Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Identification of Gaussian Process State-Space Models

Author:
William Sternberg

Supervisor:
Marc Peter Deisenroth

Submitted in partial fulfilment of the requirements for the MSc degree in Computing (Machine Learning) of Imperial College London

September 2017

Abstract

Gaussian process state-space models (GPSSMs) have been shown to be a competitive way of learning from time series due to their ability to identify complex systems; as a result, GPSSMs have the potential to be applied in areas such as Economics, Engineering and Physics. However, the current techniques for learning GPSSMs suffer from a variety of issues; for example, some methods have long training times or constraints on expressiveness while other methods have difficulties in dealing with high-dimensional latent spaces.

We provide an overview of all the key background material and focus on learning GPSSMs using a recent method that combines Hilbert reduced-rank Gaussian processes and sequential Monte Carlo. We also present several novel contributions, which add additional features to this recent method; for example, online learning, distributed learning, learning under Student-t noise and learning with Student-t processes. Additionally, we improve the Hilbert reduced-rank Gaussian process model using neural networks and state-of-the-art MCMC methods. This gives us a significant improvement in training times for high-dimensional inputs compared with the original Hilbert reduced-rank model. Finally, we use our improved reduced-rank Gaussian process model to create a novel GPSSM. We demonstrate all these contributions in a range of examples.

Contents

Chapter 1

Introduction

Consider a sequence of time-ordered observations $y_1, ..., y_T$. Our aim is to learn something about the system which has generated these observations and to predict future observations. Most interesting systems contain noise and this means that we cannot predict the future observations exactly even if we have perfect knowledge of the system; however, we can search for a predictive distribution. This distribution should ideally take into account all sources of uncertainty; for example, the system noise and the uncertainty in any estimated parameters. Although we are looking for a predictive distribution, we can compute statistics of this distribution such as the mean and variance in order to provide a point prediction and a confidence interval. The idea is that, though the point prediction might not be correct, the true value should have a high probability of falling inside the confidence interval.

As an example, consider the following autoregressive model known as an AR(1) model:

$$y_{t+1} = 0.5y_t + v_t \tag{1.1}$$

$$v_t \overset{iid}{\sim} \mathcal{N}(0, 1) \tag{1.2}$$

Despite the fact that we know all the parameters of this system, given $y_{1:T} = y_1, ..., y_T$ we cannot predict y_{T+1} with certainty; however, there does exist an optimal prediction. Let $\hat{y}_{T+1}$ be our prediction of y_{T+1} given $y_{1:T}$, then the optimal $\hat{y}_{T+1}$ is [Hamilton, 1994]:

$$\hat{y}_{T+1} = \mathbb{E}[y_{T+1}] \tag{1.3}$$

$$= 0.5y_T \tag{1.4}$$

Although (1.4) is the optimal prediction of y_{T+1}, the true value might be far away from this prediction and that is why we also wish to provide a distribution over the possible values of y_{T+1}. In this example, we can calculate the optimal predictive distribution as:

$$y_{T+1}|y_{1:T} \sim \mathcal{N}(0.5y_T, 1) \tag{1.5}$$

Now, $y_{T+1}|y_{1:T} \sim \mathcal{N}(0.5y_T, 100)$ is also a *valid* predictive distribution but it is not the optimal one because it has more uncertainty than necessary.

Suppose next that we have observations $y_1, ..., y_T$ from the AR(1) model:

$$y_{t+1} = \alpha y_t + v_t \tag{1.6}$$

$$v_t \overset{iid}{\sim} \mathcal{N}(0, \sigma^2) \tag{1.7}$$

We could try and estimate the parameters α and σ^2 using the data $y_1, ..., y_T$ and then provide a predictive distribution of y_{T+1} by assuming that the estimated parameters have no uncertainty; for example:

$$y_{T+1}|y_{1:T}, \hat{\alpha}, \hat{\sigma}^2 \sim \mathcal{N}(\hat{\alpha} y_T, \hat{\sigma}^2) \tag{1.8}$$

where $\hat{\alpha}$ and $\hat{\sigma}^2$ is our estimate of the parameters α and σ^2 respectively. However, this predictive distribution is overconfident and we should try to take into account the uncertainty that we have in the model parameters in order to get a more accurate predictive distribution. One way to do this is to use Bayesian statistics. If we assume that we have some prior distributions for α and σ^2:

$$\alpha \sim p(\alpha) \tag{1.9}$$

$$\sigma^2 \sim p(\sigma^2) \tag{1.10}$$

then, under the Bayesian framework, we can find the posterior distributions of the parameters given the data using Bayes' Theorem:

$$p(\alpha, \sigma^2|y_{1:T}) = \frac{p(y_{1:T}|\alpha, \sigma^2)p(\alpha)p(\sigma^2)}{p(y_{1:T})} \tag{1.11}$$

where we have assumed the prior distributions are independent. Using this, we can find a posterior predictive distribution of $y_{T+1}|y_{1:T}$ which takes into account the uncertainty in the model parameters:

$$p(y_{T+1}|y_{1:T}) = \iint p(y_{T+1}|y_{1:T}, \alpha, \sigma^2)p(\alpha, \sigma^2|y_{1:T})d\alpha d\sigma^2 \tag{1.12}$$

$$= \iint p(y_{T+1}|y_T, \alpha, \sigma^2)\frac{p(y_{1:T}|\alpha, \sigma^2)p(\alpha)p(\sigma^2)}{p(y_{1:T})}d\alpha d\sigma^2 \tag{1.13}$$

$$= \frac{1}{p(y_{1:T})}\iint \mathcal{N}(\alpha y_T, \sigma^2)p(y_{1:T}|\alpha, \sigma^2)p(\alpha)p(\sigma^2)d\alpha d\sigma^2 \tag{1.14}$$

As you can see, it can get complicated quickly and with Bayesian statistics you often have intractable integrals: we will look at methods for dealing with these in Chapter 2.

Why look at Gaussian process state-space models?
So far we have been focusing on predictive distributions rather than point estimates, and one of our many reasons for pursuing Gaussian process state-space models (GPSSMs) is that they naturally fit into this Bayesian framework. Furthermore, the more recent techniques for learning GPSSMs provide predictive distributions that take into account many different sources of uncertainty: from the observation

and latent noise, to the uncertainty in the latent states and kernel hyperparameters. Moreover, Gaussian process state-space models have be shown; for example, in Frigola [2015], to be a competitive method of learning from time series data. We believe there are two main reasons for this; firstly, time series data (especially if highly autocorrelated) does not contain as much information as the same amount of independent data but it is known that Gaussian processes are efficient learners in low data environments [Rasmussen and Williams, 2006]. Secondly, Gaussian processes are very flexible and this makes them ideal for putting inside a state-space model since often the interesting time series are rather complicated.

Other methods of learning time-ordered data:
There are many techniques other than GPSSMs available in the machine learning and statistical literature for dealing with time series; here, we will briefly discuss a few of them. One of the simplest models is the autoregressive AR model, this is linear and it can learn simple systems via maximum likelihood or expectation-maximisation [Hamilton, 1994]. Other linear models include moving-average models (MA), autoregressive moving-average models (ARMA) and autoregressive integrated moving-average models (ARIMA). The key points for all the above linear models are that; firstly, they can be written as state-space models; secondly, they can only learn simple systems and thirdly, traditional forecasting methods for these models only take into account the system noise and do not include the uncertainty in the estimated parameters. More complex non-linear models include Generalized Autoregressive Conditional Heteroskedasticity (GARCH) models which allow for stochastic variance, and Holt-Winters/Exponential Smoothing models which are good at forecasting simple systems. However, none of these models are particularly expressive and they have difficulties dealing with high-dimensional data. A final example is recurrent neural networks, they have been shown to learn some complex systems but have long training times and only give point estimates. Overall, there seems to be a lack of fully Bayesian flexible models and GPSSMs aim to fill this gap.

The layout of this thesis is as follows, first we look at the key background material required to understand GPSSMs such as Gaussian processes, reduced-rank Gaussian processes, sequential Monte Carlo and state-space models. Then, we present our novel contributions with examples throughout to compare and contrast our new models with the previous ones. A summary of our contributions is stated below.

Contributions

1. Created a new covariance function called the *Adaptive* covariance which is particularly suited to the Hilbert reduced-rank GP of Solin and Särkkä [2014] and useful for situations in which it is difficult to choose a good covariance function.

2. Combined the mGP of Calandra et al. [2016] with the Hilbert reduced-rank GP of Solin and Särkkä [2014] to create the manifold-Hilbert reduced-rank GP and we show how this model retains the properties of the original mGP but is much faster due to its use of reduced-rank GPs.

3. Added new features to the model of Svensson et al. [2016] such as learning with Student-t noise, online learning, forgetful online learning, distributed learning and learning with Student-t processes.

4. Improved the Hilbert reduced-rank Gaussian process model of Solin and Särkkä [2014] to allow it to scale much better as the input dimension increases. We call this model the deep Hilbert Gaussian process model (DHGPM).

5. Used the DHGPM to create a new GPSSM that can learn systems with high-dimensional latent spaces quicker than the model of Svensson et al. [2016].

Chapter 2

Background

2.1 Gaussian Processes

Definition 2.1. [Rasmussen and Williams, 2006, p. 13] A Gaussian process (GP) is a collection of random variables, any finite number of which have a joint Gaussian distribution.

As a result of these Gaussian *finite dimensional distributions* we can define a Gaussian process solely in terms of a mean function $m(\cdot)$ and a covariance function $k(\cdot,\cdot)$.

Definition 2.2. [Rasmussen and Williams, 2006] A *scalar-valued covariance function* or *scalar-valued kernel* is a function $k(\boldsymbol{x},\boldsymbol{x}')$ with two inputs $\boldsymbol{x}\in\mathbb{R}^{n_x}$ and $\boldsymbol{x}'\in\mathbb{R}^{n_x}$ such that:

1. $k(\boldsymbol{x},\boldsymbol{x}')$ is real: $k(\boldsymbol{x},\boldsymbol{x}')\in\mathbb{R}$.
2. $k(\boldsymbol{x},\boldsymbol{x}')$ is symmetric: $k(\boldsymbol{x},\boldsymbol{x}')=k(\boldsymbol{x}',\boldsymbol{x})\ \forall\boldsymbol{x},\boldsymbol{x}'\in\mathbb{R}^{n_x}$.
3. $k(\boldsymbol{x},\boldsymbol{x}')$ is positive definite: given any n inputs $\boldsymbol{x}_1,...,\boldsymbol{x}_n$, the matrix $\boldsymbol{K}\in\mathbb{R}^{n_x\times n_x}$ with entries $\boldsymbol{K}_{ij}=k(\boldsymbol{x}_i,\boldsymbol{x}_j)\in\mathbb{R}$ is positive definite. This matrix is known as the covariance or Gram matrix.

An example of a scalar valued covariance function is the squared exponential (SE) covariance function:

$$k(\boldsymbol{x},\boldsymbol{x}')=\exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|^2}{2l^2}\right)\in\mathbb{R} \tag{2.1}$$

where l is a hyperparameter and $\boldsymbol{x},\boldsymbol{x}'\in\mathbb{R}^{n_x}$. See chapter 4 of Rasmussen and Williams [2006] for many more examples of covariance functions.

Definition 2.3. [Álvarez et al., 2012] A *multi-output covariance function* or *multi-output kernel* is a function $k(\boldsymbol{x},\boldsymbol{x}'):\mathbb{R}^{n_x\times n_x}\to\mathbb{R}^{D\times D}$ such that each (d,d') entry of the output matrix is a scalar-valued covariance function with inputs $(\boldsymbol{x},d)$ and $(\boldsymbol{x}',d')$. The (d,d') entry of the output matrix is written as $k(\boldsymbol{x},\boldsymbol{x}')_{d,d'}\in\mathbb{R}$ and we note that we must have $k(\boldsymbol{x},\boldsymbol{x}')_{d,d'}=k(\boldsymbol{x}',\boldsymbol{x})_{d',d}$ since $k(\boldsymbol{x},\boldsymbol{x}')_{d,d'}$ is a covariance function with inputs $(\boldsymbol{x},d)$ and $(\boldsymbol{x}',d')$.

An example of a multi-output covariance function is:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \begin{pmatrix} k(\boldsymbol{x}, \boldsymbol{x}')_{1,1} & k(\boldsymbol{x}, \boldsymbol{x}')_{1,2} \\ k(\boldsymbol{x}, \boldsymbol{x}')_{2,1} & k(\boldsymbol{x}, \boldsymbol{x}')_{2,2} \end{pmatrix} \tag{2.2}$$

$$k(\boldsymbol{x}, \boldsymbol{x}')_{d,d'} = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2l_1^2} \right) \exp\left(-\frac{(d - d')^2}{2l_2^2} \right) \in \mathbb{R} \tag{2.3}$$

where l_1, l_2 are hyperparameters and $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^{n_x}$. See Álvarez et al. [2012] for many more examples of multi-output covariance functions.

Although multi-output covariance functions are not that popular, we are interested in them because of their application to multi-output Gaussian processes which are particularly important for Gaussian process state-space models. It is possible to avoid using multi-output kernels but we would like to investigate whether they could be useful.

Notation:
Let f be a Gaussian process (GP) with mean function $m(\cdot)$ and covariance function $k(\cdot, \cdot)$, then f is written as:

$$f \sim \mathcal{GP}(m, k) \tag{2.4}$$

Given a single input $\boldsymbol{x} \in \mathbb{R}^{n_x}$, we have that $f(\boldsymbol{x}) \in \mathbb{R}^D$ and we will write $f_d(\boldsymbol{x}) \in \mathbb{R}$ for the dth component of $f(\boldsymbol{x})$. Also, given a single input $\boldsymbol{x}$, the mean function $m(\boldsymbol{x})$ is related to $f(\boldsymbol{x})$ by:

$$m(\boldsymbol{x}) = \mathbb{E}[f(\boldsymbol{x})] = \mathbb{E}\begin{bmatrix} f_1(\boldsymbol{x}) \\ \vdots \\ f_D(\boldsymbol{x}) \end{bmatrix} \in \mathbb{R}^D \tag{2.5}$$

Given two inputs $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^{n_x}$, the covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$ is related to $f(\boldsymbol{x})$ by:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \operatorname{cov}(f(\boldsymbol{x}), f(\boldsymbol{x}')) = \mathbb{E}[(f(\boldsymbol{x}) - m(\boldsymbol{x}))(f(\boldsymbol{x}') - m(\boldsymbol{x}'))^T] \in \mathbb{R}^{D \times D} \tag{2.6}$$

where the matrix $k(\boldsymbol{x}, \boldsymbol{x}')$ has components [Álvarez et al., 2012]:

$$k(\boldsymbol{x}, \boldsymbol{x}')_{d,d'} = \operatorname{cov}(f_d(\boldsymbol{x}), f_{d'}(\boldsymbol{x}')) \in \mathbb{R} \tag{2.7}$$

In the general case, there are no independence assumptions in (2.7) and so it is a dense matrix; however, as we shall see in section 2.1.2 making certain assumptions about this matrix can reduce the computational cost.

The notation in (2.4) is used to represent the joint distribution of the random variables $f(\boldsymbol{x})$ at all possible inputs $\boldsymbol{x} \in \mathbb{R}^{n_x}$. However, if we take a finite number of inputs $\boldsymbol{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_N] \in \mathbb{R}^{n_x \times N}$, then by the definition of Gaussian process we have that:

$$\operatorname{vec}(f(\boldsymbol{X})) \sim \mathcal{N}(\operatorname{vec}(m(\boldsymbol{X})), K(\boldsymbol{X}, \boldsymbol{X})) \tag{2.8}$$

where $\mathrm{vec}(\boldsymbol{Z})$ is the vectorisation of the matrix $\boldsymbol{Z}$ (an operation which stacks the columns of $\boldsymbol{Z}$ on top of each other to form a vector), $f(\boldsymbol{X}) = [f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)] \in \mathbb{R}^{D\times N}$, $m(\boldsymbol{X}) = [m(\boldsymbol{x}_1), ..., m(\boldsymbol{x}_N)] \in \mathbb{R}^{D\times N}$ and

$$K(\boldsymbol{X}, \boldsymbol{X}) = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & k(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1) & k(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & k(\boldsymbol{x}_N, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix} \in \mathbb{R}^{ND\times ND} \tag{2.9}$$

In the case that $D = 1$ we can drop the vec and then this matches the notation in Rasmussen and Williams [2006].

As a example, suppose we are interested in finding the joint distribution of $f(\boldsymbol{x}_1) \in \mathbb{R}^2$ and $f(\boldsymbol{x}_2) \in \mathbb{R}^2$ where $f \sim \mathcal{GP}(m, k)$ and we use a multi-output kernel. In this case, we have that:

$$\mathrm{vec}(f(\boldsymbol{X})) = \begin{pmatrix} f_1(\boldsymbol{x}_1) \\ f_2(\boldsymbol{x}_1) \\ f_1(\boldsymbol{x}_2) \\ f_2(\boldsymbol{x}_2) \end{pmatrix} \tag{2.10}$$

$$\sim \mathcal{N}\left(\begin{bmatrix} m_1(\boldsymbol{x}_1) \\ m_2(\boldsymbol{x}_1) \\ m_1(\boldsymbol{x}_2) \\ m_2(\boldsymbol{x}_2) \end{bmatrix}, \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1)_{1,1} & k(\boldsymbol{x}_1, \boldsymbol{x}_1)_{1,2} & k(\boldsymbol{x}_1, \boldsymbol{x}_2)_{1,1} & k(\boldsymbol{x}_1, \boldsymbol{x}_2)_{1,2} \\ k(\boldsymbol{x}_1, \boldsymbol{x}_1)_{2,1} & k(\boldsymbol{x}_1, \boldsymbol{x}_1)_{2,2} & k(\boldsymbol{x}_1, \boldsymbol{x}_2)_{2,1} & k(\boldsymbol{x}_1, \boldsymbol{x}_2)_{2,2} \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1)_{1,1} & k(\boldsymbol{x}_2, \boldsymbol{x}_1)_{1,2} & k(\boldsymbol{x}_2, \boldsymbol{x}_2)_{1,1} & k(\boldsymbol{x}_2, \boldsymbol{x}_2)_{1,2} \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1)_{2,1} & k(\boldsymbol{x}_2, \boldsymbol{x}_1)_{2,2} & k(\boldsymbol{x}_2, \boldsymbol{x}_2)_{2,1} & k(\boldsymbol{x}_2, \boldsymbol{x}_2)_{2,2} \end{bmatrix} \right) \tag{2.11}$$

2.1.1 Gaussian Process Regression

In this section, we will provide an overview of Gaussian process regression following the derivation in Rasmussen and Williams [2006]. However, we will also explicitly deal with the general case of $f(\boldsymbol{x}) \in \mathbb{R}^D$. In many ways, the general derivation is similar to the univariate case ($D = 1$) and the results are well known but we aim to make clear the link between the different forms of the density functions which can arise in the general case. These different forms are a result of a relationship between the matrix-normal and multivariate-normal distributions.

Consider a set of data $\mathcal{D} = \{(\boldsymbol{y_i}, \boldsymbol{x_i}) \mid i = 1, ..., N\}$ where $\boldsymbol{y}_i \in \mathbb{R}^{n_y}$ and $\boldsymbol{x}_i \in \mathbb{R}^{n_x}$ for $i = 1, ..., N$. In Gaussian process regression, the aim is to use this data $\mathcal{D}$ to find the posterior distribution of a Gaussian process relating inputs $\boldsymbol{x}_i$ to outputs $\boldsymbol{y}_i$ and then to provide a predictive distribution for $\boldsymbol{f}_* = f(\boldsymbol{x}_*)$ given a test input $\boldsymbol{x}_*$. Formally, this means that we have the following system ($i = 1, ..., N$):

$$\boldsymbol{y}_i = f(\boldsymbol{x}_i) + \boldsymbol{\epsilon}_i \tag{2.12a}$$

$$f \sim \mathcal{GP}(m, k) \tag{2.12b}$$

$$\boldsymbol{\epsilon_i} | \boldsymbol{Q} \overset{iid}{\sim} \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}) \tag{2.12c}$$

$$p(\boldsymbol{y}_i | \boldsymbol{f}, \boldsymbol{x}_i, \boldsymbol{\theta}) \sim \mathcal{N}(f(\boldsymbol{x}_i), \boldsymbol{Q}) \tag{2.12d}$$

where $\boldsymbol{Q} \in \mathbb{R}^{n_y \times n_y}$ (usually diagonal) and $\boldsymbol{\theta}$ represents all the model hyperparameters which includes $\boldsymbol{Q}$ and the hyperparameters of kernel k.

We wish to:

1. Learn the optimal hyperparameters $\boldsymbol{\theta}_*$.
2. Find the predictive distribution $p(\boldsymbol{f}_*|\boldsymbol{X}_*, \boldsymbol{Y}, \boldsymbol{X}, \theta_*)$.

where we have $\boldsymbol{Y} = [\boldsymbol{y}_1, ..., \boldsymbol{y}_N] \in \mathbb{R}^{n_y \times N}$, $\boldsymbol{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_N] \in \mathbb{R}^{n_x \times N}$, test inputs $\boldsymbol{X}_* = [\boldsymbol{x}_{*1}, ..., \boldsymbol{x}_{*M}] \in \mathbb{R}^{n_x \times M}$ and function values $\boldsymbol{f}_* = [f(\boldsymbol{x}_{*1}), ..., f(\boldsymbol{x}_{*M})] \in \mathbb{R}^{n_y \times M}$.

Definition 2.4. [Schäcke, 2013, p. 6]. Given matrices $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{Y} \in \mathbb{R}^{p \times q}$ the Kronecker product is defined as:

$$\boldsymbol{X} \otimes \boldsymbol{Y} = \begin{pmatrix} x_{11}\boldsymbol{Y} & x_{12}\boldsymbol{Y} & \cdots & x_{1n}\boldsymbol{Y} \\ x_{21}\boldsymbol{Y} & x_{22}\boldsymbol{Y} & \cdots & x_{2n}\boldsymbol{Y} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1}\boldsymbol{Y} & x_{m2}\boldsymbol{Y} & \cdots & x_{mn}\boldsymbol{Y} \end{pmatrix} \in \mathbb{R}^{mp \times nq} \tag{2.13}$$

It has properties [Schäcke, 2013, pp. 7-9]:

1. $(\boldsymbol{X} \otimes \boldsymbol{Y})(\boldsymbol{Z} \otimes \boldsymbol{T}) = \boldsymbol{XZ} \otimes \boldsymbol{YT}$
   for $\boldsymbol{X} \in \mathbb{R}^{m \times n}$, $\boldsymbol{Y} \in \mathbb{R}^{p \times q}$, $\boldsymbol{Z} \in \mathbb{R}^{n \times r}$ and $\boldsymbol{T} \in \mathbb{R}^{q \times s}$.
2. $(\boldsymbol{X} \otimes \boldsymbol{Y})^{-1} = \boldsymbol{X}^{-1} \otimes \boldsymbol{Y}^{-1}$ if $\boldsymbol{X}$ and $\boldsymbol{Y}$ are invertible.
3. Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ have eigen-decompositions $\boldsymbol{PAP}^{-1}$ and $\boldsymbol{QBQ}^{-1}$ respectively where $\boldsymbol{A}$ and $\boldsymbol{B}$ are diagonal. Then the eigen-decomposition of $\boldsymbol{X} \otimes \boldsymbol{Y}$ is $(\boldsymbol{P} \otimes \boldsymbol{Q})(\boldsymbol{A} \otimes \boldsymbol{B})(\boldsymbol{P} \otimes \boldsymbol{Q})^{-1}$

Definition 2.5. [Gupta and Nagar, 2000] A random matrix $\boldsymbol{X} \in \mathbb{R}^{n \times m}$ is said to have a matrix normal distribution $\mathcal{MN}_{n,m}(\boldsymbol{M}, \boldsymbol{U}, \boldsymbol{V})$ if the density function satisfies:

$$p(\boldsymbol{X}|\boldsymbol{M}, \boldsymbol{U}, \boldsymbol{V}) = \frac{1}{(2\pi)^{\frac{nm}{2}}|\boldsymbol{V}|^{\frac{n}{2}}|\boldsymbol{U}|^{\frac{m}{2}}} \exp\left(-\frac{1}{2} \operatorname{tr}[\boldsymbol{V}^{-1}(\boldsymbol{X} - \boldsymbol{M})^T \boldsymbol{U}^{-1}(\boldsymbol{X} - \boldsymbol{M})] \right) \tag{2.14}$$

where $\boldsymbol{M} \in \mathbb{R}^{n \times m}$, $\boldsymbol{U} \in \mathbb{R}^{n \times n}$, $\boldsymbol{V} \in \mathbb{R}^{m \times m}$ and $|\cdot|$ is the determinant.

An important result [Gupta and Nagar, 2000] is the relationship between the matrix normal and the multivariate normal distributions:

$$\boldsymbol{X} \sim \mathcal{MN}_{n,m}(\boldsymbol{M}, \boldsymbol{U}, \boldsymbol{V}) \iff \operatorname{vec}(\boldsymbol{X}) \sim \mathcal{N}_{nm}(\operatorname{vec}(\boldsymbol{M}), \boldsymbol{V} \otimes \boldsymbol{U}) \tag{2.15}$$

where $\otimes$ is the Kronecker product and $\operatorname{vec}(\boldsymbol{X})$ is the vectorisation of $\boldsymbol{X}$.

Consider $p(\boldsymbol{Y}|f(\boldsymbol{X}), \boldsymbol{\theta})$ where $f(\boldsymbol{X}) = [f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)] \in \mathbb{R}^{n_y \times N}$:

$$p(\boldsymbol{Y}|f(\boldsymbol{X}), \boldsymbol{\theta}) = \prod_{i=1}^{N} p(\boldsymbol{y}_i|f(\boldsymbol{x}_i), \boldsymbol{\theta}) \tag{2.16}$$

$$= \prod_{i=1}^{N} \mathcal{N}(\boldsymbol{y}_i|f(\boldsymbol{x}_i), \boldsymbol{Q}) \tag{2.17}$$

$$\propto \exp(-\frac{1}{2}\sum_{i=1}^{N}(\boldsymbol{y}_i - f(\boldsymbol{x}_i))^T\boldsymbol{Q}^{-1}(\boldsymbol{y}_i - f(\boldsymbol{x}_i))) \tag{2.18}$$

$$= \exp(-\frac{1}{2}[\mathrm{vec}(\boldsymbol{Y}) - \mathrm{vec}(f(\boldsymbol{X}))]^T(\boldsymbol{I}_N \otimes \boldsymbol{Q})^{-1}[\mathrm{vec}(\boldsymbol{Y}) - \mathrm{vec}(f(\boldsymbol{X}))]) \tag{2.19}$$

where $\boldsymbol{I}_N$ is the $N \times N$ identity matrix and to get (2.19) we used a standard result for the Kronecker product: $\boldsymbol{X}^{-1} \otimes \boldsymbol{Y}^{-1} = (\boldsymbol{X} \otimes \boldsymbol{Y})^{-1}$ provided both $\boldsymbol{X}$ and $\boldsymbol{Y}$ are invertible.

From (2.19) we get that:

$$p(\mathrm{vec}(\boldsymbol{Y})|f(\boldsymbol{X}), \boldsymbol{\theta}) = \mathcal{N}(\mathrm{vec}(\boldsymbol{Y})|\,\mathrm{vec}(f(\boldsymbol{X}), \boldsymbol{I}_N \otimes \boldsymbol{Q}) \tag{2.20}$$

And hence by (2.15) we get:

$$p(\boldsymbol{Y}|f(\boldsymbol{X}), \boldsymbol{\theta}) = \mathcal{MN}(\boldsymbol{Y}|f(\boldsymbol{X}), \boldsymbol{I}_N, \boldsymbol{Q}) \tag{2.21}$$

When we talk about the distribution of $\boldsymbol{Y}$ what we are really talking about is the joint distribution of all the components $\boldsymbol{Y}_{ij}$ and so the distributions of $\boldsymbol{Y}$ and $\mathrm{vec}(\boldsymbol{Y})$ must be the same: they contain the same random variables $\boldsymbol{Y}_{ij}$. This means that at any particular test point $\boldsymbol{y}$, the densities $p_{\boldsymbol{Y}}(\boldsymbol{y})$ and $p_{\mathrm{vec}(\boldsymbol{Y})}(\boldsymbol{y})$ are equal (note that the supports are also the same). However, the parametrisations will in general be different; for example $\boldsymbol{Y}$ might follow a matrix-normal distribution and $\mathrm{vec}(\boldsymbol{Y})$ multivariate-normal distribution.

We can use the above results to get $p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\theta})$:

$$p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\theta}) = \int p(\boldsymbol{Y}|f(\boldsymbol{X}), \boldsymbol{\theta})p(f(\boldsymbol{X})|\boldsymbol{\theta})df \tag{2.22}$$

$$= \int \mathcal{N}(\mathrm{vec}(\boldsymbol{Y})|\,\mathrm{vec}(f(\boldsymbol{X})), \boldsymbol{I}_N \otimes \boldsymbol{Q})\mathcal{N}(\mathrm{vec}(f(\boldsymbol{X}))|\,\mathrm{vec}(m(\boldsymbol{X})), K(\boldsymbol{X}, \boldsymbol{X}))df \tag{2.23}$$

$$= \mathcal{N}(\mathrm{vec}(\boldsymbol{Y})|\,\mathrm{vec}(m(\boldsymbol{X})), K(\boldsymbol{X}, \boldsymbol{X}) + \boldsymbol{I}_N \otimes \boldsymbol{Q}) \tag{2.24}$$

To find the hyperparameters we maximise $\log(p(\mathrm{vec}(\boldsymbol{Y})|\boldsymbol{X}, \boldsymbol{\theta}))$ with respect to $\boldsymbol{\theta}$. In the standard marginal-likelihood way, this will find a balance between data fit and model complexity. See Rasmussen and Williams [2006, Ch. 5] for a discussion and detailed overview of how to find the optimal hyperparameters. Since $\log(p(\mathrm{vec}(\boldsymbol{Y})|\boldsymbol{X}, \boldsymbol{\theta}))$ is multivariate normal there are no additional technical difficulties over the univariate case; however, there may be more hyperparameters and bigger matrices involved so it can take more time for the optimisation algorithms to complete.

Finally using (2.8) and (2.24) we get:

$$\begin{pmatrix}\mathrm{vec}(\boldsymbol{f}_*)\\ \mathrm{vec}(\boldsymbol{Y})\end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix}\mathrm{vec}(m(\boldsymbol{X}_*))\\ \mathrm{vec}(m(\boldsymbol{X}))\end{pmatrix}, \begin{pmatrix}K(\boldsymbol{X}_*, \boldsymbol{X}_*) & K(\boldsymbol{X}_*, \boldsymbol{X})\\ K(\boldsymbol{X}, \boldsymbol{X}_*) & K(\boldsymbol{X}, \boldsymbol{X}) + \boldsymbol{I}_N \otimes \boldsymbol{Q}\end{pmatrix}\right) \tag{2.25}$$

where we note that:

$$\begin{aligned} &\operatorname{cov}(\operatorname{vec}(\boldsymbol{f}_*), \operatorname{vec}(\boldsymbol{Y})) && (2.26)\\ &= \operatorname{cov}(\operatorname{vec}(f(\boldsymbol{X}_*)), \operatorname{vec}(f(\boldsymbol{X}))) && (2.27)\\ &= K(\boldsymbol{X}_*, \boldsymbol{X}) \in \mathbb{R}^{Mn_y \times Nn_y} && (2.28)\end{aligned}$$

Hence, via Gaussian conditioning we get $p(\boldsymbol{f}_*|\boldsymbol{X}_*, \boldsymbol{Y}, \boldsymbol{X}, \boldsymbol{\theta})$:

$$p(\boldsymbol{f}_*|\boldsymbol{X}_*, \boldsymbol{Y}, \boldsymbol{X}, \boldsymbol{\theta}) = \mathcal{N}(\operatorname{vec}(\boldsymbol{f}_*)|\boldsymbol{m}_*, \boldsymbol{V}_*) \tag{2.29}$$

with:

$$\boldsymbol{m}_* = \operatorname{vec}(m(\boldsymbol{X}_*)) + K(\boldsymbol{X}_*, \boldsymbol{X})(K(\boldsymbol{X}, \boldsymbol{X}) + \boldsymbol{I}_N \otimes \boldsymbol{Q})^{-1} \operatorname{vec}(\boldsymbol{Y} - m(\boldsymbol{X})) \tag{2.30}$$

$$\boldsymbol{V}_* = K(\boldsymbol{X}_*, \boldsymbol{X}_*) - K(\boldsymbol{X}_*, \boldsymbol{X})(K(\boldsymbol{X}, \boldsymbol{X}) + \boldsymbol{I}_N \otimes \boldsymbol{Q})^{-1} K(\boldsymbol{X}, \boldsymbol{X}_*) \tag{2.31}$$

2.1.2 Reducing the Computational Cost

Both Gaussian process prediction (finding $p(\boldsymbol{f}_*|\boldsymbol{X}_*, \boldsymbol{Y}, \boldsymbol{X}, \boldsymbol{\theta})$) and hyperparameter optimisation (training) involve inverting matrices of size $Nn_y \times Nn_y$ and this requires $\mathcal{O}(N^3 n_y^3)$ operations in general. Therefore, in cases where we have a large amount of training data or in cases with high output dimension n_y, training and prediction can take a long time; furthermore, even storing the $Nn_y \times Nn_y$ matrix in memory might not be possible [Rasmussen and Williams, 2006, p. 171]. In this section, we will briefly discuss various methods for dealing with these issues.

Special multi-output kernels:
Consider the d, d' entry on the submatrix of $K(\boldsymbol{X}, \boldsymbol{X})$ associated with $\boldsymbol{x}, \boldsymbol{x}'$:

$$k(\boldsymbol{x}, \boldsymbol{x}')_{d,d'} \in \mathbb{R} \tag{2.32}$$

This could be any scalar-valued covariance function with inputs: $(\boldsymbol{x}, d)$ and $(\boldsymbol{x}', d')$ [Álvarez et al., 2012, p. 8]. For example, a valid covariance function is the multi-output kernel from (2.3) which as we shall see is known as a separable kernel. Using a covariance function on $(\boldsymbol{x}, d)$ and $(\boldsymbol{x}', d')$ allows us to easily construct $K(\boldsymbol{X}, \boldsymbol{X})$ but this leads to the covariance matrix being of size $Nn_y \times Nn_y$. To reduce the computational cost associated with the output dimension n_y, we can use special covariance functions [Álvarez et al., 2012]. A basic type of special covariance function is a separable covariance function; this is a covariance function of the form:

$$k(\boldsymbol{x}, \boldsymbol{x}')_{d,d'} = k_{\mathcal{X}}(\boldsymbol{x}, \boldsymbol{x}')k_D(d, d') \tag{2.33}$$

where $k_{\mathcal{X}}(\boldsymbol{x}, \boldsymbol{x}') \in \mathbb{R}$ and $k_D(d, d') \in \mathbb{R}$. This leads to:

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{x}') &= k_{\mathcal{X}}(\boldsymbol{x}, \boldsymbol{x}')\boldsymbol{B} && (2.34)\\ K(\boldsymbol{X}, \boldsymbol{X}) &= k_{\mathcal{X}}(\boldsymbol{X}, \boldsymbol{X}) \otimes \boldsymbol{B} && (2.35)\end{aligned}$$

where $\boldsymbol{B} \in \mathbb{R}^{n_y \times n_y}$ is a symmetric, positive definite matrix with entries $\boldsymbol{B}_{ij} = k_D(d_i, d_j)$ for $i, j = 1, ..., N$ and $k_{\mathcal{X}}(\boldsymbol{X}, \boldsymbol{X})$ is a matrix with entries $k_{\mathcal{X}}(\boldsymbol{X}, \boldsymbol{X})_{ij} =$

$k(\boldsymbol{x}_i, \boldsymbol{x}_j) \in \mathbb{R}$ for $i, j = 1, ..., N$ where $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is some scalar-valued covariance function. Immediately we can see that the storage costs have been reduced from $\mathcal{O}(N^2 n_y^2)$ to $\mathcal{O}(N^2 + n_y^2)$ and using standard properties of the Kronecker product we get:

$$K(\boldsymbol{X}, \boldsymbol{X})^{-1} = (k_{\mathcal{X}}(\boldsymbol{X}, \boldsymbol{X}) \otimes \boldsymbol{B})^{-1} = k_{\mathcal{X}}(\boldsymbol{X}, \boldsymbol{X})^{-1} \otimes \boldsymbol{B}^{-1} \tag{2.36}$$

and hence the number of operations for the inversion is reduced from $\mathcal{O}(N^3 n_y^3)$ to $\mathcal{O}(N^3 + n_y^3)$. Using eigen-decompositions it is possible to invert $K(\boldsymbol{X}, \boldsymbol{X}) + \boldsymbol{I}_N \otimes \boldsymbol{Q}$ when $K(\boldsymbol{X}, \boldsymbol{X}) = k_{\mathcal{X}}(\boldsymbol{X}, \boldsymbol{X}) \otimes \boldsymbol{B}$ in $\mathcal{O}(N^3 + n_y^3)$. In the case where $\boldsymbol{Q} = \sigma^2 \boldsymbol{I}$, Álvarez et al. [2012, p. 26] provide a method to do just this by noting:

$$(k_{\mathcal{X}}(\boldsymbol{X}, \boldsymbol{X}) \otimes \boldsymbol{B} + \boldsymbol{I}_N \otimes \sigma^2 \boldsymbol{I}_{n_y})^{-1} \operatorname{vec}(\boldsymbol{Y}) \tag{2.37}$$

$$= \sum_{d=1}^{n_y} \boldsymbol{u}_d [(\rho_d k_{\mathcal{X}}(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2 \boldsymbol{I}_N)^{-1} \overline{y}^d]^T \tag{2.38}$$

where $\boldsymbol{Y} = [\boldsymbol{y}_1, ..., \boldsymbol{y}_N]$ and $\rho_d, \boldsymbol{u}_d$ for $d = 1, ..., n_y$ are the eigenvalues/eigenvectors of $\boldsymbol{B}$ and $\overline{y}^d = (\boldsymbol{y}_1^T \boldsymbol{u}_d, ..., \boldsymbol{y}_N^T \boldsymbol{u}_d) \in \mathbb{R}^N$ for $d = 1, ..., n_y$. This can also be applied in the case where $\operatorname{vec}(\boldsymbol{Y})$ is replaced by $K(\boldsymbol{X}, \boldsymbol{X}_*)$ by using the same method on each of the columns of $K(\boldsymbol{X}, \boldsymbol{X}_*)$.

One thing we have not mentioned is how to find $\boldsymbol{B}$, GPy [2012] suggests setting $\boldsymbol{B} = \boldsymbol{W}\boldsymbol{W}^T + \epsilon \boldsymbol{I}$ ($\epsilon > 0$) and then viewing $\boldsymbol{W}$ and ϵ as covariance hyperparameters finding them in the usual way by maximising the marginal likelihood. This way is good because it ensures $\boldsymbol{B}$ is positive definite and if $\boldsymbol{W} \in \mathbb{R}^{n_y}$ then there are only $n_y + 1$ additional hyperparameters to find.

Although we motivated these special covariance functions as a means to reduce computational cost, there are alternative constructions: the linear model of coregionalization (LMC) leads to the sums of separable covariances and a special case known as the intrinsic coregionalisation model (ICM) leads to the separable covariance case we have looked at above. Furthermore, it is possible to extend Gaussian process regression to the case where each output dimension has different training data and all the above is discussed in depth by Álvarez et al. [2012]. Sometimes the special covariance functions use techniques similar to the sparse GPs discussed below and introduce inducing point variables. It is worth pointing out that designing these special covariance functions is not only about computational cost (most sparse GP techniques can be used on top of the special covariance function if necessary) but also about creating covariance functions that can express the relationship between multiple output dimensions sufficiently; for example, see Álvarez and Lawrence [2009].

Independent GPs for each output dimension:
Consider the dataset $\mathcal{D} = \{(\boldsymbol{y}_i, \boldsymbol{x}_i) \mid i = 1, ..., N\}$ with $\boldsymbol{y_i} \in \mathbb{R}^{n_y}$ and $\boldsymbol{x_i} \in \mathbb{R}^{n_x}$. The independent GP for each output dimension approach splits $\mathcal{D}$ up into datasets $\mathcal{D}_j = \{(y_{ij}, \boldsymbol{x}_i) \mid i = 1, ..., N\}$ for $j = 1, ..., n_y$ where $y_{ij} \in \mathbb{R}$ is the j^{th} component of $\boldsymbol{y}_i$. This approach then trains independent univariate-output GPs on each of the

datasets $\mathcal{D}_j$. This removes the issue of multi-output kernels since all kernels will now be scalar-valued and this approach automatically reduces training times to $\mathcal{O}(n_y N^3)$. Moreover, by performing operations in parallel and reusing parts one can *effectively* reduce the training time to $\mathcal{O}(N^3)$.

A question that should be asked is whether the computational expense of multi-output covariance functions is necessary and worth it. The reason why we might wish to consider multi-output covariance functions is that the different dimensions can give information about each other: in other words the outputs are correlated.

Consider the following example:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} f_1(x) \\ f_2(x) \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \end{pmatrix} = \begin{pmatrix} \tanh(x) \\ \tanh(x) \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \end{pmatrix} \tag{2.39}$$

where $\epsilon_1, \epsilon_2 \overset{iid}{\sim} \mathcal{N}(0,1)$. Now y_1 and y_2 are statistically independent, and $f_1(x)$, $f_2(x)$ are statistically independent (since x is a constant). That said knowing $f_1(x)$ allows us to completely determine $f_2(x)$ and vice versa. As a result, using a multi-output covariance function will give us a better predicted mean and variance for both $f_1(x)$, $f_2(x)$ compared with independent GPs since we can use information about $f_1(x)$ to understand $f_2(x)$ better and vice versa. To understand why this is the case, notice that using information about y_2 for y_1 provides two samples from $\mathcal{N}(0,1)$ at each x input instead of one and therefore for a single training point x we can use $\frac{y_1+y_2}{2}$ as a better (lower MSE) estimate of f_1 than just y_1. Extending this to D output dimensions with a single training input x and $y_i = \tanh(x) + \epsilon_i; i = 1, .., D$ where $\epsilon_i \overset{iid}{\sim} \mathcal{N}(0,1)$, we see that as $D \to \infty$, we get that $\frac{1}{D}\sum_{i=1}^{D} y_i \overset{a.s.}{\to} \mathbb{E}[y_i] = \tanh(x)$ by the strong law of large numbers.

In some cases, such as the above, we get better results using multi-output kernels; however, in general using independent GPs for each output dimension is a good choice which balances speed with model accuracy. Although it is an approximation, independent GPs can be a competitive alternative to multi-output kernels with the key advantage being a reduced computation cost; for example, see Turner et al. [2010]; Frigola [2015].

Sparse Gaussian Processes:
Previously, we looked at reducing the cost associated with high output dimensions but now we will look at reducing the costs associated with a large amount of training data. One way to do this is to use sparse Gaussian process approximations and an extensive discussion can be found in Quiñonero-Candela and Rasmussen [2005]. The idea is to introduce a set $\boldsymbol{U} = [\boldsymbol{u}_1, ..., \boldsymbol{u}_L]$ of latent (hidden) variables often called inducing variables where $p(\boldsymbol{u}) \sim \mathcal{N}(\boldsymbol{0}, K(\boldsymbol{U}, \boldsymbol{U}))$ and then assume that $\boldsymbol{f}_*$ and $\boldsymbol{f}$ are independent given $\boldsymbol{U}$. This leads to [Quiñonero-Candela and Rasmussen, 2005]:

$$p(\boldsymbol{f}_*, \boldsymbol{f}) = \int p(\boldsymbol{f}_*, \boldsymbol{f}|\boldsymbol{u}) p(\boldsymbol{u}) d\boldsymbol{u} \tag{2.40}$$

$$\approx \int p(\boldsymbol{f}_*|\boldsymbol{u})p(\boldsymbol{f}|\boldsymbol{u})p(\boldsymbol{u})d\boldsymbol{u} \tag{2.41}$$

where $p(\boldsymbol{f}_*, \boldsymbol{f})$ is the joint distribution of the training and test data. The key is to find a good set of inducing variables and the different sparse GP methods introduce different additional approximations on top of (2.41) as well as different methods for choosing the set $\boldsymbol{U}$ of inducing variables [Quiñonero-Candela and Rasmussen, 2005]. Examples include: *The Subset of Regressors*, *The Deterministic Training Conditional Approximation* and *The Fully Independent Conditional Approximation*. These are discussed in Quiñonero-Candela and Rasmussen [2005] and a comparison of their performance in given in Rasmussen and Williams [2006]. Other methods include combining variational and sparse techniques; for example in Titsias [2009]. Furthermore, sometimes the computational cost associated with the amount of training data and the high output dimensions are dealt with in the same algorithm; for example, *Variational Inducing Kernels* [Álvarez et al., 2010].

Hilbert reduced-rank Gaussian Processes:
The Hilbert reduced-rank Gaussian process approximation [Solin and Särkkä, 2014] (which we will also refer to as a reduced-rank Gaussian process) is an important Gaussian process approximation which does not fall under the above sparse GP framework. Instead of using inducing points, the Hilbert reduced-rank GP of Solin and Särkkä [2014] attempts to approximate the kernel. As we shall see, Hilbert reduced-rank GPs can be used to greatly reduce the computation cost and we are particularly interested in this approximation because of its application to GPSSMs.

For now, suppose we have a scalar-valued (i.e. $k(x, x') \in \mathbb{R}$) and *isotropic stationary* covariance function (kernel) (i.e. $k(\boldsymbol{x}, \boldsymbol{x}')$ is a function of $||\boldsymbol{x} - \boldsymbol{x}'||$) with input dimension $n_x = 1$ (i.e. $x, x' \in \mathbb{R}$). Furthermore, suppose that all the training and test data lies in the domain $[-L, L]$ for some $L \in \mathbb{R}$. In this case, the $m \in \mathbb{Z}_{>0}$ basis function approximation of the covariance function (kernel) is [Solin and Särkkä, 2014]:

$$k(x, x') \approx \sum_{j=1}^{m} S(\sqrt{\lambda_j})\phi_j(x)\phi_j(x') \tag{2.42}$$

$$\phi_j(x) = \frac{1}{\sqrt{L}} \sin\left(\frac{\pi j(x + L)}{2L}\right) \tag{2.43}$$

$$\lambda_j = \left(\frac{\pi j}{2L}\right)^2 \tag{2.44}$$

where $x, x' \in \mathbb{R}$, the λ_j and $\phi_j(x)$ are eigenvalues and eigenfunctions of the Laplace operator in the domain $[-L, L]$ and S is the spectral density of the covariance function. This spectral density always exists in the case of a stationary covariance functions (Bochner's Theorem) and is calculated as [Solin and Särkkä, 2014]:

$$S(\boldsymbol{\omega}) = \int k(\boldsymbol{r}) \exp(-i\boldsymbol{\omega}^T\boldsymbol{r})d\boldsymbol{r} \tag{2.45}$$

where $\boldsymbol{r} = ||\boldsymbol{x} - \boldsymbol{x}'||$ and we note that since we have assumed the covariance function is isotropic stationary by definition it is a function of $||\boldsymbol{x} - \boldsymbol{x}'||$. The actual derivation of (2.42)–(2.45) is complex and will not be discussed here but can be found in Solin and Särkkä [2014].

There are two sources of approximation in (2.42); firstly, we only take finitely (m) many eigenvalues/eigenfunctions whereas they are infinity many and secondly, the domain is bounded by $[-L, L]$. The bounding of the domain means that even in the limit as $m \to \infty$ this is still an approximation [Solin and Särkkä, 2014] and also means that we get edge effects (i.e. poor performance) near the domain boundary: we really do need the training and test data to be well within this domain boundary. As an example, consider the univariate RBF (Radial Basis Function) kernel; it turns out that we only need to take around the first 12 eigenfunctions (i.e. $m = 12$) before adding any more has a negligible effect on the approximation accuracy (the eigenvalues decay relatively fast). However, more complex covariance functions do require more eigenvalues/eigenfunctions and performance depends on the decay of the eigenvalues [Svensson et al., 2016; Solin and Särkkä, 2014].

Solin and Särkkä [2014] also give the result for the common case where we have a stationary kernel with inputs $\boldsymbol{x} \in \mathbb{R}^d$ and a domain $\Omega = [-L_1, L_1] \times ... \times [-L_d, L_d]$ with Dirichlet Boundary conditions (i.e. zero on the boundary). Suppose $k(\boldsymbol{x}, \boldsymbol{x}')$ is a stationary kernel and let $\boldsymbol{m} = (m_1, ..., m_d) \in \mathbb{R}^d$ be a vector giving the chosen number of basis functions in each of the d input dimensions, then the $\boldsymbol{m}$-approximation of $k(\boldsymbol{x}, \boldsymbol{x}')$ is [Solin and Särkkä, 2014]:

$$k_{\boldsymbol{m}}(\boldsymbol{x}, \boldsymbol{x}') \approx \sum_{(j_1,...,j_d)=\mathbf{1}}^{\boldsymbol{m}} S(\sqrt{\lambda_{j_1,...,j_d}})\phi_{j_1,...,j_d}(\boldsymbol{x})\phi_{j_1,...,j_d}(\boldsymbol{x}') \tag{2.46}$$

where $\mathbf{1} = (1, ..., 1) \in \mathbb{R}^d$, each j_i can take any integer between 1 and $\max(j_i) = m_i$ inclusive and the eigenvalues are:

$$\lambda_{j_1,...,j_d} = \sum_{k=1}^{d} \left(\frac{\pi j_k}{2L_k} \right)^2 \tag{2.47}$$

and the eigenfunctions are:

$$\phi_{j_1,...,j_d}(\boldsymbol{x}) = \prod_{k=1}^{d} \frac{1}{\sqrt{L_k}} \sin\left(\frac{\pi j_k (x_k + L_k)}{L_k} \right) \tag{2.48}$$

Firstly, note that the number of terms in the sum (2.46) is $\prod_{i=1}^{d} m_i$ and secondly, note that this is a scalar-valued covariance function, so in the case we wish to turn it in a multi-output covariance function we can; for example, apply this approximation to a kernel on $(\boldsymbol{x}, d)$ and $(\boldsymbol{x}', d')$ for $d, d' \in \{1, ..., D\}$ where D is the output dimension.

Given training data $\boldsymbol{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_N]$, we can apply the above results to Gaussian

processes by noting that we can write the covariance matrix $\boldsymbol{K} = K(\boldsymbol{X}, \boldsymbol{X})$ as [Solin and Särkkä, 2014]:

$$\boldsymbol{K} \approx \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^T \tag{2.49}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix with $\boldsymbol{\Lambda}_{\boldsymbol{jj}} = S(\sqrt{\lambda_{\boldsymbol{j}}})$, $\boldsymbol{\Phi}_{i\boldsymbol{j}} = \phi_{\boldsymbol{j}}(\boldsymbol{x}_i)$ for $\boldsymbol{j} = (j_1, ..., j_d)$ and data $\boldsymbol{x}_1, ..., \boldsymbol{x}_N$ with $\boldsymbol{x}_i \in \mathbb{R}^d$.

Solin and Särkkä [2014] were not that clear on what should happen in the case $d > 1$ since writing this out naively will result in complex interactions between high-dimensional objects. To deal with this issue, we introduce a simple method to map these high-dimensional arrays back to two dimensional arrays. The idea is to map each $(j_1, ..., j_d)$ to a unique integer between 1 and $\prod_1^d \max(j_i)$ using the map:

$$\mathcal{M} : (j_1, ..., j_d) \to j \in \left\{1, ..., \prod_{i=1}^{d} \max(j_i)\right\} \tag{2.50}$$

$$\mathcal{M}(j_1, ..., j_d) = 1 + \sum_{i=1}^{d} (j_i - 1) \prod_{k=i+1}^{d} \max(j_k) \tag{2.51}$$

where $\max(j_k)$ is the maximum value each j_k can take and $\prod_{k=d+1}^{d} \max(j_k) = 1$. For example, in the case that we have an input dimension of 5 and each j_i takes a value between 1 and 12 (i.e. 12 eigenfunctions for each dimension) we would map:

$$\mathcal{M}(1, 1, 1, 1, 1) = 1 \tag{2.52}$$
$$\mathcal{M}(1, 1, 1, 1, 2) = 2 \tag{2.53}$$
$$\mathcal{M}(1, 1, 1, 2, 1) = 13 \tag{2.54}$$
$$\mathcal{M}(5, 1, 1, 1, 1) = 82945 \tag{2.55}$$
$$\mathcal{M}(6, 7, 4, 5, 3) = 114531 \tag{2.56}$$
$$\mathcal{M}(12, 1, 1, 1, 1) = 228097 \tag{2.57}$$
$$\mathcal{M}(\boldsymbol{m}) = \mathcal{M}(12, 12, 12, 12, 12) = 248832 = 12^5 \tag{2.58}$$

Going back to (2.49), we can now map each $\boldsymbol{j}$ to an integer using $\mathcal{M}(j_1, ..., j_d)$ so we replace $\boldsymbol{j}$ with $\mathcal{M}(\boldsymbol{j}) = \mathcal{M}(j_1, ..., j_d)$ and hence with $\boldsymbol{j} = (j_1, ..., j_d)$; $\boldsymbol{\Lambda}_{\boldsymbol{jj}}$ becomes $\boldsymbol{\Lambda}_{\mathcal{M}(j_1,...,j_d)\mathcal{M}(j_1,...,j_d)}$ and $\boldsymbol{\Phi}_{i\boldsymbol{j}}$ become $\boldsymbol{\Phi}_{i\mathcal{M}(j_1,...,j_d)}$. This allows us to map the original high dimension objects back to two dimensions and thus leaves $\boldsymbol{\Lambda}$ and $\boldsymbol{\Phi}$ as matrices. The fact that this scales poorly as the number of input dimensions d increases is not as a result of this mapping but inherent to the approximation and in section 3.3 we shall introduce a solution to this problem. Now we understand how to transform these high-dimensional arrays to two dimensions, we can write out the results needed for reduced rank GPs. Given the system (2.12) with $n_y = 1$, $\boldsymbol{Q} = \sigma_n^2$ and a set of test inputs $\boldsymbol{x}_{*_1}, ..., \boldsymbol{x}_{*_T}$ with $\boldsymbol{x}_{*_i} \in \mathbb{R}^d$ we have [Solin and Särkkä, 2014]:

$$\mathbb{E}[f_*] = \boldsymbol{\Phi}^*(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \sigma_n^2\boldsymbol{\Lambda}^{-1})^{-1}\boldsymbol{\Phi}^T\boldsymbol{y} \tag{2.59}$$

$$\mathbb{V}[f_*] = \sigma_n^2\boldsymbol{\Phi}^*(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \sigma_n^2\boldsymbol{\Lambda}^{-1})^{-1}\boldsymbol{\Phi}^{*^T} \tag{2.60}$$

where $\boldsymbol{\Phi}^*_{i\boldsymbol{j}} = \phi_{\boldsymbol{j}}(\boldsymbol{x}_{*_i})$ and $\boldsymbol{j}$ is mapped to an integer using $\mathcal{M}$. Something similar can be done for the marginal likelihood [Solin and Särkkä, 2014, p. 10]. As a result,

the computationally expensive matrix inversions are independent of the size of the training data and the Hilbert reduced-rank GP can train in $\mathcal{O}(m^3)$ where m is the number of terms in the kernel approximation sum (2.46). However, this approximation scales very poorly as the input dimension d increases.

A final thing to note is that we can use the Karhunen-Loève expansion to write [Solin and Särkkä, 2014]:

$$f(\boldsymbol{X}) \approx \sum_{\boldsymbol{j}=\boldsymbol{1}}^{\boldsymbol{m}} f_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{X}) = \boldsymbol{F}\phi(\boldsymbol{X}) \tag{2.61}$$

where the $\phi_{\boldsymbol{j}}$ are the basis functions from (2.46) and the f_j are some constants. If $d > 1$ we use the mapping $\boldsymbol{j} \to \mathcal{M}(j_1, ..., j_d)$.

2.1.3 Examples

In this section, we look at some examples of Gaussian process regression on synthetic data.

Example 1:
One of the most useful properties of covariance functions (kernels) is that summing (or multiplying) two different covariance functions returns another covariance function which has inherited the properties of both the original covariance functions and combined them in an additive (or multiplicative) way. For example, consider training data $\boldsymbol{y}, \boldsymbol{X}$ generated by the system:

$$y_i = f(x_i) + \epsilon_i \tag{2.62a}$$
$$f(x) = x^2 + 3\,\mathrm{sgn}(x) \tag{2.62b}$$
$$\epsilon_i \overset{iid}{\sim} \mathcal{N}(0, 3^2) \tag{2.62c}$$

Using a kernel formed from summing a Rational Quadratic and MLP kernel, Gaussian process regression can learn something close to the true function: see Figure 2.1.

Example 2:
In this example, we compare some different forms of multi-output GP regression using the example we looked at in (2.39) but with four output variables:

$$\begin{pmatrix} y_{i1} \\ y_{i2} \\ y_{i3} \\ y_{i4} \end{pmatrix} = \begin{pmatrix} f_1(x_i) \\ f_2(x_i) \\ f_3(x_i) \\ f_4(x_i) \end{pmatrix} + \begin{pmatrix} \epsilon_{i1} \\ \epsilon_{i2} \\ \epsilon_{i3} \\ \epsilon_{i4} \end{pmatrix} \tag{2.63a}$$
$$f_j(x) = \tanh(x) \tag{2.63b}$$
$$\epsilon_{ij} \overset{iid}{\sim} \mathcal{N}(0, 1^2) \tag{2.63c}$$

All the multi-output models were trained on 60 data points using 10 optimisation restarts and RBF covariance functions. The plots were produced using GPy [2012] and we write the training data as $(\boldsymbol{X}, \boldsymbol{Y})$ where $\boldsymbol{X}$ is the training inputs and $\boldsymbol{Y} = (\boldsymbol{y}^1, ..., \boldsymbol{y}^4)$ is the training outputs with $\boldsymbol{y}^i$ being the training output for the i^{th} output dimension. The results are display in figures 2.2–2.4.

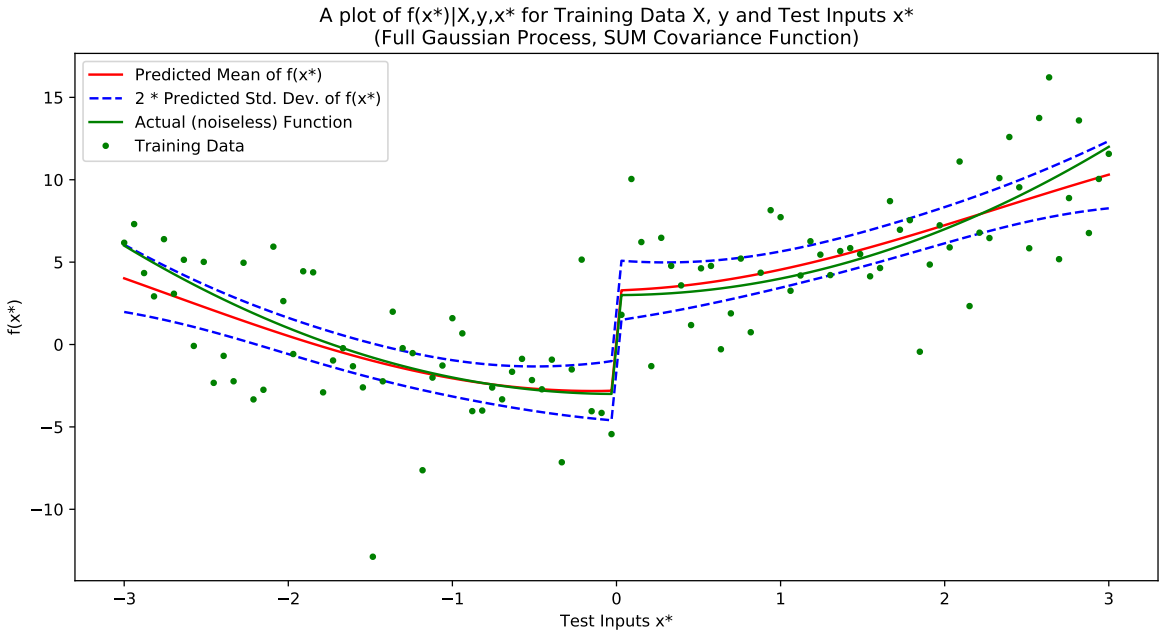

Figure 2.1: Univariate Gaussian process regression with a Rational Quadratic + MLP Kernel and 100 points (x_i, y_i) of training data generated by the system (2.62).

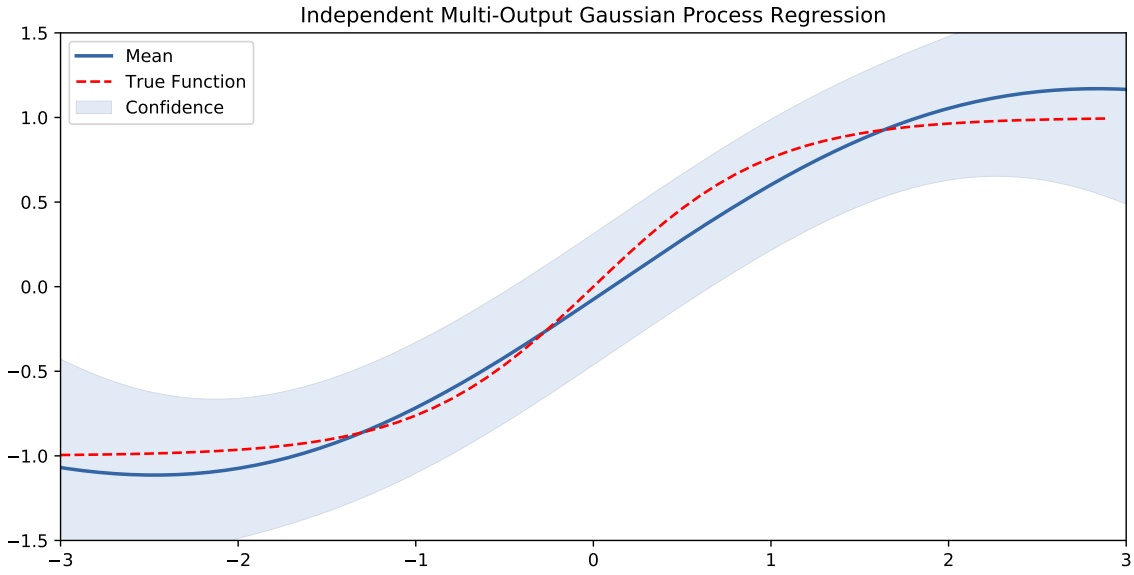

Figure 2.2: A plot of $p(f_1(x_*)|x_*, \boldsymbol{X}, \boldsymbol{y}^1)$ against test inputs x_*. This is the posterior predictive distribution of the first dimension of $f(x)$ in the case of each output dimension having a GP trained independently.

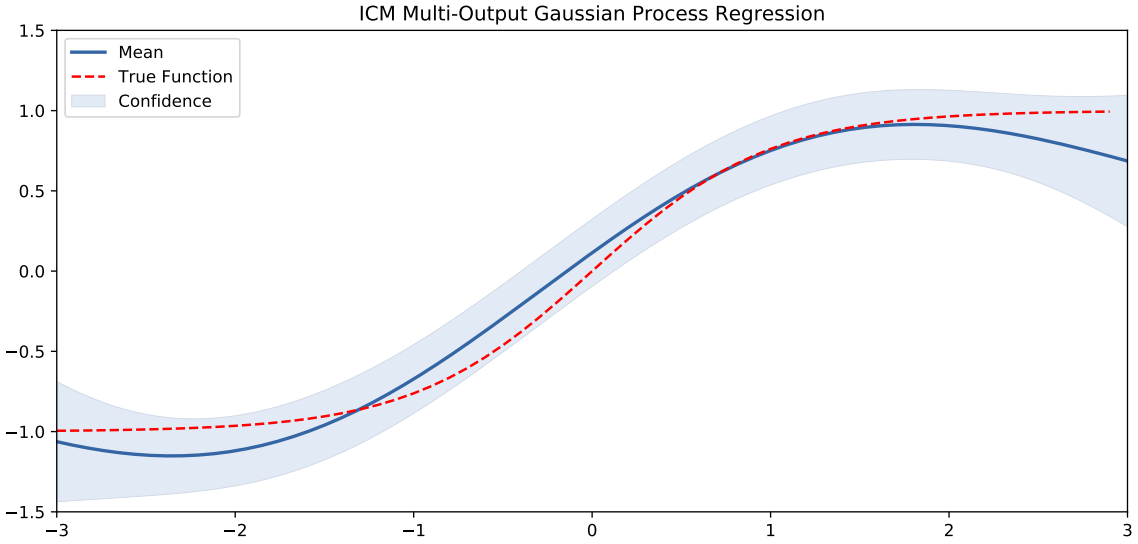

Figure 2.3: A plot of $p(f_1(x_*)|x_*, \boldsymbol{X}, \boldsymbol{Y})$ against test inputs x_* in the case of a GP with a separable multi-output covariance: $k(\boldsymbol{x}, \boldsymbol{x}')_{d,d'} = k_{\mathcal{X}}(\boldsymbol{x}, \boldsymbol{x}')k_D(d, d')$. Notice how much more confident the model is compared to the independent GP case.

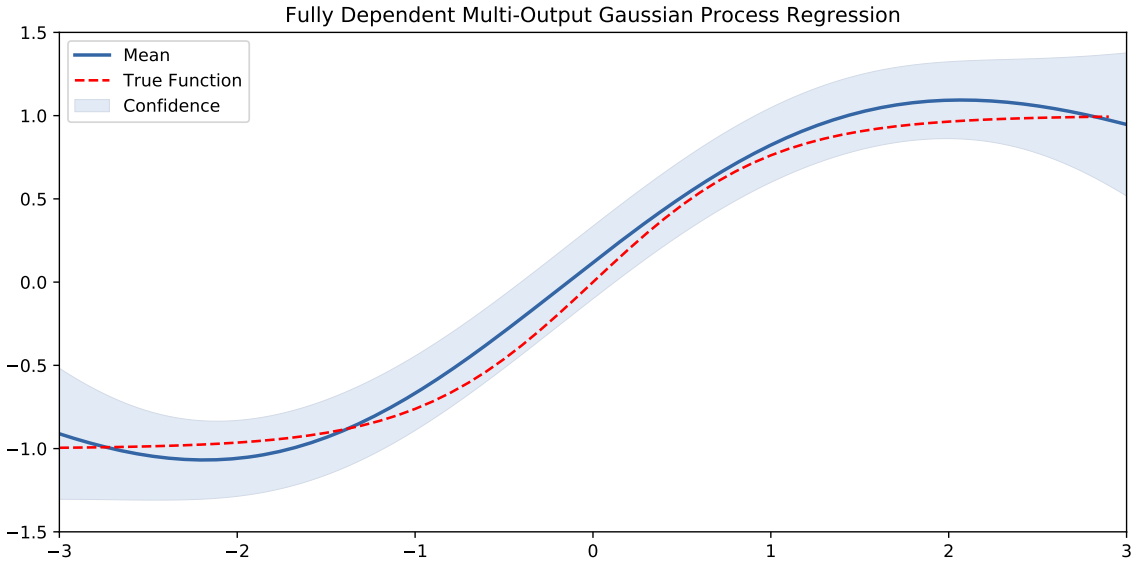

Figure 2.4: A plot of $p(f_1(x_*)|x_*, \boldsymbol{X}, \boldsymbol{Y})$ against test inputs x_* in the case of a GP with a full multi-output covariance: $k(\boldsymbol{x}, \boldsymbol{x}')_{d,d'} = \exp\left(-\frac{\|(\boldsymbol{x},d)-(\boldsymbol{x}',d')\|^2}{2l^2}\right)$. It is not that different to the separable multi-output covariance.

2.2 State-Space Models

A state-space model is a model of the form:

$$\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1) \tag{2.64}$$
$$\boldsymbol{x}_t|\boldsymbol{x}_{t-1} \sim p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) \tag{2.65}$$
$$\boldsymbol{y}_t|\boldsymbol{x}_t \sim p(\boldsymbol{y}_t|\boldsymbol{x}_t) \tag{2.66}$$

where the $\boldsymbol{x}_t \in \mathbb{R}^{n_x}$ for $t = 1, ..., T$ are latent (unobserved) variables and the $\boldsymbol{y}_t \in \mathbb{R}^{n_y}$ for $t = 1, ..., T$ are observed variables. We will refer to (2.64) as the initial distribution, (2.65) as the transition distribution and (2.66) as the observation distribution. A graphical model is provided in figure 2.5.

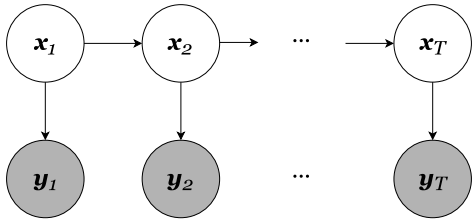

Figure 2.5: A state-space model

Two important special cases of the general state-space model are:

1. The Gaussian linear state-space model (also known as the Gaussian linear dynamical system):

$$\boldsymbol{x}_t = \boldsymbol{F}\boldsymbol{x}_{t-1} + \boldsymbol{\epsilon}_t \tag{2.67a}$$
$$\boldsymbol{y}_t = \boldsymbol{G}\boldsymbol{x}_t + \boldsymbol{w}_t \tag{2.67b}$$
$$\boldsymbol{x}_1 \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{P}_1) \tag{2.67c}$$
$$\boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_t) \tag{2.67d}$$
$$\boldsymbol{w}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t) \tag{2.67e}$$

   where $\boldsymbol{F}$, $\boldsymbol{G}$ are matrices of suitable size and $\boldsymbol{P}_1, \boldsymbol{Q}_t, \boldsymbol{R}_t$ are covariance matrices of suitable size. This model has a number of desirable properties such as a closed form recursive formula for filtering and smoothing called the Kalman filter and the Kalman smoother respectively.

2. The additive state-space model:

$$\boldsymbol{x}_t = f(\boldsymbol{x}_{t-1}) + \boldsymbol{\epsilon}_t \tag{2.68a}$$
$$\boldsymbol{y}_t = g(\boldsymbol{x}_t) + \boldsymbol{w}_t \tag{2.68b}$$
$$\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1) \tag{2.68c}$$
$$\boldsymbol{\epsilon}_t \sim p(\boldsymbol{\epsilon}_t) \tag{2.68d}$$
$$\boldsymbol{w}_t \sim p(\boldsymbol{w}_t) \tag{2.68e}$$

   where f and g are functions. As we shall see in section 2.3, a Gaussian process state-space model is an additive state-space model with Gaussian process priors on f and g.

Given an arbitrary state-space model and observations $\boldsymbol{y}_1, ..., \boldsymbol{y}_T$, important problems include:

1. Finding the filtering distributions: $p(\boldsymbol{x}_t|\boldsymbol{y}_1, ..., \boldsymbol{y}_t)$ for $t = 1, ..., T$.
2. Finding the smoothing distributions: $p(\boldsymbol{x}_t|\boldsymbol{y}_1, ..., \boldsymbol{y}_T)$ for $t = 1, ..., T$.
3. Prediction of future $\boldsymbol{y}$: $p(\boldsymbol{y}_{T+s}|\boldsymbol{y}_1, ..., \boldsymbol{y}_T)$ for some $s \in \mathbb{Z}_{\geq 1}$.
4. Finding the unknown parts of the initial, transition and observation distributions.

The general name for the first three problems is inference while the general name for the fourth problem is learning. We will discuss learning briefly at the end of this section and cover it in more depth for the specific case of Gaussian process state-space models in section 2.3. For now, we will discuss inference.

2.2.1 Inference: The General Framework

Bayesian filtering and smoothing is a framework which encompasses all the inference (filtering, smoothing and prediction) that we will look at in this thesis. We will now state the key results in this general case and throughout this section suppress the dependence on possible parameters for clarity.

Given observations $\boldsymbol{y}_{1:t}$ (where as usual $\boldsymbol{y}_{1:t}$ means $\boldsymbol{y}_1, ..., \boldsymbol{y}_t$), the joint smoothing distribution is [Doucet et al., 2001]:

$$p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t}) = \frac{p(\boldsymbol{y}_{1:t}|\boldsymbol{x}_{1:t})p(\boldsymbol{x}_{1:t})}{\int p(\boldsymbol{y}_{1:t}|\boldsymbol{x}_{1:t})p(\boldsymbol{x}_{1:t})d\boldsymbol{x}_{1:t}} \tag{2.69}$$

this can be turned into a recursion version [Doucet and Johansen, 2012]:

$$p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t}) = p(\boldsymbol{x}_{1:t-1}|\boldsymbol{y}_{1:t-1})\frac{p(\boldsymbol{y}_t|\boldsymbol{x}_t)p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})}{p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1})} \tag{2.70}$$

where $p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1})$ can be found using the integral [Doucet and Johansen, 2012]:

$$p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1}) = \int p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1})p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})p(\boldsymbol{y}_t|\boldsymbol{x}_t)d\boldsymbol{x}_{t-1:t} \tag{2.71}$$

The filtering distribution $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t})$ can be found by a two step procedure [Doucet et al., 2001; Doucet and Johansen, 2012]:

$$p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t-1}) = \int p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1})d\boldsymbol{x}_{t-1} \tag{2.72}$$

$$p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t}) = \frac{p(\boldsymbol{y}_t|\boldsymbol{x}_t)p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t-1})}{p(\boldsymbol{y}_t|\boldsymbol{y}_{1:t-1})} \tag{2.73}$$

where (2.72) is called prediction and (2.73) is called updating. Finally given observations $\boldsymbol{y}_{1:T}$, the smoothing distribution $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:T})$ can be found via the Forward-Backward Recursions [Doucet and Johansen, 2012]:

$$p(\boldsymbol{x}_t|\boldsymbol{y}_{1:T}) = p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t}) \int \frac{p(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t)}{p(\boldsymbol{x}_{t+1}|\boldsymbol{y}_{1:t})} p(\boldsymbol{x}_{t+1}|\boldsymbol{y}_{1:T})d\boldsymbol{x}_{t+1} \tag{2.74}$$

where the $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t})$ for $t = 1, ..., T$ were found during filtering (forward pass) and after we have collected all the data we can perform the smoothing (backward pass) recursively via (2.74). An alternative to this is the Generalised Two-Filter Formula which is discussed in Doucet and Johansen [2012]. As noted in Doucet et al. [2001] these equations are *'deceptively simple'* since they often require the solution of intractable integrals. To deal with this, some approximations are needed and we will now look at two different types of possible approximations: deterministic and stochastic.

2.2.2 Inference: Deterministic Methods

The two main deterministic inference frameworks are Gaussian filtering and variational inference. Gaussian filtering is in fact a form of the more general variational inference but it is very important because many methods traditionally used for inference in state-space models such as the Kalman filter, extended Kalman filter (EKF), unscented Kalman filter (UKF) and cubature Kalman filter (CKF) are all special cases of Gaussian filtering [Deisenroth, 2010].

Before discussing Gaussian filtering we will introduce the shorthand notation defined in Deisenroth [2010]:

$$\boldsymbol{\mu}^x_{t_1|t_2} = \mathbb{E}[\boldsymbol{x}_{t_1}|\boldsymbol{y}_{1:t_2}] \tag{2.75}$$
$$\boldsymbol{\Sigma}^x_{t_1|t_2} = \mathrm{cov}[\boldsymbol{x}_{t_1}|\boldsymbol{y}_{1:t_2}] \tag{2.76}$$
$$\boldsymbol{\mu}^y_{t_1|t_2} = \mathbb{E}[\boldsymbol{y}_{t_1}|\boldsymbol{y}_{1:t_2}] \tag{2.77}$$
$$\boldsymbol{\Sigma}^y_{t_1|t_2} = \mathrm{cov}[\boldsymbol{y}_{t_1}|\boldsymbol{y}_{1:t_2}] \tag{2.78}$$
$$\boldsymbol{\Sigma}^{xy}_{t_1|t_2} = \mathrm{cov}[\boldsymbol{x}_{t_1}, \boldsymbol{y}_{t_1}|\boldsymbol{y}_{1:t_2}] \tag{2.79}$$
$$\boldsymbol{\Sigma}^{yx}_{t_1|t_2} = \mathrm{cov}[\boldsymbol{y}_{t_1}, \boldsymbol{x}_{t_1}|\boldsymbol{y}_{1:t_2}] \tag{2.80}$$

The idea behind Gaussian filtering is to approximate most of the distributions which arise in the Bayesian filtering computations with Gaussian distributions by moment-matching; for example, we approximate $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t})$ with a $\mathcal{N}(\boldsymbol{\mu}^x_{t|t}, \boldsymbol{\Sigma}^x_{t|t})$ distribution. Notice that in the case where $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t})$ is already Gaussian then the above approximation will leave the distribution unchanged. The only distributions that are not (usually) approximated are the initial (2.64), transition (2.65) and observation (2.65) distributions because as long as we can find their first two moments we do not need to approximate them.

Assuming we know $\boldsymbol{\mu}^x_{t-1|t-1}$ and $\boldsymbol{\Sigma}^x_{t-1|t-1}$ and therefore have a Gaussian approximation of $p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1})$ as $\mathcal{N}(\boldsymbol{x}_{t-1}|\boldsymbol{\mu}^x_{t-1|t-1}, \boldsymbol{\Sigma}^x_{t-1|t-1})$, it is possible to derive $\boldsymbol{\mu}^x_{t|t}$ and $\boldsymbol{\Sigma}^x_{t|t}$ and therefore a Gaussian approximation of $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t})$ [Deisenroth, 2010]. Following this recursively will yield a sequence of Gaussian approximate filtering distributions. To derive the updates, we note that Gaussian filtering allows us to avoid direct computation of the prediction update integral in (2.72) because all we need are the first two moments (since we are going to approximate it with a Gaussian distribution)

and hence following Deisenroth [2010] we get:

$$p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t-1}) = \mathcal{N}(\boldsymbol{\mu}^x_{t|t-1}, \boldsymbol{\Sigma}^x_{t|t-1}) \tag{2.81}$$

where

$$\begin{aligned}
\boldsymbol{\mu}^x_{t|t-1} &= \mathbb{E}[p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t-1})] && (2.82)\\
&= \int \boldsymbol{x}_t p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t-1}) d\boldsymbol{x}_t && (2.83)\\
&= \iint \boldsymbol{x}_t p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1}) d\boldsymbol{x}_{t-1} d\boldsymbol{x}_t && (2.84)\\
&= \iint \boldsymbol{x}_t p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) d\boldsymbol{x}_t p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1}) d\boldsymbol{x}_{t-1} && (2.85)\\
&= \int \mathbb{E}[p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})] p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1}) d\boldsymbol{x}_{t-1} && (2.86)\\
&= \int \mathbb{E}[p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})] \mathcal{N}(\boldsymbol{x}_{t-1}|\boldsymbol{\mu}^x_{t-1|t-1}, \boldsymbol{\Sigma}^x_{t-1|t-1}) d\boldsymbol{x}_{t-1} && (2.87)
\end{aligned}$$

where we note that if the expectation exists (i.e. is finite) swapping the integrals is allowed by Fubini's Theorem. Via a similar method,

$$\begin{aligned}
\boldsymbol{\Sigma}^x_{t|t-1} = \int \mathbb{E}[p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})^T] \mathcal{N}(\boldsymbol{x}_{t-1}|\boldsymbol{\mu}^x_{t-1|t-1}, \boldsymbol{\Sigma}^x_{t-1|t-1}) d\boldsymbol{x}_{t-1} \\
- \boldsymbol{\mu}^x_{t|t-1} (\boldsymbol{\mu}^x_{t|t-1})^T
\end{aligned} \tag{2.88}$$

Using the same approach it is possible to derive $\boldsymbol{\Sigma}^{xy}_{t|t-1}$, $\boldsymbol{\Sigma}^{y}_{t|t-1}$ and $\boldsymbol{\Sigma}^{yx}_{t|t-1}$ [Deisenroth, 2010]. The filter updates are derived in Deisenroth [2010] and are reproduced here:

$$\boldsymbol{\mu}^x_{t|t} = \boldsymbol{\mu}^x_{t|t-1} + \boldsymbol{\Sigma}^{xy}_{t|t-1} (\boldsymbol{\Sigma}^y_{t|t-1})^{-1} (\boldsymbol{y}_t - \boldsymbol{\mu}^y_{t|t-1}) \tag{2.89}$$

$$\boldsymbol{\Sigma}^x_{t|t} = \boldsymbol{\Sigma}^x_{t|t-1} + \boldsymbol{\Sigma}^{xy}_{t|t-1} (\boldsymbol{\Sigma}^y_{t|t-1})^{-1} \boldsymbol{\Sigma}^{yx}_{t|t-1} \tag{2.90}$$

Although Gaussian filtering avoids computation of certain integrals such as (2.72), there is no guarantee that these new integrals (2.87), *etc* are tractable. As a result, the different Gaussian filtering methods make different additional approximations; for example, the extended Kalman filter performs filtering on the additive state-space model by linearisation of the non-linear functions (approximating $f(x_{t-1})$ with $F_t x_{t-1}$). This allows for the simple computation of the required integrals: see (2.91)–(2.96). Other Gaussian filtering methods; for example, the unscented Kalman filter and the cubature Kalman filter attempt to approximate the new integrals (2.87), *etc* directly. For more details see Deisenroth [2010]. In the case of the linear Gaussian dynamical system, there are no approximations made by Gaussian filtering: all the required equations can be computed exactly and the resulting update equations are the Kalman filter update equations; for example, (2.87) becomes:

$$\int \mathbb{E}[f(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})] \mathcal{N}(\boldsymbol{x}_{t-1}|\boldsymbol{\mu}^x_{t-1|t-1}, \boldsymbol{\Sigma}^x_{t-1|t-1}) d\boldsymbol{x}_{t-1} \tag{2.91}$$

$$= \int \boldsymbol{F}\boldsymbol{x}_{t-1}\mathcal{N}(\boldsymbol{x}_{t-1}|\boldsymbol{\mu}^x_{t-1|t-1}, \boldsymbol{\Sigma}^x_{t-1|t-1})d\boldsymbol{x}_{t-1} \tag{2.92}$$

$$= \boldsymbol{F}\boldsymbol{\mu}^x_{t-1|t-1} \tag{2.93}$$

and the updates become (Deisenroth [2010]):

$$\boldsymbol{\mu}^x_{t|t} = \boldsymbol{F}\boldsymbol{\mu}^x_{t-1|t-1} + \boldsymbol{\Sigma}^x_{t|t-1}\boldsymbol{G}^T(\boldsymbol{G}\boldsymbol{\Sigma}^x_{t|t-1}\boldsymbol{G}^T + \boldsymbol{R}_t)^{-1}(\boldsymbol{y}_t - \boldsymbol{G}\boldsymbol{\mu}^x_{t-1|t-1}) \tag{2.94}$$

$$\boldsymbol{\Sigma}^x_{t|t} = \boldsymbol{\Sigma}^x_{t|t-1} + \boldsymbol{\Sigma}^x_{t|t-1}\boldsymbol{G}^T(\boldsymbol{G}\boldsymbol{\Sigma}^x_{t|t-1}\boldsymbol{G}^T + \boldsymbol{R}_t)^{-1}\boldsymbol{G}\boldsymbol{\Sigma}^x_{t|t-1} \tag{2.95}$$

where

$$\boldsymbol{\Sigma}^x_{t|t-1} = \boldsymbol{F}\boldsymbol{\Sigma}^x_{t-1|t-1}\boldsymbol{F}^T + \boldsymbol{Q}_t \tag{2.96}$$

Using the same ideas as Gaussian filtering, it is possible to derive a set a recursive equations to generate the Gaussian approximated smoothing distributions, these approximate distributions become the exact smoothing distributions in the linear Gaussian dynamical case and are known as the Kalman smoother: see section 4.2.2 in Deisenroth [2010].

A more general Bayesian filtering/smoothing method is variational inference which tries to approximate the distributions inside the Bayesian integrals (2.71)–(2.72) with distributions that make the computation of the integrals tractable.

Definition 2.6. [Bishop, 2009] The Kullback-Leibler divergence is defined as:

$$\mathcal{KL}(q||p) = -\int q(x)\log\left(\frac{p(x)}{q(x)}\right)dx \tag{2.97}$$

where p and q are functions. Its properties include:

1. $\mathcal{KL}(q||p) \geq 0$.
2. $\mathcal{KL}(q||p) = 0 \iff q = p$.

Consider (2.72) which is reproduced here for convenience:

$$p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t-1}) = \int p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1})d\boldsymbol{x}_{t-1} \tag{2.98}$$

A variational method would try to make this integral tractable by approximating $p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1})$ or $p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$. One way to do this would be to consider a distribution $q(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1})$ which approximates $p(\boldsymbol{x}_{t-1}|\boldsymbol{y}_{1:t-1})$. To select the appropriate q one usually tries to find the q that minimises the Kullback-Leibler divergence $\mathcal{KL}(q||p)$. Now of course if no constraints are placed on q then the optimal q is p which is not very helpful. The key to variational inference is to come up with constraints on q that are not too tight and allow the required integrals to become tractable. These methods tend to be very problem specific and so we will not discuss them here: they will be introduced as needed.

2.2.3 Inference: Stochastic Methods

The main stochastic methods for inference in state-space models are sequential Monte Carlo (SMC) methods also known as particle filters and particle smoothers. In this section, we will look at a basic particle filter known as the bootstrap filter and then discuss the issues which can arise in SMC methods; finally, we will briefly look at smoothing and an important state-of-the-art particle smoother called Particle Gibbs with Ancestor Sampling (PGAS) [Lindsten et al., 2014].

In particle filters, the aim is (often) to devise a method to sample from the distributions $p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$ for $t = 1, ..., T$ and if we can sample from such distributions, it is trivial to sample from the filter distributions by just taking the last element of the sampled vectors. The assumption [Doucet et al., 2001] is that sampling from $p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$ directly is hard (or sampling takes a long time) and for state-space models this is almost always the case. As a result, sequential Monte Carlo (SMC) methods are based on importance sampling, which, instead of sampling directly from $p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$, samples from a *importance* or *proposal* distribution $q(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$ that is similar to $p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$ but easier to sample from. These samples are then weighted; with samples most like samples from $p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$ given more weight. Formally, an importance distribution $q(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$ can be any distribution that such that $p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t}) > 0 \implies q(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t}) > 0$ [Doucet et al., 2001]. In the case of importance sampling in state-space models we have [Doucet and Johansen, 2012]:

$$p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t}) = \frac{w_t(\boldsymbol{x}_{1:t})q(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})}{\boldsymbol{Z}_t} \tag{2.99}$$

$$\boldsymbol{Z}_t = \int w_t(\boldsymbol{x}_{1:t})q(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})d\boldsymbol{x}_{1:t} \tag{2.100}$$

$$w_t(\boldsymbol{x}_{1:t}) = \frac{p(\boldsymbol{x}_{1:t}, \boldsymbol{y}_{1:t})}{q(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})} \tag{2.101}$$

where $q(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$ is the importance distribution and the $w_t(\boldsymbol{x}_{1:t})$ are known as the unnormalised weights. This formulation relies on knowing $p(\boldsymbol{x}_{1:t}, \boldsymbol{y}_{1:t})$, which, unlike $p(\boldsymbol{x}_{1:t}|\boldsymbol{y}_{1:t})$, is usually easy to calculate for state-space models. To calculate some quantity of interest say $\mathbb{E}_p[h(\boldsymbol{x}_{1:t})]$ we can get independent samples $\boldsymbol{x}^i_{1:t}$ from q and approximate it as [Doucet and Johansen, 2012]:

$$\mathbb{E}_p[h(\boldsymbol{x}_{1:t})] \approx \sum_i W^i_t h(\boldsymbol{x}^i_{1:t}) \tag{2.102}$$

where

$$W^i_t = \frac{w_t(\boldsymbol{x}^i_{1:t})}{\sum_i w_t(\boldsymbol{x}^i_{1:t})} \tag{2.103}$$

Now the main issue here is that it is very inefficient since as noted in Doucet et al. [2001] we are not really using the sequential nature of the problem and forgetting everything we learnt at time $t-1$ for time t. In Doucet and Johansen [2012] it is shown that one can rewrite the weights allowing for sequential updates:

$$w_t(\boldsymbol{x}_{1:t}) = w_{t-1}(\boldsymbol{x}_{1:t-1})\alpha_t(\boldsymbol{x}_{1:t}) \tag{2.104}$$

$$\alpha_t(\boldsymbol{x}_{1:t}) = \frac{p(\boldsymbol{x}_{1:t}, \boldsymbol{y}_{1:t})}{p(\boldsymbol{x}_{1:t-1}, \boldsymbol{y}_{1:t-1})q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}, \boldsymbol{y}_{1:t})} \tag{2.105}$$

A *particle* refers to a vector $\boldsymbol{x}_{1:t}$ and so having N particles means we have N vectors $\boldsymbol{x}^i_{1:t}$ for $i = 1, ..., N$. Associated to each particle is a weight $w_t(\boldsymbol{x}^i_{1:t})$ which determines its importance in describing the true distribution. Before stating the bootstrap filter algorithm, there two more things we need to deal with: resampling and choosing a suitable importance/proposal distribution. One of the problems that can arise in SMC is *weight degeneracy* (also known as *path degeneracy*) [Doucet and Johansen, 2012]. This occurs when we have a continually increasing particle weight variance and eventually this leads to one particle having all the weight which is problematic since all the estimates will end up being based on only one particle. This *weight degeneracy* is measured by the effective sample size [Doucet and Johansen, 2012]:

$$ESS = \frac{1}{\sum_{i=1}^{N}(W^i_t)^2} \tag{2.106}$$

with smaller ESS meaning more weight degeneracy. This degeneracy can be reduced using resampling and Doucet and Johansen [2012, p. 13] gives a detailed overview of all the most common resampling techniques. Unfortunately, resampling causes another type of path degeneracy known as *sample degeneracy* or *sample impoverishment* whereby all particles eventually become the same [Doucet and Johansen, 2012]. One method to deal with this is to only resample when necessary (i.e. only resample if ESS is smaller than some threshold say $\frac{N}{2}$). This is known as adaptive resampling. The last thing left to do is to find a suitable importance/proposal distribution, for SMC in state-space models all we need to find is a proposal distribution of the form [Doucet and Johansen, 2012]:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}, \boldsymbol{y}_{1:t}) \tag{2.107}$$

It turns out [Doucet and Johansen, 2012, p. 20] that the optimal (in the sense of minimising weight variance which is important due to path degeneracy) proposal is:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}, \boldsymbol{y}_{1:t}) = \frac{p(\boldsymbol{y}_t|\boldsymbol{x}_t)p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})}{p(\boldsymbol{y}_t|\boldsymbol{x}_{t-1})} \tag{2.108}$$

Although this is the optimal proposal distribution, choosing a different proposal distribution is often done and the bootstrap filter uses the bootstrap proposal distribution: $q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}, \boldsymbol{y}_{1:t}) = p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$. This is non-optimal but it is still good because the optimal proposal requires computation of $p(\boldsymbol{y}_t|\boldsymbol{x}_{t-1})$ and this might not be tractable. It is possible to use MCMC methods to approximate the integral in $p(\boldsymbol{y}_t|\boldsymbol{x}_{t-1})$ but this will tend to slow everything down. Overall, the bootstrap proposal is a good and easy choice to make; however, in certain cases we will do better with different proposal distribution.

Algorithm 1: The Bootstrap Filter with Adaptive Resampling [Doucet et al., 2001; Doucet and Johansen, 2012]

Input : Number of Particles N, Observations $\boldsymbol{y}_{1:T}$, Function $M(\boldsymbol{x}_{1:t})$
Output: N samples from $p(\boldsymbol{x}_{1:T}|\boldsymbol{y}_{1:T})$, Weights W_T^i for $i = 1, ..., N$ and $\mathbb{E}[M(\boldsymbol{x}_t)]$ for $t = 1, ..., T$

set weights $w_1^i = 1/N$ for $i = 1, ..., N$;
for *t = 1, ..., T* **do**
  if $t = 1$ **then**
    sample $\boldsymbol{x}_1^i \sim p(\boldsymbol{x}_1)$ for $i = 1, ..., N$;
  else
    sample $\boldsymbol{x}_t^i \sim p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}^i)$ for $i = 1, ..., N$;
    set $\boldsymbol{x}_{1:t}^i = (\boldsymbol{x}_{1:t-1}^i, \boldsymbol{x}_t^i)$ for $i = 1, ..., N$;
  end
  compute importance weights: $w_t^i = w_{t-1}^i p(\boldsymbol{y}_t|\boldsymbol{x}_t^i)$;
  normalise importance weights: $W_t^i = \frac{w_t^i}{\sum_{i=1}^N w_t^i}$;
  compute $\mathbb{E}[M(\boldsymbol{x}_{1:t})] = \sum_{i=1}^N W_t^i M(\boldsymbol{x}_{1:t}^i)$;
  compute effective sample size: $ESS = \frac{1}{\sum_{i=1}^N (W_t^i)^2}$;
  if $ESS < \frac{N}{2}$ **then**
    resample particles and set $w_t^i = 1/N$ using techniques in Doucet and Johansen [2012, p. 13];
  end
end
return $\boldsymbol{x}_{1:T}^i$, W_T^i for $i = 1, ..., N$ and $\mathbb{E}[M(\boldsymbol{x}_{1:t})]$ for $t = 1, ..., T$;

Due to sample impoverishment, most likely the N samples of $p(\boldsymbol{x}_{1:T}|\boldsymbol{y}_{1:T})$ from the bootstrap filter will all be exactly the same (if T is large) but by setting $M(x) = x$, we can get a good value for the mean of the sequence of filtering distributions over time and something similar can be done the other moments. See section 2.2.5 for some examples.

SMC methods for smoothing follow the same ideas as filtering; however, they often have a higher time complexity. Although, in theory, we could get samples from the smoothing distribution $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:T})$ by just looking at the tth component of the vector $p(\boldsymbol{x}_{1:T}|\boldsymbol{y}_{1:T})$ for each of the N samples; in practice, due to sample impoverishment this will not give a good result [Doucet and Johansen, 2012]. Methods for smoothing include the *Fixed-Lag Approximation* and *Forward Filtering-Backward Smoothing* both of which are discussed in Doucet and Johansen [2012, pp. 34-36]. One of the most recent particle smoothing methods is Particle Gibbs with Ancestor Sampling (PGAS) [Lindsten et al., 2014], this requires an additional input compared with the bootstrap filter: a reference trajectory which the algorithm uses to guide itself to the invariant distribution of the latent states. This reference trajectory could be outputted from another method; for example, the mean trajectory in the bootstrap filter (i.e. the sequence $M(x_{1:t})$ when $M(x_{1:t}) = x_t$) or it could be the output trajectory from the previous run of the PGAS algorithm. Unlike the bootstrap filter,

PGAS only produces one sample (trajectory) per run and although degeneracy still occurs in each run, only producing one sample per run mitigates this degeneracy and also means we do not need to use as many particles as in the bootstrap filter [Lindsten et al., 2014]. The PGAS algorithm is provided in Algorithm 2.

Although we have looked at SMC from the point of view of state-space models, it can be used in a wide variety of sequential and non-sequential problems [Doucet and Johansen, 2012, p. 35]; for example, sampling via Metropolis-Hastings can use SMC to improve mixing and this leads to particle Metropolis-Hastings and other particle MCMC methods [Dahlin and Schön, 2016]. We will compare a few SMC methods with some Gaussian filtering methods in section 2.2.5.

Algorithm 2: Particle Gibbs with Ancestor Sampling [Lindsten et al., 2014; Svensson et al., 2016]

Input : Number of Particles N, Observations $\boldsymbol{y}_{1:T}$, Reference Trajectory $(\hat{\boldsymbol{x}}_{1:T})$
Output: One sample from $p(\boldsymbol{x}_{1:T}|\boldsymbol{y}_{1:T})$

sample $\boldsymbol{x}_1^i$ from the initial distribution $p(\boldsymbol{x}_1)$ for $i = 1, ..., N-1$;
set $\boldsymbol{x}_1^N = \hat{\boldsymbol{x}}_1$;
set weights $w^i = p(\boldsymbol{y}_1|\boldsymbol{x}_1^i)$ for $i = 1, ..., N$;
for *t = 2, ..., T* **do**
    `/* resampling and ancestor sampling */`
    sample with replacement $N-1$ particles $a_1, ..., a_{N-1}$ from the N particles $\boldsymbol{x}_{1:t-1}^i$ with the probability of selecting particle $\boldsymbol{x}_{1:t-1}^i$ proportional to its weight;
    compute $p^i = w^i p(\hat{\boldsymbol{x}}_t|\boldsymbol{x}_{t-1}^i)$ for $i = 1, ..., N$ where $p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is the transition distribution;
    normalise: $\hat{p}^i = \frac{p^i}{\sum_i p^i}$ for $i = 1, ..., N$;
    sample a single particle a_N from the N particles $\boldsymbol{x}_{1:t-1}^i$ with the probability of selecting particle $\boldsymbol{x}_{1:t-1}^i$ equal to $\hat{p}^i$;
    `/* particle propagation */`
    sample $\boldsymbol{x}_t^i$ from the transition distribution $p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1} = a_i)$ for $i = 1, ..., N-1$;
    set $\boldsymbol{x}_t^N = \hat{\boldsymbol{x}}_t$;
    set $\boldsymbol{x}_{1:t-1}^i = a_i$ for $i = 1, ..., N$;
    `/* weighting */`
    set weights $w^i = p(\boldsymbol{y}_t|\boldsymbol{x}_t^i)$ for $i = 1, ..., N$;
end
return one sample $\boldsymbol{x}_{1:T}$ by sampling from the particles with the probability of selecting particle $\boldsymbol{x}_{1:T}^i$ proportional to its weight w^i;

2.2.4 Learning

There are two possible kinds of learning that can be done in state-space models. Firstly, there is parametric learning, this is where we assume that the functional forms of the initial, transition and observation distributions in (2.64)–(2.66) are known but there are some unknown parameters. In this case, the task in *learning* is to find these unknown parameters; for example, in the linear state-space model one way to find the optimal parameters is via a combination of Kalman filtering/smoothing and expectation-maximisation (see Hamilton [1994] for more details). Furthermore, parametric learning in non-linear non-Gaussian models is possible via PGAS. We can construct a Gibbs sampler and alternate between sampling from the distributions $p(\boldsymbol{x}_{1:T}|\boldsymbol{y}_{1:T}, \boldsymbol{\theta})$ and $p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$ where $\boldsymbol{\theta}$ represents the unknown parameters. After an initial burn-in period this will generate (approximate) samples from the posterior distribution of the parameters: see Lindsten et al. [2014, p. 2161] for more details. Although these methods have good theoretical properties, there are still issues and one of the key problems is identifiability.

Definition 2.7. [Lehmann and Casella, 1998] Suppose the random variable X has a distribution $p_{\boldsymbol{\theta}}$ where $\boldsymbol{\theta}$ is a vector of parameters, then we say $\boldsymbol{\theta}$ is identifiable if $p_{\boldsymbol{\theta}_1} = p_{\boldsymbol{\theta}_2} \implies \boldsymbol{\theta}_1 = \boldsymbol{\theta}_2$.

When optimising the parameters, having a lack of identifiability results in the model having many equivalent optima. This might not be a problem by itself, since by the very nature of identifiability they are all equally valid given the observations, and if we just want to perform forecasting this unidentifiabilty might not be a problem [Frigola, 2015, p. 19]. However, if we wish to be able to interpret the parameters, then identifiability is an issue since not all parameter values may even make sense. In this case, placing priors on the parameters or starting the optimisations in different places may help to alleviate this issue. For a general state-space model: (2.64)–(2.66) we cannot expect to have identifiability.

The second kind of learning in state-space models is non-parametric learning and this is where we do not assume any particular functional form for the initial, transition and observation distributions. Non-parametric learning is a hard problem and generally requires us to assume something about the final form; for example, an additive state-space model with a Gaussian likelihood. We will look at non-parametric learning in depth for the particular case of Gaussian process state-space models in section 2.3.

2.2.5 Examples

In this section, we will compare Gaussian filtering/smoothing with sequential Monte Carlo methods via a number of examples.

Example 1:
The first set of examples will be based on the model:

$$\boldsymbol{x}_t = 0.9\boldsymbol{x}_{t-1} + \boldsymbol{\epsilon}_t \tag{2.109}$$
$$\boldsymbol{y}_t = 3\boldsymbol{x}_t + \boldsymbol{w}_t \tag{2.110}$$
$$\boldsymbol{x}_1 \sim \mathcal{N}(0, 0.1^2) \tag{2.111}$$
$$\boldsymbol{\epsilon}_t \overset{iid}{\sim} \mathcal{N}(0, 0.3^2) \tag{2.112}$$
$$\boldsymbol{w}_t \overset{iid}{\sim} \mathcal{N}(0, 1^2) \tag{2.113}$$

The bootstrap filter will use 500 particles and PGAS will use 5 particles starting from a reference trajectory of all zeros and a burn-in of 30 samples. We generated a single set of 300 points of simulated observations/hidden states from the model above and each of the methods were given the same set of simulated observations. The aim was to find the hidden states and we scored each of the methods by calculating the MSE between the mean of filtering/smoothing distribution and the true hidden states. The MSEs for the first example are displayed in following table:

Method:	**MSE:**
Bootstrap Filter	0.066
Bootstrap Smoother	0.055
Kalman Filter	0.065
Kalman Smoother	0.053
PGAS	**0.052**

(2.114)

On this simple linear system, the above SMC methods perform similarly to their Kalman equivalents. This is as expected because the Kalman methods are meant to be the optimal (best unbiased estimator) filters/smoothers on this sort of system [Hamilton, 1994]. The results are plotted in figures 2.6–2.8.

Example 2:
The second set of examples will compare the unscented Kalman filter (UKF), bootstrap filter, unscented Kalman smoother (UKS), bootstrap smoother and PGAS. This set of examples will be based on the model:

$$\boldsymbol{x}_t = \sin(\boldsymbol{x}_{t-1}) + \boldsymbol{\epsilon}_t \tag{2.115}$$
$$\boldsymbol{y}_t = \exp(\boldsymbol{x}_t) + \boldsymbol{w}_t \tag{2.116}$$
$$\boldsymbol{x}_1 \sim \mathcal{N}(0, 0.1^2) \tag{2.117}$$
$$\boldsymbol{\epsilon}_t \sim \mathcal{N}(0, 0.3^2) \tag{2.118}$$
$$\boldsymbol{w}_t \sim \mathcal{N}(0, 1^2) \tag{2.119}$$

with everything else the same as in the first set of examples. The results are plotted in figures 2.9–2.11 and the MSEs are displayed in the following table:

Method:	**MSE:**
Bootstrap Filter	0.17
Bootstrap Smoother	**0.14**
UKF	0.21
UKS	0.20
PGAS	**0.14**

(2.120)

The key takeaway from both examples is that SMC methods are a competitive inference method and can provide excellent estimates of the required distributions compared to the traditional methods such as the UKF/UKS.

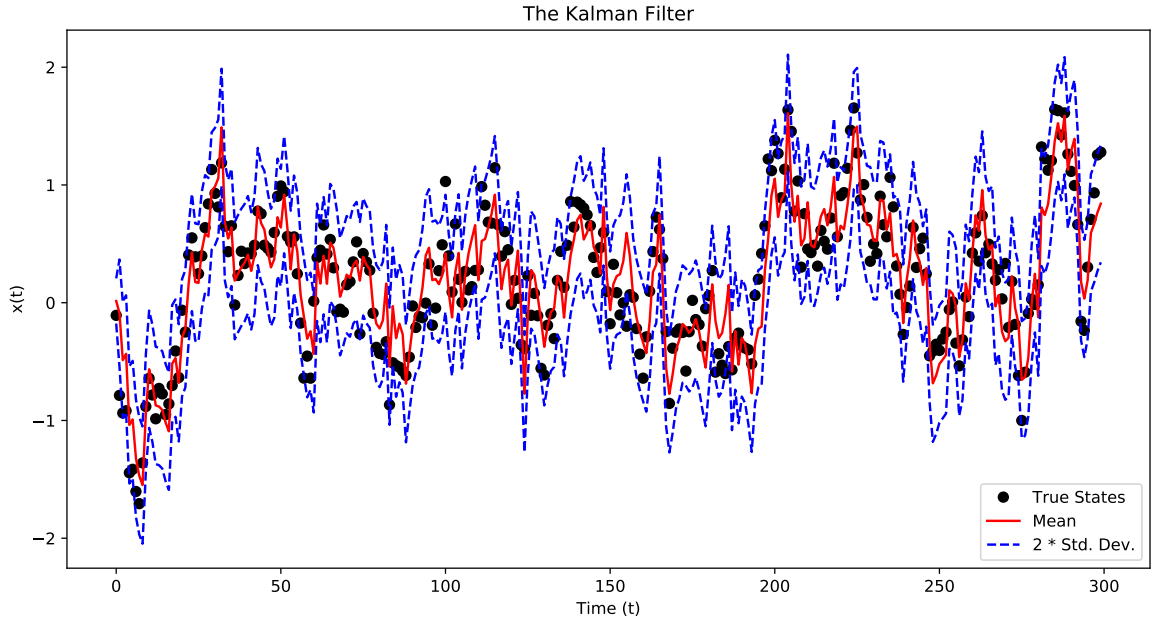

Figure 2.6: The Kalman Filter (Example 1)

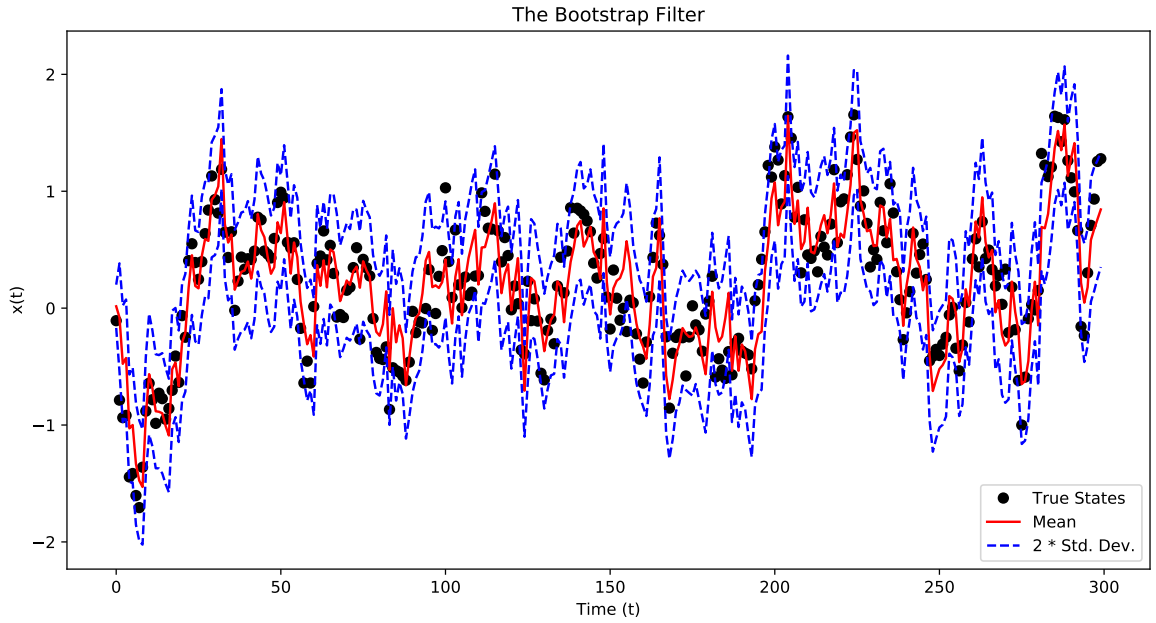

Figure 2.7: The Bootstrap Filter (Example 1)

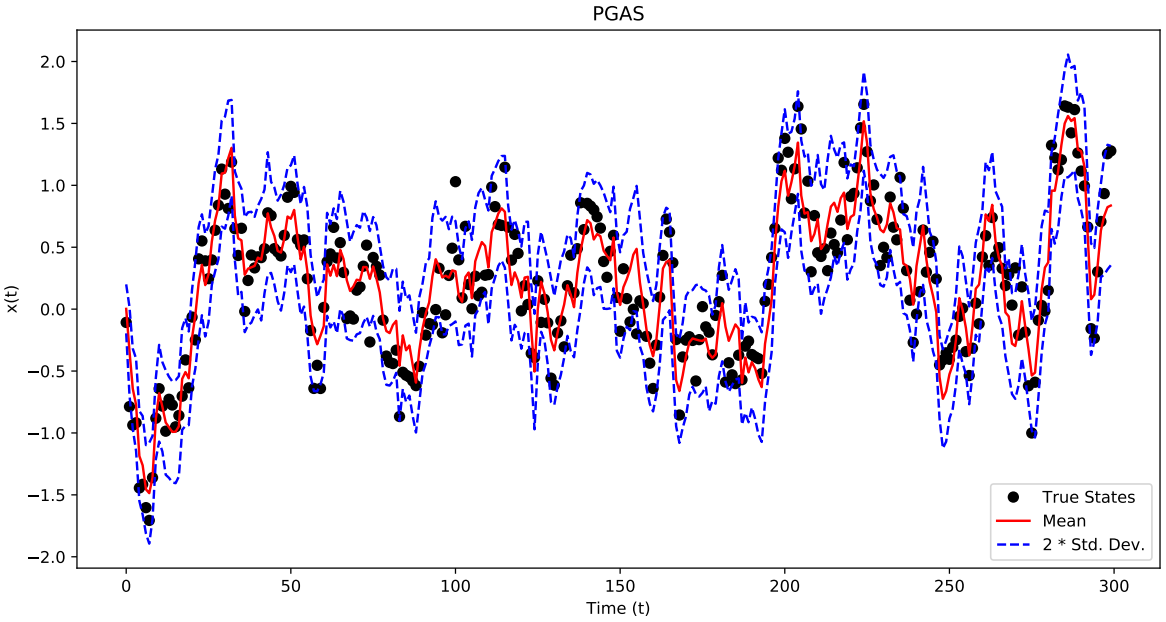

Figure 2.8: Particle Gibbs with Ancestor Sampling (PGAS) (Example 1)

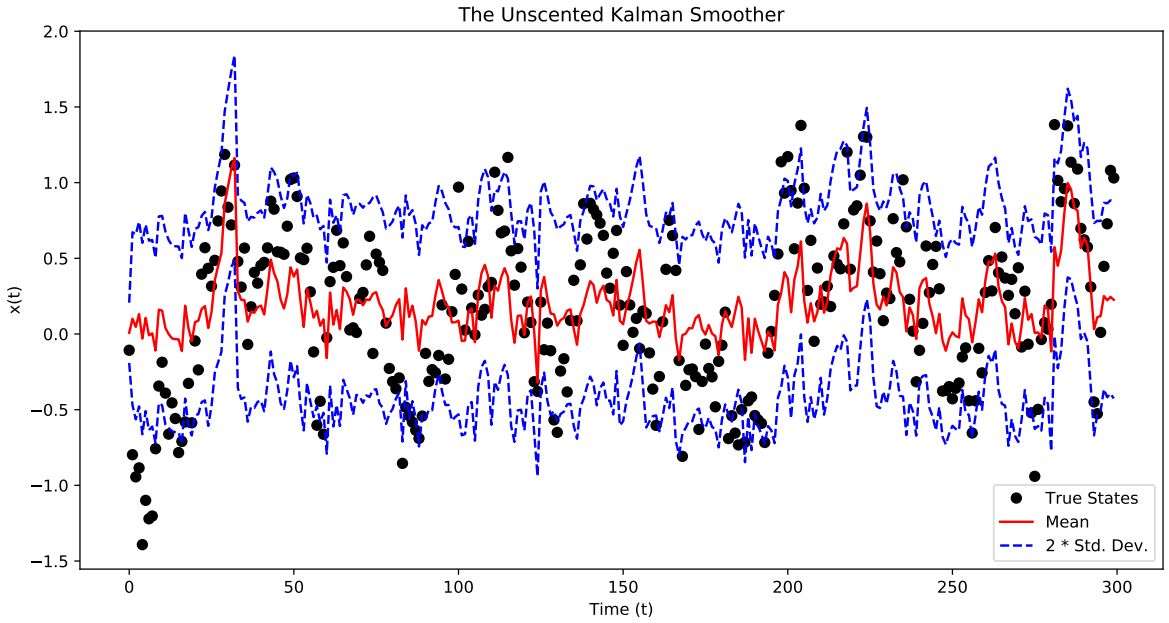

Figure 2.9: The Unscented Kalman Filter (Example 2)

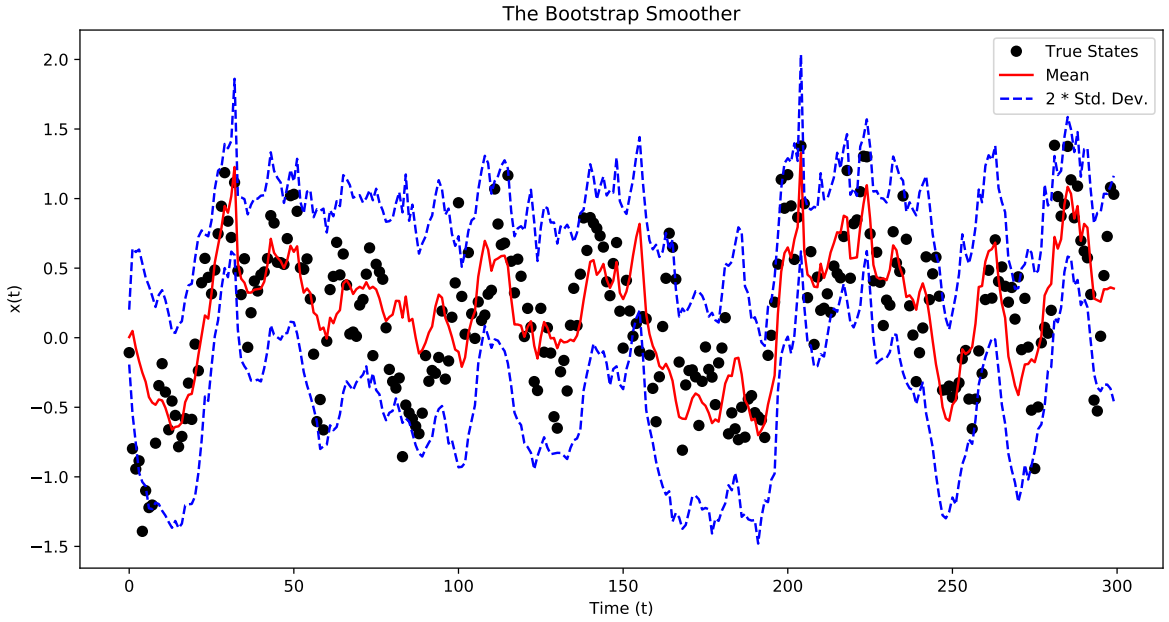

Figure 2.10: The Bootstrap Smoother (Example 2)

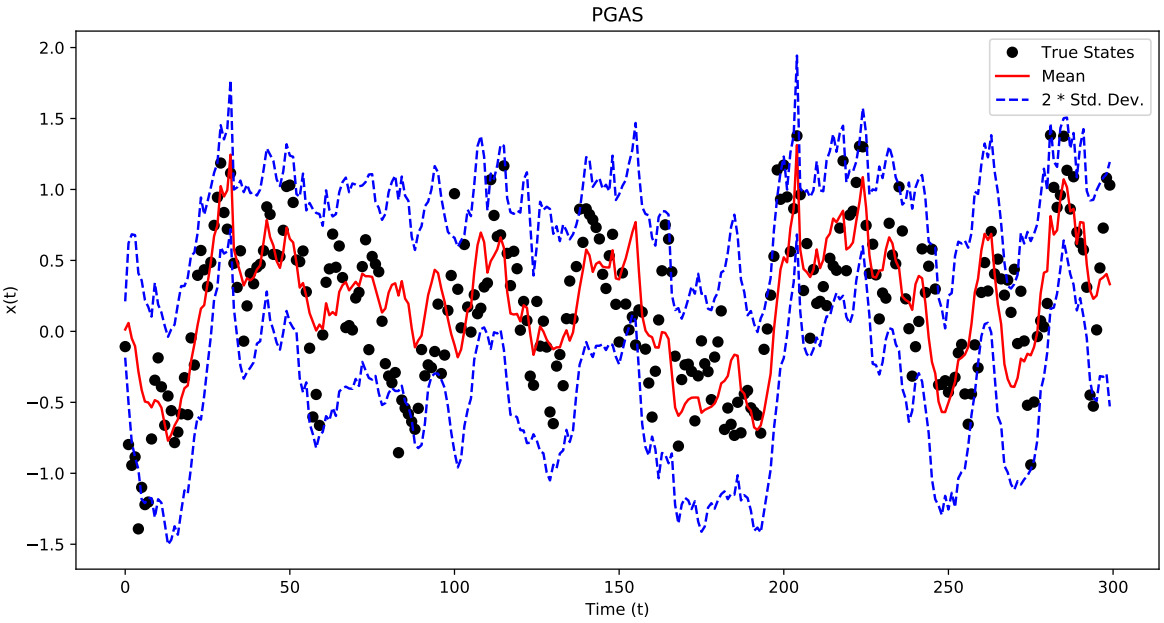

Figure 2.11: Particle Gibbs with Ancestor Sampling (PGAS) (Example 2)

2.3 Gaussian Process State-Space Models

A Gaussian process state-space model (GPSSM) is a model of the form:

$$\boldsymbol{x}_t = f(\boldsymbol{x}_{t-1}) + \boldsymbol{\epsilon}_t \tag{2.121a}$$
$$\boldsymbol{y}_t = g(\boldsymbol{x}_t) + \boldsymbol{w}_t \tag{2.121b}$$
$$\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1) \tag{2.121c}$$
$$f \sim \mathcal{GP}(m_f, k_f) \tag{2.121d}$$
$$g \sim \mathcal{GP}(m_g, k_g) \tag{2.121e}$$

where $\boldsymbol{\epsilon}_t$ and $\boldsymbol{w}_t$ are random variables, the $\boldsymbol{x}_t \in \mathbb{R}^{n_x}$ for $t = 1, ..., T$ are latent (unobserved) variables and the $\boldsymbol{y}_t \in \mathbb{R}^{n_y}$ for $t = 1, ..., T$ are observed variables. In the case where we have:

$$\boldsymbol{\epsilon}_t \overset{iid}{\sim} \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}) \tag{2.122}$$
$$\boldsymbol{w}_t \overset{iid}{\sim} \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}) \tag{2.123}$$

then this is called a GPSSM with (independent) Gaussian latent and observation noise. When we refer to a GPSSM we will be referring to a GPSSM with Gaussian latent and observation noise unless stated otherwise. Sometimes the functional form of f or g is known and so we only have one GP; in the case where the functional form of f is known and g is a GP we will call this a GPSSM with a known transition function and in the case where the functional form of g is known and f is a GP we will call this a GPSSM with a known observation function. As with all state-space

models there are two key problems we wish to solve: learning and inference. We will now look briefly at inference and then discuss the current status of research for learning GPSSMs.

2.3.1 Inference in GPSSMs

Once we have knowledge of the posterior predictive distributions for the transition function f in (2.121a) and observation function g in (2.121b), many of the inference techniques we discussed previously (unscented Kalman filter, bootstrap filter, etc) can be used directly for GPSSMs. However, this assumes that we can come up with some method for learning the covariance hyperparameters in f and g as well as finding their respective posterior predictive distributions. This is non-trivial because, in general, we will not have knowledge of the true hidden states $\boldsymbol{x}_t$ even during training. Furthermore, even if somehow we could learn the functions f and g, as they are GPs there will always be some posterior uncertainty; but the inference techniques we have seen so far do not take into account this addtional uncertainty and so will tend to give overconfident predictions [Deisenroth et al., 2009; Deisenroth, 2010].

As a result, several inference techniques have been devised which are tailored to GPSSMs. Gaussian filtering has been applied to GPSSMs [Deisenroth et al., 2009; Deisenroth, 2010] in a form known as the Gaussian process Assumed Density Filter (GP-ADF). Here, the Gaussian filtering framework is unchanged in the sense that we only need to compute to moments $\boldsymbol{\Sigma}^{xy}_{t|t-1}$, $\boldsymbol{\Sigma}^{y}_{t|t-1}$, etc. However, in the GPSSM case the required moment calculations need to take into account the uncertainty in f and g. Moreover, the GPs f, g have uncertain inputs: when trying the calculate the next hidden state, the previous hidden state has a distribution of possible values and this uncertainty increases the GP posterior uncertainty. All this additional complexity leads to the required integrals (e.g. for computing $\boldsymbol{\Sigma}^{xy}_{t|t-1}$) only being tractable for a very small set of covariance functions such as the square exponential or rational quadratic covariances [Deisenroth et al., 2009]. Furthermore, during training, knowledge of the true hidden states is required [Deisenroth et al., 2009] which may not be possible depending on the problem. That said, in cases where these integrals are tractable the GP-ADF performs much better [Deisenroth et al., 2009, sec. 6.1] than the standard unscented Kalman filter (UKF) and it also performs better than a form of the UKF which takes GP uncertainty into account (GP-UFK) [Ko and Fox, 2008].

This has been extended to smoothing in Deisenroth [2010]; Deisenroth et al. [2012] and is known as GP-RTSS. This performs better than the standard unscented Kalman smoother (UKS) [Deisenroth et al., 2012, p. 5] but for the same reason as the GP-ADF, we have a very limited choice of covariance functions. For both the GP-ADF and GP-RTSS, the results are only derived for the case of the different output dimensions using independent GPs with scalar-valued covariances rather than using a single GP with a multi-output covariance. That said, it is probably possible to derive

something similar for the multi-output case.

Other inference techniques used successfully in GPSSMs are expectation propagation [Deisenroth and Mohamed, 2016] (although it was only derived and tested using an SE kernel) and particle filters [Frigola et al., 2013]. Although, the SE kernel is expressive, for some tasks other kernels are better suited (say for forecasting a periodic function) and as a result we are really looking for methods that work well across a wide range of covariance functions.

2.3.2 Learning in GPSSMs

Suppose we have a sequence of observations $\boldsymbol{Y}_{1:T} = [\boldsymbol{y}_1, ..., \boldsymbol{y}_T]$ and we wish to learn a GPSSM; this means that we wish to find the covariance hyperparameters for the functions f and g (or in the Bayesian case a posterior distribution for these hyperparameters). Wang et al. [2008] tried to combine learning the hyperparameters for the latent/observation GPs with learning the true hidden states, effectively trying to perform smoothing and hyperparameter learning at the same time. To do this, Wang et al. [2008] use a model based on the Gaussian process latent variable model (GPLVM) [Lawrence, 2004] and find the distributions of $p(\boldsymbol{Y}|\boldsymbol{X})$ and $p(\boldsymbol{X})$ by marginalising out g and f respectively. Although Wang et al. [2008] only states these results, a similar derivation can be found in Frigola [2015, pp. 29-30] and we can use the relationship between the matrix and multivariate normal distributions to get the results in Wang et al. [2008]. Wang et al. [2008] seem to get some good results but the issues are that training can take a long time, the model is very prone to overfitting since we have a lot of freedom when minimising over all the hidden states (especially if the hidden states have a higher dimension than the observations) and the model only provides point-estimates rather than distributions for the parameters. However, it is not limited in its choice of covariance function unlike many of the filtering methods we discussed before.

Turner et al. [2010] use the GP-Gaussian filtering methods from Deisenroth et al. [2009] to learn a GPSSM via expectation-maximisation. In the original GP filtering/smoothing methods knowledge of the true hidden states are required but Turner et al. [2010] side-step this by introducing pseudo training sets (for both the observation and transition equations) that are learnt along side the covariance hyperparameters. The idea is that in the E-step we can filter using a GP-ADF which has been constructed using the pseudo-training set rather than the actual true hidden states (which we do not know) and in the M-step as well as optimising the covariance hyperparameters we also optimise the pseudo-training set. Then, in the standard EM way we iterate between the two steps waiting for convergence; however, due to the approximations required in the M-step, we do not have any guarantees that it will converge. A limiting factor for learning using this method is the restriction on possible choices of covariance functions inherited from Gaussian filtering with Gaussian processes as well as only providing point-estimates of parameters; although, unlike Wang et al. [2008] we do get a distribution over the hidden states (using GP-ADF or GP-RTSS).

A popular set of methods for learning GPSSMs are variational methods. In all these methods, the aim is to derive an approximation to the marginal likelihood $p(\boldsymbol{Y}_{1:T}|\boldsymbol{\theta})$ (i.e. marginalising out the hidden states) where $\boldsymbol{\theta}$ are the covariance hyperparameters and then optimising it; finding (approximately) the covariance hyperparameters with a trade off between data fit and model complexity. This is done by defining the evidence lower bound (ELBO) as:

$$\mathcal{L}(q) = \int q(\boldsymbol{Z}|\boldsymbol{\theta}) \log \frac{p(\boldsymbol{Y}_{1:T}, \boldsymbol{Z}|\boldsymbol{\theta})}{q(\boldsymbol{Z}|\boldsymbol{\theta})} d\boldsymbol{Z} \tag{2.124}$$

where $\boldsymbol{Z}$ is all the things we do not know (excluding the covariance hyperparameters $\boldsymbol{\theta}$); for example, the hidden states $\boldsymbol{X}_{1:T}$, functions f, g and any inducing variables for the sparse GP approximations. This is a lower bound to the marginal likelihood $p(\boldsymbol{Y}_{1:T}|\boldsymbol{\theta})$ and so we use it to approximate the marginal likelihood and optimise it with respect to q, any other variational parameters and $\boldsymbol{\theta}$. Optimising with respect to q without any constraints would give the optimal q as $p(\boldsymbol{Y}_{1:T}, \boldsymbol{Z}|\boldsymbol{\theta})$ resulting in an intractable integral. Hence, the different variational methods introduce different additional assumptions on q. Having decided on the additional assumptions about q, the ELBO is optimised using some numerical techniques; for example, gradient ascent. Damianou et al. [2011] use the GPLVMs applied to GPSSMs idea from Wang et al. [2008] but additionally marginalise out the hidden states $\boldsymbol{X}$ using variational techniques and as a result overfitting is less of a problem compared to Wang et al. [2008]. Other uses of variational inference include Frigola et al. [2014]; Frigola [2015] which provide two variational methods (although certain calculations require a particle smoother) that use sparse GPs and learn from challenging data reasonable well and relatively fast (couple of minutes for 500 data points [Frigola et al., 2014]). Also, they allow for online learning [Frigola, 2015, pp. 66-67]. The main issue with all these variational methods is that they are not fully Bayesian and only provide point estimates of the parameters. Furthermore, they require that the variational assumptions are approximately correct. Other variational models include the model by Eleftheriadis et al. [2017] that combines variational methods and recurrent neural networks in order to efficiently learn a GPSSM.

Another set of methods for learning GPSSMs are based on sequential Monte Carlo and in particular Particle Gibbs with Ancestor Sampling (PGAS); for example, both Svensson et al. [2016] and Frigola et al. [2013] use PGAS to learn the hyperparameters of a GPSSM. The models suggested in these two papers both use a fully Bayesian approach (i.e. we get posterior distributions over all the hyperparameters) with Svensson et al. [2016] improving the work of Frigola et al. [2013] by using the Hilbert reduced-rank approximation of the GP to speed up learning quite considerably.

We will now focus on the model by Svensson et al. [2016]. Here, we will describe the model and in the next chapters we will attempt to improve upon this model. Suppose we have a set of observations $\boldsymbol{y}_{1:T} = \boldsymbol{y}_1, ..., \boldsymbol{y}_T$, a reduced-rank GPSSM is a model of the form [Svensson et al., 2016]:

$$\boldsymbol{x}_t = \boldsymbol{A}\phi(\boldsymbol{x}_{t-1}) + \boldsymbol{w}_t \tag{2.125a}$$
$$\boldsymbol{y}_t \sim p(\boldsymbol{y}_t|\boldsymbol{x}_t) \tag{2.125b}$$
$$\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1) \tag{2.125c}$$
$$\boldsymbol{w}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}) \tag{2.125d}$$
$$\boldsymbol{Q} \sim \mathcal{IW}(l_Q, \boldsymbol{\Lambda}_Q) \tag{2.125e}$$
$$\boldsymbol{A}|\boldsymbol{Q} \sim \mathcal{MN}(\boldsymbol{A}|\boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{V}) \tag{2.125f}$$
$$\boldsymbol{V} = \text{diag}(S^{-1}(\sqrt{\lambda_1}), ..., S^{-1}(\sqrt{\lambda_m})) \tag{2.125g}$$
$$\boldsymbol{\theta} \sim p(\boldsymbol{\theta}) \tag{2.125h}$$

Notes:

1. S is the spectral density (2.45) of the chosen covariance function for the transition GP. As in Hilbert reduced-rank GPs, this covariance function must be stationary.
2. $\phi(\boldsymbol{x}_t) = [\phi_1(\boldsymbol{x}_t), ..., \phi_m(\boldsymbol{x}_t)]^T$ with ϕ_j, λ_j the eigenfunctions/eigenvalues used to approximate the covariance function. See (2.46)–(2.48) and (2.51) if required.
3. For simplicity, $p(\boldsymbol{y}_t|\boldsymbol{x}_t)$ and $p(\boldsymbol{x}_1)$ are assumed to be known; although, the model can be adapted to accommodate unknown parameters in these distributions as well.
4. $\mathcal{IW}$ is the inverse Wishart distribution and $\mathcal{MN}$ is the matrix-normal distribution described in (2.14).
5. The model comes from transforming the GPs to their reduced rank form using the Karhunen-Loève expansion: see (2.61).
6. $\boldsymbol{\theta}$ represents all the covariance hyperparameters and $p(\boldsymbol{\theta})$ is the prior distribution of $\boldsymbol{\theta}$.

Following Svensson et al. [2016], let

$$\boldsymbol{\Phi} = \sum_{t=1}^{T} \boldsymbol{\zeta}_t \boldsymbol{\zeta}_t^T \tag{2.126}$$
$$\boldsymbol{\Psi} = \sum_{t=1}^{T} \boldsymbol{\zeta}_t \boldsymbol{z}_t^T \tag{2.127}$$
$$\boldsymbol{\Sigma} = \sum_{t=1}^{T} \boldsymbol{z}_t \boldsymbol{z}_t^T \tag{2.128}$$
$$\tag{2.129}$$

where $\boldsymbol{z}_t = [\phi_1(\boldsymbol{x}_t), ..., \phi_m(\boldsymbol{x}_t)]^T$, $\boldsymbol{\zeta}_t = \boldsymbol{x}_{t+1}$. Then some posterior distributions of $\boldsymbol{A}$, $\boldsymbol{Q}$ and $\boldsymbol{\theta}$ given data $\boldsymbol{y}_{1:T} = \boldsymbol{y}_1, ..., \boldsymbol{y}_T$ are [Svensson et al., 2016]:

$$p(\boldsymbol{Q}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T}) = \mathcal{IW}(\boldsymbol{Q}|T + l_Q, \boldsymbol{\Lambda}_Q + \boldsymbol{\Phi} - \boldsymbol{\Psi}(\boldsymbol{\Sigma} + \boldsymbol{V})^{-1}\boldsymbol{\Psi}^T) \tag{2.130}$$

$$p(\boldsymbol{A}|\boldsymbol{Q}, \boldsymbol{x}_{1:T}\boldsymbol{y}_{1:T}) = \mathcal{MN}(\boldsymbol{A}|\boldsymbol{\Psi}(\boldsymbol{\Sigma}+\boldsymbol{V})^{-1}, \boldsymbol{Q}, (\boldsymbol{\Sigma}+\boldsymbol{V})^{-1}) \tag{2.131}$$

$$p(\boldsymbol{\theta}|\boldsymbol{Q}, \boldsymbol{A}, \boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T}) \propto p(\boldsymbol{\theta})p(\boldsymbol{Q}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})p(\boldsymbol{A}|\boldsymbol{Q}, \boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T}) \tag{2.132}$$

Algorithm 3: Learning of reduced-rank GPSSMs [Svensson et al., 2016]

Input : data $\boldsymbol{y}_{1:T}$, priors on $\boldsymbol{A}$, $\boldsymbol{Q}$, $\boldsymbol{\theta}$
Output: K samples with $p(\boldsymbol{x}_{1:T}, \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}|\boldsymbol{y}_{1:T})$ as invariant distribution.

1 Sample initial $\boldsymbol{x}_{1:T}[0], \boldsymbol{A}[0], \boldsymbol{Q}[0], \boldsymbol{\theta}[0]$;
2 **for** $k = 0, ..., K-1$ **do**
3 sample $\boldsymbol{x}_{1:T}[k+1]|\boldsymbol{A}[k], \boldsymbol{Q}[k], \boldsymbol{\theta}[k]$ using PGAS [Lindsten et al., 2014, p. 2160];
4 sample $\boldsymbol{Q}[k+1]|\boldsymbol{A}[k], \boldsymbol{\theta}[k], \boldsymbol{x}_{1:T}[k+1], \boldsymbol{\theta}[k]$ using (2.130);
5 sample $\boldsymbol{A}[k+1]|\boldsymbol{Q}[k+1], \boldsymbol{x}_{1:T}[k+1], \boldsymbol{\theta}[k]$ using (2.131);
6 sample $\boldsymbol{\theta}[k+1]|\boldsymbol{x}_{1:T}[k+1], \boldsymbol{A}[k+1], \boldsymbol{Q}[k+1]$ using Metropolis-Hastings
7 **end**
8 return all samples of $\boldsymbol{x}_{1:T}, \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}$

The training algorithm for reduced-rank GPSSMs is provided in Algorithm 3. From the output of this algorithm, we discard some of the early samples (burn-in) and then use the remaining samples to compute statistics of certain parameters such as the mean and variance. In particular, the mean and (marginal) variance of the predictive transition function given a test state $\boldsymbol{x}_*$ is [Svensson et al., 2016]:

$$\mathbb{E}[f_*(\boldsymbol{x}_*)] \approx \text{mean}[\boldsymbol{A}]\phi(\boldsymbol{x}_*) \tag{2.133}$$

$$\mathbb{V}[f_*(\boldsymbol{x}_*)] \approx \text{diag}(\text{cov}[\boldsymbol{A}]\phi(\boldsymbol{x}_*)\,\text{cov}[\boldsymbol{A}]^T) \tag{2.134}$$

where ϕ is as in (2.125a).

The state-of-the-art methods for learning GPSSMs all have a number of desirable properties; however, they vary in their expressiveness and how quickly they can learn. An ideal method for learning a GPSSM would have the following properties:

1. Allows for real-time online parameter learning.
2. Can efficiently learn from a wide range of possible datasets.
3. Can produce a predictive distribution of multi-step-ahead observations in real-time.
4. Robustness to outliers.
5. Allows for non-Gaussian latent and observation noise.
6. Can learn from large datasets in reasonable time.

The rest of this thesis will be focused on improving some of the state-of-the-art methods for learning GPSSMs by focusing on these ideal properties.

Chapter 3

Contributions

In this chapter, we introduce several novel contributions to the theory of Hilbert reduced-rank Gaussian processes and Hilbert reduced-rank Gaussian process state-space models.

3.1 Extending Hilbert reduced-rank Gaussian Processes

A key limitation of reduced-rank Gaussian processes is the difficulty in using anything more than the most basic covariance functions. In theory, the equations we described in (2.46)–(2.48) can be applied to any stationary covariance function but in practice it can be challenging to derive the spectral density. Moreover, although it is theoretically possible to extend the Hilbert reduced-rank GP to use any covariance function by changing the eigenfunctions and eigenvalues in (2.46), finding these eigenvalues and eigenfunctions is hard. To mitigate these issues we propose two solutions. Firstly, we introduce a new *adaptive* kernel that removes the need to find spectral densities while being well suited for use in reduced-rank GPs. Secondly, we combine the Hilbert reduced-rank GP model with the work of Calandra et al. [2016] to form the manifold-Hilbert reduced-rank Gaussian process model and this can introduce non-stationarity to stationary kernels.

3.1.1 The Adaptive Kernel

In their derivation of Hilbert reduced-rank GPs, Solin and Särkkä [2014] define the following inner-product and operator:

$$\langle f, g \rangle = \int f(\boldsymbol{x}) g(\boldsymbol{x}) w(\boldsymbol{x}) d\boldsymbol{x} \tag{3.1}$$

$$\mathcal{K} f(\boldsymbol{x}) = \int k(\boldsymbol{x}, \boldsymbol{x}') f(\boldsymbol{x}') w(\boldsymbol{x}') d\boldsymbol{x}' \tag{3.2}$$

with $w(\boldsymbol{x})$ some positive weight function such that $\int w(\boldsymbol{x})$ exists (i.e. is finite). Then they note that since $\mathcal{K}$ is self-adjoint with respect to the above inner-product (covariance functions are symmetric) via the spectral theorem there exists a decomposition

of $\mathcal{K}$ into eigenvalues and orthogonal (with respect to (3.1)) eigenfunctions. Solin and Särkkä [2014] use this to write the covariance function as

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{j_1,...,j_d} \gamma_{j_1,...,j_d} \phi_{j_1,...,j_d}(\boldsymbol{x}) \phi_{j_1,...,j_d}(\boldsymbol{x}') \tag{3.3}$$

for some positive eigenvalues $\gamma_{j_1,...,j_d} \in \mathbb{R}_{>0}$, orthogonal eigenfunctions $\phi_{j_1,...,j_d}$ and inputs $\boldsymbol{x} \in \mathbb{R}^d$. Then they try to find these eigenvalues/eigenfunctions for particular covariance functions.

We use (3.3) to motivate the construction of a new covariance function: the *adaptive* covariance function. Instead of trying to find the eigenvalues/eigenfunctions for a particular covariance function, we fix a set of orthogonal eigenfunctions and let the eigenvalues become covariance function hyperparameters. This allows us to create some expressive covariance functions that naturally fit into the Hilbert reduced-rank GP framework while eliminating the need to find spectral densities, which can be challenging for certain covariance functions.

Solin and Särkkä [2014] required the covariance functions to be isotropic-stationary because it allowed them to find the eigenvalues/eigenfunctions in a simple manner. Furthermore, it turns out that the same set of eigenfunctions can be used for all *stationary* covariance functions [Solin and Särkkä, 2014]. However, we note that any valid covariance function will be self-adjoint with respect to (3.1) and hence can be decomposed in the form of (3.3). As a result, given the right set of eigenfunctions, we can construct any covariance function. In the adaptive covariance, we fix the eigenfunctions and this means that we are restricting ourselves to a particular *family* of covariance functions. The corresponding eigenvalues are covariance hyperparameters and so the adaptive covariance function can adapt itself into any covariance function of this chosen family.

Definition 3.1 (The Adaptive Covariance Function)**.** For inputs $\boldsymbol{x} \in \mathbb{R}^d$ and given a fixed set of eigenfunctions $\{\phi_{j_1,...,j_d}; j_i = 1, ..., m_i; i = 1, ..., d;\}$ such that $\phi_{j_1,...,j_d} : \mathbb{R}^d \to \mathbb{R}$, the adaptive covariance function is:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{j_1,...,j_d=\mathbf{1}}^{\boldsymbol{m}} \gamma_{j_1,...,j_d} \phi_{j_1,...,j_d}(\boldsymbol{x}) \phi_{j_1,...,j_d}(\boldsymbol{x}') \tag{3.4}$$

where $\gamma_{j_1,...,j_d} > 0$ are hyperparameters, $\boldsymbol{m} = (m_1, ..., m_d)$ and $\mathbf{1} = (1, ..., 1) \in \mathbb{R}^d$. The $\boldsymbol{m}$ determines the complexity with larger m_i creating a more complex and expressive kernel with increased computational cost. As in (2.46), if $d > 1$ we can map $(j_1, ..., j_d) \to \mathbb{N}$ via (2.51). We also introduce a shorthand notation:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{\boldsymbol{j}=\mathbf{1}}^{\boldsymbol{m}} \gamma_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}) \phi_{\boldsymbol{j}}(\boldsymbol{x}') \tag{3.5}$$

where $\boldsymbol{j} = (j_1, ..., j_d)$. This equation is equivalent to (3.4).

Theorem 1. *The adaptive covariance function is a valid scalar-valued covariance function.*

Proof.

1. It is real: $k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{\boldsymbol{j}=\boldsymbol{1}}^{\boldsymbol{m}} \gamma_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}) \phi_{\boldsymbol{j}}(\boldsymbol{x}') \in \mathbb{R}$
2. It is symmetric:

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{x}') &= \sum_{\boldsymbol{j}=\boldsymbol{1}}^{\boldsymbol{m}} \gamma_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}) \phi_{\boldsymbol{j}}(\boldsymbol{x}') \\ &= \sum_{\boldsymbol{j}=\boldsymbol{1}}^{\boldsymbol{m}} \gamma_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}') \phi_{\boldsymbol{j}}(\boldsymbol{x}) \\ &= k(\boldsymbol{x}', \boldsymbol{x}) \end{aligned}$$

   since $\phi_{\boldsymbol{j}}(\boldsymbol{x}) \in \mathbb{R}$.
3. It is positive definite: given an arbitrary collection of inputs $\boldsymbol{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_N]$, the matrix $K = K(\boldsymbol{X}, \boldsymbol{X})$ with ij entry $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ can be written as $K = \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^T$ where $\boldsymbol{\Phi}$, $\boldsymbol{\Lambda}$ are defined in (2.49). Since the covariance hyperparameters (and hence eigenvalues) are constrained to be positive, we have that K is positive definite.

□

Learning a reduced-rank Gaussian process with this adaptive kernel is simple, given an eigenbasis (i.e. given $\phi_{\boldsymbol{j}}$) replace (2.46) with (3.4) and let the $\gamma_{j_1,...,j_d}$ in (3.4) become hyperparameters, then optimise the marginal likelihood [Solin and Särkkä, 2014, p. 10] with respect to the $\gamma_{j_1,...,j_d}$ and the noise parameters.

One useful family of eigenfunctions (the $\phi_{\boldsymbol{j}}$) for the adaptive kernel is the Fourier eigenbasis from (2.48). This leads to every stationary covariance function being a special case of our adaptive covariance function because any (piecewise-continuous) periodic covariance function can be decomposed into these eigenfunctions (Fourier sine series) and non-periodic (piecewise-continuous) functions can be decomposed into this form as long as we restrict the domain (represented by lengths L_k in (2.48)). This means for periodic covariance functions the lengths L_k in (2.48) should become additional parameters so we can find the periodicity automatically and in this case we should be able to extrapolate to data outside of the domain Ω (defined as in (2.48)). On the other hand, in the non-periodic case the L_k must be fixed in a manner that allows the domain Ω to encompass all the training and test data and we cannot extrapolate outside of Ω. This inability to extrapolate is usually not an issue, but as we shall see combining the *adaptive* covariance function with a manifold Gaussian process (mGP) [Calandra et al., 2016] can allow for extrapolation of non-periodic covariance functions.

3.1.2 The Manifold-Hilbert reduced-rank Gaussian Process

The manifold Gaussian process (mGP) uses a simple idea: given a valid covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$ where $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^d$ and an arbitrary function $M : \mathbb{R}^d \to \mathbb{R}^f$ then

$k(M(\boldsymbol{x}), M(\boldsymbol{x}'))$ is also a valid covariance function [Mackay, 1998]. Calandra et al. [2016] introduced the mGP and they used a neural network to create the mapping function while finding the neural network parameters during hyperparameter optimisation. It turns out that a combination of the reduced-rank GP, adaptive kernel and the mGP can produce some very interesting results. From the perspective of the reduced-rank GP, using the mGP is equivalent to having an adaptive eigenbasis:

$$\hat{\phi}_{j_1,\ldots,j_d} = \phi_{j_1,\ldots,j_d} \circ M \tag{3.6}$$

$$k_{\boldsymbol{m}}(\boldsymbol{x}, \boldsymbol{x}') \approx \sum_{(j_1,\ldots,j_d)=\mathbf{1}}^{\boldsymbol{m}} S(\sqrt{\lambda_{j_1,\ldots,j_d}})\hat{\phi}_{j_1,\ldots,j_d}(\boldsymbol{x})\hat{\phi}_{j_1,\ldots,j_d}(\boldsymbol{x}') \tag{3.7}$$

However, with this greatly increased flexibility comes an increased computational cost. We will refer to this combination of the mGP and reduced-rank GP as the manifold-Hilbert reduced-rank GP or the reduced-rank mGP.

3.1.3 Examples

In this section, we compare the Hilbert reduced-Rank GP and the manifold-Hibert reduced-Rank GP to the full GP (the standard case from (2.12)) across a variety of kernels. Throughout this set of examples, we will only consider univariate input and output GPs. The performance of the various models will be tested using the root mean square error (on test data) and the mean log likelihood of the test data. Both of these statistics are defined below.

Given a trained model and a test dataset $\{(x_i^*, y_i^*)$ for $i = 1, ..., K\}$, we can construct the posterior predictive distribution of $f_* = f(x_i^*)$ given the training data $\boldsymbol{X}, \boldsymbol{y}$ and a test input x_i^* using (2.29). Suppose that the posterior predictive distribution of $f_*|\boldsymbol{X}, \boldsymbol{y}, x_i^*$ is $\mathcal{N}(\mu_i^*, \sigma_i^{*2})$, then we define:

1. The root mean square error (RMSE) (smaller is better):

$$RMSE = \sqrt{\frac{1}{K}\sum_{i=1}^{K}(y_i^* - \mu_i^*)^2} \tag{3.8}$$

2. The mean log likelihood (LL) (larger is better):

$$LL = \frac{1}{K}\sum_{i=1}^{K}\log \mathcal{N}(y_i^*|\mu_i^*, \sigma_i^{*2}) \tag{3.9}$$

Finally, all the models will be given 10 optimisation restarts, all reduced-rank models will use 12 basis functions and the training time will be defined as the mean time for an optimisation restart.

Example 1:
Aim: It is known from Calandra et al. [2016] that the mGP can produce better results than the full GP; however, the mGP is slower than using the full GP. The aim of this example is to demonstrate that the manifold-Hilbert reduced-rank GP retains the properties of the original mGP but is now faster than the full GP.

Training Data: 2000 pairs (x, y) with each x generated by sampling from $\mathcal{N}(0, 1)$ and y generated using system:

$$y = \text{sgn}(x) + w_t \tag{3.10}$$
$$w_t \sim \mathcal{N}(0, 1^2) \tag{3.11}$$

Test Data: 300 pairs (x, y) with x equally spaced between -3 and 3 and y generated as above.

Additional Information: All the reduced-rank GP models will use a domain length $L = 6$ and all the manifold-Hilbert reduced-rank models will use a neural network with 3 hidden layers of 1, 6 and 2 neurons respectively with log-sigmoid activations.

Results:

Model	**Kernel**	**RMSE (on test data)**	**LL (on test data)**	**Mean Time (seconds)**
Full GP	RBF	1.10	-1.51	29.41
Full GP	Matern32	1.09	-1.51	34.64
Full GP	MLP	**1.04**	**-1.46**	160.00
Hilbert reduced-rank GP	RBF	1.10	-1.52	0.34
Hilbert reduced-rank GP	Matern32	1.10	-1.52	**0.31**
Hilbert reduced-rank GP	Adaptive	1.10	-1.52	0.95
Manifold-Hilbert reduced-rank GP	RBF	**1.04**	**-1.46**	5.49
Manifold-Hilbert reduced-rank GP	Matern32	**1.04**	**-1.46**	5.81
Manifold-Hilbert reduced-rank GP	Adaptive	**1.04**	**-1.46**	9.54

Table 3.1: Results for Example 1

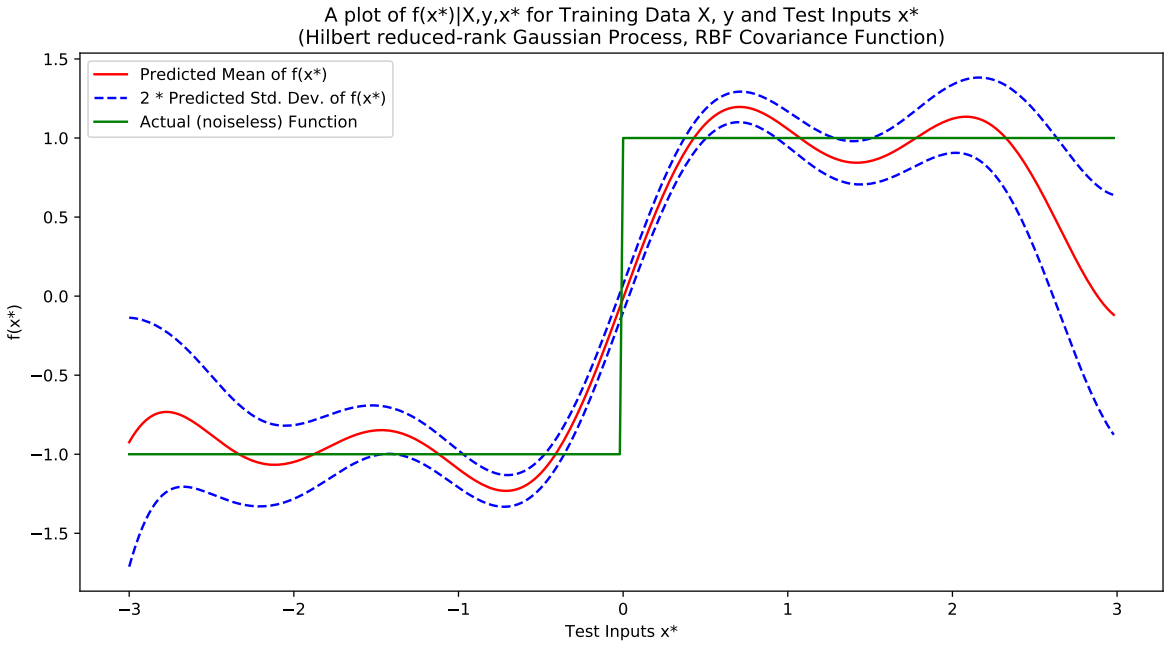

Figure 3.1: A Hilbert reduced-rank GP with an RBF covariance function (table 3.1, row 4)

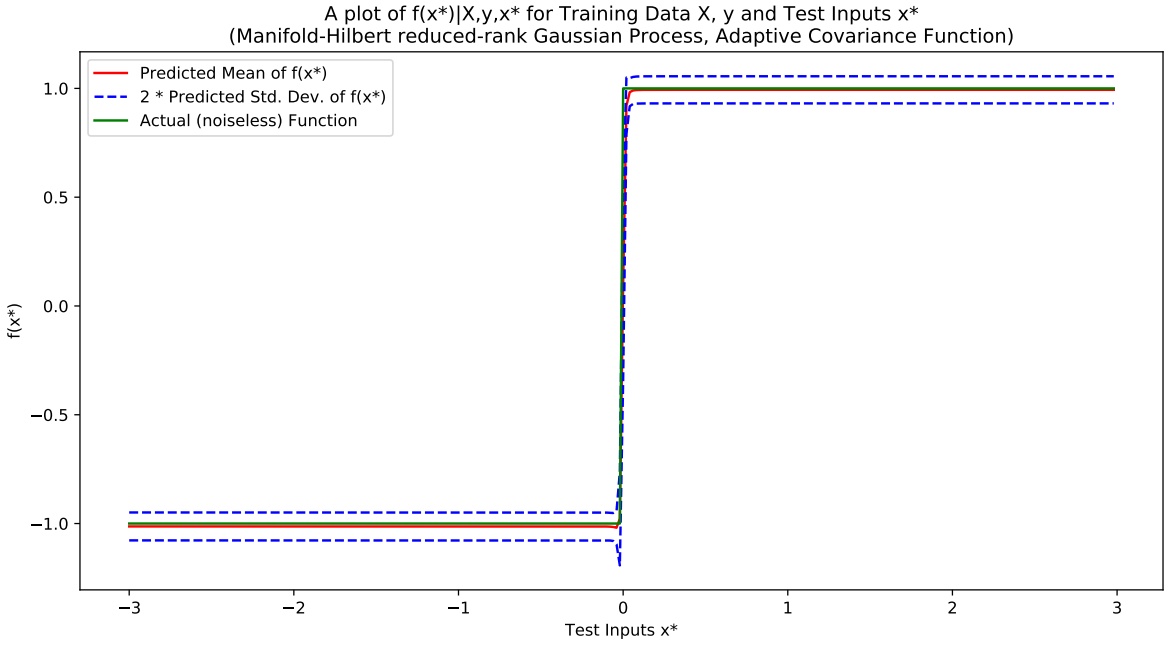

Figure 3.2: A manifold-Hilbert reduced-rank GP with an adaptive covariance function (table 3.1, row 9)

Example 2:
Aim: To demonstrate that the manifold-Hilbert reduced-rank GP can learn some challenging periodic functions.

Training Data: 1200 pairs (x, y) with x equally spaced between -15 and 15 and y generated using system:

$$y = \text{sgn}(\cos(0.8x)) + w_t \tag{3.12}$$
$$w_t \sim \mathcal{N}(0, 0.5^2) \tag{3.13}$$

Test Data: 1000 pairs (x, y) with x equally spaced between -50 and 50 and y generated as above.

Additional Information: All the reduced-rank GP models will use a variable domain length (i.e. we optimise the domain length) and all the manifold-Hilbert reduced-rank models will use a neural network with 3 hidden layers of 1, 6 and 2 neurons respectively with $\sin(x)$ activation functions.

Results:

Model	**Kernel**	**RMSE (on test data)**	**LL (on test data)**	**Mean Time (seconds)**
Full GP	RBF	0.99	-1.35	10.17
Full GP	Matern32	0.99	-1.33	6.71
Full GP	Periodic Exponential	0.56	-0.84	23.71
Hilbert reduced-rank GP	RBF	0.60	-0.92	0.37
Hilbert reduced-rank GP	Matern32	0.60	-0.92	**0.34**
Hilbert reduced-rank GP	Adaptive	0.60	-0.92	1.23
Manifold-Hilbert reduced-rank GP	RBF	**0.52**	**-0.76**	6.81
Manifold-Hilbert reduced-rank GP	Matern32	**0.52**	**-0.76**	7.48
Manifold-Hilbert reduced-rank GP	Adaptive	**0.52**	**-0.76**	8.12

Table 3.2: Results for Example 2

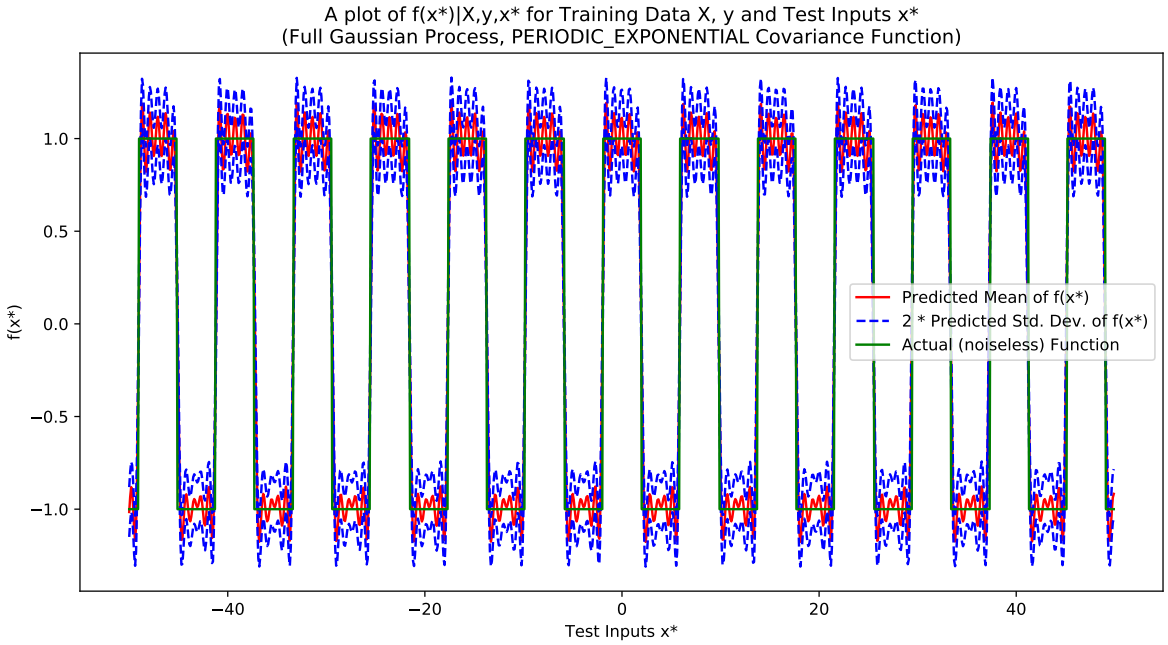

Figure 3.3: A full GP with a periodic exponential covariance function (table 3.2, row 3)

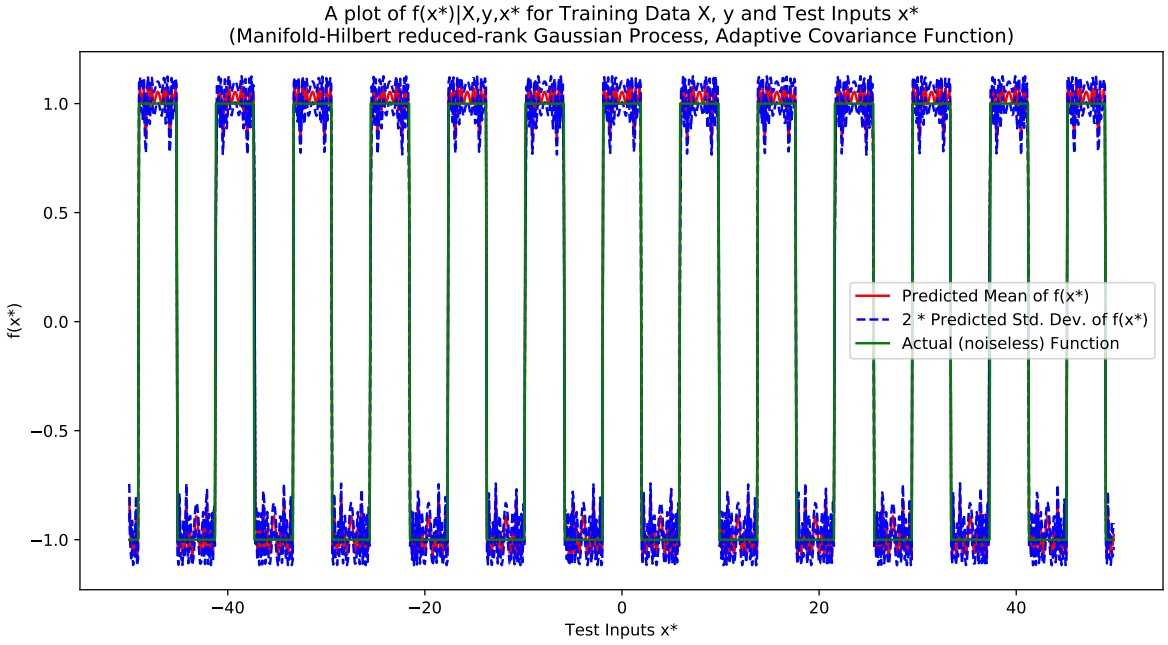

Figure 3.4: A manifold-Hilbert reduced-rank GP with an adaptive covariance function (table 3.2, row 9)

Example 3:
Aim: To demonstrate that the adaptive covariance function can sometimes give better results than the RBF or Matern32 covariance functions.

Training Data: 1000 pairs (x, y) with x equally spaced between -1 and 1 and y generated using system:

$$y = \left| \frac{sin(x)}{x} \right| + w_t \tag{3.14}$$

$$w_t \sim \mathcal{N}(0, 0.1^2) \tag{3.15}$$

Test Data: 1200 pairs (x, y) with x equally spaced between -3 and 3 and y generated as above.

Additional Information: All the reduced-rank GP models will use a variable domain length.

Results:

Model	**Kernel**	**RMSE (on test data)**	**LL (on test data)**	**Mean Time (seconds)**
Full GP	RBF	0.03	1.16	7.00
Full GP	Matern32	0.17	0.77	7.89
Full GP	Periodic Exponential	0.18	0.52	17.25
Hilbert reduced-rank GP	RBF	0.06	1.10	**0.54**
Hilbert reduced-rank GP	Matern32	0.12	0.96	**0.54**
Hilbert reduced-rank GP	Adaptive	**0.02**	**1.23**	2.89

Table 3.3: Results for Example 3

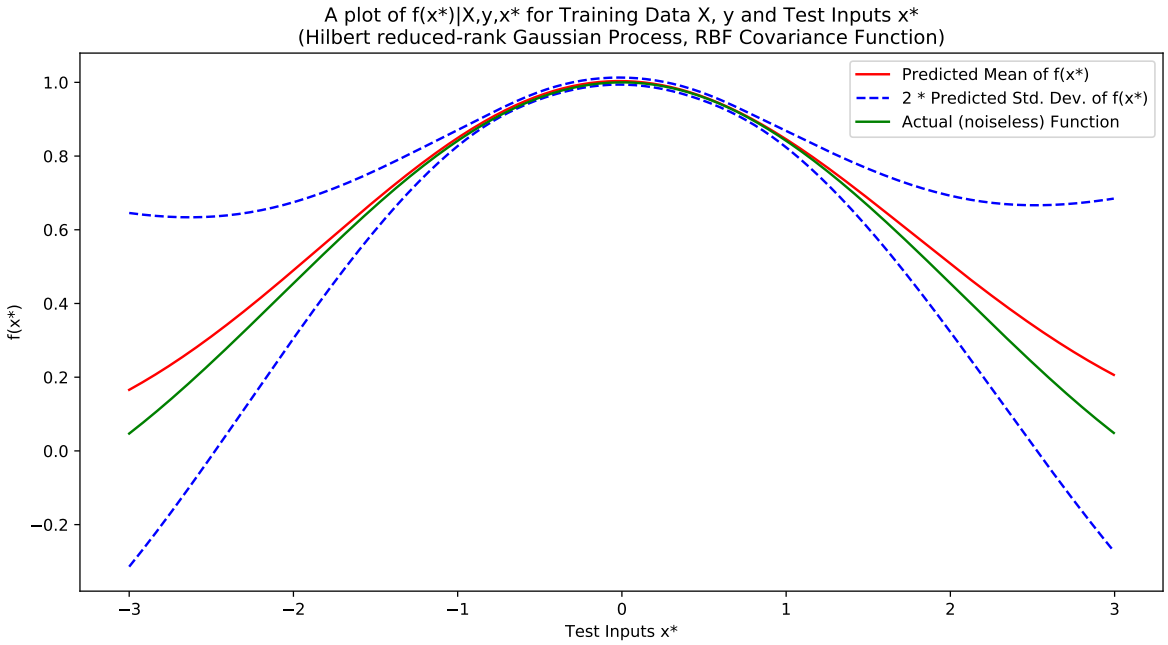

Figure 3.5: A Hilbert reduced-rank GP with an RBF covariance function (table 3.3, row 4)

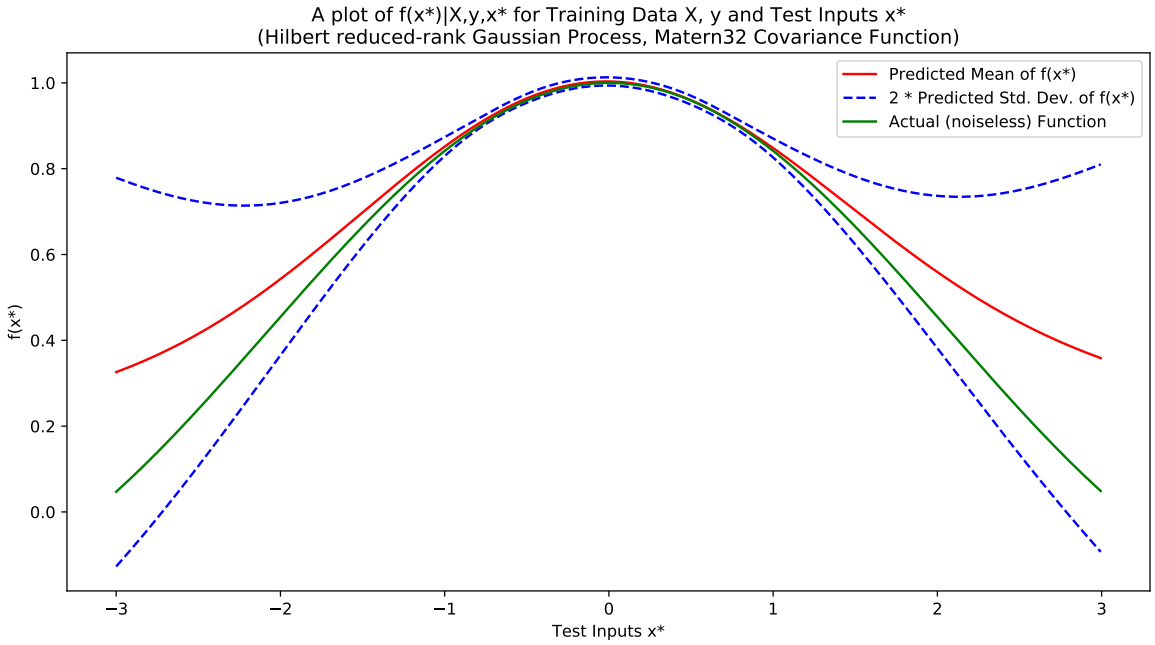

Figure 3.6: A Hilbert reduced-rank GP with a Matern32 covariance function (table 3.3, row 5)

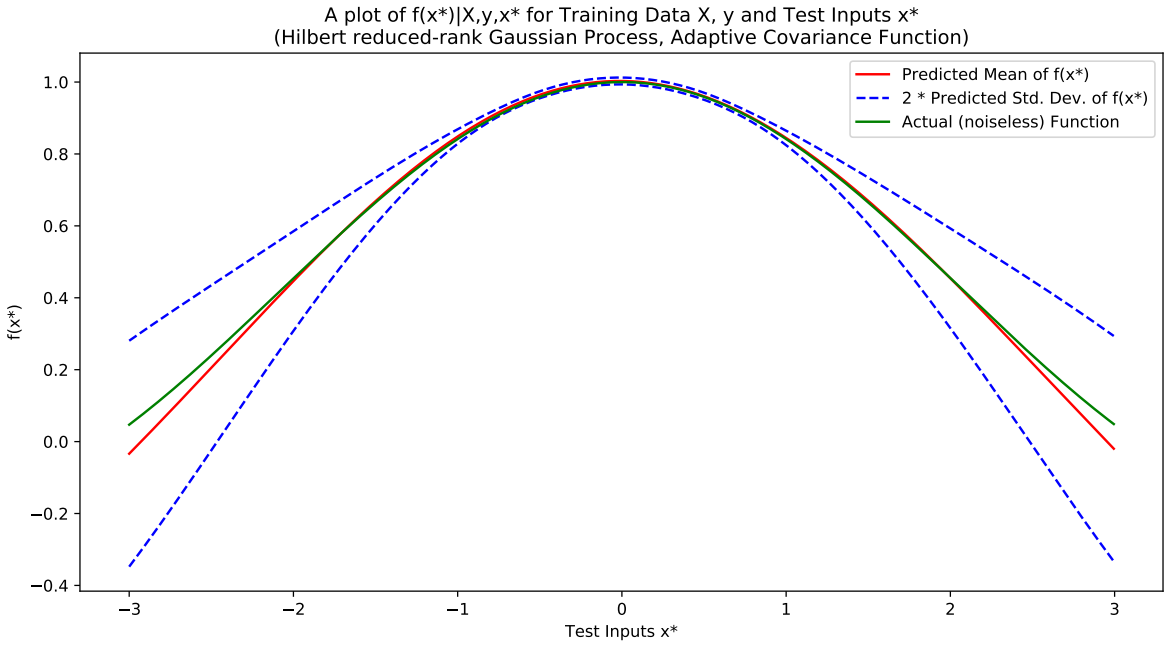

Figure 3.7: A Hilbert reduced-rank GP with an adaptive covariance function (table 3.3, row 6)

Discussion:
These examples demonstrate that using the adaptive covariance function does incur a computation cost; however, its performance rivals picking the best covariance function for the job. This suggests that it could be very useful for the latent GP in a GPSSM: we cannot observe any of the latent data so trying to pick the best covariance function can be challenging. These examples also show that the manifold-Hilbert reduced-rank GP has retained the properties of the original mGP but is much faster.

3.2 Extending Hilbert reduced-rank GPSSMs

The work by Svensson et al. [2016] suggests that reduced-rank GPSSMs could be very useful for time series analysis; however, there are several desirable features that the model by Svensson et al. [2016] lacks. In this section, we introduce several novel learning methods for reduced-rank GPSSMs such as online learning, forgetful online learning, distributed learning, learning with student-t distributed transition noise and learning with Student-t process state-space models (STPSSMs). All these new methods augment the model and allow it to be used in many new situations; for example, in reinforcement learning. Throughout this section, we will assume that the basic GPSSM model is the same as in (2.125).

3.2.1 Online Learning

Currently, the reduced-rank GPSSM by Svensson et al. [2016] does not allow for online learning and so here we present a novel algorithm for online learning in Hilbert reduced-rank GPSSMs. Online learning means being able to update the model when we receive a new data point without having to retrain the whole model; in a sense, online learning can be viewed as efficient updates in the presence of new data. These online updates are useful for applications where we do not receive all the data in one go (e.g. reinforcement learning) because GPSSMs do take time to train: around a few minutes for 1000 single-dimension observations. Without online learning, every time we added new data, we would have to spend some time waiting for the model to update and it would not be realistic to continuously repeat the process of receiving new data and updating the model. However, we propose a new method that reduces update times to less than a second (depending on a variety of factors).

Before we introduce our new algorithm, we require a part of the original algorithm by Svensson et al. [2016], which is summarised in Algorithm 4. Svensson et al. [2016] gave a suggestion on how the model might be adapted for online learning: use the posterior distribution at time t for the prior when we receive the new data at time $t+1$. For example, the update for (2.130) would be:

$$p(\boldsymbol{Q}|\boldsymbol{x}_{1:t+1}, \boldsymbol{y}_{1:t+1}) \propto p(\boldsymbol{x}_{t+1}, \boldsymbol{y}_{t+1}|\boldsymbol{Q}, \boldsymbol{x}_{1:t}, \boldsymbol{y}_{1:t}) p(\boldsymbol{Q}|\boldsymbol{x}_{1:t}, \boldsymbol{y}_{1:t}) \tag{3.16}$$

However, there is no point trying to make this distribution update efficient because we have to completely recompute (2.126)–(2.128) since we have a new set of states from the PGAS sample. A similar situation occurs for the other posterior distributions (2.131) and (2.132). Since we know (2.130)–(2.131) for arbitrary t and since there no need to produce efficient updates, we can just use the same distributions as before. However, the quality of the samples from the posterior distributions will increase as the amount of data increases and so we need to introduce sample weights. For our model, a given sample's weight is equal to the number of data points used for creating that sample and with online learning these weights will increase over time. When we compute various statistics based on the samples (e.g. mean or variance), we must take the weights into account i.e. calculate the mean/variance with samples weighted using these given weights.

The Online Learning Algorithm
The algorithm is split into two parts; firstly, the inner loop (Algorithm 4), which samples from the latent space and parameter space given the observations and secondly, the main learning algorithm (Algorithm 5) which combines Algorithm 4 with our method for updating the GPSSM given new data.

Tuning Parameters
In addition to the tuning parameters for the model by Svensson et al. [2016], there is one tuning parameter for online learning: the number of update samples. This is the number of times to run the sampling algorithm (Algorithm 4) when we receive a new data point. We recommend only using one sampling round per data update unless the data update provides a significant amount of extra data (≥ 5 pieces of data). In a similar way to the model by Svensson et al. [2016], we assume that the initial distribution of the latent state $\boldsymbol{x}_1$ is known but this has very little importance so if we do not know the initial distribution, we set it to $\mathcal{N}(\mathbf{0}, 100\boldsymbol{I})$ for a suitably sized identity matrix $\boldsymbol{I}$.

Using the Posterior Samples
As the online learning algorithm (Algorithm 5) progresses, we have some samples $\boldsymbol{A}[1:K], \boldsymbol{Q}[1:K], \boldsymbol{\theta}[1:K]$ and weights$[1:K]$. Let $\omega[k]$ for $k = 1, ..., K$ be the normalised weights, then useful statistics such as the predictive mean and (marginal) variance of the transition function f given a test state $\boldsymbol{x}_*$ can be computed as:

$$\text{mean}[A] = \sum_{k=\text{burn-in}}^{K} \boldsymbol{A}[k]\omega[k] \tag{3.17}$$

$$\mathbb{E}[f_*(\boldsymbol{x}_*)] \approx \text{mean}[A]\phi[\boldsymbol{x}_*] \tag{3.18}$$

$$\text{cov}[A] = \left(\sum_{k=\text{burn-in}}^{K} \omega[k]\boldsymbol{A}[k]\boldsymbol{A}[k]^T \right) - \text{mean}[A]\,\text{mean}[A]^T \tag{3.19}$$

$$\mathbb{V}[f_*(\boldsymbol{x}_*)] \approx \text{diag}(\text{cov}[\boldsymbol{A}]\phi[\boldsymbol{x}_*]\,\text{cov}[\boldsymbol{A}]^T) \tag{3.20}$$

where ϕ is as in (2.125a). This can all be computed and updated during the algorithm letting us see how the model learns as it receives more data.

3.2.2 Forgetful Online Learning

Another type of online learning is *forgetful online learning* whereby while we add data, we also lose some data. This can be useful for systems that are evolving over time and is effective at keeping the amount of data in the system constant. This can be helpful because as the amount of data in the GPSSM increases, a slow down will be experienced. However, even if we start losing some past data, we do not lose all the past information because we retain all the latent state and parameter samples. The forgetful online learning algorithm uses a similar idea to the original online learning algorithm but it has to take into account that we lose information. As a result, there is an additional tuning parameter called the *Maximum Memory Length*:

this is the maximum amount of data to use for each call to InnerLoop (Algorithm 4). The forgetful online learning algorithm is summarised in Algorithm 6 and for performance statistics we can do something similar to the online learning case. For simplicity, we will keep number of samples per update (see tuning parameters for online learning) at one but the algorithm can be extended to multiple samples per update.

3.2.3 Distributed Learning

It is simple to extend all the above algorithms to make use of multicore/multiprocessor machines and this is summarised in Algorithm 8. Let $\omega^i[k]$ for $k = 1, ..., K$ be the normalised (over both k and i) weights for each thread i, then useful statistics such as the predictive mean and (marginal) variance of the transition function can be computed as:

$$\text{mean}[\boldsymbol{A}] = \sum_{i=1}^{\#threads} \left(\sum_{k=\text{burn-in}}^{K} \boldsymbol{A}^i[k]\omega^i[k] \right) \tag{3.21}$$

$$\mathbb{E}[f_*(\boldsymbol{x}_*)] \approx \text{mean}[\boldsymbol{A}]\phi[\boldsymbol{x}_*] \tag{3.22}$$

$$\text{cov}[\boldsymbol{A}] = \sum_{i=1}^{\#threads} \left(\sum_{k=\text{burn-in}}^{K} \boldsymbol{A}^i[k](\boldsymbol{A}^i[k])^T\omega^i[k] \right) - \text{mean}[A]\,\text{mean}[A]^T \tag{3.23}$$

$$\mathbb{V}[f_*(\boldsymbol{x}_*)] \approx \text{diag}(\text{cov}[\boldsymbol{A}]\phi[\boldsymbol{x}_*]\,\text{cov}[\boldsymbol{A}]^T) \tag{3.24}$$

where ϕ is as in (2.125a) and $\boldsymbol{x}_*$ is a test point.

Algorithm 4: Inner Loop (based on Svensson et al. [2016])

Input : $\boldsymbol{x}_{1:t}[k]$, $\boldsymbol{A}[k]$, $\boldsymbol{Q}[k]$, $\boldsymbol{\theta}[k]$, $\boldsymbol{y}_{1:t}$, InitialDistribution
Output: $\boldsymbol{x}_{1:t}[k+1]$, $\boldsymbol{A}[k+1]$, $\boldsymbol{Q}[k+1]$, $\boldsymbol{\theta}[k+1]$

1 sample $\boldsymbol{x}_{1:t}[k+1]|\boldsymbol{A}[k], \boldsymbol{Q}[k], \boldsymbol{\theta}[k]$ using PGAS [Lindsten et al., 2014, p. 2160] with observations $\boldsymbol{y}_{1:t}$, reference trajectory $\boldsymbol{x}_{1:t}[k]$ and initial distribution: InitialDistribution;
`/*` $\boldsymbol{A}[k], \boldsymbol{Q}[k]$ `with (2.125a) gives us the transition distribution for PGAS and (2.125b) (which is assumed to be completely known) gives us the observation distribution. */`
2 sample $\boldsymbol{Q}[k+1]|\boldsymbol{A}[k], \boldsymbol{\theta}[k], \boldsymbol{x}_{1:t}[k+1], \boldsymbol{\theta}[k]$ using (2.130) with $T = t$;
3 sample $\boldsymbol{A}[k+1]|\boldsymbol{Q}[k+1], \boldsymbol{x}_{1:t}[k+1], \boldsymbol{\theta}[k]$ using (2.131) with $T = t$;
4 sample $\boldsymbol{\theta}[k+1]|\boldsymbol{x}_{1:t}[k+1], \boldsymbol{A}[k+1], \boldsymbol{Q}[k+1]$ using (2.132) with $T = t$ and Metropolis-Hastings;
5 **return** $\boldsymbol{x}_{1:t}[k+1]$, $\boldsymbol{A}[k+1]$, $\boldsymbol{Q}[k+1]$, $\boldsymbol{\theta}[k+1]$;

Algorithm 5: Online learning of a reduced-rank GPSSM

Input : Initial data $\boldsymbol{y}_{1:t}$, Number of samples per update, InitialDistribution
Output: K samples with the k^{th} sample having $p(\boldsymbol{x}_{1:t_k}, \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}_A | \boldsymbol{y}_{1:t_k})$ as the invariant distribution.

initialise $\boldsymbol{x}_{1:t}, \boldsymbol{A}[0], \boldsymbol{Q}[0], \boldsymbol{\theta}[0]$ randomly;
set weights[0] = t;
set $t_0 = t$;
set $\boldsymbol{x}^0_{1:t_0} = \boldsymbol{x}_{1:t}$;
set k = 1;
while $k \leq K$ **do**
    newdata = getData();
    i = number of new data pieces;
    $t_k = t_{k-1} + i$;
    append the new data to the old data: $\boldsymbol{y}_{1:t_k} = (\boldsymbol{y}_{1:t_{k-1}}$, new data);
    for *r = 1, ..., i* **do**
        sample newstates$[r]$ from $\mathcal{N}(\bar{\boldsymbol{x}}_{t_{k-1}}, \boldsymbol{I}_{d_{Latent}})$ where $\bar{\boldsymbol{x}}_{t_{k-1}}$ is the state at time t_{k-1} (weighted average over all samples using weights$[k-1]$) and d_{Latent} is the dimension of the latent space;
        `/* Using` $\boldsymbol{I}_{d_{Latent}}$ `rather than the sample covariance ensures that we always have some exploration and this helps PGAS. */`
    end
    set the new reference trajectory: $\hat{\boldsymbol{x}}_{1:t_k} = (\boldsymbol{x}^{k-1}_{1:t_{k-1}}$, newstates);
    $\boldsymbol{x}^k_{1:t_k}$, $\boldsymbol{A}[k]$, $\boldsymbol{Q}[k]$, $\boldsymbol{\theta}[k]$ = InnerLoop($\hat{\boldsymbol{x}}_{1:t_k}$, $\boldsymbol{A}[k-1]$, $\boldsymbol{Q}[k-1]$, $\boldsymbol{\theta}[k-1]$, $\boldsymbol{y}_{1:t_k}$, InitialDistribution);
    weights$[k] = t_k$;
    for *r = 1, ..., Number of samples per update - 1* **do**
        $t_{k+r} = t_{k-1+r}$;
        weights$[k+r] = t_{k+r}$;
        $\boldsymbol{x}^{k+r}_{1:t_{k+r}}$, $\boldsymbol{A}[k+r]$, $\boldsymbol{Q}[k+r]$, $\boldsymbol{\theta}[k+r]$ = InnerLoop($\boldsymbol{x}^{k+r-1}_{1:t_{k+r-1}}$, $\boldsymbol{A}[k+r-1]$, $\boldsymbol{Q}[k+r-1]$, $\boldsymbol{\theta}[k+r-1]$, $\boldsymbol{y}_{1:t_k}$, InitialDistribution);
    end
    $k = k$+Number of samples per update;
end
return $\boldsymbol{x}^1_{1:t_1}, ..., \boldsymbol{x}^K_{1:t_K}, \boldsymbol{A}[1:K], \boldsymbol{Q}[1:K], \boldsymbol{\theta}[1:K]$ and weights$[1:K]$.

Algorithm 6: Forgetful online learning of reduced-rank GPSSM

Input : Initial data $\boldsymbol{y}_{1:t}$, Maximum Memory Length
Output: K samples with the k^{th} sample having $p(\boldsymbol{x}_{t_k^{start}:t_k^{end}}, \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta} | \boldsymbol{y}_{t_k^{start}:t_k^{end}})$ as the invariant distribution.

Initialise $\boldsymbol{x}_{1:t}, \boldsymbol{A}[0], \boldsymbol{Q}[0], \boldsymbol{\theta}[0]$ randomly;
weights[0] = t;
$(t_0^{start}, t_0^{end}) = (1, t)$;
$\boldsymbol{x}^0_{t_0^{start}:t_0^{end}} = \boldsymbol{x}_{1:t}$;
for $k = 1, ..., K$ **do**
  newdata = getData();
  i = number of new data pieces;
  $t_k^{end} = t_{k-1}^{end} + i$;
  $t_k^{start} = \max(t_k^{end} -$ Maximum Memory Length $, 0)$ + 1;
  weights$[k] = t_k^{end} - t_k^{start}$;
  append the new data to the old data: $\boldsymbol{y}_{1:t_k^{end}} = (\boldsymbol{y}_{1:t_{k-1}^{end}},$ new data$)$;
  for *r = 1, ..., i* **do**
    sample newstates$[r]$ from $\mathcal{N}(\bar{\boldsymbol{x}}_{t_{k-1}}, \boldsymbol{I}_{d_{Latent}})$ where $\bar{\boldsymbol{x}}_{t_{k-1}}$ is the state at time t_{k-1}^{end} (weighted average over all samples using weights$[k-1]$) and d_{Latent} is the dimension of the latent space;
    /* Using $\boldsymbol{I}_{d_{Latent}}$ rather than the sample covariance ensures that we always have some exploration and this helps PGAS. */
  end
  $\boldsymbol{y}^* = \boldsymbol{y}_{t_k^{start}:t_k^{end}}$;
  $\boldsymbol{x}^* = (\boldsymbol{x}^{k-1}_{t_k^{start}:t_{k-1}^{end}},$ newstates$)$;
  $\mu = \text{mean}(\boldsymbol{x}_{t_k^{start}})$;
  $\Sigma = \text{cov}(\boldsymbol{x}_{t_k^{start}})$;
  /* We compute the mean and variance using the samples of the latent states at time t_k^{start}. Some of the latent state samples will not have a sample for time t_k^{start} and so they are ignored. */
  NewInitialDistribution = $\mathcal{N}(\mu, \Sigma)$
  $\boldsymbol{x}^k_{t_k^{start}:t_k^{end}} | \boldsymbol{A}[k], \boldsymbol{Q}[k], \boldsymbol{\theta}[k] =$ Inner Loop($\boldsymbol{x}^*$, $\boldsymbol{A}[k-1]$, $\boldsymbol{Q}[k-1]$, $\boldsymbol{\theta}[k-1]$, $\boldsymbol{y}^*$, NewInitialDistribution);
end
return $(\boldsymbol{x}^1_{t_1^{start}:t_1^{end}}, ..., \boldsymbol{x}^K_{t_K^{start}:t_K^{end}}, \boldsymbol{A}[1:K], \boldsymbol{Q}[1:K], \boldsymbol{\theta}[1:K]$ and weights$[1:K]$.

Algorithm 7: Distributed learning of a reduced-rank GPSSM

Input : Data $\boldsymbol{y}_{1:T}$, #threads

Output: K samples with $p(\boldsymbol{x}_{1:T}, \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta} | \boldsymbol{y}_{1:T})$ as the invariant distribution.

split the observed data $\boldsymbol{y}_{1:T}$ into #threads chunks of approximately equal size;

/* One way to split is sequentially i.e. first chunk is $\boldsymbol{y}_{1:t_1}$, second is $\boldsymbol{y}_{t_1+1:t_2}$ with $t_1 < t_2$. An alternative way is ordered random splitting in which each chunk contains a random ordered set of $\boldsymbol{y}_{1:T}$; for example, if the first chunk is $\boldsymbol{y}_{t_1}, \boldsymbol{y}_{t_2}, ...\boldsymbol{y}_{t_m}$ then the times must satisfy $t_1 < t_2 < ... < t_m$. Usually we would expect the chunks of data to be disjoint; however, this is not a requirement. */

let these data chunks be $\boldsymbol{y}^1, ..., \boldsymbol{y}^{\#threads}$;

for *each* $\boldsymbol{y}^i$ **in parallel do**

  $\boldsymbol{x}^i_{1:T_i}[1:K], \boldsymbol{A}^i[1:K], \boldsymbol{Q}^i[1:K], \boldsymbol{\theta}^i[1:K]$ = LearningAlgorithm($\boldsymbol{y}^i$, priors on $\boldsymbol{A}$, $\boldsymbol{Q}$, $\boldsymbol{\theta}$);

  /* LearningAlgorthim can either be the standard reduced rank learning of Svensson et al. [2016] or our online/forgetful online learning algorithm. For the chosen algorithm the observed data will be the $\boldsymbol{y}^i$ only and the prior distributions should be the same for each thread. */

  if not known already calculate the weight of each sample;

  /* For online/forgetful online learning we will already have a set of weights but for the standard algorithm there are no sample weights so we set the weights of each sample equal to the number of data points in the corresponding data chunk. */

  discard burn in samples;

  /* This *must* be done for each thread. */

end

return weight-sample pairs of $\boldsymbol{A}$, $\boldsymbol{Q}$ and $\boldsymbol{\theta}$ for all threads.

3.2.4 GPSSMs with Student-t noise

In this section, we extend the model of Svensson et al. [2016] to allow for Student-t noise in the latent states. Learning with Student-t noise is more challenging than with Gaussian noise because we no longer have closed form posterior distributions for $p(\boldsymbol{Q}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$ or $p(\boldsymbol{A}|\boldsymbol{Q}, \boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$. As a result, we either have to use Monte Carlo methods to sample from these posterior distributions or variational methods to find a closed form approximation to these posterior distributions that we can sample from. Since we wanted to keep the same structure as the model by Svensson et al. [2016], it seemed reasonable to try and find a Monte Carlo solution.

Definition 3.2. [Roth, 2013]
Let $\boldsymbol{X} \in \mathbb{R}^d$ be a multivariate-t distributed random variable with mean $\boldsymbol{\mu} \in \mathbb{R}^d$, symmetric positive definite scale parameter $\boldsymbol{\Sigma} \in \mathbb{R}^{d\times d}$ and $\nu \in \mathbb{R}_{>0}$ degrees of freedom. The probability density of $\boldsymbol{X}$ at the point $\boldsymbol{x} \in \mathbb{R}^d$ is :

$$\mathcal{T}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \nu) = \frac{\Gamma\left(\frac{\nu+d}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{(\nu\pi)^{\frac{d}{2}}|\boldsymbol{\Sigma}|^{\frac{1}{2}}} \left(1 + \frac{1}{\nu}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)^{-\frac{\nu+d}{2}} \tag{3.25}$$

A key property of the multivariate-t distribution is that when the degrees of freedom ν tends to infinity, the distribution tends to a multivariate normal distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$.

A GPSSM with Student-t noise can be written as follows:

$$\boldsymbol{x}_t = \boldsymbol{A}\phi(\boldsymbol{x}_{t-1}) + \boldsymbol{w}_t \tag{3.26a}$$
$$\boldsymbol{y}_t \sim p(\boldsymbol{y}_t|\boldsymbol{x}_t) \tag{3.26b}$$
$$\boldsymbol{w}_t \sim \mathcal{T}(\boldsymbol{0}, \boldsymbol{Q}, \nu) \tag{3.26c}$$
$$\boldsymbol{A}|\boldsymbol{Q} \sim \mathcal{MN}(\boldsymbol{A}|\boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{V}) \tag{3.26d}$$
$$\boldsymbol{Q} \sim \mathcal{IW}(l_Q, \boldsymbol{\Lambda}_Q) \tag{3.26e}$$

where $\boldsymbol{V}$, ϕ and the other terms are defined in (2.125a). We will now present an algorithm which uses the same Gibbs structure as the model by Svensson et al. [2016] but has different MCMC steps inside the Gibbs sampler to deal with the Student-t likelihood.

The first step is to find a suitable method of sampling from $p(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}|\boldsymbol{x}_{1:T})$ ($\boldsymbol{\theta}$ is the kernel hyperparameters). To do this, the main options are either to try and sample from $p(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}|\boldsymbol{x}_{1:T})$ directly or to split it up into parts and use Gibbs sampling. However, unlike Svensson et al. [2016] who had closed form posterior distributions, splitting $p(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}|\boldsymbol{x}_{1:T})$ can cause us issues because traditional Monte Carlo methods, such as Metropolis-Hastings or hybrid Monte Carlo, and even newer methods, such as slice-sampling, can have poor performance when sampling from high-dimensional posteriors i.e. they can be slow at producing good samples. So either we split the $p(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}|\boldsymbol{x}_{1:T})$ into many low-dimensional parts and get very slow mixing in the Gibbs sampler or we split it into few high-dimensional parts and still get slow mixing because we are sampling from high-dimensional distributions. Note

that even if the latent dimension $d = 1$, then we still need to sample $\boldsymbol{A}$ which will have m parameters if we use m basis functions. We tried Metropolis-Hastings, hybrid Monte Carlo and slice-sampling with none of them producing good results.

However, we were able to get good results by sampling from $p(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}|\boldsymbol{x}_{1:T})$ directly using a recent MCMC method called the Affine-Invariant Ensemble Sampler which was originally developed by Goodman and Weare [2010] with a practical implementation (called EMCEE) produced by Foreman-Mackey et al. [2013]. EMCEE has several advantages compared with other methods; for example, it only has a few tuning parameters (two single dimensional parameters for most cases) unlike Metropolis-Hastings or its variants that have a performance highly dependant on the proposal distribution. Furthermore, EMCEE can give a measure of whether the drawn samples are 'good' samples from the distribution.

The Affine-Invariant Ensemble Sampler algorithm is relatively complex and is discussed in depth by Goodman and Weare [2010] and Foreman-Mackey et al. [2013]. Here, we will only give a brief overview of the EMCEE algorithm. Suppose we want to produce samples from the distribution $p(\boldsymbol{x})$ but only have access to $\hat{p}(\boldsymbol{x})$ where $p(\boldsymbol{x}) = \frac{1}{Z}\hat{p}(\boldsymbol{x})$ and Z does not depend on $\boldsymbol{x}$. In this case, the EMCEE algorithm takes two (main) arguments: $\log \hat{p}(\boldsymbol{x})$ and the initial positions of the 'walkers'. The walkers can be seen as a similar idea to particles in SMC methods with more walkers leading to samples that more accurately represent the distribution. Each of the walkers perform their own semi-independent acceptance sampling and have their own Markov chain; however, more walkers allows EMCEE to better explore the sample space. One method suggested by Foreman-Mackey et al. [2013] to initialise the walkers is to start all the walkers very close to the point $\boldsymbol{x}^*$ which maximises $\log \hat{p}(\boldsymbol{x})$ (although the walkers must start in different places so random noise is added). After a brief burn-in period, each walker can start to generate samples from $p(\boldsymbol{x})$. Finally, the EMCEE algorithm provides a indicator of whether the samples are 'good' known as the acceptance fraction and ideally this should be between 0.2 and 0.5 (see Foreman-Mackey et al. [2013] for more details).

In order to use EMCEE for learning GPSSMs with Student-t latent noise we note:

$$p(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}|\boldsymbol{x}_{1:t}) \propto p(\boldsymbol{x}_{1:t}|\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta})p(\boldsymbol{A}|\boldsymbol{Q}, \boldsymbol{\theta})p(\boldsymbol{Q}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{3.27}$$

$$\propto p(\boldsymbol{x}_1)\prod_{k=2}^{t} p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta})p(\boldsymbol{A}|\boldsymbol{Q}, \boldsymbol{\theta})p(\boldsymbol{Q}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{3.28}$$

With Student-t noise the latent distribution is:

$$p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, \boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta}) = \mathcal{T}(\boldsymbol{x}_k|\boldsymbol{A}\phi(\boldsymbol{x}_{k-1}), \boldsymbol{Q}, \nu) \tag{3.29}$$

Hence we can write that:

$$\log p(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta} | \boldsymbol{x}_{1:t}) = Z + \sum_{k=2}^{t} \log \mathcal{T}(\boldsymbol{x}_k | \boldsymbol{A}\phi(\boldsymbol{x}_{k-1}), \boldsymbol{Q}, \nu) + \log \mathcal{MN}(\boldsymbol{A} | \boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{V}) + \log \mathcal{IW}(\boldsymbol{Q} | l_Q, \boldsymbol{\Lambda}_Q) + \log p(\boldsymbol{\theta}) \quad (3.30)$$

Therefore, to produce samples from $p(\boldsymbol{A}, \boldsymbol{Q}, \boldsymbol{\theta} | \boldsymbol{x}_{1:t})$ using EMCEE we use:

$$\log \hat{p} = \sum_{k=2}^{t} \log \mathcal{T}(\boldsymbol{x}_k | \boldsymbol{A}\phi(\boldsymbol{x}_{k-1}), \boldsymbol{Q}, \nu) + \log \mathcal{MN}(\boldsymbol{A} | \boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{V}) \quad (3.31)$$

$$+ \log \mathcal{IW}(\boldsymbol{Q} | l_Q, \boldsymbol{\Lambda}_Q) + \log p(\boldsymbol{\theta}) \quad (3.32)$$

In many ways, the algorithm for learning GPSSMs with Student-t noise in the latent states is simple: all we do is switch between sampling the latent states using PGAS and sampling the parameters via EMCEE with occasional optimisation steps in order to initialise the EMCEE walkers. There is only one additional tuning parameter for this algorithm compared with that of Svensson et al. [2016]: the optimisation distance. This optimisation distance controls the number of sampling iterations before optimising $\log \hat{p}$ again. Based on the examples we have tried, an optimisation distance of 5 is recommended since this balances speed (the optimisation step is slower than a sampling step) with performance.

Another thing to mention about our algorithm is that we only use two walkers in each call to EMCEE. Although this may seem at odds with what Foreman-Mackey et al. [2013] recommend (the number of walkers should be at least twice the dimension of the sample space), we are implicitly using more walkers since we initialise the start point at each call to EMCEE rather than using the last position of the previous walkers. Also, the number of samples we require at each stage is very small so there is little need for a large number of walkers (which would be able to produce a *large number* of quasi-independent samples that accurately represent the distribution).

Finally, we make use of automatic differentiation to find the gradient of (3.32) and hence we can use a gradient based optimisation method: this is much faster than a non-gradient based method such as Nelder-Mead. Our choice of optimisation method is L-BFGS-B because it allows for the use of bounds which is particularly important for several parameters. We can now write down the algorithm for learning GPSSMs with Student-t latent noise: see Algorithm 8. Furthermore, it is quite easy to adapt this algorithm to online learning: just use the same idea as in Algorithm 5.

Algorithm 8: Learning a GPSSM with Student-t latent noise

Input : Observed data $\boldsymbol{y}_{1:T}$, InitialDistribution $p(\boldsymbol{x}_1)$, *OptimisationDistance*.
Output: $\boldsymbol{x}_{1:T}[k]$, $\boldsymbol{A}[k]$, $\boldsymbol{Q}[k]$, $\boldsymbol{\theta}[k]$ for $k = 1, ..., K$.

set $\boldsymbol{x}_{1:T}[0]$, $\boldsymbol{A}[0]$, $\boldsymbol{Q}[0]$, $\boldsymbol{\theta}[0]$ to arbitrary but valid (e.g. $\boldsymbol{Q}[0]$ must be positive definite) values;
for *k = 0, ..., K-1* **do**
  sample $\boldsymbol{x}_{1:T}[k+1] | \boldsymbol{A}[k], \boldsymbol{Q}[k], \boldsymbol{\theta}[k]$ using PGAS [Lindsten et al., 2014, p. 2160] with observations $\boldsymbol{y}_{1:T}$, reference trajectory $\boldsymbol{x}_{1:T}[k]$ and initial distribution: InitialDistribution;
  /* $\boldsymbol{A}[k], \boldsymbol{Q}[k]$ with (3.29) gives us the transition distribution for PGAS and (3.26c) (which is assumed to be completely known) gives us the observation distribution. */
  if $k \equiv 0 \pmod{OptimisationDistance}$ **then**
    find the maximum $(\boldsymbol{A}^*, \boldsymbol{Q}^*, \boldsymbol{\theta}^*)$ of (3.32) given $\boldsymbol{x}_{1:T}[k+1]$ using L-BFGS-B with the maximum iterations not too large (50-100 is fine);
    /* The low maximum iterations ensure that we do not spend too much time in this step: for small k we will not have good samples of $\boldsymbol{x}_{1:T}$ so there is no need to spend a lot of time in optimisation and for larger k we should not deviate too far from the maximum between each optimisation period. */
    Set initial walker positions equal to $(\boldsymbol{A}^*, \boldsymbol{Q}^*, \boldsymbol{\theta}^*)$ + vector of random noise;
    /* The random noise should be small ($|\text{noise}_i| < 0.05$ for all i) and independently chosen for each parameter of each walker (e.g. for each walker, each component of $\boldsymbol{A}$ should have some independent noise). You must be careful that the noise does not lead to the initial walker positions being invalid; for example, degrees of freedom ν must be positive and $\boldsymbol{Q}$ must be positive definite. */
  end
  else
    Set initial walker positions equal to $(\boldsymbol{A}[k], \boldsymbol{Q}[k], \boldsymbol{\theta}[k])$ + vector of random noise;
    /* above comments still apply */
  end
  sample $\boldsymbol{A}[k+1], \boldsymbol{Q}[k+1], \boldsymbol{\theta}[k+1] | \boldsymbol{x}_{1:T}[k+1]$ using EMCEE and (3.32) with #walkers = 2, burn-in = 10;
end
return $\boldsymbol{x}_{1:T}[1:K]$, $\boldsymbol{A}[1:K]$, $\boldsymbol{Q}[1:K]$, $\boldsymbol{\theta}[1:K]$;

3.2.5 STPSSMs: A State-Space Model with a Heavy Tailed Process

Utilising the methods of the previous section allow us to present a new state-space model with a Student-t process rather than a Gaussian process for the prior on the latent state function f. Student-t processes (in the context of regression) were first introduced by Shah et al. [2014] and have several advantages over Gaussian processes; in particular, they are more flexible and are more robust against outliers because of the heavy-tailed nature of multivariate-t distribution [Shah et al., 2014]. Furthermore, Student-t processes also possess many of the important properties of Gaussian processes such as analytic marginal and predictive distributions [Shah et al., 2014]; however, to keep the analytic nature we have to incorporate the noise model into the kernel rather than having it separate (see Shah et al. [2014] for more details). If we do incorporate the noise model into the kernel then Student-t processes have the same computational cost as Gaussian processes [Shah et al., 2014].

Definition 3.3. Shah et al. [2014]
f is a Student-t process with mean function $m : \mathbb{R}^d \to \mathbb{R}$, kernel function $k : \mathbb{R}^{d\times d} \to \mathbb{R}$ and $\nu \in \mathbb{R}_{>0}$ degrees of freedom, if any finite collection of function values satisfies:

$$(f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_n))^T \sim \mathcal{T}(m(\boldsymbol{X}), K(\boldsymbol{X}, \boldsymbol{X}), \nu) \tag{3.33}$$

where $\boldsymbol{X} = (\boldsymbol{x}_1, ..., \boldsymbol{x}_n)^T$, $m(\boldsymbol{X}) = (m(\boldsymbol{x}_1), ..., m(\boldsymbol{x}_n))^T$ and $K(\boldsymbol{X}, \boldsymbol{X})_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. This is written as: $f \sim \mathcal{TP}(m, k, \nu)$.

On the whole, Student-t processes are very similar to Gaussian processes; for example, training still scales in $\mathcal{O}(N^3)$ as the amount of data (N) increases, and many of the approximation techniques used for Gaussian processes can be applied to Student-t processes. Also, for Student-t processes with multiple outputs, we can either train separate Student-t processes for each output dimension or use similar ideas (e.g. linear model of coregionalisation) as for Gaussian processes with multiple outputs. Furthermore, a key property of the Student-t process is that when the degrees of freedom ν tend to infinity, the Student-t process tends towards a Gaussian process (proof in Shah et al. [2014]). One important difference is that previously we have used *kernel function* and *covariance function* interchangeably but in the case of the Student-t process technically $k(\boldsymbol{x}, \boldsymbol{x}')$ is no longer the covariance. However, it is approximately the covariance and this approximation gets better as the degrees of freedom increases.

We will now look at reduced-rank Student-t processes. In the same way as reduced-rank Gaussian processes, the idea behind the reduced rank approximation is to find a decomposition of the kernel function $\boldsymbol{K}$ into $\boldsymbol{K} \approx \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^T$ where we are assuming that the kernel function is stationary. The derivation of the basis function approximation for Student-t processes is exactly the same as for Gaussian processes since it relies on finding an approximation to the kernel function rather than any specific properties of Gaussian processes [Solin and Särkkä, 2014]. Hence, we can use the same decomposition of the kernel (2.46) and the same basis functions (3.70) combined with the matrix inversion lemma and the original posterior/predictive equations for f_* in Shah et al. [2014] to produce a reduced-rank Student-t process which

trains in $\mathcal{O}(m^3)$ where m is the number of basis functions.

We will now define an important distribution required for reduced-rank Student-t processes: the matrixvariate-t distribution. As this distribution is not very common there are actually several competing probability density functions all of which have the multivariate-t distribution as a special case. Furthermore, not all the definitions of the matrixvariate-t distribution tend to the obvious matrix-normal distribution as the degrees of freedom tend to infinity. The definition we present below is slightly different to that in Gupta and Nagar [2000]; Zhu et al. [2008]; Iranmanesh et al. [2010] but tends to the obvious matrix-normal distribution as the degrees of freedom tends to infinity which is very important for Student-t processes.

Definition 3.4. Let $\boldsymbol{X} \in \mathbb{R}^{m\times n}$ be a matrixvariate-t distributed random variable with mean $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, symmetric positive definite row-scale parameter $\boldsymbol{\Sigma} \in \mathbb{R}^{m\times m}$, symmetric positive definite column-scale parameter $\boldsymbol{\Omega} \in \mathbb{R}^{n\times n}$ and $\nu \in \mathbb{R}_{>0}$ degrees of freedom. The probability density of $\boldsymbol{X}$ at the point $\boldsymbol{x} \in \mathbb{R}^{m\times n}$ is :

$$\mathcal{MT}(\boldsymbol{x}|\boldsymbol{M},\boldsymbol{\Sigma},\boldsymbol{\Omega},\nu) = Z\Big|\boldsymbol{I}_m + \frac{1}{\nu}\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{M})\boldsymbol{\Omega}^{-1}(\boldsymbol{x}-\boldsymbol{M})^T\Big|^{-\frac{\nu+m+n-1}{2}} \tag{3.34}$$

where

$$Z = \frac{\Gamma_n\big(\frac{\nu+m+n-1}{2}\big)}{\Gamma_n\big(\frac{\nu+n-1}{2}\big)}\frac{1}{(\nu\pi)^{\frac{mn}{2}}|\boldsymbol{\Sigma}|^{\frac{n}{2}}|\boldsymbol{\Omega}|^{\frac{m}{2}}} \tag{3.35}$$

and $\Gamma_n(\cdot)$ is the n-multivariate Gamma function, $\boldsymbol{I}_m$ is the $m \times m$ identity matrix and $|\cdot|$ is the determinant.

This definition has the property that:

$$\lim_{\nu\to\infty} \mathcal{MT}(\boldsymbol{x}|\boldsymbol{M},\boldsymbol{\Sigma},\boldsymbol{\Omega},\nu) = \mathcal{MN}(\boldsymbol{x}|\boldsymbol{M},\boldsymbol{\Sigma},\boldsymbol{\Omega}) \tag{3.36}$$

For completeness, the definitions in Gupta and Nagar [2000]; Iranmanesh et al. [2010] have that a definition of the matrixvariate-t distribution that has the convergence property:

$$\lim_{\nu\to\infty} \mathcal{MT}(\boldsymbol{x}|\boldsymbol{M},\boldsymbol{\Sigma},\nu\boldsymbol{\Omega},\nu) = \mathcal{MN}(\boldsymbol{x}|\boldsymbol{M},\boldsymbol{\Sigma},\boldsymbol{\Omega}) \tag{3.37}$$

so using our definition slightly simplifies matters.

Consider the regression problem:

$$\boldsymbol{y}_i = f(\boldsymbol{x}_i) + \boldsymbol{\epsilon}_i \tag{3.38}$$

$$f \sim \mathcal{TP}(0, k, \nu) \tag{3.39}$$

where we have data $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ for $i = 1, ..., N$, $\boldsymbol{y}_i \in \mathbb{R}^{d_y}$, $\boldsymbol{x}_i \in \mathbb{R}^{d_x}$ and $\boldsymbol{\epsilon}_i$ is some noise distribution with location $\mathbf{0}$ and scale $\boldsymbol{Q}$. Under the reduced rank approximation this can be written as [Solin and Särkkä, 2014]:

$$\boldsymbol{y}_i = \boldsymbol{A}\phi(\boldsymbol{x}_i) + \boldsymbol{\epsilon}_i \tag{3.40}$$

$$\boldsymbol{A} \sim \mathcal{MT}(0, \boldsymbol{Q}, \boldsymbol{V}, \nu) \tag{3.41}$$

where $\boldsymbol{A} \in \mathbb{R}^{d_y \times m}$, $\phi(\boldsymbol{x}_i) = (\phi_1(\boldsymbol{x}_i), ...\phi_m(\boldsymbol{x}_i))^T$ for basis functions ϕ_i, $\boldsymbol{Q}$ is the noise scale, $\boldsymbol{V} = \text{diag}(S^{-1}(\sqrt{\lambda_1}), ..., S^{-1}(\sqrt{\lambda_m}))$ where the λ_i are defined as in (2.47) and S is the spectral density of the kernel function (which is assumed to be stationary). However, unlike for Gaussian processes the posterior distribution of $\boldsymbol{A}$ does not exist in closed form unless we incorporate the noise into the kernel function. If we wish to use PGAS to sample the latent states, then we cannot incorporate the noise into the kernel function. As a result, we are forced to use Monte Carlo methods to sample from the posterior of $\boldsymbol{A}$ but fortunately this can be done by using the same method as in GPSSMs with Student-t latent noise: EMCEE.

We can now write down the equations of a Student-t process state-space model:

$$\boldsymbol{x}_t = \boldsymbol{A}\phi(\boldsymbol{x}_{t-1}) + \boldsymbol{w}_t \tag{3.42a}$$
$$\boldsymbol{y}_t \sim p(\boldsymbol{y}_t|\boldsymbol{x}_t) \tag{3.42b}$$
$$\boldsymbol{A}|\boldsymbol{Q} \sim \mathcal{MT}(\boldsymbol{A}|\boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{V}, \nu_A) \tag{3.42c}$$
$$\boldsymbol{Q} \sim \mathcal{IW}(l_Q, \boldsymbol{\Lambda}_Q) \tag{3.42d}$$

where $\boldsymbol{w}_t$ can be any noise distribution with mean $\boldsymbol{0}$ and scale $\boldsymbol{Q}$. For example, Gaussian noise:

$$\boldsymbol{w}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}) \tag{3.43}$$

or Student-t noise:

$$\boldsymbol{w}_t \sim \mathcal{T}(\boldsymbol{0}, \boldsymbol{Q}, \nu_w) \tag{3.44}$$

or even Laplace noise:

$$\boldsymbol{w}_t \sim \mathcal{L}(\boldsymbol{0}, \boldsymbol{Q}) \tag{3.45}$$

where $\mathcal{L}$ is the multivariate Laplace distribution. Learning in all these models is very similar to learning in the GPSSM with Student-t noise: we use a Gibbs sampler with PGAS to sample the latent states and EMCEE to sample the parameter space. For EMCEE with latent states $\boldsymbol{x}_{1:t}$ we would use:

$$\log \hat{p} = \sum_{k=2}^{t} \log p(\boldsymbol{x}_k|\boldsymbol{A}\phi(\boldsymbol{x}_{k-1}), \boldsymbol{Q}, \boldsymbol{\theta}) + \log \mathcal{MT}(\boldsymbol{A}|\boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{V}, \nu_A) + \log \mathcal{IW}(\boldsymbol{Q}|l_Q, \boldsymbol{\Lambda}_Q) + \log p(\boldsymbol{\theta}) \tag{3.46}$$

where $p(\boldsymbol{x}_k|\boldsymbol{A}\phi(\boldsymbol{x}_{k-1}), \boldsymbol{Q}, \boldsymbol{\theta})$ is the transition pdf and it will depend on the chosen noise distribution.

We can now present an algorithm for online learning of Student-t process state-space models (STPSSMs). The algorithm uses the same idea as online learning for GPSSMs but is combined with the learning structure of GPSSM with latent Student-t noise. This can be easily extended for multiple data pieces per call to getData(), multiple PGAS/EMCEE samples per call to getData() or forgetful online learning by following the same ideas as in GPSSMs. Furthermore, this method can be extended to other function processes such as Dirichlet processes, Laplace processes and inverse Wishart processes by changing the prior distribution on $\boldsymbol{A}$ to a matrixvariate-Dirichlet, matrixvariate-Laplace or an inverse Wishart distribution respectively.

Algorithm 9: Online learning of a STPSSM

Input : Initial observed data $\boldsymbol{y}_{1:t_0}$, InitialDistribution $p(\boldsymbol{x}_1)$, *OptimisationDistance*.

Output: $\boldsymbol{x}^k_{1:t_k}$, $\boldsymbol{A}[k]$, $\boldsymbol{Q}[k]$, $\boldsymbol{\theta}[k]$ for $k = 1, ..., K$ and *weights*.

set $\boldsymbol{x}^0_{1:t_0}$, $\boldsymbol{A}[0]$, $\boldsymbol{Q}[0]$, $\boldsymbol{\theta}[0]$ to arbitrary but valid (e.g. $\boldsymbol{Q}[0]$ must be positive definite) values;

set *weights* = $[t_0]$;

for *k = 0, ..., K* **do**

  newData = getData();

  $t_{k+1} = t_k + 1$;

  append the new data to the old data: $\boldsymbol{y}_{1:t_{k+1}} = (\boldsymbol{y}_{1:t_k}$, new data);

  sample $\hat{\boldsymbol{x}}$ from $\mathcal{N}(\bar{\boldsymbol{x}}_{t_k}, \boldsymbol{I}_{d_{Latent}})$ where $\bar{\boldsymbol{x}}_{t_k}$ is the state at time t_k (mean over all samples weighted using *weights*) and d_{Latent} is the dimension of the latent space;

  append t_{k+1} to *weights*;

  set $\hat{\boldsymbol{x}}_{1:t_{k+1}} = (\boldsymbol{x}^k_{1:t_k}, \hat{\boldsymbol{x}})$;

  sample $\boldsymbol{x}^{k+1}_{1:t_{k+1}} | \boldsymbol{A}[k], \boldsymbol{Q}[k], \boldsymbol{\theta}[k]$ using PGAS [Lindsten et al., 2014, p. 2160] with observations $\boldsymbol{y}_{1:t_{k+1}}$, reference trajectory $\hat{\boldsymbol{x}}_{1:t_{k+1}}$ and initial distribution: InitialDistribution;

  `/* ` $\boldsymbol{A}[k], \boldsymbol{Q}[k]$ `with the noise distribution gives us the transition distribution for PGAS and (3.42c) (which is assumed to be completely known) gives us the observation distribution. */`

  if $k \equiv 0 \pmod{OptimisationDistance}$ **then**

    find the maximum $(\boldsymbol{A}^*, \boldsymbol{Q}^*, \boldsymbol{\theta}^*)$ of (3.32) given $\boldsymbol{x}^{k+1}_{1:t_{k+1}}$ using L-BFGS-B with the maximum iterations not too large (50-100 is fine);

    Set initial walker positions equal to $(\boldsymbol{A}^*, \boldsymbol{Q}^*, \boldsymbol{\theta}^*)$ + vector of random noise;

    `/* see comments from learning GPSSMs with Student-t noise */`

  end

  else

    Set initial walker positions equal to $(\boldsymbol{A}[k], \boldsymbol{Q}[k], \boldsymbol{\theta}[k])$ + vector of random noise;

    `/* see comments from learning GPSSMs with Student-t noise */`

  end

  sample $\boldsymbol{A}[k+1], \boldsymbol{Q}[k+1], \boldsymbol{\theta}[k+1] | \boldsymbol{x}^{k+1}_{1:t_{k+1}}$ using EMCEE and (3.32) with #walkers = 2, burn-in = 10;

end

return $\boldsymbol{x}^1_{t_1^{start}:t_1^{end}}, ..., \boldsymbol{x}^K_{t_K^{start}:t_K^{end}}$, $\boldsymbol{A}[1:K]$, $\boldsymbol{Q}[1:K]$, $\boldsymbol{\theta}[1:K]$, weights;

`/* In a similar manner to online Learning of GPSSMs, if we wish to compute values such as the mean or covariance of` $f(\boldsymbol{x}) = \boldsymbol{A}\phi(\boldsymbol{x})$ `then we should use a weighted mean and weighted covariance with the samples weighted by` *weights*. `*/`

3.2.6 Examples

In this section, we look at some examples of the various new features we have added to the model of Svensson et al. [2016]. We only consider models with fixed observation distributions in order to allow for an identifiable latent function and we only look at models with single-dimensional observation and latent spaces. Furthermore, we test the performance of the models by computing the root mean square error of the predicted latent states compared with the true latent states and by computing the mean log likelihood of the latent states given the learnt model. These statistics are defined below.

Given a trained model and a test set of latent states $(x_1^*, ..., x_T^*)$, we can construct key statistics for the posterior predictive distribution of $f(x_i^*)$ using the samples from the parameter posterior. This has already been looked at for online/forgetful online learning see: (3.18) and (3.20). Now, for learning with Student-t noise or Student-t processes, suppose that we have samples $\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_K$ from the parameter posterior (here we assume that $\boldsymbol{\theta}_i$ is a vector containing an $\boldsymbol{A}$ sample, a $\boldsymbol{Q}$ sample and a sample of each of the hyperparameters) and we write $f_{\boldsymbol{\theta}}(x)$ for the latent function $f(x)$ with model parameters equal to $\boldsymbol{\theta}$. Then we can define:

$$\mathbb{E}[f(x^*)] \approx \frac{1}{K} \sum_{i=1}^{K} f_{\boldsymbol{\theta}_i}(x^*) \tag{3.47}$$

$$\mathbb{V}[f(x^*)] \approx \left(\frac{1}{K} \sum_{i=1}^{K} (f_{\boldsymbol{\theta}_i}(x^*))^2 \right) - \mathbb{E}[f(x^*)]^2 \tag{3.48}$$

where these tend to the true results as K tends to infinity.

Now we can define:

1. The root mean square error (RMSE) (smaller is better):

$$RMSE = \sqrt{\frac{1}{T-1} \sum_{t=1}^{T-1} (x_{t+1}^* - \mathbb{E}[f(x_t^*)])^2} \tag{3.49}$$

2. For models which assume a Gaussian noise distribution for the latent states, the mean log likelihood (LL) is defined as (larger is better):

$$LL_{gauss} = \frac{1}{T-1} \sum_{t=1}^{T-1} \log \mathcal{N}(x_{t+1}^* | \mathbb{E}[f(x_t^*)], \mathbb{V}[f(x_t^*)] + \mathbb{E}[\boldsymbol{Q}]) \tag{3.50}$$

   where $\mathbb{E}[\boldsymbol{Q}] \approx \frac{1}{K} \sum_{i=1}^{K} \boldsymbol{Q}_{\boldsymbol{\theta}_i}$ and $\boldsymbol{Q}_{\boldsymbol{\theta}_i}$ is the value of $\boldsymbol{Q}$ when the model parameters are equal to $\boldsymbol{\theta}_i$.

3. For models which assume a Student-t noise distribution for the latent states, calculating the mean log likelihood (LL) is more complex. The idea is still the

same: we wish to find the probability of the test data under the model so in this case, we define:

$$LL_{student-t} = \frac{1}{(T-1)K} \sum_{t=1}^{T-1} \sum_{i=1}^{K} \log \mathcal{T}(x_{t+1}^* | f_{\boldsymbol{\theta}_i}(x_t^*), \boldsymbol{Q}_{\boldsymbol{\theta}_i}, \nu_{\boldsymbol{\theta}_i}) \tag{3.51}$$

where $\mathcal{T}(x|\mu, \Sigma, \nu)$ is a multivariate-t density function (parametrised as in (3.25)) with mean μ, scale Σ and ν degrees of freedom. The notation $\boldsymbol{Q}_{\boldsymbol{\theta}_i}$ and $\nu_{\boldsymbol{\theta}_i}$ simply means the values of $\boldsymbol{Q}_{\boldsymbol{\theta}_i}$ and $\nu_{\boldsymbol{\theta}_i}$, respectively, when the model parameters are equal to $\boldsymbol{\theta}_i$.

Example 1:
Aim: This is a 'sanity-check' example which allows us to compare the various models under a simple system. The idea is that we expect these models to produce similar results (RMSE and LL) not only to each other but also to the results stated in Svensson et al. [2016]. This should demonstrate that the different ways we compare particular performance statistics lead to the same results when the models are equivalent. Note that we do expect the training times to be different.

Training Data: A sequence of 500 observations generated by the system:

$$x_{t+1} = f(x_t) + w_t \tag{3.52}$$
$$y_t = x_t + e_t \tag{3.53}$$
$$w_t \sim \mathcal{N}(0, 1) \tag{3.54}$$
$$e_t \sim \mathcal{N}(0, 1) \tag{3.55}$$

where

$$f(x) = \begin{cases} x + 1 & x < 4 \\ -4x + 21 & x \geq 4 \end{cases} \tag{3.56}$$

The models only receive the observed states $y_1, ..., y_{500}$. The latent states $x_1, ..., x_{500}$ are discarded.

Test Data: A sequence of 10000 observations and latent states generated by the above system.

Additional Information: The initial point will be zero (known), the observation distribution will be known beforehand (i.e. not learnt), all the reduced-rank GPSSMs will use 12 basis functions, 20 PGAS particles, domain length $L = 12$, the $\boldsymbol{Q}$ prior hyperparameters will both be equal to 1 and we will use a burn-in of 50 samples. The offline models will have $K = 180$ sampling rounds and the online models will have $K = 200$ sampling rounds; furthermore, all the online/forgetful online models will start with two data points and receive an additional 5 data points every other sampling round (i.e. number of update samples = 2). Finally, all forgetful models will have a maximum memory length of 250 data points (see section 3.2.2).

Results:

Method	Kernel	Assumed Latent Noise	Process	RMSE	LL	Training Time (seconds)
Offline	RBF	Gaussian	Gaussian	1.13	**-1.50**	83.36
Online	RBF	Gaussian	Gaussian	1.12	-1.52	58.06
Forgetful	RBF	Gaussian	Gaussian	**1.11**	**-1.50**	**45.66**
Offline	Adaptive	Student-t	Gaussian	1.13	-1.53	126.01
Online	RBF	Student-t	Student-t	1.12	-1.59	77.50

Table 3.4: Results for Example 1

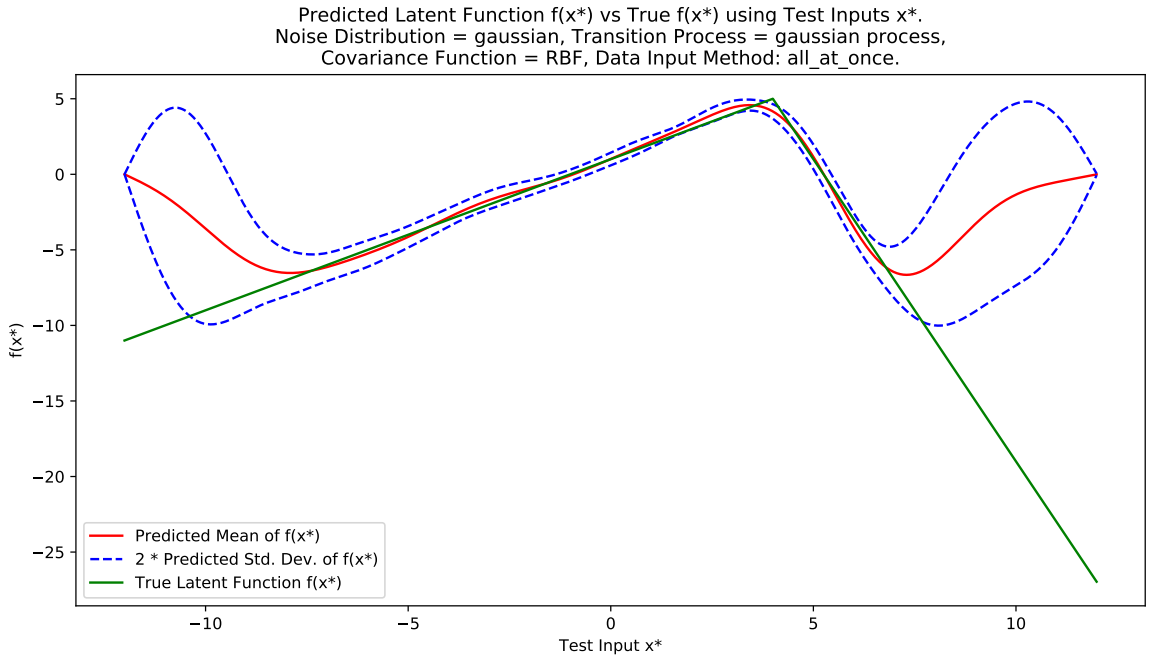

Figure 3.8: A GPSSM learnt using the algorithm of Svensson et al. [2016] (Table 3.4, Row 1)

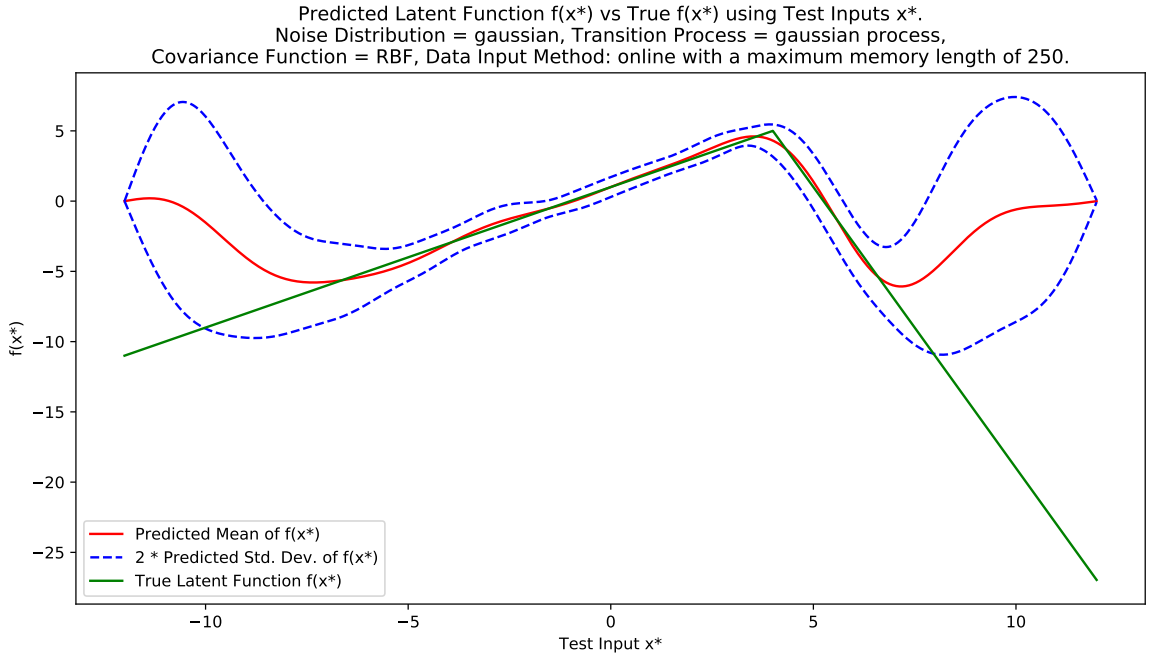

Figure 3.9: A GPSSM learnt using our forgetful online learning algorithm (Table 3.4, Row 3)

Example 2:
Aim: To compare the different models under a system with Student-t noise in the latent states.

Training Data: A sequence of 1000 observations generated by the system:

$$x_{t+1} = 0.5x_t + 3\sin(x_t) + w_t \tag{3.57}$$
$$y_t = x_t + e_t \tag{3.58}$$
$$w_t \sim \mathcal{T}(0, 1, \nu = 1) \tag{3.59}$$
$$e_t \sim \mathcal{N}(0, 1) \tag{3.60}$$

The models only receive the observed states $y_1, ..., y_{1000}$. The latent states $x_1, ..., x_{1000}$ are discarded.

Test Data: A sequence of 10000 observations and latent states generated by the above system.

Additional Information: The initial point will be zero (known), the observation distribution will be known beforehand (i.e. not learnt), all the reduced-rank GPSSMs will use 8 basis functions, 20 PGAS particles, domain length $L = 10$, the $\boldsymbol{Q}$ prior hyperparameters will both be equal to 1 and we will use a burn-in of 100 samples. The offline models will have $K = 180$ sampling rounds and the online models will have $K = 200$ sampling rounds; furthermore, all the online/forgetful online models will start with two data points and receive an additional 10 data points every other sampling round (i.e. number of update samples $= 2$). Finally, all forgetful models

will have a maximum memory length of 250 data points (see section 3.2.2).

Results:
Since the true noise distribution is Student-t, we do not provide a RMSE because it is unhelpful under Student-t noise due to the presence of large outliers.

Method	Kernel	Assumed Latent Noise	Process	LL	Training Time (seconds)
Offline	RBF	Gaussian	Gaussian	-573.47	138.72
Online	RBF	Student-t	Gaussian	-17.30	121.85
Forgetful	RBF	Student-t	Gaussian	-3.23	**59.83**
Offline	RBF	Student-t	Gaussian	**-3.06**	175.64

Table 3.5: Results for Example 2.

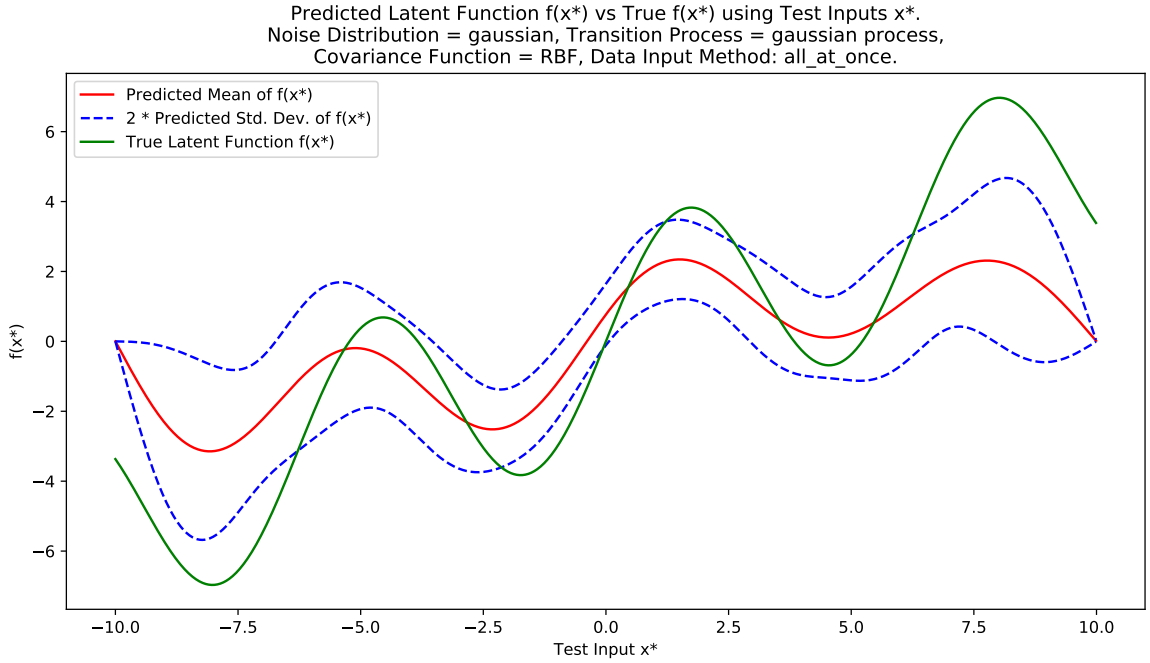

Figure 3.10: A GPSSM with Student-t noise in the latent states learnt using the algorithm of Svensson et al. [2016] (Table 3.5, Row 1)

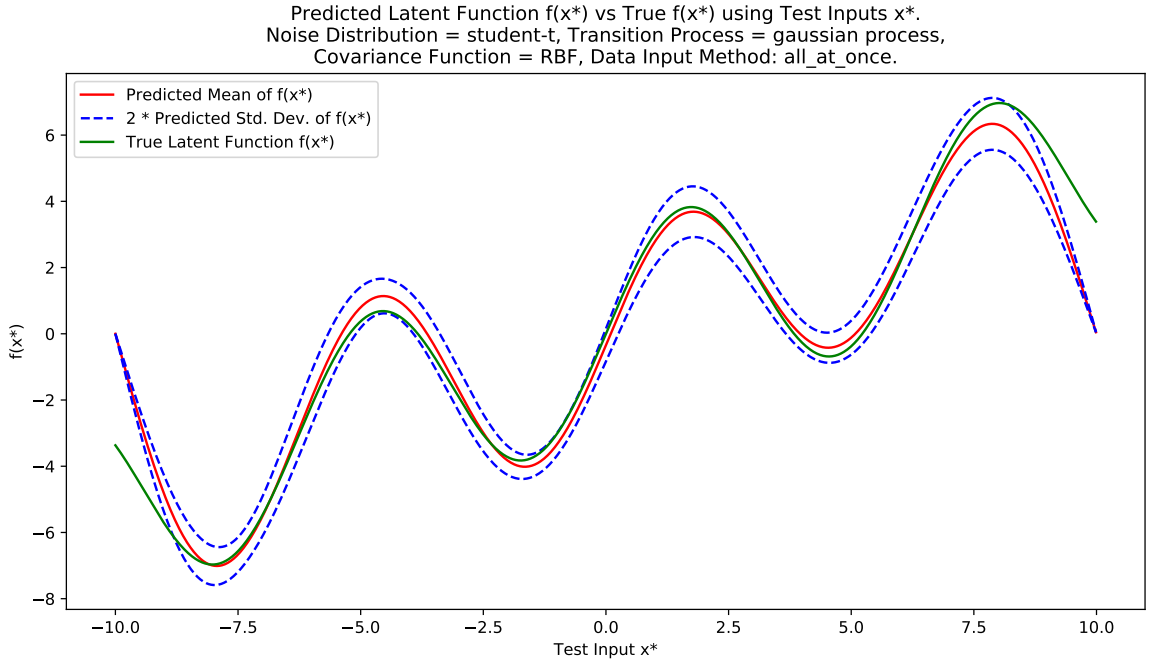

Figure 3.11: A GPSSM with Student-t noise in the latent states learnt using algorithm 8 (Table 3.5, Row 4)

Example 3:
Aim: To compare some combinations of features we have not looked at previously.

Training Data: A sequence of 1000 observations generated by the system:

$$x_{t+1} = 3\tanh(0.5x_t) + w_t \tag{3.61}$$
$$y_t = x_t + e_t \tag{3.62}$$
$$w_t \sim \mathcal{N}(0, 1.2) \tag{3.63}$$
$$e_t \sim \mathcal{N}(0, 0.8) \tag{3.64}$$

The models only receive the observed states $y_1, ..., y_{1000}$. The latent states $x_1, ..., x_{1000}$ are discarded.

Test Data: A sequence of 10000 observations and latent states generated by the above system.

Additional Information: The initial point will be zero (known), the observation distribution will be known beforehand (i.e. not learnt), all the reduced-rank GPSSMs will use 8 basis functions, 20 PGAS particles, domain length $L = 8$, the $\boldsymbol{Q}$ prior hyperparameters will both be equal to 1 and we will use a burn-in of 50 samples. The offline models will have $K = 180$ sampling rounds and the online models will have $K = 200$ sampling rounds; furthermore, all the online/forgetful online models will start with two data points and receive an additional 10 data points every other sampling round (i.e. number of update samples = 2). Finally, all forgetful models will have a maximum memory length of 250 data points (see section 3.2.2).

Results:

Method	Kernel	Assumed Latent Noise	Process	RMSE	LL	Training Time (seconds)
Offline	Adaptive	Gaussian	Gaussian	1.16	-1.59	143.38
Online	Matern52	Gaussian	Gaussian	1.25	-1.70	92.62
Forgetful	Matern32	Gaussian	Gaussian	1.16	**-1.57**	**43.80**
Offline	RBF	Student-t	Gaussian	**1.11**	-1.59	177.96
Offline	RBF	Student-t	Student-t	1.12	-1.59	178.55

Table 3.6: Results for Example 3.

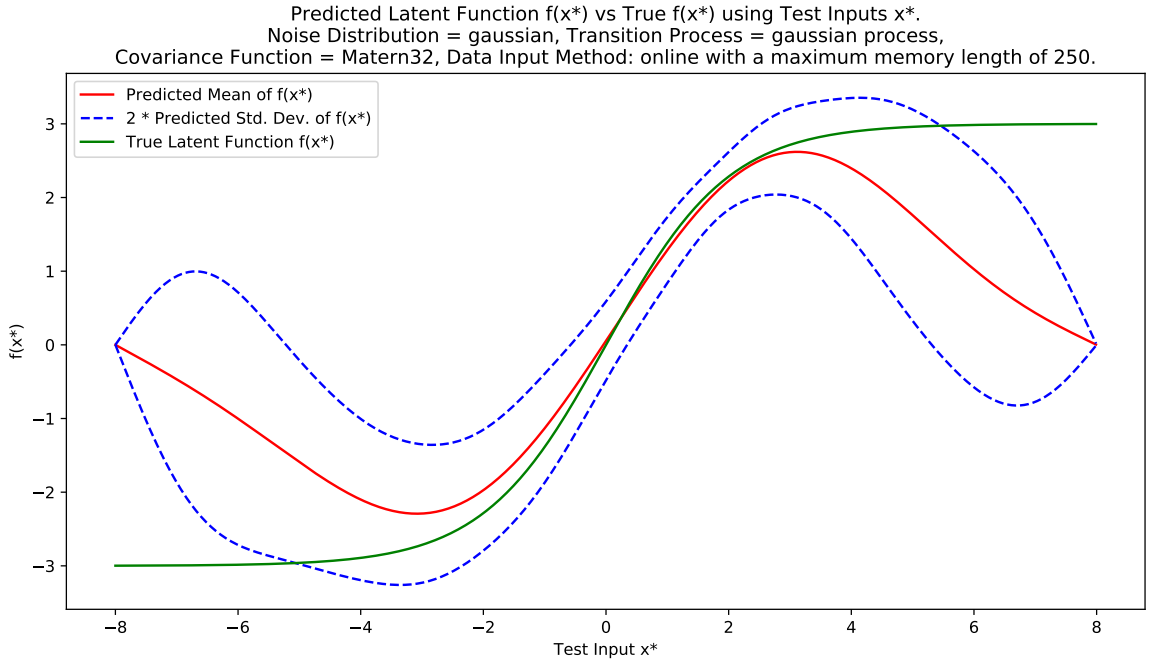

Figure 3.12: Forgetful online learning with a Matern32 covariance function (Table 3.6, Row 3)

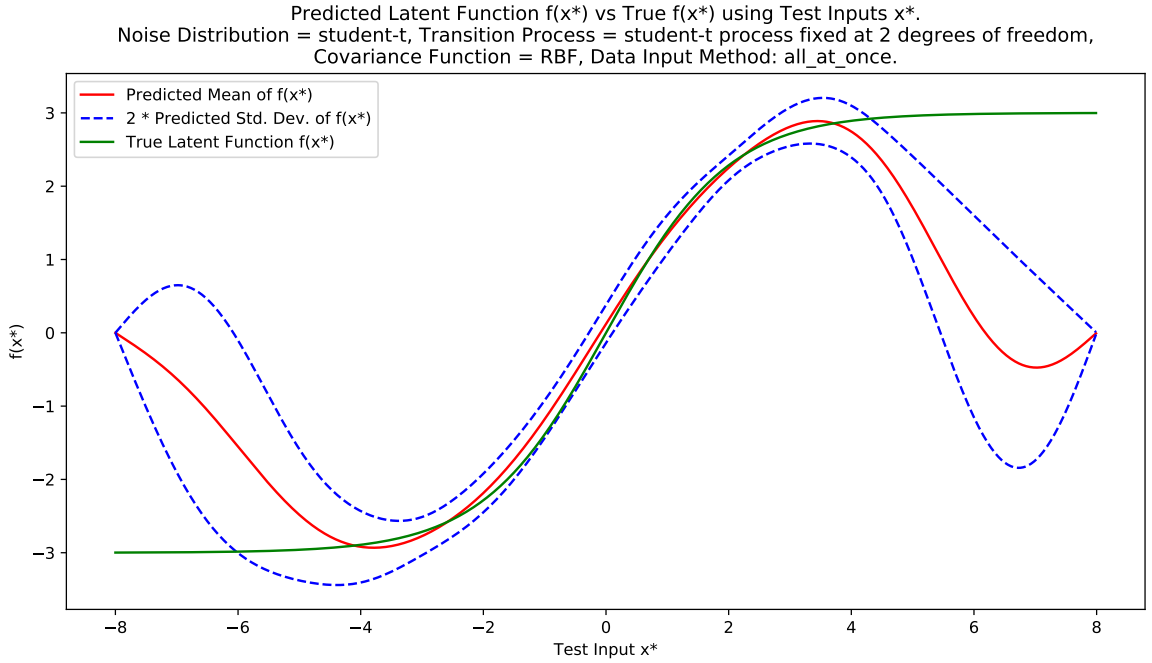

Figure 3.13: An example of learning with a Student-t process (Table 3.6, Row 5)

Discussion:
From example 1, we can see that all the different models will produce similar results in a simple system: this is as expected. Furthermore, in example 2, we see that our novel algorithm for learning in the presence of latent Student-t noise outperforms the algorithm of Svensson et al. [2016]. This is not surprising because Svensson et al. [2016] assume the latent noise is Gaussian. Overall, we see that as the amount of data increases, online learning and forgetful online learning tend to similar results as the offline learning case. Sometimes these methods perform better than offline learning and we believe this might be because the weighting system can lead to bad samples (which are often the early samples even after burn-in) having a small weight. Moreover, forgetful online learning, in particular, can 'forget' outliers which would usually have a detrimental effect on the sample quality.

3.3 Learning Hilbert reduced-rank Gaussian Processes in High Dimensions

Consider the Gaussian process regression problem:

$$\boldsymbol{y}_i = f(\boldsymbol{x}_i) + \boldsymbol{\epsilon_i} \tag{3.65}$$
$$f \sim \mathcal{GP}(\mathbf{0}, k) \tag{3.66}$$
$$\boldsymbol{\epsilon_i} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{Q}) \tag{3.67}$$

where $\boldsymbol{y}_i \in \mathbb{R}^{d_{out}}$, $\boldsymbol{x}_i \in \mathbb{R}^{d_{in}}$, $\boldsymbol{Q} \in \mathbb{R}^{d_{out} \times d_{out}}$ is the noise covariance and we have training data $\{(\boldsymbol{y}_i, \boldsymbol{x}_i)$ for $i = 1, ..., N\}$.

When we talk about dimensionality in the context of Gaussian processes, we are either referring to the input dimension ($\mathbb{R}^{d_{in}}$) or the output dimension ($\mathbb{R}^{d_{out}}$). Usually, high input dimensions are not mentioned that much because the training time of a full Gaussian process largely depends on the amount of training data (N) and not the dimension of the input data ($\mathbb{R}^{d_{in}}$). However, it could be said that as the input dimension increases, more training data is required to produce a good model. When we use the Hilbert reduced-rank approximation of the Gaussian process, the picture is different. As we have noted previously: Hilbert reduced-rank Gaussian processes do not scale well when the input dimension increases. For completeness, neither the full Gaussian process nor the Hilbert reduced-rank Gaussian process have much issue with high output dimensions ($\mathbb{R}^{d_{out}}$) because we can just train independent GPs for each output dimension and although this *is an approximation* compared with a full multi-output GP, it is generally a very good approximation. We will now discuss in depth why the Hilbert reduced-rank Gaussian process model of Solin and Särkkä [2014] does not scale well as the input dimension increases and the approach we take to deal with this.

Consider the above regression problem but take $d_{out} = 1$ and $d_{in} = d$. In this case, the Hilbert reduced-rank model of Solin and Särkkä [2014] with domain $\Omega = [-L_1, L_1] \times ... \times [-L_d, L_d]$ plus Dirichlet boundary conditions (zero on the boundary) would approximate the kernel as:

$$k_{\boldsymbol{m}}(\boldsymbol{x}, \boldsymbol{x}') \approx \sum_{j_1,...,j_d=\mathbf{1}}^{\boldsymbol{m}} S(\sqrt{\lambda_{j_1,...,j_d}})\phi_{j_1,...,j_d}(\boldsymbol{x})\phi_{j_1,...,j_d}(\boldsymbol{x}') \tag{3.68}$$

where $\mathbf{1} \in \mathbb{R}^d$ is a vector of all ones and $\boldsymbol{m} = (m_1, ..., m_d) \in \mathbb{R}^d$ is a vector giving the chosen number of basis functions in each of the d input dimensions. As before we have eigenvalues:

$$\lambda_{j_1,...,j_d} = \sum_{k=1}^{d} \left(\frac{\pi j_k}{2L_k} \right)^2 \tag{3.69}$$

and the eigenfunctions/basis functions:

$$\phi_{j_1,...,j_d}(\boldsymbol{x}) = \prod_{k=1}^{d} \frac{1}{\sqrt{L_k}} \sin\left(\frac{\pi j_k (x_k + L_k)}{L_k} \right) \tag{3.70}$$

For clarification, when we refer to the *number of basis functions in dimension* i, we mean m_i; furthermore, when we refer to the *total number of basis functions*, we mean the total number of $\phi_{j_1,...,j_d}(\boldsymbol{x})$ in the sum (3.68) which is $\prod_{i=1}^{d} m_i$.

As the dimension of the input space d increases, the number of terms in the kernel approximation sum (3.68) is $\prod_{i=1}^{d} m_i$ and if the number of basis functions is the same (say m) in all dimensions then this simplifies to m^d. The chosen number of basis functions per dimension (the m_i) can be small but not too small: we do need at least 8 basis functions per dimension (i.e. $m_i \geq 8$ for $i = 1, ..., d$) for a good performance. To emphasise the scaling issue, if we have 5 input dimensions and 8 basis

functions for each of the input dimensions (i.e. $m_i = 8$ for $i = 1, ..., 5$), then the total number of basis functions is $8^5 = 32768$. This means not only do we have to sum up $8^5 = 32768$ terms every time we need to compute the kernel but we also have to invert a $\prod_{i=1}^{d} m_i \times \prod_{i=1}^{d} m_i$ (here equal 32768×32768) basis function matrix during training/prediction. Clearly this becomes unreasonable to use for input dimensions higher than 2 or 3.

We will now present a novel reduced-rank Gaussian process model that scales considerably better with high input dimensions than the model by Solin and Särkkä [2014]. This new model is based on a combination of neural networks and Gaussian processes inspired by the mGP of Calandra et al. [2016] and the deep Gaussian processes of Damianou and Lawrence [2013]. Previous methods involving deep Gaussian process, such as Damianou and Lawrence [2013], use GPs for the activation functions but the method here is different and, in fact, we fix the activation functions and refer to them as *kernel augmentation functions*.

Consider the regression problem in (3.65)–(3.67) with $d_{out} = 1$ and $d_{in} = d$. As discussed previously in (2.61), under the Hilbert reduced-rank Gaussian process model, this can be written as Svensson et al. [2016]:

$$y_i = \boldsymbol{A}\phi(\boldsymbol{x}_i) + \epsilon_i \tag{3.71}$$
$$\boldsymbol{A} \sim \mathcal{MN}(\boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{V}) \tag{3.72}$$
$$\epsilon_i \sim \mathcal{N}(0, \boldsymbol{Q}) \tag{3.73}$$

where $\boldsymbol{A} \in \mathbb{R}^{1\times d}$, $\boldsymbol{Q} \in \mathbb{R}^{1\times 1}$ is the noise covariance, $\phi(\boldsymbol{x}_i) = (\phi_1(\boldsymbol{x}_i), ...\phi_{\mathcal{M}(\boldsymbol{m})}(\boldsymbol{x}_i))^T$ for basis functions ϕ_i and $\boldsymbol{m} = (m_1, ..., m_d)$, $\boldsymbol{V} = \text{diag}(S^{-1}(\sqrt{\lambda_1}), ..., S^{-1}(\sqrt{\lambda_{\mathcal{M}(\boldsymbol{m})}}))$ where the λ_i are defined as in (3.69), S is the spectral density of the chosen kernel function (which is assumed to be stationary) and we have used the mapping $\mathcal{M}$ from (2.51). Of course, this could be simplified since $y_i \in \mathbb{R}$ but we keep it in this format just to keep continuity between the previous discussion and the model we are about to present. The Hilbert reduced-rank GP demonstrates nicely that GP regression is ultimately just Bayesian linear regression with infinitely many features: the Hilbert reduced-rank GP just uses finitely many of those features.

In the same way as the neural networks extend linear regression, our new model which we call the deep Hilbert Gaussian process model will extend the Hilbert reduced-rank Gaussian process model. Consider a basic multilayer perceptron (MLP) with activation function σ and $n \in \mathbb{Z}_{\geq 0}$ hidden layers. In the case of regression, for a given input $\boldsymbol{x}$ the output y is constructed as:

$$z_1 = \sigma(\boldsymbol{W}_0\boldsymbol{x} + \boldsymbol{b}_0) \tag{3.74}$$
$$z_{i+1} = \sigma(\boldsymbol{W}_i\boldsymbol{z}_i + \boldsymbol{b}_i) \text{ for } i = 1, ..., n-1 \tag{3.75}$$
$$y = \boldsymbol{W}_n\boldsymbol{z}_n + \boldsymbol{b}_n \tag{3.76}$$

Important things to note are that we have a linear final activation and that the activation function σ is applied to each element of the input vector (i.e. is vectorised).

Also, the backpropagation algorithm is used to find the weights $\boldsymbol{W}_i$ and biases $\boldsymbol{b}_i$ for $i = 0, ..., n$. As with all standard neural networks, the MLP is prone to overfitting; although, there are a variety of methods (e.g. l_1, l_2 regularisation and dropout) designed to reduce it. Furthermore, we only get point estimates of the predicted outputs but having an understanding of the confidence in our estimate can be very useful so this is a big drawback of the MLP. The deep Hilbert Gaussian process model (DHGPM) combines the MLP with reduced-rank GPs, and this reduces the risk of overfitting while providing both point estimates and confidence regions (which take into account all sources of uncertainty) for the model's predictions.

The DHGPM arises from breaking the neural net into more pieces:

$$v_i = \boldsymbol{W}_i \boldsymbol{z}_i + \boldsymbol{b}_i \tag{3.77}$$
$$z_{i+1} = \sigma(v_i) \tag{3.78}$$

hence we can write:

$$v_i = \boldsymbol{W}_i \sigma(v_{i-1}) + \boldsymbol{b}_i \tag{3.79}$$

Therefore, the MLP with activation ϕ, $n \in \mathbb{Z}_{\geq 0}$ hidden layers and given input $\boldsymbol{x}$ could have the output y constructed as:

$$\boldsymbol{v}_0 = \boldsymbol{A}_0 \boldsymbol{x} + \boldsymbol{b}_0 \tag{3.80}$$
$$\boldsymbol{v}_i = \boldsymbol{A}_i \phi(\boldsymbol{v}_{i-1}) + \boldsymbol{b}_i \text{ for } i = 1, ..., n-1 \tag{3.81}$$
$$y = \boldsymbol{A}_n \phi(\boldsymbol{v}_{n-1}) + \boldsymbol{b}_n \tag{3.82}$$

for weights $\boldsymbol{A}_i$ and biases $\boldsymbol{b}_i$. If we imagine that the ϕ in (3.81)–(3.82) is the same the ϕ in (3.70), then this is starting to look very much like layers of Hilbert reduced-rank Gaussian processes but without any prior distributions. Hence, to construct the deep Hilbert Gaussian process model (DHGPM) we add these priors.

3.3.1 The Deep Hilbert Gaussian Process Model (DHGPM)

Suppose we have training data $(y_k, \boldsymbol{x}_k)$ for $k = 1, ..., N$ where $y_k \in \mathbb{R}$ and $\boldsymbol{x}_k \in \mathbb{R}^d$, then for a particular input $\boldsymbol{x}_k \in \mathbb{R}^d$ and kernel augmentation function ϕ, the DHGPM with $n \in \mathbb{Z}_{\geq 0}$ hidden layers is constructed as follows:

$$\boldsymbol{v}_0^k = \boldsymbol{A}_0 \boldsymbol{x}_k + \boldsymbol{b}_0 \tag{3.83}$$

$$\boldsymbol{v}_j^k = \boldsymbol{A}_j \phi(\boldsymbol{v}_{j-1}^k) + \boldsymbol{b}_j \text{ for } j = 1, ..., n-1 \tag{3.84}$$

$$f(\boldsymbol{x}_k) = \boldsymbol{A}_n \phi(\boldsymbol{v}_{n-1}^k) \tag{3.85}$$

$$y_k = f(\boldsymbol{x}_k) + \boldsymbol{\epsilon}_k \tag{3.86}$$

$$\boldsymbol{\epsilon}_k \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_n) \tag{3.87}$$

$$\boldsymbol{A}_j | \boldsymbol{Q}_j \sim \mathcal{MN}(\boldsymbol{0}, \boldsymbol{Q}_j, \boldsymbol{V}_j) \text{ for } j = 0, ..., n \tag{3.88}$$

$$\boldsymbol{Q}_j \sim \mathcal{IW}(r_j + l, \boldsymbol{\Lambda}) \text{ for } j = 0, ..., n \tag{3.89}$$

$$\boldsymbol{V}_j = \text{diag}(S_j^{-1}(\sqrt{\lambda_1^j}), ..., S_j^{-1}(\sqrt{\lambda_{c_j}^j})) \text{ for } j = 0, ..., n \tag{3.90}$$

$$\lambda_s^j = \left(\frac{\pi s}{2L_j}\right)^2 \text{ for } j = 0, ..., n \tag{3.91}$$

As k goes from 1 to N, the above set-up gives us the input-output pairs $(y_k, \boldsymbol{x}_k)$ for $k = 1, ..., N$. The various variables and constants are discussed below.

Notes:

1. Inputs are assumed to be column vectors. This is important as it means the given matrix normal distribution is the correct one.

2. The kernel augmentation function ϕ is vectorised; for example, if we have a vector $\boldsymbol{v} = (v_1, ..., v_m)^T \in \mathbb{R}^m$ then $\phi(\boldsymbol{v}) = (\phi(v_1), ..., \phi(v_m))^T \in \mathbb{R}^m$.

3. Since $y_k \in \mathbb{R}$, (3.87) is just a univariate normal distribution with mean 0 and variance $\boldsymbol{Q}_n \in \mathbb{R}$.

4. $\mathcal{MN}(\boldsymbol{M}, \boldsymbol{Q}, \boldsymbol{V})$ is the matrixvariate normal distribution with mean $\boldsymbol{M}$, row covariance $\boldsymbol{Q}$ and column covariance $\boldsymbol{V}$.

5. $\mathcal{IW}(\nu, \boldsymbol{\Lambda})$ is the inverse Wishart distribution with degrees of freedom ν and scale $\boldsymbol{\Lambda}$. Although $\boldsymbol{Q}_n \in \mathbb{R}$ and so the inverse Wishart distribution simplifies to an inverse Gamma distribution, the remaining $\boldsymbol{Q}_i$ for $i = 0, ..., n-1$ will be matrices in general.

6. For simplicity, the same kernel should be used for all layers. This chosen kernel appears in its spectral density form S_j in each layer j. Although the form of the kernel should be the same for all layers; for example, all layers could have an RBF kernel, these kernels are not constrained to have the same hyperparameters. In other words, for each layer j, if the chosen kernel S_j has hyperparameters $\boldsymbol{\theta}_j$, then $\boldsymbol{\theta}_j$ is not constrained to equal $\boldsymbol{\theta}_i$ for *different* layers i and j (of course within a layer the $\boldsymbol{\theta}_j$ must be the same for all occurrences of S_j).

7. The outputs y_k for $k = 1, ..., N$ should be single-dimensional i.e. $y_k \in \mathbb{R}$. For multiple outputs, fit separate DHGPMs for each output dimension.

8. A layout $[d_0, ..., d_{n-1}]$ gives the dimensions of $\boldsymbol{v}_0, ..., \boldsymbol{v}_{n-1}$. So when we refer to DHGPM with layout $[d_0, ..., d_{n-1}]$ we mean a model with n hidden layers with input $\boldsymbol{x}_k \in \mathbb{R}^d$ mapped to $v_0 \in \mathbb{R}^{d_0}$, v_0 mapped to $v_1 \in \mathbb{R}^{d_1}$, ..., v_{n-2} mapped to $v_{n-1} \in \mathbb{R}^{d_{n-1}}$ and then v_{n-1} mapped to $y_k \in \mathbb{R}$. This gives us the dimensions of the $\boldsymbol{A}_j$ and $\boldsymbol{b}_j$ for $j = 0, ..., n$; for example, suppose with have a DHGPM with layout $[3, 12]$ with input dimension 4, then this is a DHGPM with 2 hidden layers and $\boldsymbol{A}_0 \in \mathbb{R}^{3\times 4}$, $\boldsymbol{A}_1 \in \mathbb{R}^{12\times 3}$, $\boldsymbol{A}_2 \in \mathbb{R}^{1\times 12}$, $\boldsymbol{b}_0 \in \mathbb{R}^3$ and $\boldsymbol{b}_1 \in \mathbb{R}^{12}$ (assuming the input is a column vector).

9. $\boldsymbol{Q}_j \in \mathbb{R}^{r_j \times r_j}$ where r_j is the number of rows in $\boldsymbol{A}_j$. In the inverse Wishart prior (3.89) the degrees of freedom depends on this parameter r_j, which equals the number of columns (or rows) of $\boldsymbol{Q}_j$.

10. For simplicity, either we take the $\boldsymbol{Q}_j$ to be diagonal matrices or we take $\boldsymbol{Q}_j = \sigma_j \boldsymbol{I}$ where $\sigma_j \in \mathbb{R}_{>0}$ and $\boldsymbol{I}$ is the identity matrix or we take the $\boldsymbol{Q}_i$ to be of the form $\boldsymbol{Q}_j = \boldsymbol{p}_j \boldsymbol{p}_j^T + \sigma_j \boldsymbol{I}$ where $\sigma_j \in \mathbb{R}_{>0}$ and $\boldsymbol{p} \in \mathbb{R}^{r_j}$.

11. $\boldsymbol{V}_j \in \mathbb{R}^{c_j \times c_j}$ where c_j is the number of columns in $\boldsymbol{A}_j$. In the definition of $\boldsymbol{V}_j$ (3.90) there is a parameter c_j and this c_j equals the number of columns (or rows) of $\boldsymbol{V}_j$.

12. The prior parameters l and $\boldsymbol{\Lambda}$ for the Inverse Wishart prior (3.89) should be the same for all layers and fixed before training. The degrees of freedom $(r_j + l)$ might not be the same for all layers (depending on the r_j) but the l should be the same.

13. The last layer has no bias term: this helps prevent overfitting.

14. The biases have no explicit priors; however, they are somewhat constrained by the priors placed on the $\boldsymbol{A}_j$.

15. There is no explicit noise terms in each of the hidden or inputs layers (unlike the output layer) but the $\boldsymbol{Q}_j$ inside the matrix-normal distribution do take potential noise into account which is why we need to use the inverse Wishart prior: it stops the model assuming everything is noise.

16. Unlike in the Hilbert reduced-rank Gaussian process model, the L_j in λ_s^j (3.91) are not fixed: they are variables for optimisation and sampling.

Training:

Let $\boldsymbol{\theta}$ be a vector[1] of all the unknown parameters in the model, this includes all the weights $\boldsymbol{A}_j$, the biases $\boldsymbol{b}_j$, the $\boldsymbol{Q}_j$, the L_j from (3.91) and the unknown parameters $\boldsymbol{\theta}_j$ of the spectral densities S_j. It does not include various fixed parameters like

[1]When we refer to $\boldsymbol{\theta}$, we assume that it is an element of $\mathbb{R}^D$ for some $D > 0$ i.e. all the matrices have been flattened.

the l and $\boldsymbol{\Lambda}$ in (3.89). Given training data $(\boldsymbol{y}, \boldsymbol{X}) = \{(y_k, \boldsymbol{x}_k) \text{ for } k = 1, ..., N\}$, the parameters $\boldsymbol{\theta}$ are learnt by maximising the posterior $p(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{X})$. To do this, we note that

$$p(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{X}) \propto p(\boldsymbol{y}|\boldsymbol{\theta}, \boldsymbol{X})p(\boldsymbol{\theta}|\boldsymbol{X}) \tag{3.92}$$

$$\propto \left(\prod_{i=1}^{N} \mathcal{N}(y_i|f(\boldsymbol{x}_i), \boldsymbol{Q}_n)\right)\left(\prod_{j=0}^{n} \mathcal{MN}(\boldsymbol{A}_j|\boldsymbol{0}, \boldsymbol{Q}_j, \boldsymbol{V}_j)\mathcal{IW}(\boldsymbol{Q}_j|r_j + l, \boldsymbol{\Lambda})\right)p(\hat{\boldsymbol{\theta}}) \tag{3.93}$$

where n is the number of hidden layers and $p(\hat{\boldsymbol{\theta}})$ is the joint prior of all the model parameters in $\boldsymbol{\theta}$ except for the $\boldsymbol{A}_j$ and the $\boldsymbol{Q}_j$. Finding an analytic formula for the posterior distribution of the parameters is impossible so we learn the posterior via a two step process. Firstly, we find the maximum of the posterior log likelihood via L-BFGS-B and then we use EMCEE to sample from the posterior (recall that we get good results starting the sampler at the maximum). We use L-BFGS-B since gradient based optimisers are faster than non-gradient based optimisers like Nelder-Mead and it is trivial to compute the derivatives of (3.93) (with respect to $\boldsymbol{\theta}$) using automatic differentiation. Furthermore, L-BFGS-B allows for the use of bounds which is particularly important for several parameters. The above is summarised in algorithm 10.

The model we presented above aims to balance the number of parameters to be optimised with the expressiveness of the model while also limiting the number of user chosen tuning parameters. However, there are still some tuning parameters which are set before training and we will discuss them below.

Tuning Parameters

1. **Covariance (kernel) function**: Due to the requirement to have a spectral density, we are limited to choosing a stationary kernel. However, the neural network mitigates this limitation somewhat and extrapolation outside the training data is better than a standard Gaussian process (with a stationary kernel) particularly when using the adaptive kernel we introduced previously.

2. **Prior hyperparameters for** Q_i: $l = 1$ and $\boldsymbol{\Lambda} = \boldsymbol{I}$ are reasonably good choices but if it is believed that the data is very noisy, then they could be changed.

3. **Model Layout** $[d_0, ..., d_{n-1}]$: This is the most important tuning parameter because the performance of the DHGPM is heavily dependant on the size of the neural network. The model of Solin and Särkkä [2014] depends most heavily on the input dimension and the number of basis functions; in a sense, the DHGPM depends on similar things both of which manifest themselves in the model layout $[d_0, ..., d_{n-1}]$: the larger the input dimension and the more complex the function, the more the DHGPM will require a bigger number of hidden layers and larger d_i in $[d_0, ..., d_{n-1}]$ in order to produce good results. We recommend starting with the layout as small as possible (i.e. one hidden layer and small d_0) and if the results are not good (i.e. underfitting) expand the size of the layout. Also, it is important to remember that for one dimensional inputs a

DHGPM with kernel augmentation function $\sin(x)$ and layout $[m]$ has a similar flexibility to the model of Solin and Särkkä [2014] with m basis functions. We have found good results for DHGPMs with kernel augmentation function $\sin(x)$ and layouts of the form $[d, 3d]$ where d is the input dimension.

4. **Kernel augmentation function (activation function)**: This function is similar to the activation function in a neural network and it is fixed before training. We call it a kernel augmentation function because it is able to extend the power of the kernel beyond those functions which are expected to be drawn from it. For example, an RBF kernel augmented with $\sin(x)$ (i.e. $\sin(x)$ augmentation function) allows us to find smooth periodic functions. In fact, $\sin(x)$ is almost always a good choice of augmentation function due to its ability to extrapolate periodic functions and learn non-periodic functions inside a fixed domain (similar to Fourier analysis). In the context of DHGPMs, $\sin(x)$ is able to learn more complex functions in a much smaller layout than other kernel augmentation functions like $\tanh(x)$. However, for extrapolating non-periodic functions outside the data regime a $\tanh(x)$ or log-sigmoid kernel augmentation function can also work well.

Algorithm 10: Learning of a DHGPM

Input : Training data $\boldsymbol{X}, \boldsymbol{y}$.
Output: Samples from parameter posterior $p(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y})$.

1 initialise unknown parameters $\boldsymbol{\theta}$ randomly;
`/* Make sure the random initial parameters are valid; for example, some parameters must be positive. See the Notes section. */`
2 maximise (3.93) via L-BFGS-B to give the optimal parameters $\boldsymbol{\theta}_*$;
`/* Take note of parameter bounds. Also, using several restarts (with different random initial parameters) can alleviate the issue of getting stuck in a poor local maximum. */`
3 sample from posterior $p(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y})$ using EMCEE with each walker having initial parameters $\boldsymbol{\theta}_* +$ small random noise;
`/* See comments about noise in algorithm 8. For DHGPMs use at least` $2p + 2$ `walkers where` p `is the length of the vector` $\boldsymbol{\theta}_*$`. A total of` 500 `posterior samples with each walker having a burn-in of` 60 `samples usually gives a good characterisation of the posterior. */`
4 **return** samples from parameter posterior;

Let $\Theta = \{\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_m\}$ be our collection of samples from parameter posterior and define $f_{\hat{\boldsymbol{\theta}}}(\boldsymbol{x})$ to be $f(\boldsymbol{x})$ from (3.85) but with the parameters of the DHGPM equal to $\hat{\boldsymbol{\theta}}$. Given a test input $\boldsymbol{x}_*$, we can define (implicitly conditioned on the training data):

$$\mathbb{E}[y_*|\boldsymbol{x}_*] = \mathbb{E}[f(\boldsymbol{x}_*)|\boldsymbol{x}_*] \approx \frac{1}{m}\sum_{i=1}^{m} f_{\boldsymbol{\theta}_i}(\boldsymbol{x}_*) \tag{3.94}$$

$$\mathbb{V}[f(\boldsymbol{x}_*)|\boldsymbol{x}_*] \approx \left(\frac{1}{m}\sum_{i=1}^{m}(f_{\boldsymbol{\theta}_i}(\boldsymbol{x}_*))^2\right) - (\mathbb{E}[f(\boldsymbol{x}_*)|\boldsymbol{x}_*])^2 \tag{3.95}$$

$$\mathbb{V}[y_*|\boldsymbol{x}_*] \approx \mathbb{V}[f(\boldsymbol{x}_*)|\boldsymbol{x}_*] + \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{Q}_n^{\boldsymbol{\theta}_i} \tag{3.96}$$

where $\boldsymbol{Q}_n^{\boldsymbol{\theta}_i}$ is the $\boldsymbol{Q}_n$ from (3.87) with DHGPM parameters $\boldsymbol{\theta} = \boldsymbol{\theta}_i$ and y_* is the test observation corresponding to test input $\boldsymbol{x}_*$.

Issues associated with DHGPMs

As with all Gaussian process models, there are situations in which the DHGPM performs poorly. The potential issues which can occur are discussed below as well as potential remedies.

1. **Local Optima**: In the same way as with standard Gaussian process regression and with the Hilbert reduced-rank model of Solin and Särkkä [2014], it is possible for the optimisation procedure to return a local optimum, and this can lead to poor performance. The solution is to restart the optimisation from several random starting points and this usually solves the issue but has the draw back of increasing training time.

2. **Amount of Data**: Generally we need more data for the DHGPM than with the standard GP. In fact, in the low data regime (less than 200 data points) we would recommend using the standard GP rather than the DHGPM.

3. **Overfitting**: Fortunately, overfitting is rare in this model but there are two situations in which overfitting can occur. Firstly, when the number of data points is small compared with the dimensions of $\boldsymbol{v}_0, ..., \boldsymbol{v}_{n-1}$ and secondly when the training data contains outliers. In the first case, we simply have to reduce the dimensions of $\boldsymbol{v}_0, ..., \boldsymbol{v}_{n-1}$ and in the second case, we should replace (3.87) with a Student-t distribution (see next section). That said, if it is thought that overfitting has occurred in the DHGPM, it is worth testing whether we are in a local optima since the nature of the model (especially with a $\sin(x)$ augmentation function) means that local optima have many of the same characteristics as overfitting. Finally, it is worth pointing out that overfitting is not unique to the DHGPM; in fact, overfitting affects all Gaussian process models and even standard Gaussian process regression often overfits in the presence of outliers.

4. **Underfitting**: We recommended having the layout as small as possible (i.e. the dimensions of $\boldsymbol{v}_0, ..., \boldsymbol{v}_{n-1}$ as small as possible) because this can reduce training times significantly. However, this can lead to underfitting and the solution here is to increase the dimensions of $\boldsymbol{v}_0, ..., \boldsymbol{v}_{n-1}$.

3.3.2 DHGPMs with non-Gaussian noise and Student-t processes

The DHGPM can be adapted to use different noise distributions or to use Student-t processes: all we have to do is change some prior distributions; for example, if we wish to change the noise distribution, we should change (3.87) to the desired noise

distribution and if we wish to use Student-t processes we should change (3.88) to a matrixvariate-t distribution (3.34). However, changing the model may introduce additional parameters that will need to be included in $\boldsymbol{\theta}$ if we want them to be optimised/sampled from. The changes to the prior distributions lead to changes in (3.93); for example, using a Student-t process with Gaussian noise leads to (3.93) becoming:

$$\Big(\prod_{i=1}^{N}\mathcal{N}(y_i|f(\boldsymbol{x}_i),\boldsymbol{Q}_n)\Big)\Big(\prod_{j=0}^{n}\mathcal{MT}_{\nu}(\boldsymbol{A}_j|\boldsymbol{0},\boldsymbol{Q}_j,\boldsymbol{V}_j)\mathcal{IW}(\boldsymbol{Q}_j|r_j+l,\boldsymbol{\Lambda})\Big)p(\hat{\boldsymbol{\theta}}) \tag{3.97}$$

where $\mathcal{MT}_{\nu}$ is a matrixvariate-t distribution with ν degrees of freedom and ν is usually fixed before training and not optimised. Another example is a Student-t process with Student-t noise which leads to (3.93) becoming:

$$\Big(\prod_{i=1}^{T}\mathcal{T}(y_i|f(\boldsymbol{x}_i),\boldsymbol{Q}_n,\nu_1)\Big)\Big(\prod_{j=0}^{n}\mathcal{MT}_{\nu_2}(\boldsymbol{A}_j|\boldsymbol{0},\boldsymbol{Q}_j,\boldsymbol{V}_j)\mathcal{IW}(\boldsymbol{Q}_j|r_j+l,\boldsymbol{\Lambda})\Big)p(\hat{\boldsymbol{\theta}}) \tag{3.98}$$

where $\mathcal{T}$ is a multivariate-t distribution. In this case, the ν_1 in $\mathcal{T}$ is optimised and so must be added to $\boldsymbol{\theta}$. Since we use EMCEE and automatic differentiation these changes to (3.93) do not introduce any additional difficulties; we just follow the same learning algorithm as before while using a suitable replacement to (3.93) and potentially optimising some additional parameters. However, note that (3.96) assumes that the noise distribution is Gaussian and for a general noise distribution (with mean zero) we use that (via iterated variances):

$$\mathbb{V}[y_*|\boldsymbol{x}_*] = \mathbb{V}[f(\boldsymbol{x}_*)|\boldsymbol{x}_*] + \frac{1}{m}\sum_{i=1}^{m}\mathbb{V}[p(y_*|\boldsymbol{x}_*,\boldsymbol{\theta}_i)] \tag{3.99}$$

where as before $\Theta = \{\boldsymbol{\theta}_1,...,\boldsymbol{\theta}_m\}$ is our collection of samples from the parameter posterior and $\mathbb{V}[f(\boldsymbol{x}_*)|\boldsymbol{x}_*]$ is calculated using (3.94) and (3.95). In the case of Student-t noise, this becomes:

$$\mathbb{V}[y_*|\boldsymbol{x}_*] = \mathbb{V}[f(\boldsymbol{x}_*)|\boldsymbol{x}_*] + \frac{1}{m}\sum_{i=1}^{m}\frac{\nu^{\boldsymbol{\theta}_i}}{\nu^{\boldsymbol{\theta}_i}-2}\boldsymbol{Q}_n^{\boldsymbol{\theta}_i} \tag{3.100}$$

if $\min_i(\nu^{\boldsymbol{\theta}_i}) > 2$ otherwise it does not exist (is infinite). Here, $\nu^{\boldsymbol{\theta}_i}$ and $\boldsymbol{Q}_n^{\boldsymbol{\theta}_i}$ are respectively the noise degrees of freedom ν and the noise scale $\boldsymbol{Q}_n$ under parameters $\boldsymbol{\theta} = \boldsymbol{\theta}_i$.

3.3.3 Examples

In this section, we look at several examples of DHGPMs on a variety of synthetic and non-synthetic datasets. When comparing models, we look at three performance statistics: the root mean square error (RMSE), the mean log likelihood (LL) and the mean training time. Suppose we have a test dataset $\mathcal{D}^* = \{(\boldsymbol{x}_i^*, y_i^*) \text{ for } i = 1, ..., N^*\}$, then for the full GP and Hilbert reduced-rank GP we define the RMSE and LL as in (3.8) and (3.9) respectively. Further suppose that we have samples $\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_K$ from the parameter posterior, then for the DHGPM we define the RMSE as:

$$RMSE = \sqrt{\frac{1}{K}\sum_{i=1}^{N^*}(y_i^* - \mathbb{E}[f(\boldsymbol{x}_i^*)])^2} \tag{3.101}$$

where $\mathbb{E}[f(\boldsymbol{x}_i^*)]$ is defined in (3.94). If the DHGPM assumes a Gaussian noise distribution, then the mean log likelihood (LL) is defined as:

$$LL_{gauss} = \frac{1}{N^*}\sum_{i=1}^{N^*}\log\mathcal{N}(y_i^*|\mathbb{E}[f(\boldsymbol{x}_i^*)], \mathbb{E}[f(\boldsymbol{x}_i^*)] + \mathbb{E}[\boldsymbol{Q}_n]) \tag{3.102}$$

where $\mathbb{E}[\boldsymbol{Q}_n] \approx \frac{1}{K}\sum_{j=1}^{K}\boldsymbol{Q}_n^{\boldsymbol{\theta}_j}$ and $\boldsymbol{Q}_n^{\boldsymbol{\theta}_j}$ is the value of $\boldsymbol{Q}_n$ when the model parameters are equal to $\boldsymbol{\theta}_j$. Finally, for models which assume a Student-t noise distribution, we define the mean log likelihood (LL) as:

$$LL_{Student-t} = \frac{1}{KN^*}\sum_{i=1}^{N^*}\sum_{j=1}^{K}\log\mathcal{T}(y_i^*|f_{\boldsymbol{\theta}_j}(x_i^*), \boldsymbol{Q}_n^{\boldsymbol{\theta}_j}, \nu^{\boldsymbol{\theta}_j}) \tag{3.103}$$

where $\mathcal{T}(x|\mu, \Sigma, \nu)$ is a Student-t density function with mean μ, scale Σ and ν degrees of freedom (using the parametrisation of (3.25)). The notation $\boldsymbol{Q}_n^{\boldsymbol{\theta}_j}$ and $\nu^{\boldsymbol{\theta}_j}$ simply means the values of $\boldsymbol{Q}_n$ and ν (noise degrees of freedom), respectively, when the model parameters are equal to $\boldsymbol{\theta}_j$.

All models will be given 10 optimisation restarts and the training time will be defined as the mean time for an optimisation restart. In the DHGPM diagrams, we plot both the model mean and the model optimum. The *model optimum* is the value of the model under the parameters $\boldsymbol{\theta}_*$ found during the optimisation (Algorithm 10, line 2) and the *model mean* is the mean value of the model using the parameter samples and (3.94). Ideally, these two values should be very close and if they are far apart, then this suggests that the sampling was not successful.

Example 1:
Aim: This is a 'sanity-check' example. It compares the performance of the full GP, Hilbert reduced-rank GP and the DHGPM on a simple system. We expect all the models to produce similar results.

Training Data: 250 pairs (x, y) with each x generated by sampling from $\mathcal{U}(-3, 3)$ and y generated using system:

$$y = 2.5e^{-x^2} + w_t \tag{3.104}$$

$$w_t \sim \mathcal{N}(0, 1^2) \tag{3.105}$$

Test Data: 500 pairs (x, y) with x equally spaced between -3 and 3 and y generated as above.

Additional Information:

- The manifold-Hilbert reduced-rank models will use a neural network with $\tanh$ activations and 2 hidden layers of 1 and 6 neurons respectively.
- The DHGPMs will use $sin(x)$ activation functions.

More information is summarised in table 3.7.

Model	**Kernel**	**Process**	**Assumed Noise Distribution**	**Additional Information**
Full GP 1	RBF	Gaussian	Gaussian	N/A
Full GP 2	Matern32 + MLP	Gaussian	Gaussian	N/A
Hilbert reduced-rank GP	RBF	Gaussian	Gaussian	12 basis functions, $L = 5$
Manifold-Hilbert reduced-rank GP	Matern32	Gaussian	Gaussian	12 basis functions, $L = 5$
DHGPM 1	Matern52	Gaussian	Student-t	layout = $[6]$
DHGPM 2	Adaptive	Student-t	Gaussian	layout = $[2, 2]$

Table 3.7: Additional Information for Example 1

Results:

Model	**RMSE (on test data)**	**LL (on test data)**	**Mean Time (seconds)**
Full GP 1	**0.99**	-1.42	0.14
Full GP 2	**0.99**	**-1.41**	0.90
Hilbert reduced-rank GP	**0.99**	**-1.41**	**0.19**
Manifold-Hilbert reduced-rank GP	**0.99**	-1.42	3.20
DHGPM 1	**0.99**	-1.44	2.95
DHGPM 2	**0.99**	**-1.41**	3.10

Table 3.8: Results for Example 1

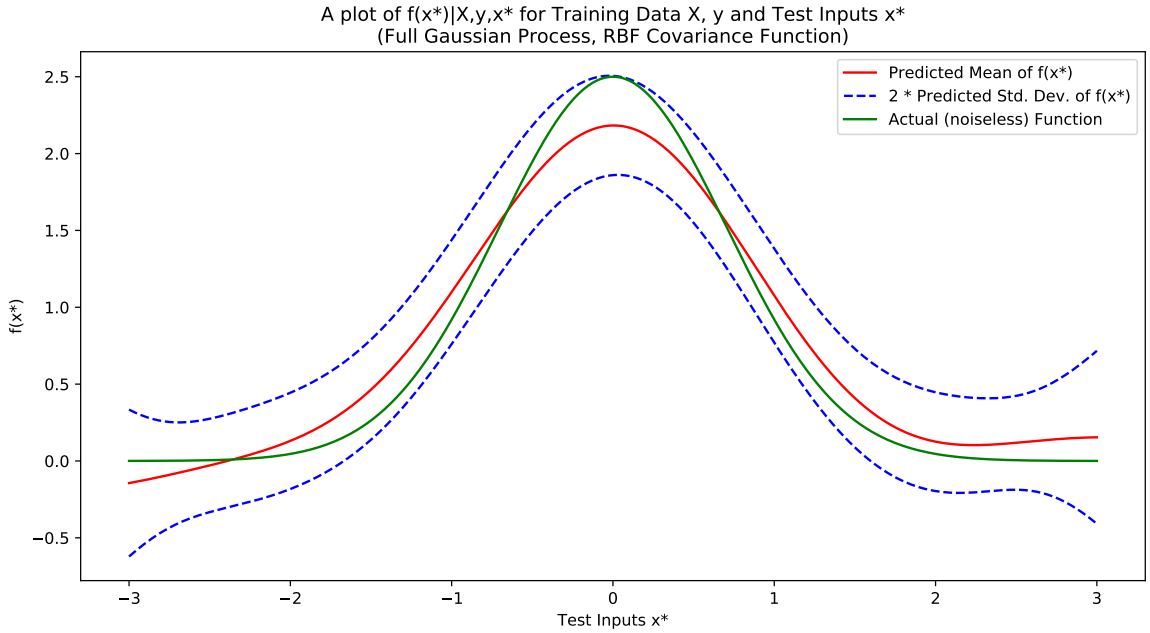

Figure 3.14: A full Gaussian process
(Table 3.7, Row 1)

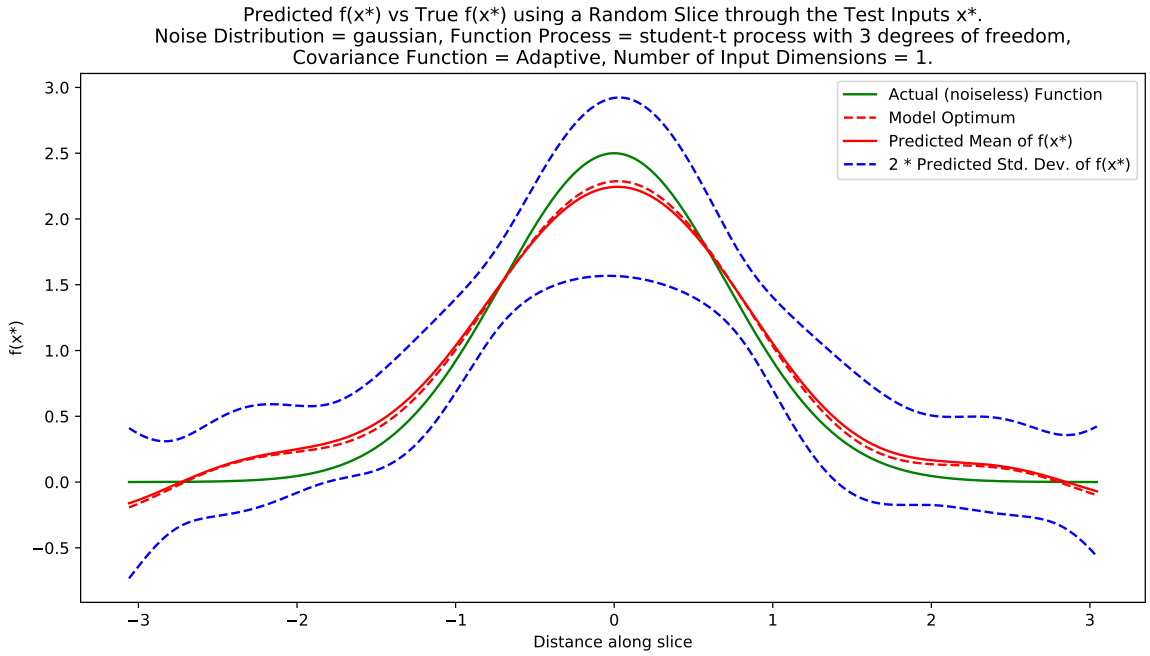

Figure 3.15: A DHGPM with a Student-t process
(Table 3.7, Row 6)

Example 2:
Aim: To compare the performance of the full GP, Hilbert reduced-rank GP and the DHGPM on dataset with Student-t noise.

Training Data: 250 pairs (x, y) with each x generated by sampling from $\mathcal{U}(-3, 3)$ and y generated using system:

$$y = 4|\tanh(0.5x + 0.5\cos(0.8x))| + w_t \tag{3.106}$$
$$w_t \sim \mathcal{N}(0, 1^2, \nu = 1) \tag{3.107}$$

Test Data: 2000 pairs (x, y) with x equally spaced between -3.3 and 3.3 and y generated as above.

Additional Information:

- The manifold-Hilbert reduced-rank models will use a neural network with log-sigmoid activations and 2 hidden layers of 3 and 2 neurons respectively.
- The DHGPMs will use $sin(x)$ activation functions.

More information is summarised in table 3.9.

Model	**Kernel**	**Process**	**Assumed Noise Distribution**	**Additional Information**
Full GP 1	RBF	Gaussian	Gaussian	N/A
Full GP 2	Matern32 + RatQuad	Gaussian	Gaussian	N/A
Hilbert reduced-rank GP	RBF	Gaussian	Gaussian	12 basis functions, $L = 5$
Manifold-Hilbert reduced-rank GP	Matern32	Gaussian	Gaussian	12 basis functions, $L = 5$
DHGPM 1	RBF	Gaussian	Student-t	layout = $[4, 1]$
DHGPM 2	Adaptive	Student-t	Student-t	layout = $[4, 1]$

Table 3.9: Additional Information for Example 2

Results:
Since the true noise distribution is Student-t, we do not provide a RMSE because it is unhelpful under Student-t noise due to the presence of large outliers.

Model	**LL (on test data)**	**Mean Time (seconds)**
Full GP 1	-8.91	0.40
Full GP 2	-9.04	0.92
Hilbert reduced-rank GP	-26.57	**0.11**
Manifold-Hilbert reduced-rank GP	-27.10	0.72
DHGPM 1	-2.63	3.52
DHGPM 2	**-2.62**	2.76

Table 3.10: Results for Example 2

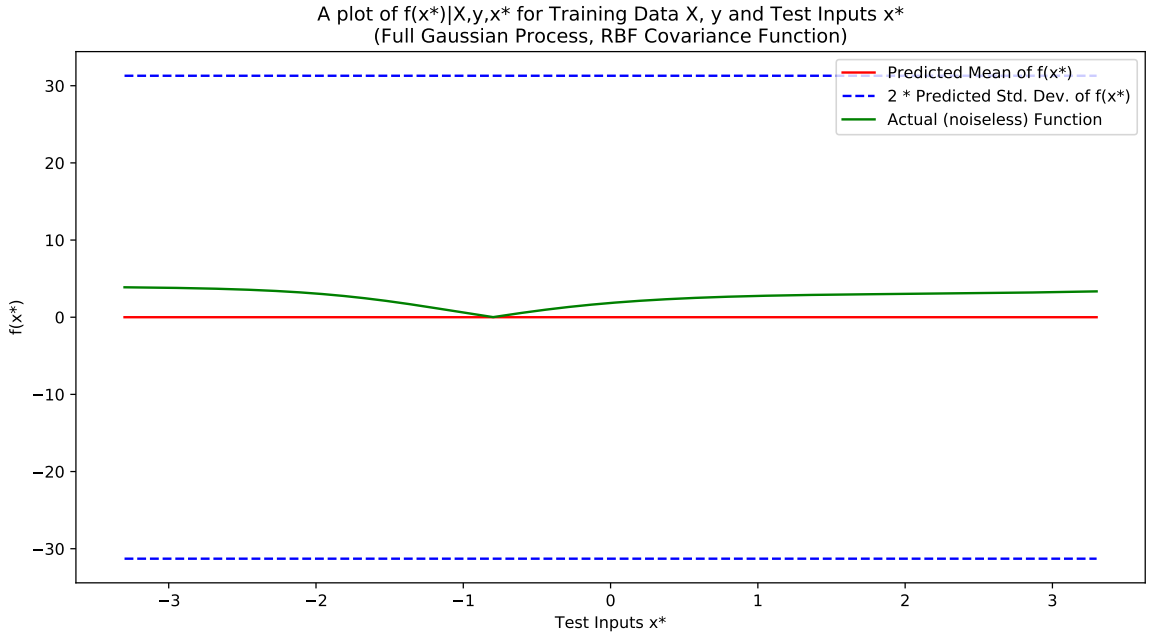

Figure 3.16: A full Gaussian process with the noise assumed to be Gaussian (Table 3.9, Row 1)

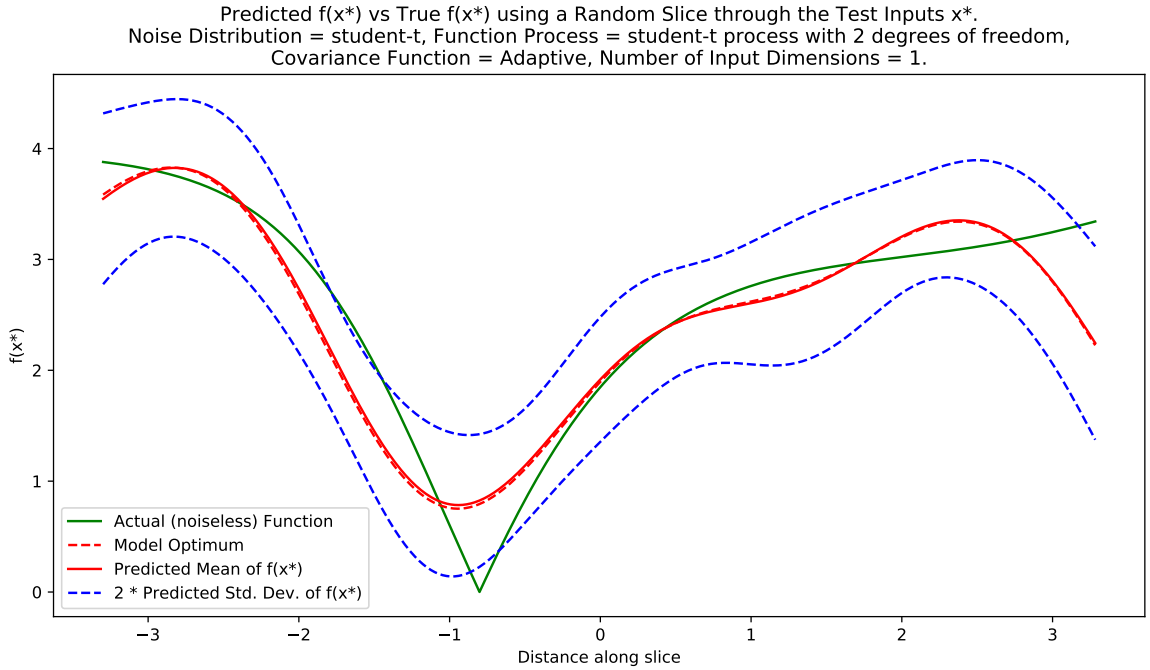

Figure 3.17: A DHGPM with the noise assumed to be Student-t (Table 3.9, Row 6)

Now consider the same example but with 2500 data points for training:

Model	Kernel	Process	Assumed Noise Distribution	Additional Information
Full GP	RBF	Gaussian	Gaussian	N/A
DHGPM	RBF	Gaussian	Student-t	layout $= [4, 2]$

Table 3.11: Additional Information for Example 2 (part 2)

Model	LL (on test data)	Mean Time (seconds)
Full GP	-6.08	101.19
DHGPM	**-2.55**	**10.45**

Table 3.12: Results for Example 2 (part 2)

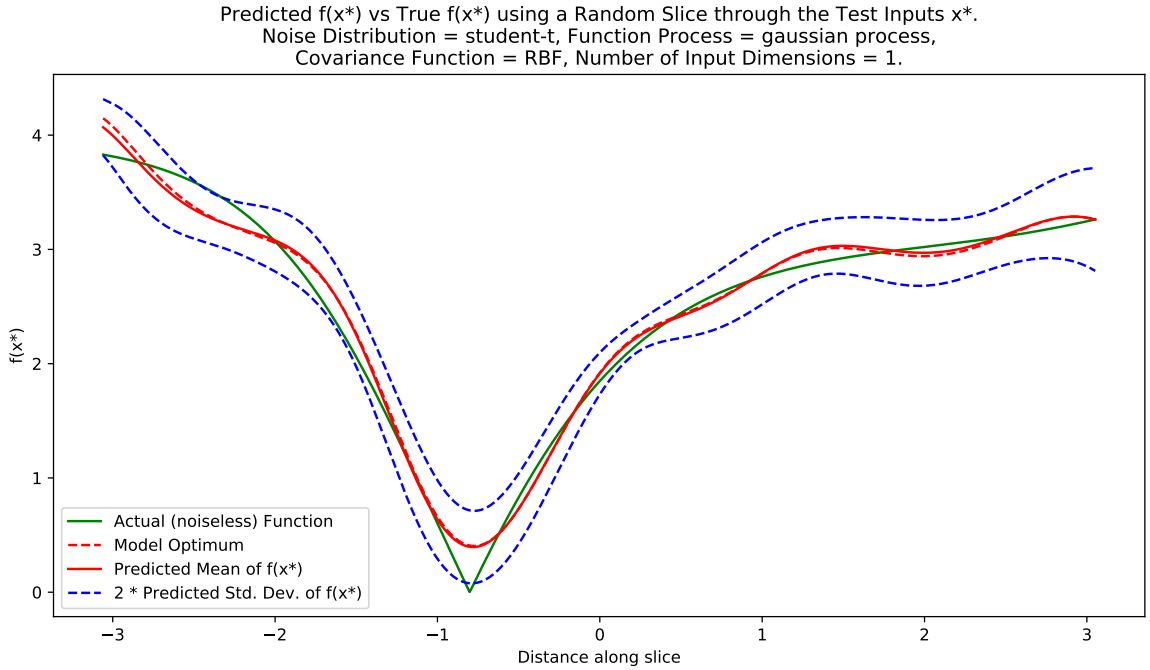

Figure 3.18: A DHGPM (Table 3.11, Row 2)

Example 3:
Aim: To demonstrate that the DHGPM can work well with multidimensional input data and also to show that the DHGPM does not overfit in the presence of very noisy data.

Training Data: 1000 pairs $(\boldsymbol{x}, y)$ with each $\boldsymbol{x} = (x_1, x_2, x_3)$ generated by sampling x_i from $\mathcal{U}(-3, 3)$ for $i = 1, .., 3$ and y generated using system:

$$y = x_1 + x_2 + x_3 + w_t \tag{3.108}$$
$$w_t \sim \mathcal{N}(0, 5^2) \tag{3.109}$$

Test Data: 10000 pairs (x, y) with each $\boldsymbol{x} = (x_1, x_2, x_3)$ generated by sampling x_i from $\mathcal{U}(-3.6, 3.6)$ for $i = 1, .., 3$ and y generated as above.

Additional Information:

- The DHGPMs will use $sin(x)$ activation functions.

More information is summarised in table 3.13.

Model	Kernel	Process	Assumed Noise Distribution	Additional Information
Full GP 1	RBF	Gaussian	Gaussian	N/A
Full GP 2	Matern32 + RatQuad	Gaussian	Gaussian	N/A
DHGPM 1	Adaptive	Gaussian	Gaussian	layout $= [2, 1]$
DHGPM 2	Adaptive	Student-t	Gaussian	layout $= [2, 1]$

Table 3.13: Additional Information for Example 3

Results:

Model	RMSE (on test data)	LL (on test data)	Mean Time (seconds)
Full GP 1	5.03	**-3.03**	10.24
Full GP 2	5.03	**-3.03**	31.96
DHGPM 1	5.06	-3.04	**2.22**
DHGPM 2	**5.02**	**-3.03**	2.91

Table 3.14: Results for Example 3

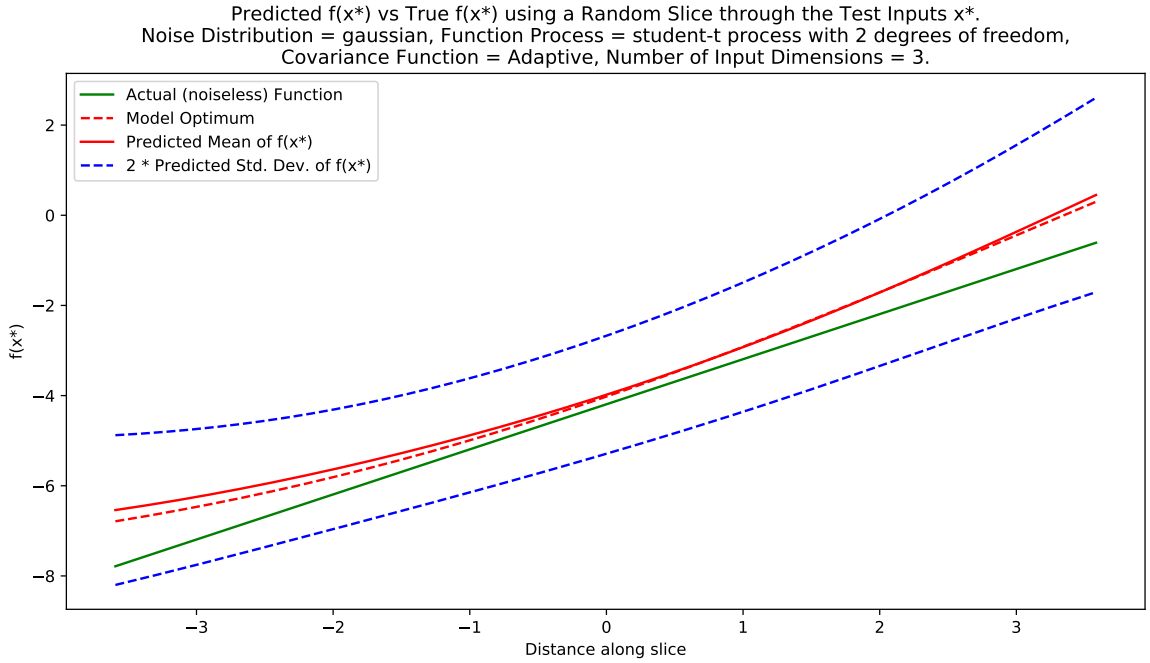

Figure 3.19: A DHGPM with a Student-t process (Table 3.13, Row 4)

Example 4:
Aim: To demonstrate that the DHGPM can work well with multidimensional input data in low noise environments.

Training Data: 1000 pairs $(\boldsymbol{x}, y)$ with each $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5)$ generated by sampling x_i from $\mathcal{U}(-3, 3)$ for $i = 1, .., 5$ and y generated using system:

$$y = |0.1x_1 - 0.2x_2 + 0.3x_3 - 0.4x_4 + 0.5x_5| + w_t \tag{3.110}$$
$$w_t \sim \mathcal{N}(0, 0.1^2) \tag{3.111}$$

Test Data: 10000 pairs (x, y) with each $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5)$ generated by sampling x_i from $\mathcal{U}(-3.6, 3.6)$ for $i = 1, .., 5$ and y generated as above.

Additional Information:

- The DHGPMs will use $sin(x)$ activation functions.

More information is summarised in table 3.15.

Model	**Kernel**	**Process**	**Assumed Noise Distribution**	**Additional Information**
Full GP 1	RBF	Gaussian	Gaussian	N/A
Full GP 2	Matern32 + RatQuad	Gaussian	Gaussian	N/A
DHGPM 1	RBF	Gaussian	Gaussian	layout = $[10]$
DHGPM 2	RBF	Student-t	Student-t	layout = $[5, 3]$

Table 3.15: Additional Information for Example 4

Results:

Model	**RMSE (on test data)**	**LL (on test data)**	**Mean Time (seconds)**
Full GP 1	0.22	0.20	**7.85**
Full GP 2	0.17	0.37	34.70
DHGPM 1	0.16	0.31	12.35
DHGPM 2	**0.11**	**0.72**	15.18

Table 3.16: Results for Example 4

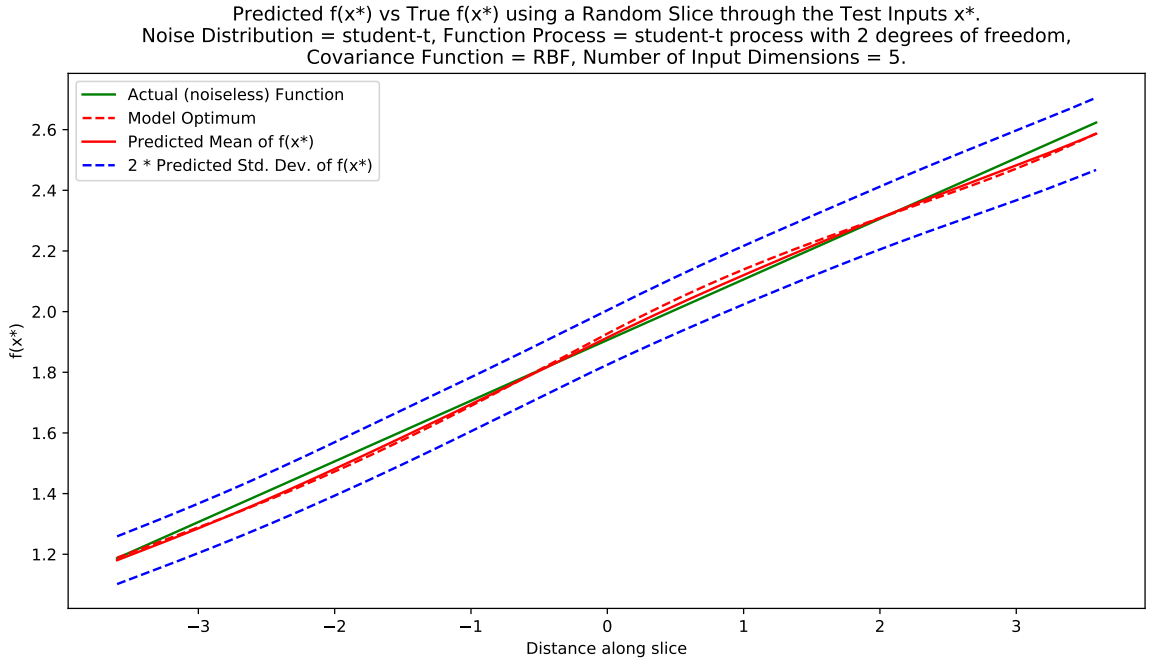

Figure 3.20: A DHGPM (Table 3.15, Row 4)

Example 5:
Aim: To demonstrate that the DHGPM can work well on non-synthetic datasets.

The Datasets:
The first three datasets are from Lichman [2013] (see footnotes for the links to the datasets) and the final dataset was created using data from Chicago Board Options Exchange [2017] and Yahoo Finance [2017]. Associated with each of the datasets is a corresponding task, for the first three datasets this task is described in the provided links and for the final dataset the task is described below. We compare the performance of the DHGPM and the full GP for each of these tasks.

1. The Airfoil Noise Dataset[2]: this dataset contains 1503 datapoints of which we use a randomly selected 1000 points for training and the rest as a test set. The inputs have 5 dimensions and we try to learn a 1 dimensional output.

2. The Combined Cycle Power Plant Dataset[3]: this dataset contains 9568 datapoints of which we use a randomly selected 5000 points for training and the rest as a test set. The inputs have 4 dimensions and we try to learn a 1 dimensional output.

3. The Concrete Compressive Strength Dataset[4]: this dataset contains 1039 datapoints of which we use a randomly selected 800 points for training and the rest as a test set. The inputs have 8 dimensions and we try to learn a 1 dimensional output.

[2]https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise
[3]https://archive.ics.uci.edu/ml/datasets/Combined+Cycle+Power+Plant
[4]https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength

4. The final dataset consists of a collection of Open, High, Low, Close and Volume data points for the S&P 500[5] and the Open Price of the VIX Index[6]. The aim is to calculate the VIX Open Price from the S&P 500 data (on the same day). We note that VIX is the implied volatility of the S&P 500 and so their is indeed a relation between the two. This dataset contains 3436 datapoints of which we use first 3000 for training and the rest as a test set. The inputs have 5 dimensions and we try to learn a 1 dimensional output.

Additional Information:

- The DHGPMs will use $sin(x)$ activation functions.

More information is summarised in table 3.17.

Model	**Kernel**	**Process**	**Assumed Noise Distribution**	**Additional Information**
Full GP 1	RBF	Gaussian	Gaussian	N/A
DHGPM 1	RBF	Gaussian	Gaussian	layout = $[5, 8]$
DHGPM 2	RBF	Gaussian	Student-t	layout = $[8, 24]$
DHGPM 3	RBF	Gaussian	Gaussian	layout = $[16]$
DHGPM 4	Adaptive	Student-t	Student-t	layout = $[3, 2]$

Table 3.17: Additional Information for Example 5

Results:

Model	**RMSE (on test data)**	**LL (on test data)**	**Mean Time (seconds)**
Full GP 1	2.35	-0.41	**5.71**
DHGPM 1	**2.11**	**-0.26**	35.18

Table 3.18: Results for the Airfoil Noise Dataset

Model	**RMSE (on test data)**	**LL (on test data)**	**Mean Time (seconds)**
Full GP 1	3.94	0.04	414.94
DHGPM 2	**3.88**	**0.07**	**257.07**

Table 3.19: Results for the Combined Cycle Power Plant Dataset

[5]https://finance.yahoo.com/quote/%5EGSPC/

[6]http://www.cboe.com/products/vix-index-volatility/vix-options-and-futures/vix-index/vix-historical-data

Model	RMSE (on test data)	LL (on test data)	Mean Time (seconds)
Full GP 1	5.31	-0.27	**4.92**
DHGPM 3	**5.01**	**-0.22**	28.78

Table 3.20: Results for the Concrete Compressive Strength Dataset

Model	RMSE (on test data)	LL (on test data)	Mean Time (seconds)
Full GP 1	10.15	-1.25	145.58
DHGPM 4	**4.92**	**-0.91**	**24.62**

Table 3.21: Results for the S&P 500 – VIX Dataset

Discussion:
Overall these examples demonstrate that the DHGPM is a very competitive model that can often outperform the full GP. While on small datasets (< 1000 data points) the DHGPM is slower than the full GP, the training time scales well as the dataset size increases and on larger datasets, the DHGPM will be much faster than the full GP. In comparison to the Hilbert reduced-rank GP of Solin and Särkkä [2014], the DHGPM scales significantly better as the input dimension increases.

3.4 Learning GPSSMs with High-Dimensional Latent Spaces

3.4.1 The Deep Hilbert Gaussian Process State-Space Model

In a similar way to how Svensson et al. [2016] uses the Hilbert reduced-rank GP to create a reduced-rank GPSSM, we will use the DHGPM to construct a novel GPSSM that is particularly useful when we wish to have high-dimensional latent spaces. We will call our new GPSSM the deep Hilbert Gaussian process state-space model (DHGPSSM) and in this section, we will explain the model and provide both a training and prediction algorithm.

The main idea for learning DHGPSSMs is similar to the learning algorithm we created for reduced-rank GPSSMs with Student-t noise: we alternate between sampling the latent states given the model parameters and optimising/sampling the model parameters given the latent states. We will focus on DHGPSSMs with a latent DHGPM but a known observation distribution because there is little reason to have both an observation and transition DHGPM since this increases the computational cost while not providing any additional flexibility. The reason is due to identifiability and the same issue effects all GPSSMs; for more details see Frigola [2015, pp. 32-33].

Set-up:
Suppose we have observations $\boldsymbol{y}_1, ..., \boldsymbol{y}_T$ where $\boldsymbol{y}_t \in \mathbb{R}^{n_y}$ for $t = 1, ..., T$ and that the latent state dimension is n_x i.e. $\boldsymbol{x}_t \in \mathbb{R}^{n_x}\ \forall t$. A deep Hilbert Gaussian process State Space model (DHGPSSM) with $n \in \mathbb{Z}_{\geq 0}$ hidden layers and kernel augmentation function ϕ is constructed as follows:

$$x_{t+1}^i = f_i(\boldsymbol{x}_t) + \epsilon_t^i \text{ for } i = 1, ..., n_x \tag{3.112}$$
$$\boldsymbol{x}_{t+1} = (x_{t+1}^1, ..., x_{t+1}^{n_x}) \in \mathbb{R}^{n_x} \tag{3.113}$$
$$\epsilon_t^i \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_{n,i}) \text{ for } i = 1, ..., n_x \tag{3.114}$$
$$\boldsymbol{v}_{0,i}^t = \boldsymbol{A}_{0,i}\boldsymbol{x}_t + \boldsymbol{b}_{0,i} \text{ for } i = 1, ..., n_x \tag{3.115}$$
$$\boldsymbol{v}_{j,i}^t = \boldsymbol{A}_{j,i}\phi(\boldsymbol{v}_{j-1,i}^t) + \boldsymbol{b}_{j,i} \text{ for } j = 1, ..., n-1 \text{ for } i = 1, ..., n_x \tag{3.116}$$
$$f_i(\boldsymbol{x}_t) = \boldsymbol{A}_{n,i}\phi(\boldsymbol{v}_{n-1,i}^t) \text{ for } i = 1, ..., n_x \tag{3.117}$$
$$f(\boldsymbol{x}_t) = (f_1(\boldsymbol{x}_t), ..., f_{n_x}(\boldsymbol{x}_t)) \in \mathbb{R}^{n_x} \tag{3.118}$$
$$\boldsymbol{A}_{j,i}|\boldsymbol{Q}_{j,i} \sim \mathcal{MN}(\boldsymbol{0}, \boldsymbol{Q}_{j,i}, \boldsymbol{V}_{j,i}) \text{ for } j = 0, ..., n \text{ for } i = 1, ..., n_x \tag{3.119}$$
$$\boldsymbol{Q}_{j,i} \sim \mathcal{IW}(r_{j,i} + l, \boldsymbol{\Lambda}) \text{ for } j = 0, ..., n \text{ for } i = 1, ..., n_x \tag{3.120}$$
$$\boldsymbol{V}_{j,i} = \text{diag}(S_{j,i}^{-1}(\sqrt{\lambda_1^{ji}}), ..., S_{j,i}^{-1}(\sqrt{\lambda_{c_{j,i}}^{ji}})) \text{ for } j = 0, ..., n \text{ for } i = 1, ..., n_x \tag{3.121}$$
$$\lambda_s^{ji} = \left(\frac{\pi s}{2L_{ji}}\right)^2 \text{ for } j = 0, ..., n \text{ for } i = 1, ..., n_x \tag{3.122}$$
$$\boldsymbol{y}_t \sim p(\boldsymbol{y}_t|\boldsymbol{x}_t, \boldsymbol{\theta}_y) \tag{3.123}$$

where equations (3.112) to (3.122) form a latent multi-output DHGPM on $\boldsymbol{x}_1, ..., \boldsymbol{x}_T$ and we have an observation equation (3.123) linking the latent and observed states. The various variables are discussed in the *Notes* section and a simple graphical model is provided in figure 3.21.

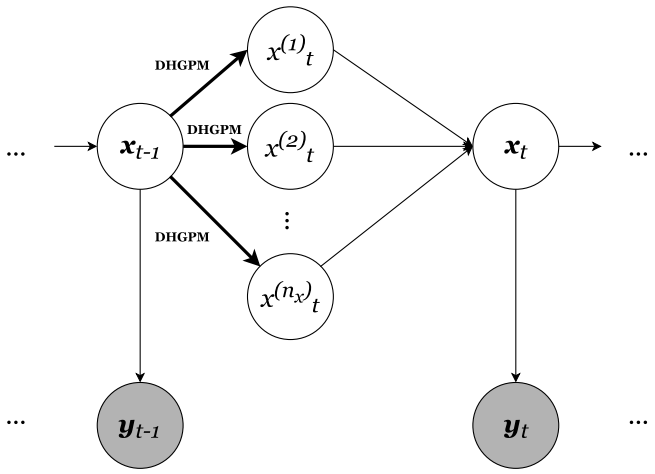

Figure 3.21: A DHGPSSM: the model trains independent DHGPMs that map $\boldsymbol{x}_{t-1}$ to x_t^i for $i = 1, ..., n_x$ where x_t^i (written as $x_t^{(i)}$ in the diagram) is the i^{th} dimension of $\boldsymbol{x}_t$.

Although this model looks complex, the idea is conceptually straightforward: learn a latent DHGPM f that maps $\boldsymbol{x}_t$ to $\boldsymbol{x}_{t+1}$. This means that given the latent states

$\boldsymbol{x}_1, ..., \boldsymbol{x}_T$, we need to find the parameters of the DHGPM using training data $\mathcal{D} = \{(\boldsymbol{x}_{t+1}, \boldsymbol{x}_t) \text{ for } t = 1, ..., T-1\}$ where for each t, $\boldsymbol{x}_t$ is the input of the DHGPM and $\boldsymbol{x}_{t+1}$ is the output. The challenge here is that the output $\boldsymbol{x}_{t+1}$ is multidimensional in general, so we could have multidimensional inputs and outputs whereas the DHGPM assumes we have a single-dimensional output. However, as we mentioned in the section on DHGPMs, to deal with multiple outputs, we fit independent DHGPMs to each of the output dimensions. For example, suppose that the latent dimension is n_x and hence $\boldsymbol{x}_t = (x_t^1, ..., x_t^{n_x})\ \forall t$, then all we need to do is to split the training data $\mathcal{D}$ into $\mathcal{D}_i = \{(x_{t+1}^i, \boldsymbol{x}_t) \text{ for } t = 1, ..., T-1\}$ for $i = 1, ..., n_x$ and for each $\mathcal{D}_i$ we train an independent univariate output DHGPM i.e. we learn a map between $\boldsymbol{x}_t$ and x_{t+1}^i for $i = 1, ..., n_x$.

Notes:

1. We will use $\boldsymbol{\rho}_i$ to refer to all the parameters in (3.112)–(3.123) with an i subscript; for example, the $\boldsymbol{A}_{j,i} \forall j$, $\boldsymbol{b}_{j,i} \forall j$, $\boldsymbol{Q}_{j,i} \forall j$, $L_{ji} \forall j$ and the hyperparameters of the spectral density $S_{j,i}$ for all j are included in $\boldsymbol{\rho}_i$. Therefore, all the parameters $\boldsymbol{\theta}$ can be written as $(\boldsymbol{\rho}_1, ..., \boldsymbol{\rho}_{n_x}, \boldsymbol{\theta}_y)$.

2. We use independent DHGPMs when dealing with multiple outputs i.e. a multidimensional latent space. This means that $\boldsymbol{\rho}_{i_1}$ and $\boldsymbol{\rho}_{i_2}$ are independent for all $i_1 \neq i_2$ and hence can be optimised/sampled separately.

3. For fixed i, equations (3.112)–(3.122) form a DHGPM which maps $\boldsymbol{x}_t \in \mathbb{R}^{n_x}$ to $x_{t+1}^i \in \mathbb{R}$. All the points from the *Notes* part in the section on DHGPMs still apply to each of these independent DHGPMs and explain all the parameters not defined here.

4. For simplicity, we generally use the same tuning parameters for each of the independent latent DHGPMs; for example, for each i the form covariance function of each DHGPM should be the same; however, there is no such constraint on the covariance hyperparameters since the $\boldsymbol{\rho}_i$ are independent. Moreover, the layout should be the same for all DHGPMs and so the layout of the i^{th} DHGPM $[d_0^i, ..., d_{n-1}^i] = [d_0, ..., d_{n-1}] \forall i$ which leads to $c_{j,i} = c_j \forall i$ and $r_{j,i} = r_j \forall i$. Hence, when we refer to the layout of a DHGPSSM we are referring to $[d_0, ..., d_{n-1}]$, which is the layout for all the independent latent DHGPMs.

5. It is simple to adapt the model to have both an observation and latent DHGPM but we assume the form of the observation distribution is known (3.123) with some unknown parameters $\boldsymbol{\theta}_y$ which are optimised/sampled.

6. Adapting the model for latent Student-t noise or latent Student-t processes is simple. In a similar way to in DHGPMs, for Student-t noise just replace (3.114) with a Student-t distribution and for a Student-t process replace (3.119) with a matrixvariate-t distribution. Of course, this may introduce some additional parameters which need to be optimised.

7. Due to the independent DHGPMs, the noise (3.114) must be independent for different i. In particular, this means that we cannot have a multivariate-normal or multivariate-t distribution for $\boldsymbol{\epsilon}_t = (\epsilon_t^1, ..., \epsilon_t^{n_x})$: we require independent univariate-normal or univariate-t distributions for each of the $\boldsymbol{\epsilon}_t^i$.

8. Adapting the model for online or forgetful online learning is simple using exactly the same techniques we explained in section 3.2.1.

9. The model is also required to have an initial latent distribution $p(\boldsymbol{x}_1)$; however, this distribution is of little importance and so we usually take $\boldsymbol{x}_1 \sim \mathcal{N}(\mathbf{0}, 100\boldsymbol{I})$ where $\boldsymbol{I}$ is an identity matrix of suitable size.

Training:
The training process for this model is slightly more complex than in DHGPMs. The idea behind the training algorithm for DHGPSSMs is to balance optimisation/sampling of parameters with learning the latent states: we alternate between these two tasks with our knowledge of the latent states determining how much we can optimise.

One of the key parts of the training process is to optimise the model parameters given the observations and latent states i.e. to maximise $p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$ with respect to $\boldsymbol{\theta}$. Now given observations $\boldsymbol{y}_{1:T}$ and latent states $\boldsymbol{x}_{1:T}$ we get:

$$p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T}) \tag{3.124}$$
$$\propto p(\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{3.125}$$
$$\propto p(\boldsymbol{y}_{1:T}|\boldsymbol{x}_{1:T}, \boldsymbol{\theta}_y)p(\boldsymbol{x}_{1:T}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{3.126}$$

since we have independent DHGPMs for each output dimension in the latent states we get:

$$p(\boldsymbol{y}_{1:T}|\boldsymbol{x}_{1:T}, \boldsymbol{\theta}_y)p(\boldsymbol{x}_{1:T}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{3.127}$$
$$\propto p(\boldsymbol{y}_{1:T}|\boldsymbol{x}_{1:T}, \boldsymbol{\theta}_y)p(\boldsymbol{x}_{1:T}|\boldsymbol{\rho}_1, ..., \boldsymbol{\rho}_{nx})p(\boldsymbol{\rho}_1, ..., \boldsymbol{\rho}_{nx}, \boldsymbol{\theta}_y) \tag{3.128}$$
$$\propto p(\boldsymbol{y}_{1:T}|\boldsymbol{x}_{1:T}, \boldsymbol{\theta}_y)p(\boldsymbol{\theta}_y)\prod_{i=1}^{n_x} p(\boldsymbol{x}_{1:T}|\boldsymbol{\rho}_i)p(\boldsymbol{\rho}_i) \tag{3.129}$$

It can be seen that we have split the parameters into independent positive parts that are multiplied and hence they can be optimised separately. Now consider $p(\boldsymbol{x}_{1:T}|\boldsymbol{\rho}_i)p(\boldsymbol{\rho}_i)$:

$$p(\boldsymbol{x}_{1:T}|\boldsymbol{\rho}_i)p(\boldsymbol{\rho}_i) \tag{3.130}$$
$$\propto \left(\prod_{t=1}^{T-1} \mathcal{N}(x_{t+1}^i|f_i(\boldsymbol{x}_t), \boldsymbol{Q}_{n,i})\right)\left(\prod_{j=0}^{n} \mathcal{MN}(\boldsymbol{A}_{j,i}|\mathbf{0}, \boldsymbol{Q}_{j,i}, \boldsymbol{V}_{j,i})\mathcal{IW}(\boldsymbol{Q}_{j,i}|r_{j,i}+l, \boldsymbol{\Lambda})\right)p(\hat{\boldsymbol{\rho}}_i) \tag{3.131}$$

where n is the number of hidden layers for the i^{th} latent DHGPM (which is the same for all i) and $\hat{\boldsymbol{\rho}}_i$ is all the parts of $\boldsymbol{\rho}_i$ except for the $\boldsymbol{A}_{j,i}$ and the $\boldsymbol{Q}_{j,i}$. As expected, this is the likelihood of a DHGPM with parameters $\boldsymbol{\rho}_i$ and training data $\mathcal{D}_i = \{(x_{t+1}^i, \boldsymbol{x}_t)$

for $t = 1, ..., T-1\}$. We can use this to write the optimisation part of the DHGPSSM training algorithm: see Algorithm 11.

Another key part of the training process is to sample new latent states $\boldsymbol{x}_{1:T}^{new}$ given the observations $\boldsymbol{y}_{1:T}$, the previous latent states $\boldsymbol{x}_{1:T}^{old}$ and the current model parameters $\boldsymbol{\theta}$. We do this using the PGAS sampler of Lindsten et al. [2014, p. 2160] with observations $\boldsymbol{y}_{1:T}$, reference trajectory $\boldsymbol{x}_{1:T}^{old}$, initial samples drawn using a $\mathcal{N}(\boldsymbol{0}, 100\boldsymbol{I})$ distribution and both the transition distribution and observation distribution constructed using the current model parameters $\boldsymbol{\theta}$ with (3.112) and (3.123) respectively. We will now present the training algorithms and discuss some tuning parameters.

List of Algorithms:

1. **Optimisation**: This algorithm maximises (3.124) with respect to $\boldsymbol{\theta}$ using the independence of DHGPMs to split up the optimisation as we have discussed in (3.129).

2. **OptimisationLoop**: This uses Algorithm 11 to quickly get both the latent states $\boldsymbol{x}_{1:T}$ and parameters $\boldsymbol{\theta}$ near to their respective invariant distributions. The idea is that we alternative between PGAS and Algorithm 11 until (3.124) decreases for the first time and then we stop. Since, at this point (3.124) is no longer just strictly increasing so we must be relatively near to some sort of invariant distribution.

3. **BurnerLoop**: In a similar way to the OptimisationLoop Algorithm, the aim is to get to the invariant distribution but here we add sampling of $\boldsymbol{\theta}$ which allows us to burn-in EMCEE while getting even closer to the invariant distribution compared with OptimisationLoop.

4. **Learning of a DHGPSSM**: This combines the above algorithms to go from a set of observations to a learnt DHGPSSM.

5. **Prediction**: Given a learnt DHGPSSM and test observations $\boldsymbol{y}_1^*, ..., \boldsymbol{y}_t^*$, this algorithm samples from $\boldsymbol{y}_{t+k}^*$ for some $k \in \mathbb{Z}_{>0}$.

Tuning Parameters:

1. **Latent state dimension**: Unless we know the dimension of the latent state, we will need to choose it. In general, the dimension of the latent states depends on the complexity of the process which generates the observations as well as the level of dependency of the future of the process on its history. In other words, if we have a complex system or the system depends on observations which happened many time steps ago, then we will need a correspondingly bigger latent state dimension.

2. **Form of the observational distribution**: This might be known depending on the use case, otherwise set (3.123) to:

$$\boldsymbol{y}_t = \boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{b} + \boldsymbol{v}_t \tag{3.132}$$

for suitably sized matrices $\boldsymbol{W}$ and $\boldsymbol{b}$ and a chosen noise distribution $\boldsymbol{v}_t$. The unknown parameters form $\boldsymbol{\theta}_y$.

3. **Number of PGAS particles**: 20 particles is almost always a good choice but lowering it might improve PGAS speed (not by much though).

4. **Tuning parameters for the latent DHGPMs**: See section on tuning parameters for DHGPMs.

Prediction:
Having looked at how to train the model, the only remaining question is how to use a trained model i.e. prediction. The aim of prediction is: given a learnt model and a collection of ordered test observations $\boldsymbol{y}_1^*, ..., \boldsymbol{y}_t^*$, find the predictive distribution of $\boldsymbol{y}_{t+k}^*$ for some $k \in \mathbb{Z}_{>0}$. Unfortunately for DHGPSSMs, it is not possible to find the exact predictive distribution; however, we can construct a procedure for sampling from the predictive distribution. Furthermore, given samples $\boldsymbol{y}_{t+k}^i$ for $i = 1, ..., N$, we can approximate statistics such as the mean and variance of the predictive distribution; for example,

$$\mathbb{E}[\boldsymbol{y}_{t+k}] \approx \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{y}_{t+k}^i \tag{3.133}$$

$$\mathbb{V}[\boldsymbol{y}_{t+k}] \approx \left(\frac{1}{N} \sum_{i=1}^{N} (\boldsymbol{y}_{t+k}^i)^2 \right) - \left(\frac{1}{N} \sum_{i=1}^{N} \boldsymbol{y}_{t+k}^i \right)^2 \tag{3.134}$$

The prediction algorithm for DHGPSSMs is summarised in Algorithm 15.

Algorithm 11: Optimisation

Input : Observations $\boldsymbol{y}_{1:T}$, Latent States $\boldsymbol{x}_{1:T}$, Initial Point $\boldsymbol{\theta}_0$, maxIterations
Output: Optimised Parameters $\boldsymbol{\theta}_*$

```
/* All the following optimisations should be done using L-BFGS-B
   with the maximum iterations equal to maxIterations.  Also start
   the optimisation at the corresponding part of θ_0.            */
```

optimise $\log p(\boldsymbol{y}_{1:T} | \boldsymbol{x}_{1:T}, \boldsymbol{\theta}_y)$ with respect to $\boldsymbol{\theta}_y$ to get $\boldsymbol{\theta}_y^*$;
for $i = 1, ..., n_x$ **do**
  optimise $\log p(\boldsymbol{x}_{1:T} | \boldsymbol{\rho}_i) + \log p(\boldsymbol{\rho}_i)$ with respect to $\boldsymbol{\rho}_i$ to get $\boldsymbol{\rho}_i^*$;
  `/* In other words, create a DHGPM with initial parameters`
  $\boldsymbol{\rho}_{i_0} \subset \boldsymbol{\theta}_0$ `and training data` $\mathcal{D}_i = \{(x_{t+1}^i, \boldsymbol{x}_t)$ `for` $t = 1, ..., T-1\}$
  `then optimise and return` $\boldsymbol{\rho}_i^*$. `*/`
end
return $\boldsymbol{\theta}_* = (\boldsymbol{\rho}_1^*, ..., \boldsymbol{\rho}_{n_x}^*, \boldsymbol{\theta}_y^*)$;

Algorithm 12: OptimisationLoop

Input : Observations $\boldsymbol{y}_{1:T}$, Current Latent States $\boldsymbol{x}_{1:T}$, Current Parameters $\boldsymbol{\theta}$
Output: New Latent States $\boldsymbol{x}_{1:T}^{new}$, New Parameters $\boldsymbol{\theta}_{new}$, #Loops $maxJ$

bestObjective $= \log p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$;
for $j = 1, 2, ...$ **do**
    /* Usually this reaches the break point but it is sensible to set a maximum number of iterations. */
    sample new latent states $\boldsymbol{x}_{1:T}^{new}$ given observations $\boldsymbol{y}_{1:T}$, current latent states $\boldsymbol{x}_{1:T}$ and current parameters $\boldsymbol{\theta}$ via PGAS;
    $\boldsymbol{\theta}_{new}$ = Optimisation(Observations=$\boldsymbol{y}_{1:T}$, Latent States=$\boldsymbol{x}_{1:T}^{new}$, Initial Point =$\boldsymbol{\theta}$, maxIterations=$2j + 5$);
    if $\log p(\boldsymbol{\theta}_{new}|\boldsymbol{x}_{1:T}^{new}, \boldsymbol{y}_{1:T}) <$ *bestObjective* **then**
        $maxJ = j$;
        break;
    end
    else
        bestObjective $= \log p(\boldsymbol{\theta}_{new}|\boldsymbol{x}_{1:T}^{new}, \boldsymbol{y}_{1:T})$;
        $\boldsymbol{x}_{1:T} = \boldsymbol{x}_{1:T}^{new}$;
        $\boldsymbol{\theta} = \boldsymbol{\theta}_{new}$;
    end
end
return $\boldsymbol{x}_{1:T}^{new}$, $\boldsymbol{\theta}_{new}$, $maxJ$.

Algorithm 13: BurnerLoop

Input : Observations $\boldsymbol{y}_{1:T}$, Current Latent States $\boldsymbol{x}_{1:T}$, Current Parameters $\boldsymbol{\theta}$, Base Iterations $maxJ$

Output: New Latent States $\boldsymbol{x}_{1:T}^{new}$, New Parameters $\boldsymbol{\theta}_{new}$

bestObjective $= \log p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$;
for $i = 1, 2, ...$ **do**
    `/* Usually this reaches the break point but it is sensible to set a maximum number of iterations. */`
    sample new latent states $\boldsymbol{x}_{1:T}^{new}$ given observations $\boldsymbol{y}_{1:T}$, current latent states $\boldsymbol{x}_{1:T}$ and current parameters $\boldsymbol{\theta}$ via PGAS;
    if $i \equiv 0 \pmod 5$ *and* $i > 0$ **then**
        $\boldsymbol{\theta}_* =$ Optimisation(Observations=$\boldsymbol{y}_{1:T}$, Latent States=$\boldsymbol{x}_{1:T}^{new}$, Initial Point =$\boldsymbol{\theta}$, maxIterations=$2(maxJ + i) + 5$);
        if $\log p(\boldsymbol{\theta}_*|\boldsymbol{x}_{1:T}^{new}, \boldsymbol{y}_{1:T}) <$ *bestObjective* **then**
            break;
        end
        else
            bestObjective $= \log p(\boldsymbol{\theta}_*|\boldsymbol{x}_{1:T}^{new}, \boldsymbol{y}_{1:T})$;
            $\boldsymbol{x}_{1:T} = \boldsymbol{x}_{1:T}^{new}$;
            $\boldsymbol{\theta} = \boldsymbol{\theta}_*$;
        end
        sample parameters $\boldsymbol{\theta}_{new}$ from $p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$ using EMCEE with initial positions $\boldsymbol{\theta}$ + vector of random noise for each walker and some EMCEE burn-in steps. We recommend using 10 burn-in steps per walker and 16 walkers;
        `/* See previous discussions about adding noise to` $\boldsymbol{\theta}$`; for example, in Algorithm 8. Also, we can use a small amount of walkers for the same reasons as in Algorithm 8. */`
    end
    else
        $\boldsymbol{x}_{1:T} = \boldsymbol{x}_{1:T}^{new}$;
        sample parameters $\boldsymbol{\theta}$ from $p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$ using EMCEE, with initial walker positions equal to the walker positions from the previous loop and 1 burn-in step per walker and 16 walkers;
        `/* For the first loop, set the initial walker positions to` $\boldsymbol{\theta}$ `+ vector of random noise for each walker. */`
    end
end
return $\boldsymbol{x}_{1:T}^{new}$, $\boldsymbol{\theta}_*$

Algorithm 14: Learning of a DHGPSSM

Input : Observations $\boldsymbol{y}_{1:T}$, Number of Samples K
Output: Samples from the parameter posterior $p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T})$ and samples from the latent state distribution $p(\boldsymbol{x}_{1:T}|\boldsymbol{y}_{1:T}, \boldsymbol{\theta})$.

initialise unknown parameters $\boldsymbol{\theta}$ randomly;
`/* Make sure the random initial parameters are valid. */`
`/* Optimisation Stage: */`
initialise latent states $\boldsymbol{x}_{1:T}$ by setting each dimension of $\boldsymbol{x}_{1:T}$ to a random dimension of $\boldsymbol{y}_{1:T}$. Repeat this random allocation of $\boldsymbol{x}_{1:T}$ around 10-100 times and finally set the initial latent states equal to the random allocation that maximised (3.123);
$\boldsymbol{x}_{1:T}$, $\boldsymbol{\theta}$, $maxJ$ = OptimisationLoop(Observations=$\boldsymbol{y}_{1:T}$, Latent States=$\boldsymbol{x}_{1:T}$, Current Parameters=$\boldsymbol{\theta}$);
`/* Burn-in Stage: */`
$\boldsymbol{x}_{1:T}[0]$, $\boldsymbol{\theta}[0]$ = BurnerLoop(Observations=$\boldsymbol{y}_{1:T}$, Latent States=$\boldsymbol{x}_{1:T}$, Current Parameters=$\boldsymbol{\theta}$, Base Iterations = $maxJ$);
`/* Sampling Stage: */`
for $k = 1, ..., K$ **do**
    sample new latent states $\boldsymbol{x}_{1:T}[k]$ given observations $\boldsymbol{y}_{1:T}$, latent states $\boldsymbol{x}_{1:T}[k-1]$ and parameters $\boldsymbol{\theta}[k-1]$ via PGAS;
    if $i \equiv 0 \pmod 5$ **then**
        $\boldsymbol{\theta}_* =$ Optimisation(Observations=$\boldsymbol{y}_{1:T}$, Latent States=$\boldsymbol{x}_{1:T}[k]$, Initial Point =$\boldsymbol{\theta}[k-1]$, maxIterations=25);
        sample parameters $\boldsymbol{\theta}[k]$ from $p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}[k], \boldsymbol{y}_{1:T})$ using EMCEE with initial positions $\boldsymbol{\theta}_*$ + vector of random noise for each walker and some EMCEE burn-in steps. We recommend using 10 burn in steps per walker and 16 walkers;
        `/* See previous discussions about adding noise to` $\boldsymbol{\theta}_*$`; for example, in Algorithm 8. Also, we can use a small amount of walkers for the same reasons as in Algorithm 8. */`
    end
    else
        sample parameters $\boldsymbol{\theta}[k]$ from $p(\boldsymbol{\theta}|\boldsymbol{x}_{1:T}[k], \boldsymbol{y}_{1:T})$ using EMCEE with initial walker positions equal to the walker positions from the previous loop with 1 burn in step per walker and 16 walkers;
    end
end
return $\boldsymbol{x}_{1:T}[1:K]$, $\boldsymbol{\theta}[1:K]$

Algorithm 15: DHGPSSM Prediction

Input : Samples $\boldsymbol{\theta}^1, ..., \boldsymbol{\theta}^K$ from the posterior parameter distribution of a DHGPSSM, test observations $\boldsymbol{y}_1^*, ..., \boldsymbol{y}_t^*$

Output: Samples from $\boldsymbol{y}_{t+k}^*$ for some $k \in \mathbb{Z}_{>0}$

for $i = 1, ..., nSamples$ **do**
  select $\boldsymbol{\theta}^i$ uniformly from $\boldsymbol{\theta}^1, ..., \boldsymbol{\theta}^K$;
  set parameters of DHGPSSM equal to $\boldsymbol{\theta}^i$;
  find the latent states $\boldsymbol{x}_1^*, ..., \boldsymbol{x}_t^*$ corresponding to $\boldsymbol{y}_1^*, ..., \boldsymbol{y}_t^*$ via PGAS;
  for $j = 1, ..., k$ **do**
    sample $\boldsymbol{x}_{t+j}$ from $p(\boldsymbol{x}_{t+j}|\boldsymbol{x}_{t+j-1}, \boldsymbol{\theta}^i)$ (equation (3.112));
  end
  sample $\boldsymbol{y}_{t+k}$ from $p(\boldsymbol{y}_{t+k}|\boldsymbol{x}_{t+k}, \boldsymbol{\theta}_y^i)$ (equation (3.123));
end
return all samples of $\boldsymbol{y}_{t+k}$.

3.4.2 Examples

In this section, we look at several examples of DHGPSSMs and compare them with an autoregressive full GP and the GPSSM by Svensson et al. [2016]. In the examples that look at comparing the latent states, we use the formulas of (3.49), (3.50) and (3.51) to calculate the RMSE, LL (Gaussian) and LL (Student-t) respectively. Due to unidentifiabilty, in models with unknown observation distributions or multi-dimensional latent states, we are unable to compare the latent states of different models; however, we can look at comparing predictions from the model and we will look at two types of predictions:

1. Prediction of fixed k: Given a sequence of test observations $y_1^*, ..., y_{T^*}^*$, we use the models to predict y_{t+k}^* given $y_{1:t}^*$ for $t = 1, ..., T^* - k$. Note that we use a particle filter (bootstrap filter) to get the states at time t rather than a smoother because we wish to compare predictions given only the history of the process. We use the formulas of (3.101), (3.102) and (3.103) to compute the statistics RMSE, LL (Gaussian) and LL (Student-t), respectively, with y_i^* replaced with y_{t+k}^* and $f(x_i^*)$ replaced with the model's prediction for y_{t+k}^*. Note that the assumed model distribution now refers to the assumed observation distribution.

2. Multi-step forecast: Suppose that the model training data is $y_1, ..., y_{T_{train}}$ and the model test data is $y_{T_{train}+1}, ..., y_{T^*}$, then given the training data we predict $y_{T_{train}+k}$ for $k = 1, ..., \min(30, T^*)$ using Algorithm 15. We compare the predictions to the true values using (3.101), (3.102) and (3.103).

Example 1:
Aim: This is a 'sanity-check' example comparing the performance of the DHGPSSM with the model of Svensson et al. [2016] on a simple system. We expect all models to produce similar results. The set-up is the same as in section 3.2.6, example 1 and for clarification, we have the same training and test data as in the original example.

Additional Information:

- The model called GPSSM refers to the model of Svensson et al. [2016] with none of our new features added.
- The model called DHGPSSM is a deep Hilbert Gaussian process state-space model.
- The latent DHGPMs in the DHGPSSM will use $sin(x)$ activation functions.

More information is summarised in table 3.22.

Model	Latent Kernel	Latent Process	Assumed Latent Noise	Observational Distribution	Additional Information
GPSSM	RBF	Gaussian	Gaussian	Fixed	N/A
DHGPSSM	RBF	Gaussian	Gaussian	Fixed	layout = $[12]$

Table 3.22: Additional Information for Example 1

Results:

Model	RMSE (on test data)	LL (on test data)	Mean Time (seconds)
GPSSM	1.13	-1.50	83.36
DHGPSSM	**1.06**	**-1.48**	**45.53**

Table 3.23: Results for Example 1

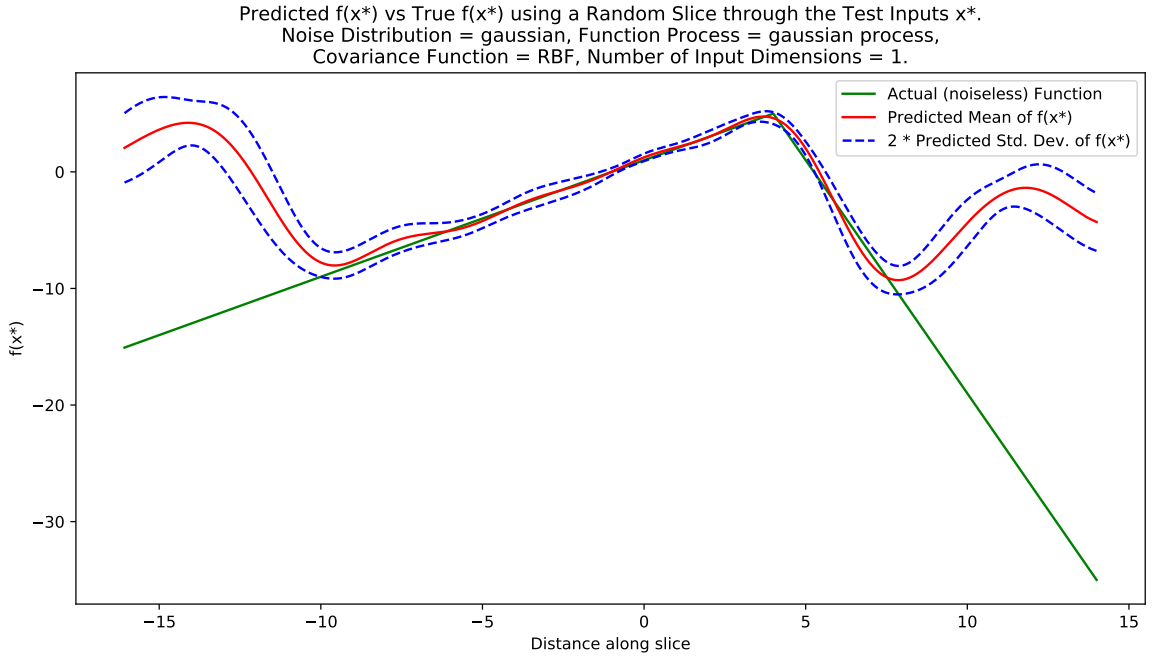

Figure 3.22: A DHGPSSM
(Table 3.22, Row 2)

Example 2:
Aim: This is another 'sanity-check' example comparing the performance of the DHGPSSM with Student-t noise and our Student-t noise version of the model of Svensson et al. [2016]. We expect all models to produce similar results. The set-up is the same as in section 3.2.6, Example 2 and for clarification, we have the same training and test data as in the original example.

Additional Information:

- The model called GPSSM refers to our Student-t noise version of the model of Svensson et al. [2016].
- The model called DHGPSSM is a deep Hilbert Gaussian process state-space model with latent Student-t noise assumed.
- The latent DHGPMs in the DHGPSSM will use $sin(x)$ activation functions.

More information is summarised in table 3.24.

Model	**Latent Kernel**	**Latent Process**	**Assumed Latent Noise**	**Observational Distribution**	**Additional Information**
GPSSM	RBF	Gaussian	Student-t	Fixed	See 3.2.6, Ex. 2
DHGPSSM	RBF	Gaussian	Student-t	Fixed	layout = $[4]$

Table 3.24: Additional Information for Example 2

Results:

Model	LL (on test data)	Mean Time (seconds)
GPSSM	-3.06	175.64
DHGPSSM	**-2.85**	**69.97**

Table 3.25: Results for Example 2

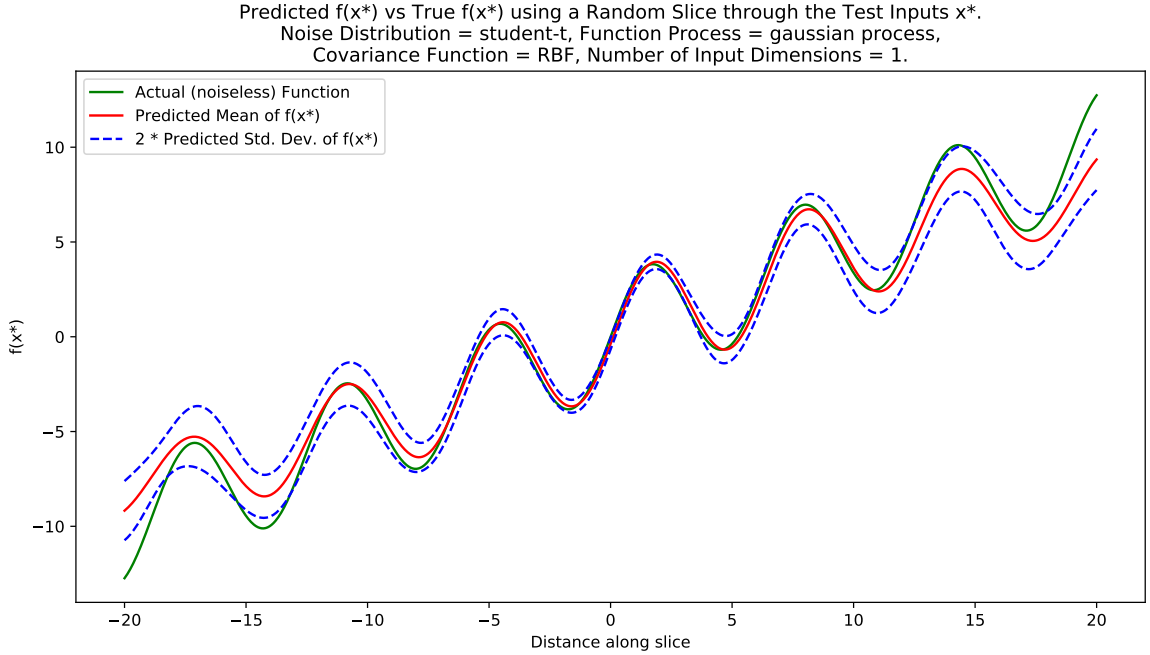

Figure 3.23: A DHGPSSM
(Table 3.24, Row 2)

Example 3:
Aim: To demonstrate that the DHGPSSM can learn with multi-dimensional latent spaces in reasonable time.

Training/Test Data: Given 500 points $x_1, ..., x_{500}$ evenly spread between -50 and 50, we generate the corresponding observations $y_1, ..., y_{500}$ using the system:

$$y = \sin(\pi x) + w_t \tag{3.135}$$
$$w_t \sim \mathcal{N}(0, 0.1^2) \tag{3.136}$$

Then, we discard the x_i. The training observations are $y_1, ..., y_{300}$ and the test observations are $y_{301}, ..., y_{500}$.

Additional Information:

- We test the ability of the models to predict 4 steps ahead.
- The model called full GP is a full Gaussian process and it learns a mapping between y_t and y_{t+4} using training observations $y_1, ..., y_{300}$.
- The model called DHGPSSM is a deep Hilbert Gaussian process state-space model with a three dimensional latent space and layout $= [6, 2]$. We will also have an observation distribution:

$$y_t = \boldsymbol{\alpha}^T \boldsymbol{x}_t + v_t \tag{3.137}$$

$$v_t \overset{iid}{\sim} \mathcal{N}(0, R) \tag{3.138}$$

  where $\boldsymbol{\alpha}^T = (a, b, c)$ and the parameters a, b, c and R are unknown and found during training.
- The latent DHGPMs in the DHGPSSM will use $\tanh(x)$ activation functions (only because using $\sin(x)$ might be seen as an unfair test).

Results:

Model	Kernel	RMSE (on test data)	LL (on test data)	Mean Time (seconds)
Full GP	RBF	0.51	-0.80	**2.93**
DHGPSSM	RBF	**0.14**	**0.40**	174.22

Table 3.26: Results for Example 3

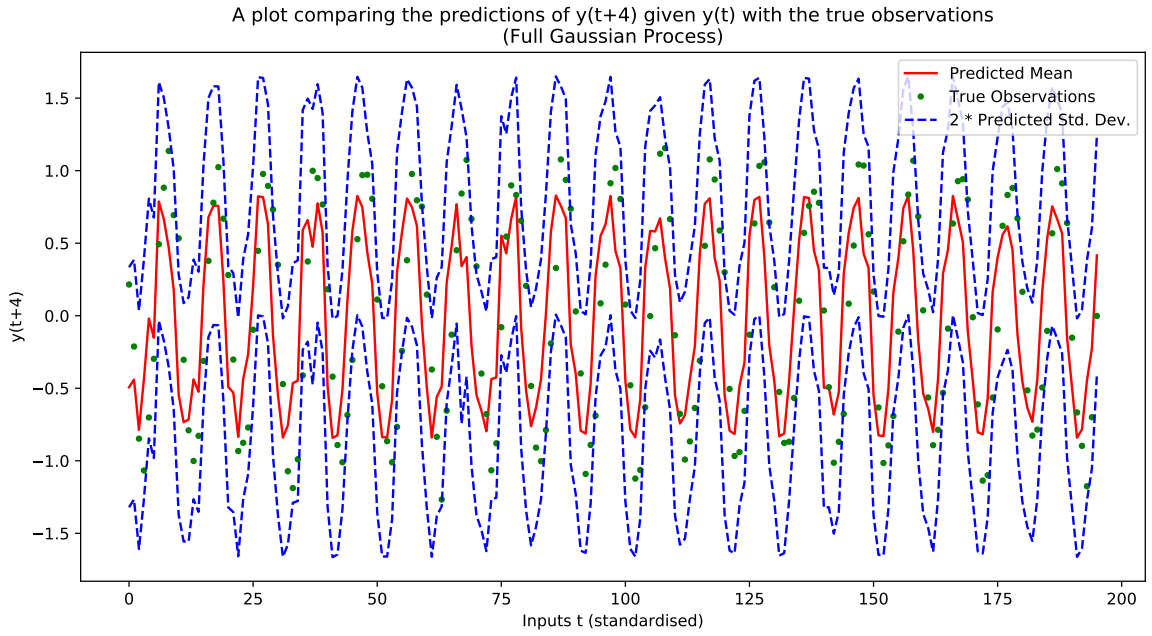

Figure 3.24: Predicting y_{t+4} given y_t using the full GP

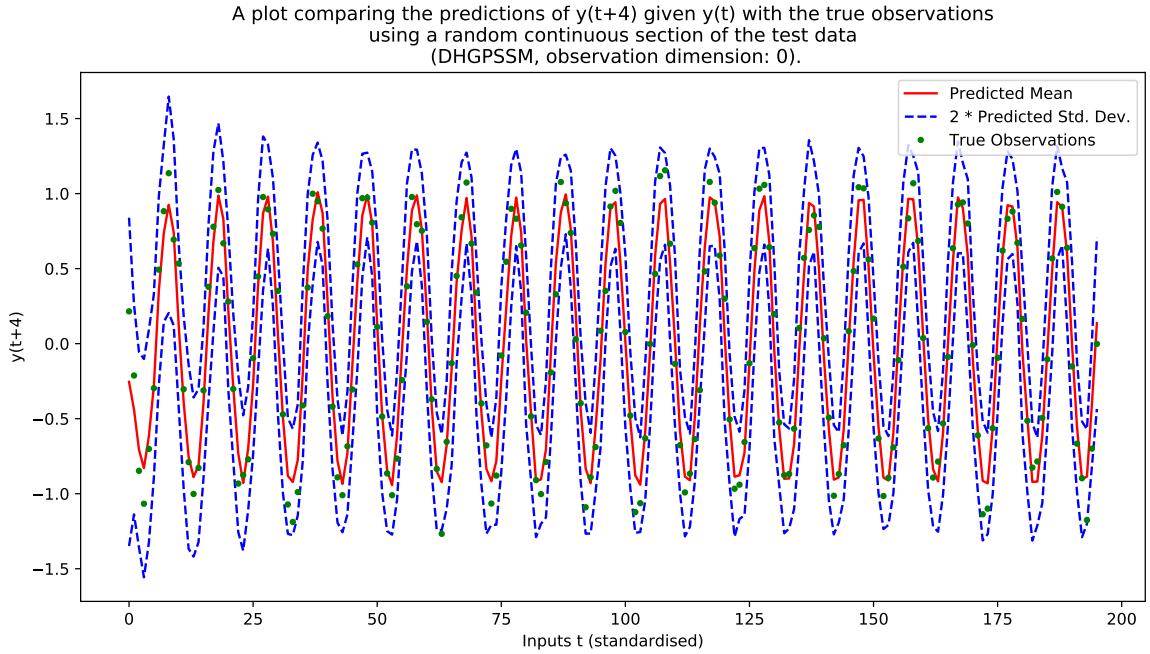

Figure 3.25: Predicting y_{t+4} given y_t using the DHGPSSM. The poor performance with small t is because we use a particle filter and do not have a lot of information which can help to calculate the latent states at this point.

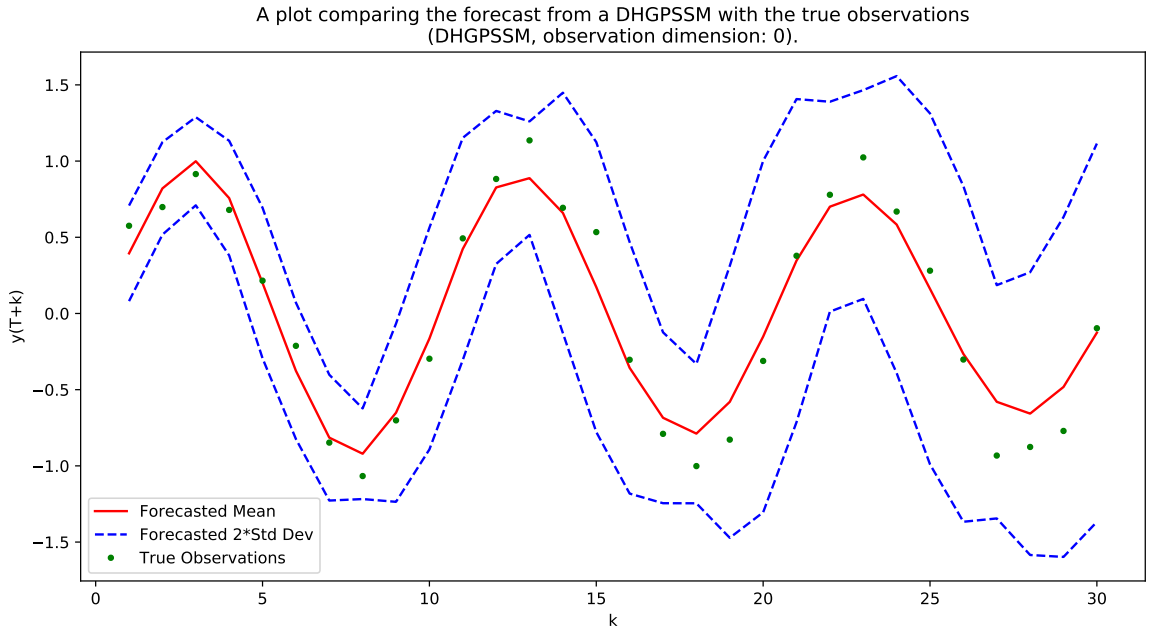

Figure 3.26: Forecasting using a DHGPSSM (RMSE: 0.17, LL = 0.05)

Example 4:
Aim: To demonstrate that the DHGPSSM can learn with multi-dimensional latent spaces and with non-synthetic data.

The dataset and task: Predicting yearly sunspot numbers using the SILSO sunspot dataset[7].

Training Data: The first 200 observations.

Test Data: The remaining 115 observations.

Additional Information:

- We test the ability of the models to predict 2 steps ahead.
- The model called full GP is a full Gaussian process and it learns a mapping between y_t and y_{t+2} using training observations $y_1, ..., y_{200}$.
- The model called DHGPSSM is a deep Hilbert Gaussian process state-space model with a three dimensional latent space and layout = $[2]$. We will also have an observation distribution:

$$y_t = \boldsymbol{\alpha}^T \boldsymbol{x}_t + v_t \tag{3.139}$$
$$v_t \overset{iid}{\sim} \mathcal{N}(0, R) \tag{3.140}$$

  where $\boldsymbol{\alpha}^T = (a, b, c)$ and the parameters a, b, c and R are unknown and found during learning.

- The latent DHGPMs in the DHGPSSM will use $\sin(x)$ activation functions.

Results:

Model	Kernel	RMSE (on test data)	LL (on test data)	Mean Time (seconds)
Full GP	RBF	43.57	-1.77	**1.47**
DHGPSSM	RBF	**34.94**	**-1.68**	96.39

Table 3.27: Results for Example 4

[7]https://datamarket.com/data/set/4apm/yearly-mean-total-sunspot-number

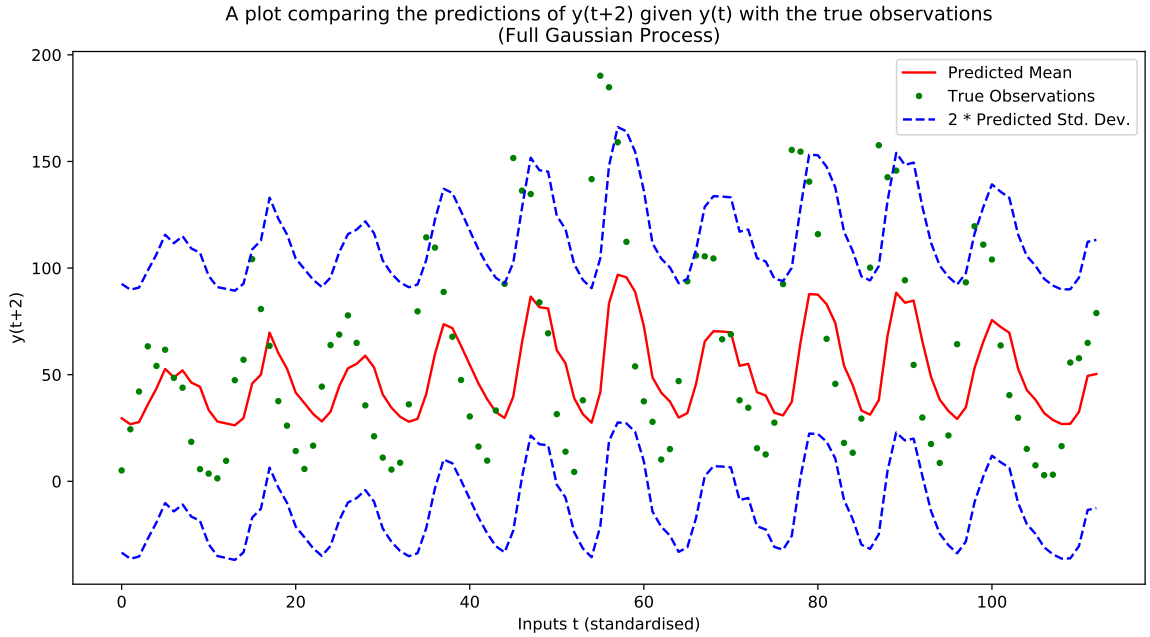

Figure 3.27: Predicting y_{t+2} given y_t using the full GP

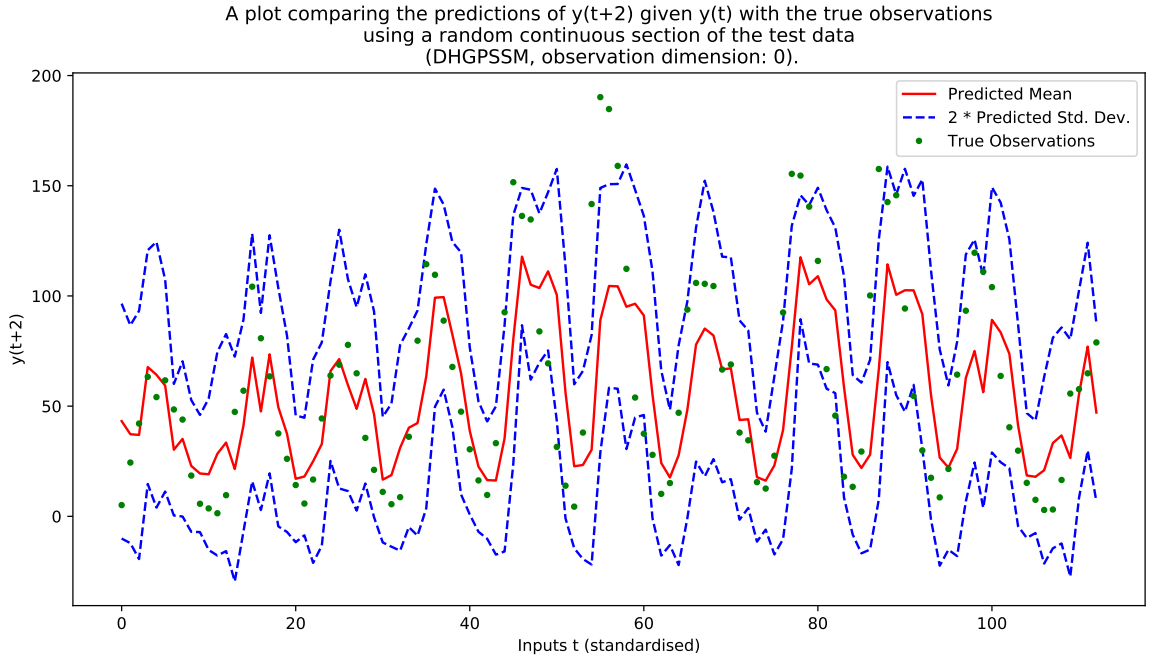

Figure 3.28: Predicting y_{t+2} given y_t using the DHGPSSM.

Chapter 4

Conclusion

4.0.3 Conclusions

In this thesis, we have added features to the GPSSM of Svensson et al. [2016], improved the Hilbert reduced-rank GP of Solin and Särkkä [2014] and created a novel GPSSM called the deep Hilbert Gaussian process state-space model. Here, we look at each contribution in turn and provide some concluding remarks.

1. The *Adaptive* covariance seems to be promising and although there is a computation cost compared with other covariance functions such as the RBF or Matern32 covariance, its performance is good and it seems to be particularly useful for DHGPMs and when combined with the manifold-Hilbert reduced-rank Gaussian process.

2. The manifold-Hilbert reduced-rank GP retains the properties of the original mGP; moreover, it is faster than the full GP even though the original mGP was slower than the full GP. From the examples, we can see that it is particularly good at learning functions with sharp edges. Furthermore, when using a $\sin(x)$ activation, the manifold-Hilbert reduced-rank GP can learn some challenging periodic functions.

3. Our methods for learning with Student-t noise work even under distributions with small degrees of freedom and it is not surprising that it outperforms methods that do not assume a Student-t noise distribution. From the examples, we can see that it is almost always a good choice to use a Student-t noise distribution even when the true noise distribution is Gaussian. This is because it does not incur a significant computational cost (especially in the DHGPM) but with high degrees of freedom, the Student-t distribution tends to a Gaussian distribution: the models can *learn* the noise distribution.

4. Online learning and forgetful online learning both tend to a good solution as the amount of samples and data in the system increases. It is worth pointing out that sometimes these methods perform better than offline learning and we believe this might be because the weighting system can lead to bad samples (which are often the early samples even after burn-in) having a small weight.

Moveover, forgetful online learning, in particular, can 'forget' outliers which would usually have a detrimental effect on the sample quality.

5. Learning with Student-t process state-space models was somewhat disappointing because it often had slightly worse (or similar) performance compared with using Gaussian processes. This is contrary to the results stated in Shah et al. [2014]; however, we believe this might be because EMCEE has a challenging time sampling from models which use Student-t processes.

6. The deep Hilbert Gaussian process model trains much faster than the Hilbert reduced-rank GP when we have high-dimensional inputs. However, for small datasets it often takes longer to train than the full GP but it does seem to consistently produce better results than the full GP. With large datasets, the DHGPM produces good results while also having a training time much less than the full GP. As a result, we only recommend using the DHGPM for datasets larger than 1000 data points.

7. The deep Hilbert Gaussian process state-space model (DHGPSSM) can learn with high-dimensional latent spaces significantly quicker than the model of Svensson et al. [2016]. However, the limiting factor is the speed of latent state samples via PGAS, which can be very with large datasets. That said, it has shown competitive performance compared with the autoregressive full GP.

4.0.4 Future Work

Having studied GPSSMs in depth, we have identified three areas for future work.

1. High-Performance implementation of GPSSMs: We have implemented our models in pure Python; however, rewriting some of the key bottlenecks (optimisation steps, PGAS samples) in a low-level language or using GPUs should give a significant performance benefit.

2. Combine DHGPMs and recurrent neural networks: Instead of using a DHGPM inside a state-space model, it might be worth trying to combine DHGPMs with recurrent neural networks by placing matrix-normal priors on all the weight matrices in a similar way to how we constructed the DHGPMs. It would be interesting to see if this improves the performance of the recurrent neural network, and it could provide predictive distributions rather than point estimates.

3. Understand relationship between machine learning and functional analysis: it seems that there is a link between statistical machine learning and functional analysis. Considering that machine learning is a relatively young topic compared with function analysis, it is worth investigating whether any of the techniques of function analysis can be applied to machine learning problems.

Overall, we believe that GPSSMs could become one of the best methods for identification of time-dependant complex systems; however, scalability needs to be improved.

Bibliography

Álvarez, M. A. and Lawrence, N. D. (2009). Sparse Convolved Gaussian Processes for Multi-output Regression. In *Advances in Neural Information Processing Systems 21*, pages 57–64. Curran Associates, Inc. [Cited on page 11.]

Álvarez, M. A., Luengo, D., Titsias, M. K., and Lawrence, N. D. (2010). Efficient Multioutput Gaussian Processes through Variational Inducing Kernels. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*. [Cited on page 13.]

Álvarez, M. A., Rosasco, L., and Lawrence, N. D. (2012). Kernels for Vector-Valued Functions: a Review. `https://arxiv.org/pdf/1106.6251.pdf`. [Cited on pages 5, 6, 10, and 11.]

Bishop, C. M. (2009). *Pattern Recognition and Machine Learning*. Springer, New York, USA. [Cited on page 23.]

Calandra, R., Peters, J., Rasmussen, C. E., and Deisenroth, M. P. (2016). Manifold Gaussian processes for regression. In *Proceedings of the IEEE International Joint Conference on Neural Networks*. [Cited on pages 3, 39, 41, 42, 43, and 73.]

Chicago Board Options Exchange (2017). VIX Historical Data. `http://www.cboe.com/products/vix-index-volatility/vix-options-and-futures/vix-index/vix-historical-data`. [Cited on page 90.]

Dahlin, J. and Schön, T. (2016). Getting started with particle Metropolis-Hastings for inference in nonlinear dynamical models. *Journal of Statisical Software*. [Cited on page 27.]

Damianou, A., Titsias, M. K., and Lawrence, N. D. (2011). Variational Gaussian process dynamical systems. In *Advances in Neural Information Processing Systems*. [Cited on page 36.]

Damianou, A. C. and Lawrence, N. D. (2013). Deep Gaussian Processes. In *Proceedings of the 16th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 31. [Cited on page 73.]

Deisenroth, M. P. (2010). *Efficient Reinforcement Learning using Gaussian Processes*. Karlsruher Institut für Technologie Scientific Publishing, Straße am Forum 2, D-76131 Karlsruhe. [Cited on pages 21, 22, 23, and 34.]

Deisenroth, M. P., Huber, M. F., and Hanebeck, U. D. (2009). Analytic Moment-based Gaussian process filtering. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 225–232. [Cited on pages 34 and 35.]

Deisenroth, M. P., Huber, M. F., and Hanebeck, U. D. (2012). Robust Filtering and Smoothing with Gaussian Processes. In *IEEE Transactions on Automatic Control*, volume 57(7), pages 1865–1871. [Cited on page 34.]

Deisenroth, M. P. and Mohamed, S. (2016). Expectation propagation in Gaussian process dynamical systems. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2609–2617. [Cited on page 35.]

Doucet, A., Freitas, N., and Gordon, N. (2001). *An Introduction to Sequential Monte Carlo Methods*. Springer, New York, USA. [Cited on pages 20, 21, 24, and 26.]

Doucet, A. and Johansen, A. M. (2012). A Tutorial On Particle Filtering and Smoothing: Fifteen years later. Oxford, UK. [Cited on pages 20, 21, 24, 25, 26, and 27.]

Eleftheriadis, S., Nicholson, T. F. W., Deisenroth, M. P., and Hensman, J. (2017). Identification of Gaussian Process State Space Models. `https://arxiv.org/pdf/1705.10888.pdf`. [Cited on page 36.]

Foreman-Mackey, D., Hogg, D. W., Lang, D., and Goodman, J. (2013). EMCEE: The MCMC Hammer. *PASP*, 125:306–312. [Cited on pages 57 and 58.]

Frigola, R. (2015). Bayesian Time Series Learning with Gaussian Processes. Cambridge, UK. [Cited on pages 3, 12, 28, 35, 36, and 92.]

Frigola, R., Lindsten, F., Schön, T., and Rasmussen, C. E. (2013). Bayesian inference and learning in Gaussian process state-space models with particle MCMC. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3156–3164. [Cited on pages 35 and 36.]

Frigola, R., Lindsten, F., Schön, T., and Rasmussen, C. E. (2014). Variational Gaussian process state-space models. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3680–3688. [Cited on page 36.]

Goodman, J. and Weare, J. (2010). Ensemble samplers with affine invariance. In *Communications in Applied Mathematics and Computational Science*. [Cited on page 57.]

GPy (since 2012). GPy: A Gaussian process framework in python. `http://github.com/SheffieldML/GPy`. [Cited on pages 11 and 17.]

Gupta, A. K. and Nagar, D. K. (2000). *Matrix Variate Distributions*. CRC Press. [Cited on pages 8 and 61.]

Hamilton, J. D. (1994). *Time Series Analysis*. Princeton University Press, Princeton, New Jersey. [Cited on pages 1, 3, 28, and 29.]

Iranmanesh, A., Arashi, M., and Tabatabaey, S. M. M. (2010). On Conditional Applications of Matrix Variate Normal Distributions. In *Iranian Journal of Mathematical Sciences and Informatics*, volume Vol. 5, No. 2, pages 33–44. [Cited on page 61.]

Ko, J. and Fox, D. (2008). GP-BayesFilters: Bayesian Filtering Using Gaussian Process Prediction and Observation Models. In *Proceedings of the 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. [Cited on page 34.]

Lawrence, N. D. (2004). Gaussian Process Latent Variable Models for Visualisation of High Dimensional Data. In *Advances in Neural Information Processing Systems (NIPS)*. [Cited on page 35.]

Lehmann, E. L. and Casella, G. (1998). *Theory of Point Estimation (Second Edition)*. Springer, New York, USA. [Cited on page 28.]

Lichman, M. (2013). UCI machine learning repository. `http://archive.ics.uci.edu/ml`. [Cited on page 90.]

Lindsten, F., Jordan, M. I., and Schön, T. (2014). Particle Gibbs with Ancestor Sampling. In *Journal of Machine Learning Research*, volume 15, pages 2145–2184. [Cited on pages 24, 26, 27, 28, 38, 52, 59, 63, and 96.]

Mackay, D. J. C. (1998). *Introduction to Gaussian Processes*. Springer-Verlag. [Cited on page 42.]

Quiñonero-Candela, J. and Rasmussen, C. E. (2005). A Unifying View of Sparse Approximate Gaussian Process Regression. In *Journal of Machine Learning Research*, volume 6, pages 1939–1959. [Cited on pages 12 and 13.]

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, Massachusetts. [Cited on pages 3, 5, 7, 9, 10, and 13.]

Roth, M. (2013). On the Multivariate-*t* Distribution. `http://users.isy.liu.se/en/rt/roth/student.pdf`. [Cited on page 56.]

Schäcke, K. (2013). On the Kronecker Product. `https://www.math.uwaterloo.ca/~hwolkowi/henry/reports/kronthesisschaecke04.pdf`, Department of Mathematics, University of Waterloo, Canada. [Cited on page 8.]

Shah, A., Wilson, A., and Ghahramani, Z. (2014). Student-t Processes as Alternatives to Gaussian Processes. In *Proceedings of the 17th International Conference on Articial Intelligence and Statistics (AISTATS)*, Reykjavik, Iceland. [Cited on pages 60 and 110.]

Solin, A. and Särkkä (2014). Hilbert Space Methods for Reduced-Rank Gaussian Process Regression. `https://arxiv.org/pdf/1401.5508.pdf`. [Cited on pages 3, 4, 13, 14, 15, 16, 39, 40, 41, 60, 61, 72, 73, 77, 78, 79, 92, and 109.]

Svensson, A., Solin, A., Särkkä, S., and Schön, T. B. (2016). Computationally efficient Bayesian learning of Gaussian process state space models. In *Proceedings of 19th International Conference on Artificial Intelligence and Statistics (AISTATS)*, Cadiz, Spain. [Cited on pages 4, 14, 27, 36, 37, 38, 50, 51, 52, 55, 56, 58, 64, 65, 66, 68, 71, 73, 92, 101, 102, 103, 109, and 110.]

Titsias, M. K. (2009). Variational Learning of Inducing Variables in Sparse Gaussian Processes. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTAT)*. [Cited on page 13.]

Turner, R., Deisenroth, M. P., and Rasmussen, C. E. (2010). State-Space Inference and Learning with Gaussian Processes. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9, pages 868–875. [Cited on pages 12 and 35.]

Wang, J. M., Fleet, D. J., and Hertzmann, A. (2008). Gaussian process dynamical models. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 30, pages 283–298. [Cited on pages 35 and 36.]

Yahoo Finance (2017). S&P 500 Index. `https://finance.yahoo.com/quote/`. [Cited on page 90.]

Zhu, S., Yu, K., and Gong, Y. (2008). Predictive Matrix-Variate *t* Models. In *Advances in Neural Information Processing Systems*. [Cited on page 61.]