Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. doi: 10.18637/jss.v000.i00

GaussianProcesses.jl: A Nonparametric Bayes package for the Julia Language

Jamie Fairbrother
Lancaster University

Christopher Nemeth
Lancaster University

Maxime Rischard
Cervest

Johanni Brea
EPFL

Thomas Pinder
Lancaster University

Abstract

Gaussian processes are a class of flexible nonparametric Bayesian tools that are widely used across the sciences, and in industry, to model complex data sources. Key to applying Gaussian process models is the availability of well-developed open source software, which is available in many programming languages. In this paper, we present a tutorial of the GaussianProcesses.jl package that has been developed for the Julia programming language. GaussianProcesses.jl utilises the inherent computational benefits of the Julia language, including multiple dispatch and just-in-time compilation, to produce a fast, flexible and user-friendly Gaussian processes package. The package provides many mean and kernel functions with supporting inference tools to fit exact Gaussian process models, as well as a range of alternative likelihood functions to handle non-Gaussian data (e.g., binary classification models) and sparse approximations for scalable Gaussian processes. The package makes efficient use of existing Julia packages to provide users with a range of optimization and plotting tools.

Keywords: Gaussian processes, nonparametric Bayesian methods, regression, classification, Julia.

1. Introduction

Gaussian processes (GPs) are a family of stochastic processes which provide a flexible non-parametric tool for modelling data. In the most basic setting, a Gaussian process models a *latent function* based on a finite set of observations. The Gaussian process can be viewed as an extension of a multivariate Gaussian distribution to an infinite number of dimensions, where any finite combination of dimensions will result in a multivariate Gaussian distribution, which is completely specified by its mean and covariance functions. The choice of mean and covariance function, also known as the *kernel*, impose smoothness assumptions on the latent function of interest and determines the correlation between output observations $\mathbf{y}$ as a function of the Euclidean distance between their respective input data points $\mathbf{x}$.

Gaussian processes have been widely used across a vast range of scientific and industrial fields, for example, to model astronomical time series (Foreman-Mackey, Agol, Ambikasaran, and Angus 2017) and brain networks (Wang, Durante, Jung, and Dunson 2017), or for improved soil mapping (Gonzalez, Cook, Oberthur, Jarvis, Bagnell, and Dias 2007) and robotic control (Deisenroth, Fox, and Rasmussen 2015). Arguably, the success of Gaussian processes in these various fields stems from the ease with which scientists and practitioners can apply Gaussian processes to their problems, as well as the general flexibility afforded to GPs for modelling various data forms.

Gaussian processes have a longstanding history in geostatistics (Matheron 1963) for modelling spatial data. However, more recent interest in GPs has stemmed from the machine learning and other scientific communities. In particular, the successful uptake of GPs in other areas has been a result of high-quality and freely available software. There are now a number of excellent Gaussian process packages available in several computing and scientific programming languages. One of the most mature of these is the **GPML** package Rasmussen and Nickisch (2017) for the MATLAB language which was originally developed to demonstrate the algorithms in the book by Rasmussen and Williams (2006) and provides a wide range of functionality. Packages written for other languages, including Python packages, e.g., **GPy** GPy (since 2012) and **GPFlow** Matthews, Van Der Wilk, Nickson, Fujii, Boukouvalas, León-Villagrá, Ghahramani, and Hensman (2017), have incorporated more recent developments in the area of Gaussian processes, most notably implementations of sparse Gaussian processes and graphics processing unit (GPU) accelerations.

This paper presents a new package, **GaussianProcesses.jl**, for implementing Gaussian processes in the recently developed Julia programming language. Julia (Bezanson, Edelman, Karpinski, and Shah 2017), an open source programming language, is designed specifically for numerical computing and has many features which make it attractive for implementing Gaussian processes. Two of the most useful and unique features of Julia are *just-in-time (JIT) compilation* and *multiple dispatch*. JIT compilation compiles a function into binary code the first time it is used, which allows code to run much more efficiently compared with interpreted code in other dynamic programming languages. This provides a solution to the "two-language" problem: in contrast to e.g., R or Python, where performance-critical parts are often delegated to libraries written in C/C++ or Fortran, it is possible to write highly performant code in Julia, while keeping the convenience of a high-level language. Multiple dispatch allows functions to be dynamically dispatched based on inputted arguments. In the context of our package, this allows us to have a general framework for operating on Gaussian processes, while allowing us to implement more efficient functions for the different types of objects which will be used with the process. Similar to the R language, Julia has an excellent package manager system which allows users to easily install packages from inside the Julia

REPL as well as many well-developed packages for statistical analysis.

GaussianProcesses.jl is an open source package which is entirely written in Julia. It supports a wide choice of mean, kernel and likelihood functions (see Appendix A) with a convenient interface for composing existing functions via summation or multiplication. The package leverages other Julia packages to extend its functionality and ensure computational efficiency. For example, hyperparameters of the Gaussian process are optimised using the **Optim.jl** package (Mogensen and Riseth 2018) which provides a range of efficient and configurable unconstrained optimization algorithms; prior distributions for hyperparameters can be set using the **Distributions.jl** package (Besançon, Anthoff, Arslan, Byrne, Lin, Papamarkou, and Pearson 2019). Additionally, this package has now become a dependency of other Julia packages, for example, **BayesianOptimization.jl**, a demo of which is given in Section 4.4. The run-time speed of **GaussianProcesses.jl** has been heavily optimised and is competitive with other Gaussian process packages. A run-time comparison of the package against **GPML** and **GPy** is given in Section 5.

Within the Julia language, Gaussian process software is currently limited. Aside from **GaussianProcesses.jl**, two other packages currently under development are **Stheno.jl** and **AugmentedGaussianProcesses.jl**. The **Stheno.jl** package is designed to provide ease of compatibility with other Julia packages to allow users to leverage the benefits of Bayesian inference (**AdvancedHMC.jl**), optimization (**Optim.jl**) and automatic differentiation (**Zygote.jl**) when fitting Gaussian process models. **Stheno.jl** is designed for Gaussian process regression and does not currently provide features to handle non-Gaussian likelihoods, e.g., classification data. The **AugmentedGaussianProcesses.jl** package is developed primarily for data-augmented sparse Gaussian processes. The data-augmentation structure of this package allows users to utilize Gaussian and non-Gaussian likelihoods by transforming the likelihood functions into conditionally conjugate likelihoods, which allows for faster inference via block coordinate updates. **AugmentedGaussianProcesses.jl** is geared towards variational inference instead of Markov chain Monte Carlo-based inference that is primarily used in **GaussianProcesses.jl**.

The paper is organised as follows. Section 2 provides an introduction to Gaussian processes, how they can be applied to model Gaussian and non-Gaussian observational data, and how enhanced computational efficiency can be achieved through sparse approximations. Section 3 gives an overview of the main functionality of the package which is presented through a simple application of fitting a Gaussian process to simulated data. This is then followed by five application demos in Section 4 which highlight how Gaussian processes can be applied to classification problems, time series modelling, count data, black-box optimization and computationally-efficient large-scale nonparametric modelling via sparse Gaussian process approximations. Section 5 gives a run-time comparison of the package against popular alternatives which are listed above. Finally, the paper concludes (Section 6) with a discussion of ongoing package developments which will provide further functionality in future releases of the package.

2. Gaussian processes in a nutshell

Gaussian processes are a class of models which are popular tools for nonlinear regression and classification problems. They have been applied extensively in scientific disciplines ranging from modelling environmental sensor data (Osborne, Roberts, Rogers, and Jennings 2008) to astronomical time series data (Wilson, Dann, and Nickisch 2015) all within a Bayesian nonparametric setting. A *Gaussian Process* (GP) defines a distribution over functions, $p(f)$, where $f : \mathcal{X} \to \mathbb{R}$ is a function mapping from the input space $\mathcal{X}$ to the space of real numbers. The space of functions f can be infinite-dimensional, for example when $\mathcal{X} \subseteq \mathbb{R}^d$, but for any subset of inputs $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\} \subset \mathcal{X}$ we define $\mathbf{f} := \{f(\mathbf{x}_i)\}_{i=1}^n$ as a random variable whose marginal distribution $p(\mathbf{f})$ is a multivariate Gaussian.

The Gaussian process framework provides a flexible structure for modelling a wide range of data types. In this package we consider models of the following general form,

$$\begin{aligned} \mathbf{y} \,|\, \mathbf{f}, \boldsymbol{\theta} &\sim \prod_{i=1}^{n} p(y_i \mid f_i, \boldsymbol{\theta}), \\ f(\mathbf{x}) \,|\, \boldsymbol{\theta} &\sim \mathcal{GP}\left(m_{\boldsymbol{\theta}}(\mathbf{x}), k_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{x}')\right), \\ \boldsymbol{\theta} &\sim p(\boldsymbol{\theta}), \end{aligned} \tag{1}$$

where $\mathbf{y} = (y_1, y_2, \dots, y_n) \in \mathcal{Y}$ and $\mathbf{x} \in \mathcal{X}$ are the observations and covariates, respectively, and $f_i := f(\mathbf{x}_i)$. We assume that the responses $\mathbf{y}$ are independent and identically distributed, and as a result, the likelihood $p(\mathbf{y} \,|\, \mathbf{f}, \boldsymbol{\theta})$ can be factorised over the observations. For the sake of notational convenience, we let $\boldsymbol{\theta} \in \boldsymbol{\Theta} \subseteq \mathbb{R}^d$ denote the vector of model parameters for both the likelihood function and Gaussian process prior.

The Gaussian process prior is completely specified by its mean function $m_{\boldsymbol{\theta}}(\mathbf{x})$ and covariance function $k_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{x}')$, also known as the *kernel*. The mean function is commonly set to zero (i.e., $m_{\boldsymbol{\theta}}(\mathbf{x}) = 0, \forall \mathbf{x}$), which can often by achieved by centring the observations (i.e., $\mathbf{y} - \mathbb{E}[\mathbf{y}]$) resulting in a mean of zero. If the observations cannot be re-centred in this way, for example if the observations display a linear or periodic trend, then the zero mean function can still be applied with the trend modelled by the kernel function.

The kernel determines the correlation between any two function values f_i and f_j in the output space as a function of their corresponding inputs $\mathbf{x}_i$ and $\mathbf{x}_j$. The user is free to choose any appropriate kernel that best models the data as long as the covariance matrix formed by the kernel is symmetric and positive semi-definite. Perhaps the most common kernel function is the *squared exponential*, $\mathrm{Cov}[f(\mathbf{x}), f(\mathbf{x}')] = k(\mathbf{x}, \mathbf{x}') = \sigma^2 \exp(-\frac{1}{2\ell^2}|\mathbf{x} - \mathbf{x}'|^2)$. For this kernel the correlation between f_i and f_j is determined by the Euclidean distance between $\mathbf{x}_i$ and $\mathbf{x}_j$ and the hyperparameters $\boldsymbol{\theta} = (\sigma, \ell)$, where ℓ determines the speed at which the correlation between $\mathbf{x}$ and $\mathbf{x}'$ decays. There exists a wide range of kernels that can flexibly model a wide range of data patterns. It is possible to create more complex kernels from the sum and product of simpler kernels (Duvenaud 2014), (see Chapter 4 of Rasmussen and Williams (2006) for a detailed discussion of kernels). Figure 2 shows one-dimensional Gaussian processes sampled from three simple kernels (squared exponential, periodic and linear) and three composite kernels, and demonstrates how the combination of these kernels can provide a richer correlation structure to capture more intricate function behaviour.

Often we are interested in predicting function vales $\mathbf{f}^*$ at new inputs $\mathbf{x}^*$. Assuming a finite set of function values $\mathbf{f}$, the joint distribution between these observed points and the test points $\mathbf{f}^*$ forms a joint Gaussian distribution,

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}^* \end{pmatrix} \,|\, \mathbf{X}, \mathbf{x}^*, \boldsymbol{\theta} \sim \mathcal{N}\left(0, \begin{bmatrix} \mathbf{K}_{\mathbf{f},\mathbf{f}} & \mathbf{K}_{\mathbf{f},*} \\ \mathbf{K}_{*,\mathbf{f}} & \mathbf{K}_{*,*}) \end{bmatrix}\right), \tag{2}$$

where $\mathbf{K}_{\mathbf{f},\mathbf{f}} = k_{\boldsymbol{\theta}}(\mathbf{X}, \mathbf{X})$, $\mathbf{K}_{\mathbf{f},*} = k_{\boldsymbol{\theta}}(\mathbf{X}, \mathbf{x}^*)$ and $\mathbf{K}_{*,*} = k_{\boldsymbol{\theta}}(\mathbf{x}^*, \mathbf{x}^*)$.

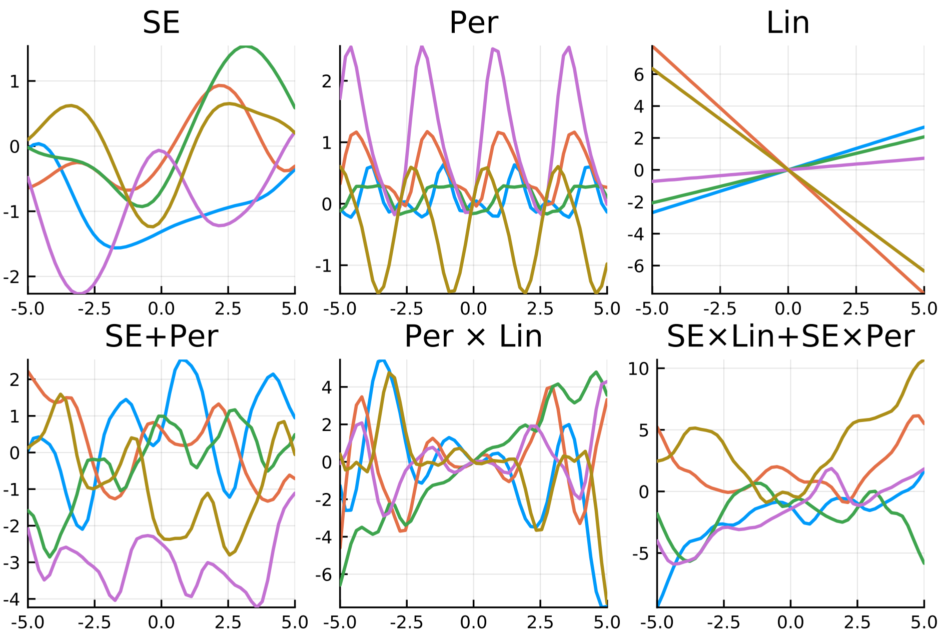


Figure 1: Clockwise from the top left. Five random samples from the Gaussian process prior using the following kernels (refer to the help file for details): `SE(0.5,0.0)`, `Periodic(0.5,0.0,1.0)`, `Lin(0.0)`, `SE(0.5,0.0) * Lin(0.0) + SE(0.5,0.0) * Periodic(0.5,0.0,1.0)`, `Periodic(0.5,0.0,1.0) * Lin(0.0)` and `SE(0.5,0.0) + Periodic(0.5,0.0,1.0)`

By the properties of the multivariate Gaussian distribution, the conditional distribution of $\mathbf{f}^*$ given $\mathbf{f}$ is also a Gaussian distribution for fixed $\mathbf{X}$ and $\mathbf{x}^*$. Extending to the general case, the conditional distribution for the latent function $\mathbf{f}(\mathbf{x}^*)$ is a Gaussian process

$$\mathbf{f}(\mathbf{x}^*) \,|\, \mathbf{f}, \boldsymbol{\theta} \sim \mathcal{GP}(k_{\boldsymbol{\theta}}(\mathbf{x}^*, \mathbf{X})\mathbf{K}_{\mathbf{f},\mathbf{f}}^{-1}\mathbf{f}, k_{\boldsymbol{\theta}}(\mathbf{x}^*, \mathbf{x}^*) - k_{\boldsymbol{\theta}}(\mathbf{x}^*, \mathbf{X})\mathbf{K}_{\mathbf{f},\mathbf{f}}^{-1}k_{\boldsymbol{\theta}}(\mathbf{X}, \mathbf{x}^*)). \tag{3}$$

Using the modelling framework in eq. (1), we have a Gaussian process prior $p(f \,|\, \boldsymbol{\theta})$ over the function $f(\mathbf{x})$. If we let $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ represent our observed data, where $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, then the likelihood of the data, conditional on function values $\mathbf{f}$, is $p(\mathcal{D} \,|\, \mathbf{f}, \boldsymbol{\theta})$. Using Bayes theorem, we can show that the posterior distribution for the function $\mathbf{f}$ is $p(\mathbf{f} \,|\, \mathcal{D}, \boldsymbol{\theta}) \propto p(\mathcal{D} \,|\, \mathbf{f}, \boldsymbol{\theta})p(\mathbf{f} \,|\, \boldsymbol{\theta})$. In the general setting, the posterior is non-Gaussian (see Section 2.1 for an exception) and cannot be expressed in an analytic form, but can often be approximated using a Laplace approximation (Williams and Barber 1998b), expectation-propagation (Minka 2001), or variational inference (Opper and Archambeau 2009) (see Nickisch and Rasmussen (2008) for a full review). Alternatively, simulation-based inference methods including Markov chain Monte Carlo (MCMC) algorithms (Robert 2004) can be applied.

From the posterior distribution, we can derive the marginal predictive distribution of y^*, given test points $\mathbf{x}^*$, by integrating out the latent function,

$$p(y^* \,|\, \mathbf{x}^*, \mathcal{D}, \boldsymbol{\theta}) = \int\int p(y^* \,|\, f^*, \boldsymbol{\theta})p(f^* \,|\, \mathbf{f}, \mathbf{x}^*, \mathbf{X}, \boldsymbol{\theta})p(\mathbf{f} \,|\, \mathcal{D}, \boldsymbol{\theta})df^* d\mathbf{f}. \tag{4}$$

Calculating this integral is generally intractable, with the exception of nonlinear regression with Gaussian observations (see Section 2.1). In settings such as seen with classification models, the marginal predictive distribution is intractable, but can be approximated using the methods mentioned above. In Section 2.2 we will introduce a MCMC algorithm for sampling exactly from the posterior distribution and use these samples to evaluate the marginal predictive distribution through Monte Carlo integration.

2.1. Nonparametric regression: the analytic case

We start by considering a special case of eq. (1), where the observations follow a Gaussian distribution,

$$y_i \sim \mathcal{N}(f(\mathbf{x}_i), \sigma^2),\ \ i = 1, \dots, n. \tag{5}$$

In this instance, the posterior for the latent variables, conditional on the data, can be derived analytically as a Gaussian distribution (see Rasmussen and Williams (2006)),

$$\mathbf{f} \,|\, \mathcal{D}, \boldsymbol{\theta} \sim \mathcal{N}(\mathbf{K}_{\mathbf{f},\mathbf{f}}(\mathbf{K}_{\mathbf{f},\mathbf{f}} + \sigma_n^2\mathbf{I})^{-1}\mathbf{y}, \mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{K}_{\mathbf{f},\mathbf{f}}(\mathbf{K}_{\mathbf{f},\mathbf{f}} + \sigma_n^2\mathbf{I})^{-1}\mathbf{K}_{\mathbf{f},\mathbf{f}}). \tag{6}$$

The predictive distribution for y^* in eq. (4) can also be calculated analytically by noting that the likelihood in eq. (5), the posterior in eq. (6) and the conditional distribution for f^* in eq. (3) are all Gaussian and integration over a product of Gaussians produces a Gaussian distribution,

$$y^* \,|\, \mathbf{x}^*, \mathcal{D}, \boldsymbol{\theta}, \sigma^2 \sim \mathcal{N}(\mu(\mathbf{x}^*), \Sigma(\mathbf{x}^*, \mathbf{x}^{*'}) + \sigma^2\mathbf{I}), \tag{7}$$

where

$$\mu(\mathbf{x}^*) = k(\mathbf{x}^*, \mathbf{X})(\mathbf{K}_{\mathbf{f},\mathbf{f}} + \sigma_n^2\mathbf{I})^{-1}\mathbf{y}$$

and

$$\Sigma(\mathbf{x}^*, \mathbf{x}^{*'}) = k(\mathbf{x}^*, \mathbf{x}^*) - k(\mathbf{x}^*, \mathbf{X})(\mathbf{K}_{\mathbf{f},\mathbf{f}} + \sigma_n^2\mathbf{I})^{-1}k(\mathbf{X}, \mathbf{x}^*)$$

(see Chapter 2 of Rasmussen and Williams (2006) for the full derivation).

The quality of the Gaussian process fit to the data is dependent on the model hyperparameters, $\boldsymbol{\theta}$, which are present in the mean and kernel functions as well as the observation noise σ^2. Estimating these parameters requires the marginal likelihood of the data,

$$p(\mathcal{D} \,|\, \boldsymbol{\theta}, \sigma) = \int p(\mathbf{y} \,|\, \mathbf{f}, \sigma^2) p(\mathbf{f} \,|\, \mathbf{X}, \boldsymbol{\theta}) d\mathbf{f},$$

which is given by marginalising over the latent function values $\mathbf{f}$. Assuming a Gaussian observation model in eq. (5), the marginal distribution is $p(\mathbf{y} \,|\, \mathbf{X}, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(0, \mathbf{K}_{\mathbf{f},\mathbf{f}} + \sigma^2\mathbf{I})$. For convenience of optimisation we work with the log-marginal likelihood

$$\log p(\mathcal{D} \,|\, \boldsymbol{\theta}, \sigma) = -\frac{1}{2}\mathbf{y}^\top(\mathbf{K}_{\mathbf{f},\mathbf{f}} + \sigma^2\mathbf{I})^{-1}\mathbf{y} - \frac{1}{2}\log|\mathbf{K}_{\mathbf{f},\mathbf{f}} + \sigma^2\mathbf{I}| - \frac{n}{2}\log 2\pi. \tag{8}$$

The tractablility of the log-marginal likelihood allows for the straightforward calculation of the gradient with respect to the hyperparameters. Efficient gradient-based optimisation techniques (e.g., L-BFGS and conjugate gradients) can be applied to numerically maximise the log-marginal likelihood function. In practice, we utilise the excellent **Optim.jl** package (Mogensen and Riseth 2018) and provide an interface for the user to specify their choice of optimisation algorithm. Alternatively, a Bayesian approach can be taken, where samples are drawn from the posterior of the hyperparameters using the in-built MCMC algorithm, see Section 3 for an example.

2.2. Gaussian processes with non-Gaussian data

In Section 2.1 we considered the simple tractable case of nonlinear regression with Gaussian observations. The modelling framework given in eq. (1) is general enough to extend the Gaussian process model to a wide range of data types. For example, Gaussian processes can be applied to binary classification problems (see Rasmussen and Williams (2006) Chapter 3), by using a Bernoulli likelihood function (see Section 4.1 for more details).

When the likelihood $p(\mathbf{y} \,|\, \mathbf{f}, \boldsymbol{\theta})$ is non-Gaussian, the posterior distribution of the latent function, conditional on observed data $p(\mathbf{f} \,|\, \mathcal{D}, \boldsymbol{\theta})$, does not have a closed form solution. A popular approach for addressing this problem is to replace the posterior with an analytic approximation, such as a Gaussian distribution derived from a Laplace approximation (Williams and Barber 1998b) or an expectation-propagation algorithm (Minka 2001). These approximations are simple to employ and can work well in practice on specific problems (Nickisch and Rasmussen 2008), however, in general these methods struggle if the posterior is significantly non-Gaussian. Alternatively, rather than trying to find a tractable approximation to the posterior, one could sample from it and use the samples as a stochastic approximation and evaluate integrals of interest through Monte Carlo integration (Ripley 2009).

Markov chain Monte Carlo methods (Robert 2004) represent a general class of algorithms for sampling from high-dimensional posterior distributions. They have favourable theoretical support to guarantee algorithmic convergence (Roberts and Rosenthal 2004) and are generally easy to implement only requiring that it is possible to evaluate the posterior density pointwise. We use the centred parameterization as given in Murray and Adams (2010); Filippone, Zhong, and Girolami (2013); Hensman, Matthews, Filippone, and Ghahramani (2015), which has been shown to improve the accuracy of MCMC algorithms by de-coupling the strong dependence between $\mathbf{f}$ and $\boldsymbol{\theta}$. Re-parameterising eq. (1) we have,

$$\begin{aligned} \mathbf{y} \,|\, \mathbf{f}, \boldsymbol{\theta} \quad &\sim \quad \prod_{i=1}^{n} p(y_i \mid f_i, \boldsymbol{\theta}), \\ \mathbf{f} = L_{\boldsymbol{\theta}} \mathbf{v}, \qquad & \qquad L_{\boldsymbol{\theta}} L_{\boldsymbol{\theta}}^\top = K_{\boldsymbol{\theta}}, \\ \mathbf{v} \sim \mathcal{N}\left(\mathbf{0}_n, \mathbf{I}_n\right), \qquad & \qquad \boldsymbol{\theta} \sim p(\boldsymbol{\theta}), \end{aligned} \tag{9}$$

where $L_{\boldsymbol{\theta}}$ is the lower Cholesky decomposition of the covariance matrix $K_{\boldsymbol{\theta}}$, with (i,j)-element $K_{i,j} = k_{\boldsymbol{\theta}}(x_i, x_j)$. The random variables $\mathbf{v}$ are now independent under the prior and a deterministic transformation gives the function values $\mathbf{f}$. The posterior distribution for $p(\mathbf{f} \,|\, \mathcal{D}, \boldsymbol{\theta})$, or in the transformed setting, $p(\mathbf{v} \,|\, \mathcal{D}, \boldsymbol{\theta})$ usually does not have a closed form expression. Using MCMC we can instead sample from this distribution, or in the case of unknown model parameters $\boldsymbol{\theta}$, we can sample from

$$p(\boldsymbol{\theta}, \mathbf{v} \,|\, \mathcal{D}) \propto p(\mathcal{D} \,|\, \mathbf{v}, \boldsymbol{\theta}) p(\mathbf{v}) p(\boldsymbol{\theta}). \tag{10}$$

Numerous MCMC algorithms have been proposed to sample from the Gaussian process posterior (see Titsias, Lawrence, and Rattray (2008) for a review). In this package we use the highly efficient Hamiltonian Monte Carlo (HMC) algorithm (Neal 2010), which utilises gradient information to efficiently sample from the posterior. Under the re-parametrised model eq. (9), calculating the gradient of the posterior requires the derivative of the Cholesky factor $L_{\boldsymbol{\theta}}$. We calculate this derivative using the blocked algorithm of Murray (2016).

After running the MCMC algorithm we have N samples $\{\mathbf{v}^{(j)}, \boldsymbol{\theta}^{(j)}\}_{j=1}^{N}$ from the posterior $p(\boldsymbol{\theta}, \mathbf{v} \,|\, \mathcal{D})$. Function values $\mathbf{f}$ are given by the deterministic transform of the Monte Carlo samples, $\mathbf{f}^{(i)} = L_{\boldsymbol{\theta}^{(i)}}\mathbf{v}^{(i)}$. Monte Carlo integration is then used to estimate for the marginal predictive distribution from eq. (4),

$$\hat{p}(y^* \,|\, \mathbf{x}^*, \mathcal{D}, \boldsymbol{\theta}) \simeq \frac{1}{N}\sum_{i=1}^{N}\int p(y^* \,|\, f^*, \boldsymbol{\theta}^{(i)})p(f^* \,|\, \mathbf{f}^{(i)}, \mathbf{x}^*, \mathbf{X}, \boldsymbol{\theta}^{(i)})df^*, \tag{11}$$

where we have a one-dimensional integral for f^* that can be efficiently evaluated using Gauss-Hermite quadrature (Liu and Pierce 1994).

In contrast to MCMC, a variational approach can be employed to perform inference in a non-conjugate Gaussian process. Unlike MCMC which seeks to approximate the process using sampling, variational inference (VI) uses optimisation techniques to minimise a discrepancy term between an approximating distribution and the true posterior of the process (Opper and Archambeau 2009). Many schemes exist to perform VI, however, under the general framework, one seeks to find an optimal distribution $q^\star$ from a family of probability distributions $\mathcal{Q}$, that are parameterised by a set of variational parameters λ. To find $q^\star$, the Kullback-Léibler (KL) divergence between the true posterior and the approximating distribution is minimised such that

$$q^\star(\mathbf{f} \,|\, \lambda) = \underset{q(\mathbf{f} \,|\, \lambda)\in\mathcal{Q}}{\arg\min}\, \mathrm{KL}\left(q(\mathbf{f} \,|\, \lambda)||p(\mathbf{f} \,|\, \mathcal{D}, \theta)\right). \tag{12}$$

It is common to let $q \sim \mathcal{N}(\mathbf{m}, \mathbf{V})$, where $\mathbf{m}$ is a mean vector and $\mathbf{V}$ a covariance matrix. Consequently, the variational parameters to be estimated through VI are $\lambda = (\mathbf{m}, \mathbf{V})$.

Optimising (12) in closed form is not possible due to the need to evaluate the intractable marginal likelihood when computing $p(\mathbf{f} \,|\, \mathcal{D}, \theta)$. To circumvent this, the objective function can be reformulated in terms of an evidence lower bound (ELBO):

$$\mathrm{ELBO}(q) = \sum_{n=1}^{N}\mathbb{E}_{q(\mathbf{f}_n \,|\, \lambda_n)}\left[\log p(y_n \,|\, f_n)\right] - \mathrm{KL}\left(q(\mathbf{f}|\lambda)||p(\mathbf{f}|\theta)\right). \tag{13}$$

While the KL-divergence term is now tractable as it only involves the evaluation of the Gaussian process' prior distribution, the newly required need to compute the expectation of $\log p(y_n \,|\, f_n)$ is an intractable sum for all n. Using the variational objective presented in Khan, Mohamed, and Murphy (2012), a tractable objective function can be expressed as

$$\mathcal{L}_J(\lambda) := \frac{1}{2}\Big[\log|\mathbf{V}\boldsymbol{\Omega}| - \mathrm{tr}(\mathbf{V}\boldsymbol{\Omega}) - (\mathbf{m}-\boldsymbol{\mu})^T\boldsymbol{\Omega}(\mathbf{m}-\boldsymbol{\mu}) + N\Big] + \sum_{n=1}^{N} g\left(y_n, m_n, V_{nn}\right), \tag{14}$$

where $\boldsymbol{\Omega} = \mathbf{K}_{\mathbf{f},\mathbf{f}}^{-1}$, N is the number of observations and $\boldsymbol{\mu}$ is the Gaussian process prior mean. The function $g(\cdot)$ is used to denote the likelihood-specific, local variational bound function that is defined such that $\mathbb{E}\left[\log p\left(y_n|f_n\right)\right] \geq g\left(y_n, m_n, V_{nn}\right)$. An extensive list of local variational bounds can be found in Khan *et al.* (2012), however, an example for Poisson data is $g\left(y_n, m_n, V_{nn}\right) = ym - \exp(m + v/2) - \log y!$, where v denotes a single term from the diagonal of $\boldsymbol{V}$. Upon convergence of the optimiser, $\mathbf{K}_{\mathbf{f},\mathbf{f}}$ is approximated using $\boldsymbol{V}$.

Using the **Optim.jl** package, we can maximise (14) with respect to $\boldsymbol{m}$ and $\boldsymbol{V}$ respectively to find $q^\star$. In order to perform optimisation, gradients of the objective function's lower variational bound must be taken. To enable variational inference to be easily extended to new likelihoods, the computation of derivatives is handled using the automatic differentiation functionality in **Zygote.jl**.

2.3. Scaling Gaussian processes with sparse approximations

When applying Gaussian processes to a dataset of size n, an unfortunate by-product is the $\mathcal{O}(n^3)$ scalability of the Gaussian process. This is due to the need to invert and compute the determinant of the $n \times n$ kernel matrix $\mathbf{K}_{\mathbf{f},\mathbf{f}}$. There exist a number of approaches to deriving more scalable Gaussian processes: sparsity inducing kernels (Melkumyan and Ramos 2009), Nyström-based eigendecompositions (Williams and Seeger 2001), variational posterior approximations (Titsias 2009), neighbourhood partitioning schemes (Datta, Banerjee, Finley, and Gelfand 2016), and divide-and-conquer strategies (Guhaniyogi, Li, Savitsky, and Srivastava 2017). In this package, scalability within the Gaussian process model is achieved by approximating the Gaussian process' prior with a subset of inducing points $\mathbf{u}$ of size m, such that $m << n$ (Quiñonero-Candela and Rasmussen 2005).

Due to the consistency of a Gaussian process[1] the joint prior in eq. (2) can be recovered from a sparse Gaussian process through the marginalisation of $\mathbf{u}$

$$p(\mathbf{f}, \mathbf{f}^*) = \int p(\mathbf{f}, \mathbf{f}^*, \mathbf{u}) d\mathbf{u} = \int p(\mathbf{f}, \mathbf{f}^* \,|\, \mathbf{u}) p(\mathbf{u}) d\mathbf{u},$$

where $\mathbf{u} \sim \mathcal{N}(0, \mathbf{K}_{\mathbf{u},\mathbf{u}})$ and $\mathbf{K}_{\mathbf{u},\mathbf{u}} = k_\theta(\mathbf{u}, \mathbf{u})$ is an $m \times m$ covariance matrix. An approximation is only induced under the sparse framework through the assumption that $\mathbf{f}$ and $\mathbf{f}^*$ are conditionally independent, given $\mathbf{u}$

$$p(\mathbf{f}, \mathbf{f}^*) \approx q(\mathbf{f}, \mathbf{f}^*) = \int q(\mathbf{f} \,|\, \mathbf{u}) q(\mathbf{f}^* \,|\, \mathbf{u}) p(\mathbf{u}) d\mathbf{u}.$$

From this dependency structure, it can be seen that $\mathbf{f}$ and $\mathbf{f}^*$ are only dependent through the information expressed in $\mathbf{u}$. The fundamental difference between each of the four sparse Gaussian process schemes implemented in this package is the additional assumptions that each scheme imposes upon the conditional distributions $q(\mathbf{f} \,|\, \mathbf{u})$ and $q(\mathbf{f}^* \,|\, \mathbf{u})$. In the exact case, these two conditional distributions can be expressed as

$$p(\mathbf{f} \,|\, \mathbf{u}) = \mathcal{N}(\mathbf{K}_{\mathbf{f},\mathbf{u}} \mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1} \mathbf{u}, \mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{Q}_{\mathbf{f},\mathbf{f}}) \tag{15}$$

$$p(\mathbf{f}^* \,|\, \mathbf{u}) = \mathcal{N}(\mathbf{K}_{*,\mathbf{u}} \mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1} \mathbf{u}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}), \tag{16}$$

where $\mathbf{Q}_{\mathbf{f},\mathbf{f}} = \mathbf{K}_{\mathbf{fu}} \mathbf{K}_{\mathbf{uu}}^{-1} \mathbf{K}_{\mathbf{uf}}$.

The simplest, and most computationally efficient, sparse method is the subset of regressors (SoR) strategy. SoR assumes a deterministic relationship between each $\mathbf{f}$ and $\mathbf{u}$, making the Gaussian process marginal predictive distribution (4) equivalent to

$$q(y^* \,|\, \mathbf{x}^*, \mathcal{D}, \boldsymbol{\theta}) = \mathcal{N}\left(\sigma^{-2} \mathbf{K}_{\mathbf{f}^*,\mathbf{u}} \Sigma \mathbf{K}_{\mathbf{u},\mathbf{f}} \mathbf{y}, \mathbf{K}_{\mathbf{f}^*,\mathbf{u}} \Sigma \mathbf{K}_{\mathbf{u},\mathbf{f}^*}\right), \tag{17}$$

where $\Sigma = \left(\sigma^{-2} \mathbf{K}_{\mathbf{u},\mathbf{f}} \mathbf{K}_{\mathbf{f},\mathbf{u}} + \mathbf{K}_{\mathbf{u},\mathbf{u}}\right)^{-1}$. Such scalability comes at the great cost of wildly inaccurate predictive uncertainties that often underestimate the true posterior variance as

[1]A required assumption for any valid stochastic process, consistency assumes that if we marginalise out part of the process, then the resulting marginal distribution will be the same as the distribution defined in the original sequence of the process.

the model can only express m degrees-of-freedom. This result occurs as at most m linearly independent functions can be drawn from the prior, and consequently prior variances are poorly approximated.

A more elegant sparse method, is the deterministic training conditional (DTC) approach of Seeger, Williams, and Lawrence (2003). DTC addresses the issue of inaccuracy within the Gaussian process posterior variance by computing the Gaussian process' likelihood using information from all n data points; not just $\mathbf{u}$. This is achieved by projecting $\mathbf{f}$ such that $\mathbf{f} = \mathbf{K}_{\mathrm{f,u}}\mathbf{K}_{\mathrm{u,u}}^{-1}\mathrm{u}$. With an exact likelihood computation, an approximation is still required on the Gaussian process' joint prior

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}^* \end{pmatrix} \mid \mathbf{X}, \mathbf{x}^*, \boldsymbol{\theta} \sim \mathcal{N}\left(0, \begin{bmatrix} \mathbf{Q}_{\mathbf{f},\mathbf{f}} & \mathbf{Q}_{\mathbf{f},*} \\ \mathbf{Q}_{*,\mathbf{f}} & \mathbf{K}_{*,*} \end{bmatrix}\right). \tag{18}$$

Through retention of an exact likelihood, coupled with an approximate prior, a deterministic relationship need only be imposed on $\mathbf{u}$ and $\mathbf{f}$, allowing for an exact test conditional ((16)) to be computed. Given that the test conditional is now exact, and the prior variance of $\mathbf{f}^*$ is computed using $\mathbf{K}_{*,*}$, not $\mathbf{Q}_{*,*}$, more reasonable predictive uncertainties are now produced. Note, while an exact test conditional is now being computed, a DTC approximation is not an exact Gaussian process as the process is no longer consistent across training and test cases due to the inclusion of $\mathbf{K}_{*,*}$ in (18).

The fully independent training conditional (FITC) scheme enables a richer covariance structure by preserving the exact prior covariances along the diagonal of the sparse covariance matrix (Snelson and Ghahramani 2006). This can be seen in the model's joint prior

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}^* \end{pmatrix} \mid \mathbf{X}, \mathbf{x}^*, \boldsymbol{\theta} \sim \mathcal{N}\left(0, \begin{bmatrix} \mathbf{Q}_{\mathbf{f},\mathbf{f}} - \operatorname{diag}\left[\mathbf{Q}_{\mathbf{f},\mathbf{f}} - \mathbf{K}_{\mathbf{f},\mathbf{f}}\right] & \mathbf{Q}_{\mathbf{f},*} \\ \mathbf{Q}_{*,\mathbf{f}} & \mathbf{K}_{*,*} \end{bmatrix}\right). \tag{19}$$

As with the DTC, FITC imposes an approximation to the training conditional from (15), but computes (16) exactly. An important extension to (19), is proposed in Quiñonero-Candela and Rasmussen (2005) whereby the prior variance for $\mathbf{f}^*$ is reformulated as $\mathbf{Q}_{*,*} - \operatorname{diag}\left[\mathbf{Q}_{*,*} - \mathbf{K}_{*,*}\right]$. This assumption of full independence within the conditionals of both $\mathbf{f}$ and $\mathbf{f}^*$ ensures that the FITC approximation is equivalent to exact inference within a non-degenerate Gaussian process; a property not enjoyed by the aforementioned sparse approximations[2].

The final sparse method implemented within the package is the full-scale approximation of Sang and Huang (2012). A full-scale approximation further enriches the prior covariance structure by imposing a series blocked matrix corrective terms along diagonal of $\mathbf{f}$

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}^* \end{pmatrix} \mid \mathbf{X}, \mathbf{x}^*, \boldsymbol{\theta} \sim \mathcal{N}\left(0, \begin{bmatrix} \mathbf{Q}_{\mathbf{f},\mathbf{f}} - \operatorname{blockdiag}\left[\mathbf{Q}_{\mathbf{f},\mathbf{f}} - \mathbf{K}_{\mathbf{f},\mathbf{f}}\right] & \mathbf{Q}_{\mathbf{f},*} - \operatorname{blockdiag}\left[\mathbf{Q}_{\mathbf{f},*} - \mathbf{K}_{\mathbf{f},*}\right] \\ \mathbf{Q}_{*,\mathbf{f}} - \operatorname{blockdiag}\left[\mathbf{Q}_{*,\mathbf{f}} - \mathbf{K}_{*,\mathbf{f}}\right] & \mathbf{K}_{*,*} \end{bmatrix}\right). \tag{20}$$

The predictive uncertainties that a full-scale approach yields will be far superior to any of the previous sparse approximation, however, this comes at the cost of an increased computational complexity due to the presence of a more dense covariance matrix. As with the DTC and FITC approximations, the exact test conditional of eq. (16) is preserved,

[2]Note, in this package (19) has been implemented, however, the proposed extension by Quiñonero-Candela and Rasmussen (2005) is left for future work within the package

while the approximation of the training conditional in eq. (15) takes the form $q(\mathbf{f} \mid \mathbf{u}) = \mathcal{N}(\mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \text{blockdiag}\left[\mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{Q}_{\mathbf{f},\mathbf{f}}\right])$.

Adopting a full-scale approach requires the practitioner to specify the number of blocks k, apriori. The trade-off when making this decision is that fewer blocks will result in a more accurate predictive distribution, however, the computational complexity will increase. As recommended by Tresp (2000), it is commonly advised to select $k = \frac{n}{m}$, where each block is of dimension $m \times m$. In the extreme case that $k = m$, the full-scale approach becomes a FITC approximation, and if $k = 1$, just a single block will exist, and the exact Gaussian process will be recovered.

A final note with regard to sparse approximations is that the set of inducing point locations $\mathbf{X}_\mathbf{u}$, such that $\mathbf{u} = \mathbf{f}(\mathbf{X}_\mathbf{u})$, will heavily influence the process. Modern extensions to the sparse methods detailed above seek to *learn* $\mathbf{X}_\mathbf{u}$ concurrently during hyper-parameter optimisation. However, such an approach, whilst elegant, requires first-order derivatives of $\mathbf{u}$ to be available; a functionality not currently available in the package. Instead, the practitioner is required to specify a set of points apriori that correspond to the locations of $\mathbf{X}_\mathbf{u}$. A simple, yet effective, approach to this is to divide the input space up into equidistant knots and use these knot points as $\mathbf{X}_\mathbf{u}$.

3. The package

The package can be downloaded from the Julia package repository during a Julia session by using the package manager tool. The `]` symbol activates the package manager, after which the **GaussianProcesses.jl** package can be installed with the following command `add GaussianProcesses`. Alternatively, the **Pkg** package can be used with command

```
julia>  Pkg.add("GaussianProcesses")
```

Julia will also install all of the required dependency packages. Documentation for types and functions in the package, like other documentation in Julia, can be accessed through the help mode in the Julia REPL. Help mode is activated by typing `?` at the prompt, and documentation can then be searched by entering the name of a function or type.

```
julia> ?
help?> optimize!
search: optimize!

optimize!(gp::GPBase; kwargs...)

Optimise the hyperparameters of Gaussian process gp based on type II
maximum likelihood estimation. This function performs gradient-based
optimisation using the Optim pacakge to which the user is referred to
for further details.

Keyword arguments:

* `domean::Bool`: Mean function hyperparameters should be optimized
* `kern::Bool`: Kernel function hyperparameters should be optimized
* `noise::Bool`: Observation noise hyperparameter should be optimized (GPE only)
* `lik::Bool`: Likelihood hyperparameters should be optimized (GPA only)
* `kwargs`: Keyword arguments for the optimize function from the Optim package
```

The main function in the package is `GP`, which fits the Gaussian process model to covariates $\mathbf{X}$ and responses $\mathbf{y}$. As discussed in the previous section, the Gaussian process is completely specified by its `mean` and `kernel` functions and possibly a `likelihood` when the observations $\mathbf{y}$ are non-Gaussian.

```
julia> gp = GP(X,y,mean,kernel)
julia> gp = GP(X,y,mean,kernel,likelihood)
```

This highlights the use of the Julia multiple dispatch feature. The `GP` function will, in the background, construct either an object of type `GPE` or `GPA` for exact or approximate inference, respectively, depending on whether or not a `likelihood` function is specified. If no likelihood function is given, then it is assumed that $\mathbf{y}$ are Gaussian distributed as in the case analytic case of eq. (5).

In this section we will highlight the functionality of the package by considering a simple Gaussian process regression example which follows the tractable case outlined in Section 2.1. We start by loading the package and simulating some data.

```
julia> using GaussianProcesses, Random

julia> Random.seed!(13579) # Set the seed using the 'Random' package
julia> n = 10; # number of training points
julia> x = 2π * rand(n); # predictors
julia> y = sin.(x) + 0.05*randn(n); # regressors
```

Note that Julia supports UTF-8 characters, and so one can use Greek characters to improve the readability of the code.

The first step in modelling data with a Gaussian process is to choose the mean and kernel functions which describe the data. There are a variety of mean and kernel functions available in the package (see Appendix A for a list). Note that all hyperparameters for the mean and kernel functions and the observation noise, σ, are given on the log scale. The Gaussian process is represented by an object of type `GP` and constructed from the observation data, a mean function and kernel, and optionally the observation noise.

```
# Select mean and covariance function
julia> mZero = MeanZero() # Zero mean function
julia> kern = SE(0.0,0.0) # Sqaured exponential kernel
julia> logObsNoise = -1.0 # log standard deviation of observation noise
julia> gp = GP(x,y,mZero,kern,logObsNoise) # Fit the GP
```

For this example we have used a zero mean function and squared exponential kernel with signal standard deviation and length scale parameters equal to 1.0 (recalling that inputs are on the log scale). After fitting the `GP`, a summary output is produced which provides some basic information on the GP object, including the type of mean and kernel functions used, as well as returning the value of the marginal log-likelihood eq. (8). Once the user has applied the `GP` function to the the data, a summary of the GP object is printed.

```
GP Exact object:
Dim = 1
Number of observations = 10
Mean function:
Type: GaussianProcesses.MeanZero, Params: Any[]
Kernel:
Type: GaussianProcesses.SEIso, Params: [0.0,0.0]
Input observations =
[5.66072 1.67222 ... 6.08978 3.39451]
Output observations = [-0.505287,1.02312,0.616955,-0.777658,-0.875402, ...
Variance of observation noise = 0.1353352832366127
Marginal Log-Likelihood = -6.719
```

Once we have fitted the `GP` function to the data we can calculate the predicted mean and variance of the function at unobserved points $\{\mathbf{x}^*, y^*\}$, conditional on the observed data $\mathcal{D} = \{\mathbf{y}, \mathbf{X}\}$. This is done with the `predict_y` function. We can also calculate the predictive distribution for the latent function f^* using the `predict_f` function. The `predict_y` function returns the mean vector $\mu(\mathbf{x}^*)$ and covariance matrix $\Sigma(\mathbf{x}^*, \mathbf{x}^{*'})$ of the predictive distribution in eq. (7) (or variance vector if `full_cov=false`).

```
julia> x = 0:0.1:2π # a sequence between 0 and 2π with 0.1 spacing
julia> μ, Σ = predict_y(gp,x);
```

Plotting one and two-dimensional GPs is straightforward and in the package we utilise the recipes approach to plotting graphs from the **Plots.jl**[3] package. **Plots.jl** provides a general interface for plotting with several different backends including **PyPlot.jl**[4], **Plotly.jl**[5] and **GR.jl**[6]. The default plot function `plot(gp)` outputs the predicted mean and variance of the function (i.e., uses `predict_f` in the background), with the uncertainty in the function represented by a confidence ribbon (set to 95% by default). All optional plotting arguments are given after the `;` symbol.

```
julia> using Plots
julia> pyplot() # Optionally select a plotting backend
# Plot the GP
julia> plot(gp; xlabel="x", ylabel="y",
            title="Gaussian process", legend=false, fmt=:png)
```

The parameters $\boldsymbol{\theta}$ are optimised using the **Optim.jl** package (see the right-hand side of Figure 3). This offers users a range of optimisation algorithms which can be applied to estimate the parameters using maximum likelihood estimation. Gradients are available for all mean and kernel functions used in the package and therefore it is recommended that the user utilises gradient-based optimisation techniques. As a default, the `optimize!` function uses the L-BFGS solver, however, alternative solvers can be applied and the user should refer to the **Optim.jl** documentation for further details. For highly non-convex models, gradient-based methods will only converge to a local optimum. In these settings, a popular pragmatic

[3]http://docs.juliaplots.org/latest/
[4]https://github.com/JuliaPy/PyPlot.jl
[5]https://plot.ly/julia/
[6]https://github.com/jheinen/GR.jl

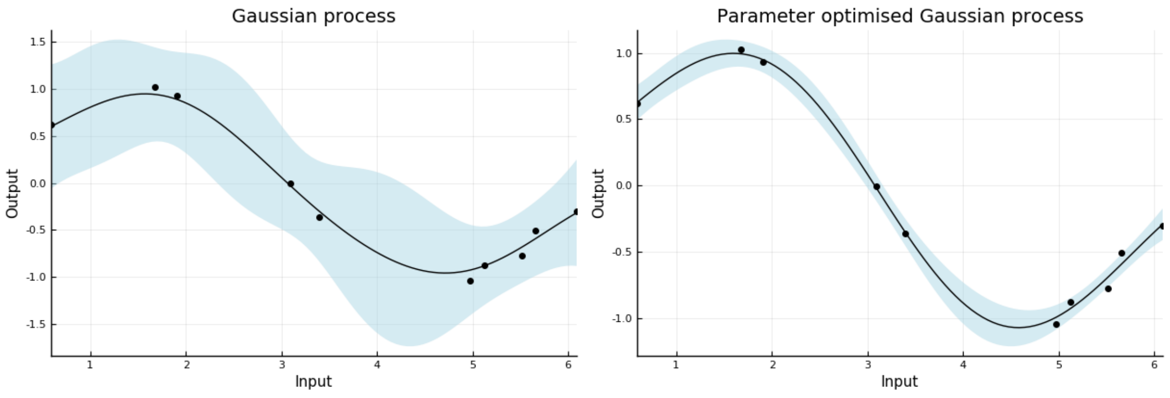


Figure 2: One dimensional Gaussian process regression with initial kernel parameters (left) and optimised parameters (right).

solution is to run the optimizer multiple times for different initial parameter values and then choose the best parameter set from these multiple runs.

```
julia> optimize!(gp) #Optimise the parameters

Results of Optimization Algorithm
* Algorithm: L-BFGS
* Starting Point: [-1.0,0.0,0.0]
* Minimizer: [-2.683055260944582,0.4342151847965596, ...]
* Minimum: -4.902989e-01
* Iterations: 9
* Convergence: true
* |x - x'| < 1.0e-32: false
* |f(x) - f(x')| / |f(x)| < 1.0e-32: false
* |g(x)| < 1.0e-08: true
* f(x) > f(x'): false
* Reached Maximum Number of Iterations: false
* Objective Function Calls: 38
* Gradient Calls: 38
```

Parameters can be estimated using a Bayesian approach, where instead of maximising the log-likelihood function, we can approximate the marginal posterior distribution $p(\boldsymbol{\theta}, \sigma \mid \mathcal{D}) \propto p(\mathcal{D} \mid \boldsymbol{\theta}, \sigma)p(\boldsymbol{\theta}, \sigma)$. We use MCMC sampling (specifically HMC sampling) to draw samples from the posterior distribution with the `mcmc` function. Prior distributions are assigned to the parameters of the mean and kernel parameters through the `set_priors!` function. The log-noise parameter σ is set to a non-informative prior $p(\sigma) \propto 1$. A wide range of prior distributions are available through the **Distributions.jl** package. Further details on the MCMC sampling of the package is given in Section 4.1.

```
julia> using Distributions

# Uniform priors are used as default if priors are not specified
julia> set_priors!(kern, [Normal(0,1), Normal(0,1)])
julia> chain = mcmc(gp)
julia> plot(chain', label=["Noise", "SE log length", "SE log scale"])
```

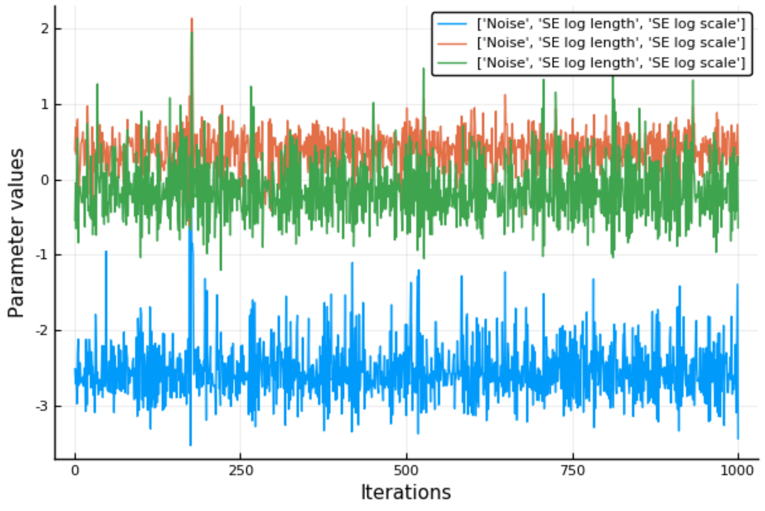


Figure 3: Trace plots of the MCMC output for the posterior samples

The regression example above can be easily extended to higher dimensions. For the purpose of visualisation, and without loss of generality, we consider a two-dimensional regression example. When $d > 1$ (recalling that $\mathcal{X} \subseteq \mathbb{R}^d$), there is the option to either use an isotropic (Iso) kernel or an automatic relevance determination (ARD) kernel. The Iso kernels have one length scale parameter ℓ which is the same for all dimensions. The ARD kernels, however, have different length scale parameters for each dimension. To obtain Iso or ARD kernels, a kernel function is called either with a single length scale parameter or with a vector of parameters. For example, below we will use the Matérn 5/2 ARD kernel, if we wanted to use the Iso alternative instead, we would set the kernel as `kern=Matern(5/2,0.0,0.0)`.

In this example we use a composite kernel represented as the sum of a Matérn 5/2 ARD kernel and a squared exponential isotropic kernel. This is easily implemented using the `+` symbol, or in the case of a product kernel, using the `*` symbol.

```
# Simulate data for a 2D Gaussian process
julia> n = 10 # number of data points
julia> X = 2π * rand(2, n) # inputs
julia> y = sin.(X[1,:]) .* cos.(X[2,:]) + 0.5 * rand(n) # outputs

julia> kern = Matern(5/2,[0.0,0.0],0.0) + SE(0.0,0.0) # sum of two kernels
julia> gp2 = GP(X,y,MeanZero(),kern)

GP Exact object:
Dim = 2
Number of observations = 10
Mean function:
Type: GaussianProcesses.MeanZero, Params: Any[]
Kernel:
 Type: GaussianProcesses.SumKernel
 Type: GaussianProcesses.Mat52Ard, Params: [-0.0, -0.0, 0.0]
 Type: GaussianProcesses.SEIso, Params: [0.0, 0.0]
Input observations =
[5.28142 6.07037 ... 2.27508 0.15818; 3.72396 2.72093 ... 3.54584 4.91657]
Output observations = [1.03981, 0.427747, -0.0330328, 1.0351, 0.889072, 0.491157, ...
Variance of observation noise = 0.01831563888873418
Marginal Log-Likelihood = -12.457
```

By default, the in-built `plot` function returns only the mean of the GP in the two-dimensional setting. There is an optional `var` argument which can be used to plot the two-dimensional variance (see Figure 3).

```
# Plot mean and variance
julia> p1 = plot(gp2; title="Mean of GP")
julia> p2 = plot(gp2; var=true, title="Variance of GP", fill=true)
julia> plot(p1, p2; fmt=:pdf)
```

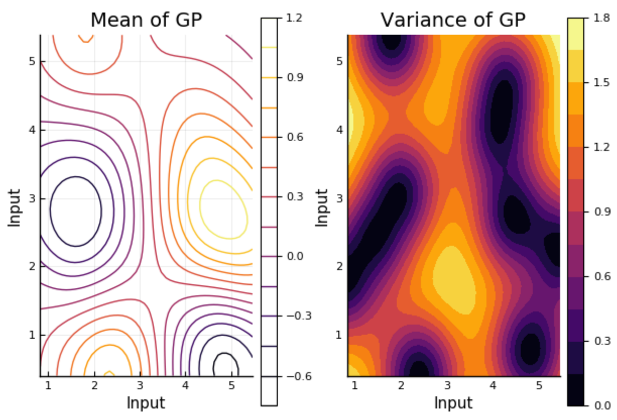


Figure 4: GP mean and variance from the two-dimensional process.

The **Plots.jl** package provides a flexible recipe structure which allows the user to change the plotting backend, e.g., **PyPlot.jl** to **GR.jl**. The package also provides a rich array of plotting functions, such as contour, surface and heatmap plots.

```
julia> gr() # use GR backend to allow wireframe plot
julia> p1 = contour(gp2)
julia> p2 = surface(gp2)
julia> p3 = heatmap(gp2)
julia> p4 = wireframe(gp2)
julia> plot(p1, p2, p3, p4; fmt=:pdf)
```

4. Demos

So far we have considered Gaussian processes where the data **y** are assumed to follow a Gaussian distribution centred around the latent Gaussian process function **f** eq. (5). As highlighted in Section 2.2, Gaussian processes can easily be extended to model non-Gaussian

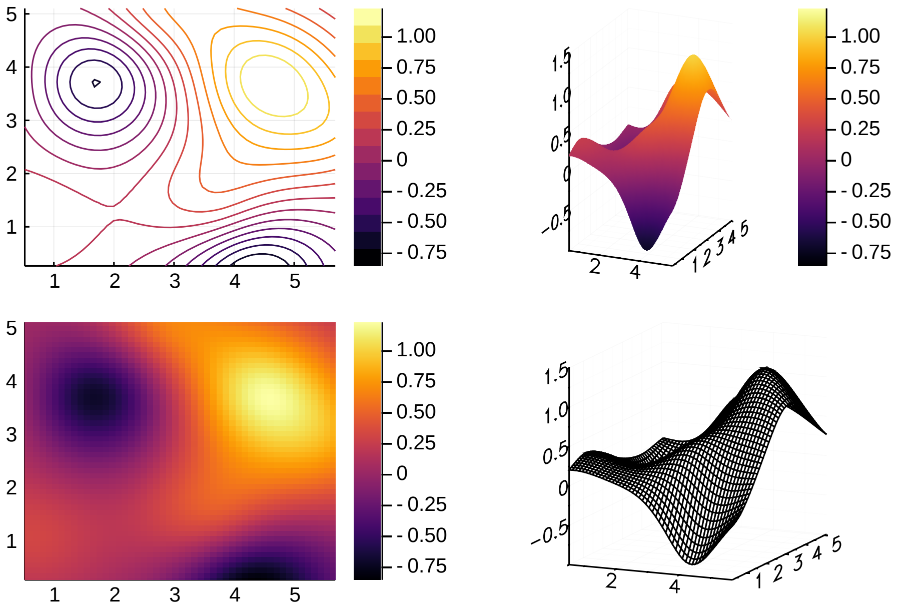


Figure 5: Two-dimensional plot of the GP mean with a range of available plotting options. Clockwise from the top left: contour, surface, wireframe and heatmap plots

data by assuming that the data are conditional on a latent Gaussian process function. This approach has been widely applied, for example, in machine learning for classification problems (Williams and Barber 1998a) and in geostatistics for spatial point process modelling (Møller, Syversveen, and Waagepetersen 1998). In this section, we will show how the **GaussianProcesses.jl** package can be used to fit Gaussian process models for binary classification, time series and count data. The code for all examples is also available in a Notebook format[7].

4.1. Binary classification

In this example we show how the approximate GP function can be used for supervised learning classification with MCMC used to estimate the latent process. We use the Crab dataset from the R package **MASS**. In this dataset we are interested in predicting whether a crab is of colour form blue or orange. Our aim is to perform a Bayesian analysis and calculate the posterior distribution of the latent GP function $\mathbf{f}$ and parameters $\boldsymbol{\theta}$ from the training data $\{\mathbf{X}, \mathbf{y}\}$.

```
julia> using GaussianProcesses, RDatasets, Random
julia> Random.seed!(113355)

julia> crabs = dataset("MASS","crabs"); # load the data
julia> crabs = crabs[shuffle(1:size(crabs)[1]), :]; # shuffle the data

julia> train = crabs[1:div(end,2),:]; # training data
julia> y = Array{Bool}(undefsize(train)[1]); # response
```

[7]https://github.com/STOR-i/GaussianProcesses.jl/tree/master/notebooks

```
julia> y[train[:Sp].=="B"]=0; # convert characters to booleans
julia> y[train[:Sp].=="O"]=1;
julia> X = convert(Array,train[:,4:end]); # predictors
```

We assume a zero mean GP with a Matérn 3/2 kernel. We use the automatic relevance determination (ARD) kernel to allow each dimension of the predictor variables to have a different length scale. As this is binary classification, we use the Bernoulli likelihood,

$$y_i \sim \text{Bernoulli}(\Phi(f_i)),$$

where $\Phi : \mathbb{R} \to [0,1]$ is the cumulative distribution function of a standard Gaussian and acts as a link function that maps the GP function to the interval [0,1], giving the probability that $y_i = 1$. Note that `BernLik` requires the observations to be of type `Bool` and unlike some likelihood functions (e.g., Student-t) does not contain any parameters to be set at initialisation.

```
#Select mean, kernel and likelihood function
julia> mZero = MeanZero(); # Zero mean function
julia> kern = Matern(3/2,zeros(5),0.0); # Matern 3/2 ARD kernel
julia> lik = BernLik(); # Bernoulli likelihood for binary data {0,1}
```

We fit the Gaussian process using the general `GP` function. This function is a shorthand for the `GPA` function, which is used to generate approximations of the latent function using MCMC or variational inference when the likelihood is non-Gaussian.

```
julia> gp = GP(X',y,mZero,kern,lik) # Fit the Gaussian process model

GP Approximate object:
Dim = 5
Number of observations = 100
Mean function:
Type: GaussianProcesses.MeanZero, Params: Float64[]
Kernel:
Type: GaussianProcesses.Mat32Ard, Params: [-0.0,-0.0,-0.0,-0.0,-0.0,0.0]
Likelihood:
Type: GaussianProcesses.BernLik, Params: Any[]
Input observations =
[16.2 11.2 ... 11.6 18.5; 13.3 10.0 ... 9.1 14.6; ... ; 41.7 26.9 ... 28.4 42.0;
Output observations = Bool[false,false,false,false,true,true,false,true, ...
Log-posterior = -161.209
```

As we are taking a Bayesian approach to infer the latent function and model parameters, we shall assign prior distributions to the unknown variables. As outlined in the general modelling framework (9), the latent function $\mathbf{f}$ is reparameterised as $\mathbf{f} = L_{\boldsymbol{\theta}}\mathbf{v}$, where $\mathbf{v} \sim \mathcal{N}(\mathbf{0}_n, \mathbf{I}_n)$ is the prior on the transformed latent function. Using the **Distributions.jl** package we can assign normal priors to each of the Matérn kernel parameters. If the mean and likelihood functions also contained parameters then we could set these priors in the way same using `gp.mean` and `gp.lik` in place of `gp.kernel`, respectively.

```
julia> set_priors!(gp.kernel,[Distributions.Normal(0.0,2.0) for i in 1:6])
```

Samples from the posterior distribution of the latent function and parameters $\mathbf{f}, \theta \,|\, \mathcal{D}$, are drawn using MCMC sampling. The `mcmc` function uses a Hamiltonian Monte Carlo sampler (Neal 2010). By default, the function runs for `nIter=1000` iterations and uses a step-size of $\epsilon = 0.01$ with a random number of leap-frog steps L between 5 and 15. Setting `Lmin=1` and `Lmax=1` gives the MALA algorithm (Roberts and Rosenthal 1998). Additionally, after the MCMC sampling is complete, the Markov chain can be post-processed using the `burn` and `thin` arguments to remove the burn-in phase (e.g., first 100 iterations) and thin the Markov chain to reduce the autocorrelation by removing values systematically (e.g., if `thin=5` then only every fifth value is retained).

```
julia> samples = mcmc(gp; ϵ=0.01, nIter=10000, burn=1000, thin=10);
```

We assess the predictive accuracy of the fitted model against a held-out test dataset

```
julia> test = crabs[div(end,2)+1:end,:]; # select test data
julia> yTest = Array{Bool}(undef,size(test)[1]); # test response data
julia> yTest[test[:Sp].=="B"]=0; # convert characters to booleans
julia> yTest[test[:Sp].=="O"]=1;
julia> xTest = convert(Array,test[:,4:end]);
```

Using the posterior samples $\{\mathbf{f}^{(i)}, \boldsymbol{\theta}^{(i)}\}_{i=1}^{N}$ from $p(\mathbf{f}, \boldsymbol{\theta} \,|\, \mathcal{D})$ we can make predictions about y^*, as in eq. (11), using the `predict_y` function and sample predictions conditional on the MCMC samples. We do this by looping over the N posterior samples and for each iteration i we fix the GP function $\mathbf{f}^{(i)}$ and hyperparameters $\boldsymbol{\theta}^{(i)}$ to their posterior sample value.

```
julia> ymean = Array{Float64}(undef,size(samples,2),size(xTest,1));

julia> for i in 1:size(samples,2)
           set_params!(gp,samples[:,i]) # Fix GP at posterior values
           update_target!(gp) # Update the GP function with the new parameters
           ymean[i,:] = predict_y(gp,xTest')[1] # Store the predictive mean
       end
```

For each of the posterior samples we plot (see Figure 6) the predicted observation y^* (given as lines) and overlay the true observations from the held-out data (circles).

```
julia> using Plots
julia> gr()

julia> plot(ymean',leg=false)
julia> scatter!(yTest)
```

4.2. Time series

Gaussian processes can be used to model nonlinear time series. We consider the problem of predicting future concentrations of CO_2 in the atmosphere. The data are taken from the Mauna Loa observatory in Hawaii which records the monthly average atmospheric concentration of CO_2 (in parts per million) between 1958 to 2015. For the purpose of testing the predictive accuracy of the Gaussian process model, we fit the GP to the historical data from 1958 to 2004 and optimise the parameters using maximum likelihood estimation.

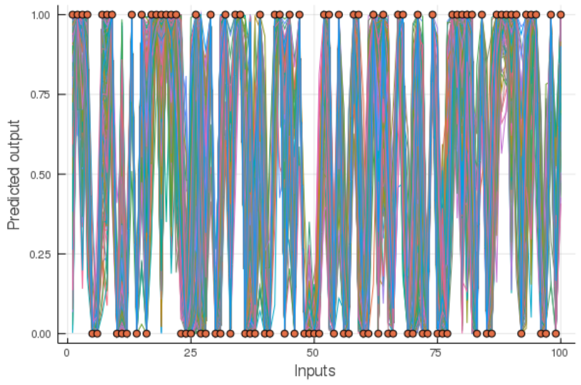


Figure 6: Posterior samples overlayed with predicted observations from a held-out test dataset.

We employ a seemingly complex kernel function to model these data which follows the kernel structure given in (Rasmussen and Williams 2006, Chapter 5). The kernel comprises of simpler kernels with each kernel term accounting for a different aspect in the variation of the data. For example, the `Periodic` kernel captures the seasonal effect of CO_2 absorption from plants. A detailed description of each kernel contribution is given in (Rasmussen and Williams 2006, Chapter 5).

```
julia> using GaussianProcesses, CSV

#Location of data within the package
julia> data_dir = joinpath(dirname(dirname(pathof(GaussianProcesses))),
                             "notebooks/data")

#Load the data (the data is in the package directory)
julia> data = CSV.read(joinpath(data_dir, "CO2_data.csv"),header=0)
julia> year = data[:,1]; co2 = data[:,2];

# Split the data into training and testing data
julia> xtrain = year[year.<2004]; ytrain = co2[year.<2004];
julia> xtest = year[year.>=2004]; ytest = co2[year.>=2004];

# Kernel is represented as a sum of kernels
julia> kernel = SE(4.0,4.0) + Periodic(0.0,1.0,0.0) * SE(4.0,0.0)
                + RQ(0.0,0.0,-1.0) + SE(-2.0,-2.0);
julia> gp = GP(xtrain,ytrain,MeanZero(),kernel,-2.0) #Fit the GP
julia> optimize!(gp) #Find the maximum likelihood estimation
julia> μ, Σ = predict_y(gp,xtest);

julia> using Plots #Load the Plots.jl package with the pyplot backend
julia> pyplot()

julia> plot(xtest,μ,ribbon=Σ, title="Time series prediction",
            label="95% predictive confidence region",fmt=:pdf)
```

```
julia> scatter!(xtest,ytest,label="Observations")
```

The predictive accuracy of the Gaussian process is plotted in Figure 7. Over the ten year prediction horizon the GP is able to accurately capture both the trend and seasonal variations of the CO_2 concentrations. Arguably, the GP prediction gradually begins to underestimate the CO_2 concentration. The accuracy of the fit could be further improved by extending the kernel function to include additionally terms. Recent work on automatic structure discovery (Duvenaud, Lloyd, Grosse, Tenenbaum, and Ghahramani 2013) could be used to optimise the modelling process.

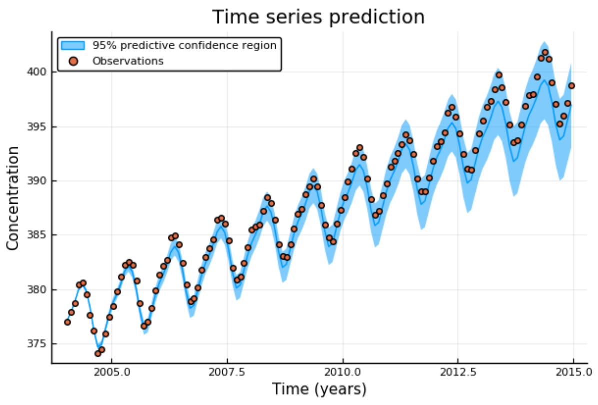


Figure 7: Predictive mean and 95% confidence interval for CO_2 measurements at the Mauna Loa observatory from 2004 to 2015

4.3. Count data

Gaussian process models can be incredibly flexible for modelling non-Gaussian data. One such example is in the case of count data $\mathbf{y}$, which can be modelled with a Poisson distribution, where the log-rate parameter can be modelled with a latent Gaussian process.

$$\mathbf{y} \mid \mathbf{f} \sim \prod_{i=1}^{n} \frac{\lambda_i^{y_i} \exp\{-\lambda_i\}}{y_i!},$$

where $\lambda_i = \exp(f_i)$ and f_i is the latent Gaussian process.

The package contains two methods for performing inference for Gaussian processes with non-Gaussian data: MCMC and variational inference, as described in Section 2.2. To demonstrate the performance of both algorithms, we will simulate 20 observations such that $\mathbf{y} \sim \text{Poisson}\left(\exp\left(f(x_i)\right)\right)$, where $f(x_i) = 2\cos(2x_i)$ and $x_i \in [-3, 3]$.

```
julia> using GaussianProcesses, Distributions, Plots, Random
julia> pyplot()

julia> n = 20
julia> X = collect(range(-3,stop=3,length=n));
julia> f = 2*cos.(2*X);
julia> Y = [rand(Poisson(exp.(f[i]))) for i in 1:n];
```

```
#GP set-up
julia> k = Matern(3/2,0.0,0.0) # Matern 3/2 kernel
julia> l = PoisLik() # Poisson likelihood
```

We can compare the accuracy of the MCMC versus the VI approximation by fitting an approximate GP using both methods.

```
# Define MCMC and VI models respectively
julia> gpmc = GP(X, vec(Y), MeanZero(), k, l)
julia> gpvi = GP(X, vec(Y), MeanZero(), k, l);

# Set the priors and sample from the posterior for the MCMC model
julia> set_priors!(gpmc.kernel,[Normal(-2.0,4.0),Normal(-2.0,4.0)])
julia> set_priors!(gpvi.kernel,[Normal(-2.0,4.0),Normal(-2.0,4.0)]);
```

The MCMC algorithm is a sampling-based approach and so we use the `mcmc()` function to draw samples from the posterior distribution. Conditional on these posterior samples we can also draw samples from the predictive distribution

```
# Perform MCMC and sample posterior realisation
julia> samples = mcmc(gpmc; nIter=10000);

julia> xtest = range(minimum(gpmc.x),stop=maximum(gpmc.x),length=50);
julia> ymean = [];
julia> fsamples = Array{Float64}(undef,size(samples,2), length(xtest));
julia> for i in 1:size(samples,2)
           set_params!(gpmc,samples[:,i])
           update_target!(gpmc)
           push!(ymean, predict_y(gpmc,xtest)[1])
           fsamples[i,:] = rand(gpmc, xtest)
       end
```

The variational inference approach for parameter estimation relies on optimisation rather than sampling to approximate the posterior. Using the ELBO (13) function as an objective, we can optimise the free variational parameters to maximise the ELBO function in a similar manner as we maximise the marginal log-likelihood function to optimise the GP parameters in the exact Gaussian processes setting (see Section 3 for an example).

```
# Optimise variational distribution Q and sample posterior realisations
julia> Q = vi(gpvi);

julia> xtest = range(minimum(gpmc.x),stop=maximum(gpmc.x),length=50);
julia> nsamps = 500
julia> ymean = [];
julia> visamples = Array{Float64}(undef, nsamps, length(xtest))

julia> for i in 1:nsamps
           visamples[i, :] = rand(gpvi, xtest, Q);
           push!(ymean, predict_y(gpvi, xtest)[1]);
       end
```

We can plot the predictive posterior samples using both MCMC and VI to give a visual comparison of the two methods.

```
# Plot realisations of MCMC and VI Gaussian processes
julia> q10 = [quantile(fsamples[:,i], 0.1) for i in 1:length(xtest)]
julia> q50 = [quantile(fsamples[:,i], 0.5) for i in 1:length(xtest)]
julia> q90 = [quantile(fsamples[:,i], 0.9) for i in 1:length(xtest)]
julia> plot(xtest,exp.(q50),ribbon=[exp.(q10), exp.(q90)],leg=true,
           fmt=:png, label="quantiles", title = "MCMC Inference")
julia> plot!(xtest,mean(ymean), label="posterior mean")
julia> xx = range(-3,stop=3,length=1000);
julia> f_xx = 2*cos.(2*xx);
julia> plot!(xx, exp.(f_xx), label="truth")
julia> scatter!(X,Y, label="data")

julia> q10 = [quantile(visamples[:,i], 0.1) for i in 1:length(xtest)];
julia> q50 = [quantile(visamples[:,i], 0.5) for i in 1:length(xtest)];
julia> q90 = [quantile(visamples[:,i], 0.9) for i in 1:length(xtest)];
julia> plot(xtest, exp.(q50), ribbon=[exp.(q10), exp.(q90)], leg=true,
           fmt=:png, label="quantiles",
           title="Variationally Approximate Inference")
julia> plot!(xtest, mean(ymean), label="posterior mean", w=2)
julia> xx = range(-3,stop=3,length=1000);
julia> f_xx = 2*cos.(2*xx);
julia> plot!(xx, exp.(f_xx), label="truth")
julia> scatter!(X,Y, label="data")
```

The results of both algorithms are presented in Figure 8. As can be seen in the left-hand panel, MCMC offers a far richer approximation to the posterior of the Gaussian process. In the asymptote, MCMC will in fact yield the true posterior, whereas variational inference is not equipped with such theoretically desirable guarantees. Further, as the posterior approximation is constrained to be a set of independent Gaussian distributions, it will often fail to capture the intricate details that exist in a non-Gaussian posterior. Typically, variational methods scaled more efficiently than MCMC as it does not require the costly evaluation of the proposal distribution. Instead modern techniques such as stochastic optimisation and graphics processing unit (GPU) accelerations can be incorporated into variational schemes, a task left for future work.

4.4. Bayesian optimization

This section introduces the **BayesianOptimization.jl**[8] package, which requires **GaussianProcesses.jl** as a dependency. We highlight some of the memory-efficiency features of Julia and show how Gaussian processes can be applied to optimize noisy or costly black-box objective functions Shahriari, Swersky, Wang, Adams, and de Freitas (2016). In Bayesian optimization, an objective function $l(\mathbf{x})$ is evaluated at some points $y_1 = l(\mathbf{x}_1), y_2 = l(\mathbf{x}_2), \ldots, y_t = l(\mathbf{x}_t)$. A model $\mathcal{M}(\mathcal{D}_t)$, e.g., a Gaussian process, is fitted to these observations $\mathcal{D}_t = \{(\mathbf{x}_i, y_i)\}_{i=1,\ldots,t}$

[8]https://github.com/jbrea/BayesianOptimization.jl

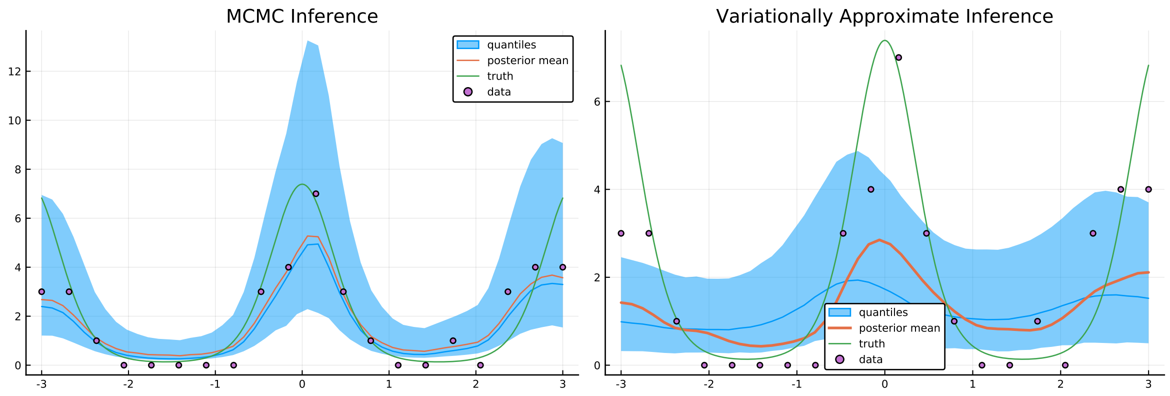


Figure 8: Samples from the MCMC approximated posterior (left) and variational posterior (right) of the Gaussian process with a Poisson observation model. The points show the simulated observations, the orange line shows the modal posterior values and the blue ribbon shows the 10% and 90% quantile range.

and used to determine the next input point $\mathbf{x}_{t+1}$ at which the objective function should be evaluated. The model is refitted with inclusion of the new observation $(\mathbf{x}_{t+1}, y_{t+1})$ and $\mathcal{M}(\mathcal{D}_{t+1})$ is used to acquire the next input point. With a clever acquisition of next input points, Bayesian optimization can find the optima of the objective function with fewer function evaluations than alternative optimization methods Shahriari *et al.* (2016).

Since the observed data sets in different time steps are highly correlated, $\mathcal{D}_{t+1} = \mathcal{D}_t \cup \{(\mathbf{x}_{t+1}, y_{t+1})\}$, it would be wasteful to refit a Gaussian process to $\mathcal{D}_{t+1}$ without considering the model $\mathcal{M}(\mathcal{D}_t)$ that was already fit to $\mathcal{D}_t$. To avoid refitting, **GaussianProcesses.jl** includes the function `ElasticGPE` that creates a Gaussian process where it costs little to add new observations. In the following example, we create an elastic and exact GP for two input dimensions with an initial capacity for 3000 observations, and an increase in capacity for 1000 observations, whenever the current capacity limit is reached.

```
julia> gp = ElasticGPE(2, # two input dimensions
                      mean = MeanConst(0.),
                      kernel = SEArd([0., 0.], 5.),
                      logNoise = 0.,
                      capacity = 3000,
                      stepsize = 1000)


GP Exact object:
Dim = 2
Number of observations = 0
Mean function:
  Type: MeanConst, Params: [0.0]
Kernel:
  Type: SEArd{Float64}, Params: [-0.0, -0.0, 5.0]
No observation data

julia> append!(gp, [1., 2.], 0.4) # append observation x = [1., 2.] and y = 0.4
```

```
GP Exact object:
Dim = 2
Number of observations = 1
Mean function:
  Type: MeanConst, Params: [0.0]
Kernel:
  Type: SEArd{Float64}, Params: [-0.0, -0.0, 5.0]
Input observations =
  [1.0; 2.0]
Output observations = [0.4]
Variance of observation noise = 1.0
Marginal Log-Likelihood = -5.919
```

Under the hood, elastic GPs allocates memory for the number of observations specified with the keyword argument `capacity` and uses `views` to select only the part of memory that is already filled with actual observations. Whenever the current capacity `c` is reached, memory for `c + stepsize` observations is allocated and the old data copied over. Elastic GPs uses efficient rank-one updates of the Cholesky decomposition that holds the covariance data of the GP.

In the following example we use Bayesian optimization on a not so costly, but noisy one-dimensional objective function, $f(x) = 0.1 \cdot (x-2)^2 + \cos(\pi/2 \cdot x) + \epsilon$, where $\epsilon \sim \mathcal{N}(0,1)$, which is illustrated in Figure 9.

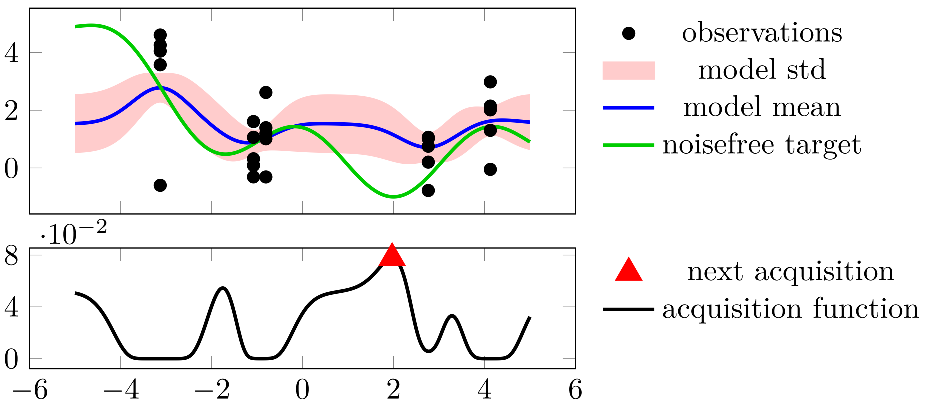


Figure 9: After evaluating the noisy function $f(x) = 0.1 \cdot (x-2)^2 + \cos(\pi/2 \cdot x) + \epsilon$, where $\epsilon \sim \mathcal{N}(0,1)$, at five positions five times (black dots) and fitting a Gaussian process (blue line with red standard deviations), the expected improvement acquisition function (black line) peaks near input value 2, where the next acquisition will occur.

```
julia> using BayesianOptimization, GaussianProcesses

julia> f(x) = 0.1*(x[] - 2)^2 + cos(pi/2*x[]) + randn() # noisy function

# Choose as a model an elastic GP with input dimensions 1.
julia> model = ElasticGPE(1, mean = MeanConst(0.),
                          kernel = SEArd([0.], 5.), logNoise = 0.)

 # Optimize the hyperparameters of the GP using maximum likelihood (ML)
 # estimates every 50 steps woth bounds on the logNoise and
 # bounds for the two parameters GaussianProcesses.get_param_names(model.kernel)
julia> modeloptimizer = MAPGPOptimizer(every = 50,
                                      noisebounds = [-2., 3],
                                      kernbounds = [[-1, 0], [4, 10]],
                                      maxeval = 40)

julia> opt = BOpt(f, model,
            ExpectedImprovement(), # type of acquisition function
            modeloptimizer,
            [-5.], [5.], # lower- and upperbounds
            repetitions = 5, # evaluate the function 5 times
            maxiterations = 200, # evaluate at 200 input positions
            sense = Min, # minimize the objective function
            acquisitionoptions = (maxeval = 4000, restarts = 50),
            verbosity = Silent)
julia> result = boptimize!(opt)
```

```
(observed_optimum = -3.737657255325198, observed_optimizer = [2.02252],
model_optimum = -0.9836243116981216, model_optimizer = [1.97054])
```

BayesianOptimization.jl uses automatic differentiation tools in **ForwardDiff.jl** (Revels, Lubin, and Papamarkou 2016) to compute gradients of the acquisition function (`ExpectedImprovement()` in the example above). After evaluating the function at 200 positions, the global minimum of the Gaussian process at `model_optimizer = [1.99274]` is close to the global minimum of the noise-free objective function.

4.5. Sparse inputs

In this section we will demonstrate how each of the sparse approximations detailed in Section 2.3 can be used. The performance of each sparse method will be demonstrated by fitting a sparse Gaussian process to a set of $n = 5000$ data points that are simulated from $f(x) = |x - 5|\cos(2x)$,

```
julia> using Random

# The true function we will be simulating from is,
julia> function fstar(x::Float64)
           return abs(x-5)*cos(2*x)
       end

julia> σy = 10.0 # observation noise
julia> n=5000 # number of observations
julia> Random.seed!(1) # for reproducibility
julia> Xdistr = Beta(7,7)
julia> ϵdistr = Normal(0,σy)
julia> x = rand(Xdistr, n)*10
julia> X = Matrix(x')
julia> Y = fstar.(x) .+ rand(ϵdistr,n)
```

The set of inducing point locations $X_{\mathbf{u}}$ used here will be consistent for each method and are defined explicitly.

```
julia> Xu = Matrix(quantile(x, [0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55,
                                0.6, 0.65, 0.7, 0.98])')
```

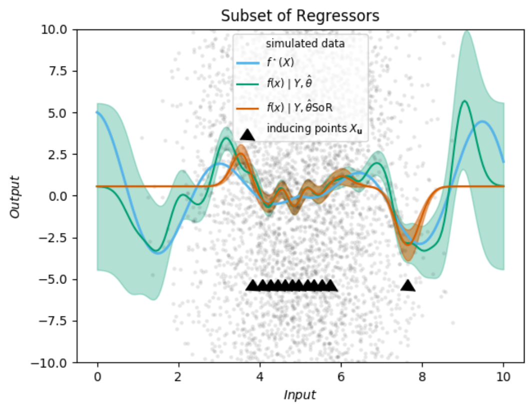
With the inducing point locations defined, we can now fit each of the sparse Gaussian process approximations. The practitioner is free to select from any of the sparse approaches outlined in Section 2.3, each of which can be invoked using the below syntax. The only syntactic deviation is the full scale approach, which requires the practitioner to choose the covariance matrix's local blocks. In this example, m blocks have been created, with a one-to-one mapping between block and inducing point locations.

```
# Subset of Regressors
julia> gp_SOR = GaussianProcesses.SoR(X, Xu, Y, MeanConst(mean(Y)),
                                      k, log(σy));

# Determinetal Training Conditional
julia> gp_DTC = GaussianProcesses.DTC(X, Xu, Y, MeanConst(mean(Y)),
                                      k, log(σy));

# Fully Independent Training Conditional
julia> gp_FITC = GaussianProcesses.FITC(X, Xu, Y, MeanConst(mean(Y)),
                                        k, log(σy));

# Full Scale
julia> inearest = [argmin(abs.(xi.-Xu[1,:])) for xi in x]
julia> blockindices = [findall(isequal(i), inearest) for i in 1:size(Xu,2)]

julia> GaussianProcesses.FSA(X, Xu, blockindices, Y, MeanConst(mean(Y)),
                             k, log(σy));
```

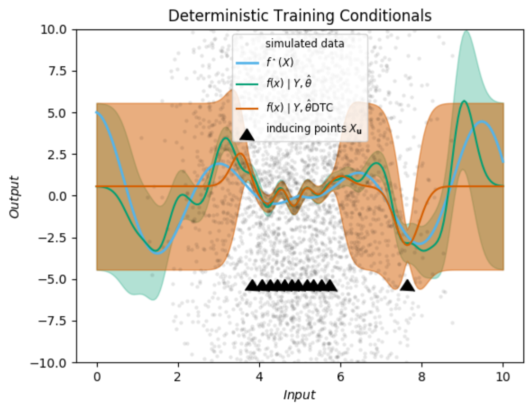
Prediction is handled in the same way as a regular Gaussian process, using the `predict_f` function.

As discussed in Section 2.3, each sparse method yields a computational acceleration, however, this often comes at the cost of poorer predictive inference, as shown in Figure 10. This is no more apparent than in the SoR approach, where the posterior predictions are excessively confident, particularly beyond the range of the inducing points. Both the DTC and FITC are more conservative in the predictive uncertainty as the process moves away from the inducing points' location, while sacrificing little in terms of computational efficiency[9] - see Table 1 for computational timing results.

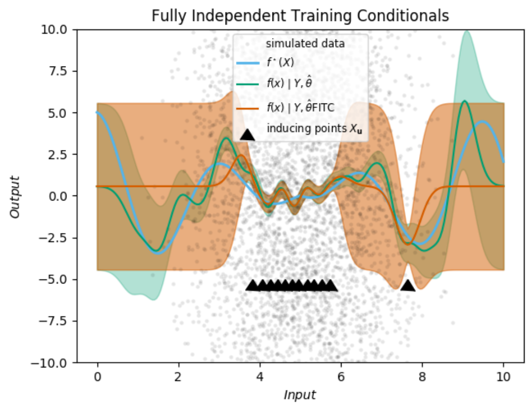
[9]All simulations run on a Linux machine with a 1.60GHz i5-8250U CPU and 16GB RAM.



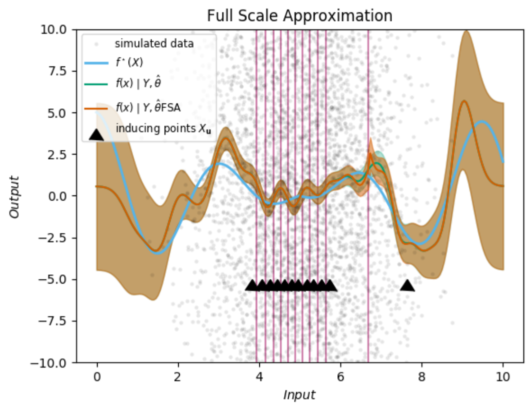
(a) SoR



(b) DTC



(c) FITC



(d) Full Scale

Figure 10: Comparison of sparse approximations to an exact Gaussian process. The vertical purple lines in panel d indicate the dividing lines between blocks where an information discontinuity will occur.

	CPU Runtime (seconds)	Memory Allocation (MiB)
Exact	1.417324	572.320
SoR	0.004076	**2.032**
DTC	**0.003104**	2.033
FITC	0.022644	3.902
Full Scale	0.383900	156.025

Table 1: The computational efficiency of fitting each of sparse approximations to the training data.

Kernel	GaussianProcesses.jl	GPy	GPML
`fix(SE(0.0,0.0),` σ`)`	**730**	1255	
`SE(0.0,0.0)`	**800**	1225	1131
`Matern(1/2,0.0,0.0)`	**836**	1254	1246
`Masked(SE(0.0,0.0), [1]))`	**819**	1327	1075
`RQ(0.0,0.0,0.0)`	**1252**	1845	1292
`SE(0.0,0.0) + RQ(0.0,0.0,0.0)`	**1351**	1937	1679
`Masked(SE(0.0,0.0), [1])` `+ Masked(RQ(0.0,0.0,0.0), collect(2:10))`	**1562**	1893	1659
`(SE(0.0,0.0) + SE(0.5,0.5)) * RQ(0.0,0.0,0.0)`	**1682**	1953	2127
`SE(0.0,0.0) * RQ(0.0,0.0,0.0)`	**1614**	1929	1779
`(SE(0.0,0.0) + SE(0.5,0.5)) * RQ(0.0,0.0,0.0)`	**1977**	2042	2206

Table 2: Benchmark results, ordered by running time in **GaussianProcesses.jl**. All times are in milliseconds, and the fastest run-time is bolded. The kernels are labelled with their function name from the package: `SE` is a squared exponential kernels; `RQ` is a rational quadratic kernel; `Matern(1/2,..)` is a Matérn 1/2 kernel; sum and product kernels are indicated with `+` and `*`; `fix(SE(0.0,0.0),` σ`)` has a fixed variance parameter (not included in the gradient); and `Masked(k,[dims])` means the `k` kernel is only applied to the covariates `dims`.

5. Comparison to other packages

In this section we compare the performance of **GaussianProcesses.jl** to two leading Gaussian process inference packages for the fundamental task of computing the log-likelihood, and its gradient, in a simulated problem with a Gaussian likelihood. We use version 4.1 of the MATLAB package **GPML** (Rasmussen and Nickisch 2010, 2017), which was originally written to demonstrate the algorithms in Rasmussen and Williams (2006), and has since become a mature package, often integrating new algorithms from the latest research on Gaussian processes. The package is mostly written in pure MATLAB, except for a small number of optimisation and linear algebra routines implemented in C. We also compare to version 1.9.2 of **GPy** (GPy since 2012), a python package dedicated to Gaussian processes, with core components written in cython. We first simulated $n = 3,000$ standard normal observations, each with 10 covariates also simulated as standard normals. We reuse the same simulated dataset for every benchmark. In each package, we benchmark the function that updates the log-likelihood and its gradient given a set of parameters, by running it 10 times and reporting the duration of the shortest run. We compare the packages' performance for a variety of covariance kernels, with all variance, length-scale, or shape parameters set to 1.0. The results are presented in Table 2, and the benchmark code for each package is available with the **GaussianProcesses.jl** source code.

We find that **GaussianProcesses.jl** is highly competitive with **GPML** and **GPy**. It has the fastest run-times for all of the 10 kernels considered, including the additive and product kernels.

6. Future developments

GaussianProcesses.jl is a fully formed package providing a range of kernel, mean and likelihood functions, and inference tools for Gaussian process modelling with Gaussian and non-Gaussian data types. The inclusion of new features in the package is ongoing and the development of the package can be followed via the Github page[10]. The following are package enhancements currently under development:

- Automatic differentiation - The package provides functionality for maximum likelihood estimation of the hyperparameters, or Bayesian inference using an MCMC algorithm. In both cases, these functions require gradients to be calculated for optimisation or sampling. Currently, derivatives of functions of interest (e.g., log-likelihood function) are hand-coded for computational efficiency. However, recent tests have shown that calculating these gradients using automatic differentiation does not incur a significant additional computational overhead. In the future, the package will move towards implementing all gradient calculations using automatic differentiation. The main advantage of this approach is that users will be able to add new functionality to the package more easily, for example creating a new kernel functions.

- Gaussian Process Latent Variable Model (GPLVM) - Currently the package is well suited for supervised learning tasks, whereby an observational value exists for each input. GPLVMs are a probabilistic dimensionality reduction method that use Gaussian processes to learn a mapping between an observed, possibly very high-dimensional, variable and a reduced dimension latent space. GPLVMs are a popular method for dimensionality reduction as they transcend principal component analysis by learning a non-linear relationship between the observations and corresponding latent space. Furthermore, a GPLVM is also able to express the uncertainty surrounding the structure of the latent space. In the future, the package will support the original GPLVM of Lawrence (2004), and its Bayesian counterpart Titsias and Lawrence (2010).

7. Acknowledgements

The authors would like to thank the users of the **GaussianProcesses.jl** package who have helped to support its development. CN gratefully acknowledges the support of EPSRC grants EP/S00159X/1, EP/R01860X/1, EP/V022636/1 and EP/V022636/1. JB was supported by Swiss National Science Foundation (no. 200020_184615) and by the European Union Horizon 2020 Framework Program under grant agreement no. 785907 (HumanBrain Project, SGA2). TP is supported by the Data Science for the Natural Environment project (EPSRC grant number EP/R01860X/1).

References

Besançon M, Anthoff D, Arslan A, Byrne S, Lin D, Papamarkou T, Pearson J (2019). “Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem.” *arXiv e-prints*, arXiv:1907.08611.

[10]https://github.com/STOR-i/GaussianProcesses.jl

Bezanson J, Edelman A, Karpinski S, Shah V (2017). "Julia: A Fresh Approach to Numerical Computing." *SIAM Review*, **59**(1), 65–98.

Datta A, Banerjee S, Finley AO, Gelfand AE (2016). "Hierarchical nearest-neighbor Gaussian process models for large geostatistical datasets." *Journal of the American Statistical Association*, **111**(514), 800–812.

Deisenroth MP, Fox D, Rasmussen CE (2015). "Gaussian processes for data-efficient learning in robotics and control." *IEEE transactions on pattern analysis and machine intelligence*, **37**(2), 408–423.

Duvenaud D, Lloyd J, Grosse R, Tenenbaum J, Ghahramani Z (2013). "Structure discovery in nonparametric regression through compositional kernel search." *Proceedings of the International Conference on Machine Learning (ICML)*, **30**, 1166–1174.

Duvenaud DK (2014). *Automatic Model Construction with Gaussian Processes.* Ph.D. thesis, University of Cambridge.

Filippone M, Zhong M, Girolami M (2013). "A comparative evaluation of stochastic-based inference methods for Gaussian process models." *Machine Learning*, **93**(1), 93–114. ISSN 08856125.

Foreman-Mackey D, Agol E, Ambikasaran S, Angus R (2017). "Fast and scalable Gaussian process modeling with applications to astronomical time series." *The Astronomical Journal*, **154**(6), 220.

Gonzalez JP, Cook S, Oberthur T, Jarvis A, Bagnell JA, Dias MB (2007). "Creating Low-Cost Soil Maps for Tropical Agriculture Using Gaussian Processes." `doi:10.1184/R1/6552461.v1`. Robotics Institute Centre for Tropical Agriculture Carnegie Mellon University.

GPy (since 2012). "GPy: A Gaussian process framework in python." `http://github.com/SheffieldML/GPy`.

Guhaniyogi R, Li C, Savitsky TD, Srivastava S (2017). "A divide-and-conquer Bayesian approach to large-scale kriging." *arXiv preprint arXiv:1712.09767*.

Hensman J, Matthews AG, Filippone M, Ghahramani Z (2015). "MCMC for Variationally Sparse Gaussian Processes." In C Cortes, N Lawrence, D Lee, M Sugiyama, R Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Khan E, Mohamed S, Murphy KP (2012). "Fast Bayesian inference for non-conjugate Gaussian process regression." In *Advances in Neural Information Processing Systems*, pp. 3140–3148.

Lawrence ND (2004). "Gaussian process latent variable models for visualisation of high dimensional data." *Advances in neural information processing systems*, pp. 329–336.

Liu Q, Pierce DA (1994). "A Note on Gauss-Hermite Quadrature." *Biometrika*, **81**(3), 624–629.

Matheron G (1963). "Principles of geostatistics." *Economic geology*, **58**(8), 1246–1266.

Matthews AG, Van Der Wilk M, Nickson T, Fujii K, Boukouvalas A, León-Villagrá P, Ghahramani Z, Hensman J (2017). "GPflow: A Gaussian Process Library using TensorFlow." *Journal of Machine Learning Research*, **18**(40), 1–6.

Melkumyan A, Ramos FT (2009). "A sparse covariance function for exact Gaussian process inference in large datasets." *Twenty-First International Joint Conference on Artificial Intelligence*.

Minka TP (2001). *A family of algorithms for approximate Bayesian inference*. Ph.D. thesis, Massachusetts Institute of Technology.

Mogensen PK, Riseth AN (2018). "Optim: A mathematical optimization package for Julia Usage in research and industry." *Journal of Open Source Software*, **3**(24). ISSN 2475-9066.

Møller J, Syversveen AR, Waagepetersen RP (1998). "Log Gaussian Cox processes." *Scandinavian Journal of Statistics*, **25**(3), 451–482.

Murray I (2016). "Differentiation of the Cholesky decomposition." *arXiv preprint arXiv:1602.07527*.

Murray I, Adams RP (2010). "Slice sampling covariance hyperparameters of latent Gaussian models." *Advances in Neural Information Processing*, **2**(1), 9.

Neal RM (2010). "MCMC Using Hamiltonian Dynamics." In *Handbook of Markov Chain Monte Carlo (Chapman & Hall/CRC Handbooks of Modern Statistical Methods)*, pp. 113–162. CRC press.

Nickisch H, Rasmussen CE (2008). "Approximations for Binary Gaussian Process Classification." *Journal of Machine Learning Research*, **9**, 2035–2078.

Opper M, Archambeau C (2009). "The variational Gaussian approximation revisited." *Neural computation*, **21**(3), 786–792.

Osborne MA, Roberts SJ, Rogers A, Jennings NR (2008). "Real-Time Information Processing of Environmental Sensor Network Data using Bayesian Gaussian Processes." *ACM Transactions of Sensor Networks*, **V**(N), 109–120.

Quiñonero-Candela J, Rasmussen CE (2005). "A unifying view of sparse approximate Gaussian process regression." *Journal of Machine Learning Research*, **6**(Dec), 1939–1959.

Rasmussen CE, Nickisch H (2010). "Gaussian processes for machine learning (GPML) toolbox." *Journal of machine learning research*, **11**(Nov), 3011–3015.

Rasmussen CE, Nickisch H (2017). "GPML v4.1." MATLAB/Octave package. Last accessed June 2018., URL `http://www.gaussianprocess.org/gpml/code/matlab/doc/index.html`.

Rasmussen CE, Williams C (2006). *Gaussian processes for machine learning*. MIT Press.

Revels J, Lubin M, Papamarkou T (2016). "Forward-Mode Automatic Differentiation in Julia." *arXiv:1607.07892*. URL `https://arxiv.org/abs/1607.07892`.

Ripley BD (2009). *Stochastic simulation*, volume 316. John Wiley & Sons.

Robert CP (2004). *Monte Carlo methods*. Wiley Online Library.

Roberts GO, Rosenthal JS (1998). "Optimal scaling of discrete approximations to Langevin diffusions." *Journal of the Royal Statistical Society B*, **60**(1), 255–268. ISSN 1369-7412.

Roberts GO, Rosenthal JS (2004). "General state space Markov chains and MCMC algorithms." *Probability Surveys*, **1**(0), 20–71. ISSN 1549-5787.

Sang H, Huang JZ (2012). "A full scale approximation of covariance functions for large spatial data sets." *Journal of the Royal Statistical Society B*, **74**(1), 111–132.

Seeger M, Williams C, Lawrence N (2003). "Fast forward selection to speed up sparse Gaussian process regression." *Ninth International Workshop on Artifi-cial Intelligence and Statistics.*

Shahriari B, Swersky K, Wang Z, Adams RP, de Freitas N (2016). "Taking the Human Out of the Loop: A Review of Bayesian Optimization." *Proceedings of the IEEE*, **104**(1), 148–175.

Snelson E, Ghahramani Z (2006). "Sparse Gaussian processes using pseudo-inputs." *Advances in neural information processing systems*, pp. 1257–1264.

Titsias M (2009). "Variational learning of inducing variables in sparse Gaussian processes." *Artificial Intelligence and Statistics*, pp. 567–574.

Titsias M, Lawrence ND (2010). "Bayesian Gaussian process latent variable model." *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 844–851.

Titsias MK, Lawrence N, Rattray M (2008). "Markov chain Monte Carlo algorithms for Gaussian processes." *Inference and Estimation in Probabilistic Time-Series Models*, **9**.

Tresp V (2000). "A Bayesian committee machine." *Neural computation*, **12**(11), 2719–2741.

Wang L, Durante D, Jung RE, Dunson DB (2017). "Bayesian network-response regression." *Bioinformatics*, **33**(12), 1859–1866.

Williams CK, Barber D (1998a). "Bayesian classification with Gaussian processes." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **20**(12), 1342–1351.

Williams CK, Seeger M (2001). "Using the Nyström method to speed up kernel machines." *Advances in neural information processing systems*, pp. 682–688.

Williams CKI, Barber D (1998b). "Bayesian classification with Gaussian processes." *IEEE Trans Pattern Analysis and Machine Intelligence*, **20**(12), 1342–1351.

Wilson AG, Dann C, Nickisch H (2015). "Thoughts on massively scalable Gaussian processes." *ArXiv e-prints*. URL http://arxiv.org/abs/1511.01870.

A. Available functions

Affiliation:

Jamie Fairbrother
Department of Management Science
Lancaster University
Lancaster, LA1 4YF, UK
E-mail: j.fairbrother@lancaster.ac.uk
URL: https://www.lancaster.ac.uk/~fairbrot/

Christopher Nemeth
Department of Mathematics and Statistics
Fylde College
Lancaster University
Lancaster, LA1 4YF, UK
E-mail: c.nemeth@lancaster.ac.uk
URL: https://www.lancaster.ac.uk/~nemeth/

Maxime Rischard
Cervest
69 Old St, Barbican
London, EC1V 9HX

Johanni Brea
Laboratory of Computational Neuroscience
EPFL-LCN
AAB 135 (BÃćtiment AAB)
Station 15
CH-1015 Lausanne

Thomas Pinder
Department of Mathematics and Statistics
Fylde College
Lancaster University
Lancaster, LA1 4YF, UK

Function	Description	$k_\theta(\mathbf{x}, \mathbf{x}^*) =$	$\boldsymbol{\theta}$
`Const`	Constant	σ^2	$\log \sigma$
`Lin`	Linear ARD	$\mathbf{x}^\top L^{-2} \mathbf{x}^*, \quad L = \operatorname{diag}(\ell_1, \dots, \ell_d)$	$(\log \ell_1, \dots, \log \ell_d)$
`Lin`	Linear Iso	$\mathbf{x}^\top \mathbf{x}^* / \ell^2$	$\log \ell$
`Matern(1/2,...)`	Matérn ARD (1/2)	$\sigma^2 \exp(-\lvert\mathbf{x} - \mathbf{x}^*\rvert / L), \quad L = \operatorname{diag}(\ell_1, \dots, \ell_d)$	$(\log \ell_1, \dots, \log \ell_d, \log \sigma)$
`Matern(3/2,...)`	Matérn ARD (3/2)	$\sigma^2 (1 + \sqrt{3}\lvert\mathbf{x} - \mathbf{x}^*\rvert / L) \exp(-\sqrt{3}\lvert\mathbf{x} - \mathbf{x}^*\rvert / L)$	$(\log \ell_1, \dots, \log \ell_d, \log \sigma)$
`Matern(5/2,...)`	Matérn ARD (5/2)	$\sigma^2 (1 + \sqrt{5}\lvert\mathbf{x} - \mathbf{x}^*\rvert / L + 5\lvert\mathbf{x} - \mathbf{x}^*\rvert^2 / 3L^2) \exp(-\sqrt{5}\lvert\mathbf{x} - \mathbf{x}^*\rvert / L)$	$(\log \ell_1, \dots, \log \ell_d, \log \sigma)$
`Matern(1/2,...)`	Matérn Iso (1/2)	$\sigma^2 \exp(-\lvert\mathbf{x} - \mathbf{x}^*\rvert / \ell)$	$(\log \ell, \log \sigma)$
`Matern(3/2,...)`	Matérn Iso (3/2)	$\sigma^2 (1 + \sqrt{3}\lvert\mathbf{x} - \mathbf{x}^*\rvert / \ell) \exp(-\sqrt{3}\lvert\mathbf{x} - \mathbf{x}^*\rvert / \ell)$	$(\log \ell, \log \sigma)$
`Matern(5/2,...)`	Matérn Iso (5/2)	$\sigma^2 (1 + \sqrt{5}\lvert\mathbf{x} - \mathbf{x}^*\rvert / \ell + 5\lvert\mathbf{x} - \mathbf{x}^*\rvert^2 / 3\ell^2) \exp(-\sqrt{5}\lvert\mathbf{x} - \mathbf{x}^*\rvert / \ell)$	$(\log \ell, \log \sigma)$
`SE`	Squared exponential ARD	$\sigma^2 \exp(-(\mathbf{x} - \mathbf{x}^*)^\top L^{-2} (\mathbf{x} - \mathbf{x}^*)/2) \quad L = \operatorname{diag}(\ell_1, \dots, \ell_d)$	$(\log \ell_1, \dots, \log \ell_d, \log \sigma)$
	Squared exponential Iso	$\sigma^2 \exp(-(\mathbf{x} - \mathbf{x}^*)^\top (\mathbf{x} - \mathbf{x}^*)/2\ell^2)$	$(\log \ell, \log \sigma)$
`Periodic`	Periodic	$\sigma^2 \exp(-2 \sin^2(\pi \lvert\mathbf{x} - \mathbf{x}^*\rvert / p) / \ell^2)$	$(\log \ell, \log \sigma, \log p)$
`Poly`	Polynomial (degree (d) is user defined)	$\sigma^2 (\mathbf{x}^\top \mathbf{x}^* + c)^d$	$(\log c, \log \sigma)$
`Noise`	Noise	$\sigma^2 \delta(\mathbf{x} - \mathbf{x}^*)$	$(\log \sigma)$
`RQ`	Rational Quadratic ARD	$\sigma^2 (1 + (\mathbf{x} - \mathbf{x}^*)^\top L^{-2} (\mathbf{x} - \mathbf{x}^*)/2\alpha)^{-\alpha} \quad L = \operatorname{diag}(\ell_1, \dots, \ell_d)$	$(\log \ell_1, \dots, \log \ell_d, \log \sigma, \log \alpha)$
`RQ`	Rational Quadratic Iso	$\sigma^2 (1 + (\mathbf{x} - \mathbf{x}^*)^\top (\mathbf{x} - \mathbf{x}^*)/2\alpha\ell^2)^{-\alpha}$	$(\log \ell, \log \sigma, \log \alpha)$
`FixedKernel`	Fixed kernels (fix some hyperparameters)	$k_\theta(\mathbf{x}, \mathbf{x}^*)$	$\theta' \subseteq \theta$
`Masked`	Masked kernels only active dimensions $i \subseteq \{1, , \dots, d\}$	$k_\theta(\mathbf{x}_i, \mathbf{x}_i^*)$	$\emptyset$
`*`	Product kernels	$\prod_i k_\theta(\mathbf{x}, \mathbf{x}^*)$	$\emptyset$
`+`	Sum kernels	$\sum_i k_\theta(\mathbf{x}, \mathbf{x}^*)$	$\emptyset$

Table 3: Table of available kernel functions. ARD: Automatic Relevance Determination, Iso: Isotropic

Function	Description	$p(y_i \mid f_i, \boldsymbol{\theta}) =$	Transform	$\boldsymbol{\theta} =$
`BernLik`	Bernoulli - $y_i \in \{0,1\}$	$g_i^{y_i}(1-g_i)^{(1-y_i)}$	$g_i = \Phi(f_i)$	f_i
`BinLik`	Binomial - $y_i \in \{0,1,\ldots,n\}$	$\frac{y_i!}{n!(n-y_i)!} g_i^{y_i}(1-g_i)^{(1-y_i)}$	$g_i = \frac{\exp(f_i)}{1+\exp(f_i)}$	f_i
`ExpLik`	Exponential - $y_i \in \mathbb{R}_+$	$g_i \exp(-g_i y_i)$	$g_i = \exp(-f_i)$	f_i
`GaussLik`	Gaussian - $y_i \in \mathbb{R}$	$1/\sqrt{2\pi\sigma^2} \exp\left(-(y_i - f_i)^2/2\sigma^2\right)$	f_i	$(f_i, \log\sigma)$
`PoisLik`	Poisson - $y_i \in \mathbb{N}_0$	$g_i^{y_i} \exp(-g_i)/y_i!$	$g_i = \exp(f_i)$	f_i
`StuTLik`	Student-t - $y_i \in \mathbb{R}$	$\frac{\Gamma((\nu+1)/2)}{\sqrt{\Gamma(\nu/2)\pi\nu\sigma}}(1 + \frac{1}{\nu}(\frac{(y_i-f_i)}{\sigma})^2)^{-(\nu+1)/2}$	f_i	$(f_i, \log\sigma)$

Table 4: List of available likelihood functions

Function	Description	$m_\theta(\mathbf{x}) =$	$\boldsymbol{\theta} =$
`MeanZero`	Zero	0	$\emptyset$
`MeanConst`	Constant	$\boldsymbol{\theta}, \quad \boldsymbol{\theta} = (\theta_1, \ldots, \theta_d)$	$\boldsymbol{\theta}$
`MeanLin`	Linear	$\mathbf{x}^\top\boldsymbol{\theta}, \quad \boldsymbol{\theta} = (\theta_1, \ldots, \theta_d)$	$\boldsymbol{\theta}$
`MeanPoly`	Polynomial (of degree D)	$\sum_{j=1}^{D} \boldsymbol{\theta}_j \mathbf{x}^j, \quad \boldsymbol{\theta}_j = (\theta_{1j}, \ldots, \theta_{dj})$	$\boldsymbol{\theta}_j \quad \forall j \in \{1,2,\ldots,D\}$
`+`	Sum	$\sum_i m_\theta(\mathbf{x})$	$\emptyset$
`*`	Product	$\prod_i m_\theta(\mathbf{x})$	$\emptyset$

Table 5: List of available mean functions

Function	Description	$q(\mathbf{f} \mid \mathbf{u}) =$	$q(\mathbf{f}^* \mid \mathbf{u}) =$
`SoR`	Subset of regressors	$\mathcal{N}(\mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \mathbf{0})$	$\mathcal{N}(\mathbf{K}_{*,\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \mathbf{0})$
`DTC`	Deterministic Training Conditional	$\mathcal{N}(\mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \mathbf{0})$	$\mathcal{N}(\mathbf{K}_{*,\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*})$
`FITC`	Fully Independent Training Conditional	$\mathcal{N}(\mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \operatorname{diag}\left[\mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{Q}_{\mathbf{f},\mathbf{f}}\right])$	$\mathcal{N}(\mathbf{K}_{*,\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*})$
`FSA`	Full Scale Approximation	$\mathcal{N}(\mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \operatorname{blockdiag}\left[\mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{Q}_{\mathbf{f},\mathbf{f}}\right])$	$\mathcal{N}(\mathbf{K}_{*,\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{u}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*})$

Table 6: List of available sparse approximations and their imposed conditional distributions