilistic formalism

This book is about building more realistic training environments for machine learning systems in the real world. In the last chapter we formalised, designed and built a generalised simulation engine which could serve as the essential scaffolding for these environments. So why then, in this chapter, do we want to extend our formalism to talk about probabilistic learning methods?

For many of these realistic environments where only partial state observations are possible, probabilistic learning tools can immediately become an essential tool to robustly infer the state of a system and predict its future states from any given point in time. Given that these tools are often so important for extending the capability of learning algorithms to deal with partially-observed domains; we propose to extend the idea of what is more conventionally considered a 'machine learning environment' to include tools for system state inference and future state prediction. In this sense, this chapter, and the remaining chapters of **Part 1** in this book, will consider probabilistic learning as an essential part of the environment for a machine learning system.

In this chapter, we're going to discuss the formal connections between some probabilistic learning methods and the simulation formalism we introduced in the last chapter. The novelties in this chapter are less to do with the probabilistic learning methodology and 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.

Let's start by returning to the formalism that we introduced in the previous chapter. As we discussed at that point; this formalism is appropriate for sampling from nearly every stochastic phenomenon that one can think of. We are going to extend this description to consider what happens to the probability that the state history matrix takes a particular set of values over time.

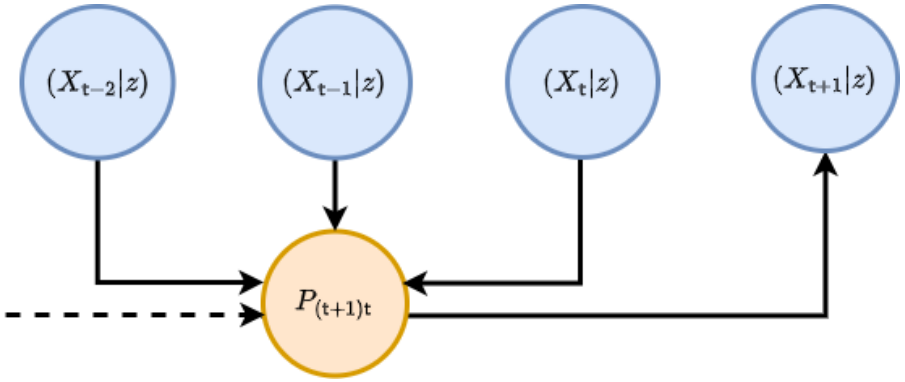

Figure 2.1: Graph representation of Eq. (2.1).

So, how do we begin? In the first chapter, we defined the general stochastic process with the formula $X^i_{t+1} = F^i_{t+1}(X_{0:t}, z, t)$. This equation also has an implicit *master equation* associated to it that fully describes the time evolution of the *probability density function* $P_{t+1}(X|z)$ of $X_{0:t+1} = X$ given that the parameters of the process are z. This can be written as

$$P_{t+1}(X|z) = P_t(X'|z)P_{(t+1)t}(x|X', z)\,, \tag{2.1}$$

where for the time being we are assuming the state space is continuous in each of the matrix elements and $P_{(t+1)t}(x|X', z)$ is the conditional probability that $X_{t+1} = x$ given that $X_{0:t} = X'$ at time t and the parameters of the process are z. To try and understand what Eq. (2.1) 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. We've also illustrated this kind of thinking in Fig. 2.1.

Consider what happens when we extend the chain of conditional probabilities in Eq. (2.1) back in time by one step. In doing so, we retrieve a joint probability of rows $X_{t+1} = x$ and $X_t = x'$ on the right hand side of the expression

$$P_{t+1}(X|z) = P_{t-1}(X''|z)P_{(t+1)t(t-1)}(x, x'|X'', z)\,. \tag{2.2}$$

Since Eqs. (2.1) and (2.2) are both valid ways to obtain $P_{t+1}(X|z)$ we can average between them without loss of generality in the original expression, like this

$$P_{t+1}(X|z) = \frac{1}{2}\left[P_t(X'|z)P_{(t+1)t}(x|X', z) + P_{t-1}(X''|z)P_{(t+1)t(t-1)}(x, x'|X'', z)\right]. \tag{2.3}$$

Following this line of reasoning to its natural conclusion, Eq. (2.1) can hence be generalised to consider all possible joint distributions of rows at different timesteps like this

$$P_{t+1}(X|z) = \frac{1}{t}\sum_{t''=0}^{t} P_{t''}(X''|z)P_{(t+1)t\ldots t''}(x, x', \ldots|X'', z)\,. \tag{2.4}$$

If we wanted to just look at the distribution over the latest row $X_{\mathsf{t}+1} = x$, we could achieve this through marginalisation over all of the previous matrix rows in Eq. (2.1) like this

$$P_{\mathsf{t}+1}(x|z) = \int_{\Omega_{\mathsf{t}}} \mathrm{d}X' P_{\mathsf{t}+1}(X|z) = \int_{\Omega_{\mathsf{t}}} \mathrm{d}X' P_{\mathsf{t}}(X'|z) P_{(\mathsf{t}+1)\mathsf{t}}(x|X', z)\,. \tag{2.5}$$

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

The symbol $\mathrm{d}X'$ in Eq. (2.5) is our shorthand notation throughout the book for computing the sum of integrals over previous state history matrices which can further be reduced via Eq. (2.4) into a product of sub-domain integrals over each matrix row

$$P_{\mathsf{t}+1}(x|z) = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}''=0}^{\mathsf{t}} \left\{ \int_{\omega_{\mathsf{t}'}} \mathrm{d}^n x' ... \int_{\Omega_{\mathsf{t}''}} \mathrm{d}X'' \right\} P_{\mathsf{t}''}(X''|z) P_{(\mathsf{t}+1)\mathsf{t}...\mathsf{t}''}(x, x', ...|X'', z) \tag{2.6}$$

$$= \frac{1}{\mathsf{t}} \sum_{\mathsf{t}''=0}^{\mathsf{t}} \int_{\Omega_{\mathsf{t}''}} \mathrm{d}X'' P_{\mathsf{t}''}(X''|z) P_{(\mathsf{t}+1)\mathsf{t}''}(x|X'', z)\,, \tag{2.7}$$

where each row measure is a Cartesian product of n elements (a Lebesgue measure), i.e.,

$$\mathrm{d}^n x = \prod_{i=0}^{n} \mathrm{d}x^i\,, \tag{2.8}$$

and lowercase $x, x', \ldots$ values will always refer to individual rows within the state matrices. Note that $1/\mathsf{t}$ here is a normalisation factor — this just normalises the sum of all probabilities to 1 given that there is a sum over t'. Note also that, if the process is defined over continuous time, we would need to replace

$$\frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \to \frac{1}{t(\mathsf{t})} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \delta t(\mathsf{t}')\,. \tag{2.9}$$

Let's go through some examples. Non-Markovian phenomena with continuous state spaces can have quite complex master equations. A relatively simple example is that of pure diffusion processes which exhibit stochastic resetting at a rate r to a remembered location from the trajectory history [18]

$$P_{\mathsf{t}+1}(x|z) = (1-r) P_{\mathsf{t}}(x|z) + \sum_{i=0}^{n} \sum_{j=0}^{n} \frac{\partial}{\partial x^i} \frac{\partial}{\partial x^j} \Big[D_{\mathsf{t}}(x, z) P_{\mathsf{t}}(x|z) \Big] + r \sum_{\mathsf{t}'=0}^{\mathsf{t}} \delta t(\mathsf{t}') K[t(\mathsf{t}) - t(\mathsf{t}')] P_{\mathsf{t}'}(x|z)\,, \tag{2.10}$$

where here K is some memory kernel. For Markovian phenomena which have a continuous state space, Eqs. (2.1) and (2.5) no longer depend on timesteps older than the immediately previous one, hence, e.g., Eq. (2.5) reduces to just

$$P_{\mathsf{t}+1}(x|z) = \int_{\omega_{\mathsf{t}}} \mathrm{d}^n x'\, P_{\mathsf{t}}(x'|z) P_{(\mathsf{t}+1)\mathsf{t}}(x|x', z)\,. \tag{2.11}$$

A famous example of this kind of phenomenon arises from approximating Eq. (2.11) with an expansion (Kramers-Moyal [19, 20]) up to second-order, yielding the Fokker-Planck equation

$$P_{\mathsf{t}+1}(x|z) = P_{\mathsf{t}}(x|z) - \sum_{i=0}^{n} \frac{\partial}{\partial x^i}\left[\mu_{\mathsf{t}}(x,z)P_{\mathsf{t}}(x|z)\right] + \sum_{i=0}^{n}\sum_{j=0}^{n} \frac{\partial}{\partial x^i}\frac{\partial}{\partial x^j}\left[D_{\mathsf{t}}(x,z)P_{\mathsf{t}}(x|z)\right], \tag{2.12}$$

which describes a process undergoing drift-diffusion.

An analog of Eq. (2.5) 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

$$P_{\mathsf{t}+1}(x|z) = \sum_{\Omega_{\mathsf{t}}} P_{\mathsf{t}}(X'|z)P_{(\mathsf{t}+1)\mathsf{t}}(x|X',z)\,, \tag{2.13}$$

where we note that the P's in the expression above all now refer to *probability mass functions*. In what follows, discrete state space can always be considered by replacing the integrals with summations over probability masses in this manner; we only use the continuous state space formulation for our notation because one could argue it's a little more general.

Analogously to continuous state spaces, we can give some examples of master equations for phenomena with a discrete state space as well. In the Markovian case, we need look no further than a simple time-dependent Poisson process

$$P_{\mathsf{t}+1}(x|z) = \lambda(\mathsf{t})\delta t(\mathsf{t}{+}1)P_{\mathsf{t}}(x{-}1|z) + \big[1 - \lambda(\mathsf{t})\delta t(\mathsf{t}{+}1)\big]P_{\mathsf{t}}(x|z)\,. \tag{2.14}$$

For such an example of a non-Markovian system, a Hawkes process [13] master equation would look something like this

$$\begin{aligned} P_{\mathsf{t}+1}(x|z) = {} & \mu\delta t(\mathsf{t}{+}1)P_{\mathsf{t}}(x{-}1|z) + \big[1 - \mu\delta t(\mathsf{t}{+}1)\big]P_{\mathsf{t}}(x|z) \\ & + \sum_{x'=0}^{\infty}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \phi[t(\mathsf{t}) - t(\mathsf{t}')]\delta t(\mathsf{t}{+}1)P_{\mathsf{t}\mathsf{t}'(\mathsf{t}'-1)}(x{-}1, x', x'{-}1|z) \\ & + \sum_{x'=0}^{\infty}\left\{1 - \sum_{\mathsf{t}'=0}^{\mathsf{t}} \phi[t(\mathsf{t}) - t(\mathsf{t}')]\delta t(\mathsf{t}{+}1)\right\}P_{\mathsf{t}\mathsf{t}'(\mathsf{t}'-1)}(x, x', x'{-}1|z)\,, \end{aligned} \tag{2.15}$$

where we note the complexity in this expression arises because it has to include a coupling between the rate at which events occur and an explicit memory of when the previous ones did occur (recorded by differencing the count between adjacent timesteps by 1).

2.2 Motivating probabilistic learning

So now that we are more familiar with the notation used by Eq. (2.5), we can use it to motivate some useful probabilistic learning methods. While it's worth going into some mathematical detail to give a better sense of where each technique comes from, we should emphasise that the methodologies we discuss here are not new to the technical literature at all. We draw on influences from Empirical Dynamical Modeling (EDM) [21], some classic nonparametric local regression techniques — such as LOWESS/Savitzky-Golay filtering [22] — and also Gaussian processes [23].

Let's begin our discussion of algorithms by integrating Eq. (2.5) over x to obtain a relation for the mean of the distribution

$$M_{\mathtt{t}+1}(z) = \int_{\omega_{\mathtt{t}+1}} \mathrm{d}^n x \, x \, P_{\mathtt{t}+1}(x|z) = \frac{1}{\mathtt{t}} \sum_{\mathtt{t}''=0}^{\mathtt{t}} \int_{\Omega_{\mathtt{t}''}} \mathrm{d}X'' P_{\mathtt{t}''}(X''|z) M_{(\mathtt{t}+1)\mathtt{t}''}(X'', z) \,, \tag{2.16}$$

where you can view the $M_{(\mathtt{t}+1)\mathtt{t}''}(X'', z)$ values as either terms in some regression model, or derivable explicitly from a known master equation. The latter of these provides one approach to statistically infer the states and parameters of stochastic simulations from data: one begins by knowing what the master equation is, uses this to compute the time evolution of the mean (and potentially higher-order statistics) and then connects these $\mathtt{t}$ and z-dependent statistics back to the likelihood of observing the data. This is what is commonly known as the 'mean-field' inference approach; averaging over the available degrees of freedom in the statistical moments of distributions. Though, knowing what the master equation is for an arbitrarily-defined stochastic phenomenon can be very difficult indeed, and the resulting equations typically require some form of approximation.

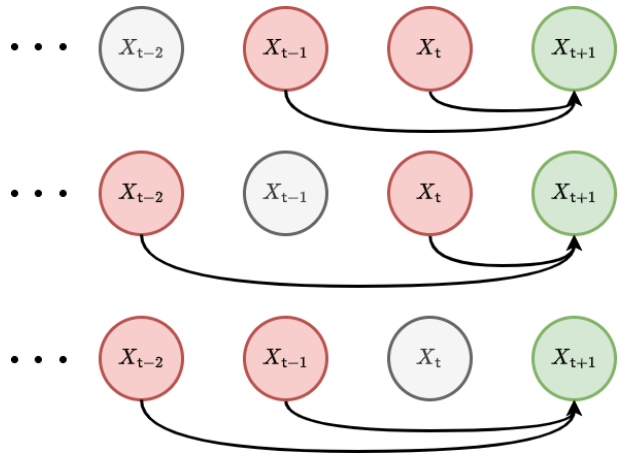

Figure 2.2: A graph representation of the correlations in Eq. (2.21).

Given that the mean-field approach isn't always going to be viable as an inference method, we should also consider other ways to describe the shape and time evolution characteristics of $P_{\mathtt{t}+1}(X|z)$. For continuous state spaces, it's possible to approximate this whole distribution with a logarithmic expansion like so

$$\ln P_{\mathtt{t}+1}(X|z) \simeq \ln P_{\mathtt{t}+1}(X_*|z) + \frac{1}{2} \sum_{\mathtt{t}'=0}^{\mathtt{t}+1} \sum_{i=0}^{n} \sum_{j=0}^{n} (x - x_*)^i \mathcal{H}^{ij}_{(\mathtt{t}+1)\mathtt{t}'}(z)(x' - x'_*)^j \tag{2.17}$$

$$\mathcal{H}^{ij}_{(\mathtt{t}+1)\mathtt{t}'}(z) = \frac{\partial}{\partial x^i} \frac{\partial}{\partial (x')^j} \ln P_{\mathtt{t}+1}(X|z) \Big|_{X=X_*} \,, \tag{2.18}$$

where the values for X_* (and its rows $x_*, x'_*, \dots$) are defined by the vanishing of the first derivative,

i.e., these are chosen such that

$$\frac{\partial}{\partial x^i} \ln P_{\mathsf{t}+1}(X|z)\Big|_{X=X_*} = 0\,. \tag{2.19}$$

This logarithmic expansion is one way to see how a Gaussian process regression is able to approximate X_t evolving in time for many different processes. By selecting the appropriate function for the kernel given by Eq. (2.18), a Gaussian process regression can be fully specified.

If we keep the truncation up to second order in Eq. (2.17), note that this expression implies a pairwise correlation structure of the form

$$P_{\mathsf{t}+1}(X|z) \to \prod_{\mathsf{t}'=0}^{\mathsf{t}} P_{(\mathsf{t}+1)\mathsf{t}'}(x, x'|z) = \prod_{\mathsf{t}'=0}^{\mathsf{t}} P_{\mathsf{t}'}(x'|z) P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)\,. \tag{2.20}$$

Given this pairwise temporal correlation structure, Eq. (2.7) reduces to this simpler sum of integrals

$$P_{\mathsf{t}+1}(x|z) = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}^n x' P_{\mathsf{t}'}(x'|z) P_{(\mathsf{t}+1)\mathsf{t}'}(x|x', z)\,. \tag{2.21}$$

We have illustrated these second-order correlations with a graph visualisation in Fig. (2.2).

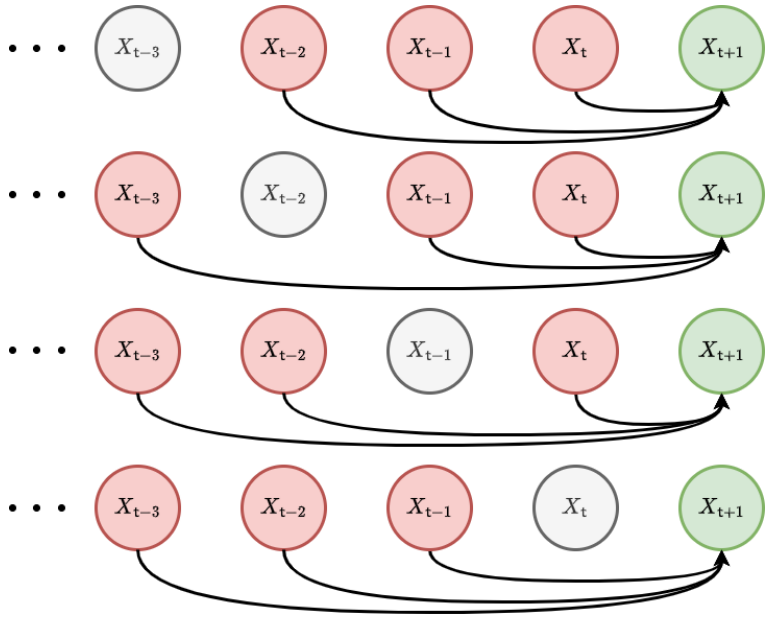

Figure 2.3: A graph representation of the correlations in Eq. (2.23).

In a similar fashion, we can increase the expansion order of Eq. (2.17) to include third-order correlations such that

$$P_{\mathsf{t}+1}(X|z) \to \prod_{\mathsf{t}'=0}^{\mathsf{t}} \prod_{\mathsf{t}''=0}^{\mathsf{t}'-1} P_{\mathsf{t}'\mathsf{t}''}(x', x''|z) P_{(\mathsf{t}+1)\mathsf{t}'\mathsf{t}''}(x|x', x'', z)\,, \tag{2.22}$$

and, in this instance, one can show that Eq. (2.7) reduces to

$$P_{\mathsf{t}+1}(x|z) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\frac{1}{\mathsf{t}'-1}\sum_{\mathsf{t}''=0}^{\mathsf{t}'-1}\int_{\omega_{\mathsf{t}'}}\mathrm{d}^n x'\int_{\omega_{\mathsf{t}''}}\mathrm{d}^n x'' P_{\mathsf{t}'\mathsf{t}''}(x',x''|z)P_{(\mathsf{t}+1)\mathsf{t}'\mathsf{t}''}(x|x',x'',z)\,. \tag{2.23}$$

We have also illustrated these third-order correlations with another graph visualisation in Fig. (2.3). Using $P_{\mathsf{t}'\mathsf{t}''}(x',x''|z) = P_{\mathsf{t}''}(x''|z)P_{\mathsf{t}'\mathsf{t}''}(x'|x'',z)$ one can also show that this integral is a marginalisation of this expression

$$P_{(\mathsf{t}+1)\mathsf{t}''}(x|x'',z) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\int_{\omega_{\mathsf{t}'}}\mathrm{d}^n x' P_{\mathsf{t}'\mathsf{t}''}(x'|x'',z)P_{(\mathsf{t}+1)\mathsf{t}'\mathsf{t}''}(x|x',x'',z)\,, \tag{2.24}$$

which describes the time evolution of the conditional probabilities. Note how this implies that the Gaussian process kernel itself can be evolved through time to replicate these higher-order temporal correlations for a regression problem, if desired.

Another probabilistic learning algorithm that we can consider is what we shall call 'empirical probabilistic reweighting'. There is another expression for the mean of the distribution, that we can derive under certain conditions, which will be valuable to motivating this algorithm. If the probability distribution over each row of the state history matrix is *stationary* — meaning that $P_{\mathsf{t}+1}(x|z) = P_{\mathsf{t}'}(x|z)$ — it's possible to go one step further than Eq. (2.16) and assert that

$$M_{\mathsf{t}+1}(z) = \int_{\omega_{\mathsf{t}+1}}\mathrm{d}^n x\, x\, P_{\mathsf{t}+1}(x|z) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\int_{\omega_{\mathsf{t}'}}\mathrm{d}^n x'\, x'\, P_{\mathsf{t}'}(x'|z)\int_{\omega_{\mathsf{t}+1}}\mathrm{d}^n x\, P_{(\mathsf{t}+1)\mathsf{t}'}(x|x',z)\,. \tag{2.25}$$

To see that Eq. (2.25) is true, first note that a joint distribution over both x and x' can be derived like this $P_{(\mathsf{t}+1)\mathsf{t}'}(x,x'|z) = P_{(\mathsf{t}+1)\mathsf{t}'}(x|x',z)P_{\mathsf{t}'}(x'|z)$. Secondly, note that this joint distribution will 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

$$\begin{aligned}
\frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\int_{\omega_{\mathsf{t}+1}}\mathrm{d}^n x\int_{\omega_{\mathsf{t}'}}\mathrm{d}^n x'\, x\, P_{(\mathsf{t}+1)\mathsf{t}'}(x,x'|z) &= \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\int_{\omega_{\mathsf{t}'}}\mathrm{d}^n x\int_{\omega_{\mathsf{t}+1}}\mathrm{d}^n x'\, x\, P_{\mathsf{t}'(\mathsf{t}+1)}(x,x'|z)\\
&= \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\int_{\omega_{\mathsf{t}'}}\mathrm{d}^n x'\int_{\omega_{\mathsf{t}+1}}\mathrm{d}^n x\, x'\, P_{(\mathsf{t}+1)\mathsf{t}'}(x,x'|z)\\
&= \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\int_{\omega_{\mathsf{t}'}}\mathrm{d}^n x'\, x'\, P_{\mathsf{t}'}(x'|z)\int_{\omega_{\mathsf{t}+1}}\mathrm{d}^n x\, P_{(\mathsf{t}+1)\mathsf{t}'}(x|x',z)\,,
\end{aligned}$$

where we've used the trivial variable swap and integration variable relabelling to arrive at the second equality in the expressions above.

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

$$\begin{aligned}
C^{ij}_{\mathsf{t}+1}(z) &= \int_{\omega_{\mathsf{t}+1}}\mathrm{d}^n x\,[x - M_{\mathsf{t}+1}(z)]^i[x - M_{\mathsf{t}+1}(z)]^j P_{\mathsf{t}+1}(x|z)\\
&= \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}}\int_{\omega_{\mathsf{t}'}}\mathrm{d}^n x'\,[x' - M_{\mathsf{t}+1}(z)]^i[x' - M_{\mathsf{t}+1}(z)]^j\, P_{\mathsf{t}'}(x'|z)\int_{\omega_{\mathsf{t}+1}}\mathrm{d}^n x\, P_{(\mathsf{t}+1)\mathsf{t}'}(x|x',z)\,.
\end{aligned} \tag{2.26}$$

While they look quite abstract, Eqs. (2.25) and (2.26) express the core idea behind how the probabilistic reweighting will function. By assuming a stationary distribution, we gain the ability to directly estimate the statistics of the probability distribution of the next sample from the stochastic process $P_{\mathsf{t}+1}(x|z)$ from past samples it may have in empirical data; which are represented here by $P_{\mathsf{t}'}(x'|z)$. More on this later.

2.3 Learning algorithms

Probabilistic reweighting depends on the stationarity of $P_{\mathsf{t}+1}(x|z) = P_{\mathsf{t}'}(x|z)$ such that, e.g., Eq. (2.25) is applicable. The core idea behind it is to represent the past distribution of state values $P_{\mathsf{t}'}(x'|z)$ with the samples from a real time series dataset. If the user then specifies a good model for the relationships in this data by providing a weighting function which returns the conditional probability mass

$$\mathsf{w}_{\mathsf{t}'}(y,z) = \int_{\omega_{\mathsf{t}+1}} \mathrm{d}^n x \, P_{(\mathsf{t}+1)\mathsf{t}'}(x|x'{=}y,z)\,, \tag{2.27}$$

we can apply this as a *reweighting* of the historical time series samples to estimate any statistics of interest. Taking Eqs. (2.25) and (2.26) as the examples; we are essentially approximating these integrals through weighted sample estimations like this

$$M_{\mathsf{t}+1}(z) \simeq \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} Y_{\mathsf{t}'} \mathsf{w}_{\mathsf{t}'}(Y_{\mathsf{t}'}, z) \tag{2.28}$$

$$C^{ij}_{\mathsf{t}+1}(z) \simeq \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} [Y_{\mathsf{t}'} - M_{\mathsf{t}+1}(z)]^i [Y_{\mathsf{t}'} - M_{\mathsf{t}+1}(z)]^j \, \mathsf{w}_{\mathsf{t}'}(Y_{\mathsf{t}'}, z)\,, \tag{2.29}$$

where we have defined the data matrix Y with rows $Y_{\mathsf{t}+1}, Y_{\mathsf{t}}, \ldots$, each of which representing specific observations of the rows in X at each point in time from a real dataset.

The goal of a learning algorithm for probabilistic reweighting would be to learn the optimal reweighting function $\mathsf{w}_{\mathsf{t}'}(Y_{\mathsf{t}'}, z)$ with respect to z, i.e., the ones which most accurately represent a provided dataset. But before we think about the various kinds of conditional probability we could use, we need to think about how to connect the post-reweighting statistics to the data by defining an objective function.

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. (2.25), 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. (2.26). These are just a few simple examples of distributions that can link the estimated statistics from Eqs. (2.25) and (2.26) 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. We can define $P_{\mathsf{t}+1}[y; M_{\mathsf{t}+1}(z), C_{\mathsf{t}+1}(z), \ldots]$ as representing the likelihood of $y = Y_{\mathsf{t}+1}$ given the estimated statistics from Eqs. (2.25) and (2.26) (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., $M_{\mathtt{t}+1}(z)$ 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

$$\ln \mathcal{L}_{\mathtt{t}+1}(Y|z) = \sum_{\mathtt{t}'=0}^{\mathtt{t}+1} \ln P_{\mathtt{t}'}[y; M_{\mathtt{t}'}(z), C_{\mathtt{t}'}(z), \dots], \tag{2.30}$$

where the summation continues until all of the past measurements $Y_{\mathtt{t}+1}, Y_{\mathtt{t}}, \dots$ which exist as rows in the data matrix Y have been taken into account. The code to compute this objective function follows the schematic we have provided in Fig. 2.4.

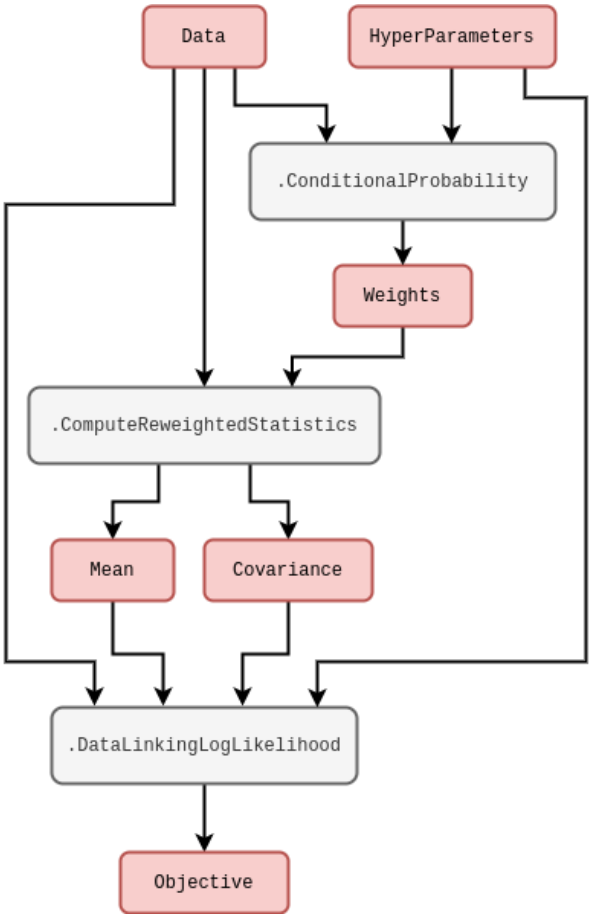

Figure 2.4: Code schematic of the probability reweighting objective computation.

In order to specify what $P_{(\mathtt{t}+1)\mathtt{t}'}(x|x', z)$ is, it's quite natural to define a set of hyperparameters for the elements of z. To get a sense of how the data-linking function relates to these hyperparameters, it's instructive to consider an example. One generally-applicable option for the conditional

probability could be a purely time-dependent kernel

$$P_{(\mathsf{t}+1)\mathsf{t}'}(x|x',z) \propto \mathcal{K}(z,\mathsf{t}+1,\mathsf{t}')\,, \tag{2.31}$$

and the data-linking distribution, e.g., could be a Gaussian

$$P_{\mathsf{t}+1}[y; M_{\mathsf{t}+1}(z), C_{\mathsf{t}+1}(z), \dots] = \mathsf{MultivariateNormalPDF}[y; M_{\mathsf{t}+1}(z), C_{\mathsf{t}+1}(z)]\,. \tag{2.32}$$

It's worth pointing out that other machine learning frameworks could easily be used to model these conditional probabilities. For example, neural networks could be used to infer the optimal reweighting scheme and this would still allow us to use the data-linking distribution.[1] It would still be desirable to keep the data-linking distribution as it can usually be sampled from very easily — something that can be quite difficult to achieve with a purely machine learning-based representation of the distribution. Sampling itself could even be made more flexible by leveraging a Variational Autoencoder (VAE) [25]; these use neural networks not just on the compression (or 'encode') step to estimate the statistics but also use them as a layer between the sample from the data distribution model and the output (the 'decode' step).

In the case of Eqs. (2.31) and (2.32) above, the hyperparameters that would be optimised could relate to the kernel in a wide variety of ways. Optimising them would make our optimised reweighting similar to (but very much *not* the same as) evaluating maximum a posteriori (MAP) of a Gaussian process regression. In a Gaussian process regression, one is concerned with inferring the the whole of X_{t} as a function of time using the pairwise correlations implied by Eq. (2.17). Based on this expression, the cumulative log-likelihood for a Gaussian process can be calculated as follows

$$\ln \mathcal{L}_{\mathsf{t}+1}(Y|z) = -\frac{n}{2}\ln(2\pi) - \frac{1}{2}\ln|\mathcal{H}(z)| - \frac{1}{2}\sum_{\mathsf{t}'=0}^{\mathsf{t}+1}\sum_{i=0}^{n}\sum_{j=0}^{n} y^i \mathcal{H}^{ij}_{(\mathsf{t}+1)\mathsf{t}'}(z)(y')^j\,. \tag{2.33}$$

As we did for the reweighting algorithm, in Fig. 2.5 we have illustrated a schematic of the basic code needed to compute the objective function of a learning algorithm based on Eq. (2.33). Note that, in the expression above, we have replaced $x - x_*$ with y and so it is assumed that the data has already been shifted such that its values are positioned around the distribution peak. Knowing where this peak will be a priori is not possible. However, for Gaussian data, an unbiased estimator for this peak will be the sample mean and so we have included an initial data standardisation in the steps outlined by Fig. 2.5.

The optimisation approach that we choose to use for obtaining the best hyperparameters in the conditional probability of Eq. (2.30) 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 [26] method or something like a particle swarm [27, 28]) 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 [29] (SGD) or Adam [30].

If the gradients of Eq. (2.30) are needed, we can always factorise each derivative with respect

[1] One can think of using this neural network-based reweighting scheme as similar to constructing a normalising flow model [24] with an autoregressive layer. Invertibility and further network structural constraints mean that these are not exactly equivalent, however.

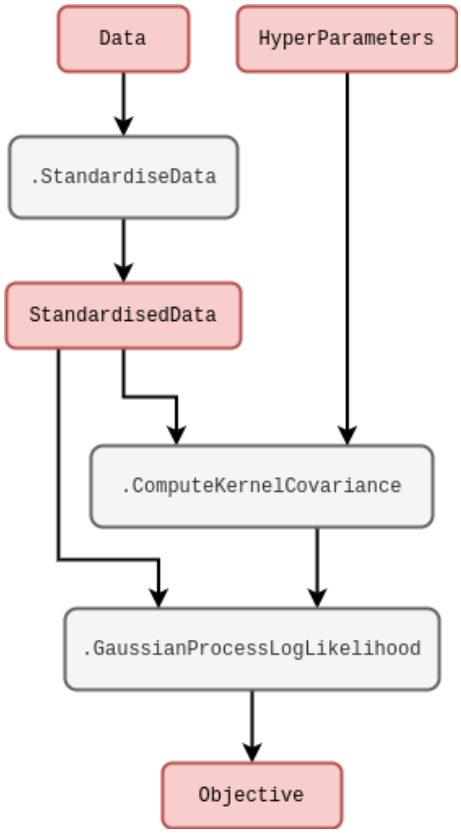

Figure 2.5: Code schematic of the Gaussian process objective computation.

to hyperparameter z^i in the following way through the chain rule

$$\frac{\partial}{\partial z^i} \ln \mathcal{L}_{t+1}(Y|z) = \sum_{t'=0}^{t+1} \frac{\partial M_{t'}}{\partial z^i} \frac{\partial}{\partial M_{t'}} \ln P_{t'}[y; M_{t'}(z), C_{t'}(z), \dots] + \sum_{t'=0}^{t+1} \frac{\partial C_{t'}}{\partial z^i} \frac{\partial}{\partial C_{t'}} \ln P_{t'}[y; M_{t'}(z), C_{t'}(z), \dots] . \tag{2.34}$$

By factoring derivatives in this manner, the computation can be separated into two parts: the derivatives with respect to $M_{t'}$ and $C_{t'}$, 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.

The same logic should apply to a learning algorithm which optimises z to obtain the MAP for a Gaussian process. If the gradients of Eq. (2.33) are required, the user would have to specify

derivatives of the kernel matrix (and its determinant) with respect to z in order to calculate

$$\frac{\partial}{\partial z^k} \ln \mathcal{L}_{\mathfrak{t}+1}(Y|z) = -\frac{1}{2} \frac{\partial}{\partial z^k} \ln |\mathcal{H}(z)| - \frac{1}{2} \sum_{\mathfrak{t}'=0}^{\mathfrak{t}+1} \sum_{i=0}^{n} \sum_{j=0}^{n} y^i \frac{\partial}{\partial z^k} \mathcal{H}^{ij}_{(\mathfrak{t}+1)\mathfrak{t}'}(z)(y')^j \,. \tag{2.35}$$

3

Online learning software

Concept. To outline the software design characteristics of a probabilistic online learning framework, built to interface directly with our generalised simulation engine. The probabilistic algorithms we introduced in the previous chapter will form the basic set of tools in this framework. As well as its own implementations of some learning algorithms, the software is also designed to enable interoperability with other machine learning APIs, e.g., Libtorch (via Gotch), and optimisation libraries, e.g., Gonum and eaopt. For the mathematically-inclined, this chapter will motivate online learning as an essential framework for our use case and how our probabilistic algorithms can work within this context. For the programmers, the software designed and described in this chapter lives in this public Git repository: https://github.com/umbralcalc/learnadex.

3.1 Online learning with any algorithm

We have discussed the importance of probabilistic learning in the context of environments where only partial state observability is possible and, in the previous chapter, we motivated the use of some specific probabilistic learning methods. However, we haven't yet discussed how we might implement a learning algorithm in practice. In particular, before covering the various aspects of software design, it's important to consider how we want to structure learning by optimisation of an objective with respect to a stream of time series data.

One of the issues that can arise when learning streams of data is 'concept drift'. In our context, this would be when the optimal value for z does not match the optimal value at some later point in time. In order to mitigate this, our learning algorithms should be able to track an up-to-date optimal value for z as data is continually passed into them. Iteratively updating the optimal parameters as new data is ingested into the objective function is typically called 'online learning' [31, 1], in contrast to 'offline learning' which would correspond to learning an optimal z only once with the entire dataset provided upfront.

The reader may recall that this book is about building more realistic environments for machine learning systems. An important part of learning from environments in a robust manner is ensuring *adaptability to new data*. In addition to this, stochastic processes are inherently sequential. Many

types of system evolve not just their states, but also dynamical description, over time. Online learning is the natural framework to use in this context.

Recall the models we discussed in the previous chapter which optimise the cumulative log-likelihood of the data matrix Y with respect to z at a particular point in time, i.e., optimise $\ln \mathcal{L}_{\mathtt{t}+1}(Y|z)$ with respect to z at time $\mathtt{t}+1$. The simplest (and most generally applicable) way to implement an online learning approach in these probabilistic learning models is to rerun the whole optimisation algorithm for z after each new datapoint has been received. For algorithms which take a long time to run in each case, this can be an extremely slow procedure.

One way to speed things up is to assume that the optimal value for z which was obtained from the previous data iterations can be related to the one we will find in the most recent iteration. We can formalise this idea into a probabilistic statement. Take the β-weighting idea and introduce it here...

Got to here in rewrite... Still need to discuss:

- the importance of gradients
- batch learning algorithms — more like Gaussian processes
- 'pure' online learning algorithms — more like empirical probabilistic reweighting
- introduce the β past discounting factor in this section and explain what it's for
- refactor the code so that it's always doing online learning under the hood — this can either be rolling refits in blocks on a refitting schedule with any optimisation algorithm of choice or full online learning Adam optimisation

$$z_*(\mathtt{t}+1) = -\alpha[\mathtt{t}+1, \text{stats of gradient history like Adam}]\frac{\partial}{\partial z}\ln \mathcal{L}_{\mathtt{t}+1} + z_*(\mathtt{t})$$

Note that other excellent online machine learning frameworks are available — see, e.g., River [32] and Vowpal Wabbit [33]. The motivation for designing our own probabilistic online learning software is to ensure maximal integration with the stochadex simulation engine. We'll aim to achieve this by designing the code to use the same data structures and and concepts as we used when building the stochadex, where possible. In the next section on software design, we will show how this can be done while still maintaining extensibility and interoperability with other machine learning libraries and APIs. So let's get on with it!

3.2 Software design

Let's now take a step back from the specifics of the probabilistic reweighting 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.

As we discussed at the end of the last section, the algorithms in the learnadex are all applied in an 'online' fashion — refitting for the optimal hyperparameters z as new data is streamed into them. A challenging aspect of online learning is in managing the computational expense of recomputing

the optimal value for z after each new datapoint is sent. To help with this; the user may configure the algorithm recompute the optimum value after larger batches of data have been ingested. The last value of optimum z will also frequently be close to the next optimum in the sequence, so using the former as the initial input into the optimisation routine for the latter is typically very valuable for aiding efficiency.

Reusing the `PartitionCoordinator` code of the stochadex to facilitate online learning 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. 3.1. 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.

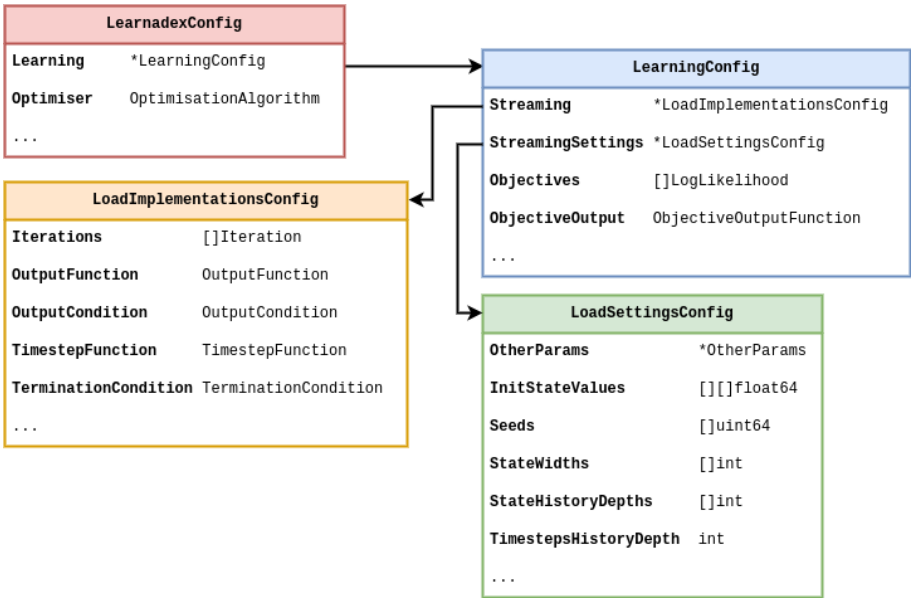

Figure 3.1: A relational summary of the core data types in the learnadex.

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. 3.2 below, we provide a schematic of the method calls of (and within) each data iterator.

- refactor the code and integrate the reweighting algorithm with Libtorch models for the conditional probabilities — describe how this is supported
- 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)
- also need to then support gradient-based algorithms (like vanilla SGD) by implementing Eq. (2.34) for the current basic implementations in the learnadex — shouldn't be too difficult!
- 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 [34], 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. 3.1 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.

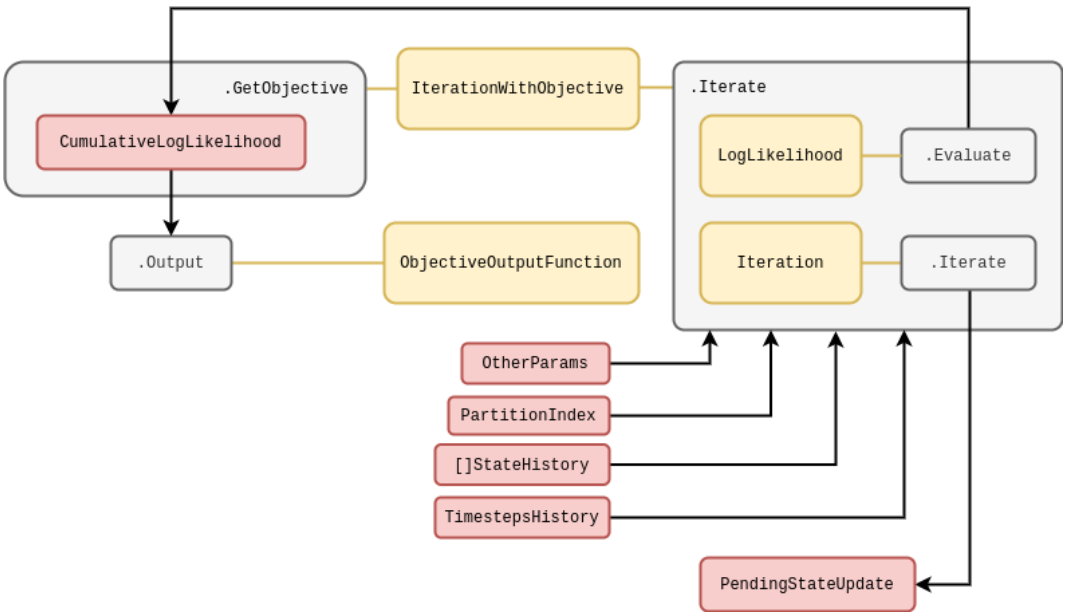

Figure 3.2: A schematic of an iteration with an objective function evaluation.

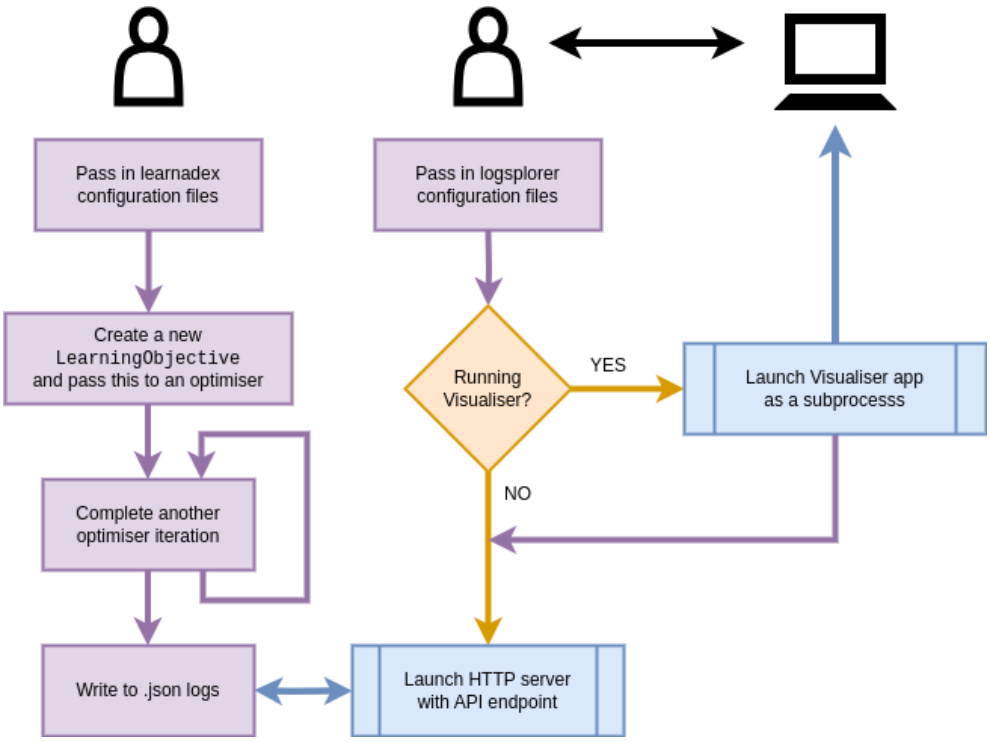

Figure 3.3: A diagram of the main learnadex and logsplorer executables.

4

Online simulation inference

Concept. To generalise the procedure of statistical inference for any simulation 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. 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.

4.1 Inference formalism

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

$$\mathcal{P}_{\mathsf{t}+1}(z|Y) \propto \mathcal{L}_{\mathsf{t}+1}(Y|z)\mathcal{P}(z)\,. \tag{4.1}$$

In the formula above, one relates the prior probability distribution over a parameter set $\mathcal{P}(z)$ and the likelihood $\mathcal{L}_{\mathsf{t}+1}(Y|z)$ of some data matrix Y up to timestep $\mathsf{t}+1$ given the parameters z of a model to the posterior probability distribution of parameters given the data $\mathcal{P}_{\mathsf{t}+1}(z|Y)$ up to some 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 $\mathcal{P}_{\mathsf{t}+1}(z|Y)$ — or its logarithm — in Eq. (4.1) 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 reweighting. 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').

How might we calulate the posterior in practice 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 measurements $\mathsf{Y}_{\mathsf{t}+1}(z), \mathsf{Y}_{\mathsf{t}}(z), \dots$ 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 states of the simulation X, so a more complete picture can be provided by the following probabilistic relation

$$P_{\mathsf{t}+1}(\mathsf{y}|z) = \int_{\omega_{\mathsf{t}+1}} \mathrm{d}^n x \, P_{\mathsf{t}+1}(\mathsf{y}|x) P_{\mathsf{t}+1}(x|z) \,, \tag{4.2}$$

where, in practical terms, the measurement probability $P_{\mathsf{t}+1}(\mathsf{y}|x)$ of $\mathsf{Y}_{\mathsf{t}+1} = \mathsf{y}$ given $X_{\mathsf{t}+1} = x$ 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 next 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.

For an arbitrary simulation model which is defined by the stochadex, the likelihood in Eq. (4.1) is typically not describable as a simple function or distribution. While we could train the probability reweighting we derived in the previous chapter to match the simulation; to do this well would require having an exact formula for the conditional probability, and this is not always easy to derive in the general case. Instead, there is a class of Bayesian inference methods which we shall lean on to help us compute the posterior distribution (and hence the MAP), which are known as 'Likelihood-Free' methods [35, 36, 37, 38].

'Likelihood-Free' methods work by separating out the components of the posterior which relate to the closeness of rows in Y to the rows in Y from the components which relate the states X and parameters z of the simulation stochastically to Y. To achieve this separation, we can make use of chaining conditional probability like this

$$\mathcal{P}_{\mathsf{t}+1}(X, z|Y) = \int_{\Upsilon_{\mathsf{t}+1}} \mathrm{d}\mathsf{Y} \, \mathcal{P}_{\mathsf{t}+1}(\mathsf{Y}|Y) P_{\mathsf{t}+1}(X, z|\mathsf{Y}) \,, \tag{4.3}$$

where $\Upsilon_{\mathsf{t}+1}$ here corresponds to the domain of the simulated measurements matrix Y at time $\mathsf{t}+1$.

As we demonstrated in the previous chapter, it's possible for us to also optimise a probability distribution $\mathcal{P}_{\mathsf{t}'}(\mathsf{y}|Y) = P_{\mathsf{t}'}(\mathsf{y}; \mathcal{M}_{\mathsf{t}'}, \mathcal{C}_{\mathsf{t}'}, \dots)$ for each step in time to match the statistics of the measurements in Y as well as possible, given some statistics $\mathcal{M}_{\mathsf{t}'} = \mathcal{M}_{\mathsf{t}'}(Y)$ and $\mathcal{C}_{\mathsf{t}'} = \mathcal{C}_{\mathsf{t}'}(Y)$. Assuming the independence of samples (rows) in Y, this distribution can be used to construct the distribution over all of Y through the following product

$$\mathcal{P}_{\mathsf{t}+1}(\mathsf{Y}|Y) = \prod_{\mathsf{t}'=0}^{\mathsf{t}+1} P_{\mathsf{t}'}(\mathsf{y}; \mathcal{M}_{\mathsf{t}'}, \mathcal{C}_{\mathsf{t}'}, \dots) \,. \tag{4.4}$$

We do not necessarily need to obtain these statistics from the probability reweighting 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. This choice to summarise the data with statistics means we are using what is known as a Bayesian Synthetic Likelihood (BSL) method [36, 37] instead of another class of methods which approximate an objective function directly using a proximity kernel — known as Approximate Bayesian Computation (ABC) methods [35].

Let's consider a few concrete examples of $P_{\mathsf{t}'}(\mathsf{y}; \mathcal{M}_{\mathsf{t}'}, \mathcal{C}_{\mathsf{t}'}, \dots)$. If the data measurements were well-described by a multivariate normal distribution, then

$$P_{\mathsf{t}'}(\mathsf{y}; \mathcal{M}_{\mathsf{t}'}, \mathcal{C}_{\mathsf{t}'}, \dots) = \mathsf{MultivariateNormalPDF}(\mathsf{y}; \mathcal{M}_{\mathsf{t}'}, \mathcal{C}_{\mathsf{t}'})\,, \tag{4.5}$$

Similarly, if the data measurements were instead better described by a Poisson distribution, we might disregard the need for a covariance matrix statistic $\mathcal{C}_{\mathsf{t}'}$ and instead use

$$P_{\mathsf{t}'}(\mathsf{y}; \mathcal{M}_{\mathsf{t}'}, \mathcal{C}_{\mathsf{t}'}, \dots) = \mathsf{PoissonPMF}(\mathsf{y}; \mathcal{M}_{\mathsf{t}'})\,. \tag{4.6}$$

The more statistically-inclined readers may notice that the probability mass function here would require the integrals in Eq. (4.3) to be replaced with summations over the relevant domains.

Eq. (4.3) 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 reapply Bayes' rule and the chaining of conditional probability to find

$$P_{\mathsf{t}+1}(x, z|\mathsf{Y}) \propto P_{\mathsf{t}+1}(\mathsf{y}|z) P_{\mathsf{t}}(z|\mathsf{Y}') = P_{\mathsf{t}+1}(\mathsf{y}|x) P_{\mathsf{t}+1}(x|z) P_{\mathsf{t}}(z|\mathsf{Y}')\,, \tag{4.7}$$

where here $P_{\mathsf{t}}(z|\mathsf{Y}')$ is the probability of $\mathsf{Y}_{\mathsf{t}} = \mathsf{Y}'$.

The relationship between $P_{\mathsf{t}+1}(X|z)$ and previous timesteps can be directly inferred from the probabilistic iteration formula that we introduced in the previous chapter. So we can map probabilities of $X_{0:\mathsf{t}+1} = X$ throughout time and 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 there is! The marginalisation

$$P_{\mathsf{t}+1}(z|\mathsf{Y}) \propto \left[\int_{\Omega_{\mathsf{t}+1}} \mathrm{d}^n x\, P_{\mathsf{t}+1}(\mathsf{y}|x) P_{\mathsf{t}+1}(x|z) \right] P_{\mathsf{t}}(z|\mathsf{Y}')\,, \tag{4.8}$$

shows how the z updates can occur in an iterative fashion. The reader may also recognize the factor above in brackets as Eq. (4.2). To complete the picture, one can combine the X and z updates into a joint distribution update which takes the following form

$$P_{\mathsf{t}+1}(X, z|\mathsf{Y}) \propto P_{\mathsf{t}+1}(\mathsf{y}|x) P_{(\mathsf{t}+1)\mathsf{t}}(x|X', z) P_{\mathsf{t}}(X', z|\mathsf{Y}')\,. \tag{4.9}$$

We can also marginalise this distribution over the past state history rows to get a distribution over the latest state row $X_{\mathsf{t}+1} = x$ like this

$$P_{\mathsf{t}+1}(x, z|\mathsf{Y}) = \int_{\Omega_{\mathsf{t}}} \mathrm{d}X' P_{\mathsf{t}+1}(X, z|\mathsf{Y}) \propto P_{\mathsf{t}+1}(\mathsf{y}|x) \int_{\Omega_{\mathsf{t}}} \mathrm{d}X' P_{(\mathsf{t}+1)\mathsf{t}}(x|X', z) P_{\mathsf{t}}(X', z|\mathsf{Y}')\,. \tag{4.10}$$

In the next section, we're going to discuss how to translate all of this probabilistic language into some MAP inference algorithms. Before we do this, however, it will be instructive (particularly for

'online' learning algorithms) to consider what happens if the model changes over time and z needs to change in order to better represent the real data. In such situations, we propose to apply the same formula as Eq. (4.10) but instead replace the distribution over (X', z) on the right hand side with its 'past discounted' version[1]

$$\int_{\Omega_{\mathsf{t}}} \mathrm{d}X' P_{\mathsf{t}}(X', z|\mathsf{Y}') \;\longrightarrow\; \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}^n x' \beta^{\mathsf{t}-\mathsf{t}'} P_{\mathsf{t}'}(x', z|\mathsf{Y}') \,, \tag{4.11}$$

where $0 < \beta < 1$ and we recall the notation which considers distributions over the individual rows x' within the matrix X' in this new version. This time-dependent discount factor could be used to reduce the dependence of the update on data which is much further in the past, and hence will ultimately lead to a more responsive algorithm. This responsiveness would have to be balanced with the tradeoffs associated with discounting potentially valuable data that may offer greater long-term stability. Readers who are familiar with reinforcement learning may be starting to feel in familiar territory here — they will have to wait for the latter parts of the book to see more on discounting though!

4.2 Online learning the MAP

Eq. (4.9) tells us how to probabilistically translate the current state of knowledge about (x, z) forward through time in response to the arrival of new data. We also know how to connect the simulated measurements to the real data because Eq. (4.3) essentially gives us an objective function to maximise for each step in time. This is all great in theory; 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 of measurements $\mathsf{Y}_{\mathsf{t}+1}(z), \mathsf{Y}_{\mathsf{t}}(z), \ldots$, 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. Lastly, given that the simulation model in the stochadex needs to be running multiple times for each timestep, we need a way of mitigating computational expense.

So how should we proceed? To solve this problem in the general case, Eqs. (4.3) and (4.9) tell us we need to synthesize the following components into a single algorithm:

1. A process $P_{(\mathsf{t}+1)\mathsf{t}}(x|X', z)$ which iterates the state matrix of the simulation X forward in time.
2. A process $P_{\mathsf{t}+1}(\mathsf{y}|x)$ which generates a simulated measurement from the simulated state x.
3. A probability distribution $P_{\mathsf{t}'}(\mathsf{y}; \mathcal{M}_{\mathsf{t}'}, \mathcal{C}_{\mathsf{t}'}, \ldots)$ which represents the posterior distribution of the simulated measurement vector y given an optimised compression of the real data into summary statistics.
4. Some way of representing samples from the distribution $P_{\mathsf{t}+1}(X, z|\mathsf{Y})$ so that their distribution can be updated and will coverge towards the posterior over (X, z).

[1] In the continuous-time version, this past-discounting factor can depend on the stepsize such that we replace

$$\beta^{\mathsf{t}-\mathsf{t}'} \longrightarrow \frac{1}{\beta[\delta t(\mathsf{t})]} \prod_{\mathsf{t}''=\mathsf{t}'}^{\mathsf{t}} \beta[\delta t(\mathsf{t}'')] \,.$$

Cite this nice paper which outlines all the recent kinds of simulation inference: [39].

Keep the heuristic Bayes posterior estimator method as it is an example of recursive Bayes estimation - it can also be used to filter the ensemble at every step to make a particle filter [40]. Amortize this online learning process by training a neural net to produce the best estimates for the filter from the input real data!

Before writing this up, should read this paper on efficient amortized inference using neural networks with `BayesFlow` here in particular: [41]. But also, should cite other works to make amortized inference more efficient by using neural networks to learn convenient functions of the Bayes factor in Evidence networks [42].

- amortized online inference of the posterior update over just z can be achieved by running lots of simulations and solving the inverse problem with the `y` outputs i.e., neural network modelling of the update in Eq. (4.8)

The algorithm is specifically: 1. if this is a refit step, sample new values for (X, z) for all members of the ensemble from the current (X, z) distribution points and run the iterations for all of these ensemble members from the back of the window all the way up to the current point in time (hence the full matrix X is sampled) 2. take all of the ensemble members a step forward in time 3. approximate the mode by computing the average values of z within the q-th percentile of the sampled probability mass (where q is set by the user and is ideally $< 68\%$) — this idea comes from nested sampling 4. stream in the data for the next point in time and go to 1.

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 and optimizing the current posterior with respect to (X, z) to compute the MAP.

Readers of the previous section may also have recognized that Eq. (4.9) contains the same conditional probability $P_{(\mathrm{t}+1)\mathrm{t}}(x|X', z)$ as the reweighting algorithm. This structure enables us to reuse all of the exposition we provided for the probabilistic reweighting and highlights how the reweighting 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 [43, 44, 23] naturally emerges.

4.3 Software design

Take a step back at this point and consider all the use-cases for the learnadex:

- Core functionality is to enable iterative updates to the $P_{\mathrm{t}}(X, z)$ distribution at every timestep. There must also be flexibility in how this distribution can represented, i.e., either:
  - a set of Monte Carlo samples, or
  - a set of distribution parameters
- For Monte Carlo samples, we must also keep the flexibility to use either a BSL or ABC-style data-to-sim comparison in order to facilitate the update

- For distribution parameters, the update can be custom-built by the user with online likelihood-based inference/Bayes estimator methods
- Separate thread context runs of online learning methods using an update method, e.g.,
  - Gradient-free batch optimisation with any chosen learning algorithm
  - Gradient-based parameter updates
  - Arbitrary parameter updates from a different method

5

Optimising system interactions

Concept. To design and build software which enables the optimisation of automated control objectives over stochastic phenomena of any kind. The theory in this chapter will overlap significantly with that of Reinforcement Learning (RL), however, in contrast to more standard RL approaches, we shall be relying on all of the work from previous parts of this book to help agents characterise, measure and learn from their environment. The software which implements our generalised control optimisation algorithm will be implemented as an extension to the learnadex. For the mathematically-inclined, this chapter will cover how we formalise model-based automated control optimisation within the frameworks that we have already introduced in this book. For the programmers, the public Git repository for the code described in this chapter can be found here: https://github.com/worldsoop/worldsoop.

5.1 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 $X_{0:\mathsf{t}} \to X_{0:\mathsf{t}+1}$ to include a new layer of possible interactions which is facilitated by a new vector-valued 'take action' function G_{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 include interactions by composition with the original state update function like so

$$X^i_{\mathsf{t}+1} = G^i_{\mathsf{t}+1}[F_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, \mathsf{t}), A_{\mathsf{t}+1}] = \mathcal{F}^i_{\mathsf{t}+1}(X_{0:\mathsf{t}}, z, A_{\mathsf{t}+1}, \mathsf{t})\,, \tag{5.1}$$

where we have also introduced the concept of the 'actions' performed $A_{\mathsf{t}+1}$ on the system; some vector of parameters which define what actions are taken at timestep $\mathsf{t}+1$. The code for the new iteration formula would look something like Fig. 5.1.

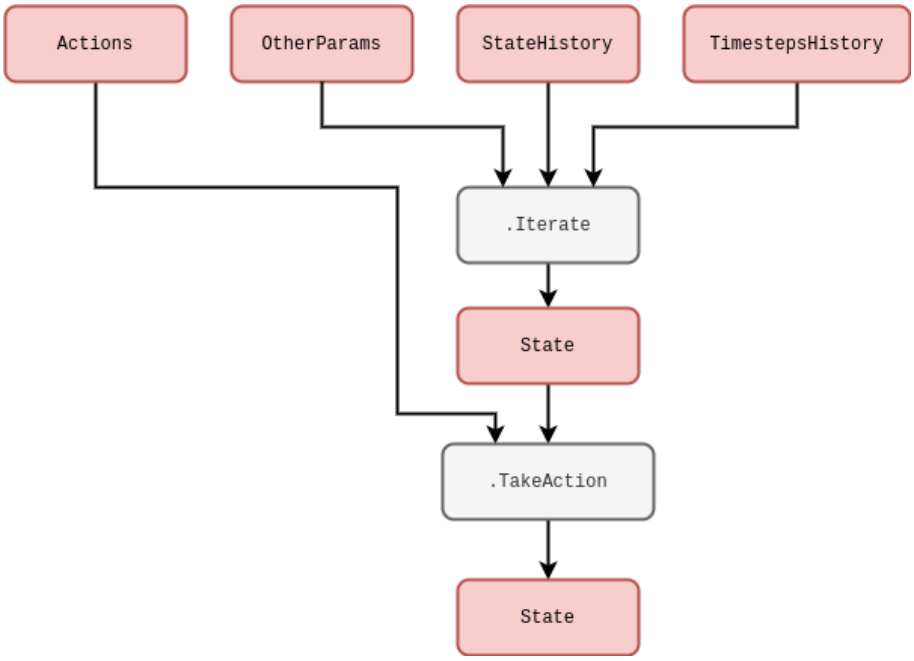

Figure 5.1: Code schematic of Eq. (5.1).

So far, Eq. (5.1) on its own will allow the agent to take actions that are scheduled up front through some fixed process or perhaps through user interaction via a game interface. So what's next? In order to start creating algorithms which will act on the system state for us, we need to develop a formalism which 'closes the loop' by feeding information back from the stochastic process to the agent's decision-making algorithm.

If we use $A_{0:\mathsf{t}+1}$ a referring to the matrix of historically-taken actions which up to time $\mathsf{t}+1$, we can build up a more generalised, non-Markovian picture of automated interactions with the system which matches the notation we are already using for $X_{0:\mathsf{t}+1}$. Let us now define a Non-Markovian Decision Process (NMDP) as a probabilistic model which draws an actions matrix $A_{0:\mathsf{t}+1} = A$ from a 'policy' distribution $\Pi_{(\mathsf{t}+1)\mathsf{t}}(A|X,\theta)$ given $X_{0:\mathsf{t}} = X$ and a new vector of parameters which fully specify the automated interations. Using the probabilistic notation from the previous part of the book, the joint probability that $X_{0:\mathsf{t}+1} = X$ and $A_{0:\mathsf{t}+1} = A$ at time $\mathsf{t}+1$ is

$$P_{\mathsf{t}+1}(X, A|z, \theta) = P_{\mathsf{t}}(X'|z, \theta)\, \Pi_{(\mathsf{t}+1)\mathsf{t}}(A|X', \theta) P_{(\mathsf{t}+1)\mathsf{t}}(x|X', z, A)\,, \tag{5.2}$$

where we recall that $P_{(\mathsf{t}+1)\mathsf{t}}(x|X', z, A)$ is the conditional probability of $X_{\mathsf{t}+1} = x$ given $X_{0:\mathsf{t}} = X'$ and z that we have encountered before, but it now requires $A_{0:\mathsf{t}+1} = A$ as another given input. We have illustrated Eq. (5.1) and how it relates to the policy distribution of Eq. (5.2) with a new graph representation in Fig. 5.2.

For additional clarity, let's take a moment to think about what $\Pi_{(\mathsf{t}+1)\mathsf{t}}(A|X,\theta)$ represents and how generally descriptive it can be. If an agent is acting under and entirely deterministic policy, then the policy distribution may be simplified to a direct function mapping which is parameterised by θ. At the other extreme, the distribution may also describe a fully stochastic policy where actions

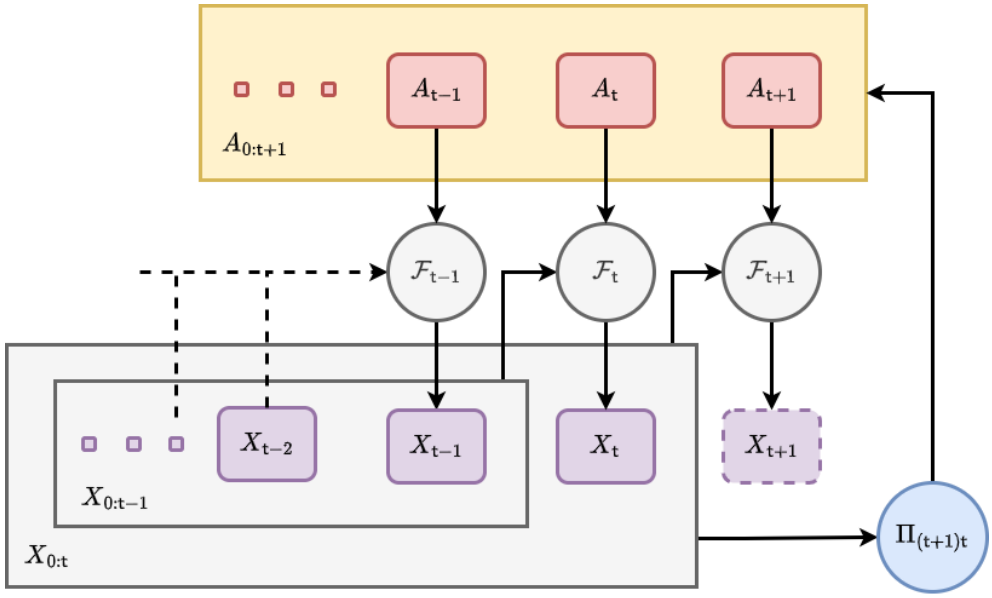

Figure 5.2: Graph representation of Eq. (5.1) with the policy distribution of Eq. (5.2).

are drawn randomly in time. If we combine this consideration of noise with the observation that policies described by a distribution $\Pi_{(t+1)t}(A|X,\theta)$ permit a memory of past actions and states, it's easy to see that this structure can be used in a wide variety of different use cases.

By marginalising over Eq. (5.2) we find an updated probabilistic iteration formula for the stochastic process state which now takes the influence of agent actions into account

$$P_{t+1}(X|z,\theta) = \int_{\Xi_{t+1}} \mathrm{d}A\, P_t(X'|z,\theta)\, \Pi_{(t+1)t}(A|X',\theta) P_{(t+1)t}(x|X',z,A)\,. \tag{5.3}$$

This relationship will be very useful in the last part of this book when we begin to look at optimising control algorithms.

What are the main categories of action which are possible in the rows of A? Since the NMDP described by $\Pi_{(t+1)t}(A|X',\theta)$ is just another form of stochastic process, the main categories of action will fall into the same as those we covered in defining the stochadex formalism. The first, and perhaps most obvious, category would probably where the actions are defined in a continuous space and are continuously applied on every timestep. Some examples of these 'continuously-acting' decision processes include controlling the temperature of chemical reactions [45] (such as those in a brewery), spacecraft control [46] and guidance systems, as well as the driving of autonomous vehicles [47]. Within a kind of subset of the continuously-acting category; we can also find the event-based acting decision processes (where actions are not necessarily taken every timestep), e.g. controlling traffic through signal timings [48], managing disease spread through treatment intervals [49] and automated trading on stock markets [50].

Many of the examples we have given above have continuous action spaces, but we might also consider classes of decision processes where actions are defined discretely. Examples of these include the famous multi-armed bandit problem [51] (like choosing between website layouts for E-

commerce [52]), managing a sports team through player substitutions, sensor measurement scheduling [53] and the sequential design prioritisation of large-scale scientific experiments [54].

5.2 States, actions and attributing rewards

In the previous parts of this book we laid out the concept for a generalised framework to simulate and learn stochastic phenomena continually as data is received. Given that we have also introduced a framework for the automated control of these phenomena, we have all the ingredients we need to create optimal decision-making algorithms. The key question to answer then, is: *optimal with respect to what objective?*

The objective of an automated control algorithm could take many forms depending on the specific context. Since there is no loss in generality in doing so, it seems natural to follow the naming convention used by Markov Decision Processes (MDP) [55, 1] by referring to the objective outcome of an action at a particular point in time as having a 'reward' value r. Since the relationship between reward, actions and states may be stochastic, it makes sense to relate the reward outcome r given a state history X and action history A at timestep $\mathsf{t}+1$ through the probability distribution $P_{\mathsf{t}+1}(r|X,A)$. Hence, generally, this reward signal is non-Markovian — as is the case in many real-world problems [56].

We can use the reward probability distribution to derive a joint distribution over both state history X' and reward r at timestep $\mathsf{t}+1$ like so

$$P_{(\mathsf{t}+1)\mathsf{t}}(r, x'|X, z, \theta) = P_{\mathsf{t}+1}(r|X', A)\Pi_{(\mathsf{t}+1)\mathsf{t}}(A|X, \theta)P_{(\mathsf{t}+1)\mathsf{t}}(x'|X, z, A)\,. \tag{5.4}$$

In this expression, let's recall that we are using the policy distribution $\Pi_{(\mathsf{t}+1)\mathsf{t}}(A|X,\theta)$ for agent interactions and the fundamental state update conditional probability for the underlying stochastic process $P_{(\mathsf{t}+1)\mathsf{t}}(x'|X,z,A)$.

Note that in most use cases, the state of real-world phenomena cannot be measured perfectly. So to enable any agent trained on simulated phenomena to potentially act in the real world, we will need to include a measurement process as part of the information retrieval step. This is the part where we can leverage our work in a previous chapter which develops an online learning system for stochastic process models. But we're jumping ahead with this thinking and will return to this point later on.

Using Eq. (5.4), we can now define a 'state value function' V_{t} at timestep t which is the expected γ-discounted future reward given the current state history X and the other parameters like this[1]

$$\begin{aligned} V_{\mathsf{t}}(X, z, \theta) &= \mathrm{E}_{\mathsf{t}}(\mathsf{Discounted\ Return}|X, z, \theta) \\ &= \sum_{\mathsf{t}'=\mathsf{t}}^{\infty} \int_{\omega_{\mathsf{t}'+1}} \mathrm{d}^n x' \int_{\rho_{\mathsf{t}'+1}} \mathrm{d}r\, r\, \gamma^{\mathsf{t}'-\mathsf{t}} \prod_{\mathsf{t}''=\mathsf{t}}^{\mathsf{t}'} P_{(\mathsf{t}''+1)\mathsf{t}''}(r, x'|X, z, \theta)\,, \end{aligned} \tag{5.5}$$

[1] The discount factor in continuous time could also be explicitly dependent on the stepsize such that we would replace the discount factor in Eq. (5.5) with

$$\gamma^{\mathsf{t}'-\mathsf{t}} \longrightarrow \frac{1}{\gamma[\delta t(\mathsf{t}+1)]} \prod_{\mathsf{t}''=\mathsf{t}}^{\mathsf{t}'} \gamma[\delta t(\mathsf{t}''+1)]\,.$$

where $0 < \gamma < 1$. The idea behind this discount factor γ is to decrease the contribution of rewards to the optimisation objective (often called the 'expected discounted return' in RL) more and more as the prediction increases into the future. Note also that the state value function is inherently recursively defined, such that

$$V_{\mathtt{t}}(X, z, \theta) = \int_{\omega_{\mathtt{t}+1}} \mathrm{d}^n x \int_{\rho_{\mathtt{t}+1}} \mathrm{d}r \, P_{(\mathtt{t}+1)\mathtt{t}}(r, x'|X, z, \theta) \big[r + \gamma V_{\mathtt{t}+1}(X', z, \theta) \big] \,, \tag{5.6}$$

and the optimal θ can hence be derived from

$$\theta^*_{\mathtt{t}}(X, z) = \underset{\theta}{\mathrm{argmax}} \big[V_{\mathtt{t}}(X, z, \theta) \big] \,. \tag{5.7}$$

By deriving the optimal policy in terms of the parameters $\theta^*_{\mathtt{t}}(X, z)$, the optimal state value function and policy distribution can therefore be derived from

$$V^*_{\mathtt{t}}(X, z) = V_{\mathtt{t}}[X, z, \theta^*_{\mathtt{t}}(X, z)] \tag{5.8}$$

$$\Pi^*_{(\mathtt{t}+1)\mathtt{t}}(A|X, z) = \Pi_{(\mathtt{t}+1)\mathtt{t}}[A|X, \theta^*_{\mathtt{t}}(X, z)] \,. \tag{5.9}$$

Note that the type of decision process optimisation which we have introduced above differs from standard RL methodology. In the more conventional 'model-free' RL approaches, the state-action value function

$$Q_{\mathtt{t}}(X, A, z) = \mathrm{E}_{\mathtt{t}}(\mathsf{Discounted\ Return}|X, A, z) \,, \tag{5.10}$$

would be used to evaluate the optimal policy instead of the state value function $V_{\mathtt{t}}(X, z, \theta)$ that we are using above. We are able to use the latter here because the simulation model gives us explicit knowledge of the $P_{(\mathtt{t}+1)\mathtt{t}}(x'|X, z, A)$ distribution which is utilised by Eq. (5.4). When this model is not known, the state-action value function $Q_{\mathtt{t}}(X, A, z)$ must be learned explicitly through sample estimation from the measured state and experienced outcomes of actions taken by the agent.

When an agent takes an action to measure the state of the system (or when it is given measurements without needing to take action) there will typically be some uncertainty in how the history of measured real-world data Y maps to the latent states of the system X and its parameters z at time $\mathtt{t}+1$. It is natural, then, to represent this uncertainty with a posterior probability distribution $\mathcal{P}_{\mathtt{t}+1}(X, z|Y)$ as we did in the previous chapters of this book.

5.3 Algorithm designs

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

- Introduce broad concept of dynamic programming — partitioning a optimal global control into smaller optimal control segments/iterations.
- Talk about the utility of the model-based online learning approach in the case of partially observed systems [57].
- Look into the overlaps with this approach and Thompson sampling for exploration — discuss here.

- Looking at a stochastic policy iteration algorithm here combined with Monte Carlo rollouts.
- 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 discounted utility in a Bayesian design framework. Draw parallels.

5.4 Software design

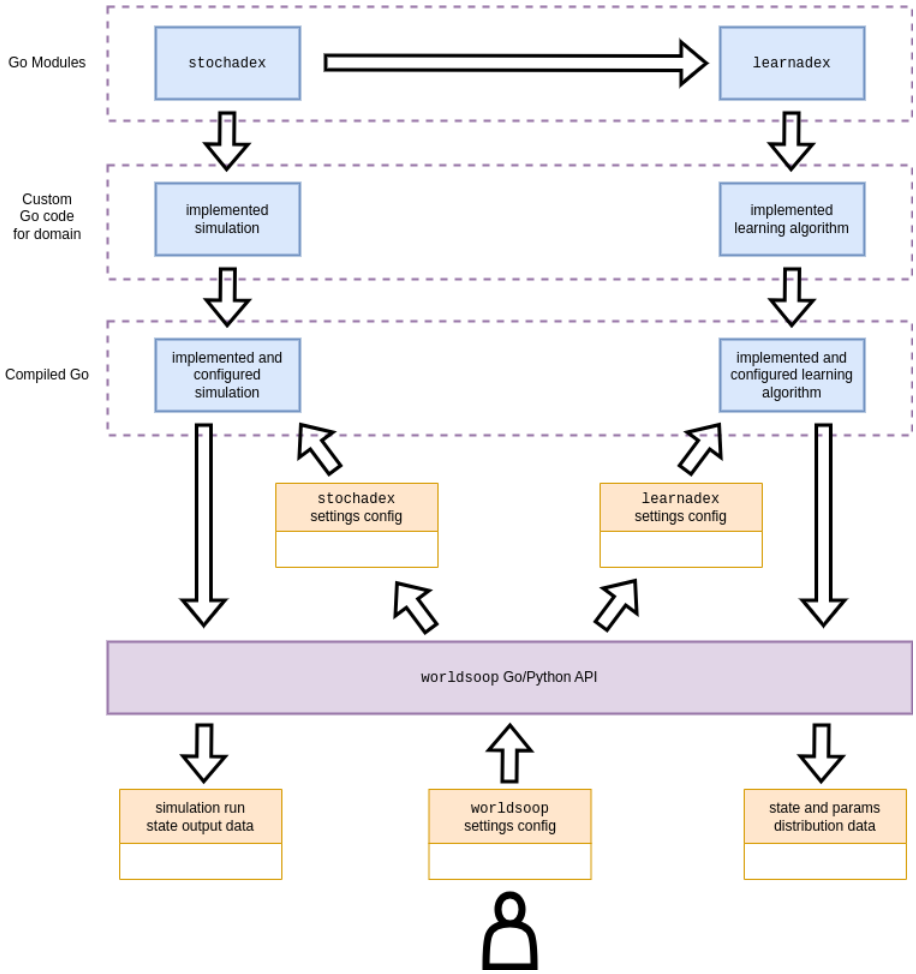

Figure 5.3: Diagram illustrating the high-level layer inter-dependencies of the Go/Python API.

Part 2

6

Simple state transitions

Concept. To define and develop an archetype simulation environment for simple state transitions. In our classification scheme, this archetype is defined by a trivial state partition graph topology and would make sense for simulations of sequential design problems, sports matches and other simple gameplay domains. We will also discuss the typical ways in which the state of the system may only partially be observed in realistic examples, and analyse how best to deal with each situation. For the mathematically-inclined, this chapter will define the mapping of our formalism to simple state transitions. For the programmers, the software which is designed and described in this chapter can be found in the public Git respository here: https://github.com/worldsoop/worldsoop.

6.1 Defining the archetype

The simple state transition archetype refers to simulation environments where there is no obvious computational benefit to partitioning the state into concurrently updating or acting on separate components. There may even be performance benefits from keeping state information all within the same common data structure in memory, but this can depend on the specific problem of study. In the interest of completeness with respect to the chapters which follow on from this one, we have illustrated the trivial state partition graph topology for this archetype in Fig. 6.1.

Figure 6.1: State partition graph topology for simple state transition archetypes.

In order to understand what sorts of data might be collected about this archetype in realistic

scenarios, let's begin by considering the types of real-world problem domain which could potentially leverage simulation environments to train control algorithms. The subset of these which may best suit the simple state transition environment archetype are:

- Event-based simulations of sports matches, e.g., football [58], rugby [59], tennis [60], etc., and other forms of game — all of which typically define a relatively simple global match/gameplay context as their 'state'.
- Sequential design simulations to change the configuration or data collection strategies of, e.g., astronomical telescopes [61, 62], biological experiments [63] and user interfaces [64], which typically define a relatively simple and finite set of possible actions that can be taken.

7

Dynamic spatial fields

Concept. To define and develop an archetype simulation environment for dynamic spatial fields. In our classification scheme, this archetype is defined by a highly-structured, bidirectional state partition graph topology and would make sense for simulations of spatial epidemiological processes, ecosystems and weather. We will also discuss the typical ways in which the spatial state partitions of the system may only partially be observed in realistic examples, and analyse how best to deal with each situation. For the mathematically-inclined, this chapter will define the mapping of our formalism to dynamic spatial fields. For the programmers, the software which is designed and described in this chapter can be found in the public Git respository here: https://github.com/worldsoop/worldsoop.

7.1 Adapting the probabilistic formalism

Let's by returning to the probabilistic formalism that we introduced earlier and noting that the covariance matrix estimate with elements $C^{ij}_{\mathtt{t}+1}(z)$ 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. 7.2 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.

- Full sim: fully individual-based spatial stochastic model

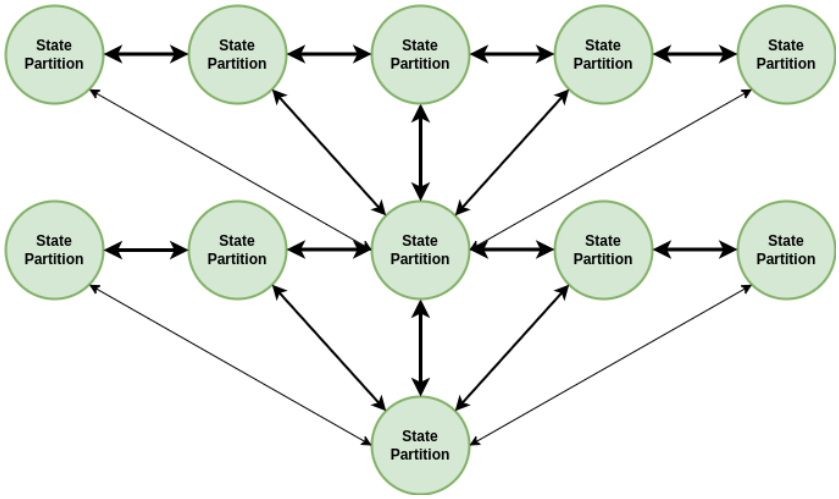

Figure 7.1: State partition graph topology for dynamic spatial field archetypes.

- Inference model: spatial mean field inference features using the probabilistic reweighting + exponentially-weighted nonlinear features
- Also use the likelihood-free inference model and amortized sim inference model.

- At some point it might be sensible to move into the Fourier domain here — at least for derivations and calculations. Probably more intuitive for the reader to keep it mostly in real space though if possible.
- 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.
- Also look into the Radial Basis Function (RBF) and higher-order derived kernels based on DALI expansion [65] in order to try and capture non-Gaussianity.

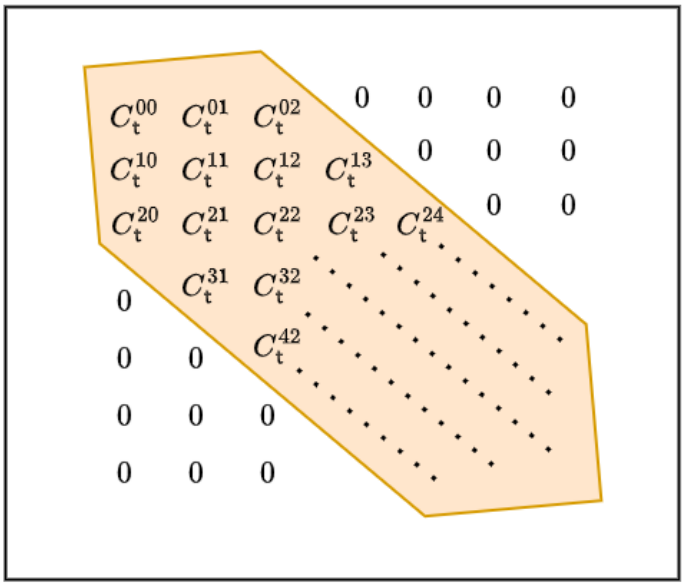

Figure 7.2: An illustration of a banded covariance matrix with a bandwidth of 2.

8

Distributed state networks

Concept. To define and develop an archetype simulation environment for distributed state networks. In our classification scheme, this archetype is defined by an arbitrary, bidirectional state partition graph topology and would make sense for simulations of human brain or traffic control networks and power or water grids. We will also discuss the typical ways in which the state of the system (and its various partitions) may only partially be observed in realistic examples, and analyse how best to deal with each situation. For the mathematically-inclined, this chapter will define the mapping of our formalism to distributed state networks. For the programmers, the software which is designed and described in this chapter can be found in the public Git respository here: https://github.com/worldsoop/worldsoop.

8.1 A large-scale Lotka-Volterra model

- Full sim: full population-based spatial stochastic model
- Inference model: mean field network inference features using the probabilistic reweighting using exponential weighting and cross-correlations
- Also use the likelihood-free inference model and amortized full sim inference method.

Inspired by the empirical dynamical modeling approach to sockeye salmon in Ref. [66], 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

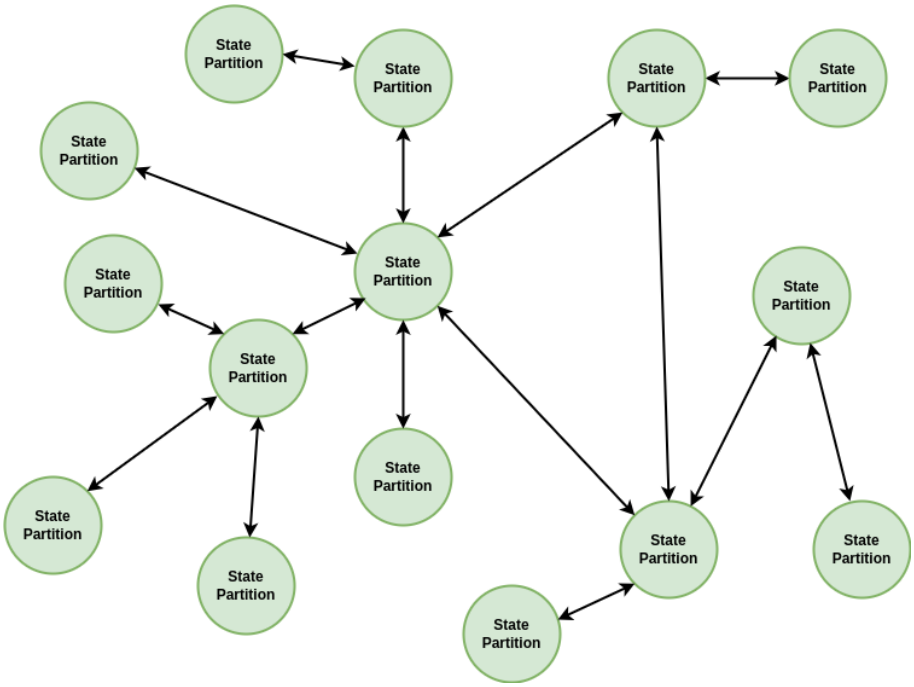

Figure 8.1: State partition graph topology for distributed state network archetypes.

$$\frac{\mathrm{d}}{\mathrm{d}t}P(\ldots,n_i,\ldots,t) = \sum_{\forall i}\mathcal{T}_i^{+}(\ldots,n_i-1,\ldots,\mathsf{f},t)P(\ldots,n_i-1,\ldots,t) \tag{8.1}$$

$$+\sum_{\forall i}\mathcal{T}_i^{-}(\ldots,n_i+1,\ldots,\mathsf{f},t)P(\ldots,n_i+1,\ldots,t) \tag{8.2}$$

$$-\sum_{\forall i}\left[\mathcal{T}_i^{+}(\ldots,n_i,\ldots,\mathsf{f},t)+\mathcal{T}_i^{-}(\ldots,n_i,\ldots,\mathsf{f},t)\right]P(\ldots,n_i,\ldots,t)\,, \tag{8.3}$$

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

$$\mathcal{T}_i^{+}(\ldots,n_i,\ldots,\mathsf{f},t) = \mathcal{T}_i^{+}(\ldots,n_i,\ldots) = \Lambda_i(n_i) + n_i\alpha_i\sum_{\forall i' \text{ prey}} n_{i'} \tag{8.4}$$

$$\mathcal{T}_i^{-}(\ldots,n_i,\ldots,\mathsf{f},t) = \mathcal{T}_i^{-}(\ldots,n_i,\ldots) = n_i\mu_i + n_i\gamma_i + n_i\beta_i\sum_{\forall i' \text{ pred}} n_{i'}\,, \tag{8.5}$$

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. [67]...

$$\mathsf{Likelihood} = \sum_{\mathsf{data}} \mathrm{NB}\big[\mathsf{data}; w_{i,\mathsf{survey}} \langle n_i(t_{\mathsf{data}})\rangle, k_{i,\mathsf{survey}}\big] \,, \tag{8.6}$$

9

Multi-stage pipelines

Concept. To define and develop an archetype simulation environment for multi-stage pipelines. In our classification scheme, this archetype is defined by an arbitrary, directional state partition graph topology and would make sense for simulations of logistics problems or data processing pipelines. We will also discuss the typical ways in which the stages within the pipeline may only partially be observed in realistic examples, and analyse how best to deal with each situation. For the mathematically-inclined, this chapter will define the mapping of our formalism to multi-stage pipelines. For the programmers, the software which is designed and described in this chapter can be found in the public Git respository here: https://github.com/worldsoop/worldsoop.

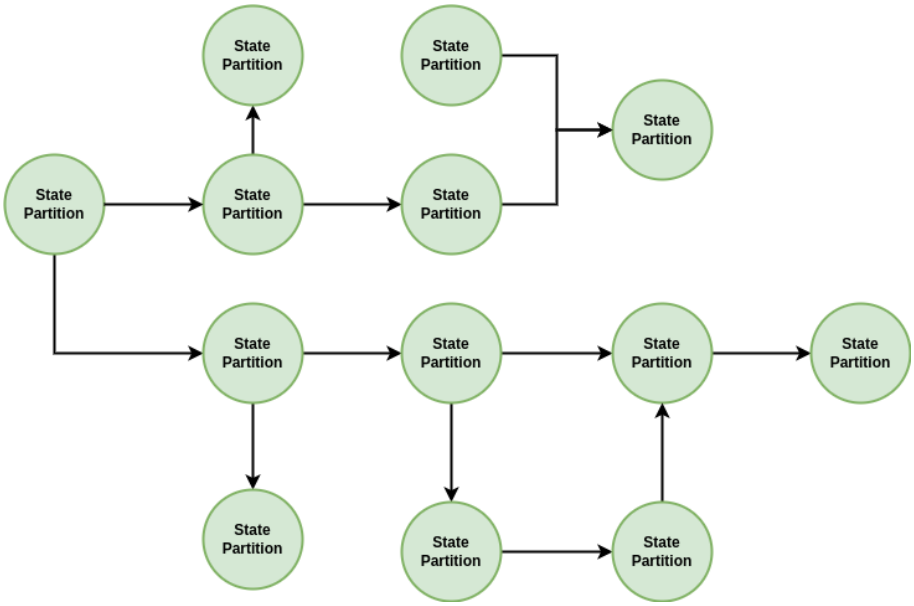

Figure 9.1: State partition graph topology for multi-stage pipeline archetypes.

10

Centralised exchanges

Concept. To define and develop an archetype simulation environment for centralised exchanges. In our classification scheme, this archetype is defined by an bidirectional star state partition graph topology and would make sense for simulations of financial, betting and housing markets as well as other forms of resource exchange. We will also discuss the typical ways in which the state partitions of the system may only partially be observed in realistic examples, and analyse how best to deal with each situation. For the mathematically-inclined, this chapter will define the mapping of our formalism to centralised exchanges. For the programmers, the software which is designed and described in this chapter can be found in the public Git respository here: https://github.com/worldsoop/worldsoop.

- Full sim: full event-based spatial stochastic model
- Inference model: spatial mean field inference features using the probabilistic reweighting
- Also use the likelihood-free inference model?

- Humanitarian aid logistics in response to flooding, fire or other natural disasters
- Routing of transportation
- Where to focus searches
- Transportation size distribution
- Supply chain logistics of resources and allocation of budget
- Example paper here with stochastic network models [68]

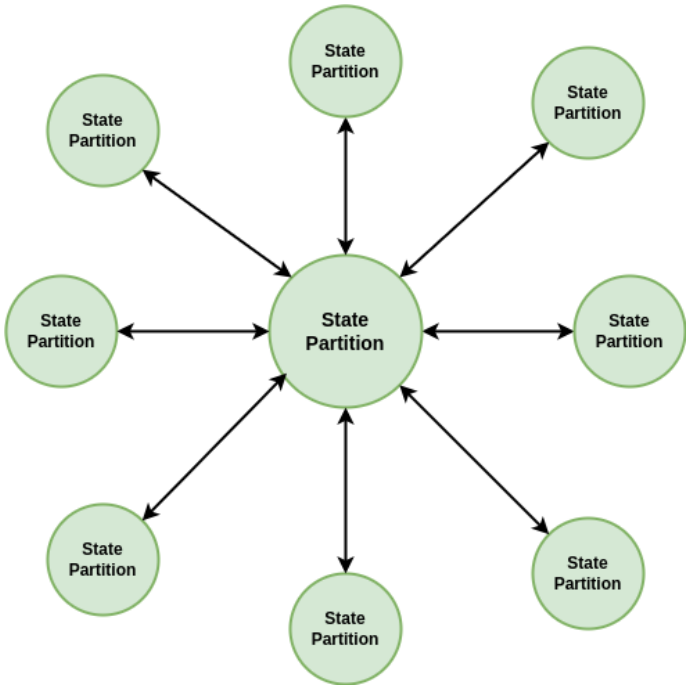

Figure 10.1: State partition graph topology for centralised exchange archetypes.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] "The Go Programming Language," https://go.dev/, accessed: 2023-02-10.

[3] "The C++ Programming Language," https://isocpp.org/, accessed: 2023-10-27.

[4] "The Python Programming Language," https://www.python.org/, accessed: 2023-02-10.

[5] "The TypeScript Programming Language," https://www.typescriptlang.org/, accessed: 2023-07-28.

[6] "Open Source Initiative: MIT License," https://opensource.org/licenses/MIT, accessed: 2023-02-10.

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

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

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

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

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

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

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

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

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

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

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

[18] D. Boyer, M. R. Evans, and S. N. Majumdar, "Long time scaling behaviour for diffusion with resetting and memory," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2017, no. 2, p. 023208, 2017.

[19] H. A. Kramers, "Brownian motion in a field of force and the diffusion model of chemical reactions," *Physica*, vol. 7, no. 4, pp. 284–304, 1940.

[20] J. Moyal, "Stochastic processes and statistical physics," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 11, no. 2, pp. 150–210, 1949.

[21] G. Sugihara and R. M. May, "Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series," *Nature*, vol. 344, no. 6268, pp. 734–741, 1990.

[22] A. Savitzky and M. J. Golay, "Smoothing and differentiation of data by simplified least squares procedures." *Analytical chemistry*, vol. 36, no. 8, pp. 1627–1639, 1964.

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

[24] I. Kobyzev, S. J. Prince, and M. A. Brubaker, "Normalizing flows: An introduction and review of current methods," *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, no. 11, pp. 3964–3979, 2020.

[25] L. Pinheiro Cinelli, M. Araújo Marins, E. A. Barros da Silva, and S. Lima Netto, "Variational autoencoder," in *Variational Methods for Machine Learning with Applications to Deep Networks*. Springer, 2021, pp. 111–149.

[26] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

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

[28] 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)*. IEEE, 1998, pp. 69–73.

[29] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

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

[31] E. Hazan *et al.*, "Introduction to online convex optimization," *Foundations and Trends® in Optimization*, vol. 2, no. 3-4, pp. 157–325, 2016.

[32] "River: Online machine learning in Python," https://riverml.xyz/latest/, accessed: 2023-12-30.

[33] "Vowpal Wabbit: Your go-to interactive machine learning library," https://vowpalwabbit.org/, accessed: 2023-12-30.

[34] "The MongoDB Webpage," https://www.mongodb.com/, accessed: 2023-08-17.

[35] S. A. Sisson, Y. Fan, and M. Beaumont, *Handbook of approximate Bayesian computation*. CRC Press, 2018.

[36] L. F. Price, C. C. Drovandi, A. Lee, and D. J. Nott, "Bayesian synthetic likelihood," *Journal of Computational and Graphical Statistics*, vol. 27, no. 1, pp. 1–11, 2018.

[37] S. N. Wood, "Statistical inference for noisy nonlinear ecological dynamic systems," *Nature*, vol. 466, no. 7310, pp. 1102–1104, 2010.

[38] C. Drovandi and D. T. Frazier, "A comparison of likelihood-free methods with and without summary statistics," *Statistics and Computing*, vol. 32, no. 3, p. 42, 2022.

[39] K. Cranmer, J. Brehmer, and G. Louppe, "The frontier of simulation-based inference," *Proceedings of the National Academy of Sciences*, vol. 117, no. 48, pp. 30 055–30 062, 2020.

[40] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *IEEE Transactions on signal processing*, vol. 50, no. 2, pp. 174–188, 2002.

[41] S. T. Radev, U. K. Mertens, A. Voss, L. Ardizzone, and U. Köthe, "Bayesflow: Learning complex stochastic models with invertible neural networks," *IEEE transactions on neural networks and learning systems*, vol. 33, no. 4, pp. 1452–1466, 2020.

[42] N. Jeffrey and B. D. Wandelt, "Evidence networks: simple losses for fast, amortized, neural bayesian model comparison," *arXiv preprint arXiv:2305.11241*, 2023.

[43] H. O. Hartley, "Maximum likelihood estimation from incomplete data," *Biometrics*, vol. 14, no. 2, pp. 174–194, 1958.

[44] 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)*, vol. 39, no. 1, pp. 1–22, 1977.

[45] C. Beeler, S. G. Subramanian, K. Sprague, N. Chatti, C. Bellinger, M. Shahen, N. Paquin, M. Baula, A. Dawit, Z. Yang *et al.*, "Chemgymrl: An interactive framework for reinforcement learning for digital chemistry," *arXiv preprint arXiv:2305.14177*, 2023.

[46] M. Tipaldi, R. Iervolino, and P. R. Massenio, "Reinforcement learning in spacecraft control applications: Advances, prospects, and challenges," *Annual Reviews in Control*, 2022.

[47] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 4909–4926, 2021.

[48] D. Garg, M. Chli, and G. Vogiatzis, "Deep reinforcement learning for autonomous traffic light control," in *2018 3rd IEEE international conference on intelligent transportation engineering (ICITE)*. IEEE, 2018, pp. 214–218.

[49] A. Q. Ohi, M. Mridha, M. M. Monowar, and M. A. Hamid, "Exploring optimal control of epidemic spread using reinforcement learning," *Scientific reports*, vol. 10, no. 1, p. 22106, 2020.

[50] T. L. Meng and M. Khushi, "Reinforcement learning in financial markets," *Data*, vol. 4, no. 3, p. 110, 2019.

[51] J. Gittins, K. Glazebrook, and R. Weber, *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[52] Y. Liu and L. Li, "A map of bandits for e-commerce," *arXiv preprint arXiv:2107.00680*, 2021.

[53] A. S. Leong, A. Ramaswamy, D. E. Quevedo, H. Karl, and L. Shi, "Deep reinforcement learning for wireless sensor scheduling in cyber–physical systems," *Automatica*, vol. 113, p. 108759, 2020.

[54] T. Blau, E. V. Bonilla, I. Chades, and A. Dezfouli, "Optimizing sequential experimental design with deep reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2022, pp. 2107–2128.

[55] D. P. Bertsekas *et al.*, "Dynamic programming and optimal control 3rd edition, volume ii," *Belmont, MA: Athena Scientific*, vol. 1, 2011.

[56] M. Gaon and R. Brafman, "Reinforcement learning with non-markovian rewards," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 3980–3987.

[57] K. J. Åström, "Optimal control of markov processes with incomplete state information i," *Journal of mathematical analysis and applications*, vol. 10, pp. 174–205, 1965.

[58] M. Pulis and J. Bajada, "Reinforcement learning for football player decision making analysis," in *StatsBomb Conference*, 2022.

[59] T. Sawczuk, A. Palczewska, and B. Jones, "Markov decision processes with contextual nodes as a method of assessing attacking player performance in rugby league," in *Advances in Computational Intelligence Systems: Contributions Presented at the 20th UK Workshop on Computational Intelligence, September 8-10, 2021, Aberystwyth, Wales, UK 20*. Springer, 2022, pp. 251–263.

[60] N. Ding, K. Takeda, and K. Fujii, "Deep reinforcement learning in a racket sport for player evaluation with technical and tactical contexts," *IEEE Access*, vol. 10, pp. 54 764–54 772, 2022.

[61] P. Jia, Q. Jia, T. Jiang, and J. Liu, "Observation strategy optimization for distributed telescope arrays with deep reinforcement learning," *The Astronomical Journal*, vol. 165, no. 6, p. 233, 2023.

[62] S. Yatawatta and I. M. Avruch, "Deep reinforcement learning for smart calibration of radio telescopes," *Monthly Notices of the Royal Astronomical Society*, vol. 505, no. 2, pp. 2141–2150, 2021.

[63] N. J. Treloar, N. Braniff, B. Ingalls, and C. P. Barnes, "Deep reinforcement learning for optimal experimental design in biology," *PLOS Computational Biology*, vol. 18, no. 11, p. e1010695, 2022.

[64] J. D. Lomas, J. Forlizzi, N. Poonwala, N. Patel, S. Shodhan, K. Patel, K. Koedinger, and E. Brunskill, "Interface design optimization as a multi-armed bandit problem," in *Proceedings of the 2016 CHI conference on human factors in computing systems*, 2016, pp. 4142–4153.

[65] E. Sellentin, M. Quartin, and L. Amendola, "Breaking the spell of gaussianity: forecasting with higher order fisher matrices," *Monthly Notices of the Royal Astronomical Society*, vol. 441, no. 2, pp. 1831–1840, 2014.

[66] H. Ye, R. J. Beamish, S. M. Glaser, S. C. Grant, C.-h. Hsieh, L. J. Richards, J. T. Schnute, and G. Sugihara, "Equation-free mechanistic ecosystem forecasting using empirical dynamic modeling," *Proceedings of the National Academy of Sciences*, vol. 112, no. 13, pp. E1569–E1576, 2015.

[67] "Electrofishing to assess a river's health," https://environmentagency.blog.gov.uk/2015/10/29/electrofishing-to-assess-a-rivers-health/, accessed: 2023-02-10.

[68] D. Alem, A. Clark, and A. Moreno, "Stochastic network models for logistics planning in disaster relief," *European Journal of Operational Research*, vol. 255, no. 1, pp. 187–206, 2016.