Image Inpainting with Gaussian Processes

Dimitar Iliev Dimitrov

4th Year Project Report

Computer Science

School of Informatics

University of Edinburgh

2016

Abstract

This project provides a critical evaluation of the applicability of Gaussian Processes with the SMP kernel, presented in [21], for image inpainting and, in particular, texture synthesis. It contributes to the original work presented in [21] by providing quantitative evaluation of the performance of the framework and comparing it to the state of the art methods presented in [1]. It also discusses extensions of the original framework that make it applicable for unconstrained texture synthesis.

Acknowledgements

I thank my supervisor, Professor Chris Williams, for all of the help he provided me with throughout the project. I also thank all 4^{th} year Computer Science students that were there to share that amazing year with me. I could not have done it without you !!!

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Dimitar Iliev Dimitrov*)

Table of Contents

Chapter 1

Introduction

This chapter aims to introduces the reader to the main goals of this project and to present a roadmap to the rest of the document. It is split into sections, as follows. Section 1.1 gives some preliminary background to allow the further reading of the chapter. Section 1.2 introduces the project and its goals and comments upon some of the choices we made when we were designing our experiments. Finally Section 1.3 gives a brief description for each of the chapters to follow.

1.1 Terminology

Image Inpainting

Image inpainting is the process of filling the lost or damaged parts of an image with colours in a matter that makes the combined image to look plausible [2].

Constrained texture synthesis

Constrained texture synthesis is image inpainting for which the domain of the image to be inpainted is restricted to the set of all textures.

Gaussian Process

Gaussian processes are a flexible non-parametric machine learning framework predominantly used for regression problems.

1.2 Project Goals

This document will explore the applicability of Gaussian Processes to image inpainting of large regions in images. Indications that such a framework can work well was first provided by Wilson at [21]. We will explore the idea more rigorously, quantifying the intuitive results presented by Wilson and comparing them to the state of the art results provided by Kivinen and Williams at [1]. [1] focuses on a more narrow version of the image inpainting problem called constrained texture synthesis, and its generalisation called, simply, texture synthesis. For our experimental set-up we chose to follow Kivinen and Williams' path and explored the texture synthesis problem in more depth instead of spreading our attention across the whole image inpainting domain. Given how vast the image inpainting field has come to be, focusing on a single subfield within it fitted better the time-frame of this project. Another justification of our choice is the more robust quality measures developed for texture synthesis which allowed for more in-depth analysis of the results.

1.3 Project Outline

Chapter 2 is split intro three sections. Section 2.1 introduces the image inpainting problem and gives a brief overview of the previous work done on it. Section 2.2 focuses on the texture synthesis problem and presents the unconstrained and the constrained types of synthesis. It also considers at length what defines the perceptual quality of textures. The chapter concludes with Section 2.3 which provides basic derivations for the Gaussian Processes inference model and demonstrates the SMP kernel and the `GPR_GRID` algorithm presented by Wilson at [21], which are essential part of our experiments.

Chapter 3 motivates the choice of the only dataset we use in our experiments — the Brodatz dataset. It also provides a list of images taken from the dataset, displaying the textures we choose to train our models.

Chapter 4 starts with a brief description of the software packages used in constructing our experiments. It then goes about defining the experiments

themselves for both the constrained and the unconstrained texture synthesis cases. Finally, it concludes by presenting the results we obtained and discusses them in length.

Chapter 5 concludes the document with a brief discussion on possible improvements of the framework we used for both the unconstrained and the constrained texture synthesis.

Chapter 2

Background

This chapter presents relevant work done on image inpainting and texture synthesis. Extensive Gaussian Processes background is also included to introduce the reader to relevant material, used in the experiments described in Chapter 4. The chapter is segmented into sections as follows. First, in Section 2.1 we give the definition of the image inpainting problem and discuss the different types of image inpainting algorithms available in the literature along with some of their applications. The chapter then switches to a more in-depth discussion of texture synthesis in Section 2.2. In particular, definitions of the constrained and the unconstrained cases of texture synthesis along with some similarities and differences are discussed. The section then considers what defines the quality of texture synthesis and finishes with a discussion of the previous work done on the matter. The last section of the chapter, Section 2.3, gives an overview and relevant derivations for Gaussian Processes. Those are followed by some of the more recent work on Gaussian Processes that enables the experiments in Chapter 4. Section 2.3 ends with a discussion of the properties of Gaussian Processes that make them a good fit for the task of texture synthesis.

2.1 Image Inpainting

In this section, image inpainting is introduced along with some of the most popular classes of algorithms used in the field and some of their applications. Image inpainting is defined in [2] as follows:

Definition 2.1. *Image inpainting refers to the process of restoring missing or damaged areas in an image.*

The term comes from a set of techniques for restoring damaged parts of old paintings or photos that used to be executed manually by professional restoration specialists. First image inpainting algorithms aimed to improve that said process making it computer-assisted. Today, image inpainting encompasses whole families of algorithms that work mostly autonomously to generate missing, damaged or occluded regions in digital images.

In mathematical terms one can pose the image inpainting as follows:

Definition 2.2. *For a partially known image function* $\mathbf{I}\colon \Omega \subseteq \mathbb{N}^2 \to \mathbb{R}^m$ *from known pixel locations to vector of measurements and a subset of known observations values* $\mathbf{K}\colon \mathcal{D} \subseteq \Omega \to \mathbb{R}^m$ *one aims to find* $\mathbf{U}\colon \Omega \setminus \{\mathcal{D}\} \to \mathbb{R}^m$ *such that* $\mathbf{U} = \arg\max_{\mathbf{U}} \mathbf{E}(\mathbf{I})$*, where* $\mathbf{E}$ *is a some measurement of the physical plausibility and the visual consistency of the image.*

Definition 2.2 makes several points about the process more explicit. First, image inpainting generally is an ill-posed problem so certain assumptions must be made. Usually, those are defined in terms of the similarity of some statistics of $\mathbf{K}$ and $\mathbf{U}$. The choice of those statistics gives rise to different families of image inpainting algorithms. Second, the quality of the result is defined in terms of the function $\mathbf{E}$. Unfortunately, for many practical applications $\mathbf{E}$ cannot be defined explicitly, and manual assessment of the inpainting results is required. There are certain domains, however, for which such function can be constructed to capture at least partially the visual consistency of the reconstructed pictures. One such domain is texture synthesis with which this dissertation is mainly concerned with.

In the following section, an outline of the image inpainting taxonomy is presented as described by [3] and [2].

2.1.1 Image Inpainting Taxonomy

[3] categorizes image inpainting algorithms in five general categories:

- Texture-synthesis-based Inpainting

- Semi-automatic and Fast Digital Inpainting
- PDE-based (Partial Differential Equation) Inpainting
- Exemplar-based and Search-based Inpainting
- Hybrid Inpainting

Each of those categories is described briefly in the next few sections.

2.1.1.1 Texture-synthesis-based Inpainting

Texture synthesis is a problem closely related to image inpainting. It constrains the domain of the pictures to be inpainted to textures. Thus to give a proper definition of the texture synthesis problem, one needs to provide a definition of what texture is as well. Giving such definition, however, is not trivial. Over the years, many have tried defining the term using a mix of image statistic and desirable human perception properties. A short collection of definitions has been assembled by [4, cited by 5]. We focus our attention on two of them:

Definition 2.3. *Texture is defined for our purposes as an attribute of a field having no components that appear enumerable. The phase relations between the components are thus not apparent. Nor should the field contain an obvious gradient.The intent of this definition is to direct attention of the observer to the global properties of the display i.e., its overall coarseness, bumpiness, or fineness. Physically, nonenumerable (aperiodic) patterns are generated by stochastic as opposed to deterministic processes. Perceptually, however, the set of all patterns without obvious enumerable components will include many deterministic (and even periodic) textures.*

Definition 2.4. *A region in an image has a constant texture if a set of local statistics or other local properties of the picture function are constant, slowly varying, or approximately periodic.*

Definition 2.3 and Definition 2.4 contrast two important properties of textures. On the one hand, a texture must possess some underlying statistical properties that characterise it but on the other, being generated by some physical process, a valid instance of the same texture can greatly vary. Those properties

of textures make the texture synthesis problem difficult. Techniques addressing texture synthesis must be flexible enough to be able to generate a possibly unbounded amount of examples of the same texture without violating its statistics. The problem is complicated even further by the fact that human perception is very well tuned at noticing inconsistencies within textures. Therefore, even the smallest visual artefacts in the generated texture can induce a severe perceptual response. Two prevalent approaches to texture synthesis are present in the literature. One of them are the exemplar techniques discussed below. The other are machine-learning-based approaches that try to extract the statistical properties describing the texture from training examples. That approach is closely related to the standard machine learning problem of feature extraction. The work presented in this document falls into the latter category. More in-depth discussion of texture synthesis techniques is available in Section 2.2.

2.1.1.2 Semi-automatic and Fast Digital Inpainting

The authors of [3] chose to group the Semi-automatic Inpainting algorithm described in [6] with the Fast Digital Inpainting algorithm described in [7] into a separate category. Both algorithms share the common property that some minimal input from a human user is required to produce the results. The idea is to exploit the fact that certain image inpainting tasks are computationally expensive but easy for the human user to do and the other way around. Therefore, by splitting the image inpainting problem into computationally inexpensive parts done by the computer and user-friendly tasks executed by a human, one can derive image inpainting algorithms that work fast and reliable.

Even though [3] considers the two algorithms above as prime examples of this category, it is reasonable to generalize this category to include other human-assisted inpainting algorithms, as well. All those algorithms fall into the same framework described above with main differentiating factor being the type of information collected by the user. We turn back to [6] and [7] for examples of such information. [6] opts for an exemplar-based technique in which the user provides curves along which a common texture should be inpainted. [7] is a PDE-based solution in which the hardest part of the solution of the PDE (the boundary regions identification) is supplied by the user to reduce the computational complexity

substantially. In general, the information required by human-assisted inpainting is semantic in nature, and either describes properties of the scene segmentation or the scene texture.

2.1.1.3 PDE-based Inpainting

[2] introduces PDE-based Inpainting in the context of the related concept of diffusion. Diffusion is the process of propagating heat within physical materials. By analogy, PDE-based Inpainting tries to propagate color information from the exterior of a hole to its interior. In physics, the process has been mathematically formalised by partial differential equations and solutions are available in the form of partial-differential-equation-regularisation procedures. Those procedures are what give PDE-based methods their name. PDE-procedures are usually iterative in nature and assume image smoothness which is increased gradually over the course of the iterations. It is this smoothness assumption that prevents PDE-inpainting of large regions. Despite that, the techniques discussed in this section have been applied very successfully to small image windows.

To make PDE-based inpainting work in practice, special care must be taken to handle properly pixels, residing on an edge separating different regions in the picture. The problem is usually addressed by extracting the local geometry of the image. [2] suggests the local geometry is traditionally extracted by image-gradient-extraction algorithms. Modified versions of the partial differential equations are then used to account for this local geometry.

The Diffusion Inpainting considered up until now is a minimization procedure of a localized energy function. Alternatively, one can consider a global energy function, instead. One such function is the total variation (TV) that gives rise to another set of image inpainting methods called variational methods. TV is presented in Equation 2.1, where $\hat{\mathbf{I}}$ is an approximation of the partially known image function $\mathbf{I}$ and the notations are consistent with those in Definition 2.2. The first term in Equation 2.1 bounds the variation of the inpainted picture while the second term forces similarity between $\hat{\mathbf{I}}$ and $\mathbf{I}$. λ is a weighting factor that determines how important the terms are relative to one another. The variational techniques just discussed tend to preserve sharp edges but have problems with their connectivity. To overcome this one can consider combining PDE-based

Inpainting and Variational Inpainting.

$$J_{\mathrm{TV}}(\hat{\mathbf{I}}) = \int_{\Omega} |\nabla \hat{\mathbf{I}}(\mathbf{x})| d\mathbf{x} + \lambda \int_{K} (\mathbf{I}(\mathbf{x}) - \hat{\mathbf{I}}(\mathbf{x}))^2 d\mathbf{x} \tag{2.1}$$

2.1.1.4 Exemplar-based and Search-based Inpainting

All methods discussed up until now work by extracting some information from the known part of the damaged image. The damaged region is then generated based on that information. Exemplar methods, in contrast, try to get around the extraction process which is, in general, the hardest and most error-prone part of those algorithms and replace it with a search procedure. The search procedure assumes that enough variety in the known part of the image is presented to able to pick regions from there and simply copy them across the unknown area. The most computationally expensive part of such a technique is then the search for good example to be placed. For that reason, those methods are sometimes referred to as Search-based Inpainting. Searching algorithms apart, the performance of Exemplar inpainting is determined by the shape of the regions copied and the similarity measure, guiding their choice.

Traditionally, Search-based inpainting has been employed to generate irregularly-textured regions which are much harder to inpaint by other means. The success of the method in this setting is based upon its guarantees to preserve local texture statistics, even though it threats them as a black box. Even though local statistics are preserved with Exemplar-based Inpainting, structural information is not. That makes certain textures not well suited for that particular type of inpainting as shown in Figure 2.1. Another common issue with exemplar algorithms is the lack of a precise example needed to fill a particular hole, e.g. hole occurring near region contours.

2.1.1.5 Hybrid Inpainting

The Hybrid Inpainting category comprises of a set of algorithms trying to combine the ideas of the algorithms already listed. Several approaches have been discussed in [3] for combining the desirable properties of inpainting algorithms. One common approach is the segmentation of the image holes to structural parts and texture parts. Once such segmentation is obtained, one can run different

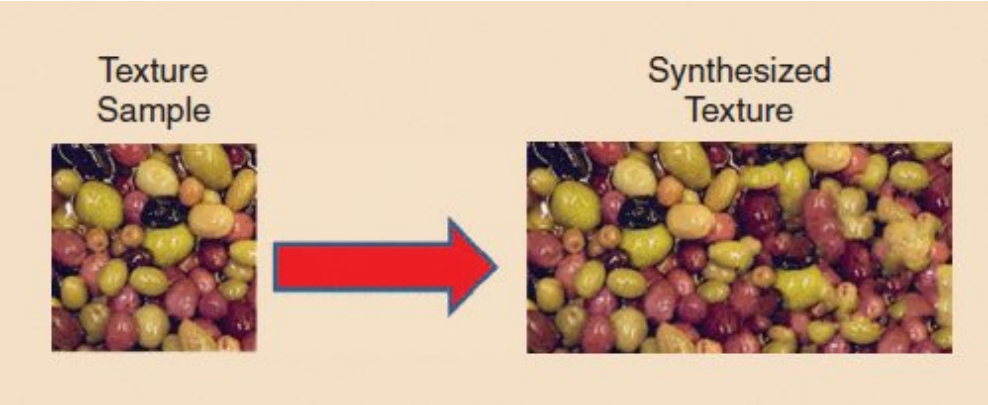

Figure 2.1: Example of Search-based Inpainting failure. The image is courtesy of [2]

algorithms chosen depending on the type of region. Another possibility, applicable for combining methods based on numerical optimization, is to introduce joint optimization problem which is based on linear combination of the individual optimization problems. Overall, Hybrid Inpainting techniques are more robust where no prior knowledge about the domain of the inpainting is available. That suggests that complete solution to the image inpainting problem is likely to be obtained by algorithms that adapt to the type of holes being inpainted.

2.1.2 Image Inpainting Applications

Image inpainting is a central issue within the computer vision field. Its applications have moved far beyond the original image restoration formulations for which it was first introduced. Nowadays one of its most prominent uses is as part of object removal algorithms which aim to remove foreground objects that partially occlude other more important objects in a picture. Another practical application of the image inpainting techniques discussed is transmission-error-recovery where parts of a video streamed across the internet are lost and need to be recovered on the fly while the video is being displayed. Last but not least, image inpainting has been successfully applied to address disocclusion. Disocclusion is a problem that occurs when one wants to enable free-roaming in 3D scenes recovered from a set of images taken from different viewpoints. It is common in such setting to find pixels that are occluded in all the available images but need to be visible from the user-specified point of view. Modified versions of the exemplar algorithms

demonstrated in this chapter can be used to recover the missing pixels and solve the disocclusion problem.

2.2 Texture Synthesis

Although in Section 2.1 we looked at texture synthesis from the perspective of the image inpainting problem, texture synthesis applications are not limited to computer vision tasks. Texture synthesis has important applications in computer graphics, as well, where another type of texture synthesis is prevalent called unconstrained texture synthesis. Unconstrained texture synthesis, first introduced in [8], unlike the constrained texture synthesis we looked at in Section 2.1.1.1, doesn't limit the size of the inpainted region to a hole in an image. Instead, it tries to generate arbitrary sized texture from a small sample called a seed. The constrained and unconstrained texture syntheses share a lot of similar properties, but as we will explore in the next section, there are some differences, as well.

2.2.1 Constrained and Unconstrained Texture Syntheses

Both the constrained and unconstrained texture syntheses domains require the results produced to look natural from a human perception point of view. What affects that natural view, however is different. In the case of unconstrained texture synthesis, the essential problem is being able to generate more from the same texture without making it look artificially repetitive. That requires the synthesis algorithms to put in place certain randomisation procedures. On the other hand, in constrained synthesis an important consideration is making the generated texture transition, as smoothly as possible, from one end of the gap to the other without leaving any visual artefacts, such as discontinuities. That usually seriously limits the family of acceptable patches that can be constructed by constrained texture synthesis algorithms but also supplies them with additional bits of information that can improve the generated results. Despite those very contrasting requirements posed by the domains, human texture perception imposes another more general set of requirements applicable to all texture synthesis algorithms that tie the two problems together. We explore those requirements in the next section.

2.2.2 Human Perception as a Measure of Texture Quality

A classic paper on pre-attentive texture discrimination written by one of the pioneers of the field, Béla Julesz, back in 1973 suggests that human texture discrimination is based on detecting changes in first and second-order texture statistics [9]. In the paper, k-ordered texture statistics are defined as the joint probability $P(x_1, ..., x_k)$ of any k pixels' luminosity values. Refinements of his work were later introduced which revised the assumption that only first and second-order statistics are required and instead conjectured that depending on the texture more statistic orders may be necessary. In essence, the conjecture captures formally that textural perception is based a great deal on joint probabilities of neighboring pixels in the texture. The paper demonstrates the validity of the conjecture by introducing texture pairs sharing common statistics that when put side-by-side look indistinguishable. Those experiments are very relevant to texture synthesis because they show that if one can provide an algorithm for texture synthesis that preserves texture statistics the results are likely to be perceptually pleasing.

Another, more contemporary view of pre-attentive texture discrimination, suggests that the brain performs some initial preprocessing on the visual input before extracting the statistical information discovered by Julesz [10]. In particular, various convolution-like filters firing to different primitive shapes and orientations are employed by the brain as initial disambiguation step. That imposes additional requirements on successful texture-generating methods to retain some structural consistency across the texture.

2.2.3 Previous Work on Texture Synthesis

In the literature, exemplar-based synthesis is the predominant technique used for both the constrained and unconstrained cases of texture synthesis. The main reason for its spread use is its computational inexpensiveness which makes it a good candidate for texture synthesis on the fly. The results produced by exemplar methods are relatively good, as well. [11] ties this back to our discussion in Section 2.2.2 by showing that the sampling procedure used by exemplar texture synthesis guarantees to preserve first and second order statistics in local windows. Since many textures experience Markovian and stationary properties on their first

and second-order statistics, preserving the window statistics in many cases preserves them across the texture, as well. Unfortunately, preserving texture structure is not guaranteed by exemplar-based methods. Another problem with them relevant to the image inpainting setting is that they not always reproduce natural variances in the texture caused by illumination and deformation changes. Those often occur in inpainting applications because the textures are usually situated in a real-world environment where such variations almost inevitably occur.

Before exemplar methods gained their current popularity, another set of techniques for texture synthesis called Markov Random Fields(MRF) were considered by the computer graphics community [8]. Back in the days, their computational cost turned to be too prohibitive to make them flexible enough to generate complicated textures. With the recent computational and theoretical advances and, in particular, the widespread use of special MRFs called Boltzmann machines, however, those techniques are becoming relevant again. Markov Random Fields are a special type of stochastic processes in which each random variable depends only on a small subset of the other random variables called its Markov blanket. To use MRFs for texture synthesis, one models each pixel luminosity as a random variable and makes it dependent on its immediate neighbours. The reason MRFs are suitable for texture synthesis is similar to the argument presented for exemplar methods. In particular, MRFs can learn the joint distribution of neighbouring pixels colours and then used it for sampling textures. Conditioning on previously known pixel colours gives MRFs the ability to be used for constrained texture synthesis as well. Examples of successful uses of MRFs for texture synthesis in recent history are [12] and [1].

2.3 Gaussian Processes Background

2.3.1 Definition

The Gaussian processes definition given in [13] is the following:

Definition 2.5. *A collection of random variables, any finite number of which obeys a joint Gaussian distribution, is called a Gaussian process.*

Any Gaussian process $f(x)$ is fully specified by a mean function $m(x)$ and a covariance (sometimes called kernel) function $k(x, x')$ that obey the following equations:

$$m(x) = \mathbb{E}[f(x)], \tag{2.2}$$

$$k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]. \tag{2.3}$$

The mathematical notation used throughout this document to denote that $f(x)$ is a Gaussian process is $f(x) \sim \mathcal{GP}(m(x), k(x, x'))$.

2.3.2 Regression

Commonly, Gaussian Processes model unknown but partially observed functions $y(x)$ as being generated by underlying unknown function $f(x)$ with added independent Gaussian observation noise $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$ [13, Section 2.2]. Given a training set of input locations $X = \begin{bmatrix} x^1 \\ x^2 \\ \vdots \\ x^n \end{bmatrix}$ and output values $y = \begin{bmatrix} y(x^1) \\ y(x^2) \\ \vdots \\ y(x^n) \end{bmatrix}$ and analogously annotated test set $X_* = \begin{bmatrix} x_*^1 \\ x_*^2 \\ \vdots \\ x_*^m \end{bmatrix}$, $y_* = \begin{bmatrix} y_*(x_*^1) \\ y_*(x_*^2) \\ \vdots \\ y_*(x_*^m) \end{bmatrix}$ that translates to:

$$y = f(X) + \epsilon \text{ and} \tag{2.4}$$

$$y_* = f(X_*) + \epsilon_* \ , \tag{2.5}$$

with $\epsilon = \begin{bmatrix} \epsilon^1 \\ \epsilon^2 \\ \vdots \\ \epsilon^n \end{bmatrix}$ and $\epsilon_* = \begin{bmatrix} \epsilon_*^1 \\ \epsilon_*^2 \\ \vdots \\ \epsilon_*^m \end{bmatrix}$ being the added noise. The regression problem then becomes equivalent to calculating $\mathbb{P}(y_*|X, X_*, y)$. The only latent parameter of this model is the unknown function $f(x)$ which is represented by the vectors $f(X)$ and $f(X_*)$. A Gaussian Process prior is imposed on $f(x)$, such that $f(x) \sim \mathcal{GP}(m(x), k(x, x'))$ which by definition implies:

$$\begin{bmatrix} f(X) \\ f(X_*) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} k(X, X) & k(X, X_*) \\ k(X_*, X) & k(X_*, X_*) \end{bmatrix}\right) \ . \tag{2.6}$$

Using the fact the joint distribution of any number of i.i.d Gaussians is multivariate Gaussian one can obtain:

$$\begin{bmatrix} \epsilon \\ \epsilon_* \end{bmatrix} \sim \mathcal{N}(0, \sigma_n^2 I) \quad . \tag{2.7}$$

Plugging back 2.4, 2.5 and 2.7 in 2.6 and taking into account the fact that sum of independent Gaussian distributions is a Gaussian results in:

$$\begin{bmatrix} y \\ y_* \end{bmatrix} = \begin{bmatrix} f(X) \\ f(X_*) \end{bmatrix} + \begin{bmatrix} \epsilon \\ \epsilon_* \end{bmatrix} \sim \mathcal{N}(\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} k(X,X) + \sigma_n^2 I & k(X,X_*) \\ k(X_*,X) & k(X_*,X_*) + \sigma_n^2 I \end{bmatrix}) \quad . \tag{2.8}$$

Imposing the additional assumption that $m(x)$ is a constant function, which is equivalent to assuming that prior to any observations the input locations are indistinguishable from one another, can greatly simplify calculations. In particular, $y(x)$ can be modified by subtracting the mean value of y to obtain new mean vector $\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix} = 0$. Therefore, throughout the calculations to follow:

$$\begin{bmatrix} y \\ y_* \end{bmatrix} \sim \mathcal{N}(0, \begin{bmatrix} k(X,X) + \sigma_n^2 I & k(X,X_*) \\ k(X_*,X) & k(X_*,X_*) + \sigma_n^2 I \end{bmatrix}) \quad . \tag{2.9}$$

Using the fact that the conditional of jointly Gaussian random variables in our case y and y_* is a Gaussian [14, Section 9.3], one can derive:

$$y_* | y, X, X_* \sim \mathcal{N}(\mu_*, \Sigma_*), \tag{2.10}$$

where

$$\mu_* = k(X_*, X)(k(X,X) + \sigma_n^2 I)^{-1} y \quad \text{and} \tag{2.11}$$

$$\Sigma_* = k(X_*, X_*) + \sigma_n^2 I - k(X_*, X)(k(X,X) + \sigma_n^2 I)^{-1} k(X_*, X)^T. \tag{2.12}$$

Interesting implication of 2.11 is that the predictions given by the Gaussian Process are simply linear combination of the values of the function at the different input locations.

2.3.3 Hyperparameters Learning

The derivations demonstrated in Section 2.3.2, assumed that the kernel function k used to define f's prior is given. In Section 2.3.4 we shall explore how one goes

about choosing it. We note here, however, that all practical covariance functions k are parametrized by a set of hyperparameters that need to be learned from the data to fit k to the specific problem in hand. Usually, those parameters control some properties of the function f, enforced by the kernel function of our choice. Examples of such properties include smoothness and periodicity.

As demonstrated in Section 2.3.2, the calculations involved in Gaussian Processes' derivations mostly boil down to equations of Gaussian distributions. That allows, amongst other things, analytical solutions to the integrals involved in Bayesian inference. In the context of Bayesian inference, [15] defines marginal likelihood as follows:

Definition 2.6. *Marginal likelihood refers to the probability of observing the data given a selected model. It takes into account every possible selection of hyperparameters of the model weighted by their prior probability.*

Combining Definition 2.6 with the results of Section 2.3.2 (in particular using Equation 2.8 to solve the integral below) one can obtain:

$$\mathbb{P}(y|X,\theta) = \int_f \mathbb{P}(y,f|X,\theta)df = \int_f \mathbb{P}(y|f,X,\theta)\mathbb{P}(f|X,\theta)df = \mathcal{N}(0, k(X,X,\theta) + \sigma_n^2 I), \tag{2.13}$$

where θ are the hyperparameters of the model used. Taking the logarithm of Equation 2.13 one can obtain the marginal log likelihood:

$$\log(\mathbb{P}(y|X,\theta)) = -\frac{1}{2}y^T(k(X,X,\theta)+\sigma_n^2 I)^{-1}y - \frac{1}{2}\log|k(X,X,\theta)+\sigma_n^2 I| - \frac{n}{2}\log 2\pi \quad . \tag{2.14}$$

Equation 2.14 can be viewed as a function of the model hyperparameters θ and optimisation problem can be posed to find θ_* that maximizes Equation 2.13 or alternately its logarithm. Equation 2.14 can be interpreted as a combination a model fit term $-\frac{1}{2}y^T(k(X,X,\theta) + \sigma_n^2 I)^{-1}y$ that specifies how well the model describes the training data and a complexity penalty term $-\frac{1}{2}\log|k(X,X,\theta) + \sigma_n^2 I|$ that makes sure the model does not overfit. Equation 2.14 thus enforces the Occam's razor principle "plurality should not be assumed without necessity" [13, Section 5.2].

Gaussian Processes commonly choose to solve the optimization problem posed above using standard gradient descent for which the calculation of the derivatives

of Equation 2.14 with respect to each hyperparameter θ_i is required. Using matrix differentiation one can derive:

$$\frac{\partial \log(\mathbb{P}(Y|X,\theta))}{\partial \theta_i} = \frac{1}{2}\alpha^T \frac{\partial k(X,X,\theta)}{\partial \theta_i}\alpha - \frac{1}{2}tr((k(X,X,\theta)+\sigma_n^2 I)^{-1}\frac{\partial k(X,X,\theta)}{\partial \theta_i}) \ , \tag{2.15}$$

where $\alpha = (k(X,X,\theta)+\sigma_n^2 I)^{-1}y$. In the work presented in this document, BFGS is used instead to optimize Equation 2.14. BFGS is a Quasi-Newton method and as such it is assuming that the function being optimized is two-times-differentiable but uses only first order derivatives to optimise Equation 2.14. In particular, it only uses Equation 2.15. A full derivation of the BFGS algorithm is given in Appendix B.

2.3.4 Covariance Functions

Section 2.3.2 demonstrated that to specify fully a Gaussian process one needs to specify a covariance function $k(x, x')$ that shows how the function values at x and x' relate. Therefore, the choice of a covariance function defines the properties one expects from the types of functions generated by the Gaussian process. Crafting a covariance function that enforces all the properties imposed by the problem in hand is thus a common approach when using Gaussian processes for regression. In Section 2.3.4.4, the Spectral Mixture Product covariance function is introduced, which adopts an entirely different approach to the problem which tries to create covariance function general enough to be able to mimic any other covariance function. Comparison between the approaches is discussed in Section 2.3.4.3.

2.3.4.1 Standard Covariance Functions

A class of covariance functions with significant practical applications are the stationary covariance functions. [13, Chapter 4] defines the stationary property as follows:

Definition 2.7. *A covariance function $k(x, x')$ is called stationary if it only depends on $\tau = x - x'$. For notational convenience one usually writes k as a function of a single argument $k(\tau)$.*

Stationary covariance functions are very useful since they incorporate translational invariance into the functions generated by a Gaussian Process. This document will only deal with stationary covariance functions. For other types of covariance functions, one should consult [13, Chapter 4]. Commonly used stationary covariance functions together with their properties are to follow, for more thorough discussion check [13, Chapter 4].

Squared Exponential

The Squared Exponential covariance function is given by Equation 2.16. The Squared Exponential kernel is the most commonly used Gaussian Process regression kernel. It assumes positive correlation between any two input points which decays exponentially fast with the distance between them. l called the length-scale of the Square Exponential and determines how fast the exponential decays, playing a similar role to the standard deviation of a Gaussian probability distribution. Functions generated with the Squared Exponential kernels are very smooth [13, Chapter 4].

$$k_{SE}(\tau) = e^{-\frac{\|\tau\|^2}{2l^2}} \tag{2.16}$$

Matern

The Matern covariance function is given by:

$$k_{MA}(\tau) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}\tau}{l} \right)^{\nu} K_{\nu} \left(\frac{\sqrt{2\nu}\tau}{l} \right) , \tag{2.17}$$

where Γ is the gamma function, K_{ν} is the modified Bessel function of the second kind and l and ν are the positive parameters of the function. Similarly to the Squared Exponential covariance function, the Matern covariance function assumes positive correlation between any two input points. The Matern covariance function is equal to the Squared Exponential in the limit $\nu \to \infty$. Functions generated by the Matern Kernel are less smooth [13, Chapter 4].

Rational Quadratic

The Rational Quadratic covariance function is given by Equation 2.18. The Rational Quadratic covariance function can be viewed as an infinite sum of Squared Exponential functions of different length scales as shown in Equation 2.19. Similarly to the other kernels in this section all points are assumed

to be positively correlated [13, Chapter 4].

$$k_{RQ}(\tau) = \left(1 + \frac{\tau^2}{2\alpha l^2}\right)^{-\alpha} \tag{2.18}$$

$$k_{RQ}(\tau) = \int \Gamma(r|\alpha, \beta) k_{SE}(\tau|r) dr \text{ , where } \beta = \frac{1}{l^2} \tag{2.19}$$

2.3.4.2 Combining Covariance Functions

In Section 2.3.4.1, standard covariance functions were listed. However, one does not usually model real world problems using a single covariance function. Instead, different covariance functions are usually combined using one of the following properties originally listed in [13, Chapter 4]:

Property 2.1. *The product $k_1(x, x')k_2(x, x')$ of two kernels functions $k_1(x, x')$ and $k_2(x, x')$ is a valid kernel function.*

Property 2.2. *The sum $k_1(x, x') + k_2(x, x')$ of two kernels functions $k_1(x, x')$ and $k_2(x, x')$ is a valid kernel function.*

Property 2.3. *The product $a(x)k(x, x')a(x')$ is a valid kernel function for any valid kernel $k(x, x')$ and any function $a(x)$.*

Property 2.4. *Given two covariance functions $k_1(x_1, x_1') : \mathcal{X}_1 \bigtimes \mathcal{X}_1 \to \mathbb{R}$ and $k_2(x_2, x_2') : \mathcal{X}_2 \bigtimes \mathcal{X}_2 \to \mathbb{R}$, the direct sum $k(x, x') = k_1(x_1, x_1') + k_2(x_2, x_2')$ is a valid kernel function.*

Property 2.5. *Given two covariance functions $k_1(x_1, x_1') : \mathcal{X}_1 \bigtimes \mathcal{X}_1 \to \mathbb{R}$ and $k_2(x_2, x_2') : \mathcal{X}_2 \bigtimes \mathcal{X}_2 \to \mathbb{R}$, the tensor product $k(x, x') = k_1(x_1, x_1')k_2(x_2, x_2')$ is a valid kernel function.*

Theorem 2.1. *(Bochner) A complex-valued function k on $\mathbb{R}^D$ is the covariance function of a weakly-stationary mean-square-continuous complex-valued random process on $\mathbb{R}^D$ if and only if it can be represented as*

$$k(\tau) = \int_{\mathbb{R}^D} e^{2\pi i s\tau} d\mu(s) \text{ ,} \tag{2.20}$$

where μ is a positive finite measure.

If μ has a density $S(s)$, S is called spectral density or power spectrum of k, and they are Fourier duals [16]:

$$S(s) = \int k(\tau) e^{-2\pi i s\tau} d\tau \text{ ,} \tag{2.21}$$

$$k(\tau) = \int S(s) e^{2\pi i s\tau} ds \text{ .} \tag{2.22}$$

Theorem 2.1 can be used in the reversed direction yelling new stationary covariance functions from a given power spectrum as it is demonstrated in Section 2.3.4.4.

2.3.4.3 The Traditional Approach's Shortcomings

Up to this point, this document considered, although somewhat briefly, methods for fitting manually crafted kernels to a given problem. That process is not trivial and requires domain-specific knowledge of the problem preventing Gaussian Process regression that we looked at so far to generalize to arbitrary problems where no prior knowledge is available. Not only that but it limits its usability to functions which can only be modelled by a person looking at example data. In the vast majority of real-life scenarios, it is the case that a person is overwhelmed by the complexity of the problem and tries to resort to the computer for help, not the other way around. It is thus, unreasonable and prone to errors to craft kernels manually all the time. To quote [13, Section 5.4]:

> One could ask why it is interesting to discuss this scenario[specifying kernels that do not align well with the problem], since one should surely simply avoid choosing such a model in practice. While this is true in theory, for practical reasons such as the convenience of using standard forms for the covariance function or because vague prior information, one inevitably ends up in a situation which resembles some level of mis-specification.

[13, Section 5.4] and [17] demonstrate the results of this kind of kernel misspecification. In particular, they show that such scenarios lead to poor results usually due to the model describing all the data as noise because of the lack of more likely explanation. [17] suggests a different approach to the problem. The usage of general enough covariance function able to model whole families of covariance functions is considered as a way of mitigating the problems above. In this scenario, the machine learning algorithm is picking up properties of the data in the process of marginal likelihood optimization, which then can be read off the hyperparameters and verified. For this approach to work, one needs to come up with an expressive enough covariance function. [17] introduces one such function, the Spectral Mixture Product (SMP) kernel function, that can model any other stationary kernel function given a sufficient number of basis functions. We look more closely into it in the next section.

2.3.4.4 The Spectral Mixture Product Kernel

This section will motivate and derive the SMP kernel in terms of its power spectrum and the power spectra of other common stationary covariance functions, in particular, the ones discussed in Section 2.3.4.1. The power spectra of the Squared Exponential kernels, the Squared Exponential Mixture kernels and the Matern kernels are shown in Equation 2.23, Equation 2.24 and 2.25, respectively.

$$S_{SE}(s) = \mathcal{N}(0, \frac{1}{\pi l}) \tag{2.23}$$

$$S_{SEM}(s) = \sum_a \omega_a \mathcal{N}(0, \frac{1}{\pi l_a}) \tag{2.24}$$

$$S_{MA}(s) = \mathbb{P}(s|2\nu, 0, \frac{1}{2\pi l}) \ , \tag{2.25}$$

where $\mathbb{P}(s|2\nu, 0, \frac{1}{2\pi l})$ is Student's t-distribution with 2ν degrees of freedom, mean 0 and standard deviation $\frac{1}{2\pi l}$. The power spectra of those covariance functions look like a scale mixture of Gaussian functions centred around 0. A reasonable generalization is to consider a kernel that incorporates a scale-location mixture of Gaussians, instead. That is precisely the way the 1-D Spectral Mixture kernel covariance is derived to obtain Equation 2.26. For full derivation, one should consider the supplementary material of [17].

$$k_{SM}(\tau|w, \mu, \sigma) = \int_{-\infty}^{\infty} e^{2\pi i s\tau} \sum_a w_a^2 e^{-\frac{(s-\mu_a)^2}{2\sigma_a^2}} ds = \sum_a w_a^2 e^{-2\pi^2\tau^2\sigma_a^2} \cos(2\pi\tau\mu_a) \tag{2.26}$$

In Section 2.3.5.1, we argue that many popular kernel functions generalize to multidimensional input by the means of the tensor product given in Equation 2.27. Significant computational gains for computing kernel matrices based on such tensor-product covariance functions are also demonstrated when a certain structure in the data is in place. It is, thus sensible to follow the same model to generalize our Spectral Mixture kernel to obtain Equation 2.28.

$$k(x, x') = \prod_{d=1}^{D} k_d(x_d, x'_d) \tag{2.27}$$

$$k_{SMP}(\tau|w, \mu, \sigma) = \prod_{d=1}^{D} k_{SM}^d(\tau|w_d, \mu_d, \sigma_d) \tag{2.28}$$

[17] argues that the SMP covariance function presented in Equation 2.28 can model any tensor-product stationary covariance function and furthermore it does it in quite a succinct way using very few Gaussian basis functions. The Spectral Mixture Product kernel is, thus able to model very complicated functions with only but few hundreds of hyperparameters without using any problem-specific knowledge. That motivates us to use it to model the complex problem of texture reconstruction.

2.3.5 Speeding up Gaussian Processes

In the preceding sections, we considered how one goes about modelling regression problems with Gaussian processes and how missing data is predicted in this setting. The framework presented is general enough to model any regression problem and is particularly suitable for texture synthesis as we discuss in Section 2.3.6. However to model constrained texture synthesis as GP, each known pixel's position must be associated with an input location in the X vector. That poses a problem because the matrix $(k(X,X)+\sigma_n^2 I)^{-1}$ used in Equation 2.11 and Equation 2.15 needs to be constructed for the inference procedures discussed to work. The generation of $(k(X,X)+\sigma_n^2 I)^{-1}$ involves the calculation of the Cholesky factorization of $k(X,X)+\sigma_n^2 I$, which requires $\mathcal{O}(n^3)$ computations and $\mathcal{O}(n^2)$ storage space, where n is the number of known pixels in the texture [13, Section 2.2]. Even for small texture patches, the computational cost of the inference can become prohibitive as the following example demonstrates. Let's assume rectangular texture patch 100×100 with a 50×50 hole carved into it. It contains 7500 known pixel values which incur approximately 3 minutes penalty per Cholesky factorization on modern PC with 2GHz processor. Taking into consideration that this cost will be induced for each iteration of the optimization procedure used to optimise the marginal likelihood we deduce that to inpaint that small region would require hours to compute. In this section, a fast and exact inference algorithm is demonstrated that reduces the computation to just a few minutes. The rest of the section is based on [18, Chapter 5] which first introduced the algorithm.

2.3.5.1 `GPR_GRID` Requirements

The `GPR_GRID` algorithm first presented in [18, Chapter 5] assumes a certain structure in the Gaussian Process to reduce the computational cost of the inference presented in Section 2.3.2 and the hyperparameter optimization presented in Section 2.3.3. `GPR_GRID` makes two assumptions. First, the D-dimensional training input points X must lie on a complete D-dimensional Cartesian grid. In particular, in mathematical notation one writes that as $X = X^1 \times X^2 \ldots \times X^D$, where $X^i \subseteq \mathbb{R}$ with $|X^i| = n_i < \infty$ for all dimensions i. It is worth noting that enforces the size of X to be $\left(\prod_{d=1}^{D} n_d\right) \times D$. The grid requirement is a good fit for applying GPs to pictures because such grid is already available. For image inpainting in particular, however, the requirement the grid to be complete is a tricky one since X cannot contain the pixel locations to be inpainted. We will relax this assumption later in the section. The other requirement is the covariance function used with the GP to be tensor product kernel. [18, Chapter 5] defines tensor product kernels as follows:

Definition 2.8. *Covariance function is called tensor product covariance function if can be decomposed into a product of one-dimensional covariance functions. In mathematical notation, that translates to:*

$$k(x, x') = \prod_{d=1}^{D} k_d(x_d, x'_d) \ . \tag{2.29}$$

The decomposition condition posed by Definition 2.8 does not limit greatly the choice of covariance functions one can use. [18, Chapter 5] shows that many popular covariance functions and, in particular, all the covariance functions listed in Section 2.3.4.1, are tensor product kernels. Furthermore, it suggests that many kernel functions are generalized to a multidimensional setting based on Property 2.5 discussed in Section 2.3.4.2 which makes them tensor product kernels by construction. For the purpose of this document, the SMP kernel will be used which falls into the latter category.

2.3.5.2 Derivation of the `Kron_mvprod` Subroutine

To derive the `GPR_GRID` algorithm, we will first derive the `kron_mvprod` subroutine on which it is is based. The `kron_mvprod` subroutine calculates $(\bigotimes_{d=1}^{D} A_d)b$ for any vector b and any set of square matrices A_i of size $n_i \times n_i$ in $\mathcal{O}\left(N \sum_{d=1}^{D} n_i\right)$ time and $\mathcal{O}\left(N + \sum_{d=1}^{D} n_i^2\right)$ space, where N is the size of b. Here the $\bigotimes$ symbol denotes the Kronecker product of the matrices A_i. The Kronecker product of two matrices is a standard operation that is defined by [18, Chapter 5] as follows:

Definition 2.9. *The Kronecker product of a $m \times n$ matrix A with a $p \times q$ matrix B, denoted by $A \bigotimes B$, is given by the formula:*

$$A \bigotimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix} . \tag{2.30}$$

Much literature has been developed to demonstrate different ways of implementing a `kron_mvprod` subroutine with the required runtime requirements e.g. [19],[18, Chapter 5] and [20]. In the lines to follow, the version described in [20] is presented because it is the most concise to derive.

One can use Corollary A.2 to rewrite $(\bigotimes_{d=1}^{D} A_d)b$ as the matrix product of the form:

$$(\bigotimes_{d=1}^{D} A_d)b = (\prod_{d=1}^{D} I_{N_d} \bigotimes A_d \bigotimes I_{\overline{N_d}})b = (\prod_{d=1}^{D} B_d)b = (B_1 \dots (B_{D-1}(B_D b))) , \tag{2.31}$$

where B_d denotes the matrix $I_{N_d} \bigotimes A_d \bigotimes I_{\overline{N_d}}$. One can rewrite the product $B_D b$ using Property A.8 as follows:

$$\begin{aligned} B_D b &= (I_{N_D} \bigotimes A_D \bigotimes I_{\overline{N_D}})b = ((I_{N_D} \bigotimes A_D) \bigotimes I_{\overline{N_D}})vec(B) = \\ &= vec(I_{\overline{N_D}} B (I_{N_D} \bigotimes A_D)^T) = vec(B(I_{N_D} \bigotimes A_D)^T) = vec(B(I_{N_D} \bigotimes A_D^T)) , \end{aligned} \tag{2.32}$$

where $B = reshape(b, \overline{N_D}, n_D N_D))$. Here the $reshape(M, r, s)$ operator, acting on a matrix M of size $p \times q$, is defined to return a matrix of size $r \times s$ whose elements are the elements of M taken column-wise ($pq = rs$) and the $vec(M)$ operator is simply defined as $reshape(M, pq, 1)$. The matrix $(I_{N_D} \bigotimes A_D^T)$, as noted by Property A.9, is block diagonal which means that each column in the

matrix has exactly n_D non-zero entries. Therefore, to calculate each entry in $vec(B(I_{N_D} \bigotimes A_D^T))$ with a sparse multiplication procedure takes $\mathcal{O}(n_D)$. Since $vec(B(I_{N_D} \bigotimes A_D^T))$ is a vector of size N, the overall computational steps required to calculate $B_D b$ is $\mathcal{O}(n_D N)$. Note that the matrix $I_{N_D} \bigotimes A_D^T$ does not need to be explicitly created for the computations to be carried out. Therefore, the memory requirement on the algorithm are $\mathcal{O}\left(N + \sum_{d=1}^{D} n_i^2\right)$. Since $vec(B(I_{N_D} \bigotimes A_D^T))$ is a vector of the same size as b, the procedure can be chained to give the Algorithm 1.

Algorithm 1 The `kron_mvprod` subroutine

1: **function** KRON_MVPROD($A_1, A_2, \ldots, A_D, b$)
2: **for** $i = D, \ldots, 1$ **do**
3: $B = reshape(b, \overline{N_i}, n_i N_i)$
4: $b = vec(B(I_{N_D} \bigotimes A_D^T))$ ▷ Requires effiecient implementation
5: **end for**
6: **return** b
7: **end function**

2.3.5.3 Derivation of the `GPR_GRID` Algorithm

It is trivial to show that the assumptions made in Section 2.3.5.1 allow rewriting the matrix $k(X, X)$ as the Kronecker product $\bigotimes_{d=1}^{D} K_d$, where D is the dimensionality of our training data stored in X. In the case of the texture synthesis, we are dealing with $D = 2$. One can use the eigendecomposition $Q_d \Lambda_d Q_d^T$ of the K_d matrices to compute $(k(X, X) + \sigma_n^2 I)^{-1} y$ efficiently, as demonstrated below:

$$\begin{aligned}(k(X,X) + \sigma_n^2 I)^{-1} y &= (Q \Lambda Q^T + \sigma_n^2 I)^{-1} y = Q(\Lambda + \sigma_n^2 I)^{-1} Q^T y = \\ &= (\bigotimes_{d=1}^{D} Q_d)((\Lambda + \sigma_n^2 I)^{-1} (\bigotimes_{d=1}^{D} Q_d^T) y) = \\ &= \texttt{kron\_mvprod}([Q_d], (\Lambda + \sigma_n^2 I)^{-1} \texttt{kron\_mvprod}([Q_d^T], y)) \ ,\end{aligned} \tag{2.33}$$

where $Q \Lambda Q^T$ is the eigendecomposition of $k(X, X)$. Λ can be efficiently computed because the Kronecker product of diagonal matrices is diagonal. The product $(\Lambda + \sigma_n^2 I)^{-1} \texttt{kron\_mvprod}([Q_d^T], y)$ is efficiently computable in $\mathcal{O}(N)$ time by a sparse multiplication procedure, where N is the number of elements in y. Therefore, the

overall computational cost of $(k(X,X)+\sigma_n^2 I)^{-1}y$ is $\mathcal{O}\left(\sum_{d=1}^{D} n_d^3 + N\sum_{d=1}^{D} n_d\right)$, where the $\sum_{d=1}^{D} n_d^3$ factor comes from the computation of the eigendecompositions of the K_d. The space requirements for the computation are still $\mathcal{O}\left(N+\sum_{d=1}^{D} n_i^2\right)$.

Efficient computation of the term $\log|k(X,X,\theta)+\sigma_n^2 I|$ is also available in this setting as demonstrated by the following calculations:

$$\log|k(X,X,\theta)+\sigma_n^2 I| = \log|Q\Lambda Q^T+\sigma_n^2 I| = \sum_{i=1}^{|X|}\log(\lambda_i+\sigma_n^2)\ , \tag{2.34}$$

where λ_i is the i^{th} diagonal entry in Λ. Equation 2.34 has a computational complexity of $\mathcal{O}(N)$ and space complexity of $\mathcal{O}\left(N+\sum_{d=1}^{D} n_i^2\right)$, provided that Λ is already computed. Based on Equation 2.33 and Equation 2.34 efficient procedure with computational and space requirements similar to those of Equation 2.33 for calculating the log marginal likelihood of the GP is trivial to derive. To derive the efficient algorithm discussed, however, efficient computation of its derivative is required, as well. We turn to this issue next.

To calculate the derivative of the log likelihood efficiently, one needs first to derive how to efficiently calculate the derivative $\frac{\partial k(X,X)}{\partial\theta_i}$. The general formula for the derivative of Kronecker product is given in Property A.12. In most practical applications, however, tensor product kernels are structured in a way in which the hyperparameters affecting one dimension do not affect the other. That, in particular, is the case for kernels generalized to multiple dimensions using Property 2.5 and the standard kernels discussed in Section 2.3.4.1. For those kernels, Property A.12 can be further simplified to:

$$\frac{\partial(\bigotimes_{d=1}^{D} K_d)}{\partial\theta_i} = (\bigotimes_{m=1}^{j-1} K_m)\bigotimes\frac{\partial K_j}{\partial\theta_i}\bigotimes(\bigotimes_{n=j+1}^{D} K_n) = (\bigotimes_{d=1}^{D} K_d^*)\ , \tag{2.35}$$

where θ_i occurs only in K_j and $K_d^* = \begin{cases}\frac{\partial K_d}{\partial\theta_i} & \text{if } d=j\\ K_d & \text{if } d\neq j\end{cases}$. Calculating the first term of the derivative $\frac{1}{2}\alpha^T\frac{\partial k(X,X,\theta)}{\partial\theta_i}\alpha$ efficiently is a simple application of the `kron_mvprod` subroutine for the Kronecker product in Equation 2.35. We pro-

ceed to derive fast computation for the second term:

$$\begin{aligned} tr\left((k(X,X)+\sigma_n^2 I)^{-1}\frac{\partial k(X,X)}{\partial \theta_i}\right) &= tr\left((Q\Lambda Q^T+\sigma_n^2 I)^{-1}\frac{\partial k(X,X)}{\partial \theta_i}\right) \\ &= tr\left(Q(\Lambda+\sigma_n^2 I)^{-1}Q^T\frac{\partial k(X,X)}{\partial \theta_i}\right) \\ &= tr\left((\Lambda+\sigma_n^2 I)^{-1}Q^T\frac{\partial k(X,X)}{\partial \theta_i}Q\right) \\ &= diag((\Lambda+\sigma_n^2 I)^{-1})^T diag(Q^T\frac{\partial k(X,X)}{\partial \theta_i}Q)\ , \end{aligned} \tag{2.36}$$

where the last line is derived using the standard trace property $tr(X^TY) = \sum_{i,j} X_{ij}Y_{ij}$. The second part of Equation 2.36 can be computed efficiently based on Corollary A.3 and Property A.13,as follows:

$$\begin{aligned} diag(Q^T\frac{\partial k(X,X)}{\partial \theta_i}Q) &= diag((\bigotimes_{d=1}^{D} Q_d^T)(\bigotimes_{d=1}^{D} K_d^*)(\bigotimes_{d=1}^{D} Q_d^T)) = diag(\bigotimes_{d=1}^{D} Q_d K_d^* Q_d) \\ &= \bigotimes_{d=1}^{D} diag(Q_d K_d^* Q_d) \end{aligned} \tag{2.37}$$

in $\mathcal{O}\left(DN+\sum_{d=1}^{D} n_d^2\right)$ computational steps and $\mathcal{O}\left(\sum_{d=1}^{D} n_d^2\right)$ space. With those computations, we are ready to present the complete `GPR_GRID` algorithm that works in $\mathcal{O}\left(\sum_{d=1}^{D} n_d^3+\sum_{d=1}^{D} n_d N\right)$ time and $\mathcal{O}\left(\sum_{d=1}^{D} n_d^2+N\right)$ storage. The algorithm is presented in Algorithm 2.

2.3.5.4 Accounting for Missing Observations

[21] presents an extension to the `GPR_GRID` algorithm that retains its exact inference properties for data residing on nearly-complete grids. The extension allows us to perform GP inference in our constrained texture synthesis setting that takes minutes opposed to hours to compute.

[21] extends the algorithms based on an augmentation procedure that adds additional imaginary observations y_W with infinite variance ϵ^{-1}, where $\epsilon \to \infty$, to the data to complete the grid. The resulting prior GP process is:

$$y \sim \mathcal{GP}(0, \begin{bmatrix} k(X,X)+\sigma_n^2 I & k(X_w,X) \\ k(X,X_w) & k(X_w,X_w)+\epsilon^{-1}I \end{bmatrix})\ , \tag{2.38}$$

Algorithm 2 The GPR_GRID algorithm

function GPR_GRID($K_1, K_2, \ldots, K_D, y, K(X_*, X), K(X_*, X_*), \theta$)
  for $d \leftarrow 1$ **to** D **do**
    $[Q_d, \Lambda_d] \leftarrow eig(K_d)$
  end for
  $\Lambda = \bigotimes_{d=1}^{D} diag(\Lambda_d)$
  $\alpha \leftarrow kron\_mvprod(Q_1^T, \ldots, Q_D^T, y)$
  $\alpha \leftarrow (\Lambda + \sigma_n^2)^{-1}\alpha$
  $\alpha \leftarrow kron\_mvprod(Q_1, \ldots, Q_D, \alpha)$
  $\log Z(\theta) \leftarrow -\frac{1}{2}(y^T\alpha + \sum_{i=1}^{N}(\Lambda(i) + \sigma_n^2) + \frac{N}{2}log(2\pi))$ ▷ Log likelihood
  for $i \leftarrow 1$ **to** $length(\theta)$ **do** ▷ Derivatives
    for $d \leftarrow 1$ **to** D **do**
      calculate K_d^* for θ_i
      $\gamma_d \leftarrow diag(QK_d^*Q^T)$
    end for
    $\gamma \leftarrow \bigotimes_{d=1}^{D} \gamma_d$
    $\kappa \leftarrow kron\_mvprod(K_1^*, \ldots, K_d^*, \alpha)$
    $\frac{\partial \log Z(\theta)}{\partial \theta_i} \leftarrow -\frac{1}{2}(\alpha^T\kappa + (\Lambda + \sigma_n^2)^{-T}\gamma)$
  end for
  $\mu_* = K(X_*, X)\alpha$ ▷ Predictive mean
  for all $i \leftarrow$ columns of $K(X_*, X)$ **do**
    $A(i,:) = kron\_mvprod(Q_1^T, \ldots, Q_D^T, K(X_*(:,i), X)$
  end for
  $A \leftarrow (\Lambda + \sigma_n^2 I)^{-1}A$
  for all $i \leftarrow$ columns of $K(X_*, X)$ **do**
    $A(i,:) = kron\_mvprod(Q_1, \ldots, Q_D, A(i,:))$
  end for
  $\Sigma_* = K(X_*, X_*) - K(X_*, X)A$ ▷ Predictive variance
end function

where X_W are the locations of the missing data points and y is $[y(X), y_W]$. The paper shows that in this setting the newly constructed covariance matrix, which we denote with K_W is equivalent to the original matrix $k(X,X) + \sigma_n^2 I$ in the sense that $K_W^{-1} y = (k(X,X) + \sigma_n^2 I)^{-1} y(X)$. Unfortunately, K_W does not retain all the good properties a regular grid matrix would have. In particular, one can no longer use the following step $(Q\Lambda Q^T + \sigma_n^2 I)^{-1} y = Q(\Lambda + \sigma_n^2 I)^{-1} Q^T y$ in computing $K_W^{-1} y$ since the noise added to K_W is no longer uniform. [21] gets around this problem by using Preconditioned Conjugate Gradients(PCG) to solve the system $K_W \alpha = y$ and obtain $\alpha = K_W^{-1} y$ that way. For computing the PCG, one needs only to compute products of the form $K_W y$ which can be done based on the `kron_mvprod` subroutine. The overall computational complexity of the modified algorithm becomes to $\mathcal{O}\left(J(\sum_{d=1}^{D} n_d + \sum_{d=1}^{D} n_d N)\right)$, where J are the number of iterations the PCG needs for converging. Other problematic calculations are the ones in Equation 2.34 and Equation 2.36. [21] approximates them using the results in [22] to arrive at:

$$\log|k(X,X,\theta) + \sigma_n^2 I| = \sum_{d=1}^{N} \log(\lambda_i + \sigma_n^2) \approx \sum_{i=1}^{|X|} \log(\frac{|X|}{|X| + |X_W|} \lambda_i^* + \sigma_n^2) \text{ and} \tag{2.39}$$

$$diag((\Lambda + \sigma_n^2 I)^{-1}) \approx \frac{|X|}{|X| + |X_W|} diag(\Lambda)(1 : |X|) + \sigma_n^2 \ , \tag{2.40}$$

where λ_i^* are $|X|$ largest eigenvalues of K_W and $diag(\Lambda)(1 : |X|)$ denotes the vector composed of them. With this approach, although an approximation of marginal likelihood and its derivatives is computed, the inference is exact. Moreover, the approximation is very accurate, especially for large inpainted regions.

2.3.6 Modelling Textures with GPs

Up until this point in Section 2.3, the discussion was focused on how to make Gaussian Processes suitable for texture synthesis. In the next few lines, the focus is switched to why Gaussian Processes are a suitable way to model textures, in the first place. The GP framework discussed in this section is a non-parametric machine-learning algorithm that models explicitly first and second order statistics of the texture. Section 2.2.2 suggested a strong correlation between first and second order statistics and the quality of texture synthesis. Therefore, one can expect that visual quality of texture synthesis to improve gradually in the pro-

cess of the covariance hyperparameter tuning. In that sense, Gaussian processes' hyperparameter tuning procedures act in the setting as a texture-quality loss optimization. Another reason that GP texture synthesis can prove advantageous over the algorithms discussed in Section 2.2.3 is that it does not make the additional Markov assumptions. Using the SMP kernel introduced in Section 2.3.4.4, GPs are able to model arbitrary relationships between pixels very far away from each other. Therefore, they can model simultaneously structure on a couple of different levels in the picture e.g. they can capture the change in the luminosity of a textured surface while capturing the periodic patterns presented in the texture itself. It is worth noting that the SMP covariance function works in the Fourier domain of the data. The abundance of image processing tools in the literature that use Fourier analysis demonstrates that a lot of useful information in images is better encoded in Fourier space [23]. That helps the SMP to model textures using only a few hundred parameters. Last but least, Gaussian Processes are stochastic processes and as such they provide natural ways to sample from them which helps in deriving unconstrained texture synthesis procedures similar to the Boltzmann machines discussed in [1].

Chapter 3

Datasets

This chapter introduces the dataset used in the experiments presented in Chapter 4. The chapter is split up into sections, as follows. Section 3.1 introduces the Brodatz dataset along with a brief justification of our choice to use it. Section 3.2 defends our decision to use a restrict subset of the textures in the dataset. Section 3.3 lists the images for the readers' convenience.

3.1 The Brodatz Dataset

The Brodatz dataset is a standard dataset containing 112 different greyscale texture images. The textures in the dataset are very different in their nature raging from microscopic images of cells, through images of brick walls, to images of star clusters. They were first presented in [24] back in 1966. In the early 1990s, the pictures were digitalised and turned into the dataset still used today. The Brodatz dataset was the first of its kind and quickly gained popularity within the Computer Vision community which it retains to this date due to the its enormous versatility of textures types and origins presented in such a small number of pictures. Recently, researchers has started to move away for the dataset claiming that its unrealistically perfect lighting conditions make it unsuitable for real world inpainting. Although there is some merit to those claims, we decided to stick with the dataset, since it is a good first step towards assessing our framework that has never been systematically used and assessed for texture synthesis before. Moreover, this allowed more direct comparison to [1].

In our experiments, we used a modified version of the data, available at [25], that has undergone normalisation preprocessing.

3.2 Restricting The Brodatz Dataset

Following [1], we restricted our experiments to a small subset of the data consisting of 8 textures. Those are D103, D16, D21, D4, D53, D6, D68 and D77. Focusing on only 8 textures reduced the computational cost of our experiments and thus allowed us to properly evaluate and experiment with the framework within the time constraints this project is subject to. The precise choice of those 8 textures again was dictated by comparability considerations. However, visually inspecting the textures we see that they are chosen to be very different from one another with a slight emphasis on more regular textures which are expected to perform better with machine learning frameworks such as ours. That makes them a good subset to perform our evaluation.

3.3 The Texture Album

In the next few pages, we provide 300×300 samples from each of the 8 textures listed above as a reference point for the reader.

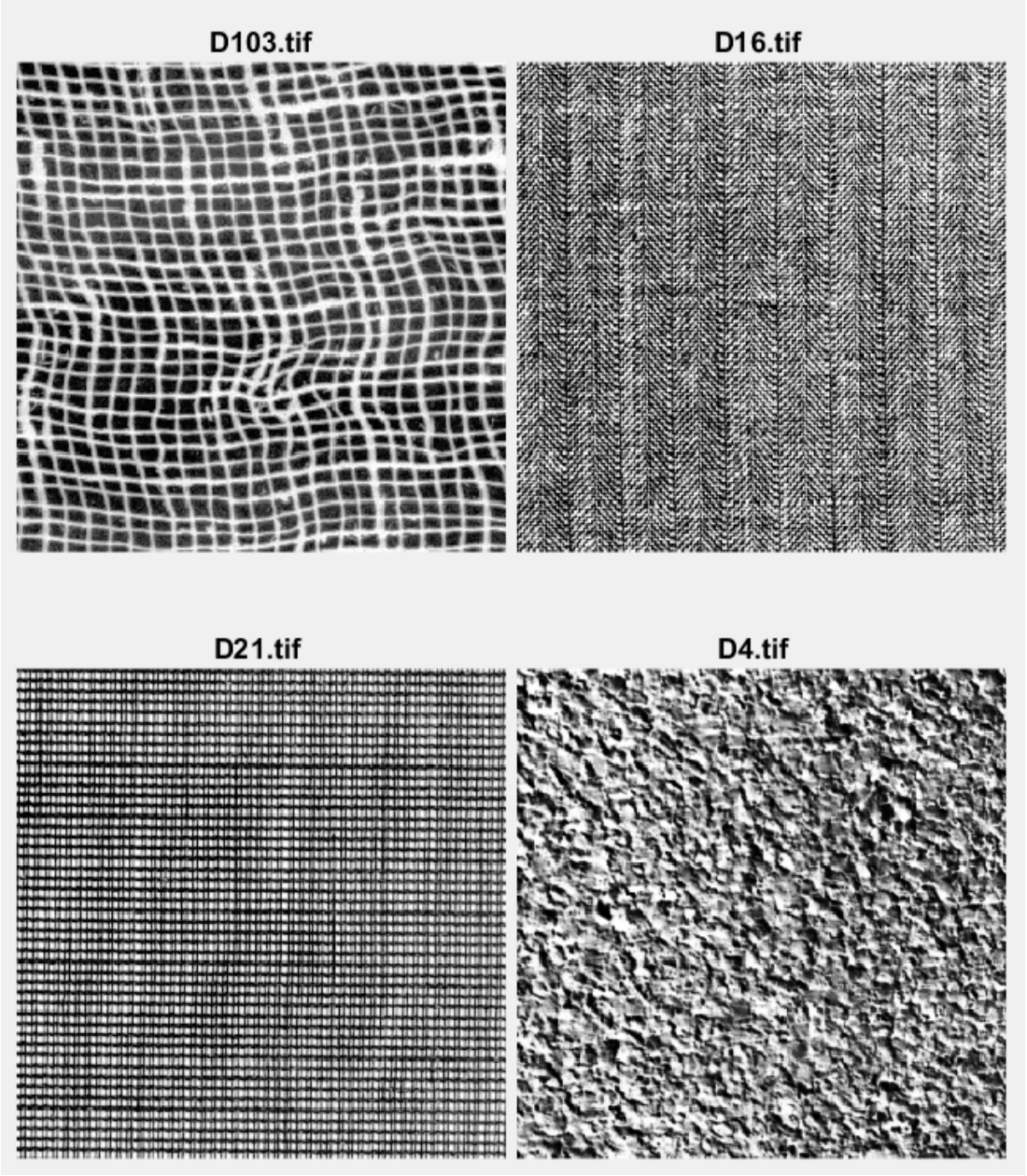

Figure 3.1: 300 × 300 samples from the Brodatz texture images — D103 (top left), D16 (top right), D21 (bottom left) and D4 (bottom right).

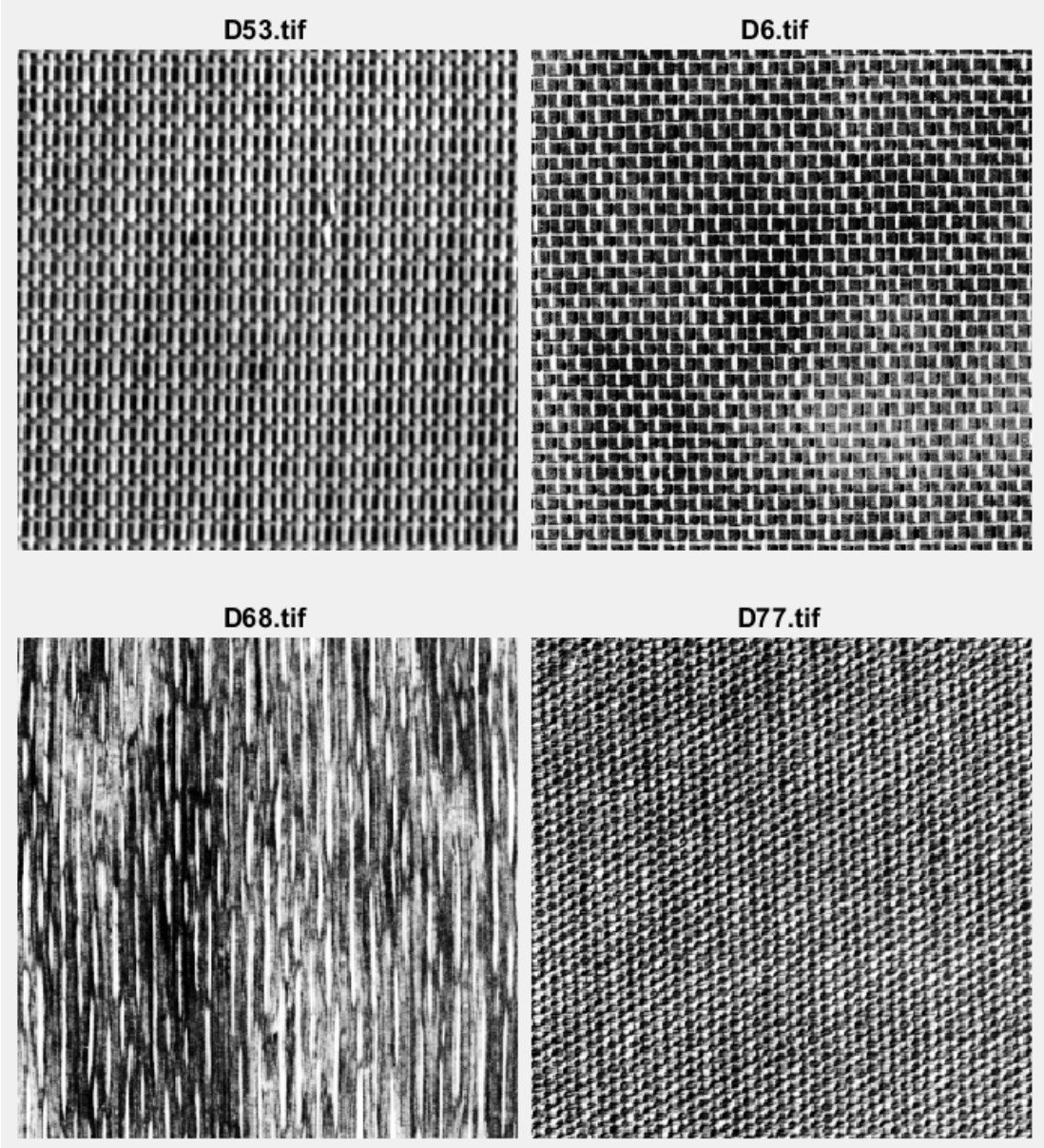

Figure 3.2: 300×300 samples from the Brodatz texture images — D53 (top left), D6 (top right), D68 (bottom left) and D77 (bottom right).

Chapter 4

Experiments and Results

In this chapter we define the experiments we performed and we critically evaluate GP's performance on them in comparison to [1]. The chapter is structured as follows. In Section 4.1 we introduce our external software choices and we motivate them. In Section 4.2 we define the experiments we performed along with the quality measures used to asses their performance. Additionally, we look at the specific choices we made in approaching the texture synthesis with Gaussian Processes. Finally, we conclude the chapter with a demonstration of the results and their critical evaluation in Section 4.3.

4.1 Software framework

To construct the experiments presented in this chapter we are using the GPML software package. The GPML tool-kit is an open source project implementing Gaussian Process regression and classification algorithms for Matlab. It is available free of charge at [26]. It is one of the most extensive packages for Gaussian Processes to date providing numerous inference algorithms, both exact and approximate, many different covariance functions, different likelihood functions and mean functions. The choice of this particular package was mainly influenced by the fact that implementations of the SMP kernel and the `GPR_GRID` algorithm required for our experiments are readily available in GPML.

Another important piece of software used in our experiments is the NFFT library. It is available at [27] and implements Non-Uniform Fast Fourier Transformation

algorithms which we need for the initialisation step of our Gaussian Processes framework. Our choice of this particular piece of software is motivated by the fact that it is one of the easiest to set-up and that it is the only one to provide a Matlab interface. With the help of the NFFT library, our Fourier transformations, that we originally tried calculating directly, were drastically sped up from an hour per transformation to just a couple of seconds.

To implement the sampling procedures for our unconstrained texture synthesis we used a Matlab implementation of the Lanczos algorithm available at [28]. At first, we tried implementing the algorithm ourselves but due to numerical problems and challenges with identifying the correct convergence conditions for the algorithm we decided to switch to the implementation provided at [28]. The implementation was chosen to the few other alternatives available because the code is easy to read and modify.

Last but not least, we used two other libraries for improving the computational cost of minor calculations in our unconstrained texture synthesis preconditioning steps. One of them was the fast implementation of the binary search algorithm available at [29] while the other was the fast implementation of the Matlab library function VChooseK available at [30]. Both of those were vital for our experiments because they execute extremely often and the implementation cited above provided computations orders of magnitude more efficient than their original counterparts.

We mention that the code written for the constrained texture synthesis experiments was influenced by the example code provided by Wilson at [31]. The code was designed as a supplementary material to [21] and demonstrates the usage of Wilson's implementation of the `GPR_GRID` algorithm. Conforming with our library choices, we implemented the rest of the code base and all of the visualisation in Matlab.

4.2 Experiments

This section will introduce the experiments we performed to evaluate the applicability of the Gaussian Processes framework introduced in Chapter 2 for texture synthesis. We split the section into two parts - experiments performed in the

inpainting setting and experiments for the unconstrained texture generation.

4.2.1 Constrained Texture Synthesis

4.2.1.1 Experimental Set-up

For our constrained texture synthesis experiments we adopted a scheme similar to the one demonstrated in [1]. That was done on purpose to enable direct comparison between our results and the results demonstrated there. The experiment's set-up is as follows. The image of a single texture is treated as the dataset for the problem. As often done in machine learning we split it into train and test sets. The train set for our purposes is the top half of the image while the test is the bottom. The purpose of the train and test sets will become clear in Section 4.2.1.2. From the training data random 76×76 patches are picked and a hole of size 54×54 is carved into them. The remaining data is fed to the Gaussian Process regression model which uses only those few pixels to predict the hole. After the prediction step is complete, a set of measurements of the quality of the inpainted region is calculated. We inpaint each random patch five times to account for the variance coming from the random initializations. Twenty patches from each texture are chosen in total. That may seem like a rather small number, but it was the one suggested by [1] and constrains the experiments' running time to few days.

4.2.1.2 Quality Measures

In the next few lines, we present the quality measures we adopted for the evaluation of our constrained texture synthesis algorithms. Again for the purpose of comparability, we adopt the quality measures presented in [1]. Those are the Normalized Cross Correlation(NCC), the Texture Similarity Score(TSS) and the Mean Structural Similarity Index (MSSIM). We turn to each of them in turn.

NCC

The Normalized Cross Correlation(NCC) score treats the inpainted region and the pixels, removed as part of the hole, as two high dimensional vectors where each dimension of the vectors is a single pixel value. The NCC

computes the cosine of the angle between those vectors using the formula:

$$NCC(x, x^*) = \frac{vec(x)^T vec(x)^*}{|vec(x)||vec(x)^*|} , \tag{4.1}$$

where x is the predicted region and x^* is the carved one. The NCC aims to compare the inpainting results to the original texture directly in illumination-independent fashion. Therefore, one cannot expect NCC to produce a good quality measurement for textures that are highly stochastic because in that setting many different inpaintings of the region will be visually plausible. NCC, however, works surprisingly well for the kind of textures this dissertation is dealing with.

TSS

The Texture Similarity Score(TSS), first introduced in [12], computes the NCC of the inpainted region with each other region in a test set and takes the maximum score. Its formula is given in Equation 4.2. The idea behind the quality measure is similar to the concept behind the exemplar inpainting introduced in Section 2.1. That is, we expect to find some other regions in the texture that look very similar to the one we are trying to inpaint. TSS motivates the introduction of the test set mentioned above based on the following observation. If the TSS was to be computed on the whole texture image, an algorithm that simply copies over patches from its training domain to the inpainted region will score high despite the fact that visually that will most probably look bad. Therefore, we want to make sure that the pixels used at training time are not part of the regions we consider in computing the TSS. The training set separation allows TSS to be looked at as kind of a generalisation error function. TSS should, in theory, perform better for more irregular textures than the NCC, for similar reasons to those presented for exemplar inpainting in Section 2.2.

$$TSS(x, x_1^*, x_2^*, \dots x_n^*) = max(NCC(x, x_1^*), NCC(x, x_2^*), \dots, NCC(x, x_n^*)) \tag{4.2}$$

MSSIM

The Mean Structural Similarity Index (MSSIM), introduced by [32], is a widely used similarity measure that measures the level of distortion present in an image. [32] demonstrates that it successfully picks out common problems typical of image processing tasks, such as blurring, salt and pepper

noise and change in the image luminosity and contrast. Those problems are weighted according to the sensitivity of the human perception to them. The measure is derived based on the decomposition of images to luminosity, contrast and structure channels with the structure channel having the most influence on the measure. More information about the exact calculations of the measure can be found at [32]. We note here that MSSIM gives reasonable scores to both the successful texture inpainting results and unsuccessful ones. Like the other scores presented in this section, its values reside between 0 and 1 where 1 means no distortion is present, and 0 means that the inpainting result is completely different from the original texture region.

4.2.1.3 The Gaussian Process Framework

In this section, we briefly outline the GP framework we use to construct the constrained texture synthesis experiments presented in the chapter. Given a patch to be inpainted, we first need to apply a normalisation to its known pixel values. The original mean μ and standard deviation σ are stored for later use. The normalised values are then modelled as a Gaussian Process with an SMP covariance function with 40 components and a zero mean function. Next, we perform inference on the unknown pixels using the inference procedure shown in Section 2.3.5. As part of the inference, some way of fitting the hyperparamaters of the model to the data is required. We choose to fit them similarly to the material described in Section 2.3.3, where the marginal log likelihood is optimised using BFGS. The efficient calculations of the log likelihood and its derivatives required for the BFGS algorithm to work are performed by the `GPR_GRID` algorithm. After the unknown pixel values are predicted, we use μ and σ to rescale the predictions back to the original image value range.

A not immediately obvious problem arising in this framework is that the marginal log likelihood function imposed by the SMP kernel has a lot of local minima. Thus initialisation procedures has huge impact on the performance of the algorithm. In our experiments we considered two different initialisation procedures.

The first, suggested in [21], work as follows. We initialise each of the components of the SMP kernel separately as equally-weighted Gaussians with weights

summing to the standard deviation of the data. Each Gaussian has uniformly chosen random mean between 0 and the Nyquist frequency of the data which in the case of image inpainting is $\frac{1}{2}$. The inverse of the standard deviation of each of the Gaussians is drawn from another truncated Gaussian distributions with mean proportional to the range of the data. 150 sample initialisations are drawn from the scheme above from which the one producing the smallest log likelihood is taken to be the initialisation fed to the BFGS.

The second initialization scheme we considered takes the known pixel values and performs a Non-Uniform Fast Fourier Transformation(NUFFT) on them using the NFFT library. The frequencies at the which the NUFFT is evaluated are equally spaced on the interval from zero to the Nyquist frequency similar to the previous initialisation scheme. The next step is to take the log magnitute of the spectrum and fit a Gaussian mixture model with a diagonal covariance to it. We use the means and the variances of the mixture components to initialize the SMP means and variances and replicate the mixture weigths over the different SMP components.

The results of both initialisation schemes are considered in Section 4.3.

4.2.2 Unconstrained Texture Synthesis

4.2.2.1 Experimental Set-up

Our experimental set-up for the unconstrained texture synthesis is as follows. First, we feed the train set, which as we already mentioned is the top half of the texture image, to the GP framework presented in Section 4.2.1.3, learning the hyperparameters associated with the texture along with its mean μ and its standard deviation σ. We then consider two different sampling strategies for the problem — sampling from the Gaussian Process prior distribution and sampling from the Gaussian Process posterior distribution. We feed the learned parameters to each of the sampling strategies, which then generate 128 samples each, each of size 120×120. Finally, the TSS score of both of the methods is calculated on the test set. The other quality measures we presented in Section 4.2.1.2 are not applicable to the unconstrained texture synthesis setting and thus are not calculated.

4.2.2.2 Sampling from the GP Prior

[13, Chapter 2] demonstrates how sampling from a Gaussian Process prior distribution can be used to generate random functions that share the statistical properties encoded into the GP prior covariance matrix. We explore the applicability of this idea to the problem of texture synthesis. In particular, we assume that the covariance matrix generated by the GP framework discussed in Section 4.2.1.3 captures the the structure of the texture inside itself and, therefore, by sampling from a GP prior with that covariance matrix we expect to generate valid texture samples. [13, Chapter 2] suggests a sampling procedure based on the Cholesky decomposition LL^T of the GP covariance matrix K. Samples are generated using the equation:

$$s = Lv + \mu \ , \tag{4.3}$$

where v is a vector of samples generated by a 1-D Gaussian distribution with zero mean and standard deviation of one. In order the calculate s for big texture patches we need an efficient way to compute L. To do that we turn back to the properties of Kronecker products discussed in Appendix A. Given that we select an rectangular patch we can express K as Kronecker product of matrices similarly to what we do in the derivations of the `GPR_GRID` algorithm in Section 2.3.5. Using property A.10 we derive the efficient to compute formula for the product Lv below:

$$\begin{gathered} Lv = Chol(K)v = Chol(\bigotimes_{d=1}^{D} K_d)v = \left(\bigotimes_{d=1}^{D} Chol(K_d)\right) v = \\ \texttt{kron\_mvprod}(Chol(K_1), Chol(K_1), \ldots, Chol(K_D), v) \ , \end{gathered} \tag{4.4}$$

where $Chol$ denotes the lower triangular matrix resulting from the Cholesky decomposition of its argument, K decomposes to the Kronecker product $\bigotimes_{d=1}^{D} K_d$ and `kron_mvprod` denotes the `kron_mvprod` subroutine derived in Section 2.3.5.2. The running time of this procedure is $\mathcal{O}\left(\sum_{d=1}^{D} n_d^3 + \sum_{d=1}^{D} n_d N\right)$, where K_d is of size $n_d \times n_d$ and $N = \prod_{d=1}^{D} n_d$. That running is a huge improvement over the $\mathcal{O}\left(N^3\right)$ required for computing L directly.

4.2.2.3 Sampling from the GP Posterior

The assumption we made in Section 4.2.2.2 was that the structural information describing the texture is completely encoded in the covariance matrix of the GP prior distribution. Observing the success of our GP framework for inpainting and the failures of the sampling procedure in Section 4.2.2.2, we conclude that the assumption made is too strong. We relax it and instead assume that the structure of the texture is encoded in both the region surrounding the inpainted hole and the covariance matrix of the Gaussian Process. This gives rise to a GP posterior sampling scheme that conditions the samples on a small texture patch placed diagonally from the generated texture region as demonstrated in Figure 4.1. That particular arrangement was chosen so that no restrictions on the size of the generated texture are imposed.

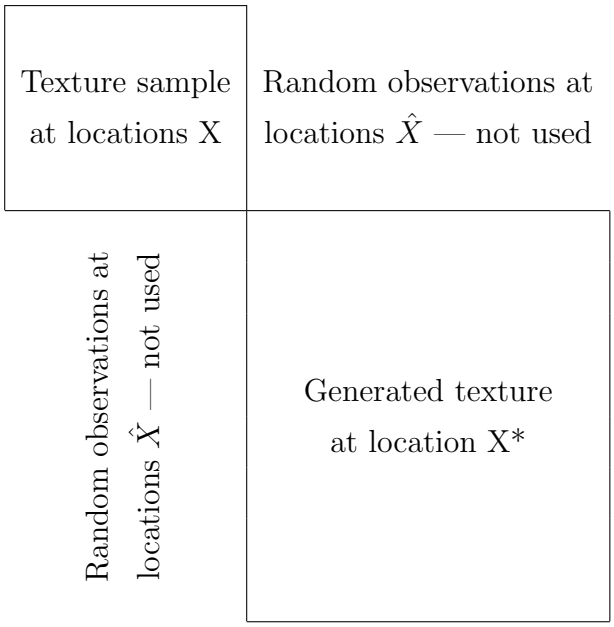

Figure 4.1: Gaussian Proccesses posterior sampling set-up

The formula for the covariance matrix of the posterior Gaussian Process was demonstrated in Section 2.3.2 to be :

$$\Sigma_* = k(X_*, X_*) + \sigma_n^2 I - k(X_*, X)(k(X, X) + \sigma_n^2 I)^{-1} k(X_*, X)^T. \tag{4.5}$$

Few things have to be noted about Equation 4.5. First, Σ_* is not a Kronecker product matrix and, therefore, the trick applied in Section 4.2.2.2 can no longer be used for sampling here. Second, an efficient computation for the product

$(k(X,X)+\sigma_n^2 I)^{-1}y$ is available as we described in Section 2.3.5. Moreover, efficient procedure for calculating $k(X_*,X)^T y$ is also available as we demonstrate in the lines to follow. To arrive at the formula we first construct the following covariance matrix:

$$\hat{K} = \begin{bmatrix} k(X,X) & k(X,X_*) & k(X,\hat{X}) \\ k(X_*,X) & k(X_*,X_*) & k(X_*,\hat{X}) \\ k(\hat{X},X) & k(\hat{X},X_*) & k(\hat{X},\hat{X}) \end{bmatrix}, \tag{4.6}$$

where $\hat{X}$ denote the locations of the imaginary observations in Figure 4.1. The locations X, X_* and $\hat{X}$ together form a multidimensional grid and therefore $\hat{K}$ can be written as the Kronecker product $\bigotimes_{d=1}^{D} K_d$. Therefore, we can use the `kron_mvprod` subroutine to efficiently calculate the product $\hat{K}y$ with any vector y. We rewrite the product $\hat{K}y$, as follows:

$$\begin{aligned} \hat{K}y = \begin{bmatrix} k(X,X) & k(X,X_*) & k(X,\hat{X}) \\ k(X_*,X) & k(X_*,X_*) & k(X_*,\hat{X}) \\ k(\hat{X},X) & k(\hat{X},X_*) & k(\hat{X},\hat{X}) \end{bmatrix} \begin{bmatrix} y_X \\ y_{X_*} \\ y_{\hat{X}} \end{bmatrix} = \\ \begin{bmatrix} k(X,X)y_X + k(X,X_*)y_{X_*} + k(X,\hat{X})y_{\hat{X}} \\ k(X_*,X)y_X + k(X_*,X_*)y_{X_*} + k(X_*,\hat{X})y_{\hat{X}} \\ k(\hat{X},X)y_X + k(\hat{X},X_*)y_{X_*} + k(\hat{X},\hat{X})y_{\hat{X}} \end{bmatrix}. \end{aligned} \tag{4.7}$$

In Equation 4.7 one can choose to set y_X and $y_{\hat{X}}$ to 0 to obtain:

$$\hat{K} \begin{bmatrix} 0 \\ y_{X_*} \\ 0 \end{bmatrix} = \begin{bmatrix} k(X,X_*)y_{X_*} \\ k(X_*,X_*)y_{X_*} \\ k(\hat{X},X_*)y_{X_*} \end{bmatrix}. \tag{4.8}$$

By selecting the first few entries in $\hat{K}y$ we obtain exactly the product we were trying to calculate. Since, for rectangular synthesis patch the matrix $k(X_*,X_*)$ is also a Kronecker product, we can efficiently calculate the product $k(X_*,X_*)y$ with the `kron_mvprod` subroutine, as well. Therefore, we can calculate the product $\Sigma_* y$ efficiently in $\mathcal{O}\left(\sum_{d=1}^{D} \hat{n_d}^3 + \sum_{d=1}^{D} \hat{n_d} N + J(\sum_{d=1}^{D} n_d^3 + \sum_{d=1}^{D} n_d N)\right)$, where $\hat{K}$ is a Kronecker product of D $\hat{n_d} \times \hat{n_d}$ matrices , $k(X,X)$ is a Kronecker product of D $n_d \times n_d$ matrices and J is the number of iterations required for the PCG algorithm to converge. For reasonably large texture samples that computation of $\Sigma_* y$ is a lot more efficient than the regular computation and furthermore it does not require the explicit creation of the matrix $k(X_*,X_*)$ which can grow over

the limit of most computers' RAM for comparatively small synthesis regions. Equipped with that procedure we are now ready to demonstrate the algorithm we used for sampling from the Gaussian posterior.

The algorithm was first presented in [33]. No implementation was supplied in the paper, so we implemented the algorithm from scratch following the guidelines provided in the paper and using the implementation of the Lanczos algorithm available at [28]. The idea behind the algorithm is to calculate the polynomial approximation $\Sigma_*^{\frac{1}{2}} y \approx \sum_{n=1}^{\infty} a_n \Sigma_*^n y$. Since the sampling scheme described in Section 4.2.2.2, actually works for any square root of the covariance matrix and not only the Cholesky decomposition i.e. any matrix M for which the covariance matrix can be expressed as MM^T, we restate the efficient sampling problem to the problem of efficiently calculating $\Sigma_*^{\frac{1}{2}} y$. The polynomial approximation described above requires only the calculation of the product $\Sigma_* y$ which we already demonstrated to be efficiently computable. To calculate the polynomial approximation the paper chooses rather indirect approach that first creates the orthogonal basis $v_1, v_2, \ldots, v_n$ of the Krylov subspace $K_n(\Sigma_*, y) = span\{y, \Sigma_* y, \ldots, \Sigma_*^{n-1} y\}$. This is done through a process similar to Gram-Schmidt. The algorithm for doing that when Σ_* is Hermitian which is always the case, since Σ_* is a covariance matrix, is called Lanczos algorithm or Lanczos process. We give a pseudo algorithm for the Lanczos process in Algorithm 3. In addition to the orthogonal basis, the Lanczos algorithm also returns as byproduct of its computations two set of values α and β that obey the following:

$$\Sigma_* V_n = V_n T_n + \beta_{n+1} v_{n+1} e_n^T \ , \tag{4.9}$$

where $V_n = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \ldots & v_n \\ | & | & & | \end{bmatrix}$, $T_n = \begin{bmatrix} \alpha_1 & \beta_2 & & & 0 \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_n \\ 0 & & & \beta_n & \alpha_n \end{bmatrix}$ and e_n is a n^{th} column of the identity matrix. Multiplying both sides of Equation 4.9 by the orthogonal matrix V_n^T gives:

$$V_n^T \Sigma_* V_n = T_n \ . \tag{4.10}$$

The optimal approximation of $\Sigma_*^{\frac{1}{2}} y$ i.e. the one that minimizes the 2-norm of the vector is given by:

$$y_* = V_n V_n^T \Sigma_*^{\frac{1}{2}} y \ . \tag{4.11}$$

Algorithm 3 The Lanczos algorithm

function LANCZOS(Σ_*, y, n)
  $\beta_1 = 0$
  $v_0 = 0$
  $v_1 = \frac{y}{\|y\|}$
  for $i = 1$ **to** n **do**
    $v = \Sigma_* v_i - \beta_i v_{i-1}$
    $\alpha_i = v_i^T v$
    $v = v - \alpha_i v_i$
    $\beta_{i+1} = \|v\|$
    if $\beta_{i+1} == 0$ **then**
      return i,$\{\alpha_1, \alpha_2, \ldots, \alpha_{i-1}\}$,$\{\beta_1, \beta_2, \ldots, \beta_i\}$,$\{v_1, v_2, \ldots, v_i\}$
    end if
    $v_{i+1} = \frac{v}{\beta_{i+1}}$
  end for
  return n,$\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$,$\{\beta_1, \beta_2, \ldots, \beta_{n+1}\}$,$\{v_1, v_2, \ldots, v_{n+1}\}$
end function

Rewriting y as $\|y\|V_n e_1$, denoting $f(\Sigma_*) = \Sigma_*^{\frac{1}{2}}$ and plugging them back in Equation 4.11 we obtain:

$$y_* = \|y\| V_n V_n^T f(\Sigma_*) V_n e_1 \ . \tag{4.12}$$

Approximating the product $V_n^T f(\Sigma_*) V_n$ with $f(V_n^T \Sigma_* V_n) = f(T_n) = T_n^{\frac{1}{2}}$ we can give the approximation:

$$\Sigma_*^{\frac{1}{2}} y \approx \|y\| V_n T_n^{\frac{1}{2}} e_1 \tag{4.13}$$

For n a lot smaller than the size of the covariance matrix Σ_*, T_n is efficient to calculate and store. To ensure that is the case, a good preconditioning procedure needs to be put in place. The preconditioning scheme discussed in [33] uses the sparse preconditioning matrix $G \approx \Sigma_*^{-\frac{1}{2}}$. Since $G \approx \Sigma_*^{-\frac{1}{2}}$, we know $G\Sigma_* G^T \approx I$. Therefore, applying the estimation procedure we just discussed to $G\Sigma_* G^T$, we expect to obtain $(G\Sigma_* G^T)^{\frac{1}{2}} y$ in just a couple of hundred iterations which is minuscule compared to the size of Σ_*. To use the calculation of $(G\Sigma_* G^T)^{\frac{1}{2}} y$ for sampling we multiply it by G^{-1}. The calculated matrix $S = G^{-1}(G\Sigma_* G^T)^{\frac{1}{2}}$ has the property:

$$SS^T = G^{-1}(G\Sigma_* G^T)^{\frac{1}{2}}(G\Sigma_* G^T)^{\frac{1}{2}} G^{-T} = \Sigma_* \tag{4.14}$$

Therefore, S is a square root of Σ_* and can be used for valid sampling.

For that preconditioning scheme to work, G and G^{-1} need to be efficiently computable, as well. If G is chosen to be a sparse approximation of the Cholesky decomposition of Σ_*^{-1}, $G^{-1}y$ can be computed line by line, solving a small linear system at each step because G would lower triangular and sparse. To compute G in this setting, we pose the optimisation problem $\|I - GL^T\|_F^2$, where L is the exact Cholesky decomposition of Σ_*, $\|\|_F$ denotes the Frobenius norm and G is a lower triangular matrix with a given sparsity pattern.[33] demonstrates that the solution of that opitmization problem is given by the solution of the equation $(\hat{G}A)_{ij} = I_{ij}$, where the pairs $\{(i,j)\}$ denote the sparsity pattern of G and $G = D\hat{G}$ for some diagonal matrix D. D in general is not known but in practice is chosen to force the diagonal entries in $G\Sigma_* G^T$ to be 1. Solving the equation for G is efficiently done line by line and can be parallelised. We use the Matlab parallel-for functionality to do that. The sparsity pattern we chose for our experiments is equally spaced grid with distance 70 between each entry. To make the pattern lower triangular we forced the diagonal entries to 1 and the entries above the diagonal to 0.

To test the applicability of the algorithm for approximate sampling from a covariance matrix created by the SMP kernel function we sampled using the procedure explained above the Gaussian Process prior described in Section 4.2.2.2. The results are shown in Figure 4.2. As seen from the pictures, the sampling procedure worked well even though it introduced few pixels that look like salt and pepper noise.

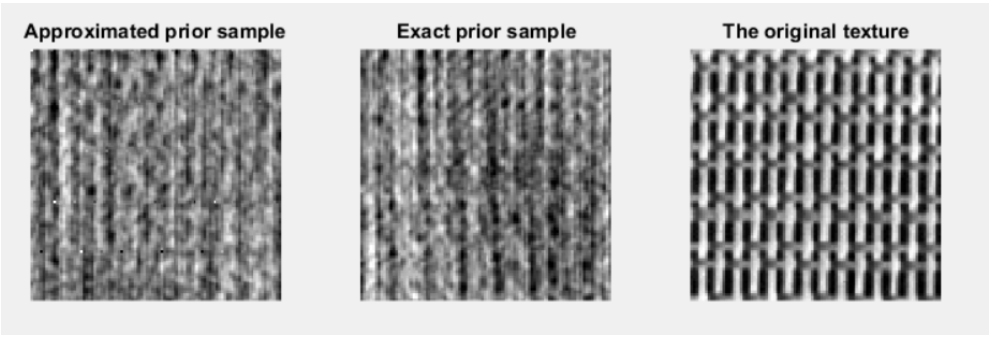

Figure 4.2: Sample textures drawn from, from left to right, approximated GP prior, exact GP prior, the original texture.

4.3 Results and Discussion

4.3.1 Constrained Texture Synthesis

Figure 4.3, Figure 4.4 and Figure 4.5 demonstrate the MSSIM, NCC and TSS scores obtained from our constrained texture synthesis experiments. The figures provide box plots for each of the textures and each of the initialisation schemes we discussed in Section 4.2.1. Box plots are standard statistical plots which expose several different statistical aspects of a dataset. The blue boxes indicate a range containing the middle 50% portion of the data and the red line inside them denote the median of the data. Furthermore, the whiskers denote the range $(-2\sigma, 2\sigma) + \mu$, where μ and σ are the mean and the standard deviation of the data, and the red crosses denote outlier that are outside of the whiskers' range.

We split the textures considered into two groups — regular textures and stochastic textures. The former consists of the set of textures with numbers D6, D21, D53 and D77, while the later are textures D103, D16, D4 and D68. The distinction is based solely on visual inspection of the textures and the patterns in them. From the box plots it is obvious that the best performing textures across all 3 measures are exactly the regular set of textures. That is consistent with the visual plausibility of the inpainted regions created for those textures, as well. The inpainted regions are demonstrated in Figures 4.6–4.13. Those observations allow us to conclude that the GP framework we chose to use for inpainting works better when the texture is generated predictably as function of the known pixels. That is not surprising given the way Gaussian Process predict the unknown regions — predictions are given as a Gaussian with a mean value and a standard deviation. If the standard deviation of a pixel is high and its the probability distribution is very different from a Gaussian e.g. we have reasonably big probability associated with a pixel value of 0 and a reasonably big probability of a pixel value of 1, but low probability of the pixel to be 0.5, we cannot expect that the mean value which will be approximately 0.5 in this example to look natural. In that setting, a prediction of 0 or 1 will be a lot more reasonable. The phenomenon described in the last few lines causes Gaussian Processes to predict grey and almost flat looking regions where no strong correlation is found between the region and the rest of the pixels in the image. It also causes the occasional blurring we see in some inpainted

regions. This lack of strong correlation can have multiple causes. One possibility is the stochastic nature of a texture. Another possibility, however, is that the SMP kernel is not able to pick out the relationship in first place. We notice that behaviour when we fail to find good initial values for the BFGS optimisation procedure we discussed and we converge to a bad local minima. Another reason for failure of the Gaussian Processes is that they do not model the geometry of the texture explicitly but rather implicitly through the probability relationship between pixels. Therefore, even though the log marginal likelihood optimisation prevents GPs of overspecification, the data itself can be selected in such a way that makes GP to overspecify the problem. That is the case with texture D103 in Figure 4.6, where the lack of examples of curvy lines forced the GP to try and fail fitting the straight lines observed in the image frame to the image interior.

We now shift our focus to the problem of initialisation. In Section 4.2.1.3 we considered two initialisation schemes. In the next few lines we will compare them based on their performance. Comparing the measurements in Figures 4.3–4.5, we notice that the Fourier transformation initialisation scheme introduces a lot more variance to the results. That is especially visible in the results for texture D53 and is further backed up by a visual inspection of Figures 4.6–4.13. That is unexpected given that the initial log marginal likelihood values we observed for each and every initialisation with the Fourier transformation scheme was similar or better than the random initialisations obtained with the Nyquist sampling procedure. Furthermore, on many occasions the observed Fourier initialisation produced log likelihood 1.5–2 times smaller than the random initialisation provided by the alternative initialisation. Two behavioural patterns contributed to the variance, both connected to the fact that this initialisation scheme even though useful, initialises the optimisation procedure close to a local minima. The first is that on many occasions the BFGS just converged to that local minima in 10–20 iterations without exploring the a lot more advantageous minima close to it. The second one is that in the process of escaping the local minima the BFGS wasted a lot of iterations and thus converged overall slower to a similar value to the one explored by the random initialisation. Since, we truncated the number of iterations for computational considerations that had a detrimental result to some Fourier initialisations' performance, as well. Most of the Fourier worse-case inpainting results presented in Figures 4.6–4.13 were results of bad

convergence. Interesting to note is that even though the Fourier initialisation introduces a lot more variance to the results, its best-case performance is still as good as the Nyquist performance and on many occasions it is even slightly better. That is especially the case with the more complicated stochastic textures. There the random sampling procedure is less likely to find a good initialisation and the advantage of choosing an already good initialisation overcomes the convergence problems. The best-case performance is a strong indication that a sampling around the value produced by the Fourier initialisation can work out as good and robust initialisation scheme but was not considered here due to time constraints.

We conclude this section with a brief discussion on the how our GP results compare to the Boltzmann machines method presented in [1]. [1] did not provide any measurements for the quality of the inpainting of the more stochastic texture subset, neither did it demonstrate any results for it in the form of pictures. The only results it demonstrated for the subset are the unconstrained texture synthesis results. Therefore, we cannot directly compare our methodology to the methodology of [1] on those textures. If the unconstrained texture synthesis demonstrated is anything to go about, however, we can expect the Boltzmann machines to perform similarly or slightly better than our top results demonstrated in Figures 4.6–4.13. Our results on the stochastic subset, even though not that visually appealing, captured a lot of the structure in the textures which is apparent from the very high NCC scores across the board. Unfortunately, they also introduced a lot of annoying artefacts which is captured by the low MSSIM scores. For the more regular subset we notice that our NCC scores are consistently better than [1]. That does not directly imply that our results are better but suggests that they are slightly closer to the original texture. That is to be expected because Gaussian Processes are very effective at regression problems whereas Boltzmann machines are expected to work better at sampling and hence in unconstrained texture synthesis. Looking at the MSSIM scores of our Nyquist initialisation inpainting we notice that, while our method performs better on D21 by a solid margin, it is few percent behind the best results presented in [1] on all other textures. Those few percent difference come from bad initialisations and the blurry regions we already discussed. The good performance on D21 is also to be expected, since D21 is most regular of the textures we experimented with. The TSS scores we observe for our GP framework are consistently 5–10%

lower than the ones of obtained by the Boltzmann machines. That suggests the inpainted regions we created are more specialised to their setting opposed to the Boltzmann machine ones which are more general. That is consistent with the observations we made for the NCC scores.

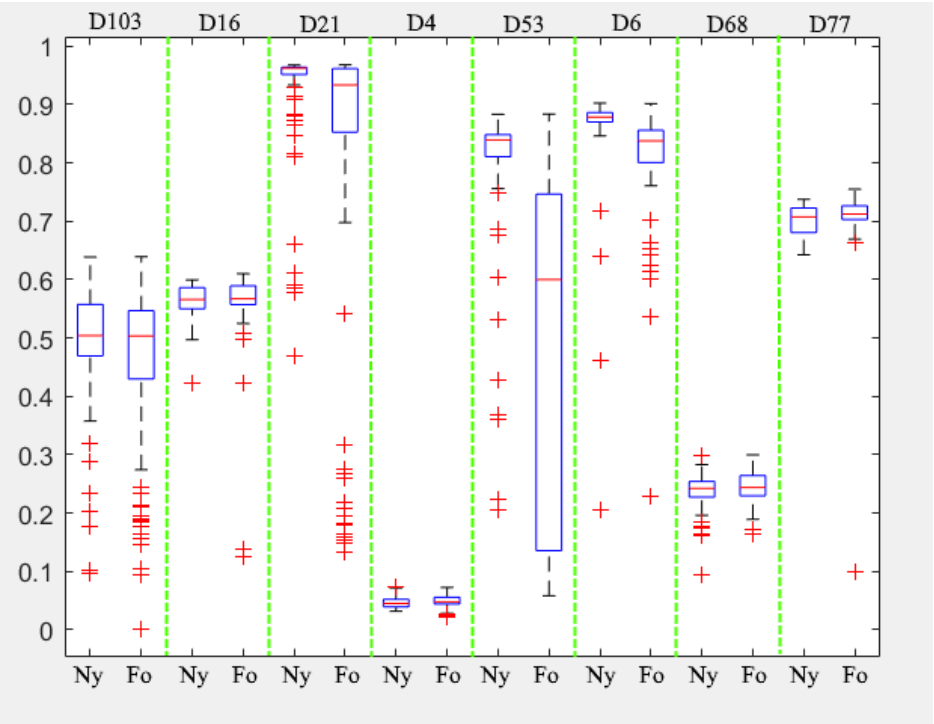

Figure 4.3: The MSSIM scores of the constrained texture synthesis with Nyquist sampling initialisation(Nq in the figure) and Fourier transformation initialisation(Fo in the figure) for all 8 textures considered.

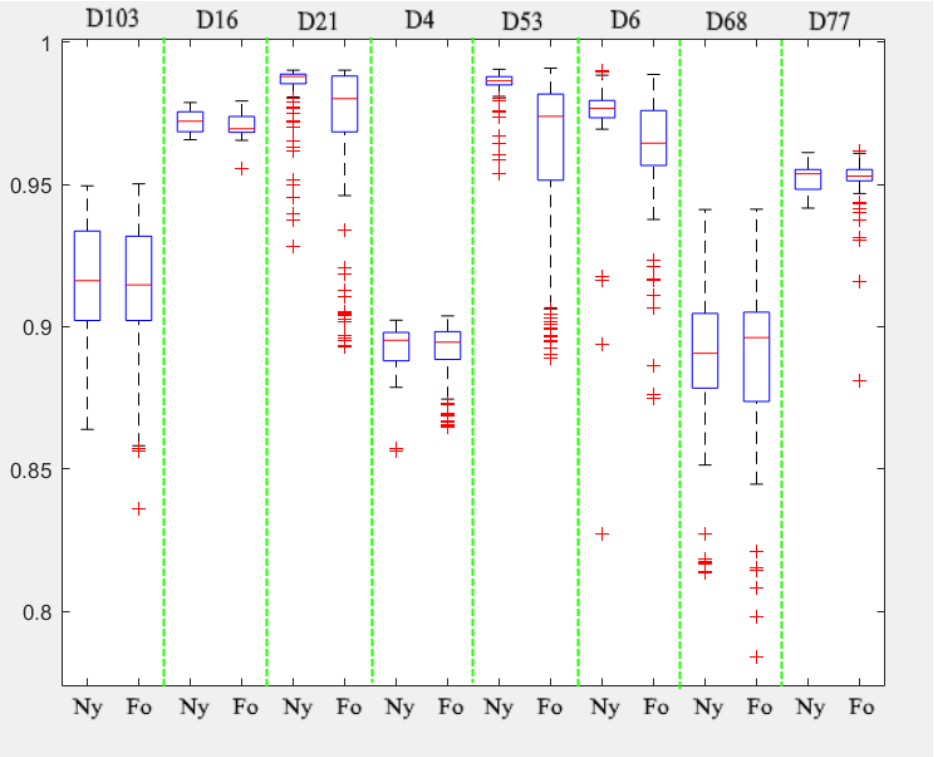

Figure 4.4: The NCC scores of the constrained texture synthesis with Nyquist sampling initialisation(Nq in the figure) and Fourier transformation initialisation(Fo in the figure) for all 8 textures considered.

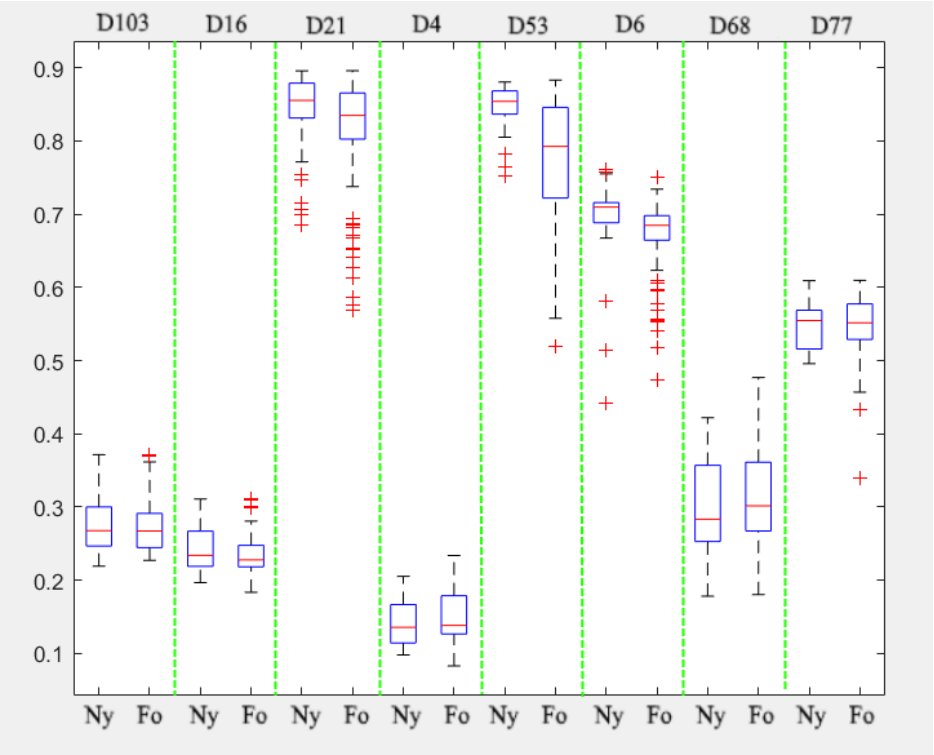

Figure 4.5: The TSS scores of the constrained texture synthesis with Nyquist sampling initialisation(Nq in the figure) and Fourier transformation initialisation(Fo in the figure) for all 8 textures considered.

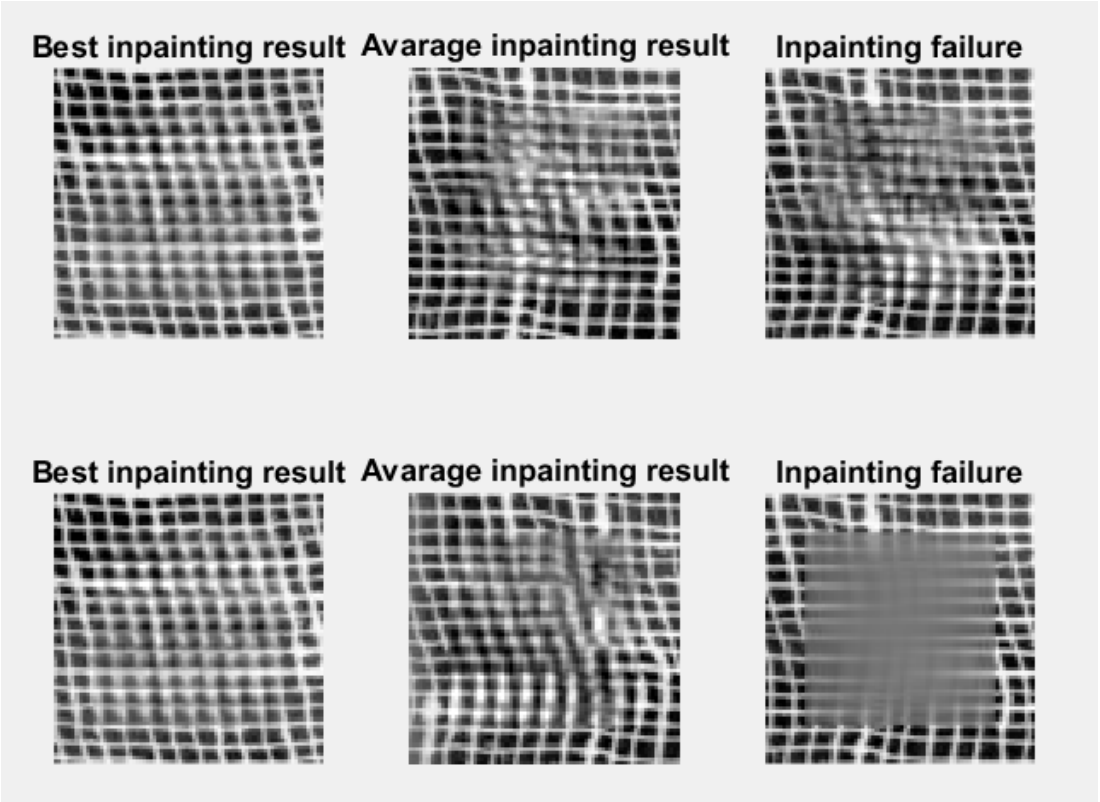

Figure 4.6: Examples results of inpainting D103 with Nyquist sampling initialisation(top) and Fourier transformation initialisation(bottom).

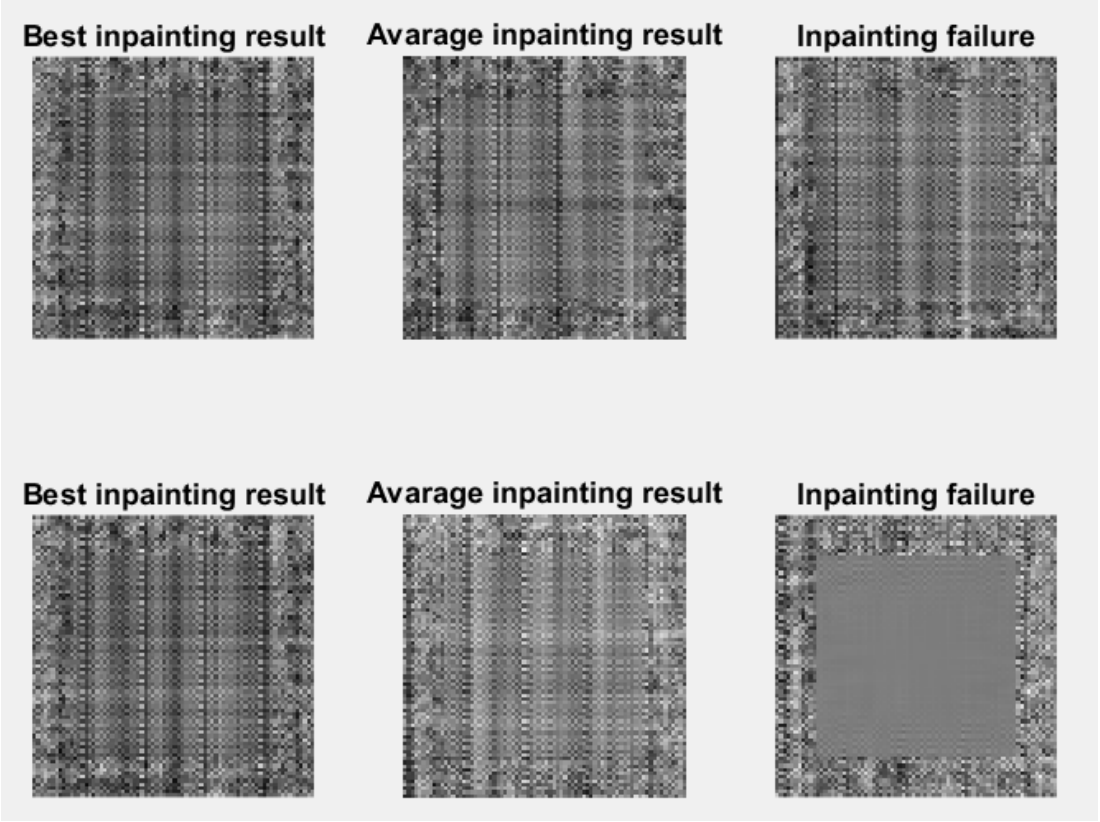

Figure 4.7: Examples results of inpainting D16 with Nyquist sampling initialisation(top) and Fourier transformation initialisation(bottom).

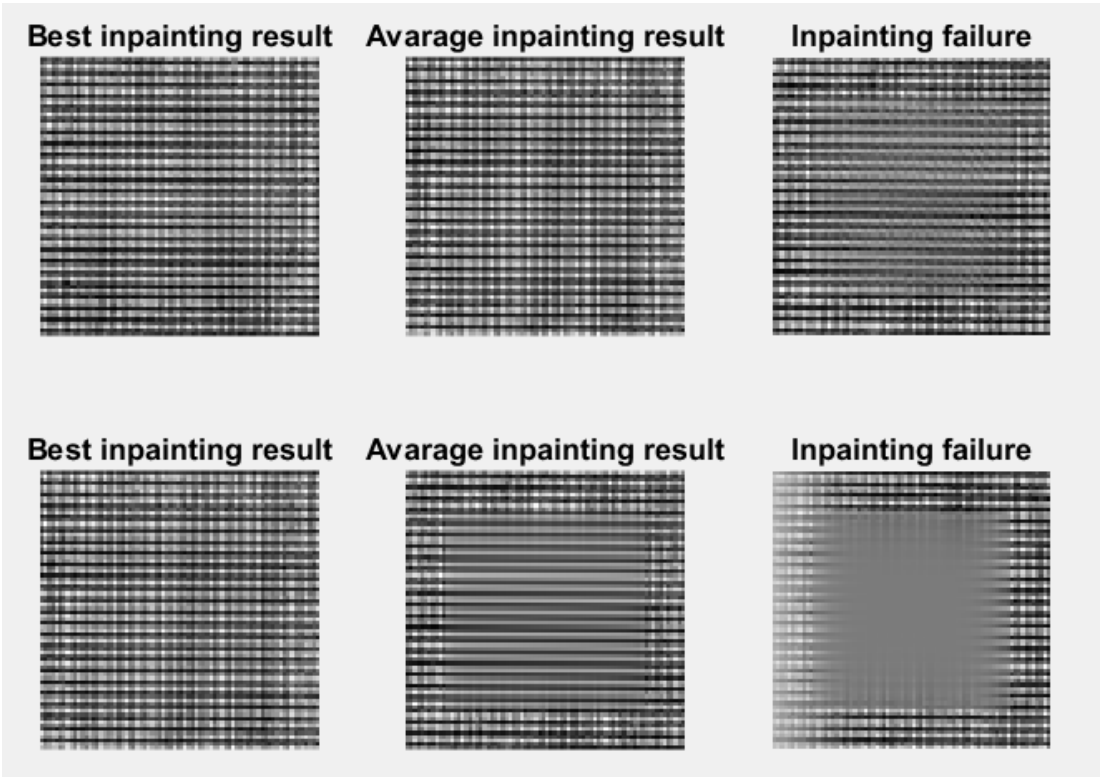

Figure 4.8: Examples results of inpainting D21 with Nyquist sampling initialisation(top) and Fourier transformation initialisation(bottom).

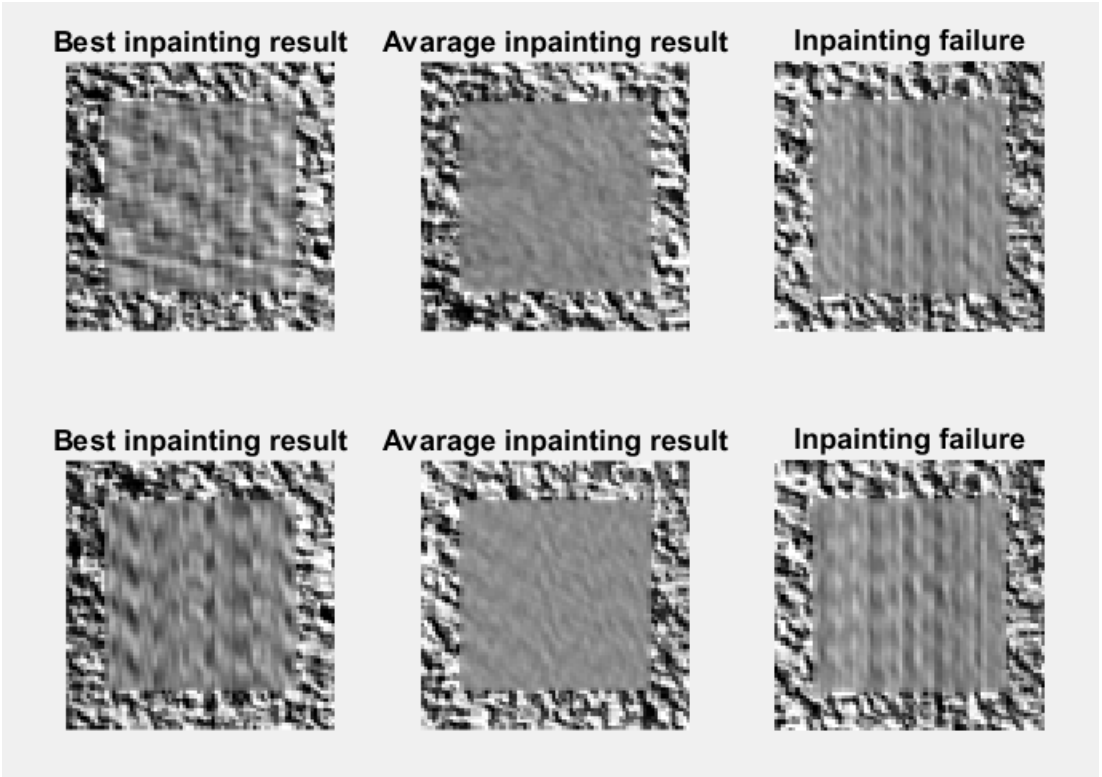

Figure 4.9: Examples results of inpainting D4 with Nyquist sampling initialisation(top) and Fourier transformation initialisation(bottom).

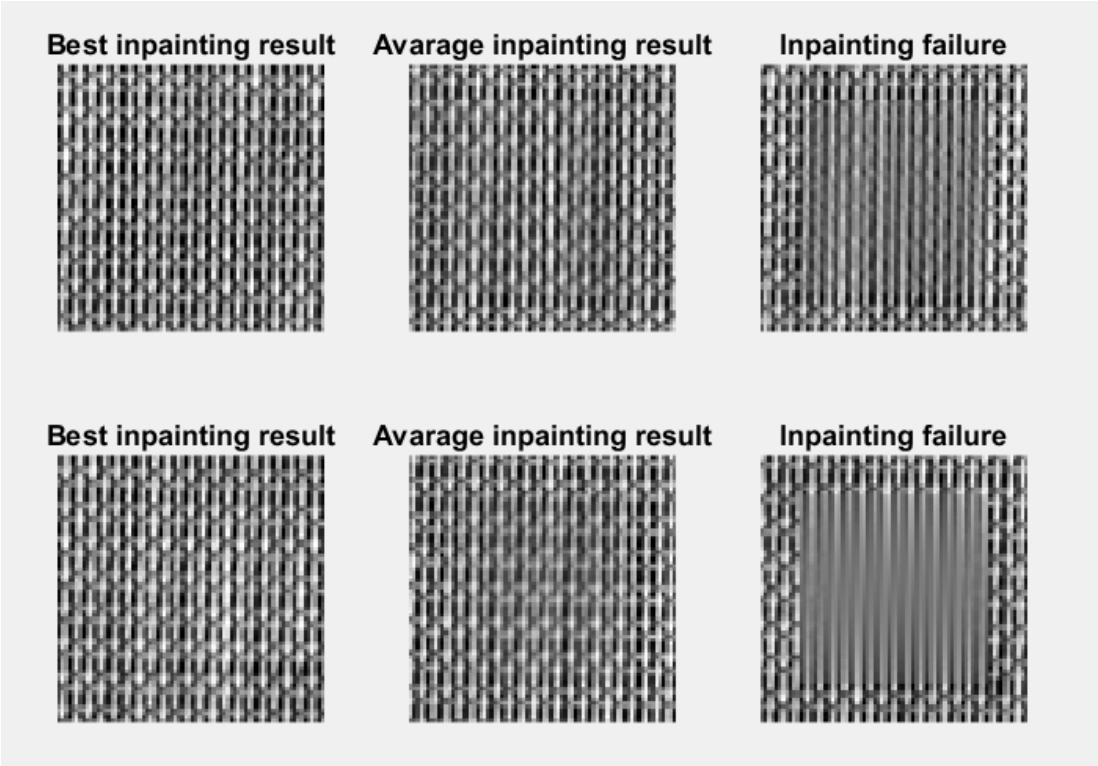

Figure 4.10: Examples results of inpainting D53 with Nyquist sampling initialisation(top) and Fourier transformation initialisation(bottom).

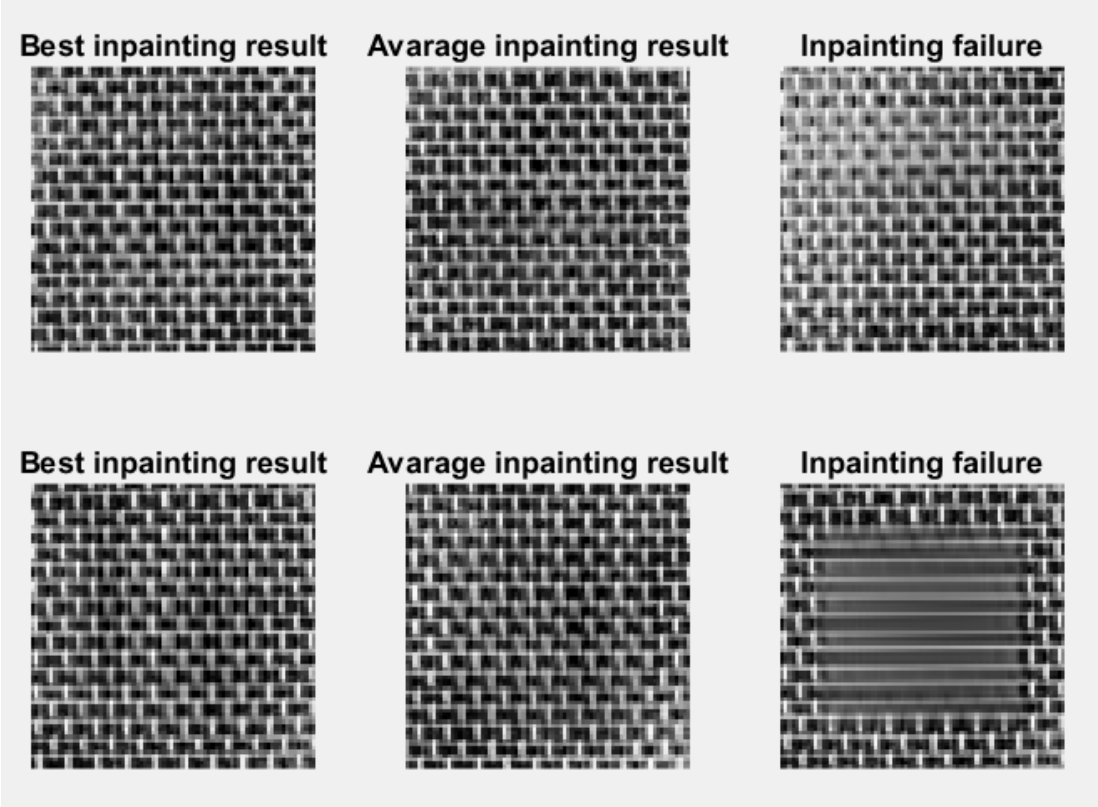

Figure 4.11: Examples results of inpainting D6 with Nyquist sampling initialisation(top) and Fourier transformation initialisation(bottom).

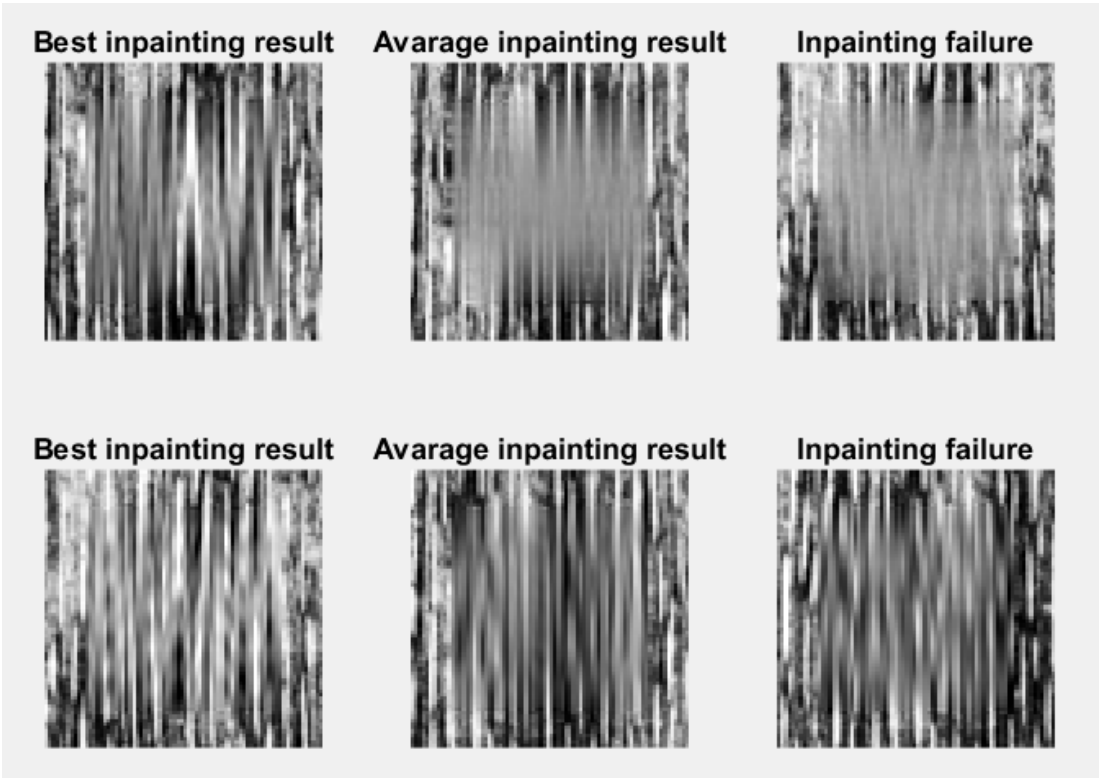

Figure 4.12: Examples results of inpainting D68 with Nyquist sampling initialisation(top) and Fourier transformation initialisation(bottom).

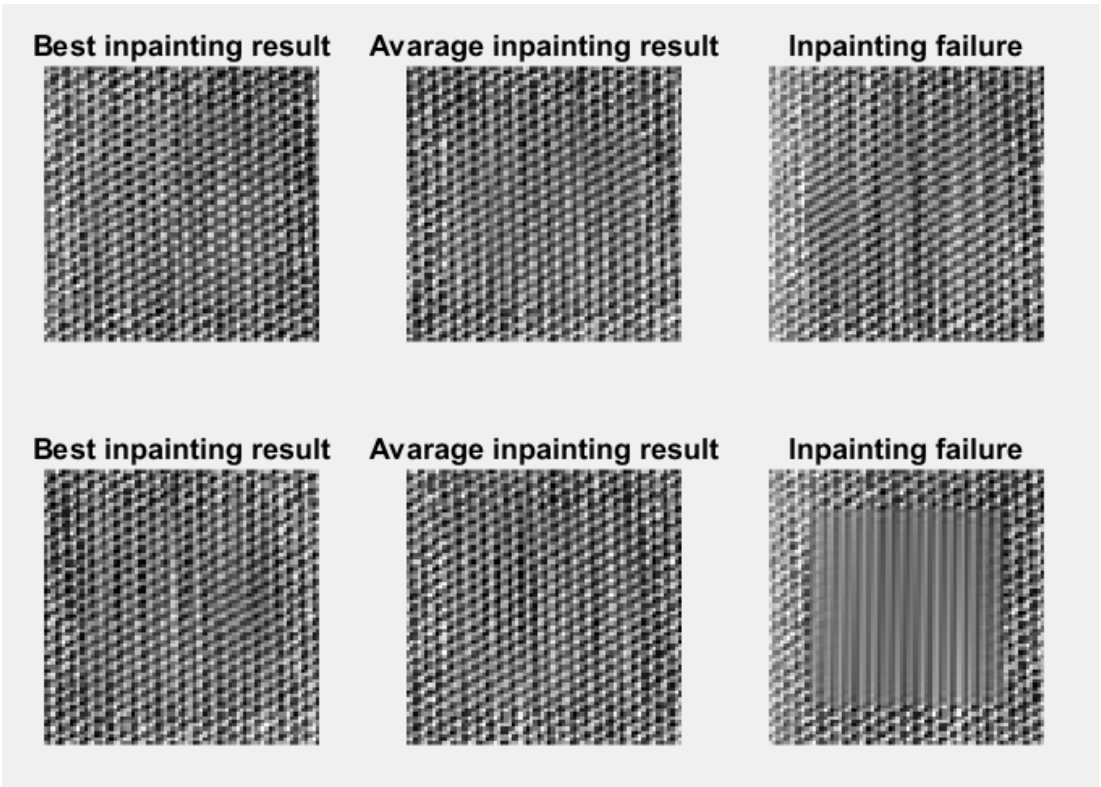

Figure 4.13: Examples results of inpainting D77 with Nyquist sampling initialisation(top) and Fourier transformation initialisation(bottom).

4.3.2 Unconstrained Texture Synthesis

Figure 4.15 shows the TSS scores for the Gaussian prior sampling procedure discussed in Section 4.2.2.2. Figures 4.17 and 4.18 show randomly chosen generated samples. As we see, we got hardly any result on textures D103, D68 and D77. The results for those look mostly like random white noise without any structure to them and thus produce TSS scores of practically 0. The reason for that was that during the training of the hyper parameters, the GP was not able to get good hyperparameter estimation for the SMP kernel. In all those texture the BFGS gave up after few iterations with marginal log likelihood above 1e+5. The reason for the bad performance of the SMP kernel here opposed to the inpainting examples is that in our prior sampling procedure we chose to optimise the log likelihood on much larger region. The top half of the image was considered opposed to the small 76×76 windows. The stochastic nature of those particular textures prevented the SMP kernel of learning any overall, across-the-image structure. Interesting to note is the relatively good performance of the Nyquist GP prior sampling on D4. D4 got a bad TSS score but that is not a big factor because visually it has a lot of similarities with the original texture patch. Its low score can be explained with the failure of the TSS measure on the highly stochastic texture. Another interesting observation is that the performance of the two initialisations schemes was drastically different. The textures for which the Nyquist sampling initialisation performed well, observed white-noise failures with the Fourier initialisation and the other way around. We ran each experiment multiple times to make sure that was not cause by nuisance factors but the results were similar each time. That shows that the two initialisation scheme are very different from one another and that both of them has their own merits and problems. The best performing textures in the set were the regular ones which as already discussed is not surprising. All the TSS score we obtained are extremely low with 0.54 being the best. That suggests that the GP framework missed a lot of the structural information encoded in the texture. Not all of is lost however, since we see that in many of the better results primitives like lines and lightning patterns were retained.

Assuming the problem with the prior sampling is indeed the variability of the larger texture region, we decided to explore the results of training the model

on a smaller patch. We chose to do that on a random window of size 76×76 for consistency with our inpainting experiments. The results are shown in Figures 4.16, 4.19 and 4.20. We immediately notice a substantial improvements for all textures. Additionally, there are no longer white noise textures, except for D16 in the Fourier initialisation case. All of the samples demonstrated in Figures 4.19–4.20, capture at least some information about the patterns they were trained on, with D21, D68, D77 and D4 looking not that perceptually different from their original counterparts. The TSS scores also improved dramatically for all textures and patches but D4. In most of the cases the median of the TSS scores improved by ≈ 0.1. Despite this huge improvement, the TSS scores are still far worse than the ones presented in [1]. In fact, we observe that our best performing patches were in many occasion scored worse than one of the worse outliers produced by the Boltzmann machines in [1].

The main problem with the samples we obtained from our prior sampling procedure is their lack of geometrical structure such as strong edges and corners. Instead of them, the prior GP covariance matrix captures the direct relationship between pixels i.e. it captures what happens with a pixel if we observe certain pixel values to which it is strongly correlated. That suggests that the structural information in the inpainted regions in the experiments in Section 4.3.1 is not completely stored in the SMP kernel parameters but also in the pixel values of the training locations. Thus, sampling from a GP prior cannot be expected to produce good unconstrained texture synthesis results. No matter how well we optimize our hyperparameters, the Gaussian Process will try to extract pixel relationships more than anything else and will try to leave as much as possible of the specific details required for the texture synthesis encoded in the texture itself. That is done as part of the marginal log likelihood optimisation to prevent overspecification and is the only way the SMP kernel is able to produce such good inpainting results with so small set of hyperparameters.

In Section 4.2.2.3 we argued that sampling from the GP posterior $GP(\mu_*, \Sigma_*)$ has the potential to overcome these problems if there was an efficient way to sample from that posterior. There, we presented an approximated framework that makes the computations associated with the sampling procedure feasible. Despite that the procedure still does not allow the sampling to be practical. In our experiments, for synthesis patches of size 120×120 with a prior texture

sample of size 100×100 the preconditioning step of the procedure took more than 60 hours of CPU time. Due to the parallelised computations that accounted to only a little more than a day on a quad-core computer. Not surprisingly, after the preconditioning step was done the sampling was a lot faster. Nevertheless, it took anywhere between 10 and 30 minutes to calculate a single texture sample on a single core of the CPU. Due to that computational cost we were unable to experiment with the framework sufficiently and instead we opted for running synthesis with a single texture and hyperparameters learned on the whole dataset. In the experiments presented in this section we showed that is not the optimal way to learn the hyperparameters but we did not know that at the time of the experiment. Sample is given in Figure 4.14. We notice that the sample is not perceptually worse than our best performing prior sample despite the inefficiencies in the experimental set-up. To properly asses the performance, however, a full set of experiments needs to be performed. The initial result suggests some merit to the concept.

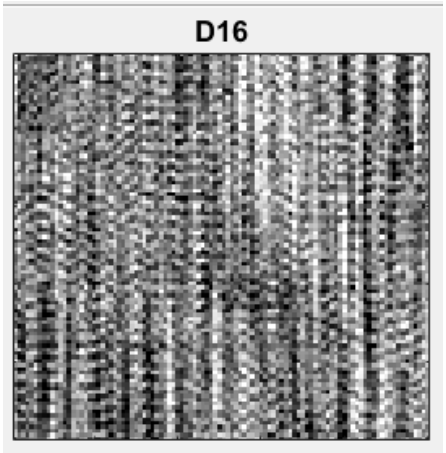

Figure 4.14: A sample from Gaussian Process posterior distribution of texture D16 created with the Nyquist sampling initialisation procedure on the full dataset.

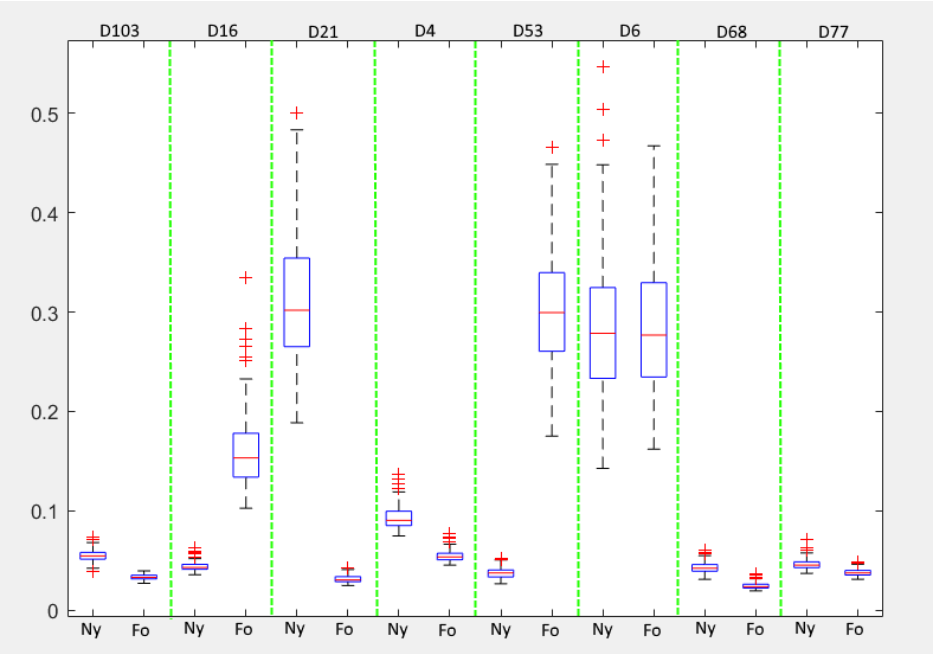

Figure 4.15: The TSS scores of the GP prior sampling synthesis trained on the whole training set with Nyquist sampling initialisation (Nq in the figure) and Fourier transformation initialisation (Fo in the figure) for all 8 textures considered.

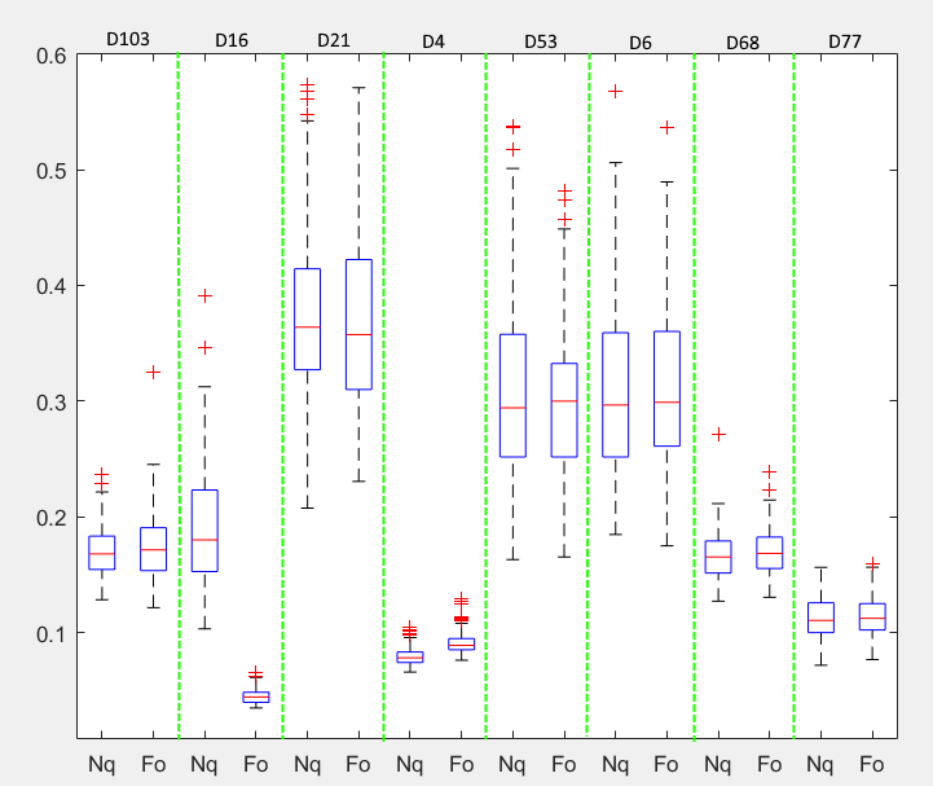

Figure 4.16: The TSS scores of the GP prior sampling synthesis trained on a random 76×76 patch with Nyquist sampling initialisation (Nq in the figure) and Fourier transformation initialisation (Fo in the figure) for all 8 textures considered.

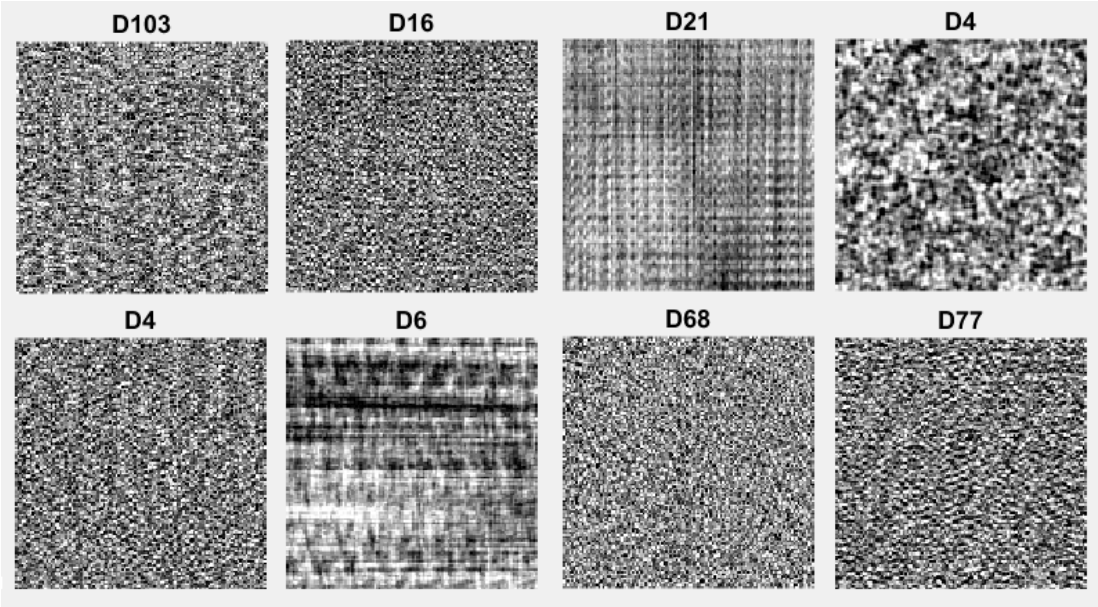

Figure 4.17: Samples from Gaussian Process prior distribution created with the Nyquist sampling initialisation procedure on the full training set.

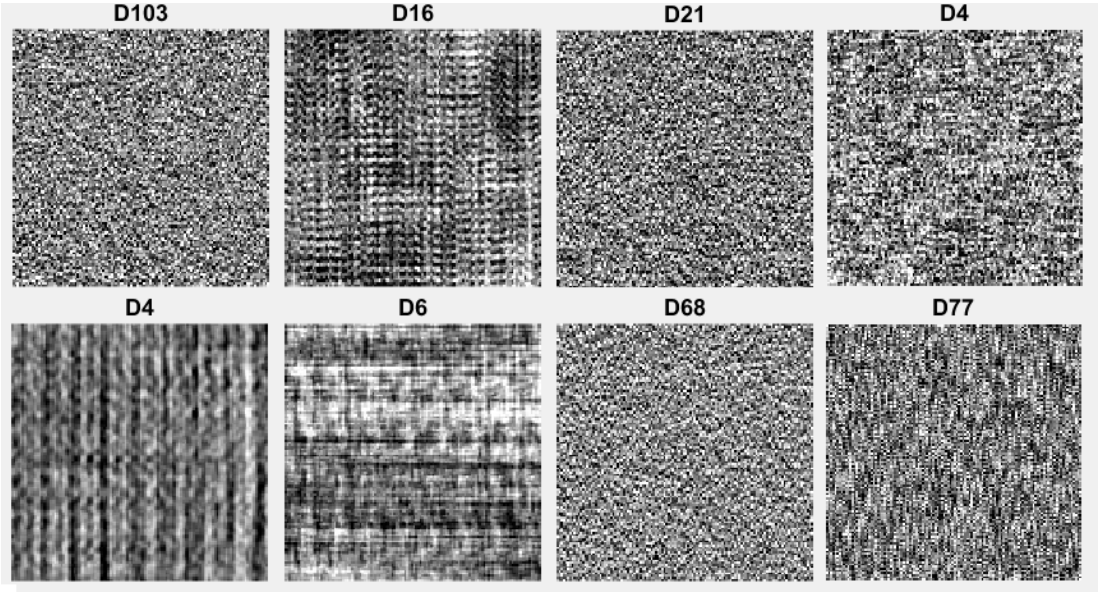

Figure 4.18: Samples from Gaussian Process prior distribution created with the Fourier initialisation procedure on the full training set.

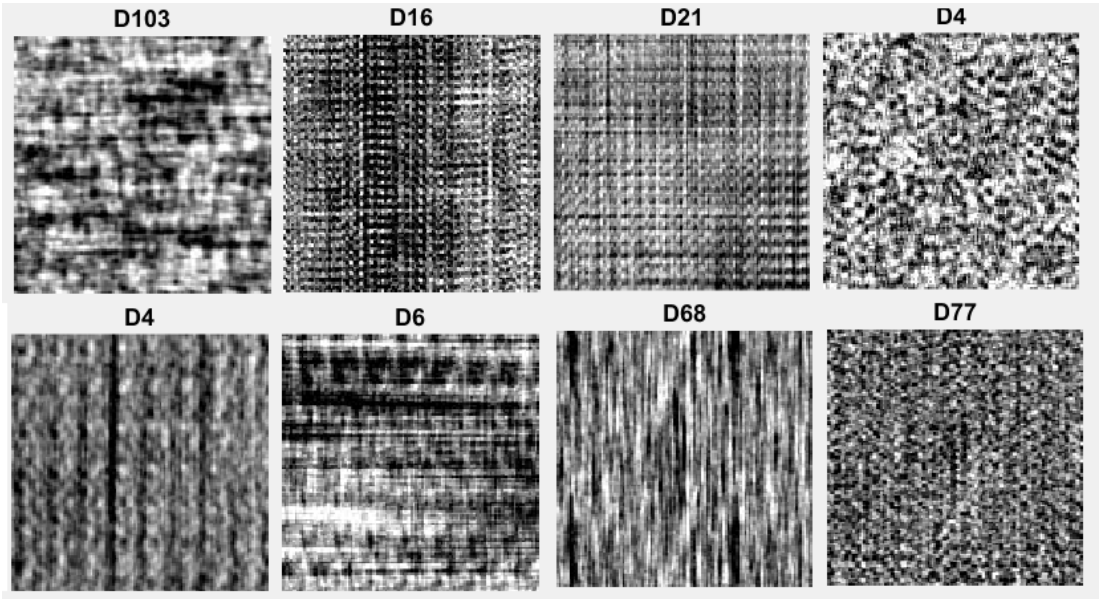

Figure 4.19: Samples from Gaussian Process prior distribution created with the Nyquist sampling initialisation procedure on a random 76×76 patch.

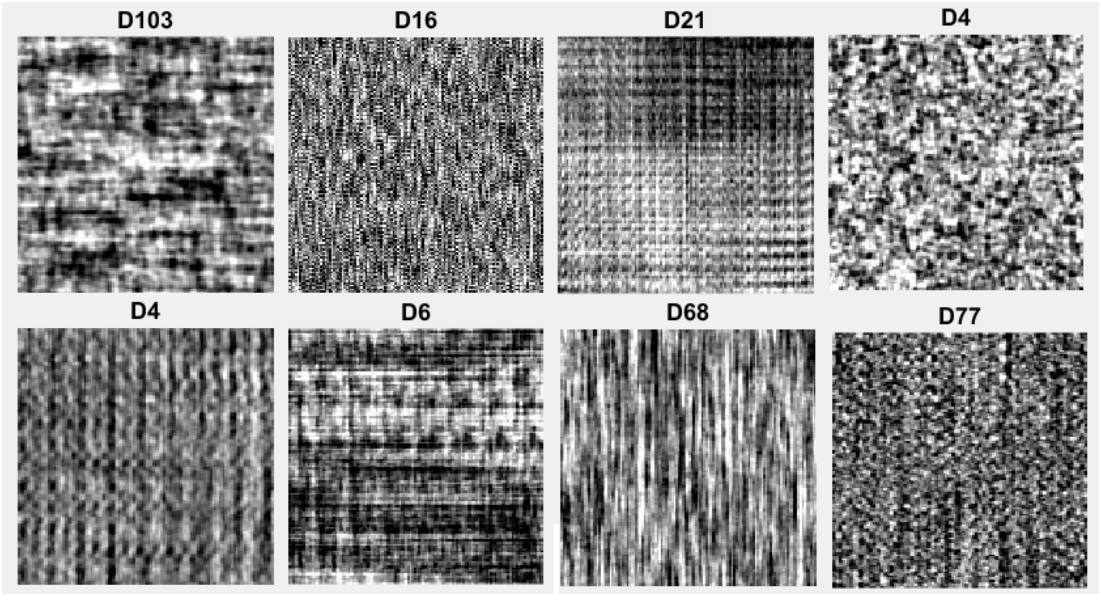

Figure 4.20: Samples from Gaussian Process prior distribution created with the Fourier initialisation procedure on a random 76×76 patch.

Chapter 5

Conclusion and Future work

In this chapter we summarise the results of the experiments, carried out as part of this dissertation, and briefly discuss directions for their further improvement. The chapter is split into two parts. In Section 5.1, we discuss the constrained texture synthesis problem, while in Section 5.2, we discuss the unconstrained case.

5.1 Constrained Texture Synthesis

In Chapter 4, we demonstrated that the expressiveness of the SMP kernel combined with the scalable inference provided by the `GPR_GRID` algorithm forms a powerful tool for inpainting of regular textures. Moreover, we showed that the inpainting results of that framework are comparable to the state of the art results presented in [1]. Last but not least, we described different initialisation strategies and their key role for the visual quality of the texture inpainting.

Given that the initialisation is so important for the described Gaussian Processes to perform well, the next logical step for improving upon the results presented in Chapter 4 is exploring more initialisation strategies or combining and modifying the existing ones. In particular, one can consider a sampling procedure for the suggested Fourier initialisation similar to the sampling procedure implemented for the Nyquist sampling, where means are sampled from Gaussians centred around the centres suggested by the Fourier initialisation. The rational here is that this can avoid the local minima problem discussed in Chapter 4. Another possible

solution to avoid the local minima problem is to rerun the EM algorithm used for the mixture model fitting until a initialisation with good convergence rate is found, possibly changing the EM parameters along the way. The convergence rate can be assessed by running the BFGS for few iterations with each of the initialisations. Delaying to commit to initialisation until after the convergence rate is observed can prove useful for selecting a suitable initialisation scheme on a per-texture basis, as well. We expect all those suggestions to reduce the variance of the inpainting results which is one of the biggest problems of the architecture, so far.

Another direction for future work is provided by our observations in Section 4.3.2 that some problems with fitting the SMP kernel to larger regions in stochastic textures are present. The claim needs to be further justified with experiments on bigger texture patches with possibly larger holes and remedial measures need to be developed.

Finally, additional set of experiments that can gain further insight in the way the SMP kernel works and how it encodes the texture in its hyperparameters can be carried out to improve our understanding of the overall framework. For example, hyperparameters from the experiments presented in this document can be used to visualise spectral density of the SMP kernel. Another useful visualisation can be a plot of the number of the meaningful SMP components i.e. the ones that do not have weights close to 0, as function of some measure of the level of regularity of the textures. This will give us some expectation as to how many SMP components are required to perform good texture inpainting on more stochastic textures, such as D4.

5.2 Unconstrained Texture Synthesis

In Chapter 4, we presented the results of a Gaussian Process prior sampling procedure as a way of achieving unconstrained texture synthesis. The results were not satisfying and we argued that is to be expected given the nature of the information stored in the GP prior. We also argued that a more suitable to the problem idea is to try sampling from a GP posterior distribution, instead, which conditions the generated region on a texture sample. Last but not least,

we presented a computationally expensive but feasible procedure for sampling from the GP posterior. Unfortunately, we failed to properly evaluate it within the time-frame of the project. An obvious direction for continuation of the work presented in this dissertation is to explore properly the applicability of that posterior framework. Moreover, alternative approaches for sampling can also be considered. Assuming, that the GP prior contains some information as to how the data can generally vary, samples from the prior covariance matrix can be combined with the predictive mean of the GP posterior to give a fast framework that has the potential to work indistinguishably from the GP posterior sampling, most of the time. Even though, we have been discussing sampling frameworks up until now, there are a lot of practical unconstrained texture synthesis problems that do not require infinitely many valid texture sample, but just a few that work well. In that type of a setting, a regular GP inference procedure can be used to give a predictive mean for the synthesised texture region similarly to the GP inpainting problem. Conditioning on different regions in that framework will still produce different texture patches, so it can prove advantageous for some applications. In conclusion, the unconstrained texture synthesis with GPs remains open to research. Given the results presented for the texture inpainting, however, we expect that further development on the topic will eventually manage to achieve it. How practical the result will be remains unknown.

Appendices

Appendix A

Kronecker Product Properties

This section contains list of properties of the Kronecker matrix product used to derive the procedures in Section 2.3.5. [18, Chapter 5] lists the following basic properties of Kronecker products of square matrices:

Property A.1. (Bilinearity) $A \bigotimes (B + C) = A \bigotimes B + A \bigotimes C$

Property A.2. (Associativity) $A \bigotimes (B \bigotimes C) = (A \bigotimes B) \bigotimes C$

Property A.3. (Mixed-product property) $(A \bigotimes B)(C \bigotimes D) = (AC) \bigotimes (BD)$

Corollary A.1. (Factorization) $A \bigotimes B = (A \bigotimes I_{n_B})(I_{n_A} \bigotimes B) = (I_{n_B} \bigotimes A)(B \bigotimes I_{n_A})$, *where* I_{n_A} *and* I_{n_B} *are the identity matrix of the size of A and B, respectively.*

Repeated application of Corollary A.1 leads to:

Corollary A.2. *For every set of square matrices* A_d *of size* $n_d \times n_d$ *one can rewrite their Kronecker product as* $\bigotimes_{d=1}^{D} A_d = \prod_{d=1}^{D} I_{N_d} \bigotimes A_d \bigotimes I_{\overline{N_d}}$, *where* $N_d = \prod_{i=1}^{d-1} n_i$, $\overline{N_d} = \prod_{i=d+1}^{D} n_i$ *and* $\prod$ *denotes the standard matrix product of the D matrices.*

Repeated application of Property A.3 can be used to derive:

Corollary A.3. *For any two sets of square matrices* A_d *and* B_d *of the same size:*

$$(\bigotimes_{d=1}^{D} A_d)(\bigotimes_{d=1}^{D} B_d) = \bigotimes_{d=1}^{D} (A_d B_d). \tag{A.1}$$

Property A.4. (Inverse) $(A \bigotimes B)^{-1} = A^{-1} \bigotimes B^{-1}$

Property A.5. (Transpose) $(A \bigotimes B)^T = A^T \bigotimes B^T$

Property A.6. (Trace) $tr(A \bigotimes B) = tr(A)tr(B)$

Property A.7. (Determinant) $det(A \bigotimes B) = (detA)^{n_A}(detB)^{n_B}$, *where A is of size* $n_A \times n_A$ *and B is of size* $n_B \times n_B$.

Property A.8. (Vec) *$vec(ABC^T) = (C \bigotimes A)vec(B)$, where the vec operation takes a matrix and returns a vector of its columns stacked one after another.*

[34] also adds the following property:

Property A.9. (Kronecker product with identity) *$I_m \bigotimes A = diag(A, \ldots, A)$, where the $diag(A, \ldots, A)$ is block diagonal matrix containing m copies of A on its diagonal.*

Additionally, [18, Chapter 5] provides proofs for some of the more advanced Kronecker results used in the algorithm. The interested reader is encouraged to read the proofs supplied there. For readers' convenience those are listed below, as well:

Property A.10. (Cholesky) *If L_d is the Cholesky decomposition of K_d i.e. $K_d = L_d L_d^T$, then $L = \bigotimes_{d=1}^{D} L_d$ is the Cholesky decomposition of the matrix $K = \bigotimes_{d=1}^{D} K_d$.*

Property A.11. (Eigendecomposition) *If $Q_d \Lambda_d Q_d^T$ is the eigendecomposition of K_d, then $Q \Lambda Q^T$ is the eigendecomposition of the matrix $K = \bigotimes_{d=1}^{D} K_d$, where $Q = \bigotimes_{d=1}^{D} Q_d$ and $\Lambda = \bigotimes_{d=1}^{D} \Lambda_d$.*

Property A.12. (Differentiation)*For any set matrices K_d and any variable θ one can differentiate the their Kronecker product with respect to θ as follows:*

$$\frac{\partial(\bigotimes_{d=1}^{D} K_d)}{\partial \theta} = \sum_{d=1}^{D} (\bigotimes_{m=1}^{d-1} K_m) \bigotimes \frac{\partial K_d}{\partial \theta} \bigotimes (\bigotimes_{n=d+1}^{D} K_n) \ . \tag{A.2}$$

Property A.13. (Diagonalisation) *If $K = \bigotimes_{d=1}^{D} K_d$, then also $diag(K) = \bigotimes_{d=1}^{D} diag(K_d)$.*

Appendix B

The BFGS Algorithm

B.1 The Newton method

The BFGS algorithm derivations to follow are based upon the material presented in [35]. We loosely follow the organisation of [35], deriving BFGS as a special case of the Quasi-Newton family of algorithms which are in turn derived as computational optimization of the Newton method. For the rest of this subsection the Newton method is presented similarly to [35, Chapter 2.2]. The Newton method is an iterative algorithm for function optimisation that given initial input location x_0 and function f, each iteration computes a point x_i, such that $f(x_i) \geq f(x_{i+1})$ for all $i \geq 0$. The Newton method assumes f is two-times differentiable and is guaranteed to converge to a local minimum of the function. The Newton method uses the second-order Taylor series approximation:

$$f(x) \approx f(a) + \nabla f(a)(x-a) + \frac{1}{2}(x-a)^T(\nabla^2 f(a))(x-a), \text{ as } x \to a \qquad \text{(B.1)}$$

Substituting $x_{n+1} = x_n + \Delta x$ for x and x_n for a in Equation B.1 one can obtain:

$$f(x_{n+1}) \approx f(x_n) + \nabla f(x_n)\Delta x + \frac{1}{2}\Delta x^T(\nabla^2 f(x_n))\Delta x, \text{ as } \Delta x \to 0 \qquad \text{(B.2)}$$

For notational convenience $h_n(\Delta x) = f(x_n) + \nabla f(x_n)\Delta x + \frac{1}{2}\Delta x^T(\nabla^2 f(x_n))\Delta x$. To minimize f it is reasonable at each iteration n to choose the direction Δx in which f decreases most rapidly. This is hard to achieve for arbitrary function f. Knowing $f(x_{n+1}) \approx h_n(\Delta x)$ for small enough Δx, one can select the direction Δx in which h_n decreases most rapidly, instead. One can obtain that direction

analytically by differentiating h_n:

$$\Delta x = -H_n^{-1} g_n \tag{B.3}$$

, where $H_n = \nabla^2 f(x_n)$ is the Hessian matrix of f and $g_n = \nabla f(x_n)$ is the gradient of f. Since $f(x_{n+1}) \approx h_n(\Delta x)$ only for small enough Δx, one usually uses only the direction of the vector Δx to come up with update for x_{n+1}:

$$x_{n+1} = x_n - \alpha(H_n^{-1} g_n), \text{ where} \tag{B.4}$$

$$\alpha = \min_{\alpha \geq 0} f(x_n - \alpha H_n^{-1} g_n) \tag{B.5}$$

Equation B.5 is usually solved via line search methods. It is recommended that the line search obeys the Wolfe conditions [35, Chapter 6.1]. The complete algorithm is given in Algorithm 4 below:

Algorithm 4 Newton method

```
function NewtonMethod(f, x_0)
    for n = 0, 1, ... do                          ▷ until converged
        Calculate g_n, H_n^{-1}
        d = H_n^{-1} g_n
        α = min_{α≥0} f(x_n − αd)
        x_{n+1} = x_n − αd
    end for
end function
```

B.2 Quasi-Newton methods

The Newton method requires the calculation of the inverse Hessian matrix H_n^{-1} of f each iteration. That imposes $\mathcal{O}(nm^3)$ computational cost and $\mathcal{O}(m^2)$ memory storage for the Newton method where m is the number of input dimensions of f. For functions whose input is highly multidimensional like the marginal likelihood functions presented in this document that makes the Newton method computationally impractical or even infeasible. The Quasi-Newton methods trade the quadratic convergence rate of the Newton method for computational gains. The resulting algorithms are very efficient to compute and still achieve sub-linear

convergence rate [35, Chapter 2.2]. That is achieved by maintaining a Hessian approximation that the Quasi-Newton family of algorithms updates every iteration by adding some low-rank matrix. The Quasi-Newton algorithms depend on the Secant Condition to choose their low-rank matrix update. The Secant Condition is derived by imposing the condition that the gradients of $f(x_{n+1})$ and $h_n(\Delta x)$ must agree at x_n and x_{n-1}:

$$\begin{aligned} \nabla h_n(x_n) &= g_n \\ \nabla h_n(x_{n-1}) &= g_{n-1} \end{aligned} \tag{B.6}$$

Solving Equation B.6 results in:

$$\begin{aligned} H_n^{-1} y_n &= s_n, \text{ where} \\ s_n &= x_{n+1} - x_n \\ y_n &= g_{n+1} - g_n \end{aligned} \tag{B.7}$$

As evident from Equation B.7, any Quasi-Newton Hessian update depends on the quantities $s_n = x_{n+1} - x_n$ and $y_n = g_{n+1} - g_n$. The exact form of the Hessian update differs from one Quasi-Newton algorithm to another but it is usually dependent only on the Hessian approximation in the previous step H_{n-1}, as well as s_n and y_n. The high level pseudocode for all Quasi-Newton algorithms is given in Algorithm 5 below:

Algorithm 5 Quasi-Newton methods

function QUASI-NEWTON($f, x_0, H_0^{-1},$)
    $g_0 = \nabla f(x_0)$
    for $n = 0, 1, \ldots$ **do** ▷ until converged
        $d = H_n^{-1} g_n$
        $\alpha = \min_{\alpha \geq 0} f(x_n - \alpha d)$
        $x_{n+1} = x_n - \alpha d$
        $g_{n+1} = \nabla f(x_{n+1})$
        $s_{n+1} = x_{n+1} - x_n$
        $y_{n+1} = g_{n+1} - g_n$
        $H_{n+1}^{-1} = \text{QUASI-UPDATE}(H_n^{-1}, s_{n+1}, y_{n+1})$
    end for
end function

B.3 BFGS

As already discussed, BFGS is a Quasi-Newton method. Thus, to fully specify it one needs to choose a Hessian updating policy satisfying the Secant Condition. The Secant Condition is not enough to obtain an update policy by itself, since H_{n+1}^{-1} is undetermined and therefore additional restrictions should be posed. To overcome the problem the BFGS creators adopted the smallest change principle that prefers solutions with smaller update cost. The update cost is measured in terms of the weighted Frobenius distance measure between H_{n+1}^{-1} and H_n^{-1}. Since the inverse of any Hessian matrix is symmetric, the BFGS algorithm chooses to enforce H_{n+1}^{-1} to be symmetric, as well to arrive at the following system:

$$\begin{aligned} H_{n+1}^{-1} = argmin \ \ & \|H_{n+1}^{-1} - H_n^{-1}\|^2 \\ \text{s.t.} \quad & H_{n+1}^{-1} y_n = s_n \\ & H_{n+1}^{-1} \text{ is symmetric} \end{aligned} \tag{B.8}$$

System B.8 has a unique solution resulting in our update policy below:

$$\begin{aligned} H_{n+1}^{-1} = (I - \rho_n s_n y_n^T) H_n^{-1} (I - \rho_n y_n s_n^T) + \rho_n s_n s_n^T, \text{ where} \\ \rho_n = \frac{1}{y_n^T s_n} \end{aligned} \tag{B.9}$$

Bibliography

[1] Jyri J. Kivinen and Christopher K. I. Williams. Multiple Texture Boltzmann Machines. In Neil D. Lawrence and Mark Girolami, editors, *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2012, La Palma, Canary Islands, April 21-23, 2012*, volume 22 of *JMLR Proceedings*, pages 638–646. JMLR.org, 2012. URL `http://jmlr.csail.mit.edu/proceedings/papers/v22/kivinen12.html`.

[2] Christine Guillemot and Olivier Le Meur. Image inpainting: Overview and recent advances. *Signal Processing Magazine, IEEE*, 31(1):127–144, 2014.

[3] S. Ravi, P. Pasupathi, S. Muthukumar, and N. Krishnan. Image in-painting techniques - a survey and analysis. In *Innovations in Information Technology (IIT), 2013 9th International Conference on*, pages 36–41, Abu Dhabi, United Arab Emirates, March 2013. Institute of Electrical and Electronics Engineers (IEEE). doi: 10.1109/Innovations.2013.6544390.

[4] James Michael Coggins. *A Framework for Texture Analysis Based on Spatial Filtering*. PhD thesis, East Lansing, MI, USA, 1983. AAI8315444.

[5] Louis-François Pau, Patrick Shen-pei Wang, and Shen-pei Wang. *Handbook of Pattern Recognition & Computer Vision*. World Scientific Publishing Company Incorporated, 1999.

[6] Jian Sun, Lu Yuan, Jiaya Jia, and Heung-Yeung Shum. Image completion with structure propagation. In *ACM Transactions on Graphics (ToG)*, volume 24, pages 861–868. ACM, 2005.

[7] Manuel M Oliveira Brian Bowen Richard and McKenna Yu-Sung Chang. Fast digital image inpainting. In *Appeared in the Proceedings of the Inter-*

national Conference on Visualization, Imaging and Image Processing (VIIP 2001), Marbella, Spain, pages 106–107, 2001.

[8] Li-Yi Wei. Texture synthesis by fixed neighborhood searching. 2002.

[9] Bela Julesz, EN Gilbert, LA Shepp, and HL Frisch. Inability of humans to discriminate between visual textures that agree in second-order statistics—revisited. *Perception*, 2(4):391–405, 1973.

[10] Russell L De Valois, Duane G Albrecht, and Lisa G Thorell. Spatial frequency selectivity of cells in macaque visual cortex. *Vision research*, 22(5): 545–559, 1982.

[11] Elizaveta Levina and Peter J Bickel. Texture synthesis and nonparametric resampling of random fields. *The Annals of Statistics*, pages 1751–1773, 2006.

[12] Nicolas Heess, Christopher KI Williams, and Geoffrey E Hinton. Learning generative texture models with extended fields-of-experts. In *BMVC*, pages 1–11, 2009.

[13] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2005. ISBN 026218253X.

[14] Richard von Mises. *Mathematical Theory of Probability and Statistics*. Academic Press, 1964. ISBN 978-1-4832-3213-3.

[15] Charles S Bos. A comparison of marginal likelihood computation methods. In *Compstat*, pages 111–116. Springer, 2002.

[16] C Chatfield. Time series analysis: an introduction. *Chapman & Hall, London*, 1989.

[17] Andrew Gordon Wilson, Elad Gilboa, Arye Nehorai, and John P Cunningham. GPatt: Fast multidimensional pattern extrapolation with Gaussian processes. *arXiv preprint arXiv:1310.5288*, 2013.

[18] Yunus Saatçi. *Scalable inference for structured Gaussian process models*. PhD thesis, University of Cambridge, 2012.

[19] Claude Tadonki and Bernard Philippe. *Parallel Multiplication of a vector by a Kronecker tensor product of matrices*. IRISA, 1998.

[20] Yuancheng Luo and Ramani Duraiswami. Fast near-grid gaussian process regression. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 424–432, 2013.

[21] Andrew Wilson, Elad Gilboa, John P Cunningham, and Arye Nehorai. Fast kernel learning for multidimensional pattern extrapolation. In *Advances in Neural Information Processing Systems*, pages 3626–3634, 2014.

[22] Christopher Williams and Matthias Seeger. Using the Nyström method to speed up kernel machines. In *Proceedings of the 14th Annual Conference on Neural Information Processing Systems*, number EPFL-CONF-161322, pages 682–688, 2001.

[23] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[24] P Brodatz. Textures: a photographic album for artists and designers vol. 66: Dover new york. 1966.

[25] Safia Abdelmounaime and He Dong-Chen. New brodatz-based image databases for grayscale color and multiband texture analysis. *ISRN Machine Vision*, 2013, 2013.

[26] Carl Rasmussen and Hannes Nickisch. GPML:Gaussian Process regression and classification toolbox. `http://www.gaussianprocess.org/gpml/code/matlab/doc/`, 2005-2015.

[27] Jens Keiner, Stefan Kunis, and Daniel Potts. Nfft 3.0-tutorial. *Chemnitz University of Technology, Department of Mathematics, Chemnitz, Germany*, 2009.

[28] Axel Ruhe. Numerical linear algebra : Experimental assignments. `https://www.nada.kth.se/~ruhe/2D5219/labs2007.html`, 2007.

[29] Aroh Barjatya. Binary search. `http://www.mathworks.com/matlabcentral/fileexchange/7552-binary-search/content/bsearch.m`, 2005.

[30] Jan Simon. VChooseK. `http://www.mathworks.com/matlabcentral/fileexchange/26190-vchoosek/content/VChooseK.m`, 2009.

[31] Andrew Wilson. Structure discovery tutorial. `http://www.cs.cmu.edu/~andrewgw/pattern/index.html`, 2013-2015.

[32] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, 2004.

[33] Edmond Chow and Yousef Saad. Preconditioned methods for sampling multivariate gaussian distributions. In *Householder Symposium XIX June 8-13, Spa Belgium*, page 45. Citeseer, 2013.

[34] Huamin Zhang and Feng Ding. On the Kronecker products and their applications. *J. Applied Mathematics*, 2013:296185:1–296185:8, 2013.

[35] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.