UNIVERSITY OF CAMBRIDGE

Linearly Constrained Gaussian Processes

Neel Maniar

Emmanuel College

Supervised by: Dr Henry Moss

June 2024

Word count: 6580

Submitted in partial fulfillment of the requirements for the Master of Philosophy in Data Intensive Science

Declaration

I, author of Emmanuel College, being a candidate for the Master of Philosophy in Data Intensive Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: Neel Maniar

Date: 30th June 2024

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Henry Moss, for his guidance and support throughout this dissertation - in particular for helping me see the general ideas when I got lost in the mathematical weeds of Gaussian Processes.

Abstract

In this report, we reproduce the mathematical and computational results of [**Jidling et al.**, 2017]. The paper exposes a systematic way in which to enforce linear functional constraints on Gaussian processes by choosing a suitable custom covariance function. This allows for an accurate, elegant and computationally cheap method of including physical priors into Bayesian inference. This idea was extended mathematically by looking at other linear constraints, and extended computationally by looking at different optimisation algorithms, using different performance metrics, and tracking performance for noisy data.

Contents

Nomenclature and Notation

0.1 Notation

Some of the notation in this report differs from that in the primary paper as below.

Bold symbols Bold symbols denote objects in any space with dimension greater than 1 (vectors and matrices).

Functionals and operators A distinction is initially made between the vector quantity $\mathbf{f}(\mathbf{x})$ and the function $\mathbf{f}$, (where the former is the latter evaluated at a single point, $\mathbf{x}$). This necessitates the different definitions of "functional" and "operator". However, due to Remark 2.2.3, this distinction is not necessary. Still, an attempt is made to be consistent.

Dimensions To avoid repeated mention of the "dimension of the domain" or "dimension of the codomain", most functions are fixed to be from $\mathbb{R}^D$ to $\mathbb{R}^K$, and most functionals have codomain $\mathbb{R}^L$. Dimensions are indexed by their lower case, d, k and l.

In addition, there are N_T training points, N_P test points, indexed by n, or n_T and n_P if the former is ambiguous.

Indexing tensors as matrices Dealing with vector Gaussian processes often involves objects with more than two indices packaged as a matrix with two indices. To this effect, a new notation is used to index them:

$$K_{(n,k),k'} := K_{n+kK,k'}$$

This is analogous to the `reshape` method in `numpy`.

Summation convention Any pair of indices is implicitly summed over, as in Einstein summation convention.

0.2 Abbreviations and Symbols

Symbol	Description	Page
$\overline{\mathbf{z}}$	Flattened vector (see definition of Vector Gaussian Process)	3
GP	Gaussian Process	3
K_{SE}, SE	Squared exponential kernel (see Definition 2.1.2)	3
$\mathbf{I}_N$	Identity matrix in N dimensions	7
$\overline{\overline{\mathbf{K}}}$	Gram matrix	8
MLL	Marginal Log-Likelihood (logarithm of Bayesian evidence)	11
GPR	Gaussian Process Regression	13
RMSE	Root Mean Squared Error	14
$\\|\mathbf{z}\\|_p$	pth norm of $\mathbf{z}$: $\left(\sum_{d=1}^{D} \|z_d\|^p\right)^{1/p}$	14
NLPD	Negative Log Predictive Density	15

Table 1: Glossary of Symbols and Abbreviations

List of Figures

List of Tables

Chapter 1

Introduction

Physical modelling is a ubiquitous task in many fields of science, including climate forecasts, civil engineering and ecology. Many of these processes are well-modelled by explicit differential or integral equations. In particular, many of these constraints are linear, for example:

1. Incompressible fluid velocities form a divergence-free vector field.
2. The concentration of metals and rocks (as in the Swiss Jura dataset [**Swan**, 1998]) follows a diffusion equation.
3. Systems of predators and prey are well-modelled by the Lotka-Volterra system.
4. In an unchanging electric field with zero current density, magnetic fields are curl-free.

In an era of Data Intensive Science, there is a new paradigm in physical modelling. Computational and algorithmic advances in the last two decades have allowed for the development of machine learning (ML) methods, which learn target functions by optimising a data-dependent loss. These are flexible models and have been shown to perform well under a variety of different tasks using different architectures.

The best architectures are able to effectively encode inductive biases. For examples, CNNs exploit spatial symmetry for effective image analysis [**Bronstein et al.**, 2021], and the self-attention layer in transformer networks is based on our understanding of natural languages. The natural next step is to use physics knowledge to aid the predictive power of ML models, especially where data is scarce and noisy.

One attempt to encode physical information can be found in PINNs (Physics Informed Neural Networks) [**Raissi, Perdikaris, and Karniadakis**, 2017]. Here, the loss function is augmented to penalise deviation from a physical constraint. However, this does not explicitly treat data as being a noisy approximation to a physical latent function, and can lead to a balance between minimising error due to noise and error due to deviation from physics. This balance is difficult to interpret.

In this paper, we consider Gaussian processes (GPs), a flexible Bayesian non-parametric framework. The advantage over traditional NN architectures is that they are interpretable, allow for very explicit prior encoding, and provide natural uncertainty bounds on predictions. They operate by introducing a prior over the space of functions and updating the distribution using the available data. Incredibly, this update can be made on a continuous spectrum with just finite data and computing time.

One way in which physical constraints can be encoded within Gaussian processes is by introducing artificial *inducing* or *collocation* points, at which the constraint is locally enforced. These methods are flexible, allowing for forcing terms. In [**Long et al.**, 2022], this method is even used to include information about partially known constraints. The disadvantage with this method is that it increases the size of the training set and does not enforce the constraint between inducing points.

In this report, the results obtained by [**Jidling et al.**, 2017] are reproduced. The authors discuss a new way of encoding arbitrary linear constraints into multivariate GPs. In particular, the method transforms an existing GP with any required conditions (such as smoothness) into one which satisfies a linear constraint, relying on the fact that GPs, much like Gaussian distributions, are closed under linear operators.

The dissertation is structured by initially covering the mathematical background of the project, including how multivariate GPs are defined, and different methods by which linear constraints may be enforced on them. The reproductions of the 2D and 3D studies are discussed in Chapter 3, along with details of the implementation in GPJax [**Pinder and Dodd**, 2022]. In Chapter 4, we extend the primary paper to test robustness to noise and test the method on a new linear constraint. The key takeaways are summarised in Chapter 5.

Chapter 2

Mathematical background

2.1 Vector Gaussian Processes

Most introductory treatments of Gaussian processes, including [Rasmussen and Williams, 2006], solely discuss scalar-valued Gaussian processes - that is to say, functions $f : \mathbb{R}^D \to \mathbb{R}$.

Here, we are dealing with vector-valued processes, so it is worth outlining what that means. A more comprehensive review is found in [Alvarez, Rosasco, and Lawrence, 2012].

Definition 2.1.1 (Vector Gaussian Process)**.** We say $\mathbf{f} : \mathbb{R}^D \to \mathbb{R}^K$ follows a Gaussian process distribution with mean $\mathbf{m}$ and covariance function $\mathbf{K}$:

$$\mathbf{m} : \mathbb{R}^D \to \mathbb{R}^K, \quad \mathbf{K} : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}^{K \times K}$$

if on any finite subset $\{\mathbf{x}_n\}_{n=1}^N \subset \mathbb{R}^D$, the flattened vector:

$$\bar{\mathbf{f}} := \begin{bmatrix} \{f_1(\mathbf{x}_n)\}_{n=1}^N \\ \vdots \\ \{f_K(\mathbf{x}_n)\}_{n=1}^N \end{bmatrix} \in \mathbb{R}^{NK}$$

is a multivariate normal vector with mean $\bar{\mathbf{m}}$ (similarly defined) and covariance matrix given by $\mathrm{Cov}[f_i(\mathbf{x}_l), f_j(\mathbf{x}_m)] = K_{ij}(\mathbf{x}_l, \mathbf{x}_m)$. The Gaussian process is denoted:

$$\mathbf{f} \sim \mathcal{GP}(\mathbf{m}, \mathbf{K})$$

This definition of a vector Gaussian process naturally demonstrates that they may be thought of equivalently as more familiar scalar Gaussian processes with an augmented mean and kernel function.

Definition 2.1.2 (Squared exponential kernel)**.** This is the kernel used at all times in this paper (but the results are generalisable to any mean-differentiable kernel).

$$K_{SE}(\mathbf{x}, \mathbf{x}'; \sigma_f, \ell) = K_{SE}(\mathbf{x} - \mathbf{x}'; \sigma_f, \ell) = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}\|\mathbf{x} - \mathbf{x}'\|_2^2\right)$$

2.2 Linear constraints

Definition 2.2.1 (Functional). The definition of functional changes depending on the context, but in this paper, it is defined to be an object mapping a function to vector, like so:

$$\mathscr{F}_{\mathbf{x}} : (\mathbb{R}^D \to \mathbb{R}^K) \to \mathbb{R}^L$$

Definition 2.2.2 (Operator). Similarly, the definition of operator is not fixed in general. In this paper, it is defined to be any function where both the domain and codomain is a function space, like so:

$$\mathscr{F} : (\mathbb{R}^D \to \mathbb{R}^K) \to (\mathbb{R}^D \to \mathbb{R}^L)$$

We will typically require that the domain of both function spaces is the same to allow the natural relation between functionals and operators.

Remark 2.2.3 (Correspondence between functionals and operators). A functional $\mathscr{F}_{\mathbf{x}}$ naturally defines an operator:

$$\begin{aligned} \mathscr{F}\mathbf{f} &: \mathbb{R}^D \to \mathbb{R}^L \\ \mathscr{F}\mathbf{f}(\mathbf{x}) &\mapsto \mathscr{F}_{\mathbf{x}}\mathbf{f} \end{aligned}$$

Similarly, an operator $\mathscr{F}$ naturally defines a functional:

$$\begin{aligned} \mathscr{F}_{\mathbf{x}} &: (\mathbb{R}^D \to \mathbb{R}^K) \to \mathbb{R}^L \\ \mathscr{F}_{\mathbf{x}}(\mathbf{f}) &\mapsto (\mathscr{F}\mathbf{f})(\mathbf{x}) \end{aligned}$$

Therefore, the distinction between operator and functional is not too important. Nonetheless, we will attempt to be precise.

Definition 2.2.4 (Linear Functional). A linear functional is a function $\mathscr{F}_{\mathbf{x}} : (\mathbb{R}^D \to \mathbb{R}^K) \to \mathbb{R}^L$, such that for every $\mathbf{x} \in \mathbb{R}^D$ and $\lambda \in \mathbb{R}$:

1. $\mathscr{F}_{\mathbf{x}}(\lambda\mathbf{f}) = \lambda\mathscr{F}_{\mathbf{x}}(\mathbf{f})$
2. $\mathscr{F}_{\mathbf{x}}(\mathbf{f} + \mathbf{g}) = \mathscr{F}_{\mathbf{x}}(\mathbf{f}) + \mathscr{F}_{\mathbf{x}}(\mathbf{g})$

Remark 2.2.5 (Linear operators as matrices). A linear operator may be defined as a $L \times K$ matrix, where each element is a scalar operator:

$$(\mathscr{F}f)_l = (\mathscr{F})_{lk} f_k$$

Definition 2.2.6 (Linear constraint). A linear constraint on a function $\mathbf{f}$ is given by

$$\forall \mathbf{x} \in \mathbb{R}^D, \quad \mathscr{F}_{\mathbf{x}}\mathbf{f} = \mathbf{0}$$

where $\mathscr{F}_{\mathbf{x}}$ is a linear functional.

Example 2.2.7 (Examples of linear constraints). The following examples illustrate the rich variety of problems which follow linear constraints.

1. Divergence-free vector field (for example, in the study of incompressible fluid flows):
$$\mathfrak{F}_\mathbf{x}\mathbf{f} := \boldsymbol{\nabla} \cdot \mathbf{f}|_\mathbf{x} = 0.$$
2. Curl-free vector field (for example, in magnetic fields):
$$\mathfrak{F}_\mathbf{x}\mathbf{f} := \boldsymbol{\nabla} \times \mathbf{f}|_\mathbf{x} = \mathbf{0}.$$
3. Constant integral (for example, for momentum or mass conservation):
$$\mathfrak{F}_\mathbf{x} f := \int_{x_1}^{x_2} f(z)\mathrm{d}z.$$
4. Evaluation at a point:
$$\mathfrak{F}_\mathbf{x}\mathbf{f} := \mathbf{f}(\mathbf{u}(\mathbf{x})) = 0$$
for some (possibly nonlinear) function $\mathbf{u} : \mathbb{R}^D \to \mathbb{R}^D$.
5. Many deep physical phenomena are described by linear PDEs. For example, the theory of gravitational waves in a vacuum involves solutions to the linearised Einstein equations:
$$\mathfrak{F}_\mathbf{x} h_{\mu\nu} := \frac{1}{2}(\partial_\sigma\partial_\mu h^\sigma_\nu + \partial_\sigma\partial_\nu h^\sigma_\mu - \partial_\mu\partial_\nu h - \Box h_{\mu\nu} - \eta_{\mu\nu}\partial_\rho\partial_\lambda h^{\rho\lambda} + \eta_{\mu\nu}\Box h) = 0$$
where $h := \eta^{\mu\nu}h_{\mu\nu}$ and $\Box := \eta^{\mu\nu}\partial_\mu\partial_\nu$.
6. The Black-Scholes equation used for financial option pricing:
$$\mathfrak{F}_\mathbf{x} V := \frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$
where $\mathbf{x} := \begin{bmatrix} S \\ t \end{bmatrix}$.

Example 2.2.8 (Non-examples of linear constraints)**.** The following non-examples show a few common nonlinear constraints, demonstrating the limitations of this method.

1. Simple pendulum:
$$\mathfrak{F}_t\theta := \left.\frac{\partial^2\theta}{\partial t^2}\right|_t + \frac{g}{L}\sin(\theta(t)) = 0.$$
2. The Fisher-KPP equation for reaction-diffusion systems:
$$\frac{\partial u}{\partial t} - D\frac{\partial^2 u}{\partial x^2} = ru(1-u).$$

2.2.1 Linear operators acting on Gaussian processes

Theorem 2.2.9. *Given a linear operator $\mathfrak{F}$ and a Gaussian process $\mathbf{f} \sim \mathcal{GP}(\mathbf{m}, \mathbf{K})$,*
$$\mathfrak{F}\mathbf{f} \sim \mathcal{GP}(\mathfrak{F}_\mathbf{x}\mathbf{m}, \mathfrak{F}_\mathbf{x}\mathbf{K}\mathfrak{F}_{\mathbf{x}'}^\top)$$
where $(\mathfrak{F}_\mathbf{x}\mathbf{K}\mathfrak{F}_{\mathbf{x}'}^\top)_{ij} := (\mathfrak{F}_\mathbf{x})_{ik}(\mathfrak{F}_{\mathbf{x}'})_{jl}K_{kl}$[1]

[1]Note that all of the $\mathfrak{F}$s here are operators rather than functionals. The notation is abused to make it clear which argument each operator acts on.

Proof. By definition, $\mathbf{f}$ is a Gaussian process if it is a joint Gaussian on any finite subset of its inputs.

The proof is given below for $f : \mathbb{R}^D \to \mathbb{R}$, but can be extended to higher output dimensions. $f \sim \mathcal{GP}(m, K)$ is a Gaussian process if for any finite subset $\{\mathbf{x}_n\}_{n=1}^{N}$,

$$\begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(\mathbf{x}_1) \\ \vdots \\ m(\mathbf{x}_n) \end{bmatrix}, \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & \dots & K(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & & \vdots \\ K(\mathbf{x}_n, \mathbf{x}_1) & \dots & K(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix} \right)$$

Since $\mathfrak{F}$ is a linear operator, by Remark 2.2.5, it can be represented as a matrix of scalar operators. It is well-known that Gaussians are preserved under linear operations like so:

$$\mathfrak{F} \begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix} \sim \mathcal{N}\left(\mathfrak{F} \begin{bmatrix} m(\mathbf{x}_1) \\ \vdots \\ m(\mathbf{x}_n) \end{bmatrix}, \mathfrak{F} \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & \dots & K(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & & \vdots \\ K(\mathbf{x}_n, \mathbf{x}_1) & \dots & K(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix} \mathfrak{F}^\top \right)$$

where $\mathfrak{F}^\top$ acts on the second argument of K. Since this holds for any finite subset of $\mathbb{R}^D$, it follows that $\mathfrak{F}f \sim \mathcal{GP}(\mathfrak{F}m, \mathfrak{F}K\mathfrak{F}^\top)$. □

2.3 Vector Gaussian Process Regression

We will cover Vector Gaussian Process Regression here briefly, in order to set up the following sections. A complete treatment may be found in [**Alvarez, Rosasco, and Lawrence**, 2012].

This is a Bayesian framework of inference over function space. Therefore, we require a prior distribution over the functions and a likelihood distribution of the data given a particular function, viz.

$$p(\mathbf{f}|\{\mathbf{x}, \mathbf{y}\}) = \frac{p(\{\mathbf{y}\}|\mathbf{f}, \{\mathbf{x}\})p(\mathbf{f}|\{\mathbf{x}\})}{p(\{\mathbf{y}\}|\{\mathbf{x}\})}$$

where $\{\mathbf{z}\}$ is a shorthand for the training data, $\{\mathbf{z}_i\}_{i=1}^{N_T}$ and $\mathbf{f}$ is the function distribution. Each of the terms are described in greater detail below.

2.3.1 Prior

The prior distribution is a Gaussian process with mean and kernel function explicitly chosen. The mean function is typically chosen to be zero to simplify calculations. In the experiments that follow, the mean function is also set to zero.

Explicitly:

$$p(\mathbf{f}|\{\mathbf{x}\}) = \mathcal{GP}(\mathbf{0}, \mathbf{K})$$

2.3.2 Likelihood

The likelihood is assumed to be a Gaussian, centred at the true function value with a heteroscedastic variance of σ_n^2, which may be thought of as measurement error. This is a parameter which gets optimised along with any kernel parameters.

Following the convention in the definition of Vector Gaussian Processes, $\{\mathbf{y}\}$ can be flattened into a single vector, $\overline{\mathbf{y}}$. A similar convention is followed for $\overline{\mathbf{f}(\mathbf{x})}$.

$$p(\{\mathbf{y}\}|\mathbf{f},\{\mathbf{x}\}) = p\left(\overline{\mathbf{y}}|\overline{\mathbf{f}(\mathbf{x})}\right) = \mathcal{N}(\overline{\mathbf{f}(\mathbf{x})}, \sigma_n^2 \mathbf{I}_{NK})$$

2.3.3 Posterior

The posterior distribution is a Gaussian process, since a Gaussian process is a conjugate prior for a Gaussian likelihood, analogously to Bayesian inference for finite dimensional Gaussians. Explicitly writing out the posterior requires some tricky notation[2], which we introduce first.

Notation

$$\overline{\mathbf{K}} : \mathbb{R}^D \to \mathbb{R}^{N_T K \times K}$$
$$\overline{K}_{(n,k),k'}(\mathbf{z}) \mapsto K_{k,k'}(\mathbf{x}_n, \mathbf{z})$$

The following is also known as a Gram matrix:

$$\overline{\overline{\mathbf{K}}} \in \mathbb{R}^{N_T K \times N_T K}$$
$$\overline{\overline{K}}_{(n,k),(n',k')} = K_{k,k'}(\mathbf{x}_n, \mathbf{x}_{n'})$$

Explicit posterior distribution

$$p(\mathbf{f}|\{\mathbf{x},\mathbf{y}\}) = \mathcal{GP}(\hat{\mathbf{m}}, \hat{\mathbf{K}})$$

where

$$\hat{\mathbf{m}} : \mathbb{R}^D \to \mathbb{R}^K$$
$$\hat{\mathbf{m}}(\mathbf{z}) \mapsto \overline{\mathbf{K}}(\mathbf{z})^\top \left[\overline{\overline{\mathbf{K}}} + \sigma_n^2 \mathbf{I}_{NK}\right]^{-1} \overline{\mathbf{y}}$$

and

$$\hat{\mathbf{K}} : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}^{K \times K}$$
$$\hat{\mathbf{K}}(\mathbf{z},\mathbf{z}') \mapsto \mathbf{K}(\mathbf{z},\mathbf{z}') - \overline{\mathbf{K}}(\mathbf{z})^\top \left[\overline{\overline{\mathbf{K}}} + \sigma_n^2 \mathbf{I}_{NK}\right]^{-1} \overline{\mathbf{K}}(\mathbf{z}')$$

Prediction

This framework gives us a probability distribution for the predicted function value at any test point $\mathbf{x}_*$ - it is predicted to be a normal distribution with mean $\hat{\mathbf{m}}(\mathbf{x}_*)$ and variance $\hat{\mathbf{K}}(\mathbf{x}_*, \mathbf{x}_*)$.

Indeed this may be extended to give the mean and covariance matrix of any test set, $\{\mathbf{x}_*\}$ - the predictive distribution is:

$$\mathbf{f}(\{\mathbf{x}_*\}) \sim \mathcal{N}\left(\hat{\mathbf{m}}\left(\{\mathbf{x}_*\}\right), \hat{\mathbf{K}}(\mathbf{x}_{*i}, \mathbf{x}_{*j})\right)$$

[2]See Nomenclature for details on matrix indexing.

2.3.4 Evidence

As this is a conjugate prior, the evidence can be analytically determined. Explicitly:

$$p(\{\mathbf{y}\}|\{\mathbf{x}\}) = p(\overline{\mathbf{y}}|\{\mathbf{x}\}) = \mathcal{N}(\mathbf{0}, \overline{\overline{\mathbf{K}}} + \sigma^2 I_{NK})$$

This is used to optimise any hyperparameters, as in Section 3.1.4.

2.4 Artificial Observations

One way to embed linear constraints into Gaussian processes is by enforcing the constraint at a finite number of points through *artificial observations*. The dataset is extended by N_c points, $\{\tilde{\mathbf{x}}_n, \tilde{\mathbf{y}}_n\}_{n=1}^{N_c}$, where $\tilde{\mathbf{y}} = \mathscr{F}_{\tilde{\mathbf{x}}}\mathbf{f} = \mathbf{0}$. In this report, we will refer to the union of the regular dataset and the artificial dataset as the *augmented dataset*.

By Theorem 2.2.9, we know how GPs are preserved under linear functionals. Therefore, applying the result to the following linear functional: $\begin{bmatrix}\mathbf{I}_{NK} \\ \mathscr{F}\end{bmatrix}$, where $\mathbf{I}_{NK}$ is the identity functional, we have:

$$\begin{bmatrix}\mathbf{f} \\ \mathscr{F}\mathbf{f}\end{bmatrix} \sim \mathcal{GP}\left(\begin{bmatrix}\mathbf{m} \\ \mathscr{F}\mathbf{m}\end{bmatrix}, \begin{bmatrix}\mathbf{K} & \mathbf{K}\mathscr{F}^\top \\ \mathscr{F}\mathbf{K} & \mathscr{F}\mathbf{K}\mathscr{F}^\top\end{bmatrix}\right)$$

Thus, this may form the prior for a joint Gaussian process including both the regular and artificial points. This method can accommodate non-linear forcing terms of the form

$$\mathscr{F}_{\tilde{\mathbf{x}}}\mathbf{f} = \mathbf{g}(\tilde{\mathbf{x}})$$

by choosing the augmented dataset $(\tilde{\mathbf{x}}, \mathbf{g}(\tilde{\mathbf{x}}))$.

The drawback of this method is that the constraints are only enforced at the artificial points. In addition, the most expensive part of optimisation and prediction is taking the inverse of the Gram matrix, an operation which scales as $\mathcal{O}(N^3)$ where N is the number of training points, so artificially increasing the number of training points reduces performance greatly.

In this paper, the $\tilde{\mathbf{x}}$ are chosen to be a subset of the test set. In other words, the results from the artificial kernel are actually *best case* scenarios, since the linear constraint is enforced exactly at points where the model performance is measured.

2.5 Custom Kernel

The proposed method in [**Jidling et al.**, 2017] encodes the linear constraint in a very different way. Here, the kernel function is carefully chosen so that any realisation of the Gaussian process definitely satisfies the linear constraint everywhere. These have been known for the commonly used linear constraints in this paper [**Wahlström**, Dec. 2015]. This paper extends this idea and introduces a systematic method to derive such a kernel function for arbitrary sets of linear constraints. The method is outlined below.

The main idea is that if there exists a linear functional $\mathscr{G}_{\mathbf{x}} : (\mathbb{R}^D \to \mathbb{R}^M) \to \mathbb{R}^K$, such that for every $\mathbf{g} : \mathbb{R}^D \to \mathbb{R}^M$ and every $\mathbf{x} \in \mathbb{R}^D$, $(\mathscr{F}\mathscr{G})_{\mathbf{x}}\mathbf{g} = \mathbf{0}$, then for any Gaussian process $\mathbf{g} \sim \mathcal{GP}(\mu_{\mathbf{g}}, K_{\mathbf{g}})$, the Gaussian process $\mathscr{G}\mathbf{g}$ will satisfy the constraint everywhere.

Theorem 2.2.9 specifies this GP completely. So we are looking for some linear operator $\mathscr{G}$ such that:

$$\underset{L\times K}{\mathscr{F}}\underset{K\times M}{\mathscr{G}} = \mathbf{0} \tag{2.1}$$

This approach is very powerful, since it allows for a rich, explicit and interpretable specification of predicted functions. The GP $\mathbf{g}$ can be chosen with a kernel $K_{\mathbf{g}}$ to enforce desired properties such as smoothness and periodicity, for example by using the squared exponential kernel or by using the periodic kernel. Then the linear operator $\mathscr{G}$ enforces the required constraint.

2.5.1 Finding $\mathscr{G}$

The main aim of the paper is to provide a systematic way to find an operator $\mathscr{G}$ given a linear constraint. We outline this method below.

The columns[3] of $\mathscr{G}$, denoted $\mathscr{g} : (\mathbb{R}^D \to \mathbb{R}) \to \mathbb{R}^K$ must satisfy

$$\mathscr{F}\mathscr{g} = \mathbf{0} \tag{2.2}$$

The biggest assumption of this method is that both $\mathscr{F}$ and $\mathscr{G}$ are in the span of a finite basis of linear operators, denoted $\boldsymbol{\xi}$. For example, $\boldsymbol{\xi} = \{\frac{\partial}{\partial x_p}\}_{p=1}^{P}$. Thus we may write:

$$\begin{aligned}\mathscr{F}_{lk} &= \Phi_{lkp}\xi_p \\ \mathscr{g}_k &= \Gamma_{kp}\xi_p\end{aligned}$$

where $\mathbf{\Phi} \in \mathbb{R}^{L\times K\times P}$ and $\mathbf{\Gamma} \in \mathbb{R}^{K\times P}$ are coefficient matrices. $\mathbf{\Phi}$ is known, and $\mathbf{\Gamma}$ needs to be found.

Then, by eq. (2.2):

$$\begin{aligned}\mathscr{F}_{lk}\mathscr{g}_k &= \Phi_{lkp}\xi_p\Gamma_{kp'}\xi_{p'} \\ &= \boldsymbol{\xi}^\top\mathbf{\Phi}_l\mathbf{\Gamma}\boldsymbol{\xi} \\ &= 0\end{aligned}$$

where $(\mathbf{\Phi}_l)_{pk} := \Phi_{lkp}$. By Theorem A.0.1, this implies that

$$\mathbf{\Phi}_l\mathbf{\Gamma} + \mathbf{\Gamma}^\top\mathbf{\Phi}_l^\top = \mathbf{0}$$

This condition allows for every possible $\mathscr{g}$ that can be written in the basis $\boldsymbol{\xi}$ to be found. These can then be concatenated to make $\mathscr{G}$.

This general procedure is summarised in Algorithm 1, but is reproduced in the following sections for the specific linear constraints studied in this paper.

[3]using the matrix definition of linear operators.

Algorithm 1 Finding $\mathfrak{G}$

```
    **Input:** Linear constraint $\mathfrak{F}\mathbf{f} = \mathbf{0}$.
    **Output:** Custom GP that satisfies the constraint.
1: Choose a basis $\boldsymbol{\xi}$ of operators for $\mathfrak{G}$.
2: Write $\mathfrak{F}_{lk} = \Phi_{lkp}\xi_p$ and $\mathfrak{g}_k = \Gamma_{kp}\xi_p$.
3: Solve $\boldsymbol{\Phi}_l\boldsymbol{\Gamma} + \boldsymbol{\Gamma}^\top\boldsymbol{\Phi}_l^\top = \mathbf{0}$.
4: If there are no solutions than choose a larger basis.
5: Let $\mathfrak{G}$ be a concatenation of linearly independent columns $\mathfrak{g}$.
6: Choose a Gaussian process $\mathbf{g} \sim \mathcal{GP}(\mathbf{m_g}, \mathbf{K_g})$ with desirable functional properties.
7: The GP $\mathfrak{G}\mathbf{g} \sim \mathcal{GP}(\mathfrak{G}\mathbf{m_g}, \mathfrak{G}\mathbf{K_g}\mathfrak{G}^\top)$ will satisfy the linear constraint.
```

Chapter 3

Reproduction of paper

3.1 GPJax

3.1.1 GPJax vs MATLAB

Original paper In this reproduction, GPJax is used as the primary library in which to implement the construction, optimisation and prediction of Gaussian processes. This is in contrast to the original paper, which uses MATLAB. Since this is really a paper on the mathematical method, the results should hold regardless of the programming framework used.

That the code used by the original paper is made publicly available allows for a far more detailed reproduction for two main reasons.

- Specific parameters can be tested, and various statistics can be used to quantify how well a reproduction matches the implementation in the paper. This is done in Section 3.2.7
- Any details that are omitted due to the realistic constraints of submitting papers to journals are made explicit. This is done in Sections 3.1.5, 3.2.11 and 3.3.7.

Automatic differentiation GPJax [**Pinder and Dodd**, 2022] is a framework which is built on JAX [**Bradbury et al.**, 2018], which is a library built for accelerator-oriented array computation, designed for high-performance numerical computing and large-scale machine learning[1]. In fact, a key difference between the two implementations is the use of automatic differentiation to optimise the marginal log-likelihood. Whereas the MATLAB code must explicitly encode the derivatives of the Marginal Log-Likelihood (MLL), resulting in extremely long files and error-prone code, code written in JAX does not need to explicitly provide derivatives with respect to variables. This makes the code more flexible, and amenable to new functions and extra variables.

Compute Time The MATLAB implementation is much faster than GPJax. For example, 50 iterations of the 2D script (including all kernels), takes around 6 minutes in GPJax, compared to less than a minute in MATLAB. The 3D script took 1.5 hours to run in GPJax versus 1 hour in MATLAB. These benchmarks are based on performance on a modern laptop with 16 CPU cores.

[1]As described in the JAX documentation.

While profiling, it was found that the most expensive operations in GPJax was computation of the MLL and prediction, especially for the artificial kernel. This is expected because of the increased computational complexity when including inducing points, as discussed in Section 2.4 and Section 3.1.3. It is interesting, however, that these operations occur much faster in MATLAB. The explicit specification of gradients and Hessian matrices also makes the MATLAB code faster.

3.1.2 Issues with GPJax

When reproducing the 3D study in GPJax, an issue was discovered. The artificial kernel was taking an exceedingly long time to fit to data, even without parameter optimisation. The `line-profiler` module was used to determine that the performance bottleneck was the matrix equation solver in the `predict` method. This is no surprise, since this is a general bottleneck for GPs, as displayed in Table 3.2, but the extent of the slowdown was surprising.

The slowdown is due to implementation of matrix equation solving in the linear algebra library used by GPJax, called CoLA. For matrices where the product of dimensions exceeds 10^6, it switches from Cholesky decomposition to the Conjugate Gradient (CG) method. CG is an iterative procedure, so it takes far longer than Cholesky decomposition and does not necessarily achieve the minimum. The advantage of CG is that it is more accurate for ill-conditioned matrices, which is common in large and sparse matrices.

This was tested and demonstrated in a script which compares three different `solve` methods by evaluating $\mathbf{X} = \mathbf{\Sigma}^{-1}\mathbf{K}$ for $\mathbf{\Sigma} \in \mathbb{R}^{d_1 \times d_1}, \mathbf{K} \in \mathbb{R}^{d_1 \times d_2}$. Here d_2 is fixed to be 500.

$\mathbf{\Sigma}$ is created by first producing a matrix $\mathbf{A}$ where each entry is drawn from a standard normal distribution, and then calculating $\mathbf{\Sigma} = \mathbf{A}\mathbf{A}^\top + 0.1 \times \mathbf{I}_{d_1}$ to ensure it is a well-defined positive definite matrix. Each entry from $\mathbf{K}$ is drawn from a standard normal distribution. The time taken for solving and the accuracy of reproduction is measured and recorded in Table 3.1.

	CoLA (Chol + CG)	**JAX.Numpy** (LU)	**CoLA** (Only Cholesky)
$d_1 = 1000$			
$\|\|\mathbf{\Sigma X} - \mathbf{K}\|\|_2$	$\mathbf{4.32 \times 10^{-10}}$	4.27×10^{-10}	$\mathbf{4.32 \times 10^{-10}}$
Time taken (s)	0.684	0.666	**0.113**
$d_1 = 1001$			
$\|\|\mathbf{\Sigma X} - \mathbf{K}\|\|_2$	0.0176	4.33×10^{-10}	$\mathbf{4.39 \times 10^{-10}}$
Time taken (s)	26.4	1.19	**0.352**

Table 3.1: Comparison of CoLA, JAX.Numpy, and CoLA with Cholesky performance for different values of d_1.

For our use, Cholesky decomposition is faster and more accurate, so a child class of `ConjugatePosterior` was created in order to update its `predict` method to use `cola.solve` with Cholesky decomposition enforced for all matrix sizes.

3.1.3 Processing data

Initially each datum is of the form $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^D \times \mathbb{R}^K$.

This is modified to D measurements: $\{((\mathbf{x}, i), y_i)\}_{i=1}^{D}$. This ensures that the output is one dimensional, so the kernel function is scalar-valued. Explicitly, denoting the matrix-valued kernel function as $\mathbf{K}$ and the scalar-valued kernel function as $\tilde{K}$:

$$K_{ij}(\mathbf{x}, \mathbf{x}') = \tilde{K}((\mathbf{x}, i), (\mathbf{x}', j))$$

This fully consistent with the definition of vector Gaussian processes (Definition 2.1.1), and allows for the GPJax framework to handle such systems.

This means that if there are N_T training data, this is effectively $N_T D$ training data in a scalar GP. If, in addition there are N_c artificial inducing points, then there are $D(N_T + N_c)$ data (where N_c is typically much greater than N_T).

Kernel	Computational Complexity
Diagonal Kernel	$\mathcal{O}(D^3 N_T^3)$
Custom Kernel	$\mathcal{O}(D^3 N_T^3)$
Artificial Kernel	$\mathcal{O}(D^3 (N_T + N_c)^3)$

Table 3.2: Computational complexities of different kernels.

3.1.4 Optimisation

In Gaussian process regression (GPR), the hyperparameters are the variables used to describe the prior and likelihood distributions.

As this is a Bayesian framework, the hyperparameters may be optimised with respect to the Bayesian evidence, the logarithm of which is referred to as the MLL. By the section on GPR evidence, the MLL is:

$$\text{MLL} := \log p(\overline{\mathbf{y}}|\{\mathbf{x}\}) = -\frac{1}{2}\overline{\mathbf{y}}^\top \left[\overline{\overline{\mathbf{K}}} + \sigma_n^2 \mathbf{I}_{NK}\right]^{-1} \overline{\mathbf{y}} - \frac{1}{2}\log\det\left(\overline{\overline{\mathbf{K}}}\right) - \frac{n}{2}\log 2\pi \qquad (3.1)$$

In this paper, the MLL is a function of three hyperparameters, σ_f, ℓ and σ_n. The first two are from the kernel and the last parametrises the likelihood.

This is a standard loss function which is minimised; one of the benefits of the GPJax framework above MATLAB is that this function is provided by the library in the form of a `ConjugateMLL` object, and does not need to be explicitly specified.

Importantly, this is not the same as the metric used to judge the performance of a model. These are detailed in Section 3.1.5.

BFGS

The first method of optimisation uses the BFGS algorithm [**Fletcher**, 1987], implemented by default in `scipy.optimize.minimize`. This algorithm may be interpreted as an optimised version of Newton's method, whereby a function $f(\mathbf{x})$ is optimised by the following update rule:

$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - [\boldsymbol{\nabla}^2 f(\mathbf{x}_k)]^{-1} \boldsymbol{\nabla} f(\mathbf{x}_k)$$

It is modified such that once the direction of the step is determined, a line search is performed to determine the size of the step. In addition, the Hessian is not calculated directly every time, but accumulated using the gradient evaluations. The gradient is supplied through autodifferentiation in JAX.

This is the optimisation used in MATLAB via the `fminunc` function. The gradients have to manually computed and passed to the function.

Adam

A popular optimisation algorithm in deep learning is Adam [**Kingma and Ba**, 2017]. It employs a form of stochastic gradient descent strategy. This optimisation is employed with the `optax` package, and can be enabled in the script with the `--adam` flag.

The update step is as follows:

$$\begin{aligned}
\mathbf{m}_{k+1} &\leftarrow \left(\beta \mathbf{m}_k + (1-\beta)\frac{\partial f}{\partial \mathbf{x}_k}\right)\left(1-\beta^{k+1}\right)^{-1} \\
\mathbf{v}_{k+1} &\leftarrow \left(\gamma \mathbf{m}_k + (1-\gamma)\left(\frac{\partial f}{\partial \mathbf{x}_k}\right)^2\right)\left(1-\gamma^{k+1}\right)^{-1} \\
\mathbf{x}_{k+1} &\leftarrow \mathbf{x}_k - \alpha \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \varepsilon}
\end{aligned} \tag{3.2}$$

Comparison of Optimisation algorithms

Whilst Adam is a common optimisation algorithm in machine learning, and does particularly well in high-dimensional data, it is not suitable for this implementation.

One reason is that here, it not applied with stochastic gradient descent (SGD) - namely, the entire dataset is used at every step of the optimisation. This makes it particularly slow, as seen in Section 3.3.6.

It is possible to implement stochastic gradient descent by adapting the MLL to only compute the Gram matrix for a each subset of a random partition of the dataset, and this could be a possible extension.

BFGS tends to be faster for smaller datasets with more convex loss-functions, especially where there is no mini-batching. Indeed this was the case in this project, where the parameter space was three-dimensional and there were less than 10^4 datapoints.

3.1.5 Performance measurement

RMSE

The primary method in which to measure performance of a model is the Root Mean Squared Error (RMSE), as in the primary paper. Explicitly:

$$\text{RMSE} := \sqrt{\frac{1}{N_P D}\sum_{n=1}^{N_P} \|\mathbf{y}_n - \mathbf{f}(\mathbf{x}_n)\|_2^2}$$

This is a minor error in the primary paper code. They define the RMSE (correctly) without the factor of D in the denominator, but implement the above definition in the code. To maintain consistency with the original paper, the factor of D is kept in the denominator.

NLPD

Another method in which to measure performance in probabilistic models such as this one is Negative Log Predictive Density (NLPD). If we let p denote the posterior distribution:

$$\text{NLPD} := -\sum_{n=1}^{N} \log p(\mathbf{y}_n|\mathcal{D}_T)$$

This is a measure of error rather than score (smaller is better).

Interestingly, the two performance measurement metrics do not always agree. A benefit of this metric above RMSE is that it is better motivated for probabilistic models like GPs. In addition, it takes the uncertainty of prediction (in other words, the variance of the predicted distribution) into account. This is particularly relevant in the setting of vector GPs, where it is difficult to visually display and check the predicted distribution - indeed all plots in this report only display the posterior mean. This metric is not used in the primary paper.

3.2 2D example of a Divergence-Free kernel

3.2.1 Latent function

As a proof of concept, the latent function is chosen to be a 2D divergence-free vector field, as in Example 2.2.7:

$$\begin{aligned} \mathbf{f} &: \mathbb{R}^2 \to \mathbb{R}^2 \\ \mathbf{f}(\mathbf{x}) &\mapsto \begin{bmatrix} e^{-ax_1x_2}\left(ax_1 \sin\left(x_1x_2\right) - x_1 \cos\left(x_1x_2\right)\right) \\ e^{-ax_1x_2}\left(x_2 \cos\left(x_1x_2\right) - ax_2 \sin\left(x_1x_2\right)\right) \end{bmatrix} \end{aligned}$$

3.2.2 Training set

We work with a simulated training dataset $\mathcal{D}_T = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N_T}$.

- The inputs $\mathbf{x}$ are sampled uniformly on $[0,4] \times [0,4]$.
- The outputs are noisy observations: $\mathbf{y} = \mathbf{f}(\mathbf{x}) + \boldsymbol{\varepsilon}$, where $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$.
- Here, the noise is $\sigma = 10^{-4}$ and the number of training points is $N_T = 50$.

3.2.3 Test set

The test dataset is denoted $\mathcal{D}_P = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N_P}$.

- The inputs are a uniform 20×20 grid on $[0,4] \times [0,4]$
- The outputs are noiseless evaluations of the latent function.
- The number of prediction points is $N_P = 400$.

3.2.4 Visualisation of data

3.2.5 Custom kernel Derivation

We will follow the same procedure used in Section 2.5. The basis considered here is $\boldsymbol{\xi} = \{\frac{\partial}{\partial x_i}\}_{i=1}^{2}$.

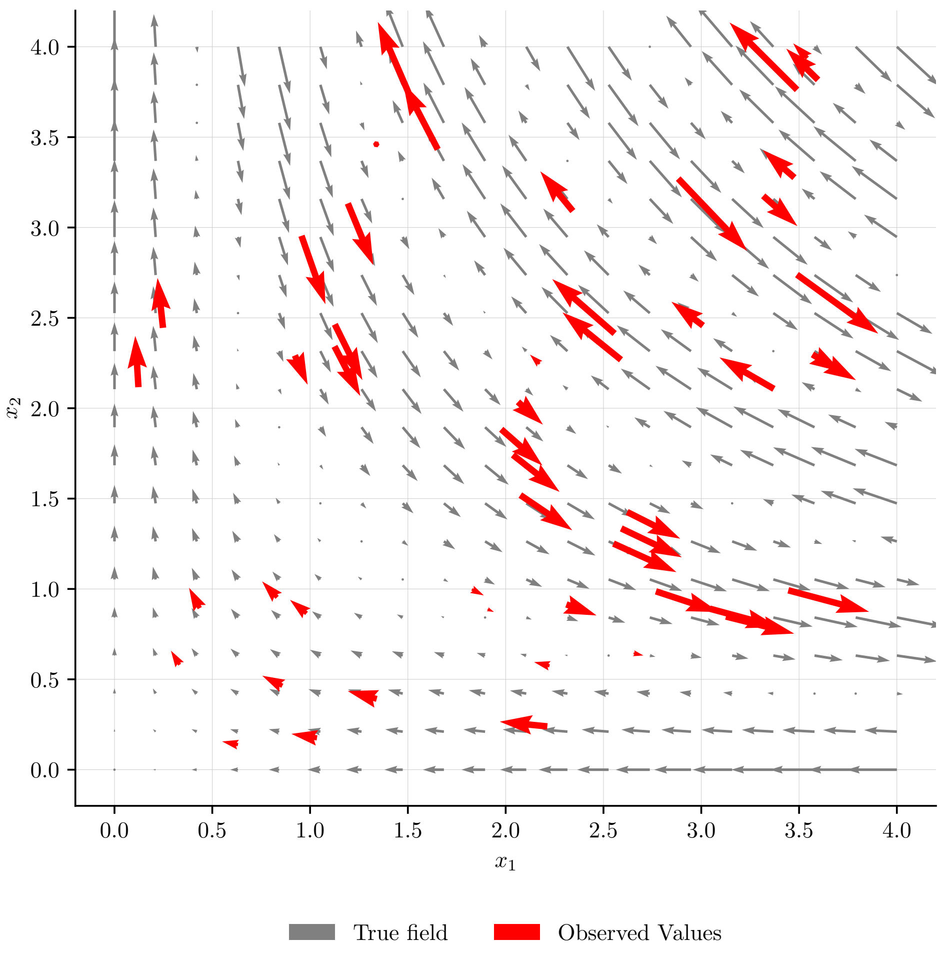

Figure 3.1: 2D simulated data, using a divergence free function. The red arrows are the train set and the grey arrows are the test set.

$$\mathscr{F} = \begin{bmatrix} \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} \end{bmatrix} \implies \mathbf{\Phi}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

By the notation introduced in Section 2.5, have

$$\begin{aligned} \mathbf{\Phi}_1 \mathbf{\Gamma} + \mathbf{\Gamma}^\top \mathbf{\Phi}_1^\top &= \mathbf{0} \\ \implies \begin{bmatrix} 2\Gamma_{11} & \Gamma_{12} + \Gamma_{21} \\ \Gamma_{21} + \Gamma_{12} & 2\Gamma_{22} \end{bmatrix} &= \mathbf{0} \\ \implies \qquad \mathbf{\Gamma} &= \lambda \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \end{aligned}$$

Therefore, have $\mathscr{G} = \begin{bmatrix} -\frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_1} \end{bmatrix}$. By Theorem 2.2.9, we have that for any mean-differentiable $g \sim \mathcal{GP}(m_g, K_g)$:

$$\begin{aligned} \mathscr{G} g &\sim \mathcal{GP}\left(\mathscr{G} m_{\mathbf{g}}, \mathscr{G} K_g \mathscr{G}^\top\right) \\ \mathscr{G} g &\sim \mathcal{GP}\left(\begin{bmatrix} -\frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_1} \end{bmatrix} m_g, \begin{bmatrix} \frac{\partial^2}{\partial x_2 x_2'} & -\frac{\partial^2}{\partial x_2 x_1'} \\ -\frac{\partial^2}{\partial x_1 x_2'} & \frac{\partial^2}{\partial x_1 x_1'} \end{bmatrix} K_g(\mathbf{x}, \mathbf{x}')\right) \end{aligned}$$

For example, here a squared exponential (SE) kernel (Definition 2.1.2) is used with zero mean. Since it is an stationary kernel, $\frac{\partial}{\partial x_d} K_{SE}(\mathbf{x} - \mathbf{x}') = -\frac{\partial}{\partial x'_d} K_{SE}(\mathbf{x} - \mathbf{x}')$, therefore, we have:

$$\mathscr{G}\mathbf{g} \sim \mathcal{GP}\left(\mathbf{0}, \begin{bmatrix} \frac{1}{\ell^2} - \frac{x_2^2}{\ell^4} & \frac{x_1 x_2}{\ell^4} \\ \frac{x_1 x_2}{\ell^4} & \frac{1}{\ell^2} - \frac{x_1^2}{\ell^4} \end{bmatrix} \sigma_f^2 \exp\left(-\frac{1}{2\ell^2} \|\mathbf{x} - \mathbf{x}'\|_2^2\right)\right)$$

3.2.6 Optimisation and Initialisation of parameters

The choice between the BFGS and Adam optimisation algorithms was determined using regular training points, specified with the `--regular` flag. Under these conditions, both algorithms exhibited similar performance, leading to the selection of BFGS. For comparison, the Adam algorithm was tested with a step size of 0.01 over 1000 steps.

Initial parameters were decided by doing a few trial runs with parameters of various scales. Typically, GP MLLs are highly multimodal distributions, but with only three parameters the Quasi-Newton optimisation seemed fairly robust, and the initialisation did not seem to matter up to two orders of magnitude around the optimum.

3.2.7 Standardised reproducibility testing

In order to compare the implementation in GPJax to the MATLAB implementation, every part of the optimisation is set to be as deterministic as possible.

- The training data is an equally-spaced, 7×7 grid.
- The test set is an equally-spaced 20×20 grid.
- For the artificial kernel, the train dataset is augmented by $\mathscr{F}\mathbf{f}$ evaluated on the same subset of test points.
- Both implementations are set to use the deterministic BFGS algorithm.
- The optimisations were performed with identical starting values.

The adapted MATLAB code can be found in the repository, and the corresponding GPJax code can be run with the command `python src/main.py --2D --regular`.

The optimised kernel parameters, as well as the resulting negative marginal log-likelihoods and the performance of the models (measured by RMSE) are listed in Table 3.3. This analysis is done for the diagonal kernel and custom kernel. The artificial kernel is actually not trained in the MATLAB code - instead, the optimised parameters from the diagonal kernel are reused. This investigated further in Section 3.2.11. Therefore, for the artificial kernel, the optimised parameters and optimal MLL are not compared.

Notably, the optimised negative marginal log-likelihoods in both implementations are remarkably similar, matching up to the fourth decimal point.

	GPJax	**MATLAB**
Diagonal		
σ_f	1.6320	1.6320
$\ell \times 10^2$	9.6439	4.5198
$\sigma_n \times 10^4$	0.99994	8.2942
NMLL	187.05	187.06
RMSE	1.4056	1.5194
Custom		
σ_f	1.3434	1.3437
ℓ	0.68482	0.68487
$\sigma_n \times 10^8$	3.2839	-0.51024
NMLL	103.91	103.91
RMSE	0.25662	0.25663
Artificial		
RMSE (25 artificial points)	1.4029	1.5194
RMSE (100 artificial points)	1.3988	1.5191
RMSE (400 artificial points)	1.3782	1.5182

Table 3.3: Comparison of GPJax and MATLAB results for the 2D simulation.

3.2.8 Visualising the fit with regular training points

Whilst the regular training points were originally created to test the quality of reproduction, they also prove to be a good way to visualise and intuit the different methods of linear constraint encoding.

Diagonal kernel

The diagonal kernel is completely ignorant to the divergence-free constaint, so naturally creates impossible flows, seen in Figure 3.2. Since the GP was trained with a zero mean function (which is a common practice when there is no prior information in the data), the GP predicts zero far from the training points. There are issues with very large flows being predicted near very small flows, as might be expected. Here the RMSE is 1.4056. For reference, predicting zero everywhere gives an RMSE of 1.581.

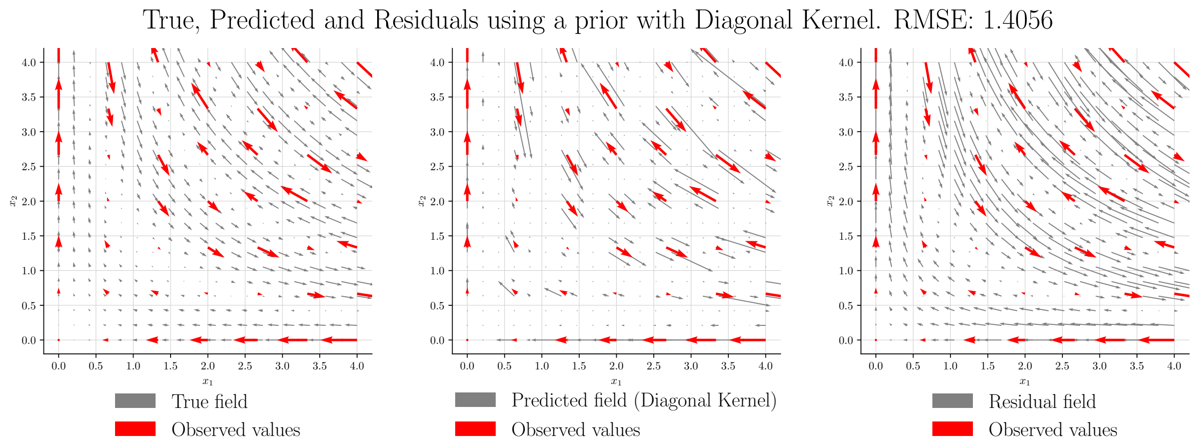

Figure 3.2: Regularly spaced training points, predicted with a Diagonal kernel

Artificial kernel

The artificial kernel enforces the linear constraint locally on a randomly selected subset of the test set. Figure 3.3 was produced with 50 artificial points, and Figure 3.4 with 400. Interestingly, when the train points are regular, enforcing the constraint at every test point does not produce very good results, with an RMSE of 1.3782. This demonstrates the importance of enforcing the linear constraint throughout the domain, as well as how local the effect of these artificial points can be.

It should also be noted that the artificial points increase the size of the dataset greatly (the original dataset has 50 points and with every test point included as an artificial point, the training dataset has 450 points).

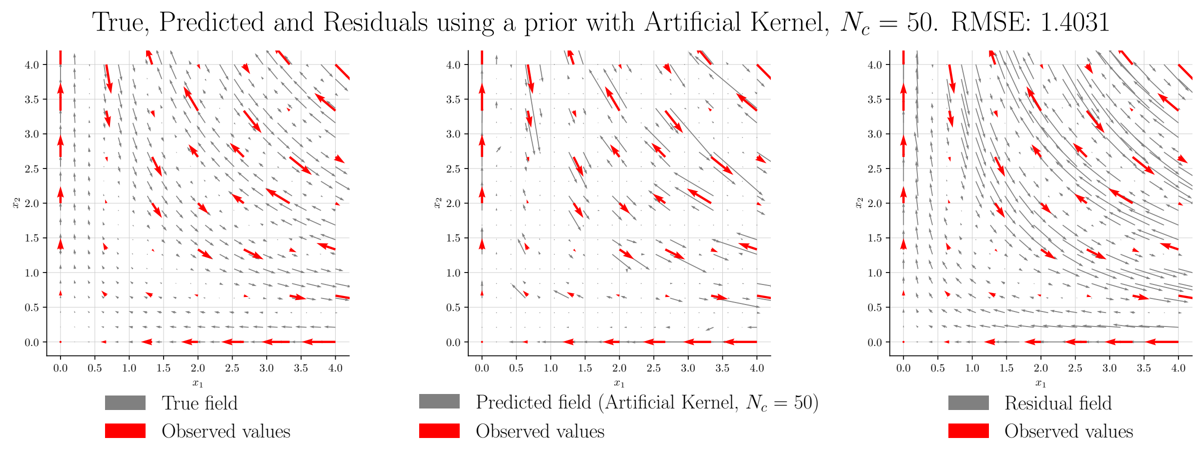

Figure 3.3: Regularly spaced training points, predicted with a Artificial kernel with 50 inducing points

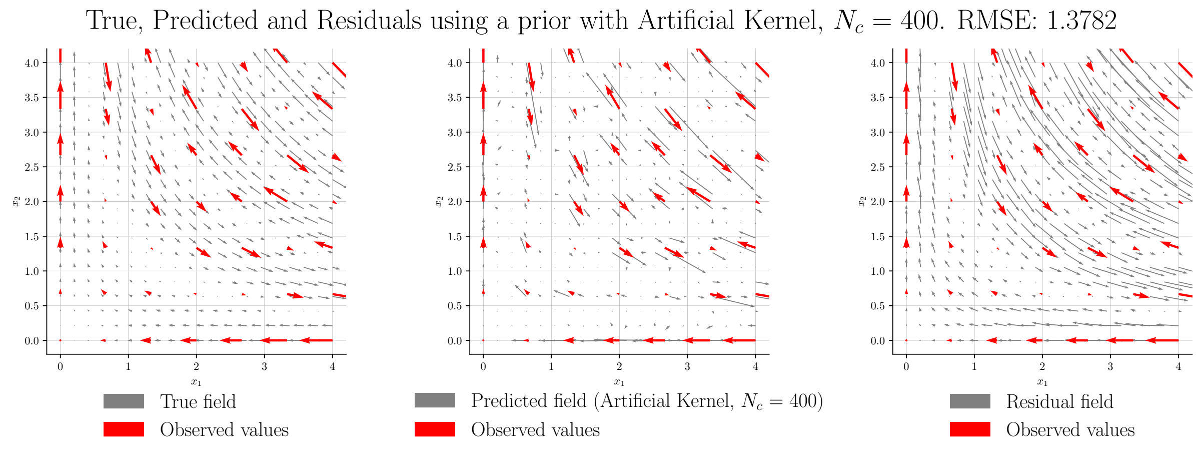

Figure 3.4: Regularly spaced training points, predicted with an Artificial kernel with 400 inducing points

Custom kernel

The custom kernel performs extremely well on regular train points, as seen in Figure 3.5, outperforming both other kernels with an RMSE of 0.2566. All realisations of the prior GP with this kernel only include divergence-free functions, so the linear constraint is enforced globally.

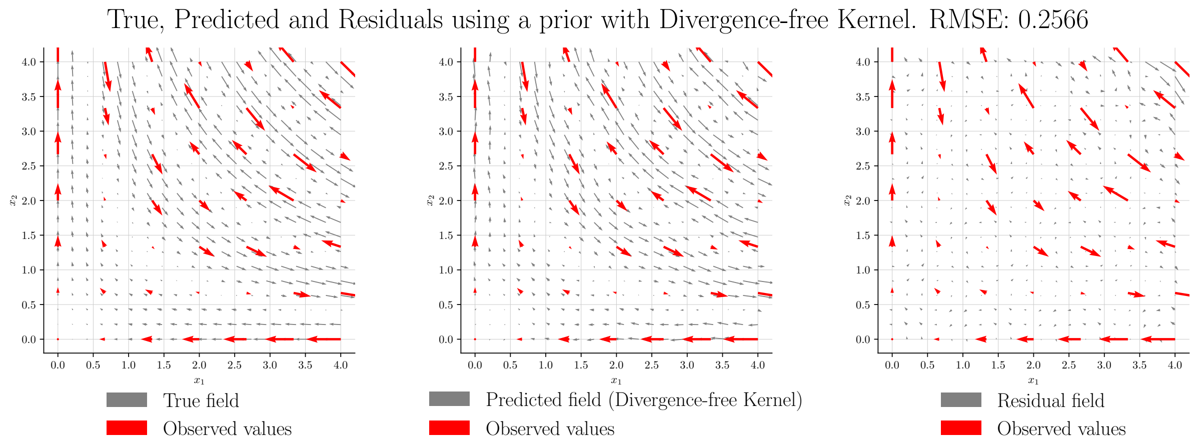

Figure 3.5: Regularly spaced training points, predicted with a Custom kernel

3.2.9 Visualising the fit with random training points

Example fits for a random selection of training points are given below, for each of the kernels. When compared to the regular train points in the previous section, the diagonal kernel and artificial kernels perform far better, whereas the custom kernel tends to perform worse.

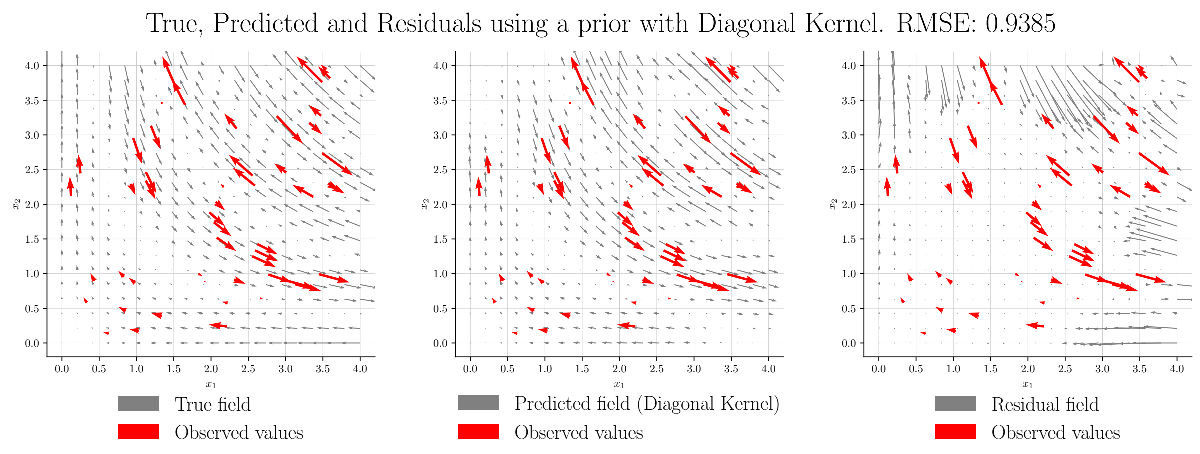

Figure 3.6: Randomly spaced training points, predicted with a Diagonal kernel

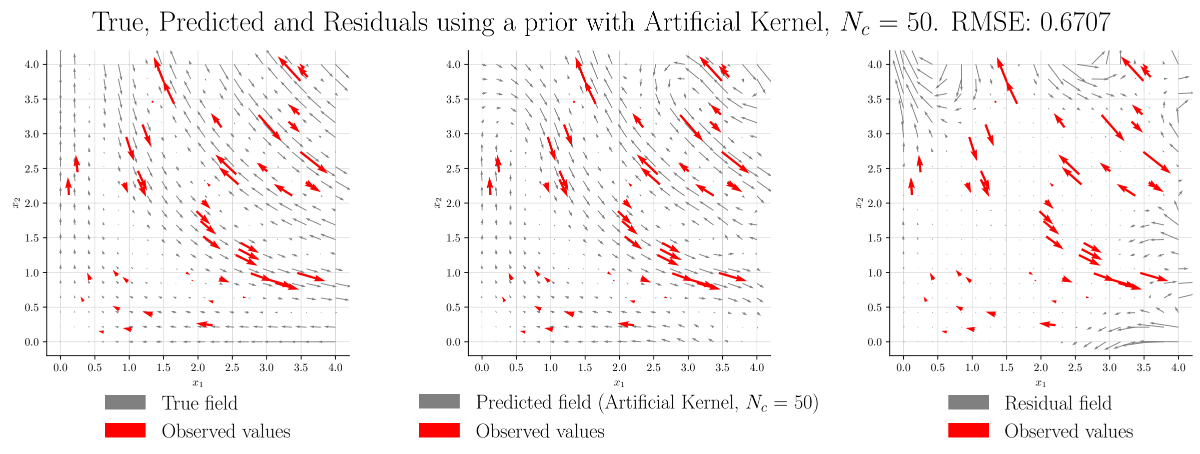

Figure 3.7: Randomly spaced training points, predicted with an Artificial kernel, using 50 inducing points

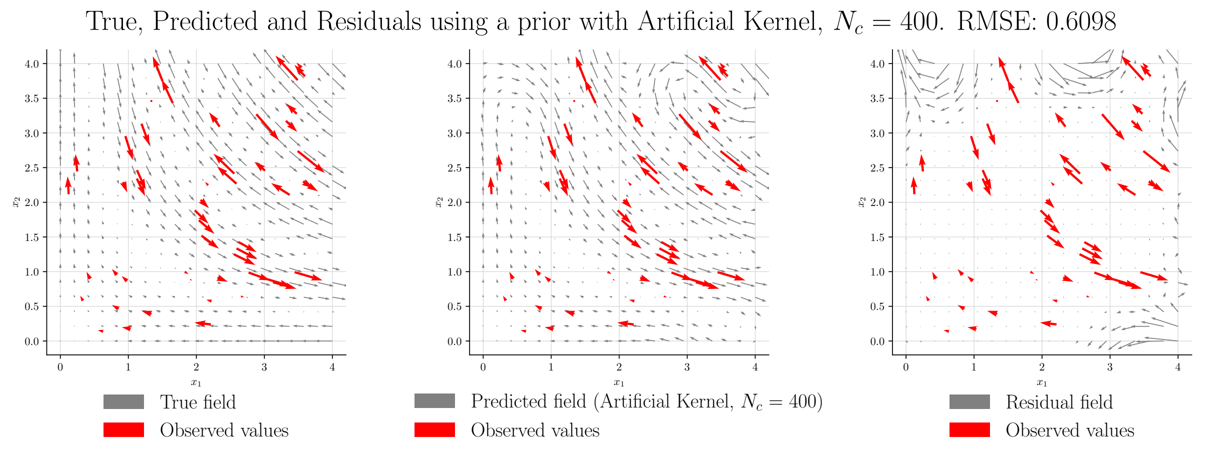

Figure 3.8: Randomly spaced training points, predicted with an Artificial kernel, using 400 inducing points

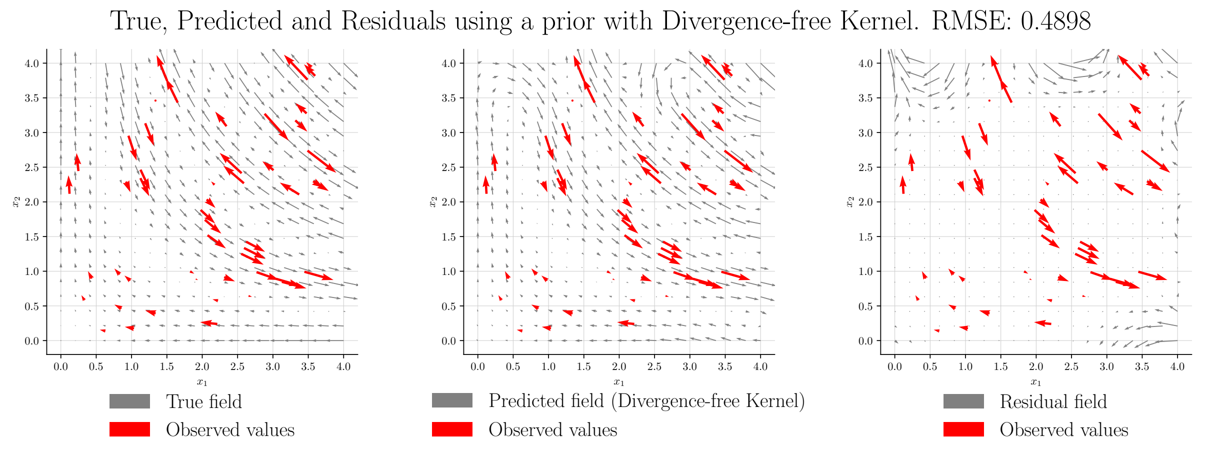

Figure 3.9: Randomly spaced training points, predicted with an Custom kernel

3.2.10 Aggregate results over 50 runs

In order to get some idea for the performance over different random training datasets, 50 runs were performed. Figure 3.10 shows the RMSE for various numbers of artificial points and is a reproduction of Figure 3(a) in the primary paper. The values of RMSE match very closely to that of the original paper. Figure 3.11 shows the NLPD for various numbers of artificial points.

The figures show the expected result that the custom kernel performs best by both performance metrics. As the number of artificial points increases, the linear constraint is encoded at more points, so the performance of the artificial kernel improves.

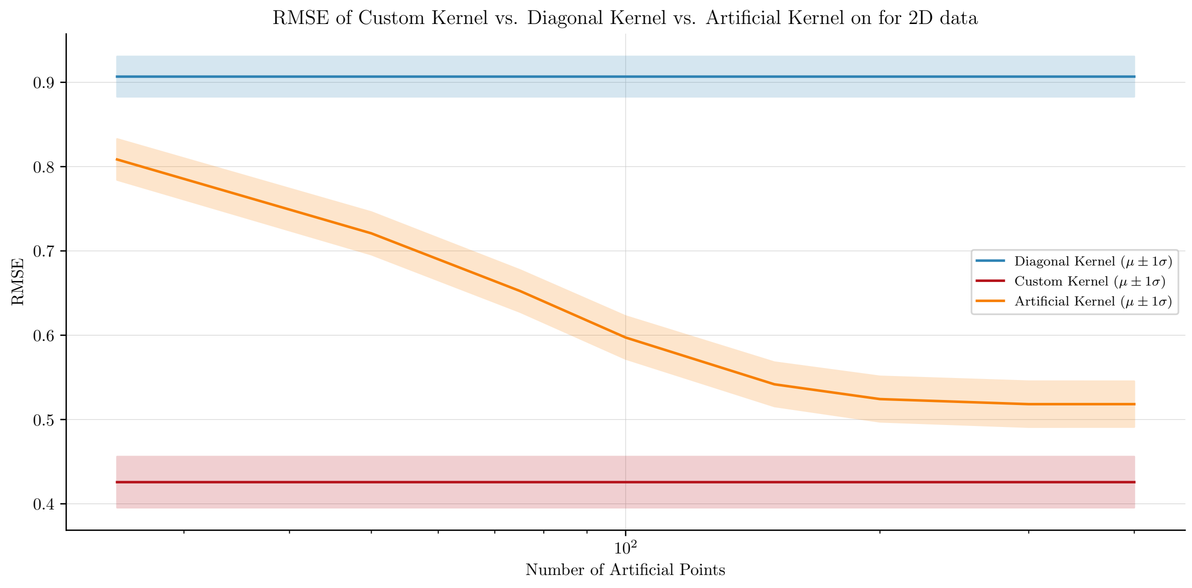

Figure 3.10: RMSE aggregated over 50 runs. The mean is plotted in a solid line and 1 standard deviation is shaded.

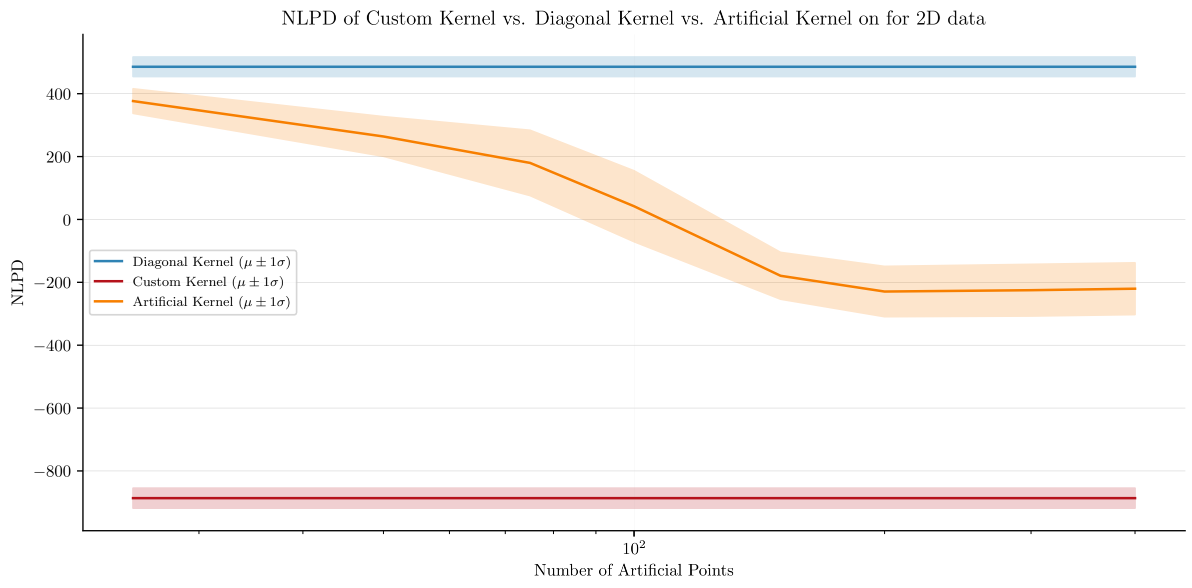

Figure 3.11: NLPD aggregated over 50 runs. The mean is plotted in a solid line and 1 standard deviation is shaded.

3.2.11 Training the artificial kernel

The original MATLAB code does not train the artificial kernel, but instead reuses the trained parameters from the diagonal kernel. This is because the likelihood (Section 2.3.2) assumes a homoscedastic error on training points, σ_n^2:

$$p(\{\mathbf{y}\}|\mathbf{f}, \{\mathbf{x}\}) = p\left(\overline{\mathbf{y}}|\overline{\mathbf{f}(\mathbf{x})}\right) = \mathcal{N}(\overline{\mathbf{f}(\mathbf{x})}, \sigma_n^2 \mathbf{I}_{NK})$$

However, the artificial points enforce the fact that $\mathfrak{F}_{\tilde{\mathbf{x}}}\mathbf{f} = 0$ for every inducing point $\tilde{\mathbf{x}}$, and this is known with complete certainty. Therefore, this augmented dataset should not be used for parameter optimisation - only prediction. Since the artificial kernel is identical to the diagonal kernel on the regular training set, it is accurate to use the same optimised parameters.

In order to demonstrate the claim that optimisation should exclude the artificial points, we ran the simulation whilst including them (this can be enabled with the `--train` flag). The aggregate results for RMSE are shown in Figure 3.12 and NLPD are shown in Figure 3.13. It can be seen that by both metrics, the fits get very poor when the number inducing points increases. For a point of comparison, the RMSE and NLPD for a zero prediction are also plotted.

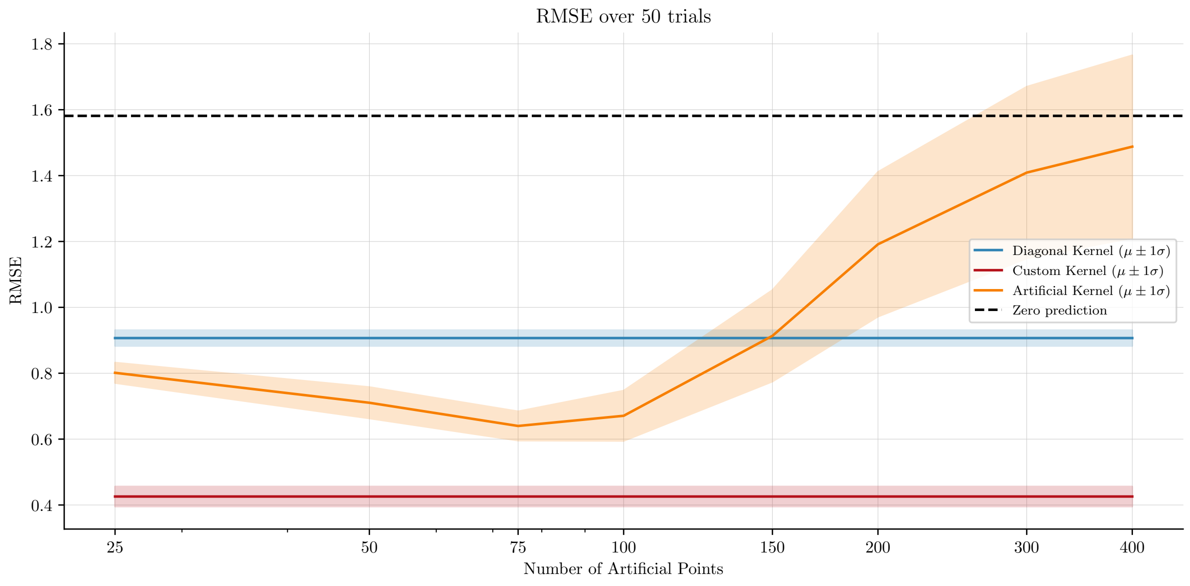

Figure 3.12: RMSE over an aggregate of 50 runs for an artificial kernel when optimising parameters over the augmented dataset.

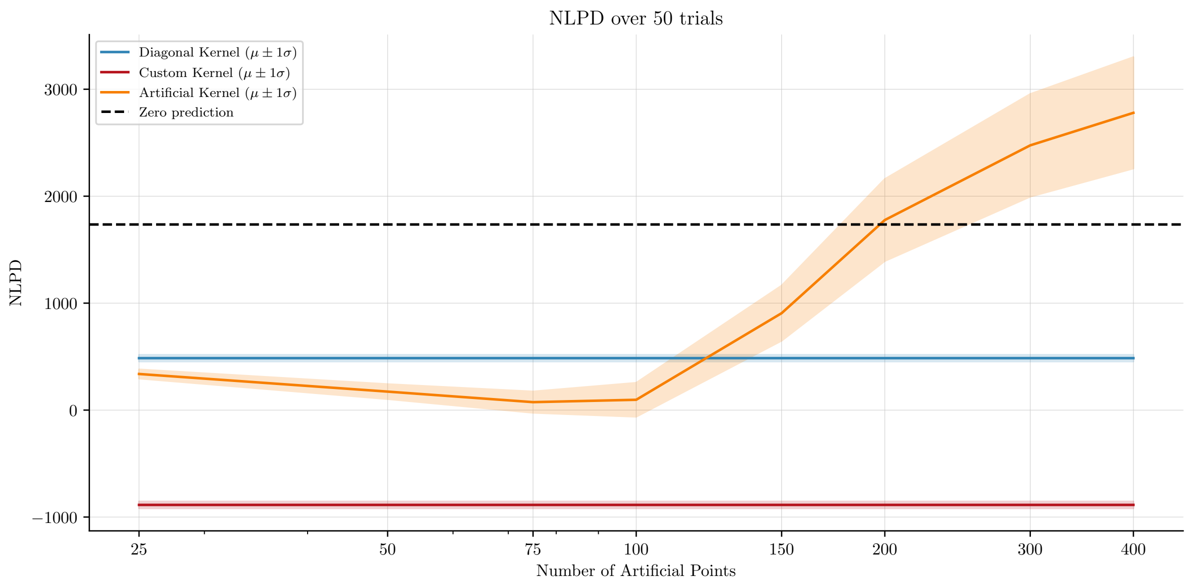

Figure 3.13: NLPD over an aggregate of 50 runs for an artificial kernel when optimising parameters over the augmented dataset.

3.3 3D example of a Curl-Free kernel

In order to demonstrate the different kernels on a real dataset, the primary paper performs an experiment, whereby a magnetic sensor is moved around a magnetically distorted indoor environment.

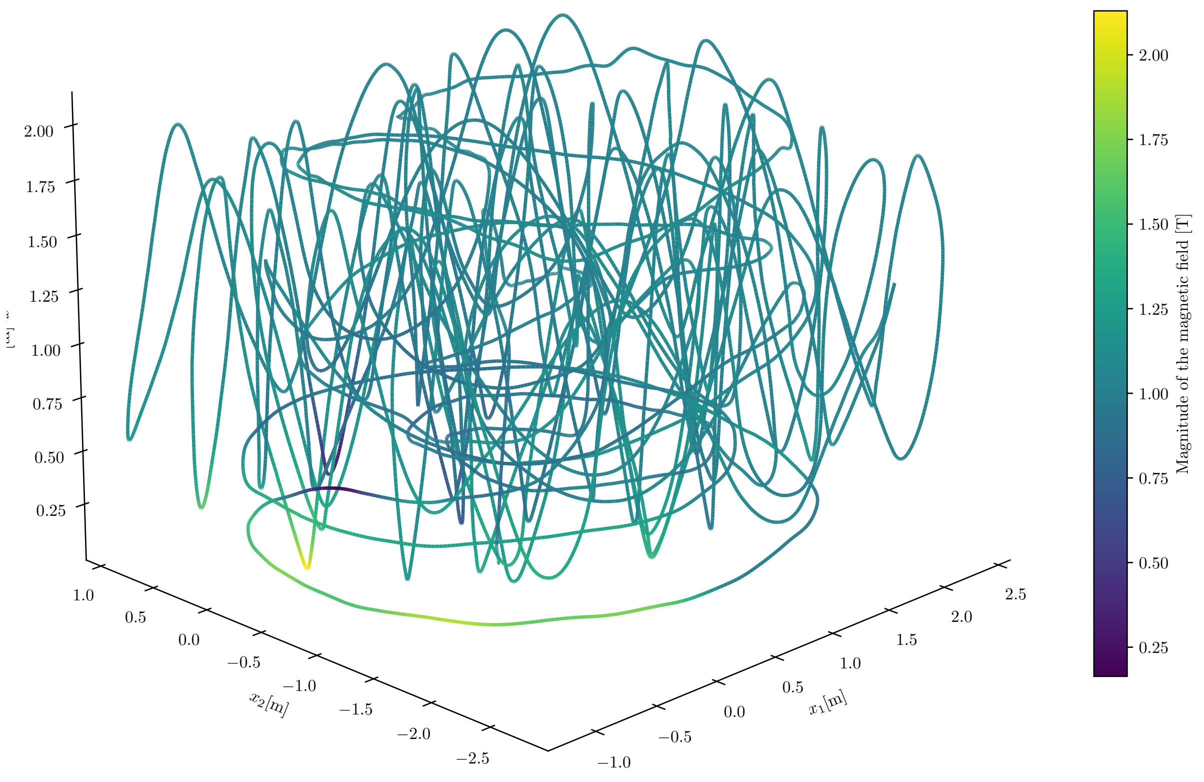

Figure 3.14: 3D observation of data. The magnitude of observed magnetic field is coloured.

3.3.1 Latent function

By Maxwell's equations, the curl of a magnetic field follows eq. (3.3). In this environment, where there is no electric field or current present, the magnetic field is curl-free.

$$\nabla \times \mathbf{B} = \mu_0 \left(\mathbf{J} + \varepsilon_0 \frac{\partial E}{\partial t} \right) \tag{3.3}$$

3.3.2 Training set

We work with time series data with 16782 entries of the position of the magnetic sensor, and the detected direction of the magnetic field. The time is irrelevant, so is discarded. The training data consists of 500 uniformly sampled points on the path.

3.3.3 Test set

The test data consists of 1000 uniformly sampled points on the path. Since this is a real dataset, the test errors are likely to have the same observation error as the train set.

3.3.4 Visualisation of data

The path is visualised in Figure 3.14, using colours to encode the magnitude of the measured magnetic field.

3.3.5 Custom kernel

We will follow the same procedure used in Section 2.5. The basis considered here is $\boldsymbol{\xi} = \{\frac{\partial}{\partial x_i}\}_{i=1}^{3}$.

$$\mathscr{F} = \begin{bmatrix} 0 & \frac{\partial}{\partial x_3} & -\frac{\partial}{\partial x_2} \\ -\frac{\partial}{\partial x_3} & 0 & \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} & -\frac{\partial}{\partial x_1} & 0 \end{bmatrix}$$

This gives us the following coefficient matrices:

$$\boldsymbol{\Phi}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}, \boldsymbol{\Phi}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \boldsymbol{\Phi}_3 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

By the notation introduced in Section 2.5, have

$$\begin{aligned} & \boldsymbol{\Phi}_l\boldsymbol{\Gamma} + \boldsymbol{\Gamma}^\top\boldsymbol{\Phi}_l^\top = \mathbf{0} \\ \implies \quad & \boldsymbol{\Gamma} = \lambda\mathbf{I}_3 \end{aligned}$$

Therefore, have $\mathscr{G} = \begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{bmatrix} = \boldsymbol{\nabla}$.

By Theorem 2.2.9, we have that for any mean-differentiable $g \sim \mathcal{GP}(m_g, K_g)$:

$$\begin{aligned} \mathscr{G}g &\sim \mathcal{GP}\left(\mathscr{G}m_g, \mathscr{G}K_g\mathscr{G}^\top\right) \\ \mathscr{G}g &\sim \mathcal{GP}\left(\begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{bmatrix} m_g, \mathbf{H}K_g(\mathbf{x}, \mathbf{x}')\right) \end{aligned}$$

where $H_{ij} := \frac{\partial^2}{\partial x_i x_j'}$

3.3.6 Optimisation and Initialisation

Similarly to Section 3.2.6, the optimisation algorithm was chosen by looking at regularly spaced training and test sets (enabled by the `--regular` flag). For Adam, the step size was set to $\alpha = 0.01$, with 80 iterations. Similarly to the 2D case, both algorithms performed similarly, as measured by optimised MLL, RMSE and NLPD values. However Adam took around 1.5 times longer to perform the optimisation, so here BFGS was preferred. The full results are displayed in Table 3.4.

	BFGS	**Adam**
Diagonal		
MLL	**2059.7132**	2059.7116
RMSE	**0.031626**	0.031634
NLPD	-5345.5801	**-5347.0050**
Curl-free		
MLL	**2237.0964**	2237.0778
RMSE	0.027767	**0.027765**
NLPD	**-4768.9589**	-4748.3381
Time taken (s)	**148.40**	235.92

Table 3.4: Comparison of BFGS and Adam Optimisation Algorithms

Similarly to the 2D case, multiple values of initial parameters were tried at different scales before settling on the following initial parameters: $\sigma_n = 0.02, \ell = 0.5, \sigma_f^2 = 2$. Once again, the optimisation seemed robust to initialisations at different scales, implying a convex loss surface, This is perhaps not surprising at low dimensions.

3.3.7 Standardised Reproducibility Testing

In an attempt to standardise the program and see how well the MATLAB code and GPJax code matched, another interesting part of the original code was discovered. The MATLAB code does not use the square exponential kernel. Instead, it uses the following kernel, labeled K_b here.

$$K_b(\mathbf{x}, \mathbf{x}'; \sigma_b, \sigma_f, \ell) := K_{SE}(\mathbf{x}, \mathbf{x}'; \sigma_f, \ell) + \sigma_b^2$$

This altered kernel introduces a new parameter which controls a global covariance between points, regardless of how far away they may be. This is another example of a subtle point that was not discussed in the original paper, but is made clear by sharing code.

Furthermore, one of the files, this parameter is named the “Standard deviation of Earth’s magnetic field components”, which perhaps implies that instead of adding σ_b^2 to each component of the Gram matrix, it was only intended to be applied to the diagonal.

In our implementation, a regular squared exponential kernel is used, so a reproducibility analysis at the same level of detail as Section 3.2.7 is not possible. Still, the aggregate RMSE and NLPD may be compared.

3.3.8 Individual fits

It is somewhat difficult to accurately display functions from $\mathbb{R}^3 \to \mathbb{R}^3$, so the performance metrics are especially important here. For completeness, the fits are displayed in the same format as the 2D plots for the diagonal kernel, artificial kernel and custom kernel in Figures 3.15, 3.16 and 3.17, but it is difficult to infer much from them.

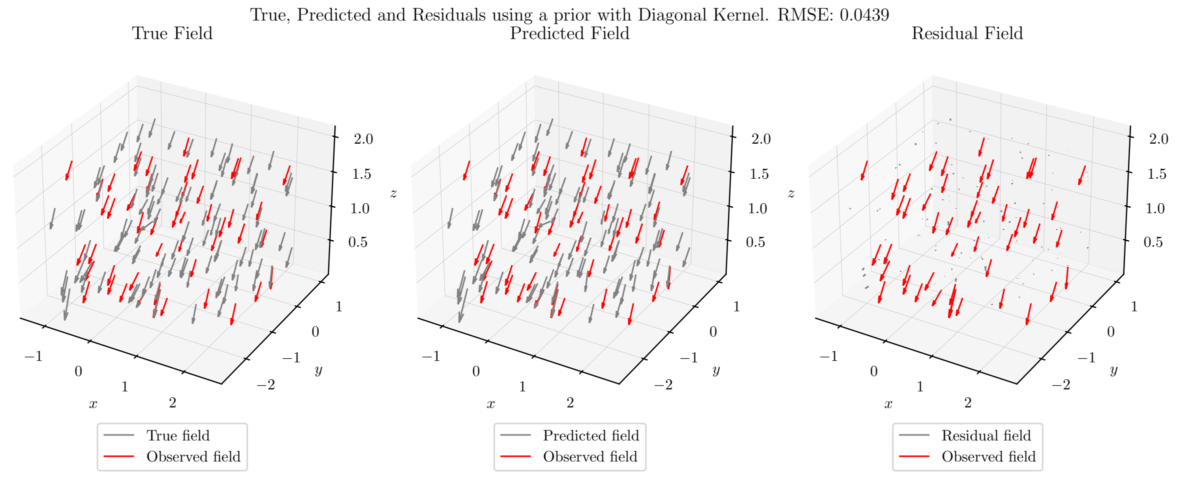

Figure 3.15: Visualisation of the fit with a diagonal kernel. Only 50 train points (red) and 100 test points (grey) are shown.

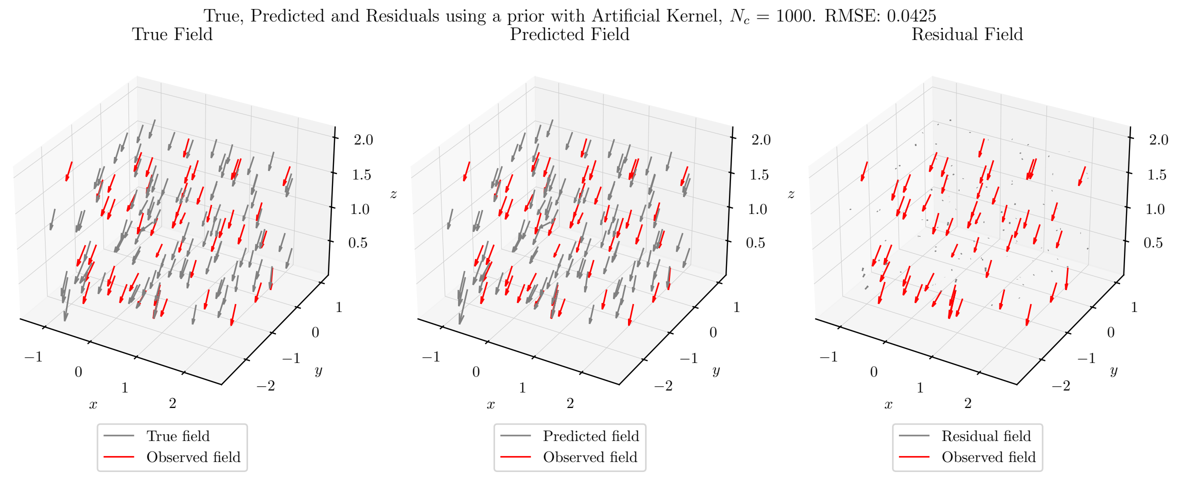

Figure 3.16: Visualisation of the fit with an artificial kernel, with $N_c = 1000$ points. Only 50 train points (red) and 100 test points (grey) are shown.

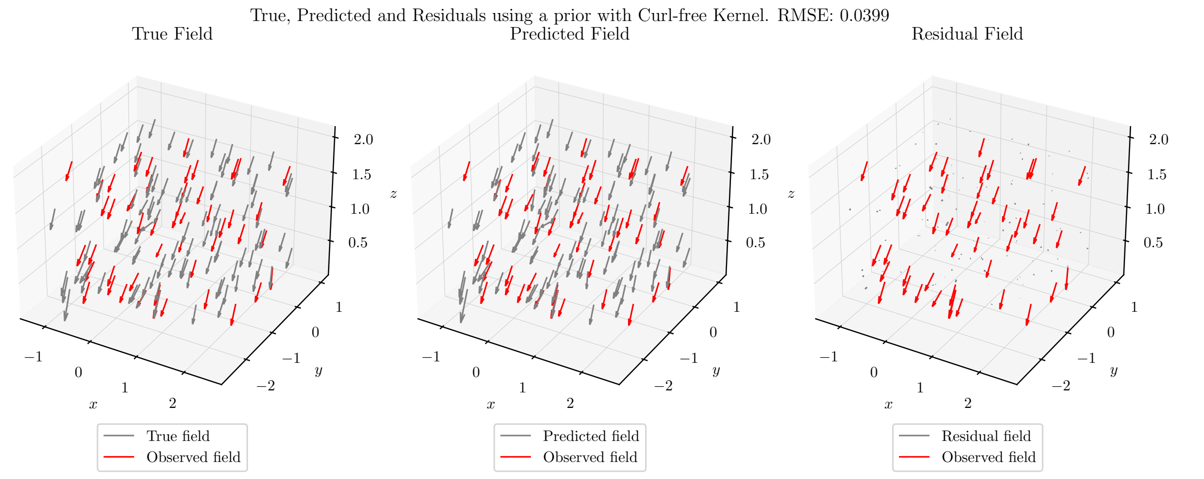

Figure 3.17: Visualisation of the fit with the custom, curl-free kernel. Only 50 train points (red) and 100 test points (grey) are shown.

3.3.9 Aggregate performance over 50 trials

Once again, 50 trials were performed with different training data in order to get an idea for the distribution. Figure 3.10 shows the RMSE for various numbers of artificial points and is a reproduction of Figure 3(b) in the primary paper. The RMSE values in the reproduction are much higher (ranging from 0.041 to 0.046), whereas in the paper, they range from 0.034 to 0.038. This is expected behaviour, since the original paper uses a more flexible kernel than the squared exponential kernel that was used in the reproduction.

Another difference is that in the original paper, the RMSE achieved by enforcing the constraint at almost every test point almost matches that achieved by the custom kernel. This is not observed in the reproduction.

This graph still demonstrates the main point of the paper, though - that the custom kernel produces far a far superior RMSE than the diagonal kernel, even with artificial points which enforce the constraint locally. In addition, as the number of artificial points increases, the performance of the artificial kernel improves. The same uptick in the final step is also observed here due to numerical errors from a very large and ill-conditioned Gram matrix. Indeed, including 1000 artificial points and flattening the three dimensions gives an effective training set size of 4500[2].

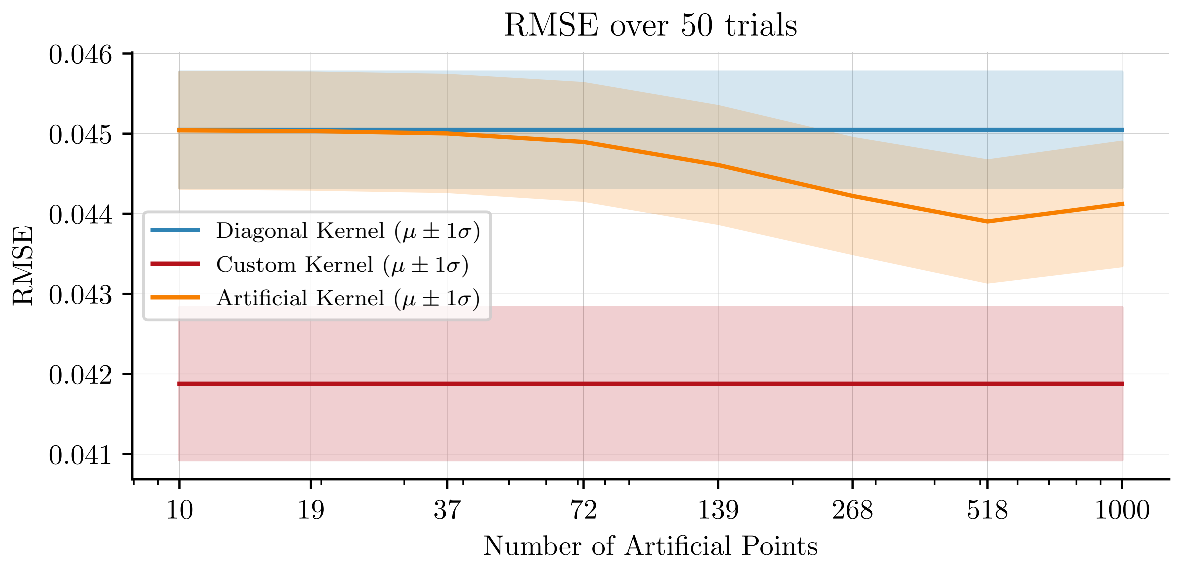

Figure 3.18: RMSE aggregated over 50 runs. The mean is plotted in a solid line and 1 standard deviation is shaded.

Figure 3.11 shows the NLPD for various numbers of artificial points. Here the picture is not so clear. By this metric, the exact opposite result holds true - the custom kernel appears to have the worst NLPD for the most part, and the diagonal the best. Increasing the artificial points worsens the NLPD of the model.

Since the NLPD is the only metric that takes predicted standard deviation into account, it could be that the kernel with greater information on the linear constraint tend to be over-confident, with a falsely small predicted covariance. This would explain the opposite trends in NLPD and RMSE.

[2] $D \times (N_T + N_c) = 3 \times (500 + 1000) = 4500$.

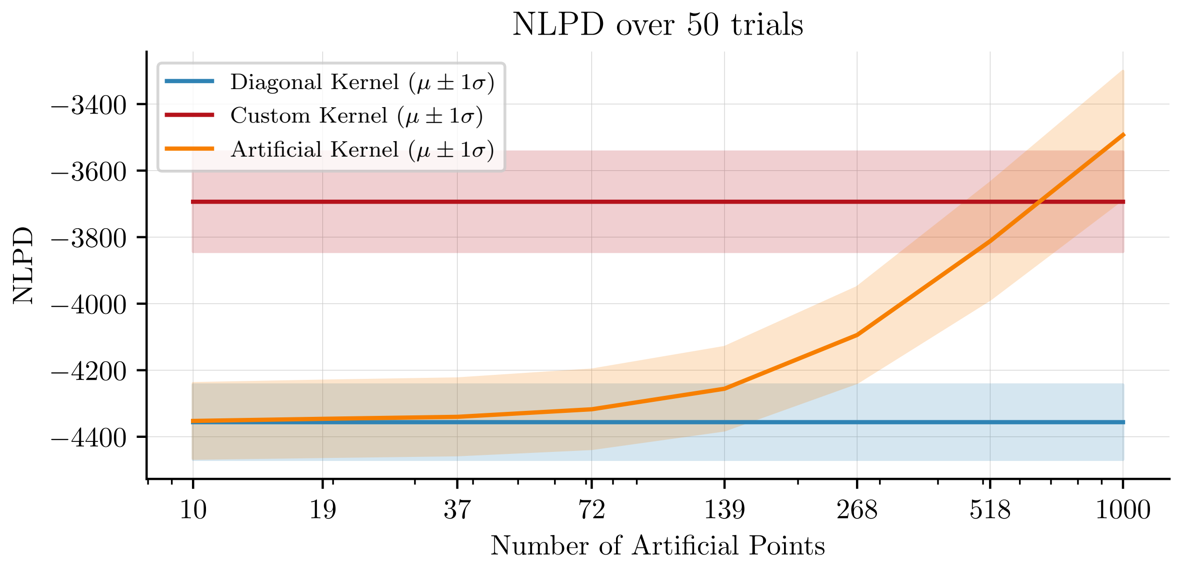

Figure 3.19: NLPD aggregated over 50 runs. The mean is plotted in a solid line and 1 standard deviation is shaded.

Chapter 4

Going further

4.1 Noise study

In order to test the robustness of the two methods of linear constraints, the standard deviation of Gaussian noise added to the training set was varied and performance was measured. The RMSE for the 2D and 3D studies are plotted in Figures 4.1 and 4.2.

As expected, as noise increases, so does the RMSE and NLPD, but it does so in a very sharp fashion - indeed, the figures are on a log scale. This is because if most data is on the same order of magnitude it should be expected that once noise is added at a greater order of magnitude, almost all predictive power is lost. Indeed, in the 2D case, the maximum magnitude of any vector is 4.60, and the mean is 1.91. This is roughly where the RMSE increases.

In addition to this, the RMSE for the different methods converges as noise is increased. This is also expected, since as the noise increases, the observed data follows the linear constraint to a lesser extent. Correspondingly, any encoding of this linear constraint into GPR is decreasingly relevant.

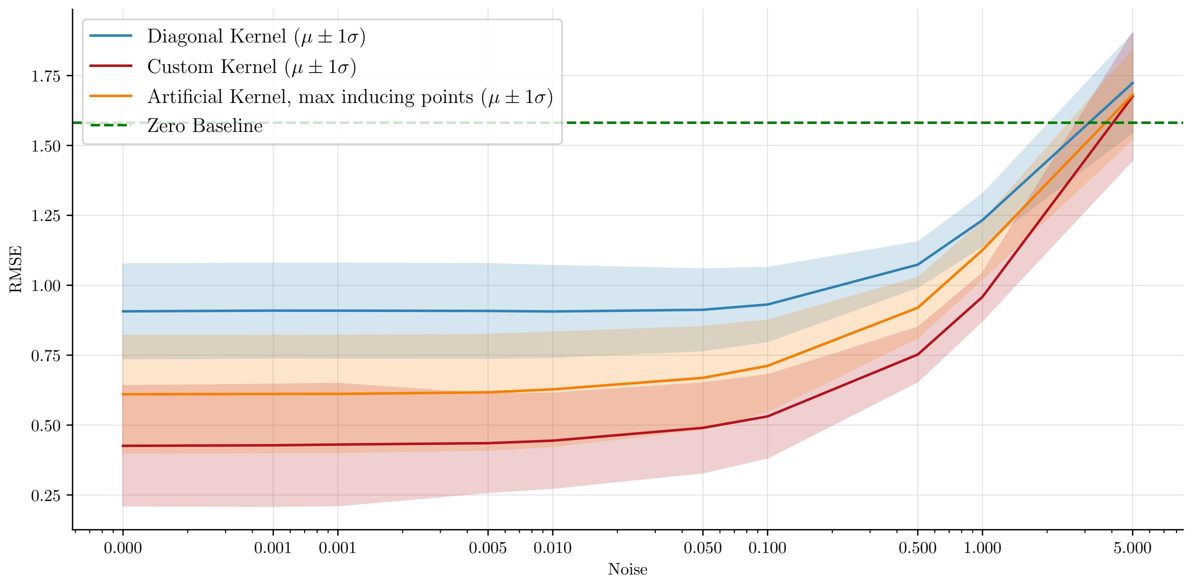

Figure 4.1: RMSE as a function of added noise for the 2D study.

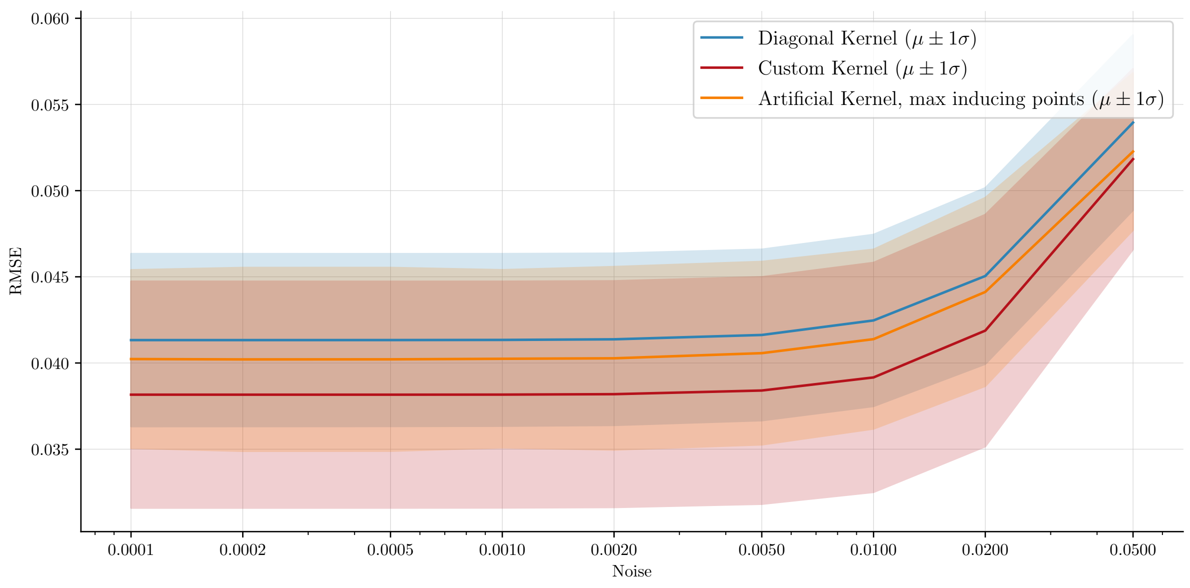

Figure 4.2: RMSE as a function of added noise for the 3D study.

4.2 Harmonic functions

The two kernels considered in this paper were already known in [**Wahlström**, Dec. 2015], but rederived using the procedure in Section 2.5.1. Here we try to extend the general procedure to derive the appropriate kernel for a Laplacian-free constraint, namely:

$$\mathscr{F} f = \nabla^2 f = \frac{\partial^2 f}{\partial x_1^2} + \frac{\partial^2 f}{\partial x_2^2} + \frac{\partial^2 f}{\partial x_3^2} = 0$$

Here, since there are second derivatives, the basis is $\mathcal{P}_{2,3}^{1\times 1}$, which is the space of homogeneous quadratics in $\left\{\frac{\partial}{\partial x_i}\right\}_{i=1}^{3}$.

Higher order operator equations are referenced in the supplementary material of the primary paper, and proceed in a similar fashion to the general process of Algorithm 1, except the basis is now $\boldsymbol{\xi} = \{\xi_i \xi_j\}_{i=1,j=1}^{3,3}$, so there is an extra index required.

Writing $\mathscr{F}_{\mathbf{x}} : (\mathbb{R}^3 \to \mathbb{R}^1) \to \mathbb{R}^1$ as a 1×1 matrix, and $\mathscr{G} : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$ as a vector of length 1:

$$\mathscr{F}_{11} = \phi_{11k_1k_2}\xi_{k_1}\xi_{k_2}$$
$$\mathscr{G}_1 = \Gamma_{1k_1'k_2'}\xi_{k_1'}\xi_{k_2'}$$

where $\phi_{11k_1k_2} = \delta_{k_1k_2}$

Thus, we require that

$$\phi_{11k_1k_2}\Gamma_{1k_1'k_2'}\xi_{k_1}\xi_{k_2}\xi_{k_1'}\xi_{k_2'} = \Gamma_{1k_1'k_2'}\xi_{k_1}\xi_{k_1}\xi_{k_1'}\xi_{k_2'} = 0$$

But then, we have that $\Gamma_{1k_1'k_2'} = 0$ for every k_1', k_2', which means that the operator $\mathscr{G}$ must be trivial. The algorithm then informs us that we should extend our basis, because the assumption that $\mathfrak{G}$ lies in $\boldsymbol{\xi}$ is insufficient. However, extending the basis does not help - fundamentally the problem is overconstrained.

Of course, this does not imply that the nullspace of the Laplacian operator is trivial - indeed, there are many harmonic functions, for example $f(x, y, z) = (x^2 + y^2 + z^2)^{-1/2}$. This particular algorithm just could not encode such functions as realisations of a GP.

The heat equation, which is a similar constraint, faces the same problem of being overconstrained.

$$\mathfrak{F}f = \nabla^2 f - \frac{\partial}{\partial t} f = 0$$

Chapter 5

Conclusion

5.1 Summary

In summary, in this report, the mathematics of the original paper was reproduced with greater detail, including rigorous definitions and proofs and some examples of where the method works. We also demonstrate that the method in the paper does not provide a useful custom kernel in the case of a Laplacian-free condition.

The 2D case study was reproduced in the GPJax library with a great degree of accuracy. This was verified firstly by running the simulation study in a deterministic setting in both the MATLAB and GPJax code, and also by comparing the aggregate results over 50 trials with randomly selected training data. In both cases, the results were extremely similar to that achieved in [**Jidling et al.**, 2017].

The 3D case study highlighted the importance of sharing code as an open science practice, since it allowed the discovery that an altered version of the squared exponential kernel was used in the original paper. The extra parameter meant that an accurate reproduction was not possible, although the GPJax implementation supported the paper's claim when measuring performance by RMSE.

The introduction of a new performance metric, the NLPD, interestingly did not support the claim in the 3D case, but did in the 2D case. This is most likely due to overconfidence in the posterior variance.

In performing the reproduction, two different optimisation algorithms were considered - BFGS, as used in the original paper, and Adam. It was found that Adam was slower and marginally less accurate. In addition, careful profiling revealed an issue in the CoLA library that was used for matrix inversion, impacting the accuracy and efficiency of prediction. This was overridden with a custom class.

Finally, a noise study was conducted to compare the robustness of different kernels to noise, and it was found that they are all affected equally, and in high noise settings, the relative benefit of using a custom kernel is low.

The method proposed in [**Jidling et al.**, 2017] offers a principled way of incorporating a large class of physical information into Gaussian Process models, improving predictive power at no additional computational cost.

5.2 Further work

Potential extensions to this project could be to

1. Find a non-trivial linear constraint which demonstrates the paper's claim, and tests the general method outlined in Section 2.5.1.
2. Attempt Adam optimisation with stochastic gradient descent - that is, use a different subset of training data to construct the MLL (eq. (3.1)).
3. Attempt this type of analysis with a real dataset governed by some known linear constraints.
4. In [**Rasmussen and Williams**, 2006], it is claimed that the Matérn kernel is better suited to real-world datasets than the squared exponential kernel. It would perhaps be interesting to compare the two.

Bibliography

Fletcher, Roger (1987). *Practical Methods of Optimization*. Second. New York, NY, USA: John Wiley & Sons.

Swan, Andy (1998). “GOOVAERTS, P. 1997. Geostatistics for Natural Resources Evaluation. Applied Geostatistics Series.” In: *Geological Magazine* 135.6. Dataset is described and provided in Appendix C, pp. 819–842. DOI: 10.1017/S0016756898631502.

Rasmussen, Carl Edward and Christopher K. I. Williams (2006). *Gaussian Processes for Machine Learning*. The MIT Press. DOI: https://doi.org/10.7551/mitpress/3206.001.0001.

Alvarez, Mauricio A., Lorenzo Rosasco, and Neil D. Lawrence (2012). *Kernels for Vector-Valued Functions: a Review*. arXiv: 1106.6251 [stat.ML].

Wahlström, Niklas (Dec. 2015). *Modeling of Magnetic Fields and Extended Object for Localization Applications*. DOI: 10.3384/diss.diva-122396.

Jidling, Carl et al. (2017). “Linearly constrained Gaussian processes”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/71ad16ad2c4d81f348082ff6c4b20768-Paper.pdf.

Kingma, Diederik P. and Jimmy Ba (2017). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

Raissi, Maziar, Paris Perdikaris, and George E. Karniadakis (2017). “Physics Informed Deep Learning Part (I): Data-driven Solutions of Nonlinear Partial Differential Equations”. In: *CoRR* abs/1711.10561. arXiv: 1711.10561. URL: http://arxiv.org/abs/1711.10561.

Bradbury, James et al. (2018). *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. URL: http://github.com/google/jax.

Bronstein, Michael M. et al. (2021). *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. arXiv: 2104.13478 [cs.LG]. URL: https://arxiv.org/abs/2104.13478.

Long, Da et al. (2022). *AutoIP: A United Framework to Integrate Physics into Gaussian Processes*. arXiv: 2202.12316.

Pinder, Thomas and Daniel Dodd (2022). “GPJax: A Gaussian Process Framework in JAX”. In: *Journal of Open Source Software* 7.75, p. 4455. DOI: 10.21105/joss.04455. URL: https://doi.org/10.21105/joss.04455.

Appendix A

Proofs

Theorem A.0.1 (Antisymmetry condition)**.** *M is antisymmetric if and only if $\boldsymbol{\xi}^\top M\boldsymbol{\xi} = 0$ for every vector $\boldsymbol{\xi}$ in a finite dimensional vector space.*

Proof. The forward direction:

If M is antisymmetric then:

$$\boldsymbol{\xi}^\top M\boldsymbol{\xi} = \boldsymbol{\xi}^\top M^\top \boldsymbol{\xi} = -\boldsymbol{\xi}^\top M\boldsymbol{\xi} \implies \boldsymbol{\xi}^\top M\boldsymbol{\xi} = 0$$

The first equality follows from the fact that transposing a scalar quantity does not change it. The second equality uses the antisymmetry assumption.

The backward direction: Since this is a finite dimensional vector space, we can consider consider two (possibly identical) basis vectors $\mathbf{e}_i$ and $\mathbf{e}_j$. Then, by assumption:

$$(\mathbf{e}_i + \mathbf{e}_j)^\top M(\mathbf{e}_i + \mathbf{e}_j) = \mathbf{e}_i^\top M\mathbf{e}_i + \mathbf{e}_i^\top M\mathbf{e}_j + \mathbf{e}_j^\top M\mathbf{e}_i + \mathbf{e}_j^\top M\mathbf{e}_j = 0$$

But by assumption, the first and last terms in the sum are zero. Also, $\mathbf{e}_i^\top M\mathbf{e}_j = M_{ij}$ by definition, so we have:

$$\forall i, j : M_{ij} = -M_{ji}, M_{ii} = 0$$

So that M is antisymmetric. $\square$

Appendix B

Use of Generative AI

B.1 Codebase

GitHub Copilot was integrated into the VSCode IDE, and used for its code autocompletion capability. This was mostly used for completing repetitive code, such as those in plotting functions.

Explicit prompts were made to Copilot and OpenAI's ChatGPT to:

- Help generate 3D plots
- Create space between subplots
- Increase the size of the font
- Edit the format of legends
- Import `.mat` files into Python
- Add docstrings to functions and classes in the Numpy style
- Convert the Markdown README file to ReST (for Sphinx documentation
- Help with making a contents page for the README

GenAI was **not** used for

- Any structural aspects of the code
- Any non-cosmetic programming logic

B.2 Report

ChatGPT was used to format tables created in Python into LaTeX. The rest of the report was written without any aid from AI.