Implementing Probabilistic Numerical Solvers for Differential Equations

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Peter Nicholas Krämer
aus Wuppertal

Kopenhagen
2023

Abstract

The numerical solution of differential equations underpins a large share of simulation methods that are used in the natural sciences and engineering, both in research and in industrial applications. The usability of a differential equation model depends, often crucially, on the choice of the simulation algorithm. Probabilistic numerical algorithms promise to combine efficient simulations with well-calibrated uncertainty quantification. Being able to handle various sources of uncertainty without a severely increased computational burden simplifies the combination of differential equation models with, for example, observational data and thereby improves the fusion of mechanistic and statistical information.

However, until now, the general usability of probabilistic numerical solvers had not reached a level comparable to non-probabilistic approaches. A lack of numerical stability and scalability, combined with a strong focus on ordinary differential equations and initial value problems, put probabilistic numerical algorithms out of the scope of an implementation in the physical and the life sciences, which would require the efficient simulation of dynamics that may exhibit spatiotemporal patterns or could be constrained by boundary information.

This thesis explains a series of contributions to the solution of this problem by discussing the implementation of a class of probabilistic numerical differential equation solvers that shares many features with collocation methods and with Gaussian filtering and smoothing:

1. A set of instructions for the numerically stable implementation of probabilistic numerical differential equation solvers that scales to high-dimensional problems.

2. The generalisation of solvers for ordinary-differential-equation-based initial value problems to boundary value problems and partial differential equations.

Many of the techniques have already been implemented successfully in various software libraries for probabilistic numerical differential equation solvers.

Altogether, the contributions improve the usability of existing and future probabilistic numerical algorithms. The simulation of challenging differential equation models and an application of the probabilistic numerical paradigm to real-world problems is no longer out of reach.

Zusammenfassung (auf Deutsch)

Die numerische Lösung von Differenzialgleichungen ist eine wichtige Grundlage für viele Simulationsmethoden, welche in den Natur- und Ingenieurwissenschaften eingesetzt werden, sowohl in der Forschung als auch in der Industrie. Die Brauchbarkeit eines Differenzialgleichungsmodells hängt entscheidend von der Wahl des Simulationsalgorithmus ab und probabilistisch-numerische Algorithmen versprechen, effiziente Simulationen mit wohlkalibrierten Unsicherheitsschätzern zu kombinieren.

Die allgemeine Anwendbarkeit probabilistisch-numerischer Löser hat jedoch noch nicht das Niveau von nicht-probabilistischen Ansätzen erreicht. Ein Mangel an numerischer Stabilität und Skalierbarkeit, kombiniert mit einer starken Konzentration auf gewöhnliche Differenzialgleichungen und Anfangswertprobleme, macht probabilistische numerische Algorithmen für viele Probleme unbrauchbar; vor allem solche, die auf partiellen Differenzialgleichungen oder Randwertproblemen basieren.

Diese Arbeit erläutert eine Reihe von Beiträgen zur Lösung dieses Problems, speziell, zur Implementierung derjenigen Klasse von probabilistisch-numerischen Differenzialgleichungslösern, welche viele Gemeinsamkeiten mit Kollokationsmethoden und mit Gaußschen Filtern hat:

1. Eine Reihe an Strategien für die numerisch stabile und skalierbare Software-Implementierung von probabilistisch-numerischen Differenzialgleichungslösern.
2. Die Verallgemeinerung von probabilistisch-numerischen Methoden für auf gewöhnlichen Differenzialgleichungen basierende Anfangswertprobleme auf Randwertprobleme und partielle Differenzialgleichungen.

Viele der Techniken sind bereits erfolgreich in verschiedenen Softwarebibliotheken für probabilistische numerische Differenzialgleichungslöser implementiert worden.

Insgesamt verbessern die in dieser Arbeit vorgestellten Methoden die Nutzbarkeit existierender und zukünftiger probabilistischer numerischer Algorithmen. Die Simulation von anspruchsvollen Differenzialgleichungsmodellen und eine Anwendung des probabilistisch-numerischen Paradigmas auf reale Probleme wird damit ermöglicht.

Acknowledgements

I greatly appreciate having worked with Hans Kersting, Jonathan Wenger, Filip Tronarp, Katharina Ott, Nathanael Bosch, Jonathan Schmidt, Nina Effenberger, Emilia Magnani, Marvin Pförtner, and many others (listed roughly in chronological order). I especially acknowledge the numerous discussions with Nathanael, Filip, and Jonathan. I learned a lot; this thesis would have been very different without you.

I am deeply grateful to my supervisor, Philipp Hennig. You were there whenever I needed guidance, feedback, or a second opinion on a spontaneous idea. I especially value your patience while I was finishing this dissertation.

I owe gratitude to Jakob Macke, Caterina de Bacco, Gerard Pons-Moll, and Matthias Hein for examining this PhD thesis, forming my thesis advisory committee, or offering to take a formal role in an (upcoming) PhD defence, respectively.

A big "thank you" goes out to Franziska Weiler and everyone in the Methods of Machine Learning group in Tübingen. Katharina and Emilia, sharing an office with you has been excellent.

I am grateful to everyone who proofread this manuscript. All the remaining inaccuracies are – of course – my fault.

Abschließend bedanke ich mich bei meiner Familie und meinen Freunden, sowie bei allen anderen die indirekt an der Erstellung dieser Arbeit beteiligt waren; vor allem, bei Libby.

Nicholas Krämer,
December 6, 2023

Originality

The writing of this thesis is my original work. The content of each chapter is either my original work – or rather, a joint effort with coauthors as summarised below – or a reference to existing work, in which case it is cited as such.

Publications

The content of this dissertation relates to the following papers.

Paper 1. Nicholas Krämer and Philipp Hennig. *Stable implementation of probabilistic ODE solvers.* 2020. Accepted with minor revisions to JMLR. The revised version is currently under review. A preprint is available on arXiv:2012.10106. This is reference [100].

Each coauthor contributed as follows:

	Ideas	Analysis	Experiments	Writing
Nicholas Krämer	90 %	90 %	95 %	85 %
Philipp Hennig	10 %	10 %	5 %	15 %

Paper 2. Nicholas Krämer and Philipp Hennig. *Linear-time probabilistic solution of boundary value problems.* Advances in Neural Information Processing Systems 34 (2021). This is reference [101].

Each coauthor contributed as follows:

	Ideas	Analysis	Experiments	Writing
Nicholas Krämer	90 %	90 %	95 %	85 %
Philipp Hennig	10 %	10 %	5 %	15 %

Paper 3. Nicholas Krämer*, Nathanael Bosch*, Jonathan Schmidt*, and Philipp Hennig. *Probabilistic ODE solutions in millions of dimensions.* International Conference on Machine Learning 2022. This is reference [102].

The superscript '$*$' indicates equal contribution. Each coauthor contributed as follows:

	Ideas	Analysis	Experiments	Writing
Nicholas Krämer	30 %	30 %	30 %	30 %
Nathanael Bosch	30 %	30 %	30 %	30 %
Jonathan Schmidt	30 %	30 %	30 %	30 %
Philipp Hennig	10 %	10 %	10 %	10 %

Paper 4. Nicholas Krämer, Jonathan Schmidt and Philipp Hennig. *Probabilistic numerical method of lines for time-dependent partial differential equations.* International Conference on Artificial Intelligence and Statistics 2022. This is reference [103].

Each coauthor contributed as follows:

	Ideas	Analysis	Experiments	Writing
Nicholas Krämer	80 %	70 %	50 %	70 %
Jonathan Schmidt	10 %	20 %	40 %	20 %
Philipp Hennig	10 %	10 %	10 %	10 %

Detailed connections

Large parts of this dissertation are an extended summary of these articles; however, some statements are new. The precise relationship between the four papers and this thesis is as follows:

The content in Chapters 1 to 3 is known, but with a new presentation, and has been included as background information for the rest of this manuscript. The figure in Chapter 1 is from Paper 1. All other figures are new.

Chapters 4 to 7 are a more comprehensive version of Paper 1 and break down as follows.

Chapter 4 extends Paper 1 to include numerically stable whitening and computing log-probabilities, and distinguishes Bayesian updates with noisy and deterministic transformations; all of these are minimal extensions.

Chapter 5 is mostly known, but the interpretation of the work by Kersting and Hennig [91] as zeroth-order statistical linear regression is new (Tronarp et al. [163] present a slightly different version of this finding), and so is the interpretation of the work by Arvanitidis et al. [10] as iterated zeroth-order linearisation. To the best of my knowledge, the downdate-free Cholesky implementation of statistical linear regression is novel. These findings are contextualised in Chapter 5.

Chapter 6 is based on Paper 1 but discusses automatic differentiation more thoroughly; for example, Chapter 6 suggests an implementation of Taylor-mode differentiation via coefficient doubling, which is not contained in Paper 1. The regression-based initialisation in Chapter 6 is from Paper 3.

Chapter 7 is based on Paper 1 as well, except for the results on calibration, both time-varying and time-constant, which are discussed by Bosch et al. [25], Schober et al. [152], Tronarp et al. [163]. The chapter cites these works where relevant. Via Chapter 5, the statistical-linear-regression-based algorithms extend the work in Paper 1. All tables and figures in Chapter 7 are taken from Paper 1.

Chapter 8 is based on Paper 3, except for what will be referred to as "dense models", which are known and cited as such. The chapter presents the content (slightly) differently than the paper; for example, the matrix-normal perspective on Kronecker models is new. The vector-valued maximum-likelihood calibration of the output scale in Kronecker models is also new and generalises the result in Paper 3. All figures in this chapter are from Paper 3.

The benchmarks in Chapter 9 are new and have been created solely for this thesis.

Chapters 10 and 11 relate to Paper 4. The presentation in both chapters refines that in Paper 4, but the results are the same. All figures in those chapters are from Paper 4.

Chapter 12 corresponds to Paper 2. The chapter explains the bridge process and the initialisation more thoroughly than the paper but refers to previous chapters for background. The figure that shows samples from the bridged Wiener velocity process is new. All others are taken from Paper 2.

Miscellaneous

Besides working on the projects above, I have contributed to other articles during my PhD, namely, [92, 114, 125, 150, 178]. But these are not included in this dissertation.

I have also created and contributed to various open-source software projects related to differential equations and probabilistic numerical algorithms, most prominently:

1. `ProbDiffEq`: I am the creator of ProbDiffEq, a JAX implementation of probabilistic numerical IVP solvers, which provides all methods presented in Chapters 4 to 8 and has been used for the benchmarks in Chapter 9. ProbDiffEq's online documentation can be found under

   `https://pnkraemer.github.io/probdiffeq/`.

2. `ProbFinDiff`: I am the creator of ProbFinDiff, a JAX implementation of probabilistic numerical finite differences. ProbFinDiff's online documentation can be found under

   `https://probfindiff.readthedocs.io/`.

3. `ProbNum:` I have been one the main contributors to ProbNum, a Python implementation of various probabilistic numerical algorithms. ProbNum's online documentation can be found under

   `https://probnum.org/`

   and the corresponding paper [178] describes the library in more detail.

Other software outputs include creating a library that collects exemplary initial value problems in NumPy and JAX,[1] co-creating a previous iteration of a JAX implementation of probabilistic numerical solvers, [2] which has been used for Paper 3, and a library that simplifies the creation of Matplotlib-figures for scientific publications.[3]

Except for (perhaps) ProbDiffEq and ProbFindiff, none of these are directly connected to this thesis.

While the writing of this thesis is my original work, some parts of this manuscript have been proofread by Jonathan Schmidt, Nathanael Bosch, and Elizabeth Baker. The correct typography has been verified by an automatic spell-checker.[4]

[1]`https://diffeqzoo.readthedocs.io/`
[2]`https://github.com/pnkraemer/tornadox/`
[3]`https://tueplots.readthedocs.io/`
[4]`https://grammarly.com/`

Contents

Part I

Background

Chapter 1

Introduction

Contents

1.1 Motivation

This document aims to explain a series of contributions to the field of probabilistic numerical methods, thereby fulfilling one of the formal requirements of obtaining a doctoral degree.

As a side product, this thesis surveys some recent developments in the field of (state-space-model-based) probabilistic numerical differential equation solvers. It is the first text that comprehensively explains how to implement a probabilistic numerical solver for initial value problems.

1.2 Prerequisites

This manuscript assumes rough familiarity with Gaussian processes, stochastic differential equations, and Bayesian inference equivalent to what is covered by the first part of the book by Hennig et al. [79], or Chapters 2, 3, 10, and 12 in the book by Särkkä and Solin [144]. Chapter 10 expects basic knowledge of finite differences.

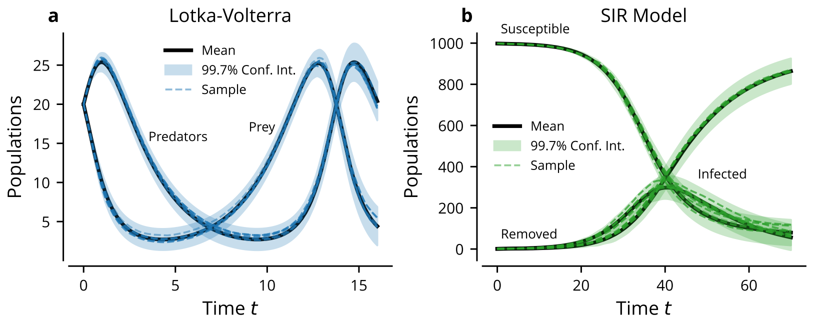

Figure 1.1: Probabilistic numerical solution of the Lotka–Volterra / predator-prey dynamics (left, a) and the susceptible-infected-removed model (right, b). Displayed are the posterior mean and confidence intervals of a probabilistic numerical solver and samples from the probabilistic numerical solution.

1.3 Simulating differential equations

Mechanistic models based on differential equations are a popular way of describing phenomena occurring in the natural sciences and engineering. For example, the outbreak of a disease approximately follows a susceptible-infected-removed model, certain instances of the Navier–Stokes equations describe the airflow around a wing, and the Hodgkin-Huxley equations emulate the spiking-behaviour of neurons.

Assume that a function $f : \mathbb{R}^d \rightarrow \mathbb{R}^d$ and vector $y_0 \in \mathbb{R}^d$ are given. Consider an ordinary differential equation,

$$\frac{\mathrm{d}}{\mathrm{d}t} y(t) = f(y(t)), \quad t \in [0, 1], \tag{1.1}$$

and an initial condition $y(0) = y_0$. For example, imagine the predator-prey dynamics or a susceptible-infected-removed model (Figure 1.1). Assume that the vector field f is locally Lipschitz-continuous on some open set V that contains y_0. Then, Equation (1.1) has a unique solution $y : [0, 1] \rightarrow \mathbb{R}^d$ [e.g., 79, Theorem 36.4]. If f is nonlinear, y does not arise in closed form (except in special cases) but requires the approximation with a numerical algorithm. This raises two questions:

First, *how much approximation error does the algorithm produce?* While approximation errors are unavoidable, we need to know whether the errors are satisfactorily low and, if not, how to refine the approximation. Second, *how well does the numerical algorithm interact with models built "around" the differential equation?* If subsequent computations rely on the solution of the differential equation, the numerical solver must be compatible with those algorithms. This is a common requirement when doing, for instance, parameter estimation with differential equation models. Phrasing the numerical simulation as inference in a probabilistic model, a perspective taken by probabilistic numerical methods, provides a single tool to answer both questions.

1.4 Probabilistic numerical methods

Probabilistic numerical methods – like those that solve differential equations – view numerical algorithms from the perspective of probabilistic inference. Usually, this involves computing a posterior distribution over the solution of the numerical problem (the differential equation) with Bayes' theorem,

$$\text{posterior} \propto \text{likelihood} \times \text{prior}. \tag{1.2}$$

The information required but also returned by Bayes' theorem is richer than with many numerical algorithms: an appropriate prior distribution may improve the performance of the numerical algorithms, the likelihood gathers "information" (in a technical sense; we discuss this in Chapter 2) about the problem to be solved, and the posterior distribution quantifies the uncertainty over the numerical approximation error. Furthermore, a probabilistic numerical method expresses an approximate solution as a probability distribution, which can be related to other quantities of interest by manipulating probability densities.

1.5 Overview of this manuscript

Chapters 1 to 3 discuss the background required for understanding the contributions of this manuscript: the connection between conditional distributions and collocation methods, as well as the sequential estimation of conditional distributions when Markovian prior distributions are used.

Chapters 4 to 9 provide the core of this manuscript: detailed instructions for implementing probabilistic numerical solvers for differential equations. This includes manipulating Gaussian probability distributions using only (generalised) Cholesky factors instead of covariance matrices (Chapter 4), linearising nonlinear constraints (Chapter 5), estimating Taylor-series (Chapter 6), and best practices for scalar- and vector-valued problems (Chapter 7, Chapter 8). Chapter 9 contains benchmarks of the probabilistic numerical solver against popular implementations of initial value problem solvers.

Chapters 10 to 12 treat selected topics related to the content of this thesis: Chapters 10 and 11 construct a probabilistic numerical solver for time-dependent partial differential equations; Chapter 12 explains how to implement a probabilistic numerical solver for boundary value problems. Both topics loosely depend on Chapter 7 and can be read independently. Figure 1.2 summarises the dependencies between the chapters and suggests in which order to read the chapters of this manuscript.

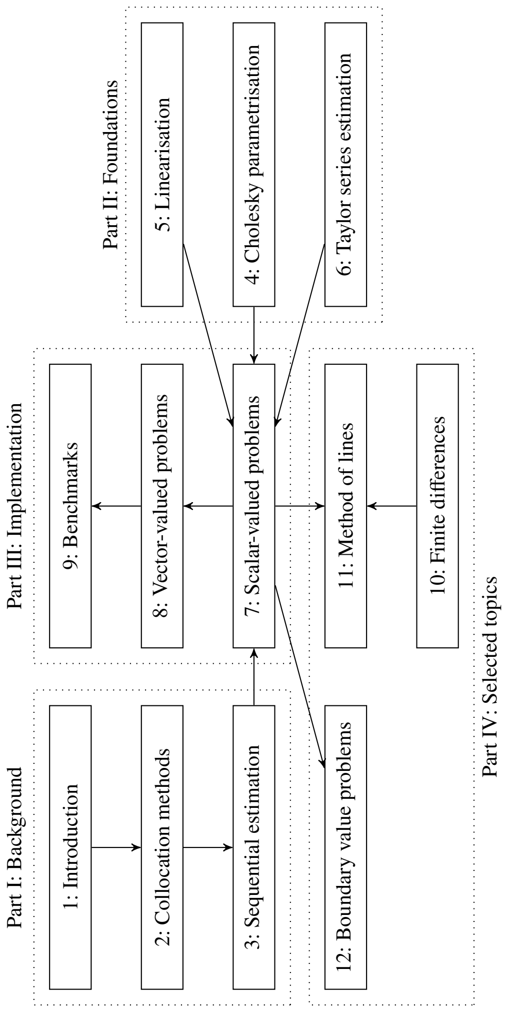

Figure 1.2: How to read this manuscript.

Chapter 2

Collocation methods

Contents

2.1 Introduction

This thesis discusses the class of probabilistic numerical solvers that implements (a probabilistic formulation of) collocation methods via estimation in state-space models. The present chapter motivates this framework, defines the terms “probabilistic numerical solution” and “probabilistic numerical solver”, and puts them into the context of contemporary definitions of Bayesian probabilistic numerical methods and collocation methods.

Let $p(\varphi)$ be an appropriate probability distribution over the set of functions $\{\varphi : [0, 1] \to \mathbb{R}^d\}$. For example, $p(\varphi)$ could be the law of a Gaussian process. Let $t_0, ..., t_N$ be known points in $[0, 1]$. Like in Chapter 1, we assume a function $f : \mathbb{R}^d \to \mathbb{R}^d$ and vector $y_0 \in \mathbb{R}^d$ to be given.

2.2 Probabilistic numerical solvers for initial value problems

The construction of probabilistic numerical solvers builds on the following idea: For a sufficiently well-posed problem and a sufficiently large number of grid points, any

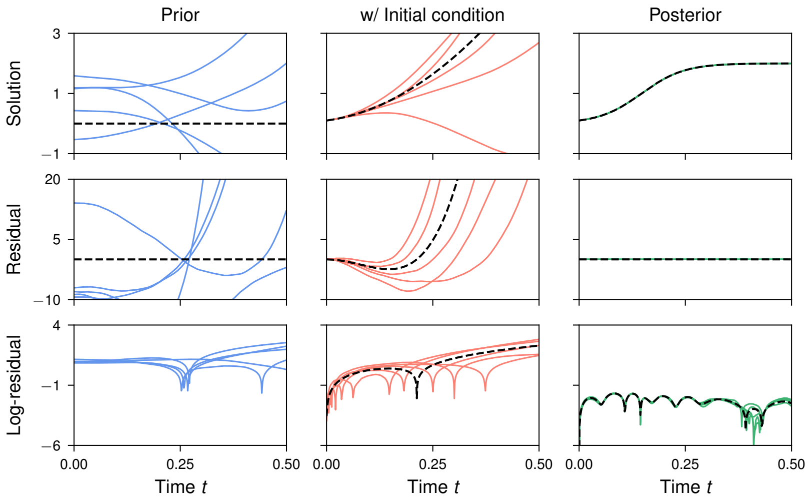

Figure 2.1: The samples from the prior, $\varphi \sim p(\varphi)$ (left column) have a residual $\frac{\mathrm{d}}{\mathrm{d}t}\varphi(t) - f(\varphi(t))$ with large magnitude (bottom left), even after incorporating the initial condition (middle column). By conditioning the prior on attaining a zero-residual (bottom right), an approximate solution of the logistic differential equation $\frac{\mathrm{d}}{\mathrm{d}t}y(t) = 10y(t)(1 - y(t))$, $y(t_0) = 0.1$, emerges (right column). Every plot shows five samples; $p(\varphi)$ is a twice-integrated Wiener process. The solver uses adaptive steps (with relative tolerance set to 10^{-1} and first-order Taylor linearisation.

sample from the conditional distribution

$$p\left(\varphi \;\middle|\; \left\{\frac{\mathrm{d}}{\mathrm{d}t}\varphi(t_n) = f\left(\varphi(t_n)\right)\right\}_{n=0}^{N},\ \varphi(0) = y_0\right), \tag{2.1}$$

approximately solves the ordinary differential equation

$$\frac{\mathrm{d}}{\mathrm{d}t}y(t) = f(y(t)), \quad t \in [0, 1], \tag{2.2}$$

and satisfies the initial condition $y(0) = y_0$. In other words, a sample from the conditional distribution approximately solves the initial value problem (Figure 2.1). Therefore, we refer to the distribution in Equation (2.1) as the *probabilistic numerical solution* of the problem in Equation (2.2). Any algorithm that computes or approximates the probabilistic numerical solution is a *probabilistic numerical solver* for the initial value problem. Constraining a hypothesis space of functions to satisfy a differential equation (and additional constraints) at a pre-specific point set is generally referred to as a *collocation method* (refer to, for example, Definition 7.6 in the book by Hairer et al. [76] or Section 10.3 in the book by Driscoll and Braun [47]). Therefore, we say that probabilistic numerical algorithms in the above definition implement probabilistic numerical variants of collocation methods (the "probabilistic" enters through formulating collocation in terms of prior and conditional distributions).

The above concept of a probabilistic numerical solution is a special case of what Cockayne et al. [37, Def. 2.5] define to be a *Bayesian probabilistic numerical method*. The specialisation lies in the fact that probabilistic numerical solutions in the above definition always assume a constraint of the form

$$\varphi \mapsto \begin{cases} \{\mathcal{L}(\varphi(t_n)) = 0\}_{n=0}^N \\ C(\varphi(0), \varphi(1)) = 0 \end{cases}, \tag{2.3}$$

with, for example,

$$\mathcal{L}(\varphi) := \frac{\mathrm{d}\varphi}{\mathrm{d}t} - f(\varphi), \quad C(\varphi(0), \varphi(1)) := \varphi(0) - y_0, \tag{2.4}$$

whereas a "Bayesian probabilistic numerical method" is more general.

However, the definition in Equations (2.3) and (2.4) itself is fairly general already: If, instead of initial value problems based on ordinary differential equations, the goal is to solve boundary value problems or partial differential equations, $\mathcal{L}$ and C differ from Equation (2.4), but the general structure remains. Through such a variation of $\mathcal{L}$ and C in Equation (2.3), the above definition of a probabilistic numerical solution includes a wide range of previously published algorithms: For example, all algorithms studied in the remainder of this manuscript fit this definition, and so do the partial differential equation solvers by Cockayne et al. [36], Owhadi [128, 129], Raissi et al. [137], and Raissi et al. [138], as well as the boundary value problem solvers by Arvanitidis et al. [10], Hennig and Hauberg [78], and John et al. [85]. Most of these algorithms are equivalent to known, non-probabilistic algorithms; concrete connections are explained in the respective chapters.

There exist other, non-collocation-based algorithms in the literature on probabilistic numerical methods for differential equations, for example, perturbation-based versions of non-probabilistic numerical integrators [e.g. 1, 2, 38]. But in the remainder of this manuscript, "probabilistic numerical solver/solution" will always refer to the above definition of a probabilistic numerical solution/solver, as it is the only class of methods to be discussed in this thesis.

2.3 Example: boundary value problems

One advantage of collocation methods over alternative algorithms is the simplicity of generalising an existing method to a new problem class. Probabilistic numerical solvers inherit this feature: For example, if boundary conditions replace the initial-value constraint, that is, if the goal is to solve the boundary value problem

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2} y(t) = f\left(y(t)\right), \quad \begin{cases} y(0) = y_0, \\ y(1) = y_{\max}, \end{cases} \tag{2.5}$$

instead of the initial value problem in Equation (2.2), the probabilistic numerical solution is a minor modification of Equation (2.1),

$$p\left(\varphi \;\middle|\; \left\{\frac{\mathrm{d}^2}{\mathrm{d}t^2}\varphi(t_n) = f\left(\varphi(t_n)\right)\right\}_{n=0}^{N}, \begin{cases} \varphi(0) = y_0 \\ \varphi(1) = y_{\max} \end{cases}\right). \tag{2.6}$$

The only difference between Equations (2.1) and (2.6) is that Equation (2.6) conditions on boundary information instead of initial-value information. As such, we define Equation (2.6) as the probabilistic numerical solution of the boundary value problem, and any algorithm that approximates Equation (2.6) is a probabilistic numerical solver for boundary value problems. Boundary value problems are the content of Chapter 12.

2.4 Example: partial differential equations

Similarly, probabilistic numerical solvers/solutions for partial differential equations emerge by modifying the constraints in Equation (2.1):

Let $\Omega \in \mathbb{R}^r$ be a sufficiently well-behaved domain with a sufficiently regular boundary $\partial\Omega$ (for example, open, bounded, Lipschitz-boundary). Let $\{x_0, ..., x_K\} \subseteq \Omega$ be a spatial grid in Ω. Assume that functions $u_0 : \Omega \longrightarrow \mathbb{R}^d$ and $u_\partial : [0,1] \times \Omega \longrightarrow \mathbb{R}^d$ are known. Let $p(\zeta)$ be a probability distribution over the space of spatiotemporal functions $\{\zeta : [0,1] \times \Omega \longrightarrow \mathbb{R}^d\}$, for example, the law of a Gaussian process.

Let the task be to solve an initial value problem based on a partial differential equation, say, a combination of the semilinear differential equation

$$\frac{\partial}{\partial t} u(t,x) = \frac{\partial^2}{\partial x^2} u(t,x) + f(u(t,x)), \quad (t,x) \in [0,1] \times \Omega, \tag{2.7}$$

with two constraints: the temporal initial condition $u(0,x) = u_0(x)$, $x \in \Omega$, and the spatial boundary condition $u(t,x) = u_\partial(t,x)$, $(t,x) \in [0,1] \times \partial\Omega$. Assume that the problem is sufficiently well-behaved that the partial differential equation admits a unique solution.

Let $\{\overline{x}_{k'}\}_{k'=0}^{K'}$ be the points in $\{x_0, ..., x_K\}$ that are on the boundary $\partial\Omega$ of Ω. The strategy for deriving the probabilistic numerical solution of the partial differential equation mirrors the previous approaches; we define it as the conditional distribution

$$p(\zeta \mid A_{\mathrm{PDE}},\ A_0,\ A_\partial), \tag{2.8a}$$

$$A_{\mathrm{PDE}} \coloneqq \left\{ \frac{\partial^2}{\partial t^2}\zeta(t_n, x_k) = \frac{\partial^2}{\partial x^2}\zeta(t_n, x_k) + f(\zeta(t_n, x_k)) \right\}_{k,n=0}^{K,N} \tag{2.8b}$$

$$A_0 \coloneqq \{\zeta(0, x_k) = u_0(x_k)\}_{k=0}^{K} \tag{2.8c}$$

$$A_\partial \coloneqq \{\zeta(t_n, \bar{x}_{k'}) = u_\partial(t_n, \bar{x}_{k'})\}_{t,k'=0}^{K',N}. \tag{2.8d}$$

Any algorithm that approximates the probabilistic numerical partial differential equation solution in Equation (2.8) is a probabilistic numerical solver for partial differential equations. Partial differential equations are treated by Chapters 10 and 11.

2.5 Discussion and outlook

Given a prior distribution and a differential-equation-based problem, it is relatively straightforward to derive a probabilistic numerical solution: introduce a discretisation, and condition the prior distribution on a set of collocation conditions; that is, on the event that samples from the distribution satisfy the differential equation and the constraints on the grid. Initial and boundary value problems based on ordinary differential equations and spatiotemporal partial differential equations were used as examples above; other types of differential equation problems (e.g. stationary partial differential equations) can be derived similarly but are out of the scope of this text.

While conceptualising a probabilistic numerical solution may be straightforward, constructing efficient solvers is more difficult. Some of the reasons are nonlinearities, boundary conditions, or spatial differential operators but the algorithms must also be able to compute the solution to low tolerances and be robust against anisotropic behaviour and numerical round-off errors. In other words, one needs to be meticulous with the selection and the implementation of approximation methods that target posterior distributions such as those in Equations (2.1), (2.6) and (2.8).

Most of this manuscript discusses the efficient and stable approximation of the posterior probability distributions that make up probabilistic numerical solutions of differential equations. As a foundation for these discussions, Chapter 3 describes the conceptualisation of probabilistic numerical solvers for ordinary-differential-equation-based initial value problems via sequential estimation in state-space models.

Chapter 3

Sequential estimation

Contents

3.1 Problem statement

Chapter 2 defined probabilistic numerical solvers/solutions and explained the connection to collocation methods. The upcoming chapter lays the foundation for discussing the efficient and stable implementation of probabilistic numerical solvers. More specifically, the following few sections explain how, if the prior distribution is Markovian, the posterior distribution can be computed with a constant number of floating-point operations per grid point.

The remainder of this chapter restricts itself to solving initial value problems (IVPs) based on affine, scalar ordinary differential equations. This class of IVPs combines an affine, scalar differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = ay(t) + b, \quad t \in [0, 1], \tag{3.1}$$

with the constraint $y(0) = y_0 \in \mathbb{R}$ on the initial state. The coefficients $a, b \in \mathbb{R}$ and the initial condition y_0 are given and shall be sufficiently well-behaved so that a unique IVP solution $y : [0, 1] \to \mathbb{R}$ exists.

Equation (3.1) assumes that the differential equation is affine because the probabilistic numerical solution of an affine problem with a Gaussian prior is Gaussian. Probabilistic numerical solutions of nonlinear problems demand approximation, algorithms for which build on the algorithms for affine problems. Therefore, approximate estimation and nonlinear problems will not be discussed before Chapter 5. Equation (3.1) further assumes that the differential equation is one-dimensional because the efficient solution of vector-valued problems (Chapter 8) builds on algorithms for scalar problems (explained in this chapter and Chapter 7). Equation (3.1) makes several notational simplifications:

- ◇ The coefficients a and b are constant instead of time-dependent.
- ◇ The differential equation describes $\frac{\mathrm{d}y(t)}{\mathrm{d}t}$ instead of, for example, $\frac{\mathrm{d}^2y(t)}{\mathrm{d}t^2}$.
- ◇ The time-domain is $t \in [0,1]$ instead of, for instance, $t \in [t_0, t_{\max}]$.

All three assumptions do not imply a loss of generality; they exclusively serve the purpose of reducing the number of symbols in the upcoming mathematical expressions. Modifications for the general cases will be discussed where relevant.

Let $\{t_0, ..., t_N\} \subseteq [0,1]$ be a set of $N+1$ grid-points. Assume that the outermost grid points coincide with the boundary of the integration domain, $t_0 = 0$, $t_N = 1$. (For general time intervals, place the grid accordingly.) Let $p(\varphi)$ be a (prior) probability distribution on the set of functions $\{\varphi : [0,1] \to \mathbb{R}\}$. Like in Chapter 2, define the probabilistic numerical IVP solution as the conditional distribution

$$p\left(\varphi \;\middle|\; \left\{\frac{\mathrm{d}\varphi(t_n)}{\mathrm{d}t} = a\varphi(t_n) + b\right\}_{n=0}^{N}, \; \varphi(0) = y_0\right). \tag{3.2}$$

The difference between the distribution in Equation (3.2) and the probabilistic numerical solution in Chapter 2 is that Equation (3.2) relates to the affine IVP in Equation (3.1) instead of a nonlinear IVP.

The remainder of this chapter explains how to estimate and calibrate the conditional distribution in Equation (3.2) sequentially if the prior distribution is a Gaussian process and comes with the Markov property. To this end, Section 3.2 introduces a formulation of integrated Wiener processes as solutions of stochastic differential equations, Section 3.3 discusses temporal discretisation, and Section 3.4 treats the ordinary differential equation constraints. This will complete the probabilistic model behind the sequential version of probabilistic numerical solvers. Section 3.5 summarises implementing the sequential decomposition of the probabilistic numerical solution, and Section 3.6 describes the calibration of estimators in this context.

3.2 Integrated Wiener processes

The prior distribution $p(\varphi)$ encodes a priori information about the IVP solution; for example, how many times the solution is differentiable or if the process decays to zero in the long term. However, strict efficiency requirements constrain the class of reasonable priors for probabilistic numerical solvers. The computational complexity of solving a one-dimensional IVP on N grid points should scale as $O(N)$ because that is how expensive most non-probabilistic numerical solvers are. The stochastic differential equation perspective taken in this work implies this linear-time complexity. One could also explore alternative concepts, for instance, via Gaussian processes based on (approximately) structured matrices [e.g., 134], but the stochastic differential equation is natural for problems with temporal structure, such as solving IVPs.

The (at the time of writing) most common prior distribution for probabilistic numerical IVP solvers is a class of multiply-integrated Wiener processes defined as follows (e.g., used by Bosch et al. [25], Kersting et al. [93], Schober et al. [151], Tronarp et al. [163] and the projects discussed in this manuscript). Let $\nu \in \mathbb{N}$ be an integer. Let $w : \mathbb{R} \to \mathbb{R}$ be a Wiener process with a constant output scale $\gamma > 0$ [e.g., 144, Definition 4.1]. Let $m_0 \in \mathbb{R}^{\nu+1}$ be a vector and let $C_0(\gamma) \in \mathbb{R}^{(\nu+1)\times(\nu+1)}$ be a symmetric, positive semidefinite matrix. Define a stack of stochastic processes

$$Y(t) := \left(Y^{(0)}(t), ..., Y^{(\nu)}(t)\right) : \mathbb{R} \to \mathbb{R}^{\nu+1}, \tag{3.3}$$

with $Y^{(q)}(t) \in \mathbb{R}$, $q = 0, ..., \nu$, as the solution of a system of stochastic differential equations (in the Itô sense [127]),

$$\mathrm{d}Y^{(q)}(t) = Y^{(q+1)}(t)\,\mathrm{d}t, \quad q = 0, ..., \nu - 1, \tag{3.4a}$$

$$\mathrm{d}Y^{(\nu)}(t) = \mathrm{d}w(t), \tag{3.4b}$$

subject to the Gaussian initial condition

$$p(Y(0) \mid \gamma) = \mathcal{N}(m_0, C_0(\gamma)). \tag{3.5}$$

This construction of $Y(t)$ as the output of a linear stochastic differential equation with a Gaussian initial condition makes it a Gauss–Markov process [127]. The zeroth component $Y^{(0)}(t)$ of $Y(t)$ is a *ν-times integrated Wiener process*. The qth component $Y^{(q)}(t)$ of $Y(t)$ is the derivative of the $(q-1)$th component $Y^{(q-1)}(t)$ (Equation (3.4a)). The νth derivative $Y^{(\nu)}$ of $Y^{(0)}$ is a Wiener process (Equation (3.4b)).

In the remainder of this chapter, we assume that m_0, $C_0(\gamma)$, ν, and $t_0, ..., t_N$ are fixed and known, and that γ is fixed and unknown. Section 3.6 discusses the estimation of the parameter γ.

3.3 Time discretisation

We must discretise the prior distribution on $\{t_0, ..., t_N\}$ to compute a probabilistic numerical IVP solution. Let $t \in [0, \infty)$ and assume a time increment $\Delta t > 0$. Define the indicator function $\mathbf{1}_{i \leq j}(i, j)$ as equalling 1 if $i \leq j$ holds and 0 otherwise.

Solutions of linear, time-invariant, stochastic differential equations subject to Gaussian initial conditions satisfy the transition rule [15]

$$p(Y(t + \Delta t) \mid Y(t), \gamma) = \mathcal{N}(\Phi_\nu(\Delta t) Y(t), \Sigma_\nu(\Delta t, \gamma)), \tag{3.6}$$

for a transition matrix $\Phi_\nu(\Delta t) \in \mathbb{R}^{(\nu+1)\times(\nu+1)}$ and a process noise covariance matrix $\Sigma_\nu(\Delta t, \gamma) \in \mathbb{R}^{(\nu+1)\times(\nu+1)}$. Equation (3.6) is the *equivalent discretisation* of $Y(t)$ [e.g. 144, p. 79]. It implies that the joint distribution of $Y(t_0), ..., Y(t_N)$ factorises into a sequence of conditional distributions. For ν-times integrated Wiener processes, $\Phi_\nu(\Delta t)$ and $\Sigma_\nu(\Delta t, \gamma)$ are available in closed form,

$$\Phi_\nu(\tau) := [\phi_{ij}(\tau)]_{i,j=0}^{\nu}, \qquad \phi_{ij}(\tau) := \mathbf{1}_{i \leq j}(i, j) \frac{\tau^{j-i}}{(j-i)!} \tag{3.7a}$$

$$\Sigma_\nu(\tau, \gamma) := \gamma^2 [\sigma_{ij}(\tau)]_{i,j=0}^{\nu}, \quad \sigma_{ij}(\tau) := \frac{\tau^{2\nu+1-i-j}}{(2\nu + 1 - i - j)(\nu - i)!(\nu - j)!}. \tag{3.7b}$$

The closed-form availability of the transition matrices simplifies (and accelerates) implementations of probabilistic numerical IVP solvers.

Remark 3.1 (General Gauss–Markov priors). Instead of ν-times integrated Wiener processes, alternatives could be considered. Let $F_0, ..., F_\nu \in \mathbb{R}$ be given. Instead of the system of linear, time-invariant, stochastic differential equations in Equation (3.4), one could model

$$\mathrm{d}Y^{(q)}(t) = Y^{(q+1)}(t)\,\mathrm{d}t, \quad q = 0, ..., \nu - 1, \tag{3.8a}$$

$$\mathrm{d}Y^{(\nu)}(t) = \sum_{q=0}^{\nu} F_q Y^{(q)}(t)\,\mathrm{d}t + \mathrm{d}w(t) \tag{3.8b}$$

subject to the same Gaussian initial condition as in Equation (3.5). This formulation includes ν-times integrated Ornstein-Uhlenbeck processes and half-integer Matèrn processes as well as the ν-times integrated Wiener process from above [164]. The parameters $\Phi_\nu(\Delta t)$ and $\Sigma_\nu(\Delta t, \gamma)$ of the transition in Equation (3.6) are still available in closed form. However, their computation requires evaluating matrix exponentials [144], which costs more than evaluating Equation (3.7). Refer to Bosch et al. [27] for more information.

By choosing a stochastic differential equation formulation of the integrated Wiener process prior, we do not estimate Equation (3.2) but

$$p\left(Y^{(0)}(t), ..., Y^{(\nu)}(t) \;\middle|\; \left\{Y^{(1)}(t_n) = aY^{(0)}(t_n) + b\right\}_{n=0}^{N},\ Y^{(0)}(t_0) = y_0\right). \tag{3.9}$$

Loosely speaking, the posterior distribution in Equation (3.9) is "richer" than the one in Equation (3.2) because Equation (3.9) additionally includes derivative estimates. It is often easier to implement Equation (3.9) in software than Equation (3.2) because the differential operator $\varphi \to \frac{\mathrm{d}}{\mathrm{d}t}\varphi^{(0)}$ (Equation (3.2)) is replaced by a selection operator $\varphi \to \varphi^{(1)}$ (Equation (3.9)). In the present setting, both operators are equivalent since for any $q \in \{0, ..., \nu\}$, $Y^{(q)}(t)$ is the qth derivative of $Y^{(0)}(t)$. Since the differential equation is affine, both conditional distributions are Gaussian and Equation (3.9) and Equation (3.2) translate into one another.

Solutions of linear, time-invariant, stochastic differential equations driven by a Wiener process have the Markov property. That means that for $t_0 < t_1 < ... < t_N$, and $Y(t)$ as in Section 3.2, future values of the process are conditionally independent of past values given present values. More formally, for $t_n \in \{t_0, ..., t_N\}$ and $s > t_n$,

$$p(Y(s) \mid Y(t_0), ..., Y(t_n)) = p(Y(s) \mid Y(t_n)) \tag{3.10}$$

holds. Similarly, past values are conditionally independent of future realisations given present values: for $t_n \in \{t_0, ..., t_N\}$ and $s < t_n$, we have

$$p(Y(s) \mid Y(t_n), ..., Y(t_N)) = p(Y(s) \mid Y(t_n)). \tag{3.11}$$

The Markov property is essential for the sequential estimation of IVP solutions. We call processes with the Markov property "Markovian" and use the identities in Equations (3.10) and (3.11) later (in Section 3.5).

3.4 Constraints

Introduce a variable $\mathcal{R}_{y_0}$ and a set of variables $\{\mathcal{R}_{a,b,n}\}_{n=0}^{N}$ as

$$\mathcal{R}_{y_0} := Y^{(0)}(t_0) - y_0, \quad \mathcal{R}_{a,b,n} := Y^{(1)}(t_n) - aY^{(0)}(t_n) - b, \quad n = 0, ..., N. \tag{3.12}$$

By definition, $\mathcal{R}_{y_0}$ and $\mathcal{R}_{a,b,n}$ depend on the differential equation. If the differential equation changes, $\mathcal{R}_{y_0}$ and $\mathcal{R}_{a,b,n}$ change; for example, if a and b depend on time, $\mathcal{R}_{a,b,n}$ becomes

$$\mathcal{R}_{a,b,n} := Y^{(1)}(t_n) - a(t_n)Y^{(0)}(t_n) - b(t_n). \tag{3.13}$$

But in the remainder of this chapter, a and b shall be constant.

Loosely speaking, the magnitudes of $\mathcal{R}_{y_0}$ and each $\mathcal{R}_{a,b,n}$ indicate "how well" a sample from $Y(t_n)$ solves the problem: the smaller each variable in magnitude, the "closer" $Y(t_n)$ is to a solution of the IVP at t_n. By construction, because each $\mathcal{R}_{y_0}$ and $\{\mathcal{R}_{a,b,n}\}_{n=0}^N$ are deterministic transformations of the state variable $Y(t)$, they are conditionally independent given $Y(t)$,

$$p(\mathcal{R}_{a,b,n} \mid \{Y(t_k)\}_{k=0}^n, \{\mathcal{R}_{a,b,k}\}_{k=0}^{n-1}, \mathcal{R}_{y_0}) = p(\mathcal{R}_{a,b,n} \mid Y(t_n)). \tag{3.14}$$

This conditional independence will be important in Section 3.5.

3.5 Sequential estimation

Probabilistically solving the IVP becomes the problem of estimating $\{Y(t_n)\}_{n=0}^N$ from the constraints $\{\mathcal{R}_{y_0} = 0\}$ and $\{\mathcal{R}_{a,b,n} = 0\}_{n=0}^N$. We will sometimes omit the "= 0" in the explanations below to simplify the exposition. In the following derivations, abbreviate $Y(t_{0:n}) := \{Y(t_k)\}_{k=0}^n$ and $\mathcal{R}_{a,b,0:n} := \{\mathcal{R}_{a,b,k}\}_{k=0}^n$.

Since the prior distribution has the Markov property and since the constraints are conditionally independent, the probabilistic numerical solution inherits Markovianity from the prior distribution; it factorises as

$$\begin{aligned} &p(Y(t_{0:N}) \mid \mathcal{R}_{a,b,0:N}, \mathcal{R}_{y_0}, \gamma) \\ &\quad = p(Y(t_N) \mid \mathcal{R}_{a,b,0:N}, \mathcal{R}_{y_0}, \gamma) \prod_{n=0}^{N-1} p(Y(t_n) \mid Y(t_{n+1}), \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma). \end{aligned} \tag{3.15}$$

As announced above, the "= 0" has been omitted in the conditionals. Together, the backward-transition densities

$$\{p(Y(t_n) \mid Y(t_{n+1}), \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)\}_{n=0}^{N-1} \tag{3.16}$$

and the terminal distribution

$$p(Y(t_N) \mid \mathcal{R}_{a,b,0:N}, \mathcal{R}_{y_0}, \gamma) \tag{3.17}$$

represent the probabilistic numerical IVP solution. Computing these transitions and the terminal distribution yields the probabilistic numerical solution. The *marginal likelihood of the constraints*, which assesses the correctness of the probabilistic model (the larger, the better), factorises similarly,

$$\begin{aligned} &p(\mathcal{R}_{a,b,0:N}, \mathcal{R}_{y_0} \mid \gamma) \\ &\quad = p(\mathcal{R}_{a,b,0}, \mathcal{R}_{y_0} \mid \gamma) \prod_{n=1}^{N} p(\mathcal{R}_{a,b,n} \mid \mathcal{R}_{a,b,0:n-1}, \mathcal{R}_{y_0}, \gamma). \end{aligned} \tag{3.18}$$

The elements in Equations (3.16) to (3.18) can be computed in a single sweep:

Algorithm 3.2 (Sequential IVP solver)**.** Compute the probabilistic numerical IVP solution and the marginal likelihood (Equations (3.16) to (3.18)) as follows.

1. Initialise the algorithm with the initial distribution $p(Y(t_0) \mid \gamma)$ from the stochastic differential equation (Equation (3.5)). Construct the joint distribution of the initialisation and the constraints at t_0, $p(Y(t_0), \mathcal{R}_{a,b,0}, \mathcal{R}_{y_0} \mid \gamma)$. Extract the conditional $p(Y(t_0) \mid \mathcal{R}_{a,b,0}, \mathcal{R}_{y_0}, \gamma)$ and, optionally, the marginal likelihood

$$p(\mathcal{R}_{a,b,0}, \mathcal{R}_{y_0} \mid \gamma) = \int p(\mathcal{R}_{a,b,0}, \mathcal{R}_{y_0}, Y(t_0) \mid \gamma)\, \mathrm{d}Y(t_0). \tag{3.19}$$

   Store both.

2. For $n = 0, ..., N-1$, assume the availability of $p(Y(t_n) \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)$ from previous computations and compute the next set of terms as follows:

   (a) Construct the joint distribution $p(Y(t_n), Y(t_{n+1}) \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)$. Extrapolate

$$\begin{aligned} &p(Y(t_{n+1}) \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma) \\ &\qquad = \int p(Y(t_{n+1}), Y(t_n) \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)\, \mathrm{d}Y(t_n) \end{aligned} \tag{3.20}$$

   and derive the backward transition

$$p(Y(t_n) \mid Y(t_{n+1}), \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma). \tag{3.21}$$

   Use the extrapolation in the step below. Store the backward transition.

   (b) Use the extrapolation $p(Y(t_{n+1}) \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)$ to construct the joint distribution $p(\mathcal{R}_{a,b,n+1}, Y(t_{n+1}) \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)$. Use the joint distribution to estimate $p(Y(t_{n+1}) \mid \mathcal{R}_{a,b,0:n+1}, \mathcal{R}_{y_0}, \gamma)$. Store the conditional. Optionally, extract the likelihood increment

$$\begin{aligned} &p(\mathcal{R}_{a,b,n+1} \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma) \\ &\qquad = \int p(\mathcal{R}_{a,b,n+1}, Y(t_{n+1}) \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)\, \mathrm{d}Y(t_{n+1}). \end{aligned} \tag{3.22}$$

Return the backward transitions, conditional distributions, and the optional marginal likelihood terms.

Since the initial value problem is affine, all distributions in Algorithm 3.2 are Gaussian, in which case marginals and conditionals can be computed exactly. In fact, almost every step in Algorithm 3.2 involves manipulating such a Gaussian probability distribution; therefore, the efficiency of this operation is crucial for the computational feasibility of the probabilistic numerical simulation. Concrete implementations of each operation in Algorithm 3.2 depend on the parametrisation of Gaussian distributions. Recommendations for parametrising Gaussian distributions in the context of probabilistic numerical solvers are contributions of this thesis and postponed to Chapter 4.

Algorithm 3.2 produces a collection of transition rules

$$\{p(Y(t_n) \mid Y(t_{n+1}), \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)\}_{n=0}^{N}, \tag{3.23}$$

a set of conditional distributions

$$\{p(Y(t_n) \mid \mathcal{R}_{a,b,0:n}, \mathcal{R}_{y_0}, \gamma)\}_{n=0}^{N}, \tag{3.24}$$

and, optionally, marginal likelihood terms

$$\{p(\mathcal{R}_{a,b,n} \mid \mathcal{R}_{a,b,0:n-1}, \mathcal{R}_{y_0}, \gamma)\}_{n=1}^{N} \cup \{p(\mathcal{R}_{a,b,0}, \mathcal{R}_{y_0} \mid \gamma)\}. \tag{3.25}$$

In the literature on filtering and Rauch–Tung–Striebel smoothing, the conditional distributions in Equation (3.24) are known as the *filtering distributions*, and the process of computing them is called *filtering* [e.g., 143].

Equation (3.24) includes the terminal distribution $p(Y(t_N) \mid \mathcal{R}_{a,b,0:N}, \mathcal{R}_{y_0}, \gamma)$, which describes the probabilistic numerical IVP solution at the terminal grid point t_N. The backward transitions in Equation (3.23) are those from Equation (3.15). The combination of a terminal distribution with backward transitions enables the extraction of information from the IVP solution, for example, via marginalisation and sampling:

- ◇ *Sampling:* To sample a realisation of the probabilistic numerical IVP solution, draw a terminal realisation $\zeta_N \sim p(Y(t_N) \mid \mathcal{R}_{a,b,0:N}, \mathcal{R}_{y_0}, \gamma)$ and sequentially draw realisations

  $$\zeta_n \sim p(Y(t_n) \mid Y(t_{n+1}) = \zeta_{n+1},\ \mathcal{R}_{a,b,0:N},\ \mathcal{R}_{y_0}, \gamma) \tag{3.26}$$

  for $n = N-1, ..., 0$.

- ◇ *Marginalisation:* The terminal distribution is $p(Y(t_N) \mid \mathcal{R}_{a,b,0:N}, \mathcal{R}_{y_0}, \gamma)$. Compute the remaining marginals sequentially using the backward transition rules produced by Algorithm 3.2. In the filtering and smoothing literature, this backward marginalisation produces the *smoothing distributions* [e.g., 143].

The factorised representation of the probabilistic numerical solution produced by

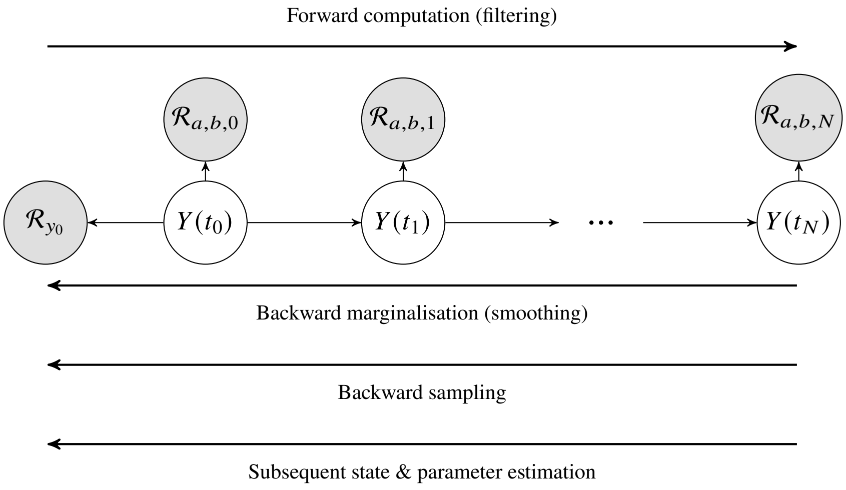

Figure 3.1: A forward pass (from $Y(t_0)$ to $Y(t_N)$) via $\mathcal{R}_{y_0} = 0$, $\mathcal{R}_{a,b,0} = 0$, ..., $\mathcal{R}_{a,b,N} = 0$) computes the filtering distributions, the backward transition densities, and, optionally, the marginal likelihood terms according to Algorithm 3.2. A backward pass (from $Y(t_N)$ to $Y(t_0)$) turns the filtering distributions into the marginal posterior distributions, computes samples, or allows using the probabilistic numerical IVP solution for subsequent state and parameter estimation.

Algorithm 3.2 can also serve as a prior for subsequent state/parameter estimation; refer to Tronarp et al. [165] for an example. Figure 3.1 visualises the scheme.

3.6 Calibration

So far, we assumed the following degrees of freedom as provided: the IVP (including the parameters of a and b and an initial condition y_0); the number $\nu \in \mathbb{N}$ of derivatives in the prior process, the output scale γ of the underlying Wiener process, the initial mean m_0 and covariance matrix $C_0(\gamma)$ of the ν-times integrated Wiener process; and the time-grid $\{t_0, ..., t_N\}$. In this section, we discuss the calibration of each of these parameters – or rather, explain why the selection of these parameters is unimportant at this point of the exposition and where to find such information instead.

We continue assuming that the IVP itself is known - this thesis focuses on state estimation; for parameter estimation with probabilistic numerical solvers, we refer to Kersting et al. [92], Schmidt et al. [150], Tronarp et al. [165].

The number of derivatives ν shall also be known. The choice of ν affects the convergence speed of the probabilistic numerical solver, but increasing ν increases the computational complexity. The trade-off between convergence speed and computational complexity is shown in Chapter 9.

Similarly, the more time points $\{t_0, ..., t_N\}$ we employ (with appropriate spacing), the better the approximation quality. However, the goal is not to compute a maximally accurate solution but rather to compute a sufficiently accurate solution on as few points as possible. Therefore, grid points for IVP solvers are chosen adaptively and during the forward computation (see [25, 152]). Grid selection for boundary value problem solvers is part of Chapter 12.

The parameters of the initial condition of the stochastic differential equation consist of the initial mean m_0 and the initial covariance $C_0(\gamma)$. In the absence of concrete knowledge of these parameters, one may use *diffuse initialisation* [39], that is, choose

$$m_0 = 0, \quad C_0(\gamma) = \kappa^2 \gamma^2 I_{(\nu+1)\times(\nu+1)} \tag{3.27}$$

with a large $\kappa > 0$ (for instance $\kappa = 10^3$). If suitable initial conditions of the stochastic differential equation emerge otherwise (for example, by corresponding to a stochastic differential equation description of a Gaussian process [e.g. 159]), use those instead of Equation (3.27).

But for initial value problems, the initial condition of the stochastic differential equation is not very important: By construction, the state $Y(t) = (Y^{(0)}(t), ..., Y^{(\nu)}(t))$ estimates the unnormalised Taylor coefficients of the IVP solution y,

$$Y^{(0)}(t_n) \approx y(t_n), \quad Y^{(1)}(t_n) \approx \frac{\mathrm{d}}{\mathrm{d}t} y(t_n), \quad ..., \quad Y^{(\nu)}(t_n) \approx \frac{\mathrm{d}^\nu}{\mathrm{d}t^\nu} y(t_n), \tag{3.28}$$

for $n = 0, ..., N$. The initial constraints $\{\mathcal{R}_{a,b,0} = 0, \mathcal{R}_{y_0} = 0\}$ initialise the zeroth and the first unnormalised Taylor coefficients correctly (see Chapter 7). In practice, probabilistic numerical IVP solvers use y_0 and the differential equation to replace $\mathcal{R}_{y_0}$ with constraints that initialise *all* Taylor coefficients (more or less) exactly and usually with a procedure that does not depend on m_0 and $C_0(\gamma)$ (Chapter 7).

The output scale γ of the underlying Wiener process is the only parameter whose selection is discussed in this section (even though concrete formulas are postponed for γ, too). The parameter γ governs the output scale of the process and must be calibrated against the IVP solution. For example, (quasi-)maximum-likelihood estimation is possible in closed-form and as a part of Algorithm 3.2. The precise formula depends on the factorisation and parametrisation of the Gaussian probability distributions. Therefore, we must postpone further instructions about calibrating the constant output scale γ to Chapters 7 and 8.

But a single γ is sometimes too restrictive: differential equations with strongly varying solutions require a time-varying output scale $\gamma : \mathbb{R} \to \mathbb{R}$. For example, the linear equation $\frac{\mathrm{d}y(t)}{\mathrm{d}t} = 2y(t)$, $y(0) = 1$, has the solution $y(t) = \exp(2t)$, and no single

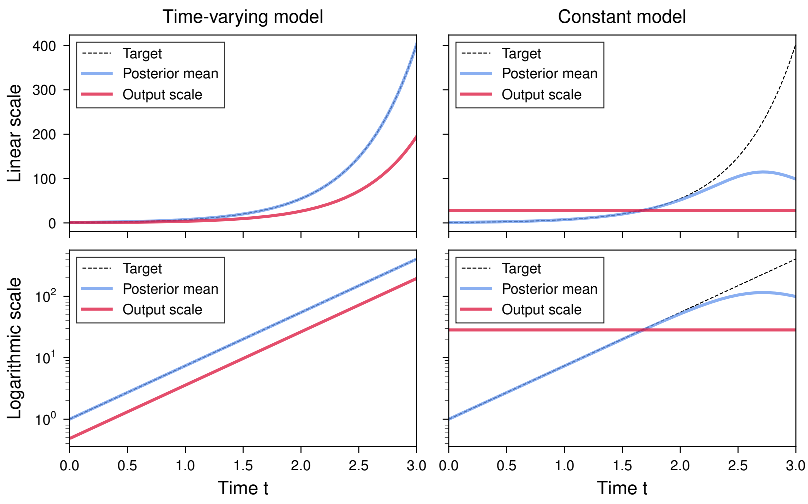

Figure 3.2: The probabilistic numerical solution of the linear differential equation $\frac{\mathrm{d}y(t)}{\mathrm{d}t} = 2y(t), y(0) = 1$, requires a time-varying output scale to adapt to the exponential growth of the solution $y = \exp(2t)$ (left column). With a constant output scale, the posterior mean does not follow the exponential growth of the solution (right column). (The prior has $\nu = 1$ derivatives. The solution is estimated on a fixed grid with $N = 200$ equispaced points.)

γ describes the scale of y accurately for all $t \in [0, 1]$ (Figure 3.2). While general, time-varying output scales as functions from $\mathbb{R}$ to $\mathbb{R}$ have not been explored in the literature, piecewise-constant output scales frequently appear [e.g. 25, 100, 102, 152]. The central assumption behind a piecewise-constant, time-varying output scale is to assume a sequence $\{\gamma_n\}_{n=0}^N \subseteq \mathbb{R}$ and to define $\gamma : \mathbb{R} \to \mathbb{R}$ as

$$\gamma(t) := \begin{cases} \gamma_0 & \text{if } t \in (-\infty, t_0], \\ \gamma_n & \text{if } t \in (t_n, t_{n+1}],\ n = 0, ..., N-1, \\ \gamma_N & \text{if } t \in (t_N, \infty]. \end{cases} \tag{3.29}$$

This output scale allows variation over time. Nevertheless, it implies a constant output scale per IVP solver step, in which case closed-form (quasi-)maximum-likelihood estimation remains available (under mild assumptions; details are in Chapters 7 and 8).

3.7 Conclusion

In summary, Markovian prior distributions imply the sequential estimation of probabilistic numerical IVP solutions. While multiply-integrated Wiener processes are the standard choice in the contemporary literature on probabilistic numerical methods, alternative prior distributions are possible. Prior distributions with the Markov property imply posterior distributions with the Markov property, and marginalisation and sampling can be computed in a constant number of operations per grid point.

One gap in the presentation is the implementation of the joint, marginal, and conditional distributions. Concrete algorithms are the content of Chapter 7. Before that, we must discuss the Cholesky parametrisation of Gaussian variables in Chapter 4, and linearisation in Chapter 5.

Part II

Foundations

Chapter 4

Cholesky parametrisation

Contents

4.1 Introduction

Manipulating Gaussian random variables is essential to implementing probabilistic numerical initial-value-problem solvers. There are many ways to parametrise Gaussian distributions, for example, using covariance matrices, precision matrices, or (generalised) Cholesky factors. In the present chapter, we explain the manipulation of Gaussian variables in Cholesky arithmetic, which means that we never assemble full covariance matrices – all operations involve only (generalised) Cholesky factors. Using only the (generalised) Cholesky factors preserves the symmetry and positive semidefiniteness of covariance matrices.

In other words, this chapter explains the numerical linear algebra necessary for implementing probabilistic numerical initial-value-problem solvers. Section 4.2 establishes the problem statement in the conventional parametrisation of Gaussian distributions (which uses mean vectors and covariance matrices): marginalising, conditioning, and evaluating the log-density functions of linearly related Gaussian variables. Sections 4.3 to 4.6 discuss implementing all those operations using (generalised) Cholesky factors of covariance matrices. At this point, the techniques will be loosely connected to probabilistic numerical solvers, but the content of this chapter does not depend on the previous chapters and can be read independently of the rest of this thesis. Section 4.7 discusses related literature, and Section 4.8 connects the results to those in previous and upcoming chapters.

4.2 Manipulation of Gaussian variables

Let $d_{\text{in}}, d_{\text{out}} \in \mathbb{N}$. Let $m_{\text{in}} \in \mathbb{R}^{d_{\text{in}}}$, $C_{\text{in}} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$, $A_{\text{cond}} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, $b_{\text{cond}} \in \mathbb{R}^{d_{\text{out}}}$, and $C_{\text{cond}} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{out}}}$ be some vectors and matrices. Assume that C_{in} and C_{cond} are symmetric and positive semidefinite and that $A_{\text{cond}} C_{\text{in}} A_{\text{cond}}{}^\top + C_{\text{cond}}$ is invertible.

Define correlated random variables X and Y,

$$p(X) := \mathcal{N}(m_{\text{in}}, C_{\text{in}}), \quad p(Y \mid X) := \mathcal{N}(A_{\text{cond}} X + b_{\text{cond}}, C_{\text{cond}}). \tag{4.1}$$

All of the following statements about marginals and conditionals are well-known [e.g. 140, Appendix A.2]. The joint distribution of X and Y is Gaussian,

$$p(X, Y) = \mathcal{N}(m_{\text{joint}}, C_{\text{joint}}), \tag{4.2}$$

with mean and covariance

$$m_{\text{joint}} := \begin{pmatrix} m_{\text{in}} \\ A_{\text{cond}} m_{\text{in}} + b_{\text{cond}} \end{pmatrix}, \tag{4.3a}$$

$$C_{\text{joint}} := \begin{pmatrix} C_{\text{in}} & C_{\text{in}} A_{\text{cond}}{}^\top \\ A_{\text{cond}} C_{\text{in}} & A_{\text{cond}} C_{\text{in}} A_{\text{cond}}{}^\top + C_{\text{cond}} \end{pmatrix}. \tag{4.3b}$$

The marginal distribution of Y is

$$p(Y) = \int p(Y, X) \, \mathrm{d}X = \mathcal{N}(m_{\text{out}}, C_{\text{out}}), \tag{4.4}$$

with parameters

$$m_{\text{out}} := A_{\text{cond}} m_{\text{in}} + b_{\text{cond}}, \tag{4.5a}$$

$$C_{\text{out}} := A_{\text{cond}} C_{\text{in}} A_{\text{cond}}{}^\top + C_{\text{cond}}. \tag{4.5b}$$

The conditional distribution is

$$p(X \mid Y) = \mathcal{N}(A_{\text{rev}} Y + b_{\text{rev}}, C_{\text{rev}}) \tag{4.6}$$

with system matrices

$$A_{\mathrm{rev}} := C_{\mathrm{in}} A_{\mathrm{cond}}{}^{\top} (C_{\mathrm{out}})^{-1}, \tag{4.7a}$$

$$b_{\mathrm{rev}} := m_{\mathrm{in}} - A_{\mathrm{rev}} (A_{\mathrm{cond}} m_{\mathrm{in}} + b_{\mathrm{cond}}), \tag{4.7b}$$

$$C_{\mathrm{rev}} := C_{\mathrm{in}} - A_{\mathrm{rev}} C_{\mathrm{out}} A_{\mathrm{rev}}{}^{\top}. \tag{4.7c}$$

The covariance matrix C_{rev} of the conditional distribution is the *Schur complement* [67, p. 103] of C_{out} in the covariance matrix C_{joint} of the joint distribution. The conditional distribution $p(Y \mid X)$ implies a parametrisation of the posterior distribution

$$p(X \mid Y = y) = \mathcal{N}(A_{\mathrm{rev}} y + b_{\mathrm{rev}}, C_{\mathrm{rev}}) \tag{4.8}$$

for a given realisation $y \in \mathbb{R}^{d_{\mathrm{out}}}$.

Finite-precision arithmetic sometimes causes issues when implementing Equations (4.4) to (4.7) as given:

- ◇ While C_{in} and C_{cond} are symmetric and positive semidefinite, C_{out} and C_{rev} are sometimes not, due to round-off errors. Such a lack of symmetry or positive definiteness is problematic; e.g., when attempting a Cholesky decomposition of C_{rev}, which fails when C_{rev} has a negative eigenvalue or lacks symmetry.
- ◇ Roughly speaking, the elements in a covariance matrix of a Gaussian random variable grow as fast as the square of the realisations of the variable. To sample such a realisation with finite precision ϵ, a covariance matrix with entries that grow as ϵ^2 needs to be stored. The quadratic precision requirement complicates the manipulation of Gaussian distributions in low-precision arithmetic.

Both issues sometimes lead to catastrophic results when implementing probabilistic numerical initial value problem solvers because small integration steps and ill-conditioned covariance matrices frequently occur (more on this in Chapter 7). As a solution, we implement Equations (4.4) to (4.7) using only (generalised) Cholesky factors of the covariance matrices, which are defined as follows.

4.3 Cholesky parametrisation

This thesis avoids the ambiguous notion of a "square root" of a matrix, which sometimes means $P = LL$ [e.g. 67], and sometimes $P = LL^{\top}$ [e.g. 143]. Instead, we use the terminology of a "generalised Cholesky factor" by Grewal and Andrews [68].

Definition 4.1 ([68])**.** Let P be a symmetric, positive semidefinite matrix. We call every (not necessarily square) matrix L that satisfies $P = LL^{\top}$ a *generalised*

Cholesky factor of P and write $(P)^{1/2} := L$.

Every symmetric, positive semidefinite matrix admits a generalised Cholesky factor; for example, we can always eigendecompose $P = UDU^\top$ and choose $(P)^{1/2} := U\sqrt{D}$. Here, D and $\sqrt{D}$ are diagonal, and the latter contains the square roots of the elements of the former. However, a generalised Cholesky factor as defined in Definition 4.1 is not unique [e.g., 68, Eqs. 1.16f.]. If P were strictly positive definite and if we required that L were square and had a positive diagonal, L were the Cholesky factor of P, which is unique. However, we do not restrict ourselves to positive definite matrices. In fact, we benefit from a lack of uniqueness in Definition 4.1 because we actively transform different generalised Cholesky factors into one another to achieve optimal efficiency while remaining robust against round-off errors.

We call any parametrisation of a Gaussian distribution that stores a generalised Cholesky factor instead of a full covariance matrix a *Cholesky parametrisation*. Mean vectors are stored as usual. We write *Cholesky arithmetic* when we manipulate probability distributions in Cholesky parametrisation without ever forming full covariance matrices.

The QR-decomposition [e.g. 67, Chapter 5.2] factorises any matrix A into the product of an orthogonal matrix Q and an upper-triangular matrix R, $A = QR$. As part of the algorithm, either Q or R is forced to be square; we always choose R (which Golub and Van Loan [67] refer to as the *thin QR factorisation*). Such a "thin" QR decomposition of a full-column-rank matrix is unique if we require that R has a positive diagonal. In this case, R becomes the Cholesky factor of $A^\top A$ (which is useful for the arguments below) [67, Theorem 5.2.2]. The factorisation of a $k \times l$ matrix costs $O(kl^2)$ floating-point operations for $k \geq l$ and $O(k^2 l)$ operations for $k \leq l$. Precise complexities depend on whether the underlying orthogonalisation relies on Householder transformations, Givens rotations, or something else. Details are in the book by Golub and Van Loan [67, Chapter 5.2].

Let $(C_{\text{in}})^{1/2}$ and $(C_{\text{cond}})^{1/2}$ be generalised Cholesky factors of C_{in} and C_{cond} respectively. Generalised Cholesky factors $(C_{\text{cond}})^{1/2}$ and $(C_{\text{rev}})^{1/2}$ of the covariance matrices of the conditional distribution $p(X \mid Y)$ and of the marginal distribution $p(Y)$ can be computed in Cholesky arithmetic:

Algorithm 4.2. Given $p(X)$ and $p(Y \mid X)$ as above and in Cholesky parametrisation, the Cholesky parametrisation of $p(Y) = \mathcal{N}(m_{\text{out}}, C_{\text{out}})$ as well as $p(X \mid Y) = \mathcal{N}(A_{\text{rev}}Y + b_{\text{rev}}, C_{\text{rev}})$ (Equations (4.4) to (4.7)) arise as follows:

1. QR-decompose the $(d_{\text{out}} + d_{\text{in}}) \times (d_{\text{out}} + d_{\text{in}})$ matrix

$$\begin{pmatrix} \left[(C_{\text{cond}})^{1/2}\right]^\top & 0 \\ \left[(C_{\text{in}})^{1/2}\right]^\top (A_{\text{cond}})^\top & \left[(C_{\text{in}})^{1/2}\right]^\top \end{pmatrix} = Q \begin{pmatrix} R_1 & R_2 \\ 0 & R_3 \end{pmatrix}, \tag{4.9}$$

   and discard Q.

2. Extract

$$(C_{\text{rev}})^{1/2} := R_3^\top \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}, \tag{4.10a}$$
$$(C_{\text{out}})^{1/2} := R_1^\top \in \mathbb{R}^{d_{\text{out}} \times d_{\text{out}}}, \tag{4.10b}$$
$$A_{\text{rev}} := (R_1^{-1} R_2)^\top \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}. \tag{4.10c}$$

3. Compute b_{rev} and m_{out} as in Equations (4.5) and (4.7).

Return A_{rev}, b_{rev}, $(C_{\text{rev}})^{1/2}$, m_{out}, and $(C_{\text{out}})^{1/2}$.

Chapter 7 applies Algorithm 4.2 to the extrapolation steps in probabilistic numerical initial value problem solvers. The output of Algorithm 4.2 is a Cholesky parametrisation of the conditionals:

Proposition 4.3. *Under the above assumptions on $p(X)$ and $p(Y \mid X)$, Algorithm 4.2 computes the Cholesky parametrisation of $p(Y)$ and $p(X \mid Y)$ correctly, in Cholesky arithmetic, and in complexity $O(d_{in}{}^3 + 4 d_{in} d_{out}{}^2 + 3 d_{in}{}^2 d_{out} + d_{out}{}^3)$.*

Proof. Multiply both sides of Equation (4.9) with their transposes from the left,

$$\begin{pmatrix} C_{\text{out}} & A_{\text{cond}} C_{\text{in}} \\ C_{\text{in}} A_{\text{cond}}{}^\top & C_{\text{in}} \end{pmatrix} = \begin{pmatrix} R_1^\top R_1 & R_1^\top R_2 \\ R_2^\top R_1 & R_2^\top R_2 + R_3^\top R_3 \end{pmatrix}. \tag{4.11}$$

Match the blocks of the matrices and deduce $(C_{\text{out}})^{1/2} = R_1^\top \in \mathbb{R}^{d_{\text{out}} \times d_{\text{out}}}$, as well as

$$R_3^\top R_3 = C_{\text{in}} - R_2^\top R_2 \tag{4.12a}$$
$$= C_{\text{in}} - (R_2^\top R_1^{-\top})(R_1^\top R_1)(R_1^{-1} R_2) \tag{4.12b}$$
$$= C_{\text{in}} - A_{\text{rev}} C_{\text{out}} A_{\text{rev}}{}^\top, \tag{4.12c}$$

which yields $(C_{\text{rev}})^{1/2} = R_3^\top \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$ and $A_{\text{rev}} = R_2^\top R_1^{-\top} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$.

The complexity of Algorithm 4.2 depends on two parts: (i) the complexity of a QR decomposition of a $(d_{\text{in}} + d_{\text{out}}) \times (d_{\text{in}} + d_{\text{out}})$ matrix, which is

$$O((d_{\text{in}} + d_{\text{out}})^3) = O(d_{\text{in}}{}^3 + 3 d_{\text{in}} d_{\text{out}}{}^2 + 3 d_{\text{in}}{}^2 d_{\text{out}} + d_{\text{out}}{}^3); \tag{4.13}$$

and (ii) the complexity of computing $R_1^{-1}R_2$, which involves d_{in} backwards substitutions using a d_{out}-dimensional, upper-triangular matrix, resulting in $O(d_{\text{in}}{d_{\text{out}}}^2)$. Adding both complexities yields the claimed computational cost. □

The precise complexity of Algorithm 4.2 is inherited from that of the QR decomposition, which depends on whether the decomposition uses Householder transformations or Given rotations, for example.

Both Algorithm 4.2 and an implementation via Equations (4.4) to (4.7) have total complexity $O({d_{\text{in}}}^3 + {d_{\text{out}}}^3)$. However, QR decompositions are more expensive than matrix multiplications, which makes Algorithm 4.2 (and its descendants presented next) more expensive than manipulating the conventional parametrisation. Still, the gain in numerical stability (more than) compensates for this in settings where numerical stability is crucial, as will be demonstrated in the upcoming chapters.

Remark 4.4. Algorithm 4.2 and Proposition 4.3 remain valid when we replace $A_{\text{cond}}(C_{\text{in}})^{1/2}$ with any generalised Cholesky factor of $A_{\text{cond}}C_{\text{in}}{A_{\text{cond}}}^\top$, as long as the cross-covariance identity

$$(A_{\text{cond}}C_{\text{in}}{A_{\text{cond}}}^\top)^{1/2}\left[(C_{\text{in}})^{1/2}\right]^\top = E[(Y - E[Y])(X - E[X])] \tag{4.14}$$

is preserved (E is the expected value). $A_{\text{cond}}(C_{\text{in}})^{1/2}$ and $(C_{\text{in}})^{1/2}$ may be the most obvious choice. However, others are possible, too, and sometimes even required (Chapter 5).

This remark also applies to Algorithm 4.5 and Algorithm 4.7 below.

4.4 Deterministic transformations

For deterministic transformations ($C_{\text{cond}} = 0$), the arguments from the previous section apply with $C_{\text{cond}} = 0$. Nevertheless, this setting allows a more efficient implementation:

Assume $d_{\text{in}} \geq d_{\text{out}}$ and suppose that $A_{\text{cond}}C_{\text{in}}{A_{\text{cond}}}^\top$ is invertible because, without those two assumptions, the transition matrix A_{rev} is not well-defined.

Algorithm 4.5. For $p(X) = N(m_{\text{in}}, C_{\text{in}})$ as above and in Cholesky parametrisation, and for $Y := A_{\text{cond}}X + b_{\text{cond}}$, compute the Cholesky parametrisation of the marginal $p(Y)$ and the conditional $p(X \mid Y)$ as follows:

1. Decompose $(A_{\text{cond}}(C_{\text{in}})^{1/2})^\top = QR$, set $(C_{\text{out}})^{1/2} := R^\top$, and discard Q.

2. Compute $A_{\text{rev}} := C_{\text{in}}{A_{\text{cond}}}^\top R^{-1}R^{-\top}$.

3. Compute $(C_{\text{rev}})^{1/2} := (C_{\text{in}})^{1/2} - A_{\text{rev}}A_{\text{cond}}(C_{\text{in}})^{1/2}$

4. Compute b_{rev} and m_{out} as in Equations (4.5) and (4.7)

Return A_{rev}, b_{rev}, $(C_{\mathrm{rev}})^{1/2}$, m_{out}, and $(C_{\mathrm{out}})^{1/2}$.

Chapter 7 shows how Algorithm 4.5 applies to the correction steps in probabilistic numerical initial value problem solvers.

Proposition 4.6. *Under the above assumptions on $p(X)$ and $p(Y \mid X)$, Algorithm 4.5 computes the Cholesky parametrisation of $p(Y)$ and $p(X \mid Y)$ correctly, in Cholesky arithmetic, and in complexity $O(4d_{in}{d_{out}}^2 + d_{out}{d_{in}}^2)$.*

Proof. R is a triangular, generalised Cholesky factor of the product $A_{\mathrm{cond}}C_{\mathrm{in}}{A_{\mathrm{cond}}}^\top$, and its QR decomposition is available in $O(d_{\mathrm{in}}{d_{\mathrm{out}}}^2)$ (recall $d_{\mathrm{in}} \geq d_{\mathrm{out}}$).

The matrix A_{rev} is the same as in Equation (4.7) and requires d_{in} forward and d_{in} backward substitutions with the triangular, d_{out}-dimensional R (respectively its transpose) at the total cost of $O(2d_{\mathrm{in}}{d_{\mathrm{out}}}^2)$.

The covariance matrix of the conditional follows from

$$\begin{aligned} C_{\mathrm{rev}} &= C_{\mathrm{in}} - A_{\mathrm{rev}}C_{\mathrm{out}}{A_{\mathrm{rev}}}^\top && \text{(4.15a)} \\ &= (I - A_{\mathrm{rev}}A_{\mathrm{cond}})C_{\mathrm{in}}(I - A_{\mathrm{rev}}A_{\mathrm{cond}})^\top, && \text{(4.15b)} \end{aligned}$$

known as the *Joseph update* [15], and requires two matrix multiplications: $(C_{\mathrm{in}})^{1/2} \mapsto A_{\mathrm{cond}}(C_{\mathrm{in}})^{1/2}$ and $A_{\mathrm{cond}}(C_{\mathrm{in}})^{1/2} \mapsto A_{\mathrm{rev}}A_{\mathrm{cond}}(C_{\mathrm{in}})^{1/2}$. The total cost of those two multiplications is $O(d_{\mathrm{in}}{d_{\mathrm{out}}}^2 + d_{\mathrm{out}}{d_{\mathrm{in}}}^2)$. □

For deterministic transformations, Algorithm 4.5 is more efficient than Algorithm 4.2 by (loosely speaking) a factor $O({d_{\mathrm{in}}}^3 + {d_{\mathrm{out}}}^3)$. However, it has the potential downside that it yields a non-triangular generalised Cholesky factor $(C_{\mathrm{rev}})^{1/2}$. If required, $(C_{\mathrm{rev}})^{1/2}$ can be triangularised with another QR decomposition in complexity $O({d_{\mathrm{in}}}^3)$. For probabilistic numerical initial value problem solvers, non-triangular generalised Cholesky factors are acceptable.

4.5 Marginalisation

Compute the marginal $p(Y)$ without the conditional distribution using Algorithm 4.7.

Algorithm 4.7. For $p(X) = \mathcal{N}(m_{\mathrm{in}}, C_{\mathrm{in}})$, the marginal $p(Y) = \mathcal{N}(m_{\mathrm{out}}, C_{\mathrm{out}})$ arises from one of the following two procedures:

If the transformation is Gaussian (Algorithm 4.2), that is,

$$p(Y \mid X) = N(A_{\mathrm{cond}}X + b_{\mathrm{cond}}, C_{\mathrm{cond}}), \tag{4.16}$$

QR-decompose

$$\begin{pmatrix} A_{\text{cond}}(C_{\text{in}})^{1/2} & (C_{\text{cond}})^{1/2} \end{pmatrix}^\top = QR, \tag{4.17}$$

discard Q, and set $(C_{\text{out}})^{1/2} := R^\top$.

If the transformation is deterministic (Algorithm 4.5), that is,

$$Y := A_{\text{cond}}X + b_{\text{cond}}, \tag{4.18}$$

QR-decompose

$$(A_{\text{cond}}(C_{\text{in}})^{1/2})^\top = QR, \tag{4.19}$$

discard Q, and set $(C_{\text{out}})^{1/2} := R^\top$.

The correctness of Algorithm 4.7 stems from

$$\begin{aligned} C_{\text{out}} &= \begin{pmatrix} A_{\text{cond}}(C_{\text{in}})^{1/2} & (C_{\text{cond}})^{1/2} \end{pmatrix} \begin{pmatrix} \left[(C_{\text{in}})^{1/2}\right]^\top {A_{\text{cond}}}^\top \\ \left[(C_{\text{cond}})^{1/2}\right]^\top \end{pmatrix} && \text{(4.20a)} \\ &= R^\top Q^\top Q R && \text{(4.20b)} \\ &= R^\top R && \text{(4.20c)} \end{aligned}$$

with the natural modifications for $C_{\text{cond}} = 0$. The QR decomposition costs $O(2d_{\text{in}}{d_{\text{out}}}^2)$ for Gaussian transformations and $O(d_{\text{in}}{d_{\text{out}}}^2)$ for deterministic transformations. Chapter 7 applies Algorithm 4.7 to the computation of marginals of the posterior distribution.

4.6 Whitening and log-probabilitities

The availability of (triangular) generalised Cholesky factors of matrices improves the numerical stability of whitening and the evaluation of log-probability-density functions. In the remainder of this section, assume that all generalised Cholesky factors are triangular. If not, QR-decompose $(C)^{1/2} = QR$ and replace the non-triangular factor with a triangular factor $(C)^{1/2} = R^\top$. Assume that $(C)^{1/2}$ or R, respectively, has a positive diagonal (which is no loss of generality; recall the properties of QR decompositions from Section 4.3).

Define the Euclidean norm of a vector as $\|x\| = x^\top x$. Let $\|x\|_A = \|((A)^{1/2})^\top x\|$ be the Mahalanobis norm of x induced by a symmetric, positive semidefinite matrix [40].

For a random variable $X \sim p(X) = N(m_{\text{in}}, C_{\text{in}})$, the log-probability of $X = x$ is

$$-2 \log p(X = x) = \|m_{\text{in}} - x\|^2_{(C_{\text{in}})^{-1}} + \log \det(C_{\text{in}}) + \log 2\pi. \tag{4.21}$$

Evaluating this expression involves *whitening* the residual $(m_{\text{in}} - x)$, which is defined as computing [e.g., 94]

$$\left(\left[(C_{\text{in}})^{1/2}\right]^\top\right)^{-1}(m_{\text{in}} - x). \tag{4.22}$$

Implementing the Mahalanobis norm $\|m_{\text{in}} - x\|_{(C_{\text{in}})^{-1}}$ as the Euclidean norm of the whitened residual (Equation (4.22)) ensures that the Mahalanobis norm is always nonnegative even in low-precision arithmetic. If $(C_{\text{in}})^{1/2}$ is triangular, computing the norm of the residual in Equation (4.22) is more efficient and more robust against round-off errors than evaluating $(m_{\text{in}}-x)^\top {C_{\text{in}}}^{-1}(m_{\text{in}}-x)$. The reason is that whitening reduces to a single backward substitution with the triangular $\left[(C_{\text{in}})^{1/2}\right]^\top$ and that the Euclidean norm of any vector is straightforward to compute.

Computing the determinant of C_{in} conventionally (for example, via an LU decomposition [67, Theorem 3.2.1]) sometimes leads to a negative determinant if the true determinant is almost zero and if the implementation suffers from round-off errors. Negative determinants do not admit a logarithm, in which case the computation would fail unexpectedly. Square-root arithmetic avoids this problem:

The determinant of C_{in} is the square of the determinant of its generalised Cholesky factor $(C_{\text{in}})^{1/2}$, provided the factor is a square matrix [67, Section 2.1.6]. If $(C_{\text{in}})^{1/2}$ is triangular, its determinant is the product of its diagonal elements [67, Theorem 3.2.1]. As a result, the logarithm of the determinant of C_{in} is always well-defined: if $(C_{\text{in}})^{1/2}$ is triangular with a positive diagonal,

$$\log\det C_{\text{in}} = 2\log\det\left[(C_{\text{in}})^{1/2}\right] \tag{4.23}$$

holds. Since the generalised Cholesky factor is triangular with a positive diagonal, the quantity $\log\det[(C_{\text{in}})^{1/2}]$ equals the sum of the logarithm of the diagonal elements of $(C_{\text{in}})^{1/2}$. Therefore, the log-determinant of a covariance matrix is always well-defined (even in finite-precision arithmetic and for determinants close to zero).

The same arguments apply to the log probabilities of Y, $Y \mid X$, and every other Gaussian random variable in Cholesky parametrisation.

4.7 Related literature

The content of this chapter is mostly known – its application to probabilistic numerical initial value problem solvers (for example in Chapter 7) is a contribution of this thesis. The formulas for the marginal and conditional distributions of Gaussian variables in conventional implementation (Section 4.2) are widespread knowledge [e.g., 143, Sections A.1f]. The Cholesky implementation of marginalisation and conditioning via a single QR decomposition as in Section 4.3 has been described by Gibson and Ninness [65, Eq. 48]. To the best of the author's knowledge, the simplifications for deterministic transformations in Section 4.4 as well as the content of Remark 4.4 are not available in previous work (except for the papers described in this thesis). Remark 4.4 leads to a novel implementation of statistical linear regression in Cholesky arithmetic, as will be shown in Chapter 5.

The marginalisation via a single QR decomposition in Section 4.5 is well-documented in the literature on square-root filtering [e.g. 68, Section 7.4.7.2]. It employs the same technique as *low-rank updates* and *low-rank downdates* of matrices [e.g., 153]. Conditioning a Gaussian variable allows implementation with low-rank downdates (for example, like in the literature on square-root sigma-point filters [e.g. 168, 185]). However, while low-rank updates are numerically stable, low-rank downdates can sometimes be numerically unstable [153]. The algorithms in this section condition Gaussian variables without downdates, thus bypassing such instability concerns.

4.8 Conclusion

In summary, while manipulating Gaussian random variables is sometimes sensitive to round-off errors, it gains numerical stability from using Cholesky arithmetic. Precise algorithms for the Cholesky implementation of marginalisation and conditioning have been provided under the assumption of a linear transformation of a Gaussian random variable. Chapter 5 describes the manipulation of nonlinearly related variables. Chapter 7 (and beyond) apply Cholesky arithmetic to the probabilistic numerical solution of differential equations.

Chapter 5

Linearisation

Contents

5.1 Introduction

Following Chapter 4's discussion of Cholesky parametrisation of Gaussian distributions, this chapter continues providing technical background information. More specifically, Chapter 4 assumed an affine relation between Gaussian variables, which is not always the case – and if two (or more) Gaussian variables are nonlinearly related (like in most initial-value-problem-solver contexts), marginalisation and conditioning are no longer possible in closed form.

The present chapter discusses linearisation. Linearisation enables the approximate manipulation of nonlinearly related Gaussian variables. The essential idea behind such an approximate manipulation is to linearise all nonlinearities, for instance, with a first-order Taylor approximation, and to apply the techniques from Chapter 4 afterwards. The present chapter discusses details by featuring the most common modes of linearisation and explaining how to combine them with Cholesky arithmetic. Most, but not all of the statements below are known. Among other things, this chapter introduces a new perspective on existing probabilistic numerical initial and boundary value problem solvers through the lens of (iterated) zeroth-order statistical linearisation and describes a novel Cholesky implementation of statistical linear regression.

The remainder of this chapter evolves as follows. Section 5.2 formally defines the problem setting of manipulating nonlinearly related Gaussian variables. Sections 5.3 and 5.4 describe the two main approaches: Taylor and statistical linearisation, respectively. The implementation of the latter, commonly referred to as statistical linear regression (Section 5.6), requires a prior discussion of cubature (Section 5.5). Section 5.7 explains iteration and Section 5.8 discusses how to apply linearisation techniques to time-series problems: both use-cases include probabilistic numerical solvers for initial and boundary value problems.

Most of the present chapter does not assume knowledge of any of the prior chapters and could be read independently from the rest of this thesis. However, Sections 5.5 and 5.6 will be easier to understand in the context of Chapter 4.

5.2 Problem setting

Let $m_{\text{in}} \in \mathbb{R}^{d_{\text{in}}}$ be a given vector and $C_{\text{in}} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$ be a given, square, symmetric, positive semidefinite matrix (like in Chapter 4). Let

$$h : \mathbb{R}^{d_{\text{in}}} \longrightarrow \mathbb{R}^{d_{\text{out}}} \tag{5.1}$$

be a known, differentiable function with Jacobian Dh. Assume an input distribution $p(X) = \mathcal{N}(m_{\text{in}}, C_{\text{in}})$ and a conditional distribution $p(Y \mid X) = \delta(h(X))$, where δ is the Dirac delta. If h were affine, $p(Y)$ and $p(X, Y)$ were Gaussian, and we could apply the algorithms from Chapter 4. In the present chapter, h is an arbitrary function, and the methods from Chapter 4 are not available immediately.

While $p(Y)$ and $p(X, Y)$ are not guaranteed to be Gaussian, estimation algorithms simplify if we approximate them as Gaussian. For instance, a Gaussian approximation of $p(Y, X)$ could result from assuming an affine transformation

$$p(Y \mid X) \approx \mathcal{N}(A_{\text{cond}} X + b_{\text{cond}}, C_{\text{cond}}) \tag{5.2}$$

for some A_{cond}, b_{cond}, C_{cond}. The remainder of the chapter takes this approach.

The following exposition only considers deterministic, fully nonlinear transformations, that is, $Y = h(X)$. But of course, the same techniques may be applied to semilinear transformations $p(Y \mid X) = \delta[MX + h(X)]$ or stochastic conditionals $p(Y \mid X) = \mathcal{N}(h(X), M)$, where M is some matrix. To transfer the techniques, one would consider the transformations as the composition of a deterministic, nonlinear conditional and an affine/stochastic one and linearise only the nonlinearity.

The affine transformation implies that we can manipulate the distributions of X and Y with the algorithms from Chapter 4. There are two options for choosing the affine transformation: Taylor linearisation and statistical linear regression (respectively statistical linearisation). Both are discussed in this chapter.

5.3 Taylor linearisation

Let z be a random variable with distribution $p(z) = \mathcal{N}(\xi, \Xi)$ that depends on known parameters. For now, only ξ matters, but we need the full distribution $p(z)$ later.

The first-order Taylor approximation of h around ξ is

$$h(x) \approx h(\xi) + Dh(\xi)(x - \xi). \tag{5.3}$$

Such a first-order Taylor approximation implies the random-variable transformation

$$p(Y \mid X) = \delta[h(X)] \approx \delta[h(\xi) + Dh(\xi)(X - \xi)] \tag{5.4}$$

which is affine in X. More specifically, a first-order Taylor approximation induces a parametrisation of Equation (5.2) as

$$A_{\text{cond}} = Dh(\xi), \quad b_{\text{cond}} = h(\xi) - A_{\text{cond}}\xi, \quad C_{\text{cond}} = 0. \tag{5.5}$$

We summarise first-order Taylor linearisation in the following algorithm.

Algorithm 5.1 (Taylor linearisation, first-order)**.** Assume that a $\xi \in \mathbb{R}^{d_{\text{in}}}$ and a nonlinear, differentiable function $h : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ are given. Approximate $p(Y \mid X)$ as follows:

1. Evaluate $h(\xi)$ and $Dh(\xi)$.
2. Assemble A_{cond} and b_{cond} according to Equation (5.5) (C_{cond} is zero)

Return A_{cond} and b_{cond}.

To evaluate the function h and its Jacobian, we usually rely on automatic differentiation [69] or a suitable finite difference formula [161]. A common choice of a linearisation point is the mean of the input variable $\xi = m_{\text{in}}$. Section 5.7 extends the discussion of linearisation points.

Instead of a first-order Taylor approximation, zeroth-order approximations are possible, too. Consider the zeroth-order approximation of h around ξ,

$$h(x) \approx h(\xi). \tag{5.6}$$

The approximating function is constant (ξ is fixed and known). It implements Equation (5.2) with parameters

$$A_{\text{cond}} = 0, \quad b_{\text{cond}} = h(\xi), \quad C_{\text{cond}} = 0. \tag{5.7}$$

Algorithm 5.2 emerges:

Algorithm 5.2 (Taylor linearisation, zeroth-order)**.** Assume that ξ and a nonlinear function h are known. Approximate $p(Y \mid X)$ as follows:

1. Evaluate $h(\xi)$
2. Assemble A_{cond} and b_{cond} according to Equation (5.7) (C_{cond} is zero).

Return A_{cond} and b_{cond}.

Again, a common choice of a linearisation point is $\xi = m_{\text{in}}$. Zeroth-order linearisation is helpful for semilinear problems (which include simulating differential equations).

5.4 Statistical linearisation

Let $E_X[X]$ be the expected value of random variable X under the law of X. Define the covariance $C_X[X, X] := E_X[(X - E_X[X])(X - E_X[X])^\top]$. Recall the linearisation point $p(z) = \mathcal{N}(\xi, \Xi)$.

Statistical linearisation derives A_{cond}, b_{cond}, and C_{cond} using the full distribution over the linearisation-point $p(z) = \mathcal{N}(\xi, \Xi)$ instead of linearising around a single vector. One advantage of this approach is that the spread of the distribution over z is taken into account: the linearisation adapts to “uncertainty” over z.

Statistical linearisation goes as follows: Choose A_{cond} and b_{cond} with the lowest mean-square error and C_{cond} as the reconstruction error covariance

$$(A_{\text{cond}}, b_{\text{cond}}) = \arg\min_{A,b} E_z[(Az + b - h(z))^\top (AX + b - h(X))], \tag{5.8a}$$

$$C_{\text{cond}} = E_z[(Az + b - h(z))(AX + b - h(z))^\top]. \tag{5.8b}$$

All expected values are with respect to the law of z. This objective function implies that the affine approximation of the nonlinear transformation h is the best possible affine reconstruction under the given assumptions on z.

The solution to the minimisation problem in Equation (5.8) can be computed in closed form: First-order optimality conditions imply [e.g., 8]

$$A_{\text{cond}} = C_z[h(z), z]C[z, z]^{-1}, \tag{5.9a}$$

$$b, = E_z[h(z)] - A_{\text{cond}}E_z[z], \tag{5.9b}$$

$$C_{\text{cond}} = C_z[h(z), h(z)] - A_{\text{cond}}C_z[z, z]{A_{\text{cond}}}^\top. \tag{5.9c}$$

Again, all expected values and covariance operators are with respect to the law of z. In some sense, this linearisation can be regarded as a generalisation of Taylor linearisation: The identity [e.g. 143, p. 77]

$$C_z[h(z), z] = E_z[Dh(z)]C_z[z, z] \tag{5.10}$$

implies that the formula in Equation (5.9) equals

$$A_{\mathrm{cond}} = E_z[Dh(z)], \tag{5.11a}$$

$$b = E_z[h(z)] - A_{\mathrm{cond}}E_z[z] \tag{5.11b}$$

$$C_{\mathrm{cond}} = C_z[h(z), h(z)] - A_{\mathrm{cond}}C_z[z, z]A_{\mathrm{cond}}^\top. \tag{5.11c}$$

The expected values of h and Dh under $p(z)$ (Equation (5.11)) replace their point-evaluations at ξ (Equation (5.5)). Equation (5.11) also shows how the larger the covariance of z is, the more statistical differs from Taylor linearisation. In the limit of $\|\Xi\| \to 0$, statistical and Taylor linearisation yield the same linearisation matrices. The choice between implementing Equation (5.9) or Equation (5.11) depends on the complexity of evaluating the Jacobian. More on this below.

The expressions above describe first-order statistical linearisation. Section 5.3 distinguished between zeroth-order and first-order Taylor approximations. We can also derive zeroth-order statistical linearisation in the same fashion as the first-order version: Instead of an affine approximation of $p(Y \mid X)$, assume a density

$$p(Y \mid X) = \delta[h(X)] \approx \mathcal{N}(b_{\mathrm{cond}}, C_{\mathrm{cond}}). \tag{5.12}$$

While first-order statistical linearisation was an affine, stochastic transformation of the input variable, zeroth-order statistical linearisation (Equation (5.12)) is constant; that is, it does not depend on X.

The minimum mean-square error

$$b_{\mathrm{cond}} = \arg\min_b E_z[(b - h(z))^\top (b - h(z))] \tag{5.13}$$

is attained when choosing

$$b_{\mathrm{cond}} = E_z[h(z)]. \tag{5.14}$$

The error covariance induced by the selection in Equation (5.14) is

$$C_{\mathrm{cond}} = E_z[(b_{\mathrm{cond}} - h(z))(b_{\mathrm{cond}} - h(z))^\top] = C_z[h(z), h(z)]. \tag{5.15}$$

Roughly speaking, the more h varies or, the larger the covariance Ξ of the linearisation point is, the bigger C_{cond} in Equation (5.15) becomes. As for the first-order version, zeroth-order statistical linearisation mirrors zeroth-order Taylor linearisation: The expected values replace the point evaluations of h, and the error covariance is strictly positive instead of constantly zero.

While (to the best of the author's knowledge), the present text is the first to formally discuss zeroth-order statistical linearisation, a special case of the expressions in Equations (5.14) and (5.15) has previously appeared (under a different name):

Remark 5.3. Kersting and Hennig [91] implement zeroth-order statistical linearisation by computing the integrals in Equations (5.14) and (5.15) with Bayesian cubature. To see this, compare Equations (29) and (30) in the paper by Kersting and Hennig [91] to Equations (5.14) and (5.15). See also Proposition 3 in the work by Tronarp et al. [163].

Zeroth-order approximations are less accurate than first-order approximations, since they are a form of first-order approximations; the optimality of the first-order approximation dictates that it must be at least as accurate as the zeroth-order model. Perhaps unsurprisingly, the respective error covariance matrices express this relationship: The error covariance of the zeroth-order linearisation dominates that of the first-order approximation.

If h is affine, the first-order linearisations yield the correct parameters because expected values can be solved in closed form, and the necessary terms cancel out. The zeroth-order linearisations do not; they compute the correct parameters only if h is constant (that is, affine with a zero Jacobian).

The integrals in Equations (5.8) to (5.11), Equation (5.14), and Equation (5.15) depend on the distribution of the linearisation point z and can usually not be evaluated in closed form.

5.5 Cubature

Recall $p(z) = \mathcal{N}(\xi, \Xi)$. Since closed-form expressions rarely exist (unless h is affine), we approximate the integrals occurring in statistical linearisation with cubature.

Let $\{x_k, w_k\}_{k=0}^K$ be the nodes and weights of a cubature rule.

Assumption 5.4. *We assume all weights are nonnegative, $w_k \geq 0$, $k = 0, ..., K$. We also assume that the cubature nodes centre around zero, $\sum_{k=0}^K w_k x_k = 0$, and that the weights sum to 1, $\sum_{k=0}^K w_k = 1$.*

Nonnegative cubature weights are a common requirement for numerically stable cubature rules [89]. We require nonnegative weights because we compute the square root of each weight below. Many, but not all, cubature rules satisfy Assumption 5.4. Admissible examples include Gauss–Hermite cubature [e.g. 143, Section 6.3], the third-order spherical cubature rule [7], or the unscented transform [87]. Bayesian cubature weights sometimes satisfy Assumption 5.4 [details are in 89].

Approximate the integrals in Equations (5.9) to (5.15) numerically as follows. Abbreviate

$$h_k := h((\Xi)^{1/2} x_k + \xi) \tag{5.16}$$

and define the stacks $\mathbf{X} \in \mathbb{R}^{d_{\text{in}} \times (K+1)}$, $\mathbf{w} \in \mathbb{R}^{K+1}$, and $\mathbf{H} \in \mathbb{R}^{d_{\text{out}} \times (K+1)}$ via

$$\mathbf{X} := \begin{pmatrix} \sqrt{w_0}((\Xi)^{1/2} x_0 + \xi) & \cdots & \sqrt{w_K}((\Xi)^{1/2} x_K + \xi) \end{pmatrix}, \tag{5.17a}$$

$$\mathbf{w} := \begin{pmatrix} \sqrt{w_0} & \cdots & \sqrt{w_K} \end{pmatrix}, \tag{5.17b}$$

$$\mathbf{H} := \begin{pmatrix} \sqrt{w_0} h_0 & \cdots & \sqrt{w_K} h_K \end{pmatrix}. \tag{5.17c}$$

Equation (5.17) is only well-defined because we assumed that the cubature weights are positive (Assumption 5.4). In violation of Assumption 5.4, approximate statistical linearisation (via cubature) still works [e.g. 143, Chapter 6], but a Cholesky implementation is more complicated (Remark 5.9).

With Equation (5.17), cubature reduces to matrix-vector and matrix-matrix arithmetic with **X**, **w**, and **H**,

$$E_z[z] = \mathbf{w}\mathbf{X}^\top \ (= \xi), \tag{5.18a}$$

$$E_z[h(z)] \approx \mathbf{w}\mathbf{H}^\top = \sum_{k=0}^{K} w_k h_k. \tag{5.18b}$$

The identity $\mathbf{w}\mathbf{X}^\top = \xi$ holds because the cubature nodes centre around zero and because the cubature weights sum to 1 (Assumption 5.4). The approximation error in the other term depends on the cubature rule and h. For example, pth order Gauss–Hermite rules are exact for polynomials up to order $2p-1$ [e.g. 143, Section 6.3].

Subtracting the expected values of z and $h(z)$ removes the bias from **X** and **H**,

$$\mathbf{X}_{\text{centre}} := (\Xi)^{1/2} \begin{pmatrix} \sqrt{w_0} x_0 & \cdots & \sqrt{w_K} x_K \end{pmatrix} \in \mathbb{R}^{d_{\text{in}} \times (K+1)}, \tag{5.19a}$$

$$\mathbf{H}_{\text{centre}} := \begin{pmatrix} (\sqrt{w_0} h_0 - \mathbf{w}\mathbf{H}^\top) & \cdots & (\sqrt{w_K} h_K - \mathbf{w}\mathbf{H}^\top) \end{pmatrix} \in \mathbb{R}^{d_{\text{out}} \times (K+1)}. \tag{5.19b}$$

As a consequence, the (cross-)covariance terms emerge approximately as

$$C_z[z, z] = \mathbf{X}_{\text{centre}} \mathbf{X}_{\text{centre}}^\top \ (= \Xi), \tag{5.20a}$$

$$C_z[h(z), h(z)] \approx \mathbf{H}_{\text{centre}} \mathbf{H}_{\text{centre}}^\top, \tag{5.20b}$$

$$C_z[h(z), z] \approx \mathbf{H}_{\text{centre}} \mathbf{X}_{\text{centre}}^\top. \tag{5.20c}$$

Again, the approximation error depends on the cubature rule. The correspondence between $\mathbf{X}_{\text{centre}}$ and Ξ, that is, $\mathbf{X}_{\text{centre}} \mathbf{X}_{\text{centre}}^\top = \Xi$, holds because the cubature rule nodes centre around zero (Assumption 5.4). Equation (5.20) shows how $\mathbf{H}_{\text{centre}}$ is a generalised Cholesky factor of $C_z[h(z), h(z)]$ by construction.

The formulas above enable approximating statistical linearisation in conventional parametrisation, that is, without relying exclusively on Cholesky factors: plug Equations (5.18) and (5.20) into Equation (5.9), Equation (5.11), or Equations (5.14) to (5.15). This technique of computing the linearisation parameters with cubature is called *statistical linear regression*. But conventional parametrisation of Gaussian variables is not enough: Section 5.6 describes a cubature-based implementation of statistical linearisation in Cholesky arithmetic.

Remark 5.5. We could have derived statistical linear regression differently: Let $\|x\|_2 := x^\top x$ be the Euclidean norm of some x. The parameters A_{cond} and b_{cond} that minimise the error of reconstructing the function evaluations h_k,

$$A_{\text{cond}}, b_{\text{cond}} = \arg\min_{A,b} \sum_{k=0}^{K} w_k \left\| h_k - A\left[(\Xi)^{1/2} x_k + \xi\right] - b \right\|_2^2 \tag{5.21}$$

are attained by the cubature approximations of Equation (5.9) [167]. The same is true for zeroth-order approximations.

Technically, statistical linear regression is an alternative to statistical linearisation or Taylor linearisation. Practically, however, we may think of statistical linear regression as a discrete (that is, cubature-based) approximation of statistical linearisation.

5.6 Statistical linear regression

The following two algorithms describe the Cholesky implementation of first-order and zeroth-order statistical linear regression. (Their correctness is proven afterwards.)

Algorithm 5.6 (First-order statistical linear regression)**.** Assume that $p(z) = \mathcal{N}(\xi, \Xi)$ and h are given as above. Let $p(z)$ be in Cholesky parametrisation. Assume a cubature rule satisfying Assumption 5.4. Linearise h around $p(z)$ as follows:

1. Evaluate $\{h_k\}_{k=0}^{K}$ according to Equation (5.16).
2. Assemble $\mathbf{w}$, $\mathbf{X}$, and $\mathbf{H}$ according to Equation (5.17).
3. Approximate $E_z[h(z)] \approx \mathbf{w}\mathbf{H}^\top$ according to Equation (5.18).
4. Assemble $\mathbf{X}_{\text{centre}}$ and $\mathbf{H}_{\text{centre}}$ according to Equation (5.19).

5. QR-decompose the $(K + 1) \times (d_{\text{out}} + d_{\text{in}})$-matrix

$$\begin{pmatrix}\mathbf{X}_{\text{centre}}^\top & \mathbf{H}_{\text{centre}}^\top\end{pmatrix} = Q \begin{pmatrix} R_1 & R_2 \\ 0 & R_3 \end{pmatrix}, \tag{5.22}$$

   and discard Q.

6. Assign

$$(C_{\text{cond}})^{1/2} := R_3^\top \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}, \tag{5.23a}$$
$$A_{\text{cond}} := (R_1^{-1} R_2)^\top \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}. \tag{5.23b}$$

7. Evaluate $b_{\text{cond}} := \mathbf{w}\mathbf{H}^\top - A_{\text{cond}}\xi$.

Return $(A_{\text{cond}}, b_{\text{cond}}, (C_{\text{cond}})^{1/2})$.

A common choice of a linearisation point is the input variable $p(z) = p(X)$. Recall from Chapter 4 that the choice between a square Q and a square R in a QR decomposition is up to the user. We always choose a square R (which Golub and Van Loan [67] call "thin QR-factorisation").

To see that Algorithm 5.6 returns the Cholesky parametrisation of Equation (5.9), multiply both sides of Equation (5.22) with their transposes to obtain

$$\begin{pmatrix} \mathbf{X}_{\text{centre}}\mathbf{X}_{\text{centre}}^\top & \mathbf{X}_{\text{centre}}\mathbf{H}_{\text{centre}}^\top \\ \mathbf{H}_{\text{centre}}\mathbf{X}_{\text{centre}}^\top & \mathbf{H}_{\text{centre}}\mathbf{H}_{\text{centre}}^\top \end{pmatrix} = \begin{pmatrix} R_1^\top R_1 & R_2 R_1^\top \\ R_2^\top R_1 & R_2^\top R_2 + R_3^\top R_3 \end{pmatrix}. \tag{5.24}$$

Comparing Equation (5.24) to Equation (5.20), conclude $R_1^\top = (\Xi)^{1/2}$. Rearranging

$$\begin{aligned} R_3^\top R_3 &= \mathbf{H}_{\text{centre}}\mathbf{H}_{\text{centre}}^\top - R_2^\top R_2 && (5.25a)\\ &= \mathbf{H}_{\text{centre}}\mathbf{H}_{\text{centre}}^\top - R_2^\top R_1^{-\top}(R_1^\top R_1) R_1^{-1} R_2 && (5.25b)\\ &= \mathbf{H}_{\text{centre}}\mathbf{H}_{\text{centre}}^\top - A_{\text{cond}} \Xi A_{\text{cond}}{}^\top, && (5.25c) \end{aligned}$$

shows $(C_{\text{cond}})^{1/2} = R_3^\top$ and $A_{\text{cond}} = R_2^\top R_1^{-\top}$ according to Equations (5.9) and (5.20). This derivation mirrors the proofs in Section 4.3, with the difference being that the stacks of cubature-node-evaluations (e.g. $\mathbf{H}_{\text{centre}}$, which is a generalised Cholesky factor but not a Cholesky factor because it is neither square nor triangular) replace the Cholesky factors of the covariance matrices.

If the Jacobian of h is available, Algorithm 5.7 below becomes an option. However, while we always know $C_z[h(z), z] = E_z[Dh(z)]C_z[z, z]$, the same identity is unclear for their respective quadrature approximations. Thus, consider Algorithm 5.7 as an approximation of statistical linear regression based on Equation (5.11).

Algorithm 5.7 (Approximate first-order statistical linear regression with a Jacobian)**.** Call Algorithm 5.6 but replace Equation (5.23b) with

$$A_{\text{cond}} := \mathbf{w}\mathbf{J}^\top, \tag{5.26a}$$

$$\mathbf{J} := \begin{pmatrix} \sqrt{w_0} Dh[(\Xi)^{1/2} x_0 + \xi] & \cdots & \sqrt{w_K} Dh[(\Xi)^{1/2} x_K + \xi] \end{pmatrix}. \tag{5.26b}$$

The advantage of Algorithm 5.7 over Algorithm 5.6 is that it requires one fewer solution of a linear system because A_{cond} directly results from a cubature-approximation of the Jacobian. However, it comes at the price of evaluating the Jacobian at $K + 1$ points.

Again, a common choice of a linearisation point is $p(z) = p(X)$.

To compute zeroth-order statistical linear regression, QR-decompositions are sometimes unnecessary:

Algorithm 5.8 (Zeroth-order statistical linear regression)**.** Assume that the linearisation point $p(z) = \mathcal{N}(\xi, \Xi)$ and the function h are given. Suppose that $p(z)$ is in Cholesky parametrisation, and assume a cubature rule satisfying Assumption 5.4. Linearise h around $p(z)$ as follows:

1. Evaluate $\{h_k\}_{k=0}^K$ according to Equation (5.16).
2. Assemble $\mathbf{w}$, and $\mathbf{H}$ according to Equation (5.17).
3. Compute $b_{\text{cond}} = \mathbf{w}\mathbf{H}^\top$ according to Equation (5.18).
4. Assemble $(C_{\text{cond}})^{1/2} = \mathbf{H}_{\text{centre}}$ according to Equation (5.19).

Return the approximations b_{cond} and $(C_{\text{cond}})^{1/2}$. $\mathbf{H}_{\text{centre}}$ is a generalised Cholesky factor of C_{out}, but not triangular. If subsequent computations demand triangularity, QR-decompose $\mathbf{H}_{\text{centre}} = QR$, discard Q and return $(C_{\text{cond}})^{1/2} = R^\top$.

As always, a common choice of a linearisation point is $p(z) = p(X)$.

To the best of the author's knowledge, Algorithm 5.6 and its relatives (Algorithm 5.7 and Algorithm 5.8) are not described in existing literature. However, alternative Cholesky implementations of statistical linear regression (or the related *sigma-point filters*, respectively) have been proposed:

Remark 5.9. Yaghoobi et al. [185] and Van Der Merwe and Wan [168] discuss a Cholesky implementation of statistical linear regression (in the context of sigma-point and unscented Kalman filtering) using what is known as a "low-rank downdate" or "Cholesky downdate". The algorithms above avoid such downdates, which can be numerically unstable [153], at the price of Assumption 5.4.

5.7 Posterior linearisation

The accuracy of the linearisation of a nonlinear transformation depends on the linearisation point. Previous sections recommended linearisation at the prior distribution, $p(z) = p(X) = \mathcal{N}(m_{\text{in}}, C_{\text{in}})$, which is the natural choice for a linearisation point in the absence of better knowledge. The following strategies describe iterated re-linearisation and apply to all aforementioned techniques equally.

Let $y \in \mathbb{R}^{d_{\text{out}}}$. When the linearisation $p(Y \mid X) \approx \mathcal{N}(A_{\text{cond}}X + b_{\text{cond}}, C_{\text{cond}})$ aims at computing the posterior distribution $p(X \mid Y = y)$ with, for example, Algorithm 4.2, we may iterate in the hope of improving the approximation:

Algorithm 5.10 (Iterated linearisation)**.** Follow the steps:

1. Initialise $p(z_0)$, for example, as $p(z_0) = p(X) = \mathcal{N}(m_{\text{in}}, C_{\text{in}})$.
2. Iterate until convergence ($i = 0, 1, 2, \ldots$):
   (a) Linearise h at $p(z_i)$ with either one of Algorithms 5.1 to 5.8, depending on preferences for Taylor linearisation or statistical linear regression, zeroth-/first-order methods, and the availability of Jacobians. All of these algorithms yield an approximation of the form
   $$p(Y \mid X) \approx \mathcal{N}(\tilde{A}^{(i)}_{\text{cond}} X + \tilde{b}^{(i)}_{\text{cond}}, \tilde{C}^{(i)}_{\text{cond}}), \tag{5.27}$$
   and are thus compatible with the steps below.
   (b) Compute $p(X \mid Y) \approx \mathcal{N}(\tilde{A}^{(i)}_{\text{rev}} Y + \tilde{b}^{(i)}_{\text{rev}}, \tilde{C}^{(i)}_{\text{rev}})$ with either Algorithm 4.2 or Algorithm 4.5, depending on whether $\tilde{C}^{(i)}_{\text{cond}}$ is zero or not.
   (c) Evaluate $p(X \mid Y = y) \approx \mathcal{N}(\tilde{A}^{(i)}_{\text{rev}} y + \tilde{b}^{(i)}_{\text{rev}}, \tilde{C}^{(i)}_{\text{rev}})$ and choose it as the new linearisation point,
   $$p(z_{i+1}) \coloneqq p(X \mid Y = y) \approx \mathcal{N}(\tilde{A}^{(i)}_{\text{rev}} y + \tilde{b}^{(i)}_{\text{rev}}, \tilde{C}^{(i)}_{\text{rev}}). \tag{5.28}$$
   Repeat from Item 2a.

Return the most recent linearisation and posterior distribution, that is, the most recent outputs from Items 2a and 2c.

The goal of Algorithm 5.10 usually is to determine a linearisation at the posterior distribution instead of the prior distribution (e.g. [60, 61]). Taylor linearisation and statistical linear regression imply different interpretations of *posterior linearisation.*

Iterated first-order Taylor linearisation (combining Algorithms 5.1 and 5.10) applied to the transformation

$$p(Y \mid X) = \mathcal{N}(h(X), R) \tag{5.29}$$

implements a Gauss–Newton scheme for the *maximum-a-posteriori estimate* [17, 18],

$$\arg\max_{x \in \mathbb{R}^{d_{\text{in}}}} p(X = x \mid Y = y) = \arg\min_{x \in \mathbb{R}^{d_{\text{in}}}} \left\{ \|x - m_{\text{in}}\|^2_{C_{\text{in}}{}^{-1}} + \|y - h(x)\|^2_{R^{-1}} \right\} \tag{5.30}$$

where "$\propto$" represents proportionality. Algorithm 5.10 together with Algorithm 5.1 initially linearises h at m_{in}, solves the optimisation problem, and re-linearises at the solution before solving the problem again. Repeating these two steps until convergence is the Gauss–Newton algorithm [17, 122]. Many related optimisation algorithms also allow an implementation via iterated first-order Taylor linearisation [e.g., 58, 59, 145].

Iterated zeroth-order Taylor linearisation implements a fixed-point scheme for the posterior mean

$$m_{\text{in}} \rightarrow b_{\text{cond}} (= b_{\text{rev}}) = h(m_{\text{in}}) \rightarrow h(b_{\text{cond}}) \rightarrow \dots . \tag{5.31}$$

and connects to a previously published probabilistic numerical solver:

Remark 5.11. Applied to the differential equation collocation problem,

$$h(y) := \frac{\mathrm{d}y}{\mathrm{d}t} - f(y(t)) \tag{5.32}$$

iterated zeroth-order Taylor linearisation of f implements

$$m_{\text{in}} \rightarrow b_{\text{cond}} := f(m_{\text{in}}) \tag{5.33a}$$

$$\rightarrow m_{\text{in}}{}^{\text{new}} \qquad \text{(Cond. } X \text{ on } \tfrac{\mathrm{d}X}{\mathrm{d}t} = b_{\text{cond}}\text{; 5.33b)}$$

$$\rightarrow b_{\text{cond}}{}^{\text{new}} := f(m_{\text{in}}{}^{\text{new}}) \tag{5.33c}$$

$$\rightarrow \dots \tag{5.33d}$$

which is the algorithm that Arvanitidis et al. [10] employ to solve boundary value problems probabilistically. Hennig and Hauberg [78] employ a similar linearisation scheme. For a certain line search, this procedure relates to a *Mann iteration* [10, 116].

The interpretation of the method by Arvanitidis et al. [10] as iterated zeroth-order linearisation not only implies a (novel) implementation of the boundary value problem solver in linear- instead of cubic-time complexity but also opens up variations that use iterated zeroth-order statistical linear regression, for example.

While iterated Taylor linearisation implements Gauss–Newton-/fixedpoint-style schemes, iterated first-order statistical linear regression approximately minimises a Kullback-Leibler divergence [60]. More specifically, the divergence between the joint distribution over $X \mid Y = y$ and a modification of Y with respect to the Gaussian approximation [60, 61]. For $\|C_{\text{in}}\| \to 0$, statistical linearisation recovers Taylor linearisation and Gauss–Newton emerges. Iterated zeroth-order statistical linearisation has not been discussed in previous literature.

5.8 Sequential problems

The application of linearisation to sequential problems, such as those in the context of probabilistic differential equation solvers, is not the main goal of this chapter. Nevertheless, while details are postponed to Chapter 7, some related literature about applying linearisation to sequential problems needs to be discussed because many of the linearisation algorithms originated in the literature on filtering and smoothing.

If the prior distribution factorises into a sequence of conditional distributions, and if the measurements are conditionally independent, we call the model "sequential". For example, Chapter 3 describes the sequential nature of probabilistic numerical solvers for initial value problems. Both sequential estimation and linearisation are central to the literature on filtering and smoothing. Generally, there are two approaches to linearising a nonlinear sequential estimation problem.

First, we can linearise all nonlinearities at, for example, the prior distribution before processing the measurements (sequentially). This creates an affine estimation problem – which admits a closed-form Gaussian solution via Kalman filtering and Rauch-Tung-Striebel smoothing [e.g. 68, 143] – and usually allows iterative re-linearisation (that is, Algorithm 5.10). Examples of this procedure include algorithms like the *iterated extended Rauch–Tung–Striebel smoother* [17] or the *posterior linearisation smoother* [60, 61]. Posterior linearisation smoothers combine Algorithm 5.10 with sequential Gaussian estimation. The iterated extended Rauch–Tung–Striebel smoother uses Algorithm 5.10 with a Taylor approximation (Algorithm 5.1 or Algorithm 5.2). Statistical linear regression in place of Taylor approximation would be an *iterated sigma-point Rauch–Tung–Striebel smoother* [60] (Algorithm 5.6, Algorithm 5.7, or Algorithm 5.8). Probabilistic numerical solvers use iterated linearisation in sequential problems to solve boundary value problems (Chapter 12).

Second, we may linearise each observation model sequentially during the forward pass of, for example, a filtering algorithm. In contrast to the methods explained in the previous paragraph, linearisation occurs while processing the observations, not before. At each step of the forward pass, the linearisation point depends on all previous observations, which is why we only use it to solve initial value problems (Chapter 7). Linearisation assuming a Gaussian conditional distribution and subsequent sequential Gaussian estimation is *assumed density filtering* [84, 183]. Examples of these algorithms include:

- ◇ The *extended Kalman filter* (Algorithm 5.1 or Algorithm 5.2) [e.g., 68, 143], which uses the exact Jacobian of the nonlinearity. Replacing the Jacobian with a finite-difference approximation yields the *central difference Kalman filter* [84, 124], which implicitly performs statistical linear regression [167]. Assuming or enforcing a diagonal Jacobian is the *node-decoupled Kalman filter* [119], which will appear in an upcoming chapter.
- ◇ Another family of assumed density filters uses statistical linearisation or statistical linear regression (Algorithm 5.6, Algorithm 5.7, or Algorithm 5.8). For example, this includes the *statistical linearisation filters*, or *quasilinear filters* [63, 160]. Statistical linear regression, statistical linearisation with cubature, yields *sigma-point Kalman filters* [167]. Different cubature rules imply different algorithms: The unscented transform implies the *unscented Kalman filter* [87, 173, 174]. Third-order spherical cubature rules imply the *cubature Kalman filter* [7, 183]. Gauss–Hermite cubature implies the *Gauss–Hermite Kalman filter* [84], also known as a *quadrature Kalman filter* [8]. Prüher and Šimandl [132] compute first-order statistical linear regression with Bayesian cubature. Kersting and Hennig [91] do the same for zeroth-order approximations (Algorithm 5.8), but without using this terminology.

All Kalman filter variations transform into Rauch-Tung-Striebel smoother variations via backward marginalisation. Using Cholesky parametrisations of Gaussian variables turns the above filters/smoothers into square-root filters/smoothers. Posterior linearisation during the forward pass yields posterior linearisation filters [61], *iterated extended Kalman filters* [18] or *iterated sigma-point filters* [157, 186]. The above list is not exhaustive but shall express the generality of the linearisation methods and their application to estimation problems. For probabilistic numerical solvers, as discussed in the remainder of this manuscript, it suffices to remember Algorithms 5.1 to 5.10.

5.9 Conclusion

This chapter studied linearisation. The motivation has been that a nonlinear transformation forbids using Gaussian state estimation. Assuming an affine transformation and finding suitable parameters of the affine model resolves the issue via approximation. To this end, either Taylor or statistical linearisation techniques can be employed. Statistical linearisation differs from Taylor linearisation in that it replaces function evaluations with cubature. Both linearisation styles admit a zeroth-order and a first-order version, can be combined with iterative re-linearisation, and are compatible with the Cholesky parametrisation of Gaussian variables. The next chapters explain how linearisation schemes are one part of the implementation of probabilistic numerical solvers.

Chapter 6

Taylor series estimation

Contents

6.1 Problem statement

The previous two chapters discussed Cholesky arithmetic (Chapter 4) and linearisation (Chapter 5). The present chapter introduces the final component of the foundations for implementing probabilistic numerical solvers: estimating Taylor series of solutions of initial value problems (IVPs) that are based on ordinary differential equations. In general, estimating a Taylor-series of an IVP solution is not only important for probabilistic numerical solvers but for many other subjects as well (for instance, refer to Griewank and Walther [69], Kelly et al. [90] for examples). However, the present chapter only demonstrates a single motivation: how Taylor-series estimation plays a special role in the initialisation of probabilistic numerical solvers.

Consider a scalar, nonlinear, autonomous differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y(t)), \quad t \in [0, 1], \tag{6.1}$$

and an initial condition $y(0) = y_0$. Like in the previous chapters, we consider a scalar equation first because techniques for vector-valued models commonly build on those for scalar models. An autonomous vector field shall be no loss of generality (modifications for other differential equations are mentioned where relevant, as usual).

From Chapter 3, recall the ν-times integrated Wiener process prior

$$Y(t) := (Y^{(0)}(t), ..., Y^{(\nu)}(t)), \tag{6.2}$$

where $Y^{(q)}(t)$ is the qth time-derivative of $Y^{(0)}(t)$, which models y. This ν-times integrated Wiener process connects to νth order Taylor series as follows: the stack of states $Y(t)$ contains the first ν time-derivatives of $Y^{(0)}(t)$, which are also the first ν unnormalised Taylor coefficients of $Y^{(0)}(t)$. At each grid-point, the probabilistic numerical solver estimates the first ν Taylor coefficients of the IVP solution.

To compute this estimate, the state needs to be initialised at the initial point $t_0 = 0$,

$$p(Y(t_0) \mid \gamma) = \mathcal{N}(m_0, C_0(\gamma)). \tag{6.3}$$

Chapter 3 assumed that this initialisation is given by assuming that a user has provided m_0 and $C_0(\gamma)$. In practice, these parameters are generally unknown – and while one could choose m_0 and $C_0(\gamma)$ to resemble a diffuse initial condition (Chapter 3) in the hope that not much information is lost in the process, we can do much better by estimating an efficient replacement of $p(Y(t_0) \mid y_0, \gamma)$.

The present chapter discusses such estimators in the following order. Section 6.2 outlines the template for estimating νth order Taylor approximations based on affine differential equations and Section 6.3 continues these ideas for nonlinear problems. Section 6.3 also points out how some of these approaches may be insufficient if ν increases. Consequently, Section 6.4 presents an alternative – a mode of automatic differentiation that is tailored to computing high-order derivatives – and Section 6.5 applies this algorithm to IVPs. Section 6.7 discusses approximations that can be used in the absence of automatic differentiation.

Most of the content presented here is known. Only Section 6.7 is a (small) contribution by this thesis – however, the greater novelty lies in applying these techniques to probabilistic numerical solvers (Chapter 7). Like the previous two chapters, this chapter can be read independently of the rest of this thesis. Readers familiar with Taylor-mode automatic differentiation and its application to ordinary differential equations may skip directly to Section 6.7 or even to the next chapter.

6.2 Affine equations

For now, consider an affine differential equation

$$\frac{\mathrm{d}y(t)}{\mathrm{d}t} = ay(t) + b. \tag{6.4}$$

It suffices to consider constant Jacobian a and bias b because we only care about a single time-point $t = 0$. The initial condition is $y(0) = y_0$, and we aim to compute the first ν time-derivatives of the initial condition.

Since y follows the differential equation, the derivative of the initial condition is

$$\frac{\mathrm{d}y(0)}{\mathrm{d}t} = ay(0) + b = ay_0 + b. \tag{6.5}$$

Via repeating this argument multiple times, the chain rule dictates

$$\frac{\mathrm{d}^q y(0)}{\mathrm{d}t^q} = a\frac{\mathrm{d}^{q-1} y(0)}{\mathrm{d}t^{q-1}}, \quad q \geq 1 \tag{6.6}$$

which shows how the first q derivatives of $y(0)$ can be evaluated recursively:

Algorithm 6.1 (Affine recursion)**.** Assume that a, b, and y_0 are given. Compute the first ν derivatives of the initial value by evaluating the recursion

$$\mathfrak{F}_0 = y_0, \quad \mathfrak{F}_1 = a\mathfrak{F}_0 + b, \quad \mathfrak{F}_q = a\mathfrak{F}_{q-1}, \quad q \geq 2 \tag{6.7}$$

and by returning $(\mathfrak{F}_0, ..., \mathfrak{F}_\nu)$.

The complexity of Algorithm 6.1 is negligibly small compared to any other computation involved in the probabilistic numerical solution of the IVP. If the IVP is vector-valued, the iteration remains the same, but the complexity now depends on the complexity of evaluating the matrix-vector product with a.

For nonlinear differential equations, we could combine Algorithm 6.1 with any of the linearisation techniques in Chapter 5. But instead, we can also generalise the recursion in Equation (6.6) to nonlinear problems:

6.3 Automatic differentiation of IVPs

For the rest of this chapter, call the qth Taylor coefficient $\mathfrak{F}_q$. If the differential equation is nonlinear,

$$\frac{\mathrm{d}y(t)}{\mathrm{d}t} = f(y(t)), \tag{6.8}$$

we may mirror the technique from Section 6.2 Via the differential equation, the derivative of the initial condition is

$$\frac{\mathrm{d}y(0)}{\mathrm{d}t} = f(y_0). \tag{6.9}$$

By the chain rule, all coefficients follow the recursion

$$\frac{\mathrm{d}^{q+1} y}{\mathrm{d}t^{q+1}} = \mathfrak{F}_{q+1}(y) := D\mathfrak{F}_q(y) f(y), \quad q \geq 1 \tag{6.10}$$

initialised with $\mathfrak{F}_1 = f$ and evaluated at $y = y_0$. The following algorithm emerges.

Algorithm 6.2 (Automatic differentiation of IVPs)**.** Assume that f and y_0 are given. Compute the first ν derivatives of the initial value by evaluating the recursion in Equation (6.10) (setting $\mathfrak{F}_0 = y_0$) and returning $(\mathfrak{F}_0, ..., \mathfrak{F}_\nu)$.

If the equation is affine, Algorithm 6.2 reduces to Algorithm 6.1. If the differential equation is not of first-order or not autonomous, Algorithm 6.2 is modified slightly:

Remark 6.3. If the differential equation differs from $\frac{\mathrm{d}y}{\mathrm{d}t} = f(y(t))$, similar recursions apply. For instance, if the vector field also depends on the first derivative of y and time,

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = f\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right), \tag{6.11}$$

with initial conditions for y and $\frac{\mathrm{d}y}{\mathrm{d}t}$ given, Equation (6.10) reads

$$\frac{\mathrm{d}^q y}{\mathrm{d}t^q} := \mathfrak{F}_q\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right), \quad q \geq 2, \tag{6.12}$$

with the recursion

$$\begin{aligned}\mathfrak{F}_{q+1}\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right) &:= D_1\mathfrak{F}_q\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right)\frac{\mathrm{d}y}{\mathrm{d}t} \\ &+ D_2\mathfrak{F}_q\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right) f\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right) + D_3\mathfrak{F}_q\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right) t\end{aligned} \tag{6.13}$$

initialised as $\mathfrak{F}_2\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right) = f\left(y, \frac{\mathrm{d}y}{\mathrm{d}t}, t\right)$. The operators D_1, D_2 and D_3 are the Jacobians of f with respect to the first ("y"), second ("$\frac{\mathrm{d}y}{\mathrm{d}t}$"), and third ("$t$") argument, respectively. All high-order differential equations follow this pattern.

In principle, this algorithm could be implemented in any automatic differentiation framework. However, due to the nested differentiation involved in this recursion, the computational complexity grows rapidly with increasing ν. Therefore, practical implementations of probabilistic numerical solvers tend to avoid this algorithm for

$\nu > 5$ and use a specialised mode of algorithmic differentiation that efficiently computes certain high-order derivatives:

6.4 Taylor-mode automatic differentiation

Let $x \in \mathbb{R}$. Let v, v_1, v_2, ... be known vectors and h_1, h_2, ..., be given functions. In the rest of this section, we need to be meticulous with the notation for derivatives: The Jacobian of a function $h : \mathbb{R}^{d_{\text{in}}} \longrightarrow \mathbb{R}^{d_{\text{out}}}$ maps an input to a linear operator, $Dh : \mathbb{R}^{d_{\text{in}}} \longrightarrow L(\mathbb{R}^{d_{\text{in}}}, \mathbb{R}^{d_{\text{out}}})$, where $L(\mathbb{R}^{d_{\text{in}}}, \mathbb{R}^{d_{\text{out}}})$ is the space of linear operators. Denote this linear operator by $Dh(x)[\cdot]$. The Hessian of h maps an input to a bilinear operator $D^2h(x)[\cdot, \cdot]$. The qth derivative maps to a q-linear operator $D^q h(x)[\cdot, ..., \cdot]$, $q \in \mathbb{N}$. From Chapter 7 onwards, this notational requirement will be relaxed again.

Let us begin with a brief review of the concept of (conventional) automatic differentiation. Jacobian-vector products (evaluations of the linear operator $Dh[\cdot]$ at vector v) follow the chain rule,

$$(z_0, z_1) \coloneqq (h_1(x), Dh_1(x)[v]) \mapsto (h_2(z_0), Dh_2(z_0)\,[z_1]). \tag{6.14}$$

Evaluation of the propagation in Equation (6.14) requires access to h_2 and Dh_2. For elementary operations (for example, addition, subtraction, polynomials, or exponentials), h_2 and Dh_2 are known in closed form. By phrasing computer programs as compositions of these functions, repeated application of Equation (6.14) yields directional derivatives of computer programs. Algorithmic evaluation of derivatives without numerical differences or symbolic differentiation is *automatic differentiation* [e.g. 69, 71], and sometimes called *algorithmic differentiation* [e.g. 172]. Computing Jacobian-vector products by repeated application of Equation (6.14) is *forward-mode* automatic differentiation [69, Section 3.1].

Higher-order differentiation allows the same strategy. Instead of Jacobian-vector products, higher-order differentiation propagates evaluations of the q-linear operators associated with qth derivatives. For example, second-order differentiation uses

$$\begin{aligned}
&(z_0, z_1, z_2, z_3) \\
&\quad \coloneqq (h_1(x), Dh_1(x)[v_1], Dh_1(x)[v_2], D^2h_1(x)[v_1, v_2]) \\
&\quad \mapsto (h_2(z_0), Dh_2(z_0)[z_1], Dh_2(z_0)[z_2], D^2h_2(z_0)[z_1, z_2] + Dh_2(z_0)[z_3]).
\end{aligned} \tag{6.15}$$

It requires access to h_2, Dh_2, and D^2h_2, which are usually known for a wide range of elementary operations. As in the context of Equation (6.14), repeated application of Equation (6.15) yields directional derivatives of computer programs. However, formulas like Equation (6.15) become costly for high-order derivatives:

The highest derivative $D^2h_2(z_0)[z_1, z_2]$ in Equation (6.15) is the sum of two terms, $D^2h_2(z_0)[z_1, z_2]$ and $Dh_2(z_0)[z_3]$. Third-order differentiation would involve the sum of five terms, fourth-order differentiation would require summing 15 terms, and for any $q \in \mathbb{N}$, qth order differentiation requires as many terms as the set $\{1, ..., q\}$ admits set partitions [86]. The number of partitions of a set is the *Bell number* [141], which grows at least as fast as $O(2^q)$ [45, 141]. Exponential complexity forbids algorithmic evaluation of directional derivatives for large q. For instance, $q = 11$ involves the sum of 678570 terms (because 678570 is the 12th Bell number [158, Entry "A000110"]).

However, many of the $O(2^q)$ terms are redundant in our application: The difference between evaluating a qth order Taylor series and evaluating any qth order derivatives is that for the Taylor series, all derivatives point in the same direction, $v_1 = v_2 = ... = v$. (For intuition, read this as $v := x - x_0$.) Acknowledging such a redundancy reduces the number of terms in the qth-order derivative from the number of set partitions of $\{1, ..., q\}$ to the number of integer partitions of q; in other words, from $O(2^q)$ to $O(\exp\sqrt{q})$ [6]. For example, third-order differentiation simplifies to

$$\begin{aligned}\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} &:= \begin{pmatrix} h_1(x) \\ Dh_1(x)[v] \\ D^2h_1(x)[v, v] \\ D^3h_1(x)[v, v, v]) \end{pmatrix} \\ &\mapsto \begin{pmatrix} h_2(z_0) \\ Dh_2(z_0)[z_1] \\ D^2h_2(z_0)[z_1, z_1] + Dh_2(z_0)[z_2] \\ D^3h_2(z_0)[z_1, z_1, z_1] + 3D^2h_2(z_0)[z_1, z_2] + Dh_2(z_0)[z_3] \end{pmatrix}. \end{aligned} \tag{6.16}$$

The highest-order derivative in Equation (6.16) involves three instead of five terms. The reduction is only possible because all derivatives use the same direction; thus, it only applies to computing Taylor series. Like Bettencourt et al. [23], we call algorithmic evaluation of high-order directional derivatives *Taylor-mode* automatic differentiation if all directions coincide, as it is the case for Taylor series.

Usually, the complexity reduces further than $O(\exp\sqrt{q})$: Most functions have comparably simple high-order derivatives and thus can be Taylor-mode differentiated in $O(q^2)$ or even $O(q)$. For example, if h is linear, $D^q h$ is zero for all $q \geq 2$. (Linear functions frequently occur in array-based computing; for example, reshaping an array or accessing one of its elements is a linear operation.) If we know that h_2 is linear, third-order Taylor-mode differentiation (Equation (6.16)) reduces to

$$(z_0, z_1, z_2, z_3) \mapsto (h_2(z_0), Dh(z_0)[z_1], Dh_2(z_0)[z_2], Dh_2(z_0)[z_3]), \tag{6.17}$$

because D^2h_2 and D^3h_2 are always zero. Equation (6.17) is cheap to evaluate because it reduces to evaluating the Jacobian of h_2 in three directions, and instead of evaluating $3 + 2 + 1 = 6$ terms like in Equation (6.16), Equation (6.17) requires only $1 + 1 + 1 = 3$ terms. In this vein, qth order Taylor-mode differentiation of a linear function costs

$O(q)$.

Other classes of functions also imply a low-cost evaluation of high-order derivatives. For instance, if h_2 is an exponential function, high-order derivatives are simple to evaluate because exponentials solve the differential equation $Dh(x) = h(x)$. Due to this differential equation perspective on exponentials, qth order Taylor-mode differentiation of exponential functions costs only $O(q^2)$ instead of $O(\exp \sqrt{q})$ [69, Proposition 13.1]. The same holds for all functions that solve a certain differential equation, for example, trigonometric functions like "sin" and "cos" [69]. As a result, Taylor-mode differentiation can usually be implemented in $O(q^2)$ complexity, making it significantly more efficient than recursive forward-mode differentiation for large q (which costs $O(2^q)$).

Let Ψ be a routine that implements Taylor-mode automatic differentiation, as discussed in the previous chapter. This means that Ψ maps a function g as well as a truncated Taylor series $(z_0, ..., z_\nu)$ to the truncated Taylor series of $g(z_0)$,

$$(g_0, ..., g_\nu) = \Psi(h, (z_0, ..., z_\nu)). \tag{6.18}$$

Each g_q is the qth Taylor-coefficient of $g(z_0)$; for example Equation (6.16) maps the third-order approximation of $h_1(x)$ to the third-order approximation of $h_2(h_1(x))$.

The operator Ψ is implemented in different software packages; for example, it is available in JAX [28] or TaylorSeries.jl [19]. In JAX, Ψ is called "jet". *Jets* are generalisations of tangent vectors [e.g., 99], and the composition of jets reduces to the propagation fo truncated Taylor series (which matches Taylor-mode automatic differentiation). We refer to Kolár et al. [99] for background on the geometry of jets, and Betancourt [22] for a jet-centric perspective on higher-order differentiation; see also the work by Pusch [133].

6.5 High-order Taylor series of IVPs

We return to the IVP setting. Recall the vector field f and the initial value y_0.

Suppose we have access to an implementation of Ψ and that evaluating Ψ at a νth order series costs $O(\nu^2)$. We can compute the first ν Taylor coefficients of the IVP solution at time $t = t_0$ with Taylor-mode differentiation:

Algorithm 6.4 (Taylor-mode differentiation of IVPs)**.** Assume that Ψ, f, y_0, and ν are known. First, initialise the zeroth-order Taylor approximation of the IVP solution with

$$y_{\text{Taylor},0} \coloneqq y_0. \tag{6.19}$$

Then, repeat the following steps for $q = 0, ..., \nu - 1$:

1. Evaluate $(f_0, ..., f_q) = \Psi(f, y_{\text{Taylor},q})$.
2. Set $y_{\text{Taylor},q+1} = (y_0, f_0, ..., f_q)$.

Finally, return $y_{\text{Taylor},\nu}$.

To see how Algorithm 6.4 computes the correct Taylor coefficients of the IVP solution, consider the following: The essential step in Algorithm 6.4 is to map the first q Taylor coefficients of $y(t)$ to the first q Taylor coefficients of $f(y(t))$ via Ψ (all Taylor coefficients shall be unnormalised). Due to the differential equation, the first q Taylor coefficients of $f(y(t))$ are the first q Taylor coefficients of $\frac{\mathrm{d}y(t)}{\mathrm{d}t}$. But by definition of Taylor coefficients, the first q Taylor coefficients of $\frac{\mathrm{d}y}{\mathrm{d}t}$ are the second-to-$(q+1)$th Taylor coefficients of y, which is how we gain one coefficient through Ψ. Hairer et al. [76, Section I.8] call Algorithm 6.4 *Newton's method* for solving differential equations.

Analogues of Algorithm 6.4 for higher-order or non-autonomous differential equations are relatively straightforward to derive; for example, by rewriting a second-order differential equation as a system of first-order equations or by reformulating a non-autonomous equation as an autonomous equation over the extended state (y, t). We leave details to the reader.

In summary, Algorithm 6.4 recursively assembles the νth order Taylor series of the initial value. Since we call Ψ exactly ν-times, and each evaluation costs $O(\nu^2)$, Algorithm 6.4 costs $O(\nu^3)$. However, perhaps surprisingly, fewer evaluations of Ψ suffice to evaluate high-order derivatives of IVP solutions:

Remark 6.5 (Coefficient doubling). The recursion in Algorithm 6.4 computes the νth order Taylor series of an IVP solution via ν calls to Ψ. It can be shown [69, Corollary 13.2] that only $O(\log(\nu))$ calls are needed. The reasons are twofold.

First, lower-order coefficients of the output do not depend on higher-order coefficients of the input, so we may pad $y_{\text{Taylor},q}$ with zeros to an arbitrary length without altering the result. (On a side note, this fact can be helpful to implement Algorithm 6.4 in a framework that requires static bounds on memory requirements, such as JAX.)

Second, for any $\frac{k}{2} < j \leq k+1$, the $(k+1)$th unnormalised Taylor coefficient of y depends linearly on some coefficients,

$$\frac{\mathrm{d}^{(k+1)}y(t_0)}{\mathrm{d}t^{(k+1)}} = \Psi(f, (y_{\text{Taylor},j-1}, 0_{k-j})) + \sum_{i=j}^{k} A_i \frac{\mathrm{d}^i y(t_0)}{\mathrm{d}t^i}. \tag{6.20}$$

Here, $0_n = (0, ..., 0) \in \mathbb{R}^n$ is a vector of zeros and $\{A_i\}$ are certain matrices that depend on the Jacobian of Ψ [69]. Equation (6.20) shows that with a single evaluation of Ψ and its Jacobian, the number of Taylor coefficients can be doubled,

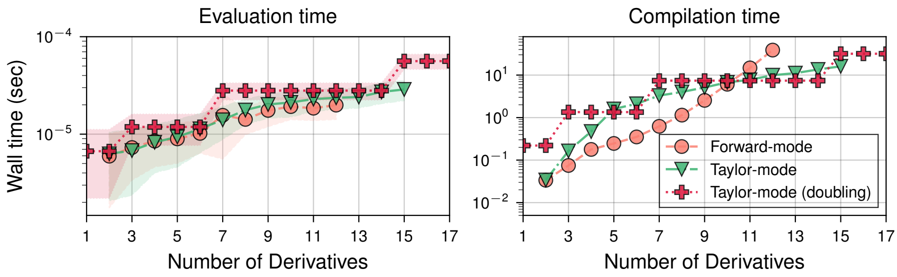

Figure 6.1: Taylor-series estimation on the FitzHugh-Nagumo problem.

which is why Griewank and Walther [69, Table 13.7] call it *coefficient doubling*.

The benchmarks in Section 6.6 will demonstrate how Taylor-mode-based algorithms eventually outperform forward-mode differentiation approaches if ν gets large.

6.6 Benchmarks

All examples are implemented in Python with `probdiffeq`[1], which builds on JAX [28]. Versions of these benchmarks are available in `probdiffeq`'s online documentation.

Fitzhugh-Nagumo

The first benchmark involves the FitzHugh-Nagumo problem [54, 120], which is a first-order, autonomous, two-dimensional differential equation. The precise parametrisation of the problem is irrelevant for this benchmark because all algorithms compute the same values; only the computational complexities matter.

Figure 6.1 compares the evaluation- and compilation-times of Taylor-mode differentiation (Algorithm 6.4), forward-mode differentiation (Algorithm 6.2), and coefficient doubling (Remark 6.5) as the number of derivatives ν increases. We measure the evaluation time as well as the compilation time because the exponential increase in the complexity of recursive Jacobian-vector products is almost invisible for a reasonably low number of derivatives when only considering the evaluation time. However, the compilation time unveils how JAX's just-in-time-compilation processes exponentially many terms for forward-mode-based Taylor series estimation, which is not the case for Taylor-mode-based algorithms. That said, until the exponential increase in compilation time makes forward-mode prohibitively costly around $\nu \approx 11$, it seems to be the fastest algorithm by a minimal margin.

The precise change-point of $\nu \approx 11$ depends on using JAX as an underlying framework and may be lower if implementing both algorithms in a different framework.

[1]See `https://github.com/pnkraemer/probdiffeq/` or install via `pip install probdiffeq`.

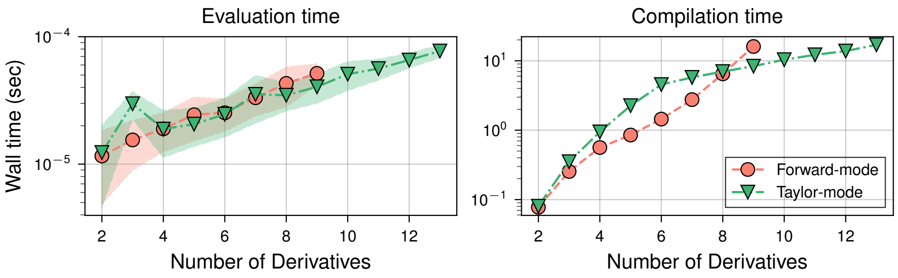

Figure 6.2: Taylor-series estimation on the Pleiades problem.

Pleiades

Next, we repeat the same benchmark on the Pleiades problem [76, p. 245]. The Pleiades problem is a second-order, autonomous, 14-dimensional equation.

Figure 6.2 again compares the evaluation- and compilation-time of forward-mode and Taylor-mode algorithms. (Coefficient doubling has not been implemented for second-order problems at the time of writing this.) The results are similar to the results of the FitzHugh-Nagumo benchmark. Both algorithms have a comparable evaluation time, and the algorithm's complexities are more aptly demonstrated by the compilation times. The compilation times express how Taylor-mode costs sub-exponentially and forward-mode costs exponentially, yet forward-mode is sufficiently well-optimised to be faster overall for $\nu < 8$. For $\nu > 8$, Taylor-mode seems to be more efficient and the curves in Figure 6.2 suggest that around $\nu \approx 10$, the compilation time of forward-mode time would become impractical.

Neural ordinary differential equation

At last, we repeat the simulations on a (medium-)high-dimensional problem, a 100-dimensional neural differential equation [e.g., 33], which is a differential equation with vector field

$$f(y) = \tanh(W_1 \cdot \tanh(W_2 \cdot y + b_2) + b_1), \tag{6.21}$$

parametrised by $W_1, W_2 \in \mathbb{R}^{100\times100}$, and $b_1, b_2 \in \mathbb{R}^{100}$, and with $y \in \mathbb{R}^{100}$. All of these matrices are populated with independent samples from a standard normal distribution. However, the values of these parameters are unimportant for this study – only the computational complexity matters.

The results of a similar benchmark to the previous ones are in Figure 6.3 and mirror the learnings from earlier: Taylor-mode differentiation is consistently efficient, forward-mode differentiation is slightly faster than Taylor-mode until it becomes

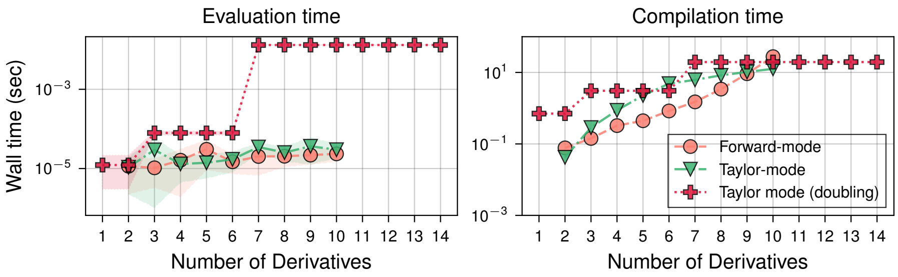

Figure 6.3: Taylor-series estimation on a 100-dimensional neural differential equation.

infeasible (around $\nu \approx 10$), and coefficient doubling is the most viable option for $\nu \gg 10$. The only difference between Figure 6.3 and the previous figures is that in Figure 6.3, the performance difference between coefficient doubling and competing methods is smaller for $\nu < 10$ than in the previous experiments.

In total, all three experiments consistently show how for lower ν, forward-mode is the fastest method, whereas, for larger ν, it may evolve from the fastest to an infeasible algorithm due to the exponential increase in compilation time. Taylor-mode differentiation is the more reliable option throughout, especially for larger ν. For extremely large ν, say, $\nu \gg 10$, coefficient doubling is the only viable option. That said, probabilistic numerical solvers rarely exceed $\nu = 12$.

6.7 Regression-based approaches

The remainder of the chapter explains how to estimate a Taylor series in the absence of (Taylor-mode) automatic differentiation. In this case, we can estimate Taylor coefficients via regression instead of implementing recursions with automatic differentiation.

Recall the ν-times integrated Wiener process $Y(t)$ with output scale γ. Let $\tau_0, ..., \tau_\nu$ be grid points. Assume that we have an accurate approximation $\hat{y}$ of the IVP solution y at $\{\tau_0, ..., \tau_\nu\}$, for example, computed by a high-precision Runge–Kutta solver.

The conditional distribution

$$p\left(Y(t_0) \;\middle|\; Y^{(0)}(\tau_0) = \hat{y}(\tau_0), ..., Y^{(0)}(\tau_\nu) = \hat{y}(\tau_0), \gamma\right) \tag{6.22}$$

estimates the ν time-derivatives of the IVP solution at time $t = t_0$. It is similar to how Schober et al. [152] initialise probabilistic numerical IVP solvers, which itself relates to the Runge–Kutta starter by Gear [62]. Even though the approximation $\hat{y}$ may be computed with any IVP solver, we follow the terminology introduced by Gear [62] and call the algorithm that estimates the Taylor series by implementing Equation (6.22) a *Runge–Kutta starter*.

Since $Y(t)$ is a Gaussian process, the conditional distribution in Equation (6.22) is Gaussian. Due to all observations being conditionally independent, and since $Y(t)$ is Markovian, Equation (6.22) allows a sequential implementation. This sequential implementation is not important for the acceleration of the algorithm (ν is usually bounded by 10), but rather because by using a sequential implementation, all stability and efficiency considerations for sequentially estimating IVP solutions discussed in this thesis apply (for example, Cholesky arithmetic from Chapter 4 as well as the preconditioners that will be part of Chapter 7).

As an initial distribution for the integrated Wiener process, we usually choose a diffuse initialisation for simplicity (Chapter 3) and because we found the precise parametrisation of the initial distribution to be unimportant for this regression task.

The advantages of Runge–Kutta starters over, say, Taylor-mode automatic differentiation are threefold:

1. Runge–Kutta starters do not require access to automatic differentiation but only to a (non-probabilistic) IVP solver. Even if such a solver is not available, a fixed-step Runge–Kutta method can usually be implemented in a few lines of code, whereas implementing Taylor-mode automatic differentiation is significantly more involved.

2. Runge–Kutta starters are compatible with the configurations of probabilistic numerical solvers for vector-valued problems, whereas automatic differentiation is limited for high-dimensional setups. This makes the Runge–Kutta starter the default for extremely high-dimensional differential equations (Chapter 8).

3. While the strategy underlying the Runge–Kutta starter generalises to all differential equations for which one can construct a non-probabilistic solver, the recursion for the automatic differentiation of IVP solutions only works for IVPs based on ordinary differential equations. For boundary value problems or differential-algebraic equations, estimating Taylor series with automatic differentiation an open problem.

However, if (Taylor-mode) automatic differentiation is available, it remains the method of choice. The reasons are (again) threefold:

1. Taylor-mode differentiation is exact; Runge–Kutta starters approximate.

2. Taylor-mode differentiation does not have any parameters that need calibration; the efficacy of Runge–Kutta starters depends on $\{\tau_q\}_{q=0}^{\nu}$.

3. Only Taylor-mode differentiation scales to (very) large ν.

Nevertheless, there are scenarios in which Runge–Kutta starters are useful, especially as the dimension of the differential equation increases to hundreds of thousands of dimensions. We refer to the benchmarks in Chapter 9 for examples.

6.8 Conclusion

This chapter presented the estimation of the truncated Taylor series of IVP solutions. In principle, all techniques except the regression-based one depend on applying the chain rule to the differential equation. The efficiency of the resulting algorithms mostly depends on the efficiency of computing high-order derivatives automatically, which is possible with Taylor-mode differentiation.

Part III

Implementation

Chapter 7

Scalar-valued problems

Contents

7.1 Problem setting

The present chapter is at the heart of this manuscript because it delivers precise instructions for implementing probabilistic numerical solvers for (scalar) initial value problems. The following sections will combine the general strategies from Chapters 1 to 3 with the algorithms in Chapters 4 to 6. For the remainder of this chapter, we assume familiarity with the content of all previous chapters, and most upcoming chapters will require reading the present Chapter 7.

Recall the problem setting from Chapter 3: Consider a one-dimensional initial value problem (IVP), that is, an ordinary differential equation

$$\frac{\mathrm{d}y(t)}{\mathrm{d}t} = f(y(t)), \quad t \in [0, 1], \tag{7.1}$$

constrained by $y(0) = y_0$. Assume that the vector field f is sufficiently well-behaved that the IVP admits a unique solution $y : [0, 1] \to \mathbb{R}$. As in all previous chapters, it is only for notational reasons that we consider autonomous problems on the interval $[0, 1]$, and modifications for other problem types will be explained where relevant. Vector-valued equations are the subject of Chapter 8.

The following concepts from Chapter 3 are important: Let $Y = (Y^{(0)}, ..., Y^{(\nu)})$ be the state-variable of the stochastic differential equation-representation of a ν-times integrated Wiener process with constant output-scale $\gamma > 0$. Let $\{t_0, ..., t_N\}$ be a point set with spacing $\Delta t_n := t_{n+1} - t_n$. Assume $t_0 = 0$ and $t_N = 1$, with the obvious modifications if an equation shall be solved on a different time-interval. Restricted to the point set, $Y(t)$ evolves as

$$p(Y(t_{n+1}) \mid Y(t_n), \gamma) = \mathcal{N}(\Phi_\nu(\Delta t_n)Y(t_n), \Sigma_\nu(\Delta t_n, \gamma)) \tag{7.2}$$

with initial distribution $p(Y(t_0) \mid \gamma) = \mathcal{N}(m_0, C_0(\gamma))$. More detailled information regarding the parameters $\Phi_\nu(\Delta t_n)$, $\Sigma_\nu(\Delta t_n, \gamma)$, m_0, and $C_0(\gamma)$ has been explained in the context of Equation (3.6) in Chapter 3.

We consider m_0, $C_0(\gamma)$, ν, $t_0, ..., t_N$, f, and y_0 to be fixed and known, and γ to be fixed and unknown. To enable estimating γ (Sections 7.5 and 7.6), we need the following assumption about the covariance matrix of the initial distribution.

Assumption 7.1. *Assume that the covariance matrix of the initial distribution depends on γ as $C_0(\gamma) = \gamma^2 C_0(1)$.*

The process noise covariance matrix $\Sigma_\nu(\Delta t_n, \gamma)$ satisfies the same criterion by construction, $\Sigma_\nu(\Delta t_n, \gamma) = \gamma^2 \Sigma_\nu(\Delta t_n, 1)$.

Recall the random variables representing the IVP constraints from Section 3.4,

$$\mathcal{R}_{y_0} := Y^{(0)}(t_0) - y_0 \tag{7.3a}$$

$$\mathcal{R}_{f,n} := Y^{(1)}(t_n) - f(Y^{(0)}(t_n)), \quad n = 0, ..., N, \tag{7.3b}$$

which measure the residual of the initial condition and the differential equation. Like in previous chapters, we abbreviate $\mathcal{R}_{f,\ell:n} = \{\mathcal{R}_{f,k}\}_{k=\ell}^n$ and $Y(t_{\ell:n}) := \{Y(t_k)\}_{k=\ell}^n$. The probabilistic numerical IVP solution is

$$p(Y(t_{0:N}) \mid \mathcal{R}_{f,0:N} = 0, \mathcal{R}_{y_0} = 0, \gamma). \tag{7.4}$$

Chapter 3 showed how the probabilistic numerical IVP solution factorises into a sequence of backward conditionals and how the likelihood of the constraints

$$p(\mathcal{R}_{y_0} = 0, \mathcal{R}_{f,0:N} = 0 \mid \gamma) \tag{7.5}$$

factorises sequentially, too. Such a sequential factorisation implies that the following strategy estimates the IVP solution and the likelihood of the constraints in a constant number of operations per grid-point:

1. *Initialisation:* Estimate the marginal and the conditional distributions from the joint distribution $p(Y(t_0), \mathcal{R}_{f,0}, \mathcal{R}_{y_0} \mid \gamma)$ or an appropriate substitution thereof

(see below). These quantities describe the state at the initial grid point t_0 and serve as the input to the following loop.

For $n = 0, ..., N - 1$, alternate extrapolation and correction:

2. *Extrapolation:* Estimate the marginal and conditional distributions from the joint distribution $p(Y(t_n), Y(t_{n+1}) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma)$. These quantities predict the state at time t_{n+1} from the information available at all previous times.

3. *Correction:* Estimate the marginal and conditional distributions from the joint distribution $p(Y(t_{n+1}), \mathcal{R}_{f,n+1} \mid \mathcal{R}_{y_0} = 0, \mathcal{R}_{f,0:n} = 0, \gamma)$. These quantities describe the solution after a new data point has been incorporated, and from here on, a new extrapolation step can be initiated.

This was Algorithm 3.2. The present chapter provides details for each of the steps.

More specifically, Section 7.2 explains how Taylor-series estimation algorithms from Chapter 6 improve the initialisation. Section 7.3 shows how appropriate preconditioning stabilises the extrapolation, Section 7.4 how to implement the correction, and Sections 7.5 and 7.6 how to estimate the output scale γ. Section 7.7 concludes by selecting templates for probabilistic numerical solvers for one-dimensional problems.

7.2 Initialisation

We have two options to initialise the state estimate: a natural option and a better one. Recall from Chapter 6 how at each time point, $Y(t)$ contains the first ν unnormalised Taylor coefficients of the IVP solution. The two options are:

1. *Gaussian conditioning:* Estimate the marginals and conditionals from the joint distribution $p(Y(t_0), \mathcal{R}_{f,0}, \mathcal{R}_{y_0} \mid \gamma)$. To this end, begin by computing the conditional $p(Y(t_0) \mid \mathcal{R}_{y_0}, \gamma)$, which is Gaussian and available in closed form by implementing any of the algorithms in Chapter 4. Optionally, track the likelihood of the constraints $p(\mathcal{R}_{y_0} \mid Y(t_0), \gamma)$. Then, proceed with a correction step using $p(Y(t_0) \mid \mathcal{R}_{y_0}, \gamma)$ as the prediction; see Section 7.4.

2. *Taylor-series estimation:* Estimate the first ν derivatives of the IVP solution,

$$y(t_0), \frac{\mathrm{d}y(t_0)}{\mathrm{d}t}, ..., \frac{\mathrm{d}^\nu y(t_0)}{\mathrm{d}t^\nu}, \tag{7.6}$$

with any of the algorithms in Chapter 6, preferably, Taylor-mode automatic differentiation, and set

$$p(Y(t_0) \mid \mathcal{R}_{f,0}, \mathcal{R}_{y_0}, \gamma) \approx p\left(Y(t_0) \,\middle|\, \left\{Y^{(q)}(t_0) = \frac{\mathrm{d}^q y(t_0)}{\mathrm{d}t^q}\right\}_{q=0}^{\nu}, \gamma\right) \tag{7.7a}$$

$$= \delta\left[\left(y(t_0), \frac{\mathrm{d}y(t_0)}{\mathrm{d}t}, ..., \frac{\mathrm{d}^\nu y(t_0)}{\mathrm{d}t^\nu}\right)\right]. \tag{7.7b}$$

It is possible to track the likelihood of the constraints in Equation (7.7a), but this step is usually skipped in practice.

Item 1 would be the technically correct initialisation for the probabilistic numerical IVP solution according to Equation (7.4). Nevertheless, Item 2 is the method of choice wherever applicable.

The advantage of the implementation in Item 1 over the alternative is generality: the necessary modifications that turn an initialiser for an initial value problem into one for a boundary value problem or a partial differential equation are relatively straightforward: replace $\mathcal{R}_{f,n}$ or $\mathcal{R}_{y_0}$ by its respective counterpart. Item 1 is possible for all problems that admit a sequential implementation of a probabilistic numerical solver. Its disadvantage is that it is inaccurate (because, while it initialises the zeroth-derivative and first derivative exactly, it does not estimate any of the higher derivatives) and requires selecting parameters m_0 and $C_0(\gamma)$.

In contrast, the advantage of the implementation in Item 2 over Item 1 is improved performance: if automatic differentiation is available, all derivatives are computed exactly and in reasonable runtime (Chapter 6). Since the full state $Y(t)$ is perfectly reconstructed, the choices of m_0 and $C_0(\gamma)$ are irrelevant, and a user does not have to choose these parameters. That said, Item 2 only applies to initial value problems based on ordinary differential equations, which is why we must revisit Item 1 in the later Chapters 11 and 12.

7.3 Extrapolation

After initialisation, the first extrapolation step occurs. The present section explains implementing numerically stable extrapolation with high-order integrated Wiener processes, but some of the techniques also apply to other stochastic processes like high-order Matèrn processes. We will comment on the algorithms accordingly.

For integers $a, b \in \mathbb{N}$, $a!$ is the factorial of a and $\binom{a}{b} = \frac{a!}{(a-b)!b!}$ is the binomial coefficient of "a over b" or "a choose b". Let $n \in \{0, ..., N-1\}$. Assume that a Cholesky parametrisation of the Gaussian distribution

$$p(Y(t_n) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) \tag{7.8}$$

is available. The first extrapolation receives this parametrisation from the initialisation routines discussed above; all remaining extrapolations build on the results of previous steps. The subscript "$n + 1 \mid n$", as in $m_{n+1|n}$, will encode the value of a variable at time t_{n+1} given all observations up to time t_n. For example, $m_{n+1|n}$ is the mean of the Gaussian process conditioned on the initial value and $\mathcal{R}_{f,0:n} = 0$.

Recall from Equation (7.2) that $Y(t_{n+1})$ is an affine transformation of $Y(t_n)$ with additive Gaussian noise. In principle, we could extrapolate with Algorithm 4.2: A single QR-decomposition of a matrix with $2(\nu + 1)$ rows and columns (and some matrix-vector arithmetic) computes a Cholesky parametrisation of the marginal

$$p(Y(t_{n+1}) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) = \mathcal{N}(m_{n+1|n}, C_{n+1|n}(\gamma)) \tag{7.9}$$

and a Cholesky parametrisation of the conditional

$$p(Y(t_n) \mid Y(t_{n+1}), \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) = \mathcal{N}(\Lambda_{n|n+1} Y(t_{n+1}) + \lambda_{n|n+1}, \Xi_{n|n+1}(\gamma)) \tag{7.10}$$

in one go. In other words, Algorithm 4.2 would compute $m_{n+1|n}$, the Cholesky factor of $C_{n+1|n}(\gamma)$, $\Lambda_{n|n+1}$, $\lambda_{n|n+1}$, and the Cholesky factor of $\Xi_{n|n+1}(\gamma)$ from the parameters of the previous state and the extrapolation model. However, this approach would suffer from numerical instability:

The computation of $\Lambda_{n|n+1}$, a part of Algorithm 4.2, requires the numerical solution of linear systems involving the Cholesky factor of $C_{n+1|n}(\gamma)$. The conditioning of this linear system, which is quantified by the magnitude of the *condition number* of the matrix, is terrible – that is, the matrix has a large condition number (we will see an example later). Condition numbers indicate how the process of solving an equation amplifies small perturbations in the input. Roughly speaking, the condition number states how much precision is lost when inverting a matrix and the larger the number, the more precision is lost. Huge condition numbers render the solution of linear systems practically infeasible.

The covariance matrix $C_{n+1|n}(\gamma)$ of the marginal distribution and its generalised Cholesky factors are one example of huge condition numbers. $C_{n+1|n}(\gamma)$ is the sum of a positive semidefinite matrix and the positive definite Hankel matrix $\Sigma_\nu(\Delta t_n, \gamma)$. Therefore, it must be invertible – in theory. In practice, it can be ill-conditioned because the condition number of a Hankel matrix of order k grows like 2^k [e.g. 16, 166] – and the process noise covariance matrix is not just a Hankel matrix with order $k = \nu$, but its condition number far exceeds 2^ν for small step sizes as demonstrated below. Figure 7.1 displays the entries of $\Sigma_\nu(\Delta t, \gamma)$ for varying ν and Δt. In general, the numerical stability of the extrapolation step should not depend on Δt.

Fortunately, the dependence of $\Sigma_\nu(\Delta t_n, \gamma)$ on Δt_n is unnecessary: We can change the coordinate system to remove the dependence on the spacing Δt_n from the process noise. The computation of the backward transition $\Lambda_{n|n+1}$ becomes more stable because the extrapolation happens in a more suitable coordinate system.

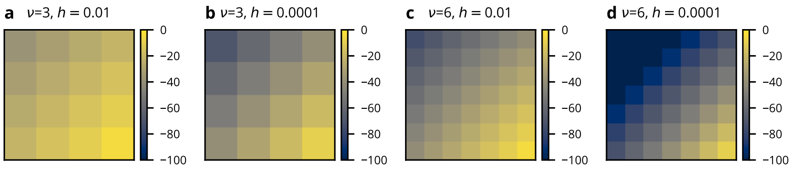

Figure 7.1: The elements in the process noise matrix $\Sigma_\nu(\Delta t, \gamma)$ decay rapidly from the bottom right to the top left element. The figure displays the logarithm of the entries in $\Sigma_\nu(\Delta t, \gamma)$ for different ν and time-steps (referred to as h instead of Δt in this figure)

To this end, define the diagonal matrix

$$T_\nu : (0, \infty) \to \mathbb{R}^{(\nu+1)\times(\nu+1)}, \quad \tau \mapsto \sqrt{\tau}\,\mathrm{diag}\left(\frac{\tau^\nu}{\nu!}, \frac{\tau^{\nu-1}}{(\nu-1)!}, ..., \tau, 1\right). \tag{7.11}$$

For every $\tau > 0$, the matrix $T_\nu(\tau)$ is invertible with inverse

$$T_\nu(\tau)^{-1} := \frac{1}{\sqrt{\tau}}\,\mathrm{diag}\left(\frac{\nu!}{\tau^\nu}, \frac{(\nu-1)}{\tau^{\nu-1}}, ..., \frac{1}{\tau}, 1\right). \tag{7.12}$$

For all $n \in \{0, ..., N-1\}$, T_ν eliminates the Δt_n-dependence from $\Phi_\nu(\Delta t_n)$ and $\Sigma_\nu(\Delta t_n, \gamma)$, because they decompose into

$$\Phi_\nu(\Delta t_n) = T_\nu(\Delta t_n)\Phi_{\nu,\mathrm{pre}}T_\nu(\Delta t_n)^{-1}, \tag{7.13a}$$
$$\Sigma_\nu(\Delta t_n, \gamma) = T_\nu(\Delta t_n)\Sigma_{\nu,\mathrm{pre}}(\gamma)T_\nu(\Delta t_n)^\top, \tag{7.13b}$$

with system matrices

$$\Phi_{\nu,\mathrm{pre}} := \left[\binom{\nu-i}{\nu-j}\right]_{i,j=0}^{\nu}, \quad \Sigma_{\nu,\mathrm{pre}}(\gamma) := \gamma^2\left[\frac{1}{2\nu+1-i-j}\right]_{i,j=0}^{\nu}. \tag{7.14}$$

$\Sigma_{\nu,\mathrm{pre}}(\gamma)$ is a positive definite Hilbert matrix and thus admits a closed-form Cholesky decomposition [e.g., via 3, 41, 131].

This extraction of the Δt_n-dependence from the transition matrix $\Phi_\nu(\Delta t_n)$ and $\Sigma_\nu(\Delta t_n,)$ is crucial for stabilising the extrapolation as seen in the following examples:

Example 7.2 (Conditioning of the (preconditioned) process noise covariance matrix). We investigate the effect of the coordinate change on the conditioning of the process noise covariance matrix $\Sigma_\nu(\Delta t, \gamma)$.

Table 7.1: Conditioning of the (preconditioned) process noise covariance matrix. A lightning (↯) marks NaN's, which occur when attempting to compute the logarithm of a negative number – numerically, the matrix is indefinite. "Prec." means the preconditioner, "Nord." represents the alternative (which is the Nordsieck representation), and "Noth." is the absence of preconditioning. All values are in $\log_{10}$-basis.

ν	Condition number			Ratio			Smallest eigenvalue		
	Prec.	Nord.	Noth.	Prec.	Nord.	Noth.	Prec.	Nord.	Noth.
1	**1.3**	**1.3**	9.1	**0.5**	**0.5**	8.5	**-1.2**	-5.2	-13.1
3	**4.2**	4.3	28.9	**0.8**	1.3	26.4	**-4.0**	-9.1	↯
5	**7.2**	7.6	43.7	**1.0**	2.3	45.2	**-7.0**	-14.1	↯
7	**10.2**	11.0	57.3	**1.2**	3.4	64.6	**-10.0**	-19.8	↯
9	**13.2**	14.5	68.5	**1.3**	4.5	84.4	**-13.0**	-25.9	↯
11	**16.2**	17.4	79.9	**1.4**	5.6	104.6	**-16.0**	↯	↯

We set $\gamma = 1$ because scalar multiplication does not affect the condition number of a matrix. We choose $\Delta t_n = 10^{-4}$, which is a realistic scenario for an IVP solver, and vary the number of derivatives ν. We compute three quantities and display them in Table 7.1. Values are displayed in $\log_{10}$ basis and rounded to a single decimal.

The left column of Table 7.1 contains condition numbers of the process noise covariance matrices, which should be as small as possible. The middle column of Table 7.1 contains the ratios of the biggest and smallest entries in $\Sigma_\nu(\Delta t = 10^{-4}, \gamma = 1)$,

$$\rho := \max_{ij} \left\{\Sigma_\nu(\Delta t = 10^{-4}, \gamma = 1)\right\}_{ij} / \min_{ij} \left\{\Sigma_\nu(\Delta t = 10^{-4}, \gamma = 1)\right\}_{ij}. \quad (7.15)$$

All elements in $\Sigma_\nu(\Delta t = 10^{-4}, \gamma = 1)$ are positive, and the ratio should be as small as possible. The right column of Table 7.1 contains the smallest eigenvalues of $\Sigma_\nu(\Delta t = 10^{-4}, \gamma = 1)$, which should be as large as possible. We evaluate all three quantities before and after the coordinate change. We also compare the preconditioner to an alternative (the Nordsieck representation, discussed below).

The "best" values (the smallest for the condition number and the ratio, the largest for the minimal eigenvalue) are bold-faced—they all use the proposed coordinate change. The preconditioner improves all quantities significantly.

Example 7.3 (Conditioning of the Cholesky factors of the extrapolated covariance). We evaluate the condition numbers of the Cholesky factors of the extrapolated

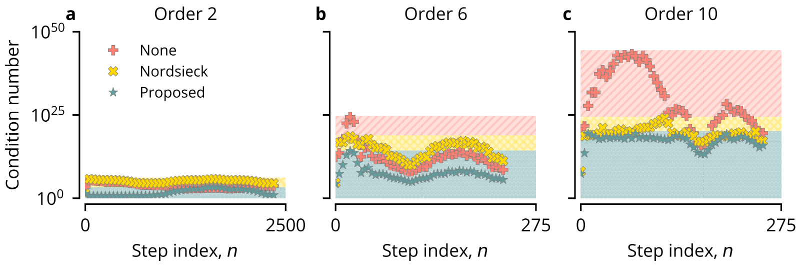

Figure 7.2: Conditioning of the Cholesky factors of the extrapolated covariance. Comparison of no preconditioning (red plus), Nordsieck representations (yellow cross) and the proposed coordinate change (blue star). The discrepancy between the maximal condition numbers is red (None vs Nordsieck) and yellow (Nordsieck vs Proposed). The range between 0 and the maximum condition number of the proposal is blue.

covariances "as seen by the solver". We simulate the Lotka–Volterra model [110], a nonlinear, two-dimensional IVP, with adaptive step-size selection and tolerance 10^{-4}. The solver employs a time-varying output scale and uses all recommendations made in this chapter.

At each extrapolation, we evaluate the condition number of the extrapolated covariance matrix with and without a coordinate change. We also compare to the same alternative preconditioner as in Example 7.2. We run the simulation with $\nu \in \{2, 6, 10\}$ and display the results in Figure 7.2. The maximum condition number of the extrapolation over the whole simulation decides the success or failure of the algorithm.

Figure 7.2 shows two phenomena: First, both preconditioners improve the condition number of the extrapolated covariance significantly. Second, the proposed change consistently leads to a lower condition number than the alternative.

The coordinate change T_ν is a preconditioner because it improves the conditioning of the extrapolated covariance matrix and its generalised Cholesky factors. More specifically, $\Phi_{\nu,\text{pre}}$ and $\Sigma_{\nu,\text{pre}}(\gamma)$ describe the transition of a stochastic process $Y_{\text{pre}}(t_n) := T_\nu(\Delta t_n)Y(t_n)$,

$$p(Y_{\text{pre}}(t_{n+1}) \mid Y_{\text{pre}}(t_n), \gamma) = \mathcal{N}(\Phi_{\nu,\text{pre}} Y_{\text{pre}}(t_n), \Sigma_{\nu,\text{pre}}(\gamma)). \tag{7.16}$$

Equation (7.16) is Equation (3.6) without a Δt_n-dependence. This lack of dependence on Δt_n in the system matrices improves the conditioning of the covariance matrix of each extrapolation. $Y_{\text{pre}}(t_n)$ and $Y(t_n)$ are deterministic, linear transformations of each other thus marginal distributions are accessible at each time step. For example,

$$
\begin{aligned}
p(Y_{\mathrm{pre}}(t_n) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) \\
= \mathcal{N}(T_\nu(\Delta t_n)^{-1} m_{n|n}, T_\nu(\Delta t_n)^{-1} C_{n|n}(\gamma) T_\nu(\Delta t_n)^{-1})
\end{aligned}
\tag{7.17}
$$

holds. The “reverse” marginalisation reduces to the multiplication of mean and covariance with $T_\nu(\Delta t_n)$ instead of its inverse.

If $(C_{n|n}(\gamma))^{1/2}$ is a square generalised Cholesky factor of $C_{n|n}(\gamma)$, then the product $T_\nu(\Delta t_n)^{-1}(C_{n|n}(\gamma))^{1/2}$ is a square generalised Cholesky factor of the covariance matrix $T_\nu(\Delta t_n)^{-1} C_{n|n}(\gamma) T_\nu(\Delta t_n)^{-1}$. The coordinate change also preserves upper and lower triangularity and the sign of the diagonal elements of generalised Cholesky factors.

Algorithm 7.4 (Preconditioned extrapolation)**.** Compute the extrapolation with a preconditioner as follows. Let $n \in \{0, ..., N-1\}$ and assume a Cholesky parametrisation of $p(Y(t_n) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) = \mathcal{N}(m_{n|n}, C_{n|n}(\gamma))$.

1. Assemble the coordinate change $T_\nu(\Delta t_n)$ and its inverse according to Equation (7.11) and Equation (7.12), respectively.

2. Compute the marginal distribution

$$
p(Y_{\mathrm{pre}}(t_n) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) = \mathcal{N}(m_{n|n,\,\mathrm{pre}}, C_{n|n,\,\mathrm{pre}}(\gamma)) \tag{7.18}
$$

according to Equation (7.17).

3. Call Algorithm 4.2 with inputs

$$
m_{\mathrm{in}} = m_{n|n,\,\mathrm{pre}} \tag{7.19a}
$$
$$
(C_{\mathrm{in}})^{1/2} = (C_{n|n,\,\mathrm{pre}}(\gamma))^{1/2} \tag{7.19b}
$$
$$
A_{\mathrm{cond}} = \Phi_{\nu,\mathrm{pre}} \tag{7.19c}
$$
$$
b_{\mathrm{cond}} = 0 \tag{7.19d}
$$
$$
(C_{\mathrm{cond}})^{1/2} = (\Sigma_{\nu,\mathrm{pre}}(\gamma))^{1/2} \tag{7.19e}
$$

Algorithm 4.2 returns $(m_{\mathrm{out}}, (C_{\mathrm{out}})^{1/2}, A_{\mathrm{rev}}, b_{\mathrm{rev}}, (C_{\mathrm{rev}})^{1/2})$. Assign the parameters

$$\begin{aligned} m_{n+1|n,\,\mathrm{pre}} &= m_{\mathrm{out}} && \text{(7.20a)}\\ (C_{n+1|n,\,\mathrm{pre}}(\gamma))^{1/2} &= (C_{\mathrm{out}})^{1/2} && \text{(7.20b)}\\ \Lambda_{n|n+1,\mathrm{pre}} &= A_{\mathrm{rev}} && \text{(7.20c)}\\ \lambda_{n|n+1,\mathrm{pre}} &= b_{\mathrm{rev}} && \text{(7.20d)}\\ (\Xi_{n|n+1,\mathrm{pre}}(\gamma))^{1/2} &= (C_{\mathrm{rev}})^{1/2} && \text{(7.20e)} \end{aligned}$$

to obtain a Cholesky parametrisation of the marginal distribution

$$\begin{aligned} &p(Y_{\mathrm{pre}}(t_{n+1}) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma)\\ &\qquad = \mathcal{N}(m_{n+1|n,\,\mathrm{pre}}, C_{n+1|n,\,\mathrm{pre}}(\gamma)) \end{aligned} \tag{7.21}$$

and a Cholesky parametrisation of the conditional distribution

$$\begin{aligned} &p(Y_{\mathrm{pre}}(t_n) \mid Y_{\mathrm{pre}}(t_{n+1}), \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma)\\ &\qquad = \mathcal{N}(\Lambda_{n|n+1,\mathrm{pre}} Y_{\mathrm{pre}}(t_{n+1}) + \lambda_{n|n+1,\mathrm{pre}}, \Xi_{n|n+1,\mathrm{pre}}(\gamma)). \end{aligned} \tag{7.22}$$

4. Reverse the coordinate change by marginalising

$$\begin{aligned} &p(Y(t_{n+1}) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma)\\ &\qquad = \mathcal{N}(T_\nu(\Delta t_n) m_{n+1|n+1}, T_\nu(\Delta t_n) C_{n+1|n+1}(\gamma) T_\nu(\Delta t_n)^\top) \end{aligned} \tag{7.23}$$

where the marginal generalised Cholesky factor inherits triangularity as before. Implement

$$\begin{aligned} \Lambda_{n|n+1} &= T_\nu(\Delta t_n) \Lambda_{n|n+1,\mathrm{pre}} T_\nu(\Delta t_n)^{-1} && \text{(7.24a)}\\ \lambda_{n|n+1} &= T_\nu(\Delta t_n) \lambda_{n|n+1,\mathrm{pre}} && \text{(7.24b)}\\ \Xi_{n|n+1}(\gamma) &= T_\nu(\Delta t_n) \Xi_{n|n+1,\mathrm{pre}}(\gamma) T_\nu(\Delta t_n)^\top && \text{(7.24c)} \end{aligned}$$

to obtain the backward transition

$$\begin{aligned} &p(Y(t_n) \mid Y(t_{n+1}), \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma)\\ &\qquad = \mathcal{N}(\Lambda_{n|n+1} Y(t_{n+1}) + \lambda_{n|n+1}, \Xi_{n|n+1}(\gamma)). \end{aligned} \tag{7.25}$$

Optionally, store the preconditioned backward transitions instead of reversing the coordinate change, and (re)apply the preconditioner during the backward pass.

Return the extrapolation and the backward transition (Equations (7.23) and (7.25)).

Assumption 7.1 implies that $C_{n|n}(\gamma)$ is of the form

$$C_{n|n}(\gamma) = \gamma^2 C_{n|n}(1). \tag{7.26}$$

(The Taylor series estimation initialises this matrix to be zero, and Gaussian conditioning inherits this form due to the same argument that Section 7.4 applies below.) Due to the form of Algorithm 4.2, the same factorisation is inherited by

$$C_{n+1|n+1}(\gamma) = \gamma^2 C_{n|n}(1), \quad \Xi_{n|n+1}(\gamma) = \gamma^2 \Xi_{n|n+1}(1) \tag{7.27}$$

and $\Lambda_{n|n+1,\text{pre}}$ and $\lambda_{n|n+1,\text{pre}}$ are independent of γ (which is straightforward to verify by considering the corresponding expressions in conventional parametrisation).

The coordinate change allows an interpretation in terms of Taylor series, similar to what has been discussed by Chapter 6. More concretely, the state

$$Y_{\text{pre}}(t_n) = \frac{1}{\sqrt{\Delta t}} \frac{\nu!}{(\Delta t)^\nu} \left(Y^{(0)}(t_n), Y^{(1)}(t_n)\Delta t, ..., Y^{(\nu)}(t_n)\frac{\Delta t^\nu}{\nu!} \right) \tag{7.28}$$

is proportional to the normalised Taylor coefficients of $Y^{(0)}(t_n)$ while the original state $Y(t_n)$ contains the unnormalised Taylor coefficients of $Y^{(0)}(t_n)$. Normalising (and scaling) Taylor coefficients thus improves the numerical stability of probabilistic numerical solvers. The extra scaling factor removes the Δt-dependence of $\Sigma_\nu(\Delta t, \gamma)$.

Schober et al. [152] use the normalised Taylor coefficient representation, also known as the *Nordsieck representation* of $Y(t)$ (after Nordsieck [123]), to show that a specific instance of a probabilistic numerical solver is equivalent to a multi-step method with time-varying weights. Multi-step methods benefit from implementation in the Nordsieck representation [e.g. 29]. The above explanation shows how an extra scaling factor in the Nordsieck representation stabilises estimation with multiply-integrated Wiener processes for (arbitrarily) many derivatives ν and small step sizes Δt.

7.4 Correction

Section 7.1 (and Chapter 3) explained how the sequential IVP solver alternates between extrapolation and correction steps after initialisation. With initialisation and extrapolation covered by Sections 7.2 and 7.3, we consider the correction step next. In general, correction combines the linearisation techniques from Chapter 5 with the manipulation of linearly related Gaussian variables in Cholesky parametrisation from Chapter 4. To this end, define the qth unit vector e_q as the qth column of the identity matrix with $\nu + 1$ rows and columns. Assume a Cholesky parametrisation of the extrapolation

$$p(Y(t_{n+1}) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) = \mathcal{N}(m_{n+1|n}, C_{n+1|n}(\gamma)). \tag{7.29}$$

Recall the definition of the residual $\mathcal{R}_{f,n}$ from Equation (7.3) or Section 3.4 respectively. For example, the logistic differential equation $\frac{\mathrm{d}y(t)}{\mathrm{d}t} = y(t)(1-y(t))$ would correspond to the residual

$$\mathcal{R}_{f,n} = Y^{(1)}(t_n) - Y^{(0)}(t_n)(1 - Y^{(0)}(t_n)). \tag{7.30}$$

The remainder of this section provides detailed instructions for the correction step in a sequential solver for nonlinear IVPs.

Affine problems can be solved with the algorithms in Chapter 4. If the vector field of the differential equation f is not affine, we linearise it with a Taylor approximation or statistical linear regression around the current best guess of the IVP solution: the marginal distribution

$$p(Y^{(0)}(t_{n+1}) \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) = \mathcal{N}(e_0 m_{n+1|n}, e_0 C_{n+1|n}(\gamma) e_0^\top), \tag{7.31}$$

which derives from the extrapolated state. The generalised Cholesky factor of $e_0 C_{n+1|n}(\gamma) e_0^\top$ results from that of $C_{n+1|n}(\gamma)$ by QR-decomposing $(C_{n+1|n}(\gamma))^{1/2} e_0^\top$. The linearised vector field becomes either one of

$$\begin{aligned}
&\text{TS0:} && f(z) \approx b, && b \in \mathbb{R} && (7.32\text{a})\\
&\text{TS1:} && f(z) \approx az + b, && b \in \mathbb{R} && (7.32\text{b})\\
&\text{SLR0:} && f(z) \approx b, && b \sim \mathcal{N}(\tilde{b}_{\text{cond}}, \tilde{C}_{\text{cond}}) && (7.32\text{c})\\
&\text{SLR1:} && f(z) \approx az + b, && b \sim \mathcal{N}(\tilde{b}_{\text{cond}}, \tilde{C}_{\text{cond}}) && (7.32\text{d})
\end{aligned}$$

Equation (7.32) abbreviates first- and zeroth-order Taylor series approximation (TS1, TS0) and first- and zeroth-order statistical linear regression (SLR1, SLR0). Each of the parameters a, b, $\tilde{b}_{\text{cond}}$, $\tilde{C}_{\text{cond}}$ result from either Algorithm 5.1, or Algorithm 5.2, Algorithm 5.6, Algorithm 5.7, or Algorithm 5.8; the choice between the methods depends on preferences for Taylor linearisation or statistical linear regression, zeroth-/first-order methods, and the availability of Jacobians. Taylor-linearisation yields a deterministic transformation, whereas statistical linear regression yields a stochastic transformation. Both setups require different routines, so the implementation for statistical and Taylor-based approaches are explained separately. We begin with Taylor-linearisation and conclude with statistical linear regression.

Combining the definition of the differential equation residual (Equation (7.3)) with that of zeroth- or first-order Taylor linearisation (Equation (7.32)) yields an approximation of the type

$$\mathcal{R}_{f,n+1} \approx (e_1 - a e_0) Y(t_{n+1}) - b, \tag{7.33}$$

which is affine in $Y(t_{n+1})$. For a zeroth-order approximation, a is zero. The correction step thus reduces to calling Algorithm 4.5:

Algorithm 7.5 (Correction, Taylor approximation)**.** To approximate the conditional distribution, proceed as follows:

1. Linearise the vector field according to Equation (7.32) with one of the Taylor approximations (TS0 or TS1).

2. Call Algorithm 4.5 with the inputs

$$m_{\text{in}} = m_{n+1|n} \tag{7.34a}$$

$$(C_{\text{in}})^{1/2} = (C_{n+1|n}(\gamma))^{1/2}, \tag{7.34b}$$

$$A_{\text{cond}} = e_1 - a e_0, \tag{7.34c}$$

$$b_{\text{cond}} = -b. \tag{7.34d}$$

   Algorithm 4.5 returns parameters $(m_{\text{out}}, (C_{\text{out}})^{1/2}, A_{\text{rev}}, b_{\text{rev}}, (C_{\text{rev}})^{1/2})$. Assign

$$m_{n+1|n+1} = b_{\text{rev}}, \quad (C_{n+1|n+1}(\gamma))^{1/2} = (C_{\text{rev}})^{1/2}, \tag{7.35}$$

   discard A_{rev}, and assign

$$s_{n+1} = m_{\text{out}}, \quad (S_{n+1}(\gamma))^{1/2} = (C_{\text{out}})^{1/2}. \tag{7.36}$$

Return the resulting approximate Cholesky parametrisation of the conditional

$$p(Y(t_{n+1}) \mid \mathcal{R}_{f,0:n+1} = 0, \mathcal{R}_{y_0} = 0, \gamma) \approx \mathcal{N}(m_{n+1|n+1}, C_{n+1|n+1}(\gamma)) \tag{7.37}$$

as well as a Cholesky parametrisation of the marginal

$$p(\mathcal{R}_{f,n+1} \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) \approx \mathcal{N}(s_{n+1}, S_{n+1}(\gamma)). \tag{7.38}$$

The linearisation parameters do not depend on γ. If $C_{n+1|n}(\gamma) = \gamma^2 C_{n+1|n}(1)$, the conditional and marginal means do not depend on γ, and the conditional and marginal covariances satisfy $C_{n+1|n+1}(\gamma) = \gamma^2 C_{n+1|n+1}(1)$ and $S_{n+1}(\gamma) = \gamma^2 S_{n+1}(1)$. If the differential equation is affine and for first-order Taylor-linearisation, Algorithm 7.5 yields the exact posterior. The same holds if the differential equation is constant and for zeroth-order Taylor linearisation.

Other than Taylor linearisation, statistical linear regression quantifies the linearisation error in an error covariance; as a result, linearised residuals are stochastic transformations,

$$\mathcal{R}_{f,n+1} \approx (e_1 - a e_0) Y(t_{n+1}) - b, \quad b \sim \mathcal{N}(\tilde{b}_{\text{cond}}, \tilde{C}_{\text{cond}}). \tag{7.39}$$

For zeroth-order approximation, a is zero. The stochastic nature of this transformation requires replacing Algorithm 4.5 with Algorithm 4.2 in Algorithm 7.5:

Algorithm 7.6 (Correction, statistical linear regression)**.** To approximate the conditional distribution, proceed as follows:

1. Linearise the vector field according to Equation (7.32) with statistical linear regression (SLR0 or SLR1)

2. Call Algorithm 4.2 with the inputs

$$m_{\text{in}} = m_{n+1|n}, \tag{7.40a}$$
$$(C_{\text{in}})^{1/2} = (C_{n+1|n}(\gamma))^{1/2}, \tag{7.40b}$$
$$A_{\text{cond}} = e_1 - a e_0, \tag{7.40c}$$
$$b_{\text{cond}} = -\tilde{b}_{\text{cond}}, \tag{7.40d}$$
$$C_{\text{cond}} = \tilde{C}_{\text{cond}}. \tag{7.40e}$$

Algorithm 4.2 returns parameters $(m_{\text{out}}, (C_{\text{out}})^{1/2}, A_{\text{rev}}, b_{\text{rev}}, (C_{\text{rev}})^{1/2})$. Assign

$$m_{n+1|n+1} = b_{\text{rev}}, \quad (C_{n+1|n+1}(\gamma))^{1/2} = (C_{\text{rev}})^{1/2}, \tag{7.41}$$

discard A_{rev}, and assign

$$s_{n+1} = m_{\text{out}}, \quad (S_{n+1}(\gamma))^{1/2} = (C_{\text{out}})^{1/2}. \tag{7.42}$$

Return the resulting Cholesky parametrisation of the conditional

$$p(Y(t_{n+1}) \mid \mathcal{R}_{f,0:n+1} = 0, \mathcal{R}_{y_0} = 0, \gamma) \approx \mathcal{N}(m_{n+1|n+1}, C_{n+1|n+1}(\gamma)) \tag{7.43}$$

as well as a Cholesky parametrisation of the marginal

$$p(\mathcal{R}_{f,n+1} \mid \mathcal{R}_{f,0:n} = 0, \mathcal{R}_{y_0} = 0, \gamma) \approx \mathcal{N}(s_{n+1}, S_{n+1}(\gamma)). \tag{7.44}$$

In contrast to Taylor approximation, all linearisation parameters depend on γ, which implies that the marginal and conditional means depend on γ. However, in the remainder of this manuscript, we sometimes ignore this – the results in Section 7.5 would not apply to statistical linear regression without pretending that $m_{n+1|n}$ and

s_{n+1} are independent of γ, among other things. Where statistical correctness is crucial, do not combine statistical linear regression with the following calibration of γ.

Algorithm 7.5 and Algorithm 7.6 summarise how to perform the correction step in a probabilistic numerical IVP solver. Surrounding both algorithms and the extrapolation algorithm in Algorithm 7.4, we discussed whether and how the parameters depend on γ. This discussion is essential for the upcoming sections in this chapter. The remainder of this chapter discusses the automatic calibration of γ and combines it with all previous algorithms in two templates for sequential IVP solvers.

7.5 Calibrating the output scale

All previous sections concerned state estimation of the IVP solution assuming an unknown but fixed parameter γ. The following two sections discuss estimating γ. To this end, recall the output scale γ of the integrated Wiener process from Chapter 3 and the Mahalanobis norm $\|x\|_M^2 = \|((M)^{1/2})^\top x\|_2^2$ from Section 4.6. Let $\det[M]$ be the determinant of a matrix M. This section proceeds with deriving a maximum-likelihood estimate of γ before the next section discusses time-varying models for γ.

Selecting an unknown parameter that maximises the likelihood of the constraints,

$$
\begin{aligned}
\mathrm{MLE}(\gamma) &:= \operatorname*{arg\,max}_{\gamma\in\mathbb{R}} p(\mathcal{R}_{y_0} = 0, \mathcal{R}_{f,0:N} = 0 \mid \gamma), &\text{(7.45a)}\\
&= \operatorname*{arg\,max}_{\gamma\in\mathbb{R}} \log p(\mathcal{R}_{y_0} = 0, \mathcal{R}_{f,0:N} = 0 \mid \gamma), &\text{(7.45b)}\\
&= \operatorname*{arg\,min}_{\gamma\in\mathbb{R}} \{-\log p(\mathcal{R}_{y_0} = 0, \mathcal{R}_{f,0:N} = 0 \mid \gamma)\}, &\text{(7.45c)}
\end{aligned}
$$

is called *maximum-likelihood estimation* and is one of the most common approaches to parameter estimation in probabilistic models. If instead of the likelihood of the constraints itself, an approximation of the likelihood of the constraints is optimised, we call the procedure *quasi-maximum-likelihood estimation*. The sequential factorisation of the likelihood of the constraints allows estimating the quasi-maximum-likelihood estimate efficiently: To see this, combine the Gaussian approximation of the marginal distributions from Algorithm 7.5 or Algorithm 7.6 with the sequential factorisation in Chapter 3 to obtain

$$
\begin{aligned}
\mathrm{MLE}(\gamma) &= \operatorname*{arg\,min}_{\gamma\in\mathbb{R}} \{-\log p(\mathcal{R}_{y_0} = 0, \mathcal{R}_{f,0:N} = 0 \mid \gamma)\} &\text{(7.46a)}\\
&\approx \operatorname*{arg\,min}_{\gamma\in\mathbb{R}} \left\{ \sum_{n=0}^{N} \left(\|s_n\|^2_{S_n(\gamma)^{-1}} + \log\left[\det\left[S_n(\gamma)\right]\right]\right) \right. &\\
&\qquad \left. + \|s_{y_0}\|^2_{S_{y_0}(\gamma)^{-1}} + \log\left[\det\left[S_{y_0}(\gamma)\right]\right] \right\}. &\text{(7.46b)}
\end{aligned}
$$

All parameters in the above equation are byproducts of the initialisation and correction steps. Whether or not the summands involving s_{y_0} and $S_{y_0}(\gamma)$ appear in this expression depends on choices made at initialisation: For initialisation via Gaussian conditioning, the terms are part of the likelihood of the constraints; for initialisation via Taylor-series estimation, computing the marginal distribution is possible but usually skipped in practice. In this case, the terms are not present in the sum. In the remainder of this chapter, we include the marginal distribution of the initial state in the exposition; adapting the formulas to the case of Taylor-series initialisation is relatively straightforward and left to the reader. In either case, the minimisation problem in Equation (7.46) can be solved in closed form:

Recall Assumption 7.1 from the introduction in Section 7.1. Assumption 7.1 together with the factorisation of all relevant covariance matrices into

$$\Sigma_\nu(\Delta t_n, \gamma) = \gamma^2 \Sigma_\nu(\Delta t_n, 1) \tag{7.47a}$$

$$C_{n|n}(\gamma) = \gamma^2 C_{n|n}(1), \qquad n = 0, ..., N, \tag{7.47b}$$

$$C_{n+1|n}(\gamma) = \gamma^2 C_{n+1|n}(1), \qquad n = 0, ..., N-1, \tag{7.47c}$$

$$\Xi_{n|n+1}(\gamma) = \gamma^2 \, \Xi_{n|n+1}(1), \qquad n = 1, ..., N, \tag{7.47d}$$

$$S_n(\gamma) = \gamma^2 \, S_n(1), \qquad n = 0, ..., N. \tag{7.47e}$$

(unless statistical linear regression is involved, in which case we resort to pretending that Equation (7.47) is true) is the reason that the quasi-maximum-likelihood problem can be solved in closed form:

1. Equation (7.47) implies that Equation (7.46) reads

$$\begin{aligned}\mathrm{MLE}(\gamma) \approx \arg\min_{\gamma \in \mathbb{R}} \frac{1}{\gamma^2} \Bigg\{ \sum_{n=0}^{N} \left(\|s_n\|^2_{S_n(1)^{-1}} + \log\left[\det\left[S_n(1)\right]\right] \right) \\ + \|s_{y_0}\|^2_{S_{y_0}(1)^{-1}} + \log\left[\det\left[S_{y_0}(1)\right]\right] \Bigg\}.\end{aligned} \tag{7.48}$$

   First-order optimality conditions imply [25, 163],

$$\mathrm{MLE}(\gamma)^2 = \frac{1}{N+2} \left(\|s_{y_0}\|^2_{S_{y_0}(1)^{-1}} + \sum_{n=0}^{N} \|s_n\|^2_{S_n(1)^{-1}} \right). \tag{7.49}$$

   Each summand in Equation (7.49) is a byproduct of Algorithm 7.5 and Algorithm 7.6, respectively, the initialisation step. All quantities can be computed during the forward pass and in a constant number of iterations.

2. All covariance matrices in Equation (7.47) are the product of γ^2 and a factor

independent of the output scale. Therefore, we may choose an appropriate output scale after the simulation and scale all covariance matrices without losing statistical correctness.

In other words, Algorithm 7.7 (below) is a valid strategy for calibrating γ [163].

Algorithm 7.7 (Maximum-likelihood estimation of the output scale)**.** Under Assumption 7.1, and if a Taylor approximation is used for linearisation, calibrate the output scale γ as follows:

1. Choose $\gamma = 1$. Estimate the backward transition densities and the conditional distributions with the algorithms in Sections 7.2 to 7.4. Store the parameters s_{y_0}, $S_{y_0}(1)$, $\{s_n\}_{n=0}^N$, and $\{S_n(1)\}_{n=0}^N$.
2. Estimate γ by evaluating Equation (7.49).
3. Scale the covariance matrices with the estimated output scale according to Equation (7.47).

The reason behind the validity of Algorithm 7.7 is Equation (7.47), which holds because of Assumption 7.1 and because the constraints are approximated as affine transformations. If statistical linear regression is used for linearisation instead of a Taylor series, Algorithm 7.7 can still be applied but requires pretending that Equation (7.47) is true (which it is generally not).

Remark 7.8. If the constraints are corrupted by (pairwise independent) additive Gaussian noise with covariance matrices $\{R_n(\gamma)\}_{n=0}^N$, Algorithm 7.7 remains true as long as $R_n(\gamma) = \gamma^2 R_n(1)$ holds for all $n = 0, \ldots, N$. Otherwise, calibration is more complicated.

In conclusion, the final unknown parameter to be discussed in this manuscript, the output scale γ, has also been calibrated automatically now: maximum-likelihood-estimating the output scale is possible in closed form and during the forward pass because all algorithm steps preserve the factorisation $M(\gamma) = \gamma^2 M(1)$ for arbitrary covariance matrix M. This technique will resurface in upcoming chapters about vector-valued and spatiotemporal models with minimal modification. For problems that fit the assumption of a time-constant output scale, Algorithm 7.7 is, therefore, a valid strategy. However, some problems are incompatible with such an assumption, and alternatives are discussed in the next section.

7.6 Time-varying output-scale

Many problems are suitable for applying a constant-time output scale and Algorithm 7.7. However, in some scenarios, time-varying output scales are more appropriate than constant output scales. For example, consider the linear differential equation $\frac{\mathrm{d}y(t)}{\mathrm{d}t} = 2y(t)$, $y(0) = 1$, which is solved by $y(t) = \exp(2t)$. This example has been discussed in the context of Figure 3.2. The exponential function grows too quickly for a single scale to describe its magnitude; assuming a constant γ is too restrictive and leads to inaccurate estimation. To solve this problem, we need to refine the model for γ. Recall the (approximate) representation of a time-varying output scale $\gamma : \mathbb{R} \to \mathbb{R}$ as a step-function

$$\gamma(t) := \begin{cases} \gamma_0 & \text{if } t \in (-\infty, t_0], \\ \gamma_n & \text{if } t \in (t_n, t_{n+1}],\ n = 0, ..., N-1, \\ \gamma_N & \text{if } t \in (t_N, \infty]. \end{cases} \tag{7.50}$$

from Chapter 3. For a piecewise-constant output scale, the prior evolves as

$$p(Y(t_{n+1}) \mid Y(t_n), \gamma_n) = \mathcal{N}(\Phi_\nu(\Delta t_n)Y(t_n), \Sigma_\nu(\Delta t_n, \gamma_n)). \tag{7.51}$$

Except for using γ_n instead of γ, Equation (7.51) is identical to Equation (3.6). For a piecewise-constant output scale, we can apply a technique similar to that from Section 7.5 with only one additional assumption.

Assumption 7.9. *Assume that the initial covariance factorises as $C_0(\gamma_0) = \gamma_0^2 C_0(1)$.*

Assumption 7.9 is almost identical to Assumption 7.1, the only difference is the notation of a time-varying output scale. In general, Assumption 7.9 is mild.

Assumption 7.10. *To implement the estimator for $\gamma_{0:N}$, assume that at each time-point, the previous estimate is error-free; that is, assume that $C_{n|n}(\gamma_{0:n}) = 0$ holds.*

Assumption 7.10's only role is to derive a formula for the parameter estimate of γ_n. It does not affect the algorithms in Sections 7.2 to 7.4. While it may seem restrictive, assuming that the previous step has been error-free is a common assumption for step-size adaptation in non-probabilistic solvers [152], so implementing a similar assumption for calibrating the probabilistic numerical solver seems reasonable. With both assumptions satisfied, local quasi-maximum-likelihood estimates can be evaluated in closed form.

Assumptions 7.9 and 7.10 contribute to the calibration of $\gamma_{0:N}$ as follows. Let $n \in \{1, ..., N-1\}$. Recall from Section 3.5 that the process noise covariance matrix decomposes into $\Sigma_\nu(\Delta t_n, \gamma_n) = \gamma_n^2 \Sigma_\nu(\Delta t_n, 1)$. This decomposition, together with

Assumption 7.9 and Assumption 7.10, implies that when Taylor-linearisation is used,

$$S_n(\gamma_n) = \gamma_n^2 S_n(1), \quad S_n(1) = (e_1 - ae_0)\Sigma_\nu(\Delta t_n, 1)(e_1 - ae_0)^\top, \tag{7.52}$$

holds; thus, the quasi-maximum-likelihood estimate of γ_n is [152]

$$\mathrm{MLE}(\gamma_n)^2 \approx \|s_n\|^2_{S_n(1)^{-1}}. \tag{7.53}$$

If the initial γ_0 shall be calibrated, use

$$\mathrm{MLE}(\gamma_0)^2 \approx \frac{1}{2}\left(\|s_0\|^2_{S_0(1)^{-1}} + \|s_{y_0}\|^2_{S_{y_0}(1)^{-1}}\right), \tag{7.54}$$

Natural modifications apply for initialisation via Taylor-series estimation; details are left to the reader. Algorithm 7.11 summarises the procedure.

Algorithm 7.11 (Maximum-likelihood estimation of the time-varying output scale)**.** Let $n = 0, ..., N$. Estimate the time-varying output scale as follows.

1. Before each extrapolation, apply Assumption 7.10 and estimate the output scale γ_n according to Equation (7.53) and Equation (7.54), respectively. Afterwards, ignore Assumption 7.10.
2. Extrapolate with Algorithm 7.4 but use Equation (7.51) with the estimated output scale instead of Equation (3.6). Other than using a different output scale, Algorithm 7.4 does not change.

Then, perform the correction as usual and proceed to the next iteration.

Note how in Item 1 of Algorithm 7.11, Assumption 7.10 is ignored as soon a calibrated output scale is available.

Estimating a time-varying output scale is similar to estimating a time-constant one but requires more heuristics and approximations. But the reward is that some problems can now be solved significantly more accurately than with a constant scale; for example, the linear differential equation in Figure 3.2 benefits greatly from a time-varying model. Some of the additional assumptions may seem overly restrictive at first, for example, Assumption 7.10. But not only are variants of this assumption typical for estimating local errors in non-probabilistic numerical IVP solvers [e.g. 76, Chapter II.4]; Algorithm 7.11 has been used successfully for many years since its introduction by Schober et al. [152] and subsequent refinement by Bosch et al. [25]. To conclude this chapter, Section 7.7 combines the algorithms in templates for probabilistic numerical IVP solvers that guide the implementation of these methods.

7.7 Conclusion

This chapter has presented the steps involved in implementing a sequential probabilistic numerical IVP solver: initialisation, extrapolation, correction, and calibration. The three algorithms below summarise the main versions of probabilistic numerical IVP solvers. The first applies to a calibration-free solver. The second refers to quasi-maximum-likelihood-estimating a constant output scale and elaborates on Algorithm 7.7. The third corresponds to time-varying output scales and extends Algorithm 7.11. It is titled “dynamic IVP solver” because the model (including the output scale) changes dynamically over time and with the IVP solution.

Algorithm 7.12 (IVP solver, calibration-free)**.** Assume an initial value problem, a ν-times integrated Wiener process prior with a constant output scale γ, and (optionally) a point set $t_0, ..., t_N$.

1. Initialise with Taylor-series estimation routines in Chapter 6, if available; otherwise, use Gaussian conditioning.
2. Then, for each $n = 0, ..., N - 1$, alternate the steps:
   (a) Extrapolate with Algorithm 7.4 using the user-provided γ.
   (b) Correct with Algorithm 7.5 or Algorithm 7.6.
   (c) Optionally, estimate the local error and adapt the step-size [25, 152].
3. Return the results.

Use Algorithm 7.12 for IVP-parameter-estimation, in which case the output-scale can be optimised jointly with the IVP parameters [165].

If γ is unknown but time-constant, benefit from maximum-likelihood estimation:

Algorithm 7.13 (IVP solver, maximum-likelihood)**.** Assume an initial value problem, a ν-times integrated Wiener process prior with a constant output scale, and (optionally) a point set $t_0, ..., t_N$.

1. Initialise with Taylor-series estimation routines in Chapter 6, if available; otherwise, use Gaussian conditioning.
2. Set an initial estimate of the output scale as

$$\mathrm{MLE}(\gamma)_0 \approx \frac{1}{\sqrt{2}} \left(\|s_0\|^2_{S_0(1)^{-1}} + \|s_{y_0}\|^2_{S_{y_0}(1)^{-1}} \right)^{1/2}. \tag{7.55}$$

If the full Taylor series has been initialised exactly, replace s_{y_0} and $S_{y_0}(1)$ accordingly or skip this step and initialise the maximum-likelihood estimate with the user-specified parameter (or one).

3. Then, for each $n = 0, ..., N-1$, alternate the steps:
   (a) Extrapolate with Algorithm 7.4 using $\gamma = 1$.
   (b) Correct with Algorithm 7.5 or Algorithm 7.6.
   (c) Update the maximum-likelihood estimate of the output scale as

$$\mathrm{MLE}(\gamma)_{n+1} \approx \left(\frac{n}{n+1} \mathrm{MLE}(\gamma)_n^2 + \frac{1}{n+1} \|s_{n+1}\|^2_{S_{n+1}(1)^{-1}} \right)^{1/2}. \tag{7.56}$$

   This update can be implemented in Cholesky arithmetic using the techniques from Chapter 4.
   (d) Optionally, estimate the local error and adapt the step-size [25, 152].
4. Scale all covariance matrices with $\mathrm{MLE}(\gamma) := \mathrm{MLE}(\gamma)_N$ (Equation (7.47)).

When using this algorithm, be aware that Algorithm 7.13 relies on assumptions only satisfied by Taylor linearisation and that statistical linear regression does not fulfil this framework unless certain dependencies are ignored.

Algorithm 7.14 (Dynamic IVP solver)**.** Assume an initial value problem, a ν-times integrated Wiener process prior with a prior output scale, and (optionally) a point set $t_0, ..., t_N$.

1. Initialise with Taylor-series estimation routines in Chapter 6, if available; otherwise, use Gaussian conditioning.
2. Initiate a current estimate of the output scale as

$$\mathrm{MLE}(\gamma)_0 \approx \frac{1}{\sqrt{2}} \left(\|s_0\|_{S_0(1)^{-1}} + \|s_{y_0}\|_{S_{y_0}(1)^{-1}} \right). \tag{7.57}$$

   with natural modifications for full Taylor series (or skip this and choose the user-specified output scale).
3. Then, for each $n = 0, ..., N-1$, alternate the steps:

(a) Estimate the local output scale γ_n with the maximum-likelihood estimator in Equation (7.53).

(b) Extrapolate with Algorithm 7.4 using the tuned output scale γ_n.

(c) Correct with Algorithm 7.5 or Algorithm 7.6.

(d) Optionally, estimate the local error and adapt the step-size [25, 152].

Use the dynamic IVP solver for problems that follow time-varying scales, for example, linear equations or the stiff version of the van-der-Pol system [72].

The main difference between the templates in Algorithms 7.13 to 7.14 and early works on probabilistic numerical IVP solvers that implemented IVP solvers as extended Kalman filters are Taylor-series initialisation, Cholesky arithmetic, and preconditioning. And these differences matter:

Example 7.15. We compute a probabilistic numerical approximation of the solution of the logistic IVP

$$\dot{x}(t) = 4x(t)(1 - x(t)), \quad x(0) = 0.15, \quad t \in [0, 2]. \tag{7.58}$$

We calibrate a time-varying output scale and solve the IVP adaptively with (relative and absolute) tolerance 10^{-5}. We employ first-order and zeroth-order Taylor linearisation and vary the number of derivatives ν.

We implement two versions of each solver. One version uses the recommendations from this chapter, including the Taylor series initialisation, preconditioning, and Cholesky arithmetic. The other version does not initialise high-order Taylor coefficients accurately, does not use a preconditioner, and implements Gaussian random variable arithmetic conventionally. Table 7.2 displays how the modifications make a drastic difference in the successful integration of IVPs.

Cholesky arithmetic and preconditioning could be used for Kalman filtering with integrated Wiener process priors; initialisation, step-size adaptation, and dynamic calibration are specific to differential equation solvers.

In summary, best practices and templates for implementing IVP solvers have been discussed in this chapter—first, separately for initialisation, extrapolation, correction, and calibration, and then put together in three holistic templates. The upcoming chapters discuss extensions to vector-valued, spatiotemporal, and boundary-value problems. The present chapter serves as the foundation for those discussions, and upcoming explanations will frequently refer back to the best practices for implementing probabilistic numerical solvers for scalar-valued IVPs.

Table 7.2: Successful approximation of an ODE solution. A "✓" indicates the successful estimation with an error below the prescribed tolerance. A "×" is a failure. "TS0" and "TS1" are zeroth- and first-order linearisation, respectively.

ν	Naive implementation		Stabilised implementation	
	TS0	TS1	TS0	TS1
2	✓	✓	✓	✓
3	✓	✓	✓	✓
4	✓	✓	✓	✓
5	×	✓	✓	✓
6	×	×	✓	✓
7	×	×	✓	✓
8	×	×	✓	✓
9	×	×	✓	✓
10	×	×	✓	✓
11	×	×	✓	✓

Chapter 8

Vector-valued problems

Contents

8.1 Introduction

The previous chapter(s) explained best practices for implementing probabilistic numerical solvers for *scalar* initial value problems (IVPs). Next, we discuss extending those algorithms to *vector-valued* problems. Even though most components will remain the same as in previous parts, the overall structure of the state-space model changes and is thus separate from scalar problems.

Let $d \in \mathbb{N}$ and assume a vector field $f : \mathbb{R}^d \to \mathbb{R}^d$. Consider an initial value problem based on an ordinary differential equation,

$$\frac{\mathrm{d}y(t)}{\mathrm{d}t} = f(y(t)), \tag{8.1}$$

constrained by the initial condition $y(0) = y_0 \in \mathbb{R}^d$. As usual, the IVP is assumed to be autonomous to simplify the notation; the same modifications as in the previous chapters apply to other types of IVPs.

Previously, the IVP has always had dimension $d = 1$. From now on, d may be

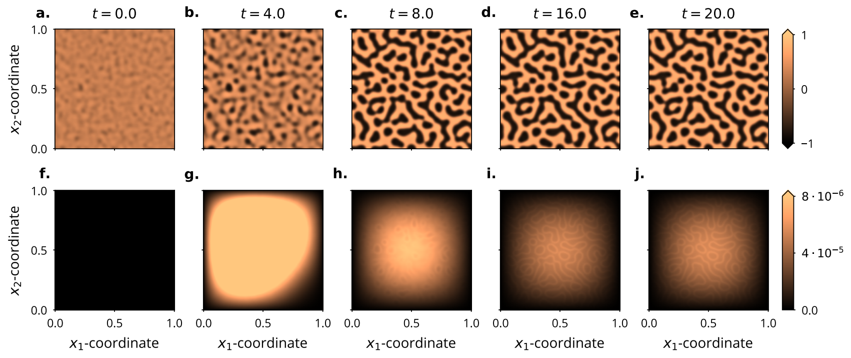

Figure 8.1: Probabilistic numerical solution of a discretised FitzHugh-Nagumo partial differential equation [5]. Marginal mean (top row) and marginal standard deviation (bottom row), from $t = 0$ (left) to $t = 20$ (right). The patterns in the uncertainties match those in the solution. The simulated IVP is 125k-dimensional.

arbitrarily large, and we invite the reader to think of a dimension up to $d = 10^6$ or beyond. High-dimensional IVPs describe large networks of dynamical systems and are ubiquitous in the natural sciences and machine learning. For example, discretising a partial differential equation on K spatial grid points yields a K-dimensional initial value problem (refer to the upcoming Chapter 11).

To construct efficient probabilistic numerical solvers for vector-valued IVPs, we explore three different approaches, which amount to choosing three different correlation structures in the (vector-valued) integrated Wiener process prior:

- ◇ *Dense models:* A vector-valued integrated Wiener process in dimension d, via a multi-dimensional version of Equation (3.4) in Chapter 3 (Section 8.4).
- ◇ *Block-diagonal models:* A collection of d statistically independent, scalar-valued integrated Wiener processes, each of which is defined as in Chapter 3 (Section 8.5). This model was used to create Figure 8.1.
- ◇ *Kronecker models:* Vector-valued integrated Wiener processes with a Kronecker-structured covariance matrix (Section 8.7).

We name the three options in line with the kind of covariance structure they impose. Some of them are known under different names in different communities. For example, the dense model is a conventional multi-output Gaussian process [e.g., 4, 118], the block-diagonal model makes an independence assumption, and Kronecker factorisation connects to separable covariance kernels [e.g., 24, 180]. Naming state-space models after their covariance matrix factorisation recommends itself for two reasons: (i) It closely relates to numerical linear algebra concerns, which are the focus of the present chapter; (ii) terms like "independence", "vector-valued", or "separability" feature prominently in all other chapters, and (re)using them here would lead to confusion.

The remainder of this chapter proceeds as follows: Section 8.2 recalls important concepts from previous parts and introduces the probabilistic numerical solution of vector-valued IVPs, Sections 8.3 to 8.7 discuss implementation strategies, Section 8.8 contains some preliminary benchmarks (more are in Chapter 9), and Section 8.9 concludes the exposition.

8.2 Vector-valued IVP solutions

Recall the setup from Chapter 3: A one-dimensional, ν-times integrated Wiener process is the zeroth element $Y^{(0)}(t)$ of the stack of states $Y(t)$, which are pairwise related through a Wiener-process-driven stochastic differential equation (that was Equation (3.4)). The underlying Wiener process is one-dimensional with output scale $\gamma > 0$. Restricted to a grid $t_0, ..., t_N$, the scalar integrated Wiener process evolves as

$$p\left(Y(t_{n+1}) \mid Y(t_n), \gamma\right) = \mathcal{N}\left(\Phi_\nu(\Delta t_n) Y(t_n), \Sigma_\nu(\Delta t_n, \gamma)\right) \tag{8.2}$$

with the Pascal matrix $\Phi_\nu(\Delta t)$ and the Hilbert matrix $\Sigma_\nu(\Delta t, \gamma) = \gamma^2 \Sigma_\nu(\Delta t, 1)$; again, detailed expressions are in Chapters 3 and 7.

Constraining $Y = (Y^{(0)}, ..., Y^{(\nu)})$ to attaining zero differential equation residual,

$$\mathcal{R}_{f,n} := Y^{(1)}(t_n) - f(Y^{(0)}(t_n)) = 0, \tag{8.3}$$

and zero initial condition residual, $\mathcal{R}_{y_0} := Y(t_0) - y_0 = 0$, approximates the IVP solution. In other words, solving the IVP amounts to estimating

$$p\left(Y(t) \;\middle|\; [\mathcal{R}_{f,n} = 0]_{n=0}^{N},\ \mathcal{R}_{y_0} = 0, \gamma\right). \tag{8.4}$$

To improve numerical stability and accuracy, constraining the prior to $\mathcal{R}_{y_0} = 0$ is replaced by Taylor-series estimation, which usually uses automatic differentiation or solves a regression problem (Chapter 6). Extrapolation, correction, and calibration implement Cholesky arithmetic with Gaussian variables (Chapters 4 and 7). Preconditioning stabilises the extrapolation; Either Taylor linearisation or statistical linear regression enables a Gaussian correction, and approximate maximum-likelihood calibrates the output scale in closed form. All of this has been discussed thoroughly in Chapter 7, and we generalise this approach to multi-dimensional models next.

The overall approach remains the same when moving from scalar to vector-valued differential equations. The only difference is that the prior model $Y(t)$ becomes an appropriate vector-valued process with a potentially vector-valued output scale Γ and that the residuals $\mathcal{R}_{f,n}$ and $\mathcal{R}_{y_0}$ are vector-valued, too.

Each one of Sections 8.4, 8.5 and 8.7 constructs one such vector-valued integrated Wiener process, exposes the covariance matrix factorisation after time discretisation, and emphasizes which linearisation strategies preserve this factorisation. Where required, the sections explain the calibration of a time-constant output scale. Turning the calibration of a time-constant output scale into that of a dynamic (time-varying) output scale follows the same strategy outlined in Chapter 7 and is therefore omitted.

The following Sections 8.3 to 8.7 can be read independently with one exception: Section 8.7 builds on the tools reviewed in Section 8.6.

8.3 Initialisation

Initialising the IVP solver at time $t = 0$ amounts to computing the first ν Taylor coefficients of the IVP solution. Chapter 6 explained how this is possible with automatic differentiation by repeatedly differentiating the IVP dynamics. The two modes for automatic differentiation were recursive forward-mode differentiation and the propagation of Taylor polynomials coined “Taylor-mode” differentiation. Chapter 6 also explained how a Taylor series can be estimated by solving a specific regression problem, in the absence of automatic differentiation. For low-dimensional problems, initialisation with Taylor-mode automatic differentiation is the state of the art.

Estimating a Taylor series of an IVP solution with recursive forward-mode differentiation is inefficient, especially when the IVP dimension increases. There exists a version of Taylor-mode differentiation that is based on propagating univariate Taylor polynomials and can thus be implemented more cheaply [69, 70], which makes it a valuable option for medium- to high-dimensional problems. For extremely high-dimensional problems, automatic differentiation can be problematic.

The regression-based approach inherits its complexity from the underlying state-space model factorisation because it uses the same extrapolation and correction steps as the IVP solver (Chapter 4; generalised to vector-valued models below). Thus, it scales to high dimensions as well as the IVP solver (discussed next) and becomes the default initialisation method for extremely high-dimensional problems.

8.4 Dense models

The first option for building vector-valued models is a dense covariance structure instead of a block-diagonal or Kronecker covariance structure (Sections 8.5 and 8.7). We choose to present dense models first because they are a natural extension of scalar-valued integrated Wiener processes and because they have received more attention than the other factorisations in the literature on probabilistic numerical IVP solvers (references below).

Solving IVPs using a dense covariance is almost identical to the scalar setup discussed in the previous chapter; it works as follows. Recall $\Phi_\nu(\Delta t)$, $\Sigma_\nu(\Delta t, \gamma)$, m_0, and $C_0(\gamma)$ from Section 8.2 above. Let $I_d \in \mathbb{R}^{d\times d}$ be the identity, and let $\Lambda \in \mathbb{R}^{d\times d}$ be a fixed and known matrix (usually, Λ is also the identity). Let W be a d-dimensional Wiener process with unit diffusion.

Define a ν-times integrated Wiener process as the solution of the system of stochastic differential equations

$$\mathrm{d}Y^{(q)}(t) = Y^{(q+1)}(t)\,\mathrm{d}t, \quad q = 0, ..., \nu - 1, \tag{8.5a}$$

$$\mathrm{d}Y^{(\nu)}(t) = \gamma\Lambda\,\mathrm{d}W(t), \tag{8.5b}$$

subject to the Gaussian initial condition

$$p(Y(0) \mid \gamma) = \mathcal{N}(m_0, C_0(\gamma)). \tag{8.6}$$

As usual, assume $C_0(\gamma) = \gamma^2 C_0(1)$. The only difference to the stochastic differential equation in Chapter 3 is that the Wiener process $W(t)$ in Equation (8.5) is vector-valued, and that $m_0 \in \mathbb{R}^{d(\nu+1)}$ and $C_0(\gamma) \in \mathbb{R}^{(\nu+1)d\times(\nu+1)d}$ have more rows and columns.

Restricted to some grid $t_0, ..., t_N$ with spacing $\Delta t_n := t_{n+1} - t_n$, the process follows

$$p\left(Y(t_{n+1}) \mid Y(t_n), \gamma\right) = \mathcal{N}\left([\Phi_\nu(\Delta t_n) \otimes I_d]Y(t_n), \Sigma_\nu(\Delta t_n, \gamma) \otimes \Lambda\Lambda^\top\right) \tag{8.7}$$

where $\otimes$ is the Kronecker product. The preconditioner for the extrapolation, discussed in Chapter 7, has the same Kronecker structure as the transition matrix. Extrapolation thus remains identical to the extrapolation for scalar problems (Chapter 7), but the system matrices have d-times as many rows and columns. The computational complexity of the extrapolation step is thus $O(\nu^3 d^3)$, including preconditioning and Cholesky parametrisation.

Recall the Euclidean basis vector e_q, which is the qth row of an identity matrix with $\nu + 1$ rows and columns. Define $E_q := e_q \otimes I_d$. By construction, e_q selects the qth derivative from the full state, $Y^{(q)}(t) = E_q Y(t)$, and we use it to define the differential equation residual

$$\mathcal{R}_{f,t} := E_1 Y(t) - f(E_0 Y(t)). \tag{8.8}$$

A first-order Taylor-linearisation of f around some $z \in \mathbb{R}^d$,

$$f(x) \approx Ax + b \tag{8.9}$$

for some $A \in \mathbb{R}^{d\times d}$ and $b \in \mathbb{R}^d$, approximates $\mathcal{R}_{f,t}$ as an affine function of $Y(t)$,

$$\mathcal{R}_{f,t} \approx (E_1 - AE_0)Y(t) - b =: HY(t) - b \tag{8.10}$$

with $H \in \mathbb{R}^{d \times d(\nu+1)}$. Zeroth-order linearisation and statistical linear regression work similarly. Like the extrapolation step, the correction step is identical to that for scalar problems after linearisation (Chapter 7), except that the system matrices have d-times as many rows/columns, which raises the computational complexity of the required numerical linear algebra to $O(d^3\nu^3)$.

With the same argument as in Chapter 7, we can show that under the above assumptions, all covariances involved in the dense state-space model are of the form $C(\gamma) = \gamma^2 C(1)$, and a quasi-maximum likelihood estimate for γ emerges in the same fashion as in Chapter 7. The detailed expression can be found in the paper by Tronarp et al. [163]; see also the work by Bosch et al. [25], Schober et al. [152].

The dense model is standard for implementing algorithms like the Kalman filter for vector-valued problems. Moreover, perhaps unsurprisingly, a dense covariance model has also been standard for solving vector-valued IVPs with a probabilistic numerical algorithm; for example, refer to Bosch et al. [25, 26], Kersting and Hennig [91], Kersting et al. [93], Krämer and Hennig [100], Schober et al. [152], Tronarp et al. [163, 164]. The computational advantages of the factorisations presented next have not been exploited until recently, which is one contribution of this thesis.

Overall, solving an IVP with the dense state-space model costs $O(\nu^3 d^3)$ per step and follows the same strategy as the implementation for scalar problems. Compared to the block-diagonal or Kronecker models below, the dense covariance structure is the most expressive because it does not factorise the covariance matrices. However, it is also the most expensive state-space model because estimation scales cubically with the IVP dimension.

8.5 Block-diagonal models

A block-diagonal state-space model factorises all covariances into block-diagonal matrices as follows. Recall the system matrices for the scalar model $\Phi_\nu(\Delta t)$, $\Sigma_\nu(\Delta t, \gamma)$, m_0, and $C_0(\gamma)$ from Section 8.2.

Define $Y(t) = [Y_i(t)]_{i=1}^d$ as a stack of d independent, scalar-valued, ν-times integrated Wiener processes, each with output scale γ_i, $i = 1, ..., d$, and some Gaussian initial distribution

$$p(Y(t_i) \mid \gamma_i) = \mathcal{N}(m_0^i, C_0^i(\gamma_i),) \tag{8.11}$$

whose parameters are assumed to be known. The joint initial distribution over $Y(t)$ has a block-diagonal covariance, the ith block being $C_0^i(\gamma)$, which is why we call this state-space model "block-diagonal". As usual, assume $C_0^i(\gamma_i) = \gamma_i^2 C_0^i(1)$.

Every process $Y_i(t)$ transitions independently from the others as

$$p\left(Y_i(t_{n+1}) \mid Y_i(t_n), \gamma_i\right) = \mathcal{N}\left(\Phi_\nu(\Delta t_n) Y_i(t_n), \Sigma_\nu(\Delta t_n, \gamma_i)\right), \quad i = 1, ..., d, \tag{8.12}$$

and extrapolation happens dimension-wise, applying the algorithms from Chapter 7 for every dimension, in overall complexity $O(d\nu^3)$.

Ideally, we would like to apply the correction-step dimension-wise as well, but this is only feasible under a specific condition on the IVP:

Proposition 8.1 (Block-diagonal correction)**.** *If the vector field of the IVP is affine with a diagonal Jacobian matrix, then:*

- ◇ *The correction step can be implemented dimension-wise.*
- ◇ *A quasi-maximum likelihood estimate of the output scale involves calibrating each γ_i independently with the technique from Chapter 7, $i = 1, ..., d$.*

Proof. If the vector field of the IVP is affine with a diagonal Jacobian, we write it as

$$f(y) = [a_i y_i + b_i]_{i=1}^d \tag{8.13}$$

and all elements in the residual $\mathcal{R}_{f,t}$ are conditionally independent given $Y(t)$. Therefore, estimation happens coordinate-wise, and both statements must be true. □

If the vector field of the IVP does not satisfy the conditions in Proposition 8.1 – for example, when it is nonlinear – we approximate.

Zeroth-order Taylor linearisation induces an affine approximation of the IVP with a zero Jacobian matrix, so it satisfies the assumptions of Proposition 8.1. In other words, correction and quasi-maximum likelihood calibration with zeroth-order linearisation can be implemented in $O(d\nu^3)$.

First-order linearisation requires additional care. Let $z_0, z \in \mathbb{R}^d$ and define A and b as the first-order Taylor linearisation of f,

$$f(z) = Az + b, \quad A := Df(z_0), \quad b := f(z_0) - Df(z_0)z_0. \tag{8.14}$$

Unless f has a diagonal Jacobian, A is not diagonal; but approximating

$$A \approx \operatorname{diag}(A) \tag{8.15}$$

yields a linearisation compatible with Proposition 8.1. This diagonal approximation has not been explored much in past literature; one rare occurrence is Murtuza and Chorian [119]'s work on the *node-decoupled Kalman filter*. Provided the diagonal of the Jacobian matrix can be assembled in $O(d)$, the correction and calibration steps can be implemented in $O(d\nu^3)$. If the diagonal of the Jacobian is unknown, Hutchinson's trick [83] yields an estimate that only requires access to Jacobian-vector products. Block-diagonal statistical linear regression is an open problem.

In conclusion, a block-diagonal correlation structure can be implemented in a complexity that increases linearly with the IVP dimension instead of the cubic complexity in Section 8.4. As such, it is significantly more efficient than the dense model but requires discarding correlations between dimensions. For some problems, this assumption may be appropriate; for others, alternatives must be considered. One such alternative is the Kronecker factorisation in Section 8.7. However, examining Kronecker models requires a detour via discussing the relationship between Kronecker products and matrix-normal distributions, presented next in Section 8.6. Afterwards, Section 8.7 studies the Kronecker-factorised state-space model.

8.6 Kronecker products and matrix-normal distribution

Before discussing the solution of IVPs in Kronecker-factorised state-space models, we recall some fundamental tools for matrix-normal distributions and linear algebra with Kronecker products. All statements below are either well-known facts about Kronecker products or follow directly from the definition of matrix-normal distributions. Textbooks like the ones by Golub and Van Loan [67] and Gupta and Nagar [74], as well as the paper by Gupta and Varga [73], treat these topics in more depth.

The product $A \otimes B$ is the Kronecker product of matrices A and B. For any matrix $M \in \mathbb{R}^{n\times m}$, let $\operatorname{vec}(M) \in \mathbb{R}^{nm}$ be the row-vectorised representation of M. Kronecker products and vectorisation interact as $(A \otimes B)\operatorname{vec}(M) = AMB^\top$. Like in Chapter 4, let m_{in}, b_{cond}, m_{joint}, C_{in}, C_{cond}, C_{joint} be arbitrary mean vectors and covariance matrices, and let A_{cond} be an arbitrary matrix. All of these vectors and matrices shall have compatible dimensions in the sense that the matrix-vector operations are well-defined.

Let $(\mathcal{MN})_{n\times m}$ be a matrix-normal distribution over the space of $n \times m$ matrices, defined as follows: Random variables with a matrix-normal distribution yield multivariate Gaussian distributions with Kronecker-factorised covariance after vectorisation [74]; this means that the matrix-normal distribution

$$p(X) = (\mathcal{MN})_{n\times m}(m_{\text{in}}, (C_{\text{in}})_1, (C_{\text{in}})_2) \tag{8.16}$$

is equivalent to the multivariate normal distribution

$$p(\operatorname{vec}(X)) = \mathcal{N}(\operatorname{vec}(m_{\text{in}}), (C_{\text{in}})_1 \otimes (C_{\text{in}})_2) \tag{8.17}$$

after vectorisation (flattening). In Equation (8.17), we call $(C_{\text{in}})_1$ the “left” Kronecker factor, and $(C_{\text{in}})_2$ the “right” Kronecker factor.

From the equivalence between Equation (8.16) and Equation (8.17), standard properties of matrix-variate Gaussian distributions emerge by translating properties of multivariate normal distributions with Kronecker-factorised covariances. For example, a matrix-normal conditional distribution $p(Y \mid X)$ that shares a right Kronecker factor with X (that is, a “row-wise” conditional distribution; note the same $(C_{\text{in}})_2$ as in

Equation (8.16)),

$$p(Y \mid X) = (\mathcal{MN})((A_{\text{cond}})_1 X + b_{\text{cond}}, (C_{\text{cond}})_1, (C_{\text{in}})_2) \tag{8.18}$$

implies a matrix-normal joint distribution

$$p(X, Y) = (\mathcal{MN})(m_{\text{joint}}, (C_{\text{joint}})_1, (C_{\text{in}})_2) \tag{8.19}$$

with parameters

$$m_{\text{joint}} := \begin{pmatrix} m_{\text{in}} \\ (A_{\text{cond}})_1 m_{\text{in}} + b_{\text{cond}} \end{pmatrix}, \tag{8.20a}$$

$$(C_{\text{joint}})_1 := \begin{pmatrix} (C_{\text{in}})_1 & (C_{\text{in}})_1 (A_{\text{cond}})_1^\top \\ (A_{\text{cond}})_1 (C_{\text{in}})_1 & (A_{\text{cond}})_1 (C_{\text{in}})_1 (A_{\text{cond}})_1^\top + (C_{\text{cond}})_1 \end{pmatrix}. \tag{8.20b}$$

The marginal and conditional distributions inherit the matrix-normal factorisation from the joint distribution, and since all terms share a right covariance factor, Gaussian conditioning involves linear algebra only with the left one. This information is crucial for implementing Kronecker-factorised IVP solvers, where the right factor describes the spatial correlation (which depends on the IVP dimension), and the left factor models the integrated Wiener process (independent of the IVP dimension).

The same would hold for "column-wise" conditional distributions, where the operation $(A_{\text{cond}})_1 X$ in Equation (8.18) becomes $X {A_{\text{cond}}}^\top$, the left Kronecker factor remains, and the right Kronecker factor changes. However, considering only row-wise operations suffices for Section 8.7.

Square-root parametrisation of matrix-normal distributions amounts to storing the generalised Cholesky factors of each covariance matrix. As discussed above, the Cholesky arithmetic of two matrix-normal variables that share one covariance factor (as in Equation (8.20b)) is carried out only in "the other" factor. This brings with it the computational advantages of manipulating matrix-normal variables with shape $n \times m$ in either $O(n^3 + n^2 m)$ or $O(nm^2 + m^3)$ instead of $O(n^3 m^3)$, depending on row-wise versus column-wise conditionals. The only task of the next section is thus to construct a state-space model where extrapolation and correction preserve a matrix-normal factorisation, and IVP simulation will be efficient by construction.

8.7 Kronecker models

Next, we apply the knowledge from Section 8.6 to vector-valued integrated Wiener processes. Let $\Gamma \in \mathbb{R}^{d \times d}$ be a fixed but unknown matrix and let W be a d-dimensional Wiener process with unit diffusion. The matrices $m_0 \in \mathbb{R}^{(\nu+1) \times d}$ and $C_0 \in \mathbb{R}^{(\nu+1) \times (\nu+1)}$ shall be known. Note how unlike in previous sections, m_0 is a $(\nu + 1) \times d$-dimensional matrix instead of a $(\nu + 1)d$-dimensional vector.

Define a d-dimensional, ν-times integrated Wiener process as the zeroth component $Y^{(0)}(t)$ of the solution of the system of stochastic differential equations

$$\mathrm{d}Y^{(q)}(t) = Y^{(q+1)}(t)\,\mathrm{d}t, \quad q = 0, ..., \nu - 1, \tag{8.21a}$$

$$\mathrm{d}Y^{(\nu)}(t) = \Gamma\,\mathrm{d}W(t), \tag{8.21b}$$

subject to the matrix-normal initial condition

$$p(Y(0) \mid \Gamma) = (\mathcal{MN})_{\nu+1\times d}(m_0, C_0, \Gamma\Gamma^\top). \tag{8.22}$$

This setup is almost the same as for the dense model in Section 8.4, but the present section replaces the product $\gamma\Lambda$ with Γ, and imposes a matrix-normal initial condition. Restricted to a grid $t_0, ..., t_N$, the integrated Wiener process follows the matrix-valued conditional

$$p\left(Y(t_{n+1}) \mid Y(t_n), \Gamma\right) = (\mathcal{MN})_{(\nu+1)\times d}\left(\Phi_\nu(\Delta t_n)Y(t_n), \Sigma_\nu(\Delta t_n, 1), \Gamma\Gamma^\top\right) \tag{8.23}$$

which is a direct result of writing Equation (8.7) as a matrix-normal distribution (and replacing $\gamma\Lambda$ with Γ). Following the argument in Section 8.6, extrapolation can be implemented in $O(d\nu^2 + \nu^3)$ because the initial condition and the conditional share the right Kronecker factor and because the left Kronecker factor has exactly $\nu + 1$ rows and columns, which is independent of d. Preconditioning happens row-wise.

The logic behind the correction step is similar to that for block-diagonal models: First, identify which kinds of affine models preserve the matrix-normal structure; then, linearise the nonlinear vector field accordingly.

Proposition 8.2. *If the vector field $f : \mathbb{R}^d \to \mathbb{R}^d$ is affine and of the form*

$$f(x) = ax + b \tag{8.24}$$

for scalar $a \in \mathbb{R}$ and vector $b \in \mathbb{R}^d$, the correction step preserves the matrix-normal structure in the unobserved states, conditionals, and constraints,

$$(\mathcal{MN})_{k\times d}(\star, \star, \Gamma\Gamma^\top) \tag{8.25}$$

where "k" is either $\nu + 1$ (for the hidden states and backward transitions) or 1 (for the constraints), and "$\star$" is a placeholder for varying values.

The proof of Proposition 8.2 mirrors that of Proposition 8.1 by combining the fact that the selection operator $Y(t) \to Y^{(q)}(t)$ is a row-wise operation – thus, it preserves matrix-normal factorisations – with the statements in Section 8.6.

Under the assumption of Proposition 8.2, the correction step preserves the matrix-normal structure and can be implemented in $O(d\nu^2 + \nu^3)$ because the right Kronecker factor, $\Gamma\Gamma^\top$, remains constant and because the size of the left Kronecker factor is independent of the IVP dimension. As such, correction in the Kronecker model is even more efficient than the block-diagonal factorisation ($O(d\nu^3)$ vs $O(d\nu^2 + \nu^3)$). A matrix-normal integrated Wiener process implies the assumption that all rows in $Y(t)$ have the same prior distribution and that all columns may vary with covariance $\Gamma\Gamma^\top$ and as such, it is neither more nor less general than the block-diagonal model.

Proposition 8.2 requires that the vector field is affine with a Jacobian that is the product of a scalar and an identity matrix. If the IVP does not satisfy this condition, for example, if the vector field is nonlinear, then a zeroth-order Taylor linearisation yields an appropriate form. Forcing first-order methods and statistical linear regression into this form is an open problem and left for future work.

By construction, the marginals of the probabilistic numerical IVP solution and the likelihood of the constraints are matrix-normal with an identical right factor, $\Gamma\Gamma^\top$. Like in Chapter 7, this implies that Γ can be calibrated offline, i.e., after the simulation. Furthermore, we can find a (quasi-)maximum likelihood estimate for Γ in closed form as follows. The derivation uses the following identities from matrix calculus and can be found in, e.g., the work by Petersen et al. [130].

Lemma 8.3. *Let $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{m\times m}$, and $M \in \mathbb{R}^{n\times m}$ be given matrices. ∇ is the gradient operator. Then,*

$$\operatorname{vec}(M)^\top (A \otimes B)\operatorname{vec}(M) = \operatorname{tr}(M^\top A M B^\top), \tag{8.26a}$$

$$\nabla_B \operatorname{tr}(AB^\top) = A, \tag{8.26b}$$

$$\log\det(A \otimes B) = m \log\det A + n \log\det B, \tag{8.26c}$$

$$\nabla_X \log\det X = -X^{-\top}. \tag{8.26d}$$

Let “const” be a placeholder for unimportant values that are independent of Γ. Assume that the IVP is of the form $f(\xi) = a\xi + b$ for $a \in \mathbb{R}$ and $b \in \mathbb{R}^d$ and satisfies Proposition 8.2; otherwise, we apply zeroth-order Taylor linearisation to derive a quasi-maximum likelihood estimate instead of a maximum-likelihood estimate. Recall the Euclidean basis vector e_q, which is the qth row of the identity matrix with $\nu + 1$ rows and columns.

At any grid-point t_n, the differential equation residual is an affine, Kronecker-factorisation-preserving transformation of the state,

$$\mathcal{R}_{f,n} \coloneqq HY(t_n) - b, \quad H \coloneqq (e_1 - ae_0) \in \mathbb{R}^{1\times(\nu+1)} \tag{8.27}$$

with a matrix-normal marginal distribution

$$p\left(\mathcal{R}_{f,n} \mid \mathcal{R}_{f,0:n-1} = 0, \Gamma\right) = (\mathcal{MN})_{1\times d}\left(Hm_n^-, HC_n^- H^\top, \Gamma\Gamma^\top\right) \tag{8.28a}$$

$$= (\mathcal{MN})_{1\times d}\left(s_n, S_n, \Gamma\Gamma^\top\right) \tag{8.28b}$$

for $s_n \in \mathbb{R}^{1\times d}$ and $S_n \in \mathbb{R}^{1\times 1}$ that depend on the IVP and the extrapolated mean m_n^- and covariance C_n^- of the state at time t_n. Notably, s_n is a "row-vector" and S_n is a scalar, which is important for the upcoming step.

For the sake of simplicity, ignore the initial condition constraints in the derivation below, which is reasonable when the solver is initialised with automatic differentiation; otherwise, the derivation below adapts straightforwardly. The negative log-likelihood of the differential equation constraints is (using Lemma 8.3)

$$
\begin{aligned}
&-\log p(\mathcal{R}_{f,0:N} = 0 \mid \Gamma) && (8.29a)\\
&\quad \propto \sum_{n=0}^{N} \left(\| \operatorname{vec}(s_n) \|^2_{(S_n \otimes \Gamma\Gamma^\top)^{-1}} - \log\det(S_n \otimes \Gamma\Gamma^\top) \right) + \text{const} && (8.29b)\\
&\quad \propto \sum_{n=0}^{N} \operatorname{tr}\left[s_n^\top S_n^{-1} s_n (\Gamma\Gamma^\top)^{-1} \right] - (N+1)\log\det\Gamma\Gamma^\top + \text{const}. && (8.29c)
\end{aligned}
$$

Setting the derivative with respect to $\left(\Gamma\Gamma^\top\right)^{-1}$ to zero yields the closed-form maximum-likelihood estimate (again, using Lemma 8.3)

$$
\Gamma\Gamma^\top := \frac{1}{N+1} \sum_{n=0}^{N} s_n^\top S_n^{-1} s_n \tag{8.30}
$$

If the residual corresponding to the initial condition is not ignored, the sum involves a corresponding extra term. If the IVP is nonlinear, zeroth-order linearisation turns Equation (8.30) into a quasi-maximum likelihood estimate.

Since $s_n \in \mathbb{R}^{1\times d}$ is a row-vector, $s_n^\top S_n(1) s_n$ is an outer product, not an inner product; and since S_n is scalar, each summand can be computed in $O(d)$. The expression in Equation (8.30) requires $O(Nd)$ storage because each $S_n^{-1/2} s_n$ must be stored. Matrix-vector multiplications with $\Gamma\Gamma^\top$ can be implemented with summand-wise matrix-vector multiplications in Equation (8.30), in complexity $O(Nd)$. Extracting Γ from $\Gamma\Gamma^\top$ is possible via a single QR-decomposition of a matrix with $N+1$ rows and d columns, which costs Nd^2.

For all scenarios where this is too expensive, we can formulate a model for the output scale like in Section 8.4. That is, if $\Gamma = \gamma\Lambda$ holds for an unknown $\gamma \in \mathbb{R}$ and a known Λ, mimicking the above derivation yields an estimator of the form

$$
\gamma := \frac{1}{d(N+1)} \sum_{n=0}^{N} \operatorname{tr}\left[s_n^\top S_n^{-1} s_n (\Lambda\Lambda^\top)^{-1} \right] = \frac{1}{N+1} \sum_{n=0}^{N} S_n^{-1} s_n (\Lambda\Lambda^\top)^{-1} s_n^\top \tag{8.31}
$$

due to the cyclic property of traces and because S_n is scalar. If $(\Lambda\Lambda^\top)^{-1}$ admits linear-complexity matrix-vector products – which includes the most common case of

Λ being the identity – the estimator can be implemented in $O(d)$. But other relevant scenarios apply, too, for example, when the covariance Λ results from the stochastic partial differential approach to simulating Gauss-Markov random fields [107].

In conclusion, zeroth-order linearisation preserves matrix-normal structure in the state-space underlying a probabilistic numerical IVP solver. As a result, a single algorithm step can be implemented in $O(d\nu^2+\nu^3)$, which is significantly more efficient than the dense model ($O(d^3\nu^3)$) and slightly more efficient than the block-diagonal model ($O(d\nu^3)$). The benchmarks in the next section corroborate this analysis.

8.8 Benchmarks

In the following section, we demonstrate the computational complexity of the factorisations in two example setups. First, Section 8.8.1 examines the runtimes of a single step of the IVP solver; second, Section 8.8.2 demonstrate how a million-dimensional IVP can be solved within a few hours using the Kronecker model. This section does not include runtime comparisons of the solver to state-of-the-art non-probabilistic algorithms because those are all in Chapter 9.

8.8.1 A single step

We begin by evaluating the cost of a single step of the IVP solver variations on the Lorenz96 problem, a chaotic dynamical system [108]. It is given by a system of $N \geq 4$ differential equations

$$\dot{y}_1 = (y_2 - y_{N-1})y_N - y_1 + F, \tag{8.32a}$$
$$\dot{y}_2 = (y_3 - y_N)y_1 - y_2 + F, \tag{8.32b}$$
$$\dot{y}_i = (y_{i+1} - y_{i-2})y_{i-1} - y_i + F \qquad i = 3, \ldots, N-1, \tag{8.32c}$$
$$\dot{y}_N = (y_1 - y_{N-2})y_{N-1} - y_N + F, \tag{8.32d}$$

with forcing term $F = 8$, initial values $y_1(0) = F + 0.01$ and $y_k(0) = F$, $k \geq 2$, as well as time span $t \in [0, 30]$. This chaotic dynamical system recommends itself for the first experiment, as its dimension can be increased freely. We time a single step with a probabilistic numerical IVP solver for increasing IVP dimension d and different solver orders $\nu \in \{2, 4, 6\}$. The results are in Figure 8.2. The experiment shows how the dense model quickly becomes infeasible due to its cubic complexity in the IVP dimension. The block-diagonal and Kronecker models exhibit their $O(d)$ complexity. Altogether, Figure 8.2 confirms the analysis in Sections 8.3 to 8.7.

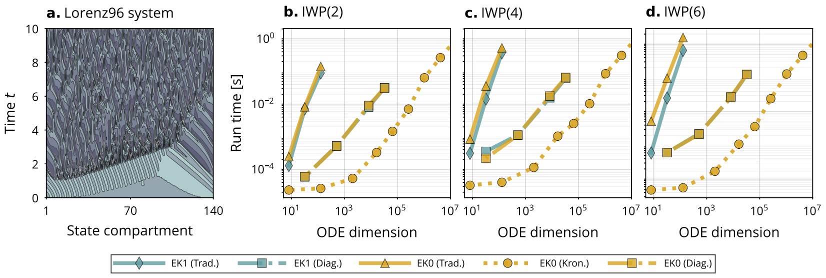

Figure 8.2: *Runtime of a single IVP solver step:* Run time (wall-clock) of a single step of IVP solver variations on the Lorenz96 problem (a) for increasing IVP dimension and $\nu = 2, 4, 6$ (b to d). The dense models (marked with "Trad.") cost $O(d^3)$ per step, and both the block-diagonal and the Kronecker model cost $O(d)$ per step. "EK1" is first-order, and "EK0" zeroth-order Taylor linearisation.

8.8.2 Extremely high dimensions

To evaluate how well the improved efficiency translates to extremely high dimensions, we solve the discretised FitzHugh–Nagumo partial differential equation (PDE) model on high spatial resolution (which translates to high-dimensional IVPs).

Let $\Delta = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}$ be the Laplacian. The FitzHugh–Nagumo PDE is [5]

$$\frac{\partial}{\partial t} u(t,x) = a\Delta u(t,x) + u(t,x) - u(t,x)^3 - v(t,x) + k, \tag{8.33a}$$

$$\frac{\partial}{\partial t} v(t,x) = \frac{1}{\tau}(b\Delta v(t,x) + u(t,x) - v(t,x)), \tag{8.33b}$$

for some parameters a, b, k, τ, and initial values $u(t_0, x) = h_0(x)$, $v(t_0, x) = h_1(x)$. We chose $a = 208 \cdot 10^{-4}$, $b = 5 \cdot 10^{-3}$, $k = -5 \cdot 10^{-3}$, $\tau = 0.1$. We used random samples from the uniform distribution on $(0, 1)$ as initial values. We solve the PDE from $t_0 = 0$ to $t_{\max} = 20$ on a range of spatial domains $x \in [0, W] \times [0, W] \subseteq \mathbb{R}^2$, with $W \in \{0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50\}$. We discretised the Laplacian with central, second-order finite differences schemes on a uniform grid to turn the PDE into a system of ordinary differential equations. The mesh size of the grid determines the number of grid points, which controls the dimension of the IVP.

All probabilistic numerical solutions are computed with a 3-times integrated Wiener process prior, a time-varying scalar diffusion, and with step-size adaptation for chosen tolerances $\tau_{\text{abs}} = 10^{-3}$, $\tau_{\text{rel}} = 10^{-1}$. The DOP853 solutions are computed with tolerance levels $\tau_{\text{abs}} = 10^{-6}$, $\tau_{\text{rel}} = 10^{-3}$. The results are in Figure 8.3. The

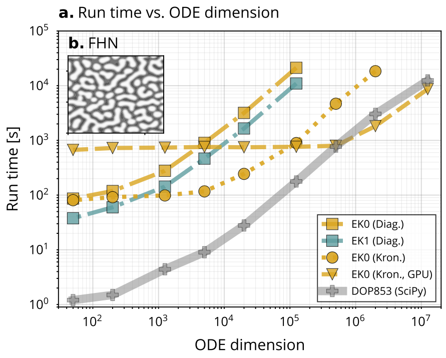

Figure 8.3: *High-dimensional PDE discretisation:* Run time of IVP solvers on the discretised FitzHugh–Nagumo model for increasing IVP dimension (i.e. increasing spatial resolution), including calibration and adaptive time steps. SciPy's DOP853 for reference. Simulating $\gg 10^6$-dimensional ODEs takes $\approx 3h$ with a Kronecker model. Like in Figure 8.2, "EK1" is a first-order Taylor linearisation and "EK0" is a zeroth-order Taylor linearisation.

main takeaway is that with Kronecker or block-diagonal models, IVPs with millions of dimensions can be solved probabilistically within a realistic time frame (hours), which is impossible with dense models. GPUs improve the runtime for extremely high-dimensional problems ($d \gg 10^5$).

8.9 Conclusion

This chapter extended the best practices for scalar models to vector-valued problems. Three different approaches have been presented: A dense model, a block-diagonal model, and a Kronecker model. Each of those has its advantages and disadvantages; for example, the dense model is the most descriptive but costs $O(d^3\nu^2)$, whereas the Kronecker model is essentially restricted to zeroth-order linearisation but costs only $O(d\nu^2)$. If the application allows, factorisations can also be combined (e.g., solve a matrix-valued differential equation by combining a Kronecker factorisation in one dimension and a block-diagonal factorisation in another).

The choice between the three options reduces to two decisions:

1. Which covariance factorisation is meaningful for a specific simulation problem?
2. Which computational complexity can be afforded?

Ultimately, the state-space model factorisations depends on the application. A JAX

[28] implementation of all three factorisations is provided by `ProbDiffEq`[1], which plays an essential role in the upcoming benchmarks in Chapter 9.

[1]`https://pnkraemer.github.io/probdiffeq/`

Chapter 9

Benchmarks

Contents

9.1 Wall-time evaluation of IVP solvers

Some numerical solvers for initial value problems (IVPs) are more efficient than others. Not only are there dedicated methods for, e.g., stiff or high-dimensional IVPs, but even within a confined problem class, some methods converge more quickly than the competition. This difference in performance is why benchmarking the numerical efficiency on realistic problems is central to the evaluation of numerical algorithms.

Among all possible benchmarks, comparing solvers' *wall times* per realised precision is most instructive because, ultimately, wall time is what every user must endure. Plotting the work required to compute an approximation against the approximation's error is called a *work-precision diagram*, a standard tool for visualising numerical efficiency. For instance, Wanner and Hairer [176, Section IV.10] (and many others) evaluate only work-precision diagrams. In the same spirit, all figures in the present chapter will contain work-precision diagrams that plot solvers' wall times against approximation errors.

By focussing narrowly on benchmarks that display the numerical efficiency of probabilistic numerical solvers, this chapter ignores many alternative considerations that determine the "usefulness" of a probabilistic numerical method: for instance, the precision per number of function evaluations or the calibration.[1] This choice is deliberate: Thus far, wall-time evaluations of probabilistic numerical IVP solvers simply have not been as popular as measuring the solver's precision per function evaluations or calibration.

Recently, this changed – not only through the papers discussed in this thesis but also in the parallel work by, e.g., Bosch et al. [25, 26]. Two confounding factors for this rise in popularity of wall-time benchmarks seem to be improved numerical stability (Chapter 7) and the broader availability of probabilistic numerics software; in other words, existing wall-time evaluations of probabilistic solvers use many of the techniques discussed in this thesis. The remainder of this chapter reproduces a selection of these benchmarks. In fact, and to the best of the author's knowledge, the work-precision diagrams that follow are the first of their kind to demonstrate rare configurations where probabilistic solvers are *more* efficient than the most efficient non-probabilistic method available in Python.

9.2 Multi-framework benchmarks

The results of a simulation study always depend on the used framework, including factors like the programming language and computing platform but also the sophistication of code optimisations. For example, both SciPy [170] and Diffrax [95] implement the same Dormand-Prince 5(4) pair [46] in Python, but the benchmarks below will demonstrate how Diffrax's implementation is many times faster than SciPy's. One reason for this difference is that Diffrax relies on JAX's JIT-compiler [28], whereas SciPy uses native Python code.

The upcoming simulations compare an implementation of probabilistic numerical solvers in JAX against the recommended algorithms in SciPy and Diffrax. The implementation of the probabilistic solvers is open-source and contained in the ProbDiffEq package, which can be installed via

```
pip install probdiffeq.
```
[2]

ProbDiffEq implements the algorithms almost exactly as recommended in this manuscript. The only differences are specialised modifications such as caching and reusing outputs of certain subfunctions. SciPy and Diffrax were selected because they provide the state-of-the-art Python-based IVP solvers: SciPy is the de-facto standard for scientific computing in Python, and Diffrax implements a wide range of non-probabilistic solvers in JAX and, as such, shares its computing framework with ProbDiffEq. In the remainder of this chapter, we will see how both ProbDiffEq and Diffrax consistently outperform SciPy in almost all benchmarks. The comparison

[1]Refer to Kersting et al. [93], Tronarp et al. [164] for the former, and read Bosch et al. [25], Schober et al. [152] for the latter.

[2]`https://pnkraemer.github.io/probdiffeq/`

between ProbDiffEq and Diffrax will reveal the actual performance differences between probabilistic and non-probabilistic methods.

The test problems cover different scenarios, including varying stiffness, dimensionality, and second-order dynamics. Specifically, we compute the solutions of the Lotka-Volterra (Section 9.3), Pleiades (Section 9.4), and Hires problem (Section 9.5), as well as a stiff version of the Van-der-Pol system (Section 9.6). A version of these benchmarks is open-source, available as part of ProbDiffEq's online documentation.[3] Comparing all implementations on these four example problems will provide a realistic image of the actual numerical efficiency of modern probabilistic numerical solvers when implemented as this manuscript recommends.

9.3 Problem: Lotka–Voltera

The Lotka–Volterra problem [109, 111, 171] describes predator-prey dynamics and is one of the most common test problems for IVP solvers. Many modern benchmark projects for dynamical systems [e.g. 113, 136] include a version of Lotka–Volterra. We parametrise the predator-prey dynamics as

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = 0.5 \cdot y_1 - 0.05 \cdot y_1 \cdot y_2, \quad \frac{\mathrm{d}y_2}{\mathrm{d}t} = -0.5 \cdot y_2 + 0.05 \cdot y_1 \cdot y_2, \tag{9.1}$$

with initial values $y_1(0) = y_2(0) = 20$, and simulate the solution at $t = 50$ from $t = 0$.

The following configurations are included:

◇ *ProbDiffEq:* All solvers use a constant output scale and calibrate it with maximum-likelihood estimation, combine adaptive step-size selection with proportional-integral control [75], and initialise with Taylor-mode automatic differentiation. The linearisation, number of derivatives, and state-space model factorisation vary.

◇ *Diffrax:* All solvers use the proportional-integral-derivative-controller [179] in Diffrax's default configuration, which replicates a proportional-integral controller, and set the `max_steps` to 10^5. The `solver` parameters vary.

◇ *SciPy:* All solvers use adaptive steps. The `method` parameters vary.

We compute the IVP solutions for a range of relative tolerances (setting the absolute tolerances to 10^{-3} times the relative tolerances) and measure each simulation's wall time and absolute root-mean-squared error. The reference solution is a high-precision estimate based on SciPy's `LSODA`. Every simulation is run 20 times, and sample mean and standard deviations are reported by Figure 9.1. In Figure 9.1, Diffrax's solvers are the most efficient methods. ProbDiffEq's algorithms consistently outperform SciPy's implementations.

[3]`https://pnkraemer.github.io/probdiffeq/`

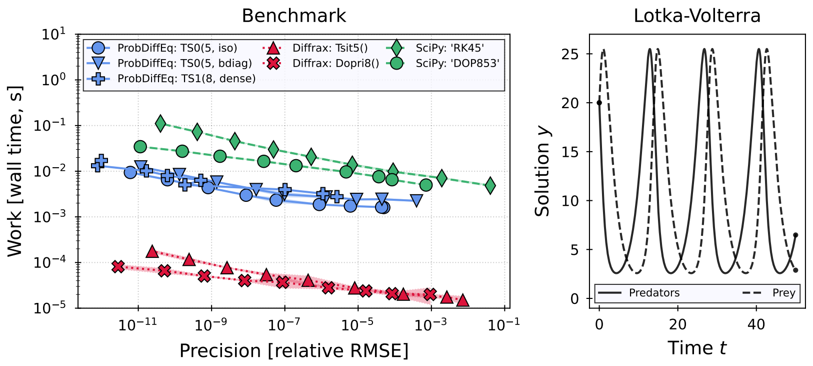

Figure 9.1: Work-precision benchmark on the *Lotka-Volterra* problem. The closer a curve is to the bottom left of the figure, the more efficient the implementation.

9.4 Problem: Pleiades

The Pleiades problem describes the motion of seven stars in a plane [76, p. 245]. It is a 14-dimensional, second-order differential equation. The Pleiades problem is part of this benchmark because it challenges a solver to scale to medium-high-dimensional problems. It is particularly interesting as a second-order equation, and solvers that target second-order equations will have an advantage over those that must transform it into a first-order equation. Since this affects the methods present in this benchmark, one can expect potentially different results from the Lotka-Volterra study.

We use the same parametrisation as Hairer et al. [76, p. 245], calling the x- and y-coordinates of the seven stars x_i and y_i with masses m_i, $i = 1, ..., 7$, and implementing

$$\frac{d^2 x_i}{dt^2} = \sum_{i \neq j} m_j (x_j - x_i)/r_{ij}, \tag{9.2a}$$

$$\frac{d^2 y_i}{dt^2} = \sum_{i \neq j} m_j (y_j - y_i)/r_{ij}, \tag{9.2b}$$

$$r_{ij} = ((x_i - x_j)^2 + (y_i - x_j)^2)^{3/2}. \tag{9.2c}$$

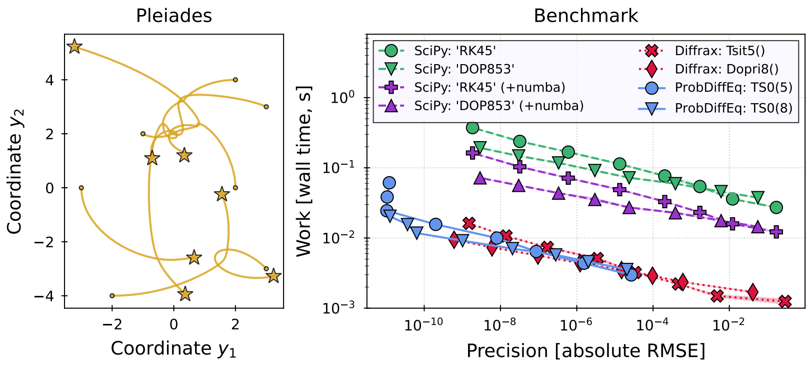

Figure 9.2: Work-precision benchmark on the *Pleiades* problem. The closer a curve is to the bottom left of the figure, the more efficient the implementation.

The initial values are

$$x(0) = (3, 3, -1, -3, 2, -2, 2), \qquad y(0) = (3, -3, 2, 0, 0, -4, 4), \tag{9.3a}$$

$$\frac{\mathrm{d}x(0)}{\mathrm{d}t} = (0, 0, 0, 0, 0, 1.75, -1.5), \qquad \frac{\mathrm{d}y(0)}{\mathrm{d}t} = (0, 0, 0, -1.25, 1, 0, 0), \tag{9.3b}$$

and we compare the wall time and absolute root-mean-square error of the approximation to a high-precision estimate at time $t = 3$, starting the simulation at $t = 0$.

The following solvers are included:

- *ProbDiffEq:* ProbDiffEq's solvers solve the second-order problem without transforming it into a first-order problem, which keeps the dimension of the IVP at 14. All methods use dynamic calibration of the output scale, zeroth-order Taylor linearisation, and proportional-integral control (interpolating at the terminal value if the controller suggests a too-large step), and initialise with Taylor-mode automatic differentiation. The state-space model has a Kronecker-factorisation, and the number of derivatives varies.

- *Diffrax & SciPy:* All solvers transform the problem into a first-order equation, which increases the dimensionality from 14 to 28. The remaining parametrisation is identical to Section 9.3. SciPy solvers include a version with and without JIT-compiling the vector field via Numba [104].

We run each code three times. The sample-mean and sample-standard-deviation are displayed in Figure 9.2. Notable differences between the results on Pleiades and Lotka-Volterra are that on Pleiades, ProbDiffEq's solvers are on par with Diffrax's solvers. Both are by a factor ∼10 faster than the SciPy/Numba combination and by a factor ~20 faster than SciPy without Numba. The relative improvement of ProbDiffEq's algorithms compared to Lotka–Volterra presumably stems from the fact that the probabilistic solver can solve the second-order IVP directly without transforming it, which keeps the IVP dimension low and improves the calibration of the probabilistic model [26]. That said, there exist non-probabilistic numerical solvers dedicated to second-order IVPs. However, neither Diffrax nor Scipy provided one at the time of implementing this benchmark.

9.5 Problem: High irradiance resistance

The "high irradiance resistance" problem [148], "Hires" in short, describes the growth and differentiation of plant tissue independent of photosynthesis at high levels of irradiance by light [176]. It is an eight-dimensional, stiff problem and a standard benchmark for solvers for stiff differential equations.

We implement the same parametrisation as Wanner and Hairer [176], with the differential equations

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = -1.71 \cdot y_1 + 0.43 \cdot y_2 + 8.32 \cdot y_3 + 0.0007 \tag{9.4a}$$

$$\frac{\mathrm{d}y_2}{\mathrm{d}t} = 1.71 \cdot y_1 - 8.75 \cdot y_2 \tag{9.4b}$$

$$\frac{\mathrm{d}y_3}{\mathrm{d}t} = -10.03 \cdot y_3 + 0.43 \cdot y_4 + 0.035 \cdot y_5 \tag{9.4c}$$

$$\frac{\mathrm{d}y_4}{\mathrm{d}t} = 8.32 \cdot y_2 + 1.71 \cdot y_3 - 1.12 \cdot y_4 \tag{9.4d}$$

$$\frac{\mathrm{d}y_5}{\mathrm{d}t} = -1.745 \cdot y_5 + 0.43 \cdot y_6 + 0.43 \cdot y_7 \tag{9.4e}$$

$$\frac{\mathrm{d}y_6}{\mathrm{d}t} = -280.0 \cdot y_6 \cdot y_8 + 0.69 \cdot y_4 + 1.71 \cdot y_5 - 0.43 \cdot y_6 + 0.69 \cdot y_7 \tag{9.4f}$$

$$\frac{\mathrm{d}y_7}{\mathrm{d}t} = 280 \cdot y_6 \cdot y_8 - 1.81 \cdot y_7 \tag{9.4g}$$

$$\frac{\mathrm{d}y_8}{\mathrm{d}t} = -280 \cdot y_6 \cdot y_8 + 1.81 \cdot y_7, \tag{9.4h}$$

and the initial values $y_1(0) = 1$, $y_2(0) = ... = y_7(0) = 0$, as well as $y_8(0) = 0.0057$. A high-accuracy reference solution is computed with SciPy's "BDF" method, which implements a backward-differentiation-formula [176, Chapter V], and tolerance 10^{-13}. We compare the relative root-mean-square error of the approximations at time $t = 321.8122$, starting the simulation at $t = 0$. The following solvers are included:

◇ *ProbDiffEq:* All methods use dynamic calibration of the output scale, first-order Taylor linearisation, proportional-integral control (clipping a step if the step-size

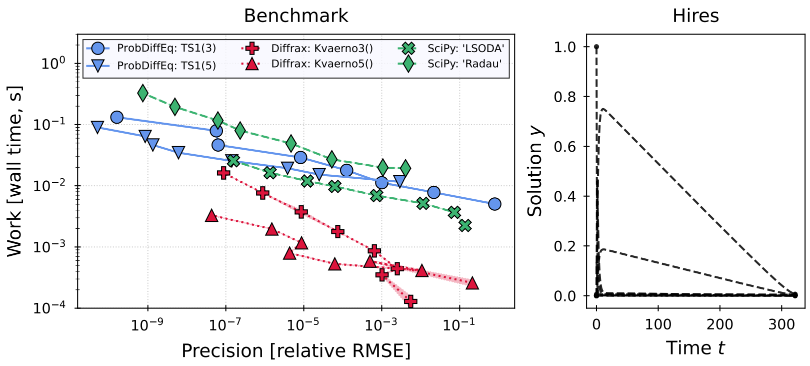

Figure 9.3: Work-precision benchmark on the *Hires* problem. The closer a curve is to the bottom left of the figure, the more efficient the implementation.

controller proposes a too-large increment), and initialise with Taylor-mode automatic differentiation. The state-space model does not factorise (i.e., the covariance matrices are dense), and the number of derivatives varies.

◇ *Diffrax & SciPy:* As in Section 9.3 but with methods for stiff problems.

We run each code ten times. The sample-mean and sample-standard-deviation are displayed in Figure 9.3. The benchmarks demonstrate how Diffrax's solvers are the fastest and that SciPy and ProbDiffEq are similarly efficient. More specifically, ProbDiffEq is slower than SciPy's LSODA for low precision (which wraps a Fortran implementation) – with the roles reversed for higher precision – and faster than SciPy's Radau (which is pure Python). Altogether, ProbDiffEq's solvers appear competitive but not superior to non-probabilistic solvers.

9.6 Problem: Stiff van-der-Pol

The van-der-Pol system is a non-conservative oscillator subject to non-linear damping due to Van der Pol [169]. It is a widespread benchmark problem because it includes a parameter μ, whose magnitude controls the stiffness of the equation. We choose it as $\mu = 10^5$, which makes the problem stiff (in which case explicit solvers such as the Dormand-Prince 5(4) pair are inefficient [176]). More specifically, we implement the same parametrisation as Wanner and Hairer [176, p. 144],

$$\frac{d^2 y}{dt^2} = 10^5((1-y^2)\cdot\frac{dy}{dt} - y)), \tag{9.5}$$

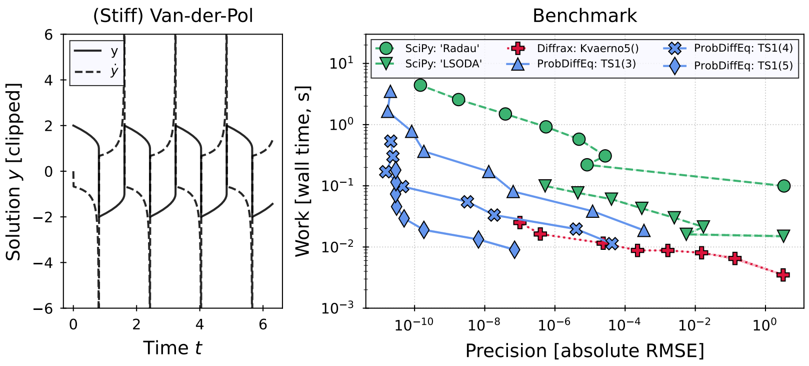

Figure 9.4: Work-precision benchmark on the *Van-der-Pol* problem. The closer a curve is to the bottom left of the figure, the more efficient the implementation. In the left figure, the y-axis is clipped to (-6, 6), even though the $\dot{y}$ values far exceed these limits.

with initial values $y(0) = 2$ and $\frac{\mathrm{d}y(0)}{\mathrm{d}t} = 0$. We solve from $t = 0$ to $t = 6.3$, calling each code three times, and use the same algorithms as in Section 9.5, with one difference: ProbDiffEq implements the differential equation as the (original) second-order problem, which keeps the dimensionality at one. In contrast, SciPy and Diffrax transform the problem into a two-dimensional, first-order equation. As in Section 9.4, this transformation has a significant effect on the numerical efficiency (Figure 9.4) to the point that (i) all of ProbDiffEq's considered solvers outperform all of SciPy's considered solvers, and (ii) high-order, probabilistic numerical solvers based on first-order Taylor linearisations are the most efficient method *overall*.

9.7 Outlook

The benchmarks expressed how ProbDiffEq's implementation of probabilistic numerical solvers has been more efficient than SciPy's methods (on all but one example) and competes with Diffrax's algorithms for the fastest method overall as soon as the differential equation is not a first-order, nonstiff problem. For example, the most numerically efficient solver on a stiff version of the van-der-Pol system was a probabilistic numerical one. For the future development of probabilistic numerical algorithms, this implies that provided one carefully implements numerically stable and scalable versions of the algorithms in "fast" frameworks, the real-time performance of a probabilistic solver is no longer a blocking issue and that the field of probabilistic numerical IVP solvers is ready for applications to real-world problems.

Part IV

Selected topics

Chapter 10

Finite differences

Contents

10.1 Numerical differentiation

Most of this thesis centres around the probabilistic numerical solution of differential equations: estimating an unknown function from derivative information. The present chapter reverses this task. More specifically, it explains a probabilistic model for *numerical differentiation*: estimating derivatives from function evaluations.

Numerical differentiation is helpful for many tasks: for example, for analysing the sensitivity of a computer program to changing one of its input parameters or the numerical simulation of partial differential equations. Numerical differentiation also enables the Taylor-linearisation of ordinary differential equations (recall Chapter 5) in settings where exact Jacobians are unavailable.

Let $\Omega \in \mathbb{R}^\ell$ be a sufficiently regular domain (for example, open, bounded, with a smooth boundary). Let $h : \Omega \to \mathbb{R}$ be a function. We assume that the functional form of h is unknown but that we can evaluate h at arbitrary inputs $x \in \Omega$. For example, such an assumption is satisfied when h is a computer program or when using numerical differentiation to estimate h itself, like in the context of partial differential equations (more details in the upcoming chapter).

Recall the Jacobian Dh of function h. For any direction $\eta \in \mathbb{R}^\ell$,

$$\frac{\partial h(x)}{\partial \eta} := (Dh)(x)\eta \tag{10.1}$$

is the derivative of h in direction η. Let $\Delta x > 0$ be a given increment. In a non-probabilistic setting, the directional derivative of h could be approximated as, e.g.,

$$\frac{\partial h(x)}{\partial \eta} \approx \frac{h(x + \Delta x \cdot \eta) - h(x)}{\Delta x} \tag{10.2a}$$

$$\frac{\partial h(x)}{\partial \eta} \approx \frac{h(x) - h(x - \Delta x \cdot \eta)}{\Delta x} \tag{10.2b}$$

$$\frac{\partial h(x)}{\partial \eta} \approx \frac{h(x + \Delta x/2 \cdot \eta) - h(x - \Delta x/2 \cdot \eta)}{\Delta x} \tag{10.2c}$$

according to forward (Equation (10.2a)), backward (Equation (10.2b)), and central differences (Equation (10.2c)), respectively.

The present chapter deals with *probabilistic numerical differentiation*. Why should we use a probabilistic approach? The advantages of probabilistic numerical differentiation over non-probabilistic numerical differentiation mirror the advantages of probabilistic numerical solvers over non-probabilistic numerical solvers: The need for explicit prior distributions enables discussing the modelling aspects of numerical algorithms, a perspective that usually receives little attention; furthermore, the setup via a probabilistic model allows combining numerical differentiation with surrounding computations: for example, an upcoming chapter describes how to combine probabilistic numerical differentiation with probabilistic numerical initial value problem solvers to construct solvers for partial differential equations.

The remainder of this chapter proceeds as follows. Section 10.2 explains probabilistic numerical differentiation. Section 10.3 modifies probabilistic numerical differentiation to more closely replicate the efficiency of finite difference algorithms and discusses other practical considerations, such as model selection. Section 10.4 links a software-implementation of probabilistic numerical differentiation.

10.2 Probabilistic numerical differentiation

Let $m_h : \Omega \to \mathbb{R}$ be known and $C_h : \Omega \times \Omega \to \mathbb{R}$ be symmetric and positive definite.

$\mathcal{D}$ shall be a linear differential operator. The reader may think of directional derivatives, $\mathcal{D} = \frac{\partial}{\partial \eta}$, but $\mathcal{D}$ could also be a gradient, divergence, curl, Laplacian, or any other differential operator. When applying $\mathcal{D}$ to a function with two arguments, $\mathcal{D}C_h$ applies $\mathcal{D}$ to the first argument and $\mathcal{D}^*C_h$ applies $\mathcal{D}$ to the second argument.

Assume that m_h and C_h are sufficiently regular that $\mathcal{D}m_h$, $\mathcal{D}C_h$, $\mathcal{D}^*C_h$, and

$\mathcal{D}\mathcal{D}^* C_h$ are continuous and that $\mathcal{D}\mathcal{D}^* C_h$ is positive definite. For example, if C_h is translation-invariant, continuity of $\mathcal{D}\mathcal{D}^* C_h$ and positive definiteness of C_h suffices for positive definitess of $\mathcal{D}\mathcal{D}^* C_h$ [177, Chapter 16].

Let $\mathcal{X} := \{x_0, ..., x_K\} \subseteq \Omega$ be a set of spatial grid points. The below exposition uses vectorised notation for the mean function $m_h(\mathcal{X}) := \{m_h(x_k)\}_{k=0}^K \in \mathbb{R}^{K+1}$ and the covariance function $C_h(\mathcal{X}) := \{C_h(x_k, x_{k'})\}_{k,k'=0}^K \in \mathbb{R}^{(K+1)\times(K+1)}$.

While the true functional form of h may be unknown, we may hypothesise that it is a sample of a Gaussian process,

$$h \sim p(h) = \mathrm{GP}(m_h, C_h). \tag{10.3}$$

Then, the joint distribution over $h(x)$ and $\mathcal{D}h(x)$ is Gaussian

$$p(h(x), \mathcal{D}h(x)) = \mathcal{N}\left(\begin{pmatrix} m_h(x) \\ \mathcal{D}m_h(x) \end{pmatrix}, \begin{pmatrix} C_h(x,x) & \mathcal{D}^* C_h(x,x) \\ \mathcal{D}C_h(x,x) & \mathcal{D}\mathcal{D}^* C_h(x,x) \end{pmatrix}\right) \tag{10.4}$$

and the derivative $\mathcal{D}h$ is a Gaussian process,

$$p(\mathcal{D}h) = \mathrm{GP}(\mathcal{D}m_h, \mathcal{D}\mathcal{D}^* C_h). \tag{10.5}$$

In other words, we can always derive the parameters of the marginal distribution over the derivative $p(\mathcal{D}h)$ from the parameters of the distribution over the function $p(h)$. Can we also compute the parameters of the conditional distribution $p(\mathcal{D}h \mid h)$ from the parametrisation of $p(h)$?

Let h be like above and assume a $\rho^2 \geq 0$. Define a random variable $\xi_\mathcal{X}$ that describes evaluations of h on $\mathcal{X}$ corrupted by additive Gaussian noise,

$$p(\xi_\mathcal{X} \mid h(\mathcal{X})) := \mathcal{N}(h(\mathcal{X}), \rho^2 I). \tag{10.6}$$

Manipulating the conditional relationship between $\xi_\mathcal{X}$, h, and $\mathcal{D}h$ implies estimators for interpolation and differentiation as follows:

Computing the conditional distribution $p(h \mid \xi_\mathcal{X})$ from $p(h)$ and $p(\xi_\mathcal{X} \mid h)$ is *interpolation*. More specifically, the conditional $p(h \mid \xi_\mathcal{X})$ is a Gaussian process,

$$p(h \mid \xi_\mathcal{X}) = \mathrm{GP}(W_{\mathrm{int},\mathcal{X}}(\cdot)\xi_\mathcal{X} + b_{\mathrm{int},\mathcal{X}}(\cdot), C_{\mathrm{int},\mathcal{X}}(\cdot,\cdot)), \tag{10.7}$$

where $W_{\mathrm{int},\mathcal{X}}$, $b_{\mathrm{int},\mathcal{X}}$, and $C_{\mathrm{int},\mathcal{X}}$ are given by

$$W_{\mathrm{int},\mathcal{X}}(x) := C_h(x,\mathcal{X}) \left[C_h(\mathcal{X},\mathcal{X}) + \rho^2 I\right]^{-1} \tag{10.8a}$$

$$b_{\mathrm{int},\mathcal{X}}(x) := m_h(x) - W_{\mathrm{int},\mathcal{X}}(x) m_h(\mathcal{X}) \tag{10.8b}$$

$$C_{\mathrm{int},\mathcal{X}}(x,x') := C_h(x,x') - W_{\mathrm{int},\mathcal{X}}(x) \left[C_h(\mathcal{X},\mathcal{X}) + \rho^2 I\right] W_{\mathrm{int},\mathcal{X}}(x')^\top. \tag{10.8c}$$

Equation (10.7) is the standard formula for Gaussian process interpolation [for example, 140]. For $\rho^2 \to 0$, $W_{\mathrm{int},\mathcal{X}}(\mathcal{X}) = I_{K+1}$ and $C_{\mathrm{int},\mathcal{X}}(\mathcal{X}) = 0_{K+1}$ hold: the observations $\xi_\mathcal{X}$ are reproduced exactly.

The conditional distribution in Equation (10.7) is a Gaussian process over Ω. The derivative of this Gaussian process is also a Gaussian process,

$$p(\mathcal{D}h \mid \xi_\mathcal{X}) = \mathrm{GP}(W_{\mathrm{diff},\mathcal{X}}(\cdot)\xi_\mathcal{X} + b_{\mathrm{diff},\mathcal{X}}(\cdot), C_{\mathrm{diff},\mathcal{X}}(\cdot,\cdot)), \tag{10.9}$$

with parameters

$$\begin{aligned} W_{\mathrm{diff},\mathcal{X}}(x) &:= \mathcal{D}W_{\mathrm{int},\mathcal{X}}(x), & (10.10a)\\ b_{\mathrm{diff},\mathcal{X}}(x) &:= \mathcal{D}b_{\mathrm{int},\mathcal{X}}(x), & (10.10b)\\ C_{\mathrm{diff},\mathcal{X}}(x,x') &:= \mathcal{D}\mathcal{D}^*C_{\mathrm{int},\mathcal{X}}(x,x'). & (10.10c) \end{aligned}$$

Equation (10.9) describes numerical *differentiation*: it estimates $\mathcal{D}h$ from point evaluations $\xi_\mathcal{X}$. Simply put, probabilistic numerical differentiation involves fitting a Gaussian process to $\xi_\mathcal{X}$ and differentiating this fit. The result approximates the derivative of h. Computing the quantities in Equation (10.10) requires access to $\mathcal{D}C_h(\cdot,\mathcal{X})$, $\mathcal{D}^*C_h(\mathcal{X},\cdot)$, and $\mathcal{D}\mathcal{D}^*C_h(\cdot,\cdot)$. For known and differentiable C_h, these derivatives are usually available in closed form or via automatic differentiation.

Algorithm 10.1 (Probabilistic numerical differentiation)**.** Assume that we know m_h, C_h, $\mathcal{D}C_h$, $\mathcal{D}^*C_h$, $\mathcal{D}\mathcal{D}^*C_h$, ρ^2, $\mathcal{X}$, and that we have access to a sample from $p(\xi_\mathcal{X})$. Diffferentiate h as follows:

1. Compute the probabilistic numerical differentiation parameters from Equation (10.10).
2. Apply Equation (10.9).

Return the mean and covariance of Equation (10.9).

The differentiation parameters in Equation (10.10) do not depend on $\xi_\mathcal{X}$ but only on the mean and covariance functions, their derivatives, and the spatial point set. The parameters in Equation (10.9) can be pre-computed.

If $\mathcal{D}$ is not a differential operator but an integral operator, Algorithm 10.1 recovers *probabilistic numerical integration* [126], also known as Bayesian quadrature, Bayesian cubature, or kernel quadrature; see the book by Hennig et al. [79] for an overview. Like probabilistic numerical differentiation, probabilistic numerical integration relies on fitting a Gaussian process to sampled evaluations and integrating this approximation, which is usually possible in closed form. The difference between probabilistic numerical differentiation and integration is that Algorithm 10.1 applies $\mathcal{D}$ and probabilistic numerical integration applies an integral operator.

By construction, probabilistic numerical differentiation (Algorithm 10.1) is a probabilistic numerical method in the definition by Cockayne et al. [37].

Proposition 10.2. *Algorithm 10.1 is a probabilistic numerical method according to the definition by Cockayne et al. [37].*

Proof. The proof of Example 2.3 in the paper by Cockayne et al. [37] applies almost verbatim, except that the quantity of interest is not $h \mapsto \int h \, \mathrm{d}\mu$ but $h \mapsto \mathcal{D}h$. □

For a zero mean $m_h = 0$ and noise-free function evaluations (that is, $\rho^2 \to 0$), the *differentiation matrix* $W_{\mathrm{diff},\mathcal{X}}(\mathcal{X})$ occurs in a method for solving partial differential equations with radial basis functions called *unsymmetric collocation*, or *Kansa's method* [81, 88, 147]. This class of algorithms connects to pseudospectral solvers for partial differential equations [53, Chapter 42]. A related algorithm, symmetric collocation [51, 52], estimates not $p(\mathcal{D}h \mid \xi_{\mathcal{X}})$ but conditional distributions like $p(h \mid \mathcal{D}h(\mathcal{X}) = 0)$, and has been translated into a probabilistic numerical solver for partial differential equations by Cockayne et al. [36]. A linear-time implementation of symmetric collocation for spatiotemporal problems is another contribution of this thesis and the content of the next chapter.

By construction, probabilistic numerical differentiation yields marginal distributions that are consistent with those in Equation (10.4),

$$p(\mathcal{D}h) = \int p(\mathcal{D}h, \xi_{\mathcal{X}}) \, \mathrm{d}\xi_{\mathcal{X}} = \mathrm{GP}(\mathcal{D}m_h(\cdot), \mathcal{D}\mathcal{D}^* C_h(\cdot,\cdot)). \tag{10.11}$$

In other words, Algorithm 10.1 provides an alternative way of accessing the distribution of $p(\mathcal{D}h)$, namely, via the conditional distribution $p(\mathcal{D}h \mid \xi_{\mathcal{X}})$. In many settings, for example, when constructing partial differential equation solvers, access to the conditional $p(\mathcal{D}h \mid \xi_{\mathcal{X}})$ simplifies the generative model for partial differential equation solvers (see upcoming chapter).

One of the advantages of Algorithm 10.1 over, say, central finite differences, is that the statistical description of Algorithm 10.1 provides a natural language for uncertainty quantification, which is less straightforward for traditional finite difference algorithms. Further advantages include the transparency of modelling assumptions behind the algorithm – every user must provide m_h and C_h – and the freedom to numerically differentiate on scattered point sets instead of being forced to equidistant grids.

The disadvantages of Algorithm 10.1 compared to traditional finite difference formulas are computational complexity and numerical stability: computing $W_{\mathrm{diff},\mathcal{X}}$ involves solving a dense linear system with $K + 1$ variables, which usually requires $O(K^3)$ floating-point operations. Cubic complexity is prohibitively expensive for large point sets. Numerical ill-conditioning is also problematic: the covariance matrix $C_h(\mathcal{X}, \mathcal{X})$ often has a condition number that grows exponentially with the number of points in the point set [146]. In contrast, finite difference formulas avoid these problems, and the next section discusses the required modifications.

10.3 Probabilistic numerical finite differences

For now, we only intend to compute the values of the derivatives at the grid points,

$$p(\mathcal{D}h(\mathcal{X}) \mid \xi_{\mathcal{X}}) = \mathcal{N}(W_{\text{diff},\mathcal{X}}(\mathcal{X})\xi_{\mathcal{X}} + b_{\text{diff},\mathcal{X}}(\mathcal{X}), C_{\text{diff},\mathcal{X}}(\mathcal{X})) \tag{10.12}$$

instead of on arbitrary point sets. The modifications below also apply to reconstructing off-grid derivatives efficiently, but the notation becomes more complicated. Remark 10.4 discusses off-grid probabilistic numerical finite differences. We proceed in two steps to reconstruct the computational efficiency of finite difference formulas:

The first step is to pretend conditional independence given $\xi_{\mathcal{X}}$,

$$p(\mathcal{D}h(\mathcal{X}) \mid \xi_{\mathcal{X}}) \approx \prod_{k=0}^{K} p(\mathcal{D}h(x_k) \mid \xi_{\mathcal{X}}) \tag{10.13}$$

$$= \prod_{k=0}^{K} \mathcal{N}(W_{\text{diff},\mathcal{X}}(x_k)\xi_{\mathcal{X}} + b_{\text{diff},\mathcal{X}}(x_k), C_{\text{diff},\mathcal{X}}(x_k)). \tag{10.14}$$

Generally, the derivatives of h on neighbouring points x_k and x_{k+1} are not independent. However, approximating them as being independent (Equation (10.13)) has positive implications for the implementation of numerical differentiation: All terms in Equation (10.13) can be assembled in parallel, and the covariance matrix $C_{\text{diff},\mathcal{X}}(\mathcal{X})$ becomes diagonal.

The second step is to restrict the computation of each parameter set

$$\left\{(W_{\text{diff},\mathcal{X}}(x_k), b_{\text{diff},\mathcal{X}}(x_k), C_{\text{diff},\mathcal{X}}(x_k))\right\}_{k=0}^{K} \tag{10.15}$$

to "small" subsets of $\mathcal{X}$: Let $k \in \{0, ..., K\}$, and define the stencil

$$\text{loc}_s(x_k) = \{s \text{ nearest neighbours of } x_k \text{ in } \mathcal{X}\}. \tag{10.16}$$

This stencil replaces the role of $\mathcal{X}$ in Equations (10.9) to (10.10), in the sense that we approximate

$$p(\mathcal{D}h(x_k) \mid \xi_{\mathcal{X}}) \approx p(\mathcal{D}h(x_k) \mid \xi_{\text{loc}_s(x_k)}) \tag{10.17}$$

which implies replacing

$$W_{\text{diff},\mathcal{X}}(x_k) \approx W_{\text{diff},\text{loc}_s(x_k)}(x_k) \tag{10.18a}$$
$$b_{\text{diff},\mathcal{X}}(x_k) \approx b_{\text{diff},\text{loc}_s(x_k)}(x_k) \tag{10.18b}$$
$$C_{\text{diff},\mathcal{X}}(x_k, x_k) \approx C_{\text{diff},\text{loc}_s(x_k)}(x_k, x_k) \tag{10.18c}$$

in Equation (10.10).

Algorithm 10.3 (Probabilistic numerical finite differences)**.** Same as Algorithm 10.1, but using Equations (10.17) to (10.18) instead of Equations (10.9) to (10.10).

Instead of solving one linear system with $K + 1$ variables, Algorithm 10.3 solves $K + 1$ linear systems with $s + 1$ variables. A small s usually suffices; see Figure 10.1.

Remark 10.4 (Off-grid numerical differences). To approximate $p(\mathcal{D}h(x) \mid \xi_{\mathcal{X}})$ for $x \notin \mathcal{X}$, proceed as follows.

1. Choose a stencil size $s \in \mathbb{N}$ and find the s closest neighbours of x in $\mathcal{X}$, $\text{loc}_s(x) \subseteq \mathcal{X}$.
2. Compute the parameters as in Algorithm 10.3.

In the same way that Algorithm 10.1 reduces to the special case of unsymmetric collocation for $\rho^2 \to 0$, $m_h = 0$, and when evaluating the derivative at $\mathcal{X}$, Algorithm 10.3 reduces to *radial-basis-function-generated finite-difference formulas* [48, 56, 155, 162] under the same restrictions. The connection to finite-difference formulas stems from the fact that if the covariance would correspond to a polynomial kernel and the mesh $\mathcal{X}$ were equidistant and one-dimensional, the differentiation weights would equal the standard finite difference coefficients [55]. The advantage of the more general, kernel-based finite difference approximation over polynomial coefficients is a more robust approximation for non-uniform grid points and in higher dimensions [56, 162]. The Bayesian point of view does not only add uncertainty quantification in the form of the differentiation error covariance $C_{\text{diff},\mathcal{X}}$ respectively $C_{\text{diff},\text{loc}_s(x_k)}$, but also reveals how a user must calibrate a probabilistic model for appropriate numerical differentiation. Since numerical differentiation derives from Gaussian process interpolation, standard model selection techniques (e.g. type-II maximum likelihood estimation) apply.

10.4 Software

An implementation of probabilistic numerical differentiation and probabilistic numerical finite differences in Python (based on JAX [28]) can be installed from

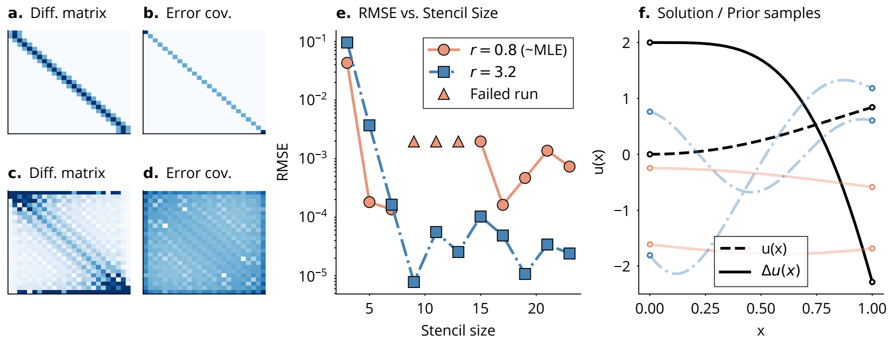

Figure 10.1: *Discretise the Laplacian with a local and global approximation:* The target is the Laplacian $\mathcal{D} = \Delta$ of $u(x) = \sin(\|x\|^2)$ (f). *Left:* Sparsity pattern of the differentiation matrix and error covariance matrix for the localised approximation (a, b) and the global approximation (c, d) on $N = 25$ points. *Centre:* The root-mean-square error between Δu and its approximation decreases with an increased stencil size. The approximation breaks down for larger stencils (e), likely due to ill-conditioned kernel Gram matrices. A maximum likelihood estimate of the input scale $r \in \mathbb{R}$ of the square exponential kernel $k(x, y) = e^{-r^2\|x-y\|^2}$ based on data $u_x(x)$ alone does not necessarily lead to well-conditioned system matrices, nor does it generally imply a low RMSE (e). *Right:* Samples from the prior GP u_x for both length scales are shown next to the solution and the target function (f; the colours match the colours in the RMSE plot). *Increasing stencil sizes improves the accuracy until stability concerns arise.*

`https://probfindiff.readthedocs.io/`.

The documentation for this code demonstrates further practical considerations for implementing probabilistic numerical differentiation.

10.5 Conclusion

Numerical differentiation is essential for many scientific applications, for example, computing the sensitivity of a computer program to changing one of the inputs. Instead of relying on non-probabilistic finite differences, probabilistic numerical differentiation fits a Gaussian process to point-evaluations of a function and differentiates this fit, which is possible in closed form. This procedure is similar to collocation methods and probabilistic numerical integration. To replicate the efficiency of finite difference methods, assume conditional independence and use only the s closest neighbours of a point x_k to compute a numerical derivative.

Chapter 11

Method of lines

Contents

11.1 Partial differential equations

All previous parts of this manuscript discussed the basics of probabilistic numerical solvers for initial value problems based on ordinary differential equations. This chapter develops a class of probabilistic numerical algorithms for the solution of initial value problems based on *partial* differential equations (PDEs). PDEs are a common way of modelling physical interdependencies between temporal and spatial variables. With the recent advent of physics-informed neural networks [139], neural operators [106, 112], and neural ordinary/partial differential equations [33, 64, 142], PDEs have rapidly gained popularity in the machine learning community, too.

Let $\Omega \subset \mathbb{R}^\ell$ be a domain with boundary $\partial\Omega$. Let F, h, and g be given (potentially nonlinear) functions. $\mathcal{D}$ and $\mathcal{B}$ shall be linear (differential) operators. The reader may think of the Laplacian, $\mathcal{D} = \sum_{i=1}^d \frac{\partial^2}{\partial x_i^2}$, but the algorithm is not restricted to this case. Solving PDEs amounts to approximating an unknown function $u : [0,1] \times \Omega \to \mathbb{R}$ that satisfies

$$\frac{\partial}{\partial t} u(t,x) = F(u(t,x), \mathcal{D}u(t,x)), \tag{11.1}$$

for $t \in [0, 1]$ and $x \in \Omega$, subject to the initial condition

$$u(0, x) = h(x), \quad x \in \Omega, \tag{11.2}$$

and the boundary condition

$$\mathcal{B}u(t, x) = g(x), \quad (t, x) \in [0, 1] \times \partial\Omega. \tag{11.3}$$

The differential operator $\mathcal{B}$ is usually the identity (Dirichlet conditions), the derivative along normal coordinates (Neumann conditions), or a combination of both. Without a loss of generality, this PDE assumes that the solution u is scalar-valued. We impose this restriction for a simpler notation, and modifications for vector-valued problems follow the same strategy as in previous chapters. Therefore, they are omitted here.

Assume that F, h, and g are sufficiently well-behaved that a unique solution u exists. One common assumption is that Ω must be open and bounded, and that $\partial\Omega$ must be differentiable, but requirements vary across differential equations [50]. Except for only a few problems, PDEs do not admit closed-form solutions, and numerical approximations become necessary.

11.2 Probabilistic numerical method of lines

One common strategy for solving PDEs, called the *method of lines* (MOL) [149], first discretises the spatial domain Ω with a grid $x_0, ..., x_K$, and then uses this grid to approximate the differential operator $\mathcal{D}$ with a matrix-vector product

$$[(\mathcal{D}u)(t, x_k)]_{k=0}^K \approx D[u(t, x_k)]_{k=0}^K, \tag{11.4}$$

for a matrix $D \in \mathbb{R}^{(K+1)\times(K+1)}$. For example, the one-dimensional Laplacian can be approximated with central differences, which leads to (ignoring boundary conditions)

$$D := \frac{1}{(\Delta x)^2} \begin{pmatrix} -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \end{pmatrix}. \tag{11.5}$$

The previous Chapter 10 discussed numerical differentiation more thoroughly. Replacing the differential operator $\mathcal{D}$ with the matrix D turns the PDE into a system of ordinary differential equations (ODEs) over the state $[u(t, x_k)]_{k=0}^K \in \mathbb{R}^{K+1}$. Standard ODE solvers can then numerically solve the resulting initial value problem.

This approach has one central problem. Discretising the spatial domain and only then applying an ODE solver turns the PDE solver into a pipeline of two numerical algorithms, each of which with its respective approximation error, instead of a single algorithm. This is bad because, due to this serialisation, the error estimates returned

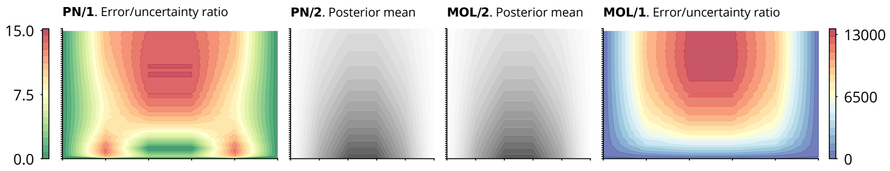

Figure 11.1: Posterior means and error/uncertainty ratios of the probabilistic numerical MOL ("PNMOL"; left) and MOL with a conventional probabilistic solver (right) on a fine time grid (y-axis) and a coarse space grid (x-axis) for the heat equation. The means are indistinguishable (PN/2, MOL/2). MOL is poorly calibrated (error/uncertainty ratios $\sim 10^5$; MOL/1), but PNMOL acknowledges all inaccuracies (PN/1).

by the ODE solver are unreliable. The algorithm lacks crucial information about whether the spatial grid consists of, say, $N = 4$ or $N = 10^7$ points. Intuitively, a coarse spatial grid puts a lower bound on the overall precision, even if the ODE solver uses small time steps. However, since this is not "known" to the ODE solver, not even to a probabilistic one, it may waste computational resources by needlessly decreasing its step size and may deliver (severely) overconfident uncertainty estimates (for example, Figure 11.1). Such overconfident uncertainty estimates are essentially useless to a practitioner, which complicates using the method-of-lines approach for the simulation of differential equations, for example, in the context of parameter estimation problems.

Instead of considering a PDE solver as a combination of two algorithms, numerical differentiation and an ODE solver, we treat it as a single estimation problem as follows. Let $p(\varphi)$ be a prior distribution over the space of functions $\{\varphi : [0,1] \times \Omega \to \mathbb{R}\}$. Let $t_0, ..., t_N$ and $x_0, ..., x_K$ be collocation points on $[0, 1]$ and Ω, respectively. As discussed in Chapter 2, any sample from the posterior distribution

$$p\left(\varphi \;\middle|\; A^{\mathrm{PDE}}, A^{\mathrm{Initial}}, A^{\mathrm{Boundary}}\right), \tag{11.6}$$

approximately solves the PDE. Here, A^{PDE}, A^{Initial}, and A^{Boundary} are shorthand for

$$\begin{aligned} A^{\mathrm{PDE}} &:= \left\{\frac{\partial}{\partial t}\varphi(t_n, x_k) = F\left(\varphi(t_n, x_k), [\mathcal{D}\varphi](t_n, x_k)\right)\right\}_{n,k=0}^{N,K} && (11.7\text{a}) \\ A^{\mathrm{Initial}} &:= \{\varphi(0, x_k) = h(x_k)\}_{k=0}^{K} && (11.7\text{b}) \\ A^{\mathrm{Boundary}} &:= \{[\mathcal{B}\varphi](t, x_{k'}) = g(x_{k'})\}_{n,k'=0}^{N,K'} && (11.7\text{c}) \end{aligned}$$

and $x_0, ..., x_{K'}$ are the points in $\{x_0, ..., x_K\}$ that lie on the boundary of Ω. We call any approximation of Equation (11.6) the *probabilistic numerical PDE solution*, and any algorithm that computes such an approximation a *probabilistic numerical PDE solver*.

Without a loss of generality, it will suffice to compute the distribution of the PDE solution on the grid, i.e., only target the conditional distribution

$$p\left([\varphi(t_n, x_k)]_{n,k=0}^{N,K} \mid A^{\text{PDE}}, A^{\text{Initial}}, A^{\text{Boundary}}\right), \tag{11.8}$$

because off-grid evaluations can be reconstructed by interpolation.

The rest of this chapter explains the implementation of probabilistic numerical PDE solvers for the setting where φ is a time-space separable Gaussian process with a Markovian time component. The resulting algorithm will share its overall structure with the non-probabilistic numerical method of lines, but the uncertainty quantification incorporates both space and time errors equally. To this end, Section 11.3 combines the state-space implementation of spatiotemporal Gauss–Markov processes [159] with the setup in Chapter 3, Section 11.4 blends the probabilistic numerical differentiation from Chapter 10 with linearisation and correction schemes from Chapter 7, and the subsequent sections evaluate the algorithm and compare it to previous work.

11.3 Spatiotemporal prior process

For the rest of this chapter, $p(\varphi)$ shall be a spatiotemporal Gaussian process, which we denote by $p(u)$ (because it models the PDE solution).

The overarching goal is to develop an algorithm that matches the computational efficiency of non-probabilistic numerical method-of-lines solvers, who inherit their efficiency from the ODE solver that they employ. To mimic this inheritance, we develop a scenario where time and space can be handled separately. Thus, the following assumption is integral for the probabilistic numerical method of lines.

Assumption 11.1. *Assume a Gaussian process prior with a time/space-separable covariance,*

$$p(u = u(t, x)) = GP(0, \gamma^2 k_t \otimes k_x) \tag{11.9}$$

for some constant output scale $\gamma > 0$. The kernel $k_t \otimes k_x$ is the product kernel, defined as

$$(k_t \otimes k_x)(t, t', x, x') = k_t(t, t')k_x(x, x'). \tag{11.10}$$

The subscripts in k_t and k_x label the temporal and spatial kernels and do not indicate partial derivatives (like they sometimes do in the PDE literature).

Compared to the traditional method of lines, where the temporal and spatial dimensions are treated independently and with black-box methods, Assumption 11.1 is mild: even though the covariance is separable, the algorithm still starts with a single, global

Gaussian process. Assumption 11.1 allows choosing temporal kernels that (eventually) lead to a fast algorithm:

Assumption 11.2. *Assume that k_t admits that any Gaussian process with covariance k_t, $\upsilon \sim GP(0, k_t)$, can be written as the output of a state-space model involving a linear, time-invariant stochastic differential equation,*

$$\upsilon(t) = H\tilde{\upsilon}(t), \;\; \mathrm{d}\tilde{\upsilon}(t) = A\tilde{\upsilon}(t)\,\mathrm{d}t + B\,\mathrm{d}w(t), \;\; p(\tilde{\upsilon}(0)) = \mathcal{N}(m_0, C_0), \quad (11.11)$$

for some hidden state $\tilde{\upsilon}$. The parameters A, B, H, m_0, and C_0 derive from k_t [144, Chapter 12], and w is a one-dimensional Wiener process with unit diffusion.

Assumption 11.2 is satisfied, for instance, by the integrated Wiener process or the Matèrn process; many more examples are given in Chapter 12 of the book by Särkkä and Solin [144]. Together, Assumption 11.1 and Assumption 11.2 unlock the machinery of probabilistic numerical ODE solvers.

Next, we add spatial correlations to the prior SDE model. As discussed in the context of Equation (11.8), it suffices to compute the PDE solution at the grid points. Mirroring the procedure for the non-probabilistic numerical method of lines, we start with a spatial discretisation and let the time variable remain continuous (for now). The space-discretised version of a spatiotemporal Gaussian process $u \sim \mathrm{GP}(0, \gamma^2 k_t \otimes k_x)$ is a vector-valued temporal Gaussian process,

$$p(\mathbf{u}(t)) := p\left([u(t, x_k)]_{k=0}^{K}\right) = \mathrm{GP}(0, \gamma^2 k_t \otimes \mathbf{K}), \quad \mathbf{K} := [k_x(x_i, x_j)]_{i,j=0}^{K}, \quad (11.12)$$

which inherits a state-space model representation from k_t [159]:

Lemma 11.3. *Let k_t be a covariance function that satisfies Assumption 11.2. The process $\mathbf{u}$ from Equation* (11.12) *is the output of the state-space model*

$$\mathbf{u} = (H \otimes I)\tilde{\mathbf{u}} \quad (11.13a)$$

$$\mathrm{d}\tilde{\mathbf{u}}(t) = (A \otimes I)\tilde{\mathbf{u}}(t)\,\mathrm{d}t + \gamma(B \otimes (\mathbf{K})^{1/2})\,\mathrm{d}\mathbf{w}(t), \quad (11.13b)$$

$$p(\tilde{\mathbf{u}}(0)) = \mathcal{N}(m_0 \otimes \mathbf{1}, \gamma^2 C_0 \otimes \mathbf{K}), \quad (11.13c)$$

where $\mathbf{w}$ is a $(K+1)$-dimensional Wiener process with unit diffusion, the matrix $I \in \mathbb{R}^{(K+1)\times(K+1)}$ is the identity, and $m_0 \otimes \mathbf{1}$ is a vertical stack of $K+1$ copies of m_0. The parameters H, A, B, m_0, C_0 are from Assumption 11.2.

Lemma 11.3 states how a spatiotemporal prior (Assumption 11.1) may be restricted to a spatial grid without losing the computational benefits of Markovian temporal priors.

Processes like $\tilde{\mathbf{u}}$ in Lemma 11.3 have a Kronecker-factorised time discretisation,

$$p(\tilde{\mathbf{u}}(t+\Delta t) \mid \tilde{\mathbf{u}}(t)) = \mathcal{N}([\Phi(\Delta t) \otimes I]\tilde{\mathbf{u}}(t), \gamma^2[\Sigma(\Delta t) \otimes \mathbf{K}]) \tag{11.14}$$

and together with $p(\tilde{\mathbf{u}}(0))$ as in Lemma 11.3, the sequential representation of $\tilde{\mathbf{u}}$ is complete. Φ and Σ are the transition matrices for the temporal process in Assumption 11.2 and depend on A and B. If k_t corresponds to an integrated Wiener process prior, they are the transition matrices from Chapter 3. The effect of Kronecker factorisations in transition matrices on the numerical efficiency of the solver has been a part of Chapter 8.

11.4 Information model

Next, we discuss how to impose the constraints on the spatiotemporal prior distribution. For the moment, assume two simplifications:

1. The absence of boundary conditions, which is the case if, for instance, Ω is the sphere or another manifold without a boundary.
2. Semilinear dynamics, i.e., a PDE of the form

$$\frac{\partial u(t,x)}{\partial t} = \mathcal{D}u(t,x) + f(u(t,x)) \tag{11.15}$$

instead of the more general form in Equation (11.1).

Omitting the boundary conditions and considering a semilinear problem simplifies the notation. Neither assumption restricts the generality of the method. Remark 11.4 explains how to incorporate boundary conditions and Remark 11.5 how to handle fully nonlinear problems; both require a minimal modification of what is explained next.

The following setup connects to Chapter 10: Let k_x be the (spatial) covariance kernel from Assumption 11.1. Assume that k_x is sufficiently differentiable so that $\mathcal{D}\mathcal{D}^* k_x$ is a positive definite function, which is usually the case if k_x is sufficiently differentiable; detailed assumptions have been listed in Chapter 10.

Fix t and recall the abbreviation $\mathbf{u}(t) := [u(t,x_k)]_{k=0}^K$ from Section 11.3. Probabilistic numerical differentiation as in Chapter 10 yields the approximation

$$p\left([(\mathcal{D}u)(t,x)]_{k=0}^K \mid \mathbf{u}(t)\right) = \mathrm{GP}(W(x)\mathbf{u}(t), E(x,x')) \tag{11.16}$$

for differentiation matrices W and E that depend on $\mathcal{D}$, k_x, and the spatial collocation points. Details are in Chapter 10. Evaluating the Gaussian process in Equation (11.16) at the collocation points $x_0, ..., x_K$, Equation (11.16) reads

$$p\left([\mathcal{D}\mathbf{u}](t) \mid \mathbf{u}(t), \mathbf{e}(t)\right) = \delta(\mathbf{W}\mathbf{u}(t) + \mathbf{e}(t)) \tag{11.17}$$

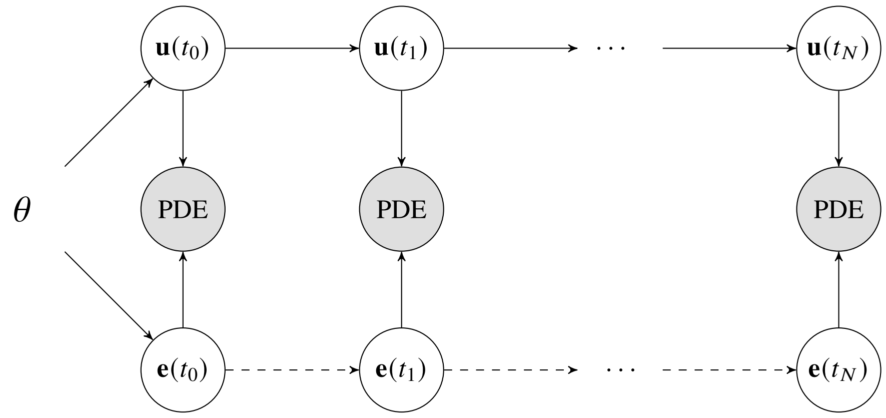

Figure 11.2: The existence of the bottom row distinguishes the probabilistic numerical version from non-probabilistic numerical method-of-lines implementations. θ includes the algorithm parameters, such as the Gaussian process kernel and the spatial collocation points. The correlations indicated by the dashed lines can be ignored to reduce the computational complexity (discussed in the context of Equation (11.22)).

for $\mathbf{W} = [W(x_k)]_{k=0}^K$ and $\mathbf{e}(t) = [e(t, x_k)]_{k=0}^K$; e is a spatiotemporal Gaussian process

$$p(e(t,x)) = \mathrm{GP}(0, \gamma^2 k_t \otimes E). \tag{11.18}$$

Restricted to $x_0, ..., x_K$, e is a vector-valued temporal Gaussian process

$$p(\mathbf{e}(t)) = \mathrm{GP}(0, \gamma^2 k_t \otimes \mathbf{E}), \quad \mathbf{E} = [E(x_i, x_j)]_{i,j=0}^K \tag{11.19}$$

with a temporal structure inherited from u. In other words, $\mathbf{e}$ has same state-space representation as $\mathbf{u}$ after exchanging $\mathbf{K}$ for $\mathbf{E}$ in Lemma 11.3.

The probabilistic PDE solution in Equation (11.8) can be reconstructed from

$$p\left(\begin{bmatrix}\mathbf{e}(t_n)\\ \mathbf{u}(t_n)\end{bmatrix}_{n=0}^N \;\middle|\; \left[\frac{\mathrm{d}\mathbf{u}(t_n)}{\mathrm{d}t} = \mathbf{W}\mathbf{u}(t_n) + \mathbf{e}(t_n) + f(\mathbf{u}(t_n))\right]_{n=0}^N, \; \mathbf{u}(0) = \mathbf{h}\right), \tag{11.20}$$

with $\mathbf{h} = [h(x_k)]_{k=0}^K$, by marginalising over $\mathbf{e}$. The magnitude of the process e describes the temporal evolution of the accumulated numerical differentiation error. Loosely speaking, it acts as a latent force on the ordinary differential equation that captures how much the ODE differs from the PDE; in other words, how much information is lost by spatial discretisation. Tracking $\mathbf{e}$ is what distinguishes the proposed algorithm from non-probabilistic numerical PDE solvers. The complete setup is in Figure 11.2.

The conditional distribution in Equation (11.20) factorises sequentially because both $\mathbf{u}$ and $\mathbf{e}$ are Markovian and because all constraints are conditionally independent given $\mathbf{u}$ and $\mathbf{e}$. It is not Gaussian when f is nonlinear, but all estimation and approximation methods in Chapters 3 to 7 apply. The unknown parameter γ can be estimated with quasi-maximum-likelihood estimation and in the same fashion as in Chapter 7; deriving the precise formulas is left to the reader.

The computational complexity of estimating Equation (11.20) is $O(N(2K)^3)$, where the factor $(2K)^3$ stems from tracking the $\mathbf{e}$ and $\mathbf{u}$ jointly, each of which has $K+1$ components. To reduce the computational complexity, simplify the model for $\mathbf{e}$ as

$$p(\mathbf{e}(t)) = \mathrm{GP}(0, \gamma^2 k_t \otimes \mathbf{E}) \approx \mathrm{GP}(0, \gamma^2 k_{\text{white}} \otimes \mathbf{E}) \tag{11.21}$$

where k_{white} is a white-noise kernel. The effect of this approximation is that $\mathbf{e}$ does not need to be tracked in the state-space model, in which case the observation model in Equation (11.17) becomes

$$\delta(\mathbf{W}\mathbf{u}(t) + \mathbf{e}(t)) \approx \mathcal{N}(\mathbf{W}\mathbf{u}(t), \gamma^2 \mathbf{E}). \tag{11.22}$$

The right-hand side in Equation (11.22) does not depend on $\mathbf{e}$. The dimension of the state-space cuts in half, and the computational complexity reduces to $O(NK^3)$ at the price of discarding temporal correlations in the latent force. The cubic complexity in the spatial variable could be reduced with further factorisation or similar techniques that are common in accelerating Gaussian process estimation; but this is future work.

This concludes the discussion of the setup underlying probabilistic numerical PDE solvers. Before continuing with related work in Section 11.5, we mention the modifications for boundary conditions and fully nonlinear equations:

Remark 11.4 (Boundary conditions). The effect of the boundary operator, $\mathcal{B}$, can be handled identically to the differential operator $\mathcal{D}$, with probabilistic numerical differentiation. Thus, including boundary information involves tracking another latent state that derives identically to $\mathbf{e}$ but with $\mathcal{B}$ instead of $\mathcal{D}$. The constraints remain an affine combination of the hidden states, and inference remains tractable. The only difference to the algorithm above is that two latent states are tracked instead of one.

Remark 11.5 (Nonlinear equations). If the differential equation is not semilinear but nonlinear, that is, if it has dynamics $F(u, \mathcal{D}u)$ instead of $\mathcal{D}u + f(u)$, linearisation remains similar. The only difference is that the nonlinear dynamics must be linearised as a function of **u** and **e**,

$$F_{\text{transformed}}(\mathbf{u}, \mathbf{e}) := F(\mathbf{u}, \mathbf{W}\mathbf{u} + \mathbf{e}), \tag{11.23}$$

instead of as a function of **u** only. The rest is identical to the semilinear case.

Naturally, if the PDE involves second- or higher-order time derivatives, the PDE solution and estimation strategy change in the usual way. If the PDE is not constrained by an initial condition but by an initial and a terminal condition, the strategies discussed in the upcoming Chapter 12 will be applicable.

11.5 Related work

Method of lines The literature on the method of lines is covered by, e.g., Schiesser [149]. Dereli and Schaback [43], Hon et al. [82] combine collocation with the method of lines. None of the above exploits the correlations between spatial and temporal errors; however, the (general) significance of estimating the interplay of both error sources has been recognised by Berzins [20], Berzins et al. [21], Lawson et al. [105].

Probabilistic numerical solvers for stationary PDEs Cockayne et al. [36], Owhadi [128, 129], Raissi et al. [137, 138] describe a probabilistic numerical solver for PDEs that relates to symmetric collocation approaches from numerical analysis. Chen et al. [34] extend the ideas to non-linear PDEs via maximum-a-posteriori estimation. The present algorithm can be loosely regarded as an efficient implementation of the method by Chen et al. [34] for time-dependent problems, but only if combined with the algorithm in the upcoming Chapter 12. In its present form, it is more suitably described as an extension of probabilistic numerical IVP solvers to spatiotemporal problems.

Probabilistic numerical solvers for time-dependent PDEs Wang et al. [175] continue the work of Chkrebtii et al. [35] in constructing an ODE/PDE initial value problem solver that uses (approximate) conjugate Gaussian updating at each time-step. Duffin et al. [49] solve time-dependent PDEs by discretising the spatial domain with finite elements and by applying ensemble and extended Kalman filtering in time. They build on the work on “statistical finite elements” by Girolami et al. [66]. Abdulle and Garegnani [2], Conrad et al. [38] compute probabilistic numerical PDE solutions by randomly perturbing non-probabilistic numerical solvers. All of the above discard the uncertainty associated with discretising the differential operator. Some papers achieve ODE-solver-like complexity for time-dependent problems [35, 49, 175], while others compute a continuous-time posterior [34, 36, 128, 129, 137, 138]. The PDE solver presented in this chapter does both.

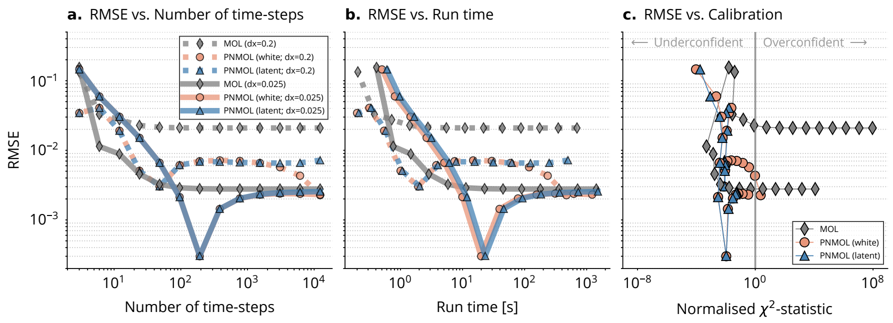

Figure 11.3: Work vs. precision vs. calibration of the probabilistic numerical MOL (PNMOL) in the latent-force version, which tracks **e** (blue) and the white-noise version (orange), compared to a traditional probabilistic numerical ODE solver combined with conventional MOL (grey), on the spatiotemporal Lotka-Volterra model. Two kinds of curves are shown: one for a coarse (dotted) and one for a fine spatial mesh (solid). A reference is computed by discretising the spatial domain with a ten times finer mesh and solving the ODE with backward differentiation formulas. The root-mean-square error of both methods stagnates once a certain accuracy is reached, but PNMOL appears to reach a slightly lower RMSE for $\Delta x = 0.2$ (left, middle). The run time of PNMOL-white is comparable to that of MOL, and the run time of PNMOL-latent is slightly longer (middle). The calibration of PNMOL, measured in the normalised χ^2-statistic of the Gaussian posterior (so that the “optimum” is 1, not d), is close to 1 but slightly underconfident. With decreasing time steps, MOL is poorly calibrated.

11.6 Experiments

Next, we investigate the numerical uncertainty quantification of the probabilistic numerical PDE solver. As a first experiment, we solve a spatiotemporal Lotka-Volterra model [80], i.e. nonlinear predator-prey dynamics with spatial diffusion, on a range of temporal and spatial resolutions. From the results in Figure 11.3, it is evident how the spatial accuracy limits the overall accuracy; but also how the combination of a probabilistic numerical ODE solver with a spatial discretisation fails to quantify numerical uncertainty reliably. At any parameter configuration, it is *either* the spatial or the temporal discretisation that dominates the error. Decreasing the time step alone not only lets the error stagnate but also worsens the calibration because the ODE solver does not know how bad the spatial approximation is.

To further examine which one of Δx or Δt dominates the approximation, we consider a second example: a spatiotemporal SIR model [57]. We investigate more formally how increasing either, time-resolution versus space-resolution, leads to a low overall error. The results are in Figure 11.4 and confirm the findings from Figure 11.3 above.

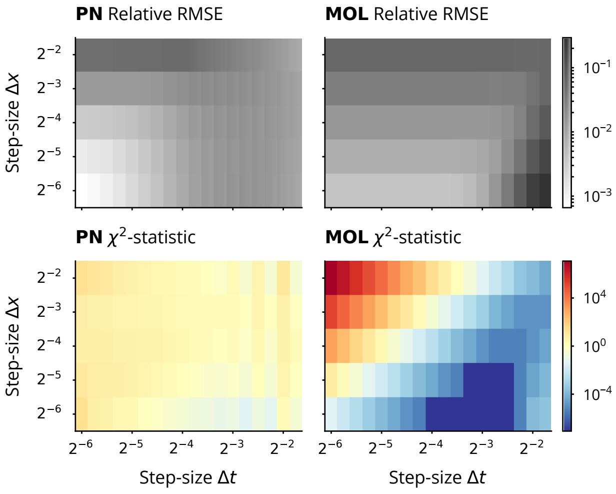

Figure 11.4: *Which error dominates?* The relative root-mean-squared error is only small if *both* Δt and Δx are small, which affects the probabilistic numerical solver ("PNMOL"; top left) as well as traditional MOL combined with a probabilistic IVP solver (top right). MOL is severely overconfident for large Δx and small Δt (bottom right), while PNMOL delivers a calibrated posterior distribution (bottom left).

Traditional probabilistic numerical ODE solvers with conventional MOL are unaware of the true, global approximation error. The PDE solver implementation is not, despite being equally accurate and requiring equal computational effort.

11.7 Conclusion

In conclusion, the derivation of a probabilistic numerical solver for spatiotemporal problems – i.e., partial differential equations – closely follows the construction of probabilistic numerical solvers for ordinary differential equations. The central difference is that probabilistic numerical differentiation induces a latent force in the ODE that results from discretising the PDE in space. However, the latent force can be estimated jointly with the PDE solution and at minimal computational overhead. Altogether, and unlike traditional PDE solvers, the probabilistic numerical method of lines unlocks the quantification of spatiotemporal correlations in an approximate PDE solution, all while preserving the efficiency of adaptive ODE solvers. This makes it a valuable algorithm in the toolboxes of probabilistic numerical algorithms and may serve as a backbone for latent force models, inverse problems, and differential-equation-centric machine learning.

Chapter 12

Boundary value problems

Contents

12.1 Boundary value problems

This chapter develops a class of algorithms for solving *boundary value problems* (BVPs) based on ordinary differential equations. BVPs are about finding a function y that satisfies a system of ordinary differential equations

$$\frac{\mathrm{d}^2 y(t)}{\mathrm{d}t^2} = f(y(t)), \quad t \in [0, 1], \tag{12.1}$$

and a set of boundary conditions

$$y(0) = y_0, \quad y(1) = y_1. \tag{12.2}$$

The vector field and the boundary conditions are known. Without a loss of generality, we assume a simplified problem:

- ⋄ We only consider a scalar, second-order, autonomous differential equation in Equation (12.1). Modifications for first- or higher-order and non-autonomous

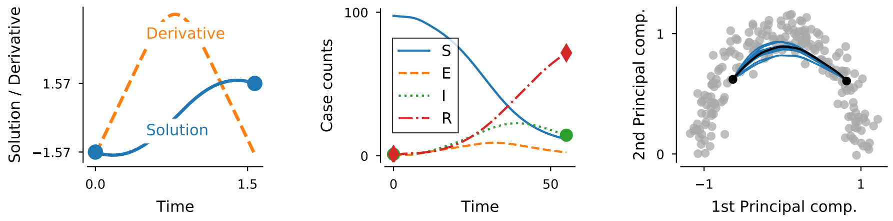

Figure 12.1: Recovering the trajectory of a pendulum between two positions is a BVP (left). A lack of initial values can be made up by boundary values in an SEIR model (middle). Straight lines on manifolds give distance measures and demand solving a BVP (right; depicted are the mean and ten samples of the probabilistic solution; principal components of 1000 MNIST images of the digit “1”). In all three figures, the ball/diamond markers express boundary conditions.

equations are identical to the previous chapters but would complicate the notation. Vector-valued problems follow the strategy discussed in Chapter 8; spatiotemporal problems would follow the logic from Chapter 11 but are not central to this chapter.

⋄ We only consider constraints of the type in Equation (12.2), i.e., linear and separable boundary conditions. The procedure for other types of separable boundary conditions is in Remark 12.1; non-separable boundary conditions are an open problem.

Loosely speaking, solving BVPs amounts to following the law of a dynamical system when “connecting two points”. This setting is relevant to several scientific applications. We consider three examples as motivation, all of which are in Figure 12.1.

First, recovering the trajectory of a pendulum between two positions amounts to solving the ordinary differential equation $\ddot{y}(t) = -9.81\sin(y(t))$ subject to the positions as boundary conditions. If the positions were “interpolated” without the ordinary differential equation knowledge, the output would be physically meaningless.

Second, BVPs arise when inferring the evolution of the case counts of people who fall victim to an infectious disease [e.g., 11]. A lack of counts of (a specific subset of) non-infected people at the initial time-point can be made up for by available counts of infected people at the final time-point of the integration domain.

Third, efficient manifold learning necessitates repeated computation of (geodesic) distances between two points, which amounts to solving BVPs [9, 44].

Let $\{t_0, ..., t_N\} \subseteq [0, 1]$ be a set of collocation points. Assume a prior distribution $p(\varphi)$ over the functions from $[0, 1]$ to $\mathbb{R}$ (or $\mathbb{R}^d$ if the problem is vector-valued). As explained by Chapter 2, the *probabilistic numerical BVP solution* is the conditional distribution

$$p\left(\varphi \;\middle|\; \left\{\frac{\mathrm{d}^2\varphi(t_n)}{\mathrm{d}t^2} = f(\varphi(t_n))\right\}_{n=0}^{N},\ \varphi(0) = y_0,\ \varphi(1) = y_1\right). \tag{12.3}$$

If the differential equation is not autonomous or of first- or higher-order, then the conditional distribution changes like in the previous chapters. Any algorithm that approximates the probabilistic numerical BVP solution is a *probabilistic numerical BVP solver*. This chapter discusses a probabilistic numerical BVP solver that is similar to the class of algorithms discussed in previous chapters but with a few differences due to handling boundary conditions instead of initial conditions.

12.2 Bridge priors

The first step in estimating the probabilistic numerical BVP solution – the conditional distribution in Equation (12.3) – is handling the boundary conditions. In the following derivation, abbreviate the probabilistic numerical BVP solution as

$$p(\varphi \mid \mathrm{ODE}, \mathrm{BC}) := p\left(\varphi \;\middle|\; \left\{\frac{\mathrm{d}^2\varphi(t_n)}{\mathrm{d}t^2} = f(\varphi(t_n))\right\}_{n=0}^{N},\ \varphi(0) = y_0,\ \varphi(1) = y_1\right) \tag{12.4}$$

where "ODE" stands for the differential equation residual, and "BC" abbreviates the boundary conditions. The BVP solution refactors as

$$p(\varphi \mid \mathrm{ODE}, \mathrm{BC}) = \frac{p(\varphi, \mathrm{ODE}, \mathrm{BC})}{p(\mathrm{ODE})p(\mathrm{BC})} = \frac{p(\mathrm{ODE} \mid \varphi)}{p(\mathrm{ODE})} p(\varphi \mid \mathrm{BC}) \tag{12.5}$$

due to Bayes' theorem as well as conditional independence of the boundary conditions and the differential equation residual given φ. Equation (12.5) expresses that in order to approximate the BVP solution, we first compute the conditional

$$p(\varphi \mid \mathrm{BC}) := p(\varphi \mid \ \varphi(0) = y_0,\ \varphi(1) = y_1) \tag{12.6}$$

and then use this conditional as a prior for conditioning on the differential equation residual being zero.

The above strategy works for any prior. Still, the remainder of this chapter concentrates on the case where φ is the same integrated Wiener process prior that has been central to the previous chapters, $p(\varphi) = p(Y(t))$. For integrated Wiener processes (and other Gauss–Markov priors), posterior distributions like the one in Equation (12.6) are available via a single forward-pass because the boundary conditions are affine functions of the state; refer back to Chapter 3 for detailed instructions. We call the

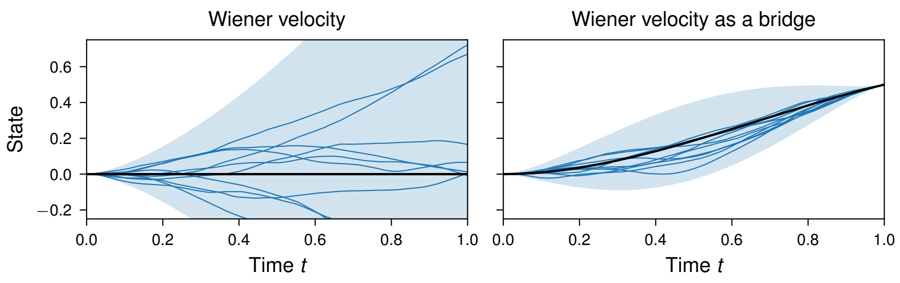

Figure 12.2: Wiener velocity model, which is a once-integrated Wiener process, in its original form (left) and bridged to hit value 0.5 at time $t = 1$ (right). Depicted are the mean, three marginal standard deviations, and ten samples from the process.

resulting stochastic process an *integrated Wiener bridge process* because it is an integrated Wiener process with fixed boundary information, similar to a Brownian bridge, for example. Figure 12.2 shows such an integrated Wiener bridge process, and Figure 12.3 summarises the overall procedure.

Remark 12.1 (Nonlinear boundary conditions). If the boundary conditions are not affine, linearise them with any of the strategies from Chapter 5.

12.3 Maximum-a-posteriori estimation

Once the boundary conditions are incorporated, proceed with the differential equation information. For affine differential equations, this is possible in a single forward pass with sequential Gaussian estimation. Nonlinear differential equations necessitate efficient approximation.

Assume a fixed grid and recall the notation $Y(t_{0:N}) \coloneqq \{Y(t_0), \dots, Y(t_N)\}$. Denote the integrated Wiener bridge process by $Y_b(t)$.

While the exact parametrisation of the probabilistic numerical BVP solution is intractable, we can approximate the maximum-a-posteriori (MAP) estimate

$$\text{MAP} \coloneqq \arg \max_{\xi \in \mathbb{R}^{(N+1)\times(\nu+1)}} p\left(Y_b(t_{0:N}) = \xi \;\middle|\; \left\{Y_b^{(2)}(t_n) = f(Y_b^{(0)}(t_n))\right\}_{n=0}^{N}\right), \quad (12.7)$$

with a state-space-model-based implementation of optimisation algorithms like the Gauss-Newton scheme, which has been discussed in Chapter 5. Applied to computing the BVP solution, the Gauss–Newton algorithm is as follows:

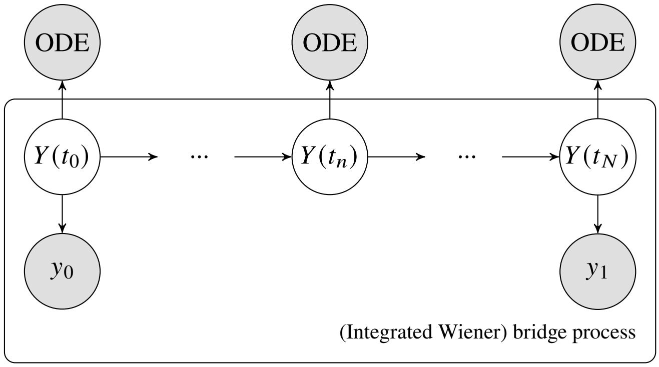

Figure 12.3: Condition on the boundary information to construct a bridge prior. Then, use the bridge process to solve the BVP.

Algorithm 12.2 (MAP-estimation of the BVP solution with Gauss–Newton)**.** Assume a grid $t_0, ..., t_N$, a sequential decomposition of the Bridge process on this grid, a differential equation with vector field f, and an initial guess ξ_0. Let e_q be the qth row of an identity matrix with $\nu + 1$ rows and columns. To compute the Gauss–Newton approximation of the MAP estimate, proceed as follows:

1. Initialise $\xi = \xi_0$.
2. Repeat until convergence:
   (a) Extract the quantity corresponding to the zeroth derivative $Y_b^{(0)}(t_{0:N})$ of the Bridge process from ξ, $\hat{\xi} := \xi e_0^\top$.
   (b) Linearize the vector field f around $\hat{\xi}$; each row of $\hat{\xi}$ corresponds to one grid point.
   (c) Solve the affine estimation problem with sequential Gaussian estimation, using preconditioning and Cholesky parametrisation like in Chapter 7.
   (d) Set ξ to the posterior mean and repeat.

Return the most recent posterior mean and covariance estimate.

Algorithm 12.2 implements the iterated (extended) Kalman smoother, which is equivalent to the Gauss–Newton algorithm [17]. Under mild assumptions on the non-linearity of f and the magnitude of the objective at the optimum, Gauss–Newton methods are locally convergent with linear rate [98]. Instead of Gauss–Newton, other optimisation algorithms could be used (refer back to Chapter 5); however, studying the feasibility of other algorithms is left for future work. Iterative linearisation of differential equations combined with closed-form solutions of the equations is known as *quasilinearisation* in the literature on differential equation solvers [e.g., 115]. As such, the general procedure of a probabilistic numerical BVP solver implements quasilinearisation but in a probabilistic framework.

The rest of this chapter discusses how to tune the algorithm by automatically selecting parameters of the prior, collocation points, and optimisation parameters:

- ◇ Like all quasilinearisation algorithms, computing the MAP estimate of the BVP solution requires an initial guess. Different options are in Section 12.4.
- ◇ The choice of the mesh $t_0, ..., t_N$ affects the accuracy of the solution; an automatic mesh-refinement scheme is in Section 12.5.
- ◇ The prior process depends on three unknowns: the output scale γ, the initial mean m_0, and the initial covariance $C_0(\gamma)$. Section 12.6 explains their calibration.

Sections 12.4 to 12.6 are independent of each other and can be read in any order.

12.4 Initialisation

Algorithm 12.2 needs to be initialised with an initial guess ξ_0. The choice of initialisation is essential: not only does the number of iterations depend on the proximity of the initial guess to the optimum, but BVPs often allow multiple solutions, and the algorithm can find only one of them [96, p. 10]. Non-probabilistic numerical solvers outsource this issue to the user by expecting that an initial guess is provided. For example, at the time of this writing, the BVP solvers in SciPy, Matlab, and DifferentialEquation.jl require the user to pass a vector of initial guesses of the solution at an initial grid to the algorithm [12, 13, 14]. While the same strategy is possible for the probabilistic numerical solver, there are natural alternatives that are not available to non-probabilistic methods. We can roughly group them as follows, presented in order of increasing computational complexity.

12.4.1 Initialising with the prior mean

Since we put computational effort into deriving a useful prior distribution (in the bridge process), we may use this information to derive an initial guess as the mean of the integrated Wiener bridge,

$$\xi_0 := E[Y_b(t_{0:N})], \tag{12.8}$$

which is available by marginalisation. If the backward transitions of the bridge prior have been derived in a previous forward pass, these marginals can be computed in a single backward pass. The most obvious benefit of using this initialisation is its simplicity, combined with the fact that the boundary conditions are always satisfied.

12.4.2 Initialising with user-provided guess

The behaviour of initialising non-probabilistic algorithms, which require a user to pass an initial guess – we may call it $\tilde{\xi}$ – can be replicated with the conditional mean

$$\xi_0 = E[Y_b(t_{0:N}) \mid \tilde{\xi}] \tag{12.9}$$

which, if $Y_b(t_{0:N})$ is represented by a terminal distribution together with a sequence of backward transition densities, is available with a backward and a subsequent forward pass. The main benefit of this approach is its similarity to the initialisation of non-probabilistic algorithms. It generalises such an initialisation in the sense that $\tilde{\xi}$ could be modelled as being subject to additive Gaussian noise or only consist of partial observations, which helps when the initial guess is uninformed (which is common).

12.4.3 Initialising with a local linearisation pass

If a user-provided guess is not available, the initial guess can be computed with a local linearisation pass,

$$\xi_0 \approx E\left[Y_b(t_{0:N}) \,\middle|\, \left\{Y_b^{(2)}(t_n) = f(Y_b^{(0)}(t_n))\right\}_{n=0}^{N}\right] \tag{12.10}$$

where the approximation stems from locally linearising the nonlinear vector field at the predicted mean at each step. This is equivalent to the extended Kalman filter [143] and the linearisation of choice for initial value problem solvers; refer back to Chapters 5 and 7. Again, if $Y_b(t_{0:N})$ is represented by a terminal distribution together with a sequence of backward transition densities, this amounts to one backward and a subsequent forward pass. Since the vector field is evaluated at each step, this is generally more expensive than initialising with a user-provided guess.

The main feature of this approximation is that the user does not need to provide an initial guess for a solution that one generally does not know much about: Once the prior bridge has been constructed, the initialisation of the Gauss–Newton algorithm is fully automatic. If required, the local linearisation pass can be combined with fitting a user-provided guess to increase the accuracy of the initial linearisation point, which yields an algorithm reminiscent of one by Schmidt et al. [150]. Ultimately, choosing between initialisation strategies only matters to the extent that it is sufficiently close to a fixed point of the Gauss-Newton algorithm, and all of the above strategies are viable.

Of course, a fixed point of the Gauss–Newton method is not necessarily a reliable

BVP solution: its accuracy depends on the number and distribution of mesh points. The following Section 12.5 develops a principled and probabilistic approach to error control and mesh refinement in the BVP solver.

12.5 Mesh refinement

So far, the mesh was assumed as given. The larger the size of this mesh is, the more accurate the solution becomes. However, the computational cost grows linearly with the mesh size. Low error tolerances thus require smart meshing via error control. There are natural candidates for error estimators, all connecting to the probabilistic formulation of solving BVPs. Sections 12.5.1 to 12.5.3 discuss those candidates; Section 12.5.4 uses them to refine the mesh, and Section 12.5.5 demonstrates numerical feasability.

12.5.1 Error estimate via standard deviations

The output of the Gauss–Newton algorithm is a Gaussian process, which can be evaluated at any point in the domain of the boundary value problem. Its associated standard deviation is a natural error estimate. The advantage over the alternatives explained below is that it comes (essentially) for free by interpolating the posterior distribution. A potential downside of this intrinsic error estimator is its dependence on the calibration of a hyperparameter (more on this in Section 12.6).

12.5.2 Error estimate via residuals

The probabilistic numerical BVP solution is constructed by conditioning the integrated Wiener process $Y(t)$ on attaining consistently small values in its residual

$$\mathcal{R}_{f,t} := \frac{\mathrm{d}^2 Y^{(0)}(t)}{\mathrm{d}t^2} - f(Y^{(0)}(t)) = Y^{(2)}(t) - f(Y^{(0)}(t)). \tag{12.11}$$

Recall that if $Y^{(0)}(t)$ were the true BVP solution, this residual would be zero on the whole domain. Thus, the magnitude of the residual of the mean of the probabilistic numerical BVP solution quantifies the error, which is also a common approach in traditional, non-probabilistic algorithms (for instance [96] or [11, Section 9.5.1]). By interpolating the BVP solution, the residual can be evaluated on off-grid points.

12.5.3 Error estimates via probabilistic residuals

Considering the full posterior *distribution* instead of only the posterior mean suggests how the residual is a deterministic transformation of a random variable. Thus – in principle – a random variable might make a more appropriate model for the residual error than a point estimate. For a Gaussian process posterior $Y(t)$, the law of $\mathcal{R}_{f,0:N}$

is intractable in general because the vector field is nonlinear. But (re)using a linearised version of the vector field unlocks a Gaussian approximation: An upper bound of the probability of $\|\mathcal{R}_{f,t}\|$ exceeding some tolerance,

$$p\left(\|\mathcal{R}_{f,t}\|^2 > \mathrm{tol}^2\right) < \left(\mathrm{Trace}[\mathrm{Cov}(\mathcal{R}_{f,t})] + \|E(\mathcal{R}_{f,t})\|_2^2\right)/\mathrm{tol}^2, \tag{12.12}$$

is due to the Markov inequality and another approach to error control. The numerator of the right-hand side will be treated as an error estimator in the benchmarks below.

The main difference to the point estimate is that the probabilistic version punishes magnitude *and uncertainty* in the residual instead of only magnitude. However, this quantity will reveal itself as underconfident in the benchmarks below and does, therefore, not play a role in the simulation studies at the end of this chapter.

12.5.4 From error estimates to a finer mesh

All three options, which we denote by a generic $e : [0,1] \to [0,\infty)$ from now on, estimate the pointwise error of the approximation, that is, the approximation error at a given t. For mesh refinement, however, it is more instructive to consider the accumulated error between two grid points t_n and t_{n+1},

$$\epsilon_n := \left(\int_{t_n}^{t_{n+1}} \|e(t)\|_2^2 \,\mathrm{d}t\right)^{1/2}, \quad n = 0, ..., N-1. \tag{12.13}$$

The integral that underlies ϵ_n can usually not be computed in closed form but needs to be approximated by a numerical integration scheme. We use Bayesian quadrature (BQ) [32]. BQ allows us to place quadrature nodes freely in each domain $[t_n, t_{n+1})$. It also provides the option of tailoring an integration kernel to e. For instance, the following reproducing kernel Hilbert spaces (RKHSs) are known [164]: (i) the RKHS of ν-times integrated Wiener process priors $Y(\cdot)$ is the Sobolev space of $(\nu+1)$-times weakly differentiable functions; (ii) under some regularity assumptions on the ODE vector field, as well as on the (assumed to be) unique solution of the ODE, the RKHS of the residual $\mathcal{R}_{f,t}$ is the Sobolev space of ν-times weakly differentiable functions. Therefore, we base the BQ scheme on a $(\nu - 1/2)$th order Matérn prior, which has the same native space as the residual [177]. (In practice, we use an exponentiated quadratic kernel for $\nu > 3$ since its kernel embeddings are more accessible [31, Appendix J]).

Once the accumulated error has been estimated, mesh refinement proceeds as follows. If each ϵ_n is below a user-provided threshold, the BVP solution is sufficiently accurate, and the mesh does not have to be refined. If at least one ϵ_n is larger than the threshold, the mesh must be refined. We introduce new grid points on those intervals where ϵ_n is too large. Assuming that the integrated error is of order $\rho > 0$, $\epsilon_n \in O((\Delta t)^\rho)$, splitting the interval into two equally large parts reduces the error by a factor $2^{-\rho}$, and splitting it into three equal parts by a factor $3^{-\rho}$. We introduce

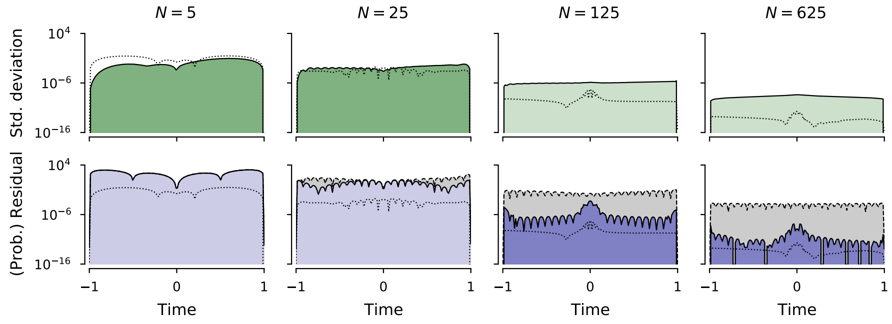

Figure 12.4: *Error estimation on the seventh testproblem in [117].* Evaluated at $N = 5$ (left), $N = 25$ (centre left), $N = 125$ (centre right), and $N = 625$ equidistant grid points (right). Standard deviation (top row) and residual (blue, bottom row), respectively the probabilistic residual (grey, bottom row). True error in black. The “winners” of each column have a darker colour.

either one or two new points depending on how far the integrated error exceeds the required threshold on a sub-interval. Like Kierzenka and Shampine [97], we never introduce more than two grid points per interval at once. For the experiments, which use a ν-times integrated Wiener processes, we choose $\rho = \nu + 1/2$ (which has not been proved yet but seems like a reasonable conjecture in light of Theorem 3 of Tronarp et al. [164] and our experiments).

12.5.5 Comparing error estimates

Which one is the most reliable error estimate? As a testbed, we use the seventh example in a collection of test problems for BVP solvers by Mazzia [117] (which will feature heavily in the remainder of this chapter). The derivative of the solution of this linear BVP approaches a singularity if a specific parameter is chosen sufficiently small (we use 10^{-3}). This poses challenges for error estimators and mesh-refinement strategies.

A reasonable estimate accurately measures the magnitude of the error as well as the location of the largest deviation. Roughly speaking, the magnitude of the error is more informative in the presence of a few grid points, and the location of the error is critical when many points are already in place. The error estimates are visualised in Figure 12.4. They suggest that at high tolerances, the standard deviation is more accurate than the residual; at low tolerances, the situation is reversed. The probabilistic residual is consistently underconfident, which is a trend that is preserved when moving to more challenging setups (see Section 12.7).

With everything explained so far, we can solve BVPs with an algorithm that adaptively refines the mesh when the solution is not sufficiently accurate. The only remaining question is that of selecting the integrated Wiener processes parameters, which affect the calibration of the BVP solution. This is discussed next.

12.6 Parameter estimation

As discussed in Chapter 3, the integrated Wiener process depends on three parameters: the mean and covariance of the initial distribution

$$p(Y(t_0)) = \mathcal{N}(m_0, C_0(\gamma)) \tag{12.14}$$

and the output scale $\gamma \in \mathbb{R}$ of the driving Wiener process. The previously discussed algorithm components assume these parameters to be fixed and known. This chapter discusses their calibration.

Denote the marginal likelihood of the BVP solution as

$$M(\gamma, m_0, C_0(\gamma)) := p\left(\{\mathcal{R}_{f,n} = 0\}_{n=0}^{N}, \mathcal{R}_{y_0} = 0, \mathcal{R}_{y_1} = 0 \mid \gamma, m_0, C_0(\gamma)\right) \tag{12.15}$$

where $\mathcal{R}_{f,n}$ is the residual of the differential equation, and $\mathcal{R}_{y_0}$ as well as $\mathcal{R}_{y_1}$ are the residuals of the initial and terminal conditions, respectively. One way of estimating the parameters is to combine maximum-likelihood estimation with coordinate ascent, which repeats alternating updates over the output scale and the parameters of the initial distribution,

$$\gamma^{\text{new}} := \arg\max_{\gamma} M(\gamma, m_0^{\text{new}}, C_0^{\text{new}}(\gamma^{\text{new}})), \tag{12.16a}$$

$$m_0^{\text{new}}, C_0^{\text{new}}(\gamma) := \arg\max_{m_0, C_0(\gamma)} M(\gamma^{\text{new}}, m_0, C_0(\gamma)), \tag{12.16b}$$

until some stopping criterion is satisfied [181].

A quasi-maximum likelihood update for γ^{new} (Equation 12.16a) is available in closed form as a by-product of the forward-pass of each Gauss–Newton iteration. The precise formula is almost identical to the one in Chapter 7 – the only difference is due to boundary conditions instead of initial conditions – and therefore omitted.

An approximate update for the parameters of the initial distribution can be implemented with the expectation-maximisation (EM) algorithm [42, 156]. The general idea of EM is to maximise a lower bound of Equation (12.16b) instead of maximising it directly by computing alternating E- and M-steps. For parameter estimates in state-space models, the E-step of the EM algorithm amounts to computing the probabilistic numerical BVP solution (on a fixed grid and with fixed parameters) (see, e.g., [121]), a Gaussian approximation of which is available through Gauss–Newton. Denote this

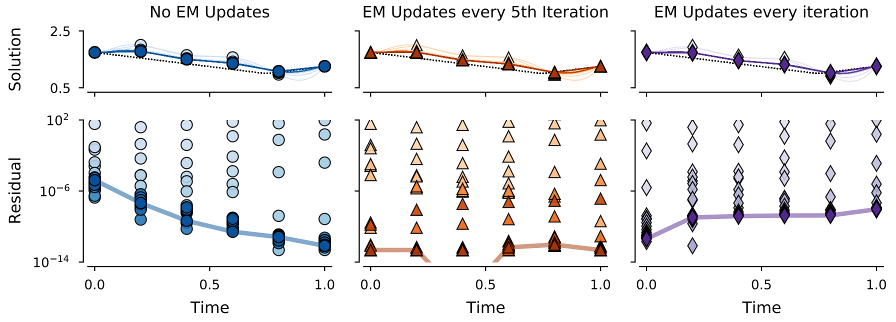

Figure 12.5: *EM helps the IEKS overcome unknown initial conditions.* Depicted are a fixed total of the first 25 Gauss–Newton iterations (light to dark in each respective colour) on $N = 6$ grid points, initialised with a local linearisation using a 7-times integrated Wiener process bridge prior and on the 20th test problem in [117]. Without any EM updates to the initial condition, the convergence of the IEKS is inhibited (left). EM updates every fifth IEKS iteration lead to the residual converging to zero reliably (centre). Too frequent EM updates are not optimal either (right), likely because the underlying Gauss–Newton algorithm fails to converge in a single step.

Gauss–Newton approximation of the maximum-a-posteriori estimate by

$$Y_{\text{MAP}}(t) \sim \mathcal{N}(m_{\text{MAP}}(t), \gamma^2 C_{\text{MAP}}(t, t')). \tag{12.17}$$

The parameters $m_{\text{MAP}}(t), C_{\text{MAP}}(t, t')$ are the result of the Gauss–Newton algorithm. The M-step now consists of [143, Theorem 12.5 and Algorithm 12.7]

$$m_0^{\text{new}} = m_{\text{MAP}}(t_0) \tag{12.18a}$$

$$C_0^{\text{new}}(\gamma) = \gamma^2 C_{\text{MAP}}(t_0, t_0) + (m_0^{\text{new}} - m_0^{\text{old}})(m_0^{\text{new}} - m_0^{\text{old}})^\top. \tag{12.18b}$$

EM steps always increase the likelihood, and for exponential families, convergence to a stationary point of the likelihood function is guaranteed [156, 182]. Thus, computing alternating E- and M-steps until convergence would eventually yield a good estimate of the parameters. But already in the pre-asymptotic regime and for a fixed total number of IEKS iterations, an EM update every few steps helps convergence of the IEKS in subsequent iterations (Figure 12.5).

In conclusion, the following BVP solver emerges.

Algorithm 12.3 (Probabilistic numerical BVP solver)**.** Assume that the following parameters are given: an initial grid $t_0, \dots, t_N$; initial parameter estimates γ, m_0,

$C_0(\gamma)$; an initial guess ξ_0. Then, solve the BVP as follows:

1. Initialise the Gauss–Newton algorithm with any of the strategies from Section 12.4 (some of them do not require ξ_0, in which case that input is not necessary). Initialise the grid.
2. Compute a calibrated BVP solution on an automatically selected mesh by repeating the following steps until convergence:
   (a) Compute a calibrated BVP solution on the current mesh. To this end, initialise $\gamma, m_0, C_0(\gamma)$ and alternate the following coordinate-descent steps until convergence:
      i. Update the output scale: compute a Gauss–Newton estimate, and update γ with the quasi-maximum likelihood estimate
      ii. Update the initial distribution: Alternate Gauss–Newton estimates with EM updates until convergence
   (b) Estimate the error on the intervals between grid points.
   (c) Refine the mesh as appropriate and repeat.

Return the final estimate.

The computational complexity of Algorithm 12.3 is $O(I_{\text{Mesh}} I_{\text{GN}} I_{\text{EM}} N \nu^3)$, where I_{GN} is the number of Gauss–Newton iterations, I_{Mesh} is the number of mesh refinements, and I_{EM} the number of EM updates. In our experiments, we found I_{GN} and I_{EM} to be small, usually bounded by 10. The mesh refinement is designed to make I_{Mesh} as small as possible. Linear complexity in N stems from the state-space implementation of the Gauss–Newton algorithm and could potentially be reduced to $\log N$ by temporal parallelisation [184]. The cubic complexity in ν stems from the matrix-matrix operations that are required in a Gaussian update [143]. An extension to vector-valued problems follows the recommendations from Chapter 8 and inherits their computational complexity.

12.7 Benchmarks

Now that all parts are in place, we evaluate the solver's performance in various scenarios. All experiments are implemented in NumPy [77] and use the CPU of a consumer-level laptop.

An efficient probabilistic numerical method should provide both a reasonable point estimate (through its posterior mean) and a calibrated error estimate (through its posterior covariance). First, the approximation error should decrease rapidly with the number of grid points; we report root-mean-square errors – the lower, the better. Second, the width of the posterior distribution should be representative of the numerical

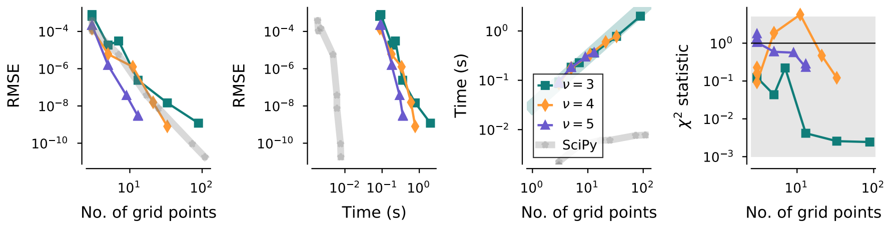

Figure 12.6: *Results on Bratu's problem.* The higher-order solvers converge at least as fast as the SciPy reference (left) and are roughly by factor ~ 100 slower (centre left, centre right; linear complexity reference line in the background). The χ^2-statistic remains within 95% confidence (right; intervals shaded in gray, mean (= 1) in black). The initial grid consists of only three points to display the efficiency of the mesh refinement. The probabilistic numerical solver initialises with local linearisation and uses the standard deviation as an error estimate.

approximation error (which has, to some extent, been shown in Section 12.5 already); we report the dimension-normalised χ^2-statistic [15]. If it is close to 1, the posterior uncertainty is calibrated.

A simulation of Bratu's problem [30] for varying tolerances and orders ν suggests that the solver performs well in both metrics (Figure 12.6). Reassuringly, higher orders of the solver lead to faster convergence, which motivates the analysis of convergence rates akin to the analysis of Tronarp et al. [164] for initial value problem solvers. The experiments also suggest that the uncertainties are calibrated but tend to be under-confident.

Efficient mesh refinement and fast convergence are evident when considering a more comprehensive range of test problems. Figure 12.7 depicts the results of simulating five BVPs (all from Mazzia [117]): the 7th problem approaches a singularity in its derivative, the 23rd problem has a boundary layer at $t_{\max}$, the 24th problem describes a fluid mechanical model of a shock wave, the 28th problem has a corner layer at $t_{\min}$, and the 32nd problem involves fourth-order derivatives. On all problems, the probabilistic numerical solver efficiently computes calibrated posteriors at specified tolerances.

12.8 Related work

How does the proposed algorithm fit into the context of state-of-the-art probabilistic and non-probabilistic BVP solvers?

Headway on the probabilistic solution of BVPs has been made by Hennig and Hauberg [78], Arvanitidis et al. [10], and John et al. [85]. Hennig and Hauberg [78] and Arvanitidis et al. [10] focus on applying BVP solvers to Riemannian statistics.

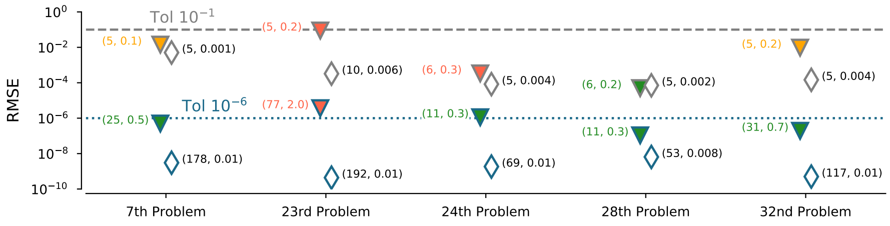

Figure 12.7: *The solver efficiently computes (mostly) calibrated posteriors on many problems.* Probabilistic solver (▽; $\nu = 6$) versus SciPy's BVP solver (◊). Markers are annotated with the number of grid points and runtime (in seconds). The tolerances are 10^{-1} (gray) and 10^{-6} (blue). The closer a coloured marker is to its reference line, the better. The fewer grid points and the less time required, the better. Fill-color describes calibration: χ^2 is within 80 % (green), within 99% (orange), or outside of these ranges (red). SciPy does not allow a notion of calibration.

Table 12.1: Comparison of probabilistic and non-probabilistic BVP solvers.

	Non-probabilistic	Probabilistic (present work)
$O(N)$ achieved by	Sparse matrices	Markov property
Error estimate	Residual (point est.)	Many options, e.g. standard dev.
Initial guess	Mandatory	Optional
Uncertainty quantification	No	Yes

None of these algorithms exploit the state-space structure of the prior with its beneficial computational complexity, nor are they concerned with mesh refinement and the other practical considerations to the extent what has been presented in this chapter.

In terms of accuracy and cost, the present approach should rather be compared to off-the-shelf non-probabilistic BVP solvers: for instance, those implemented in Matlab [96, 97, 154], Python/SciPy [170], and Julia [135]. These toolboxes contain algorithms that implement collocation formulas and gain linear-time complexity from sparse system matrices. The Markov property makes our algorithm equally fast (in terms of the number of grid points N) (Table 12.1).

12.9 Conclusion

This chapter discussed a computationally efficient BVP solver. The algorithm achieves the same linear computational complexity as off-the-shelf solvers, with high-quality point estimates and calibrated uncertainty. Algorithmic parameters can be set automatically by the method, including some that must be manually set for non-probabilistic numerical solvers.

Conclusion

Efficient algorithms for simulating differential equations are crucial for many applications in scientific or industrial environments. Since probabilistic numerical simulation promises the combination of numerical approximation with reliable uncertainty quantification, these environments are the primary candidates for benefitting from a probabilistic approach. And with contributions like the ones discussed in this manuscript, the promises of probabilistic numerical solvers are no longer out of reach.

This thesis derived a numerically stable and scalable implementation of probabilistic numerical solvers for time-dependent differential equation problems. This includes initial value problems based on ordinary differential equations – both scalar and vector-valued – but also partial differential equations and boundary value problems (Chapters 7 to 12). The central strategy was the following: first, begin by factorising conditional distributions of the type

$$p\left(\varphi(t) \;\middle|\; \left\{\frac{\mathrm{d}\varphi(t_n)}{\mathrm{d}t} = f(\varphi(t_n))\right\}_{n=0}^{N},\ \varphi(0) = y_0\right)$$

sequentially with the help of a Gauss–Markov prior; second, carefully combine concepts from numerical linear algebra, algorithmic differentiation, and probabilistic machine learning to achieve an equally efficient, numerically stable, and statistically meaningful algorithm.

For example, the sequence of Chapters 2 to 8 constructed an implementation of an eighth-order probabilistic numerical solver that was competitive to the most efficient state-of-the-art non-probabilistic solvers (in JAX) on the second-order, 14-dimensional Pleiades problem (Chapter 9). This was not only because of the stability and efficiency gains of sequential estimation algorithms but also because the probabilistic solver can seamlessly switch between solving first-order and second-order differential equations – non-probabilistic algorithms cannot do that. Previous to the contributions of this manuscript, implementing eighth-order probabilistic numerical solvers in standard floating-point arithmetic was impossible – and even if it would have been feasible to implement one, the implementation would have been orders of magnitude slower than its non-probabilistic competitors without the factorisations in Chapter 8.

For future applications of probabilistic numerical algorithms, this improved practicality implies that it is now possible to explore how powerful probabilistic numerical algorithms can be when applied to a challenging, real-world problem. And due to the (now) broad availability of probabilistic numerical software in a variety of programming languages, this exploration is no longer a task for a small community but has the potential to reach and include a broader audience, which I, as the author of this thesis, am already looking forward to.

Bibliography

[1] Assyr Abdulle and Giacomo Garegnani. Random time step probabilistic methods for uncertainty quantification in chaotic and geometric numerical integration. *Statistics and Computing*, 30(4):907–932, 2020.

[2] Assyr Abdulle and Giacomo Garegnani. A probabilistic finite element method based on random meshes: A posteriori error estimators and Bayesian inverse problems. *Computer Methods in Applied Mechanics and Engineering*, 384: 113961, 2021.

[3] Suliman Al-Homidan and M Alshahrani. Positive definite Hankel matrices using Cholesky factorization. *Computational Methods in Applied Mathematics*, 9(3):221–225, 2009.

[4] Mauricio A Alvarez, Lorenzo Rosasco, Neil D Lawrence, et al. Kernels for vector-valued functions: A review. *Foundations and Trends® in Machine Learning*, 4(3):195–266, 2012.

[5] Benjamin Ambrosio and Jean-Pierre Françoise. Propagation of bursting oscillations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1908):4863–4875, 2009.

[6] George E Andrews. *The Theory of Partitions*. Number 2. Cambridge University Press, 1998.

[7] Ienkaran Arasaratnam and Simon Haykin. Cubature Kalman filters. *IEEE Transactions on Automatic Control*, 54(6):1254–1269, 2009.

[8] Ienkaran Arasaratnam, Simon Haykin, and Robert J Elliott. Discrete-time nonlinear filtering algorithms using gauss–hermite quadrature. *Proceedings of the IEEE*, 95(5):953–977, 2007.

[9] Georgios Arvanitidis, Lars Kai Hansen, and Søren Hauberg. Latent space oddity: On the curvature of deep generative models. In *6th International Conference on Learning Representations, ICLR 2018*, 2018.

[10] Georgios Arvanitidis, Soren Hauberg, Philipp Hennig, and Michael Schober. Fast and robust shortest paths on manifolds learned from data. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1506–1515. PMLR, 2019.

[11] Uri M Ascher, Robert MM Mattheij, and Robert D Russell. *Numerical solution of boundary value problems for ordinary differential equations*. SIAM, 1995.

[12] DifferentialEquations.jl Authors. Julia's `BoundaryValueProblem` documentation (in `DifferentialEquations.jl`), 2021. URL `https://diffeq.sciml.ai/stable/types/bvp_types/`. Accessed: May 23, 2021.

[13] Matlab Authors. Matlab's `bvpinit` documentation, 2021. URL `https://uk.mathworks.com/help/matlab/ref/bvpinit.html`. Accessed: May 23, 2021.

[14] SciPy Authors. SciPy's `solve_bvp` documentation, 2021. URL `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_bvp.html`. Accessed: May 23, 2021.

[15] Yaakov Bar-Shalom, X Rong Li, and Thiagalingam Kirubarajan. *Estimation With Applications to Tracking and Navigation: Theory, Algorithms and Software*. John Wiley & Sons, 2004.

[16] Bernhard Beckermann. The condition number of real Vandermonde, Krylov and positive definite Hankel matrices. *Numerische Mathematik*, 85(4):553–577, 2000.

[17] Bradley M Bell. The iterated Kalman smoother as a Gauss–Newton method. *SIAM Journal on Optimization*, 4(3):626–636, 1994.

[18] Bradley M Bell and Frederick W Cathey. The iterated Kalman filter update as a Gauss-Newton method. *IEEE Transactions on Automatic Control*, 38(2): 294–297, 1993.

[19] Luis Benet and David P Sanders. TaylorSeries.jl: Taylor expansions in one and several variables in Julia. *Journal of Open Source Software*, 4(36):1043, 2019.

[20] M Berzins. Global error estimation in the method of lines for parabolic equations. *SIAM Journal on Scientific and Statistical Computing*, 9(4):687–703, 1988.

[21] M Berzins, PL Baehmann, JE Flaherty, and J Lawson. Towards an automated finite element solver for time-dependent fluid-flow problems. *The Mathematics of Finite Elements and Application*, 7:181–188, 1991.

[22] Michael Betancourt. A geometric theory of higher-order automatic differentiation. *arXiv preprint arXiv:1812.11592*, 2018.

[23] Jesse Bettencourt, Matthew J Johnson, and David Duvenaud. Taylor-mode automatic differentiation for higher-order derivatives in JAX. 2019.

[24] Ilias Bilionis, Nicholas Zabaras, Bledar A Konomi, and Guang Lin. Multi-output separable Gaussian process: Towards an efficient, fully Bayesian paradigm for uncertainty quantification. *Journal of Computational Physics*, 241:212–239, 2013.

[25] Nathanael Bosch, Philipp Hennig, and Filip Tronarp. Calibrated adaptive probabilistic ODE solvers. In *AISTATS 2021*, 2021.

[26] Nathanael Bosch, Filip Tronarp, and Philipp Hennig. Pick-and-mix information operators for probabilistic ODE solvers. In *International Conference on Artificial Intelligence and Statistics*, pages 10015–10027. PMLR, 2022.

[27] Nathanael Bosch, Philipp Hennig, and Filip Tronarp. Probabilistic exponential integrators. *arXiv preprint arXiv:2305.14978*, 2023.

[28] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. 2018. URL `http://github.com/google/jax`.

[29] Michał Braś, Angelamaria Cardone, and Raffaele D'Ambrosio. Implementation of explicit Nordsieck methods with inherent quadratic stability. *Mathematical Modelling and Analysis*, 18(2):289–307, 2013.

[30] Gh Bratu. Sur les équations intégrales non linéaires. *Bulletin de la Société Mathématique de France*, 42:113–142, 1914.

[31] François-Xavier Briol, Chris J Oates, Mark Girolami, Michael A Osborne, and Dino Sejdinovic. Probabilistic integration. *arXiv:1512.00933v1*, 2015.

[32] François-Xavier Briol, Chris J Oates, Mark Girolami, Michael A Osborne, and Dino Sejdinovic. Probabilistic integration: A role in statistical computation? *Statistical Science*, 34(1):1–22, 2019.

[33] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6571–6583, 2018.

[34] Yifan Chen, Bamdad Hosseini, Houman Owhadi, and Andrew M. Stuart. Solving and learning nonlinear PDEs with Gaussian processes. *Journal of Computational Physics*, 447, 2021.

[35] Oksana A Chkrebtii, David A Campbell, Ben Calderhead, and Mark A Girolami. Bayesian solution uncertainty quantification for differential equations. *Bayesian Analysis*, 11(4):1239–1267, 2016.

[36] Jon Cockayne, Chris Oates, Tim Sullivan, and Mark Girolami. Probabilistic numerical methods for PDE-constrained Bayesian inverse problems. In *AIP Conference Proceedings*, volume 1853, page 060001. AIP Publishing LLC, 2017.

[37] Jon Cockayne, Chris J Oates, Timothy John Sullivan, and Mark Girolami. Bayesian probabilistic numerical methods. *SIAM Review*, 61(4):756–789, 2019.

[38] Patrick R Conrad, Mark Girolami, Simo Särkkä, Andrew Stuart, and Konstantinos Zygalakis. Statistical analysis of differential equations: introducing probability measures on numerical solutions. *Statistics and Computing*, 27(4): 1065–1082, 2017.

[39] Piet De Jong. The diffuse Kalman filter. *The Annals of Statistics*, pages 1073–1083, 1991.

[40] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. The Mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems*, 50 (1):1–18, 2000.

[41] Cédric J Demeure. Fast QR factorization of Vandermonde matrices. *Linear Algebra and its applications*, 122:165–194, 1989.

[42] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[43] Yılmaz Dereli and Robert Schaback. The meshless kernel-based method of lines for solving the equal width equation. *Applied Mathematics and Computation*, 219(10):5224–5232, 2013.

[44] Manfredo Perdigao Do Carmo and J Flaherty Francis. *Riemannian Geometry*, volume 6. Springer, 1992.

[45] G Dobinski. Summierung der reihe $\sum nm/n!$ für $m = 1, 2, 3, 4, 5$. *Arch. für Mat. und Physik*, 61:333–336, 1877.

[46] John R Dormand and Peter J Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[47] Tobin A Driscoll and Richard J Braun. *Fundamentals of Numerical Computation*, volume 154. SIAM, 2017.

[48] Tobin A Driscoll and Bengt Fornberg. Interpolation in the limit of increasingly flat radial basis functions. *Computers & Mathematics with Applications*, 43 (3-5):413–422, 2002.

[49] Connor Duffin, Edward Cripps, Thomas Stemler, and Mark Girolami. Statistical finite elements for misspecified models. *Proceedings of the National Academy of Sciences*, 118(2), 2021.

[50] Lawrence C Evans. *Partial Differential Equations*, volume 19. American Mathematical Society, 2010.

[51] Gregory E. Fasshauer. Solving partial differential equations by collocation with radial basis functions. In *Surface Fitting and Multiresolution Methods*, pages 131–138. University Press, 1997.

[52] Gregory E Fasshauer. Solving differential equations with radial basis functions: multilevel methods and smoothing. *Advances in Computational Mathematics*, 11(2):139–159, 1999.

[53] Gregory E Fasshauer. *Meshfree Approximation Methods with MATLAB*. 2007.

[54] Richard FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal*, 1(6):445–466, 1961.

[55] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706, 1988.

[56] Bengt Fornberg and Natasha Flyer. *A Primer on Radial Basis Functions with Applications to the Geosciences*. SIAM, 2015.

[57] Chunyi Gai, David Iron, and Theodore Kolokolnikov. Localized outbreaks in an S-I-R model with diffusion. *Journal of Mathematical Biology*, 80:1389–1411, 04 2020.

[58] Rui Gao, Filip Tronarp, and Simo Särkkä. Iterated extended Kalman smoother-based variable splitting for L_1-regularized state estimation. *IEEE Transactions on Signal Processing*, 67(19):5078–5092, 2019.

[59] Rui Gao, Filip Tronarp, and Simo Särkkä. Variable splitting methods for constrained state estimation in partially observed Markov processes. *IEEE Signal Processing Letters*, 27:1305–1309, 2020.

[60] Ángel F García-Fernández, Lennart Svensson, Mark R Morelande, and Simo Särkkä. Posterior linearization filter: Principles and implementation using sigma points. *IEEE transactions on signal processing*, 63(20):5561–5573, 2015.

[61] Ángel F García-Fernández, Lennart Svensson, and Simo Särkkä. Iterated posterior linearization smoother. *IEEE Transactions on Automatic Control*, 62 (4):2056–2063, 2016.

[62] Charles William Gear. Runge–Kutta starters for multistep methods. *ACM Transactions on Mathematical Software (TOMS)*, 6(3):263–279, 1980.

[63] Arthur Gelb et al. *Applied Optimal Estimation*. MIT Press, 1974.

[64] Maximilian Gelbrecht, Niklas Boers, and Jürgen Kurths. Neural partial differential equations for chaotic systems. *New Journal of Physics*, 23(4): 043005, 2021.

[65] Stuart Gibson and Brett Ninness. Robust maximum-likelihood estimation of multivariable dynamic systems. *Automatica*, 41(10):1667–1682, 2005.

[66] Mark Girolami, Eky Febrianto, Ge Yin, and Fehmi Cirak. The statistical finite element method (statFEM) for coherent synthesis of observation data and model predictions. *Computer Methods in Applied Mechanics and Engineering*, 375, 2021.

[67] Gene H Golub and Charles F Van Loan. *Matrix Computations*. John Hopkins University Press, 2013.

[68] Mohinder S Grewal and Angus P Andrews. *Kalman Filtering: Theory and Practice with MATLAB*. John Wiley & Sons, 2014.

[69] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[70] Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Mathematics of computation*, 69(231):1117–1130, 2000.

[71] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: Recent Developments and Applications*, 6(6):83–107, 1989.

[72] John Guckenheimer. Dynamics of the van der Pol equation. *IEEE Transactions on Circuits and Systems*, 27(11):983–989, 1980.

[73] AK Gupta and T Varga. Characterization of matrix variate normal distributions. *Journal of Multivariate Analysis*, 41(1):80–88, 1992.

[74] Arjun K Gupta and Daya K Nagar. *Matrix Variate Distributions*. Chapman and Hall/CRC, 2018.

[75] Kjell Gustafsson, Michael Lundh, and Gustaf Söderlind. A PI stepsize control for the numerical solution of ordinary differential equations. *BIT Numerical Mathematics*, 28(2):270–287, 1988.

[76] Ernst Hairer, Syvert P Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems*. Springer, 1993.

[77] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825): 357–362, 2020.

[78] Philipp Hennig and Søren Hauberg. Probabilistic solutions to differential equations and their application to Riemannian statistics. In *Artificial Intelligence and Statistics*, pages 347–355, 2014.

[79] Philipp Hennig, Michael A Osborne, and Hans P Kersting. *Probabilistic Numerics*. Cambridge University Press, 2022.

[80] Elizabeth E Holmes, Mark A Lewis, JE Banks, and RR Veit. Partial differential equations in ecology: spatial interactions and population dynamics. *Ecology*, 75(1):17–29, 1994.

[81] YC Hon and Robert Schaback. On unsymmetric collocation by radial basis functions. *Applied Mathematics and Computation*, 119(2-3):177–186, 2001.

[82] YC Hon, Robert Schaback, and M Zhong. The meshless kernel-based method of lines for parabolic equations. *Computers & Mathematics with Applications*, 68(12):2057–2067, 2014.

[83] Michael F Hutchinson. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 18(3):1059–1076, 1989.

[84] Kazufumi Ito and Kaiqi Xiong. Gaussian filters for nonlinear filtering problems. *IEEE transactions on automatic control*, 45(5):910–927, 2000.

[85] David John, Vincent Heuveline, and Michael Schober. GOODE: A Gaussian off-the-shelf ordinary differential equation solver. In *International Conference on Machine Learning (ICML)*, 2019.

[86] Matthew J Johnson, Jesse Bettencourt, Dougal Maclaurin, and David Duvenaud. Taylor-made higher-order automatic differentiation. 2019. URL `https://github.com/google/jax/files/6717197/jet.pdf`.

[87] Simon J Julier, Jeffrey K Uhlmann, and Hugh F Durrant-Whyte. A new approach for filtering nonlinear systems. In *Proceedings of 1995 American Control Conference-ACC'95*, volume 3, pages 1628–1632. IEEE, 1995.

[88] Edward J Kansa. Multiquadrics—a scattered data approximation scheme with applications to computational fluid-dynamics—II solutions to parabolic, hyperbolic and elliptic partial differential equations. *Computers & Mathematics with Applications*, 19(8-9):147–161, 1990.

[89] Toni Karvonen, Motonobu Kanagawa, and Simo Särkkä. On the positivity and magnitudes of Bayesian quadrature weights. *Statistics and Computing*, 29(6): 1317–1333, 2019.

[90] Jacob Kelly, Jesse Bettencourt, Matthew James Johnson, and David Duvenaud. Learning differential equations that are easy to solve. In *Advances in Neural Information Processing Systems 33 Pre-Proceedings*, 2020.

[91] H. Kersting and P. Hennig. Active uncertainty calibration in Bayesian ODE solvers. *UAI 2016*, 2016.

[92] Hans Kersting, Nicholas Krämer, Martin Schiegg, Christian Daniel, Michael Tiemann, and Philipp Hennig. Differentiable likelihoods for fast inversion of 'likelihood-free' dynamical systems. In *International Conference on Machine Learning*, pages 5198–5208. PMLR, 2020.

[93] Hans Kersting, Timothy John Sullivan, and Philipp Hennig. Convergence rates of Gaussian ODE filters. *Statistics and Computing*, 30(6):1791–1816, 2020.

[94] Agnan Kessy, Alex Lewin, and Korbinian Strimmer. Optimal whitening and decorrelation. *The American Statistician*, 72(4):309–314, 2018.

[95] Patrick Kidger. *On Neural Differential Equations*. PhD thesis, University of Oxford, 2021.

[96] Jacek Kierzenka and Lawrence F Shampine. A BVP solver based on residual control and the Matlab PSE. *ACM Transactions on Mathematical Software (TOMS)*, 27(3):299–316, 2001.

[97] Jacek Kierzenka and Lawrence F Shampine. A BVP solver that controls residual and error. *JNAIAM J. Numer. Anal. Ind. Appl. Math*, 3(1-2):27–41, 2008.

[98] O Knoth. A globalization scheme for the generalized Gauss–Newton method. *Numerische Mathematik*, 56(6):591–607, 1989.

[99] Ivan Kolár, Peter W Michor, and Jan Slovák. *Natural operations in differential geometry*. Springer Science & Business Media, 2013.

[100] Nicholas Krämer and Philipp Hennig. Stable implementation of probabilistic ODE solvers. *arXiv preprint arXiv:2012.10106*, 2020.

[101] Nicholas Krämer and Philipp Hennig. Linear-time probabilistic solutions of boundary value problems. *Advances in Neural Information Processing Systems*, 34, 2021.

[102] Nicholas Krämer, Nathanael Bosch, Jonathan Schmidt, and Philipp Hennig. Probabilistic ODE solutions in millions of dimensions. *ICML 2022*, 2022.

[103] Nicholas Krämer, Jonathan Schmidt, and Philipp Hennig. Probabilistic numerical method of lines for time-dependent partial differential equations. In *International Conference on Artificial Intelligence and Statistics*, pages 625–639. PMLR, 2022.

[104] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[105] Jane Lawson, Martin Berzins, and Peter M Dew. Balancing space and time errors in the method of lines for parabolic equations. *SIAM Journal on Scientific and Statistical Computing*, 12(3):573–594, 1991.

[106] Zongyi Li, Nikola Borislavov Kovachki, Kamyar Azizzadenesheli, Burigede liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. In *ICLR 2021*, 2021.

[107] Finn Lindgren, Håvard Rue, and Johan Lindström. An explicit link between Gaussian fields and Gaussian Markov random fields: The stochastic partial differential equation approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(4):423–498, 2011.

[108] Edward N Lorenz. Predictability: A problem partly solved. In *Proceedings of the Seminar on Predictability*, volume 1, 1996.

[109] Alfred J Lotka. Contribution to the theory of periodic reactions. *The Journal of Physical Chemistry*, 14(3):271–274, 1910.

[110] Alfred J Lotka. The growth of mixed populations: two species competing for a common food supply. In *The Golden Age of Theoretical Ecology: 1923–1940*, pages 274–286. Springer, 1978.

[111] Alfred James Lotka. *Elements of physical biology.* Williams & Wilkins, 1925.

[112] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.

[113] Jan-Matthis Lueckmann, Jan Boelts, David Greenberg, Pedro Goncalves, and Jakob Macke. Benchmarking simulation-based inference. In Arindam Banerjee and Kenji Fukumizu, editors, *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pages 343–351. PMLR, 13–15 Apr 2021.

[114] Emilia Magnani, Nicholas Krämer, Runa Eschenhagen, Lorenzo Rosasco, and Philipp Hennig. Approximate Bayesian neural operators: Uncertainty quantification for parametric PDEs. *arXiv preprint arXiv:2208.01565*, 2022.

[115] VB Mandelzweig and F Tabakin. Quasilinearization approach to nonlinear problems in physics with application to nonlinear ODEs. *Computer Physics Communications*, 141(2):268–281, 2001.

[116] W Robert Mann. Mean value methods in iteration. *Proceedings of the American Mathematical Society*, 4(3):506–510, 1953.

[117] F Mazzia. Test set for boundary value problem solvers, release 0.5, 2014.

[118] Pablo Moreno-Muñoz, Antonio Artés, and Mauricio Alvarez. Heterogeneous multi-output Gaussian process prediction. *Advances in Neural Information Processing systems*, 31, 2018.

[119] Syed Murtuza and SF Chorian. Node decoupled extended Kalman filter based learning algorithm for neural networks. In *Proceedings of 1994 9th IEEE International Symposium on Intelligent Control*, pages 364–369. IEEE, 1994.

[120] Jinichi Nagumo, Suguru Arimoto, and Shuji Yoshizawa. An active pulse transmission line simulating nerve axon. *Proceedings of the IRE*, 50(10): 2061–2070, 1962.

[121] Radford M Neal and Geoffrey E Hinton. A view of the EM algorithm that justifies incremental, sparse, and other variants. In *Learning in Graphical Models*, pages 355–368. Springer, 1998.

[122] Jorge Nocedal and Stephen J Wright. *Numerical Optimization*. Springer, 1999.

[123] Arnold Nordsieck. On numerical integration of ordinary differential equations. *Mathematics of Computation*, 16(77):22–49, 1962.

[124] Magnus Nørgaard, Niels K Poulsen, and Ole Ravn. New developments in state estimation for nonlinear systems. *Automatica*, 36(11):1627–1638, 2000.

[125] Jonathan Oesterle, Nicholas Krämer, Philipp Hennig, and Philipp Berens. Probabilistic solvers enable a straight-forward exploration of numerical uncertainty in neuroscience models. *Journal of Computational Neuroscience*, 50(4):485–503, 2022.

[126] Anthony O'Hagan. Bayes–Hermite quadrature. *Journal of statistical planning and inference*, 29(3):245–260, 1991.

[127] Bernt Øksendal. Stochastic differential equations. In *Stochastic Differential Equations*, pages 65–84. Springer, 2003.

[128] Houman Owhadi. Bayesian numerical homogenization. *Multiscale Modeling & Simulation*, 13(3):812–828, 2015.

[129] Houman Owhadi. Multigrid with rough coefficients and multiresolution operator decomposition from hierarchical information games. *SIAM Review*, 59(1):99–149, 2017.

[130] Kaare Brandt Petersen, Michael Syskind Pedersen, et al. The Matrix Cookbook. *Technical University of Denmark*, 7(15):510, 2008.

[131] James L Phillips. The triangular decomposition of Hankel matrices. *Mathematics of Computation*, 25(115):559–602, 1971.

[132] Jakub Prüher and Miroslav Šimandl. Bayesian quadrature in nonlinear filtering. In *2015 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, volume 1, pages 380–387. IEEE, 2015.

[133] GD Pusch. Jet space as the geometric arena of automatic differentiation. *Computational Differentiation: Techniques, Applications, and Tools, M. Berz, CH Bischof, GF Corliss, and A. Griewank, eds., SIAM, Philadelphia, Penn*, pages 53–62, 1996.

[134] Joaquin Quinonero-Candela and Carl Edward Rasmussen. A unifying view of sparse approximate Gaussian process regression. *The Journal of Machine Learning Research*, 6:1939–1959, 2005.

[135] Christopher Rackauckas and Qing Nie. DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software*, 5(1), 2017.

[136] Christopher Rackauckas and Qing Nie. Confederated modular differential equation APIs for accelerated algorithm development and benchmarking. *Advances in Engineering Software*, 132:1–6, 2019.

[137] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Machine learning of linear differential equations using Gaussian processes. *Journal of Computational Physics*, 348:683–693, 2017.

[138] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Numerical Gaussian processes for time-dependent and nonlinear partial differential equations. *SIAM Journal on Scientific Computing*, 40(1):A172–A198, 2018.

[139] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[140] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006.

[141] Gian-Carlo Rota. The number of partitions of a set. *The American Mathematical Monthly*, 71(5):498–504, 1964.

[142] Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations. *Journal of Mathematical Imaging and Vision*, 62(3): 352–364, 2020.

[143] Simo Särkkä. *Bayesian Filtering and Smoothing*. Cambridge University Press, 2013.

[144] Simo Särkkä and Arno Solin. *Applied Stochastic Differential Equations*. Cambridge University Press, 2019.

[145] Simo Särkkä and Lennart Svensson. Levenberg–Marquardt and line-search extended Kalman smoothers. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5875–5879. IEEE, 2020.

[146] Robert Schaback. Error estimates and condition numbers for radial basis function interpolation. *Advances in Computational Mathematics*, 3(3):251–264, 1995.

[147] Robert Schaback. Convergence of unsymmetric kernel-based meshless collocation methods. *SIAM Journal on Numerical Analysis*, 45(1):333–351, 2007.

[148] E Schäfer. A new approach to explain the "high irradiance responses" of photomorphogenesis on the basis of phytochrome. *Journal of Mathematical Biology*, 2(1):41–56, 1975.

[149] William E Schiesser. *The Numerical Method of Lines: Integration of Partial Differential Equations*. Elsevier, 2012.

[150] Jonathan Schmidt, Nicholas Krämer, and Philipp Hennig. A probabilistic state space model for joint inference from differential equations and data. *Advances in Neural Information Processing Systems*, 34, 2021.

[151] Michael Schober, David K Duvenaud, and Philipp Hennig. Probabilistic ODE solvers with Runge–Kutta means. *NeurIPS 2014*, 2014.

[152] Michael Schober, Simo Särkkä, and Philipp Hennig. A probabilistic model for the numerical solution of initial value problems. *Statistics and Computing*, 29 (1):99–122, 2019.

[153] Matthias Seeger. Low rank updates for the Cholesky decomposition. Technical report, 2004.

[154] Lawrence F Shampine and Mark W Reichelt. The Matlab ODE suite. *SIAM Journal on Scientific Computing*, 18(1):1–22, 1997.

[155] C Shu, H Ding, and KS Yeo. Local radial basis function-based differential quadrature method and its application to solve two-dimensional incompressible Navier–Stokes equations. *Computer methods in applied mechanics and engineering*, 192(7-8):941–954, 2003.

[156] Robert H Shumway and David S Stoffer. An approach to time series smoothing and forecasting using the EM algorithm. *Journal of Time Series Analysis*, 3(4): 253–264, 1982.

[157] Gabe Sibley, Gaurav S Sukhatme, and Larry H Matthies. The iterated sigma point Kalman filter with applications to long range stereo. In *Robotics: Science and Systems*, volume 8, pages 235–244. Citeseer, 2006.

[158] Neil JA Sloane et al. The on-line encyclopedia of integer sequences. *Published electronically at https://oeis. org*, 2018.

[159] Arno Solin. *Stochastic Differential Equation Methods for Spatio-Temporal Gaussian Process Regression*. PhD thesis, Aalto University, 2016.

[160] Robert F Stengel. *Optimal Control and Estimation*. Courier Corporation, 1994.

[161] John C Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. SIAM, 2004.

[162] AI Tolstykh and DA Shirobokov. On using radial basis functions in a "finite difference mode" with applications to elasticity problems. *Computational Mechanics*, 33(1):68–79, 2003.

[163] Filip Tronarp, Hans Kersting, Simo Särkkä, and Philipp Hennig. Probabilistic solutions to ordinary differential equations as nonlinear Bayesian filtering: a new perspective. *Statistics and Computing*, 29(6):1297–1315, 2019.

[164] Filip Tronarp, Simo Särkkä, and Philipp Hennig. Bayesian ODE solvers: The maximum a posteriori estimate. *Statistics and Computing*, 31(3):1–18, 2021.

[165] Filip Tronarp, Nathanael Bosch, and Philipp Hennig. Fenrir: Physics-enhanced regression for initial value problems. In *International Conference on Machine Learning*, pages 21776–21794. PMLR, 2022.

[166] Evgenij E Tyrtyshnikov. How bad are Hankel matrices? *Numerische Mathematik*, 67(2):261–269, 1994.

[167] Rudolph Van Der Merwe. *Sigma-point Kalman filters for probabilistic inference in dynamic state-space models*. Oregon Health & Science University, 2004.

[168] Rudolph Van Der Merwe and Eric A Wan. The square-root unscented Kalman filter for state and parameter-estimation. In *2001 IEEE international conference on acoustics, speech, and signal processing. Proceedings (Cat. No. 01CH37221)*, volume 6, pages 3461–3464. IEEE, 2001.

[169] Balthasar Van der Pol. Theory of the amplitude of free and forced triode vibrations. *Radio Review*, 1:701–710, 1920.

[170] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods*, 17(3):261–272, 2020.

[171] Vito Volterra. *Variazioni e fluttuazioni del numero d'individui in specie animali conviventi*. Società anonima tipografica" Leonardo da Vinci", 1926.

[172] Sebastian Walter. Structured higher-order algorithmic differentiation in the forward and reverse mode with application in optimum experimental design. 2012.

[173] Eric A Wan and Rudolph Van Der Merwe. The unscented Kalman filter for nonlinear estimation. In *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No. 00EX373)*, pages 153–158. Ieee, 2000.

[174] Eric A Wan and Rudolph Van Der Merwe. The unscented Kalman filter. *Kalman filtering and neural networks*, pages 221–280, 2001.

[175] Junyang Wang, Jon Cockayne, Oksana Chkrebtii, Timothy John Sullivan, and Chris J. Oates. Bayesian numerical methods for nonlinear partial differential equations. *Statistics and Computing*, 31, 07 2021.

[176] Gerhard Wanner and Ernst Hairer. *Solving Ordinary Differential Equations II, Stiff Problems*, volume 375. Springer, 1996.

[177] Holger Wendland. *Scattered Data Approximation*, volume 17. Cambridge University Press, 2004.

[178] Jonathan Wenger, Nicholas Krämer, Marvin Pförtner, Jonathan Schmidt, Nathanael Bosch, Nina Effenberger, Johannes Zenn, Alexandra Gessner, Toni Karvonen, François-Xavier Briol, et al. ProbNum: Probabilistic numerics in Python. *arXiv preprint arXiv:2112.02100*, 2021.

[179] MJ Willis. Proportional-integral-derivative control. *Department of Chemical and Process Engineering University of Newcastle*, 6, 1999.

[180] Andrew Gordon Wilson, Christoph Dann, and Hannes Nickisch. Thoughts on massively scalable Gaussian processes. *arXiv preprint arXiv:1511.01870*, 2015.

[181] Stephen J Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

[182] CF Jeff Wu. On the convergence properties of the EM algorithm. *The Annals of Statistics*, pages 95–103, 1983.

[183] Yuanxin Wu, Dewen Hu, Meiping Wu, and Xiaoping Hu. A numerical-integration perspective on Gaussian filters. *IEEE Transactions on Signal Processing*, 54(8):2910–2921, 2006.

[184] Fatemeh Yaghoobi, Adrien Corenflos, Sakira Hassan, and Simo Särkkä. Parallel iterated extended and sigma-point Kalman smoothers. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2021.

[185] Fatemeh Yaghoobi, Adrien Corenflos, Sakira Hassan, and Simo Särkkä. Parallel square-root statistical linear regression for inference in nonlinear state space models. *arXiv preprint arXiv:2207.00426*, 2022.

[186] Ronghui Zhan and Jianwei Wan. Iterated unscented Kalman filter for passive target tracking. *IEEE Transactions on Aerospace and Electronic Systems*, 43 (3):1155–1163, 2007.