arXiv:1812.09064v1 [stat.CO] 21 Dec 2018

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

Jamie Fairbrother[1], Christopher Nemeth*[2], Maxime Rischard[3], and Johanni Brea[4]

[1]Department of Management Science, Lancaster University, Lancaster, UK
[2]Department of Mathematics and Statistics, Lancaster University, Lancaster, UK
[3]Department of Statistics, Harvard University, Cambridge, USA
[4]Laboratory of Computational Neuroscience, EPFL, Lausanne, Switzerland

December 24, 2018

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 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 a range of mean and kernel functions with supporting inference tools to fit the Gaussian process models, as well as a range of alternative likelihood functions to handle non-Gaussian data (e.g. binary classification models). 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 nonparametric 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 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 determine 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 et al., 2017) and brain networks (Wang et al., 2017), or for improved soil mapping (Gonzalez et al., 2007) and robotic control (Deisenroth et al., 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.

*corresponding author: c.nemeth@lancaster.ac.uk

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 in 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 closely relates to the book by Rasmussen and Williams (2006) and provides a wide range of functionality. Packages in other languages, including Python packages, e.g. **GPy** GPy (2012) and **GPFlow** Matthews et al. (2017), have ensured that new research in the area of Gaussian processes, most notably sparse Gaussian processes, has been quickly implemented in these regularly updated packages.

This paper presents a new package, **GaussianProcesses.jl**, for implementing Gaussian processes in the recently developed Julia programming language. Julia (Bezanson et al., 2017), an open source programming language, is designed specifically for numerical computation 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. 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 a well-maintained package manager system which allows users to install open source software from inside the Julia shell.

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 makes use of other Julia packages to ensure computational efficiency, for example, hyperparameters of the Gaussian process are optimized using the **Optim.jl** package (Mogensen and Riseth, 2018) and statistical distributions are derived from the **Distributions.jl** package. Additionally, this package has now become a dependency of other Julia package, for example, **BayesianOptimization.jl**, a demo of which is given in Section 4.4. The run-time speed of **GaussianProcesses.jl** has been heavily optimized and is competitive with other rival packages for Gaussian processes. A run-time comparison of the package against **GPML** and **GPy** is given in Section 5.

The paper is organised as follows. Section 2 provides an introduction to Gaussian processes and how they can be applied to model Gaussian and non-Gaussian observational data. 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 a four application demos in Section 4 which highlight how Gaussian processes can be applied to classification problems, time series modelling, count data and black-box optimization. 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 et al., 2008) to astronomical time series data (Wilson et al., 2015) all within a Bayesian nonparametric setting. A *Gaussian Process* (GP) defines a distribution over functions, $p(f)$, where $f : \mathcal{X} \rightarrow \mathbb{R}^d$ 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 $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathcal{X}$ 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 $\mathbf{y} = (y_1, y_2, \ldots, y_n) \in \mathcal{Y}$ 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*, $\mathrm{Cov}\left[f(\mathbf{x}), f(\mathbf{x}')\right] = k(\mathbf{x}, \mathbf{x}') = \sigma^2 \exp(-\frac{1}{2\ell^2}|\mathbf{x} - \mathbf{x}'|^2)$. 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. Kernels can be highly flexible and account for a wide range of behaviours. 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 gives three simple kernels (squared exponential, periodic and linear) and shows 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,

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

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

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

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

Using the modelling framework in (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, 1998) or through expectation-propagation (Minka, 2001) (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

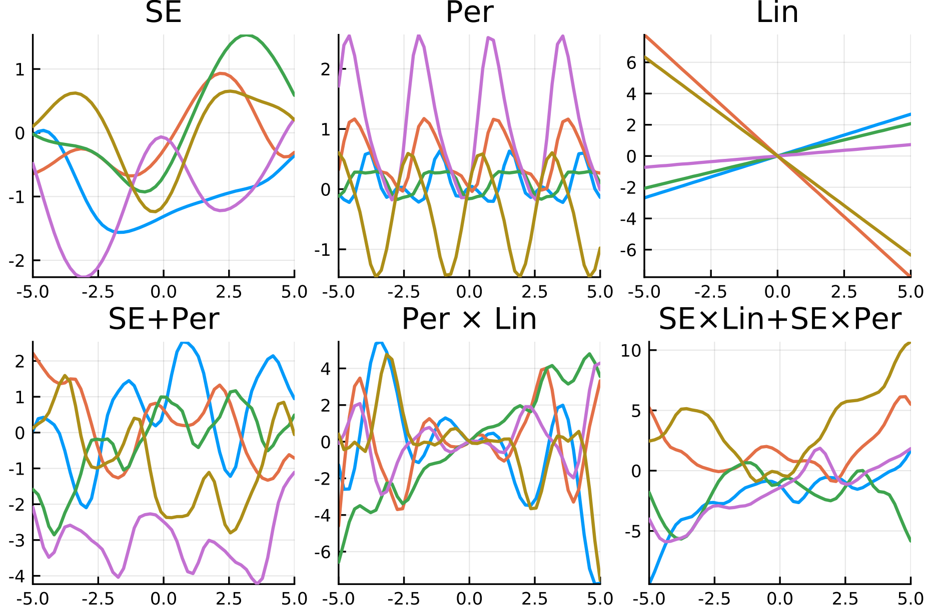


Figure 1: Clockwise from the top left. Five random samples from the Gaussian process prior using the following kernels (refer to the help file for details): `SE(0.5,0.0)`, `Periodic(0.5,0.0,1.0)`, `Lin(0.0)`, `SE(0.5,0.0)*Lin(0.0)+SE(0.5,0.0)*Periodic(0.5,0.0,1.0)`, `Periodic(0.5,0.0,1.0)*Lin(0.0)` and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

$\mathbf{x}^*$, 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. 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 (1), where the observations follow a Gaussian distribution,

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

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

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

The predictive distribution for y^* (4) can also be calculated analytically by noting that the likelihood (5), posterior (6) and conditional distribution for f^* (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 $\mu(\mathbf{x}^*) = k(\mathbf{x}^*, \mathbf{X})(\mathbf{K_{ff}} + \sigma_n^2\mathbf{I})^{-1}\mathbf{y}$ and $\Sigma(\mathbf{x}^*, \mathbf{x}^{*'}) = k(\mathbf{x}^*, \mathbf{x}^*) - k(\mathbf{x}^*, \mathbf{X})(\mathbf{K_{ff}} + \sigma_n^2\mathbf{I})^{-1}k(\mathbf{X}, \mathbf{x}^*)$ (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,

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

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

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

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, 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 in (1) is general enough to extend the Gaussian process model a wide range of data types. For example, Gaussian processes can be applied to binary classification problems (see Rasmussen and Williams (2006) Chapter 3), where $p(y_i \mid f_i, \boldsymbol{\theta})$ is a Bernoulli density function and $y_i \in \{0, 1\}$.

When the likelihood $p(\mathbf{y} \mid \mathbf{f}, \boldsymbol{\theta})$ is non-Gaussian, the posterior distribution of the latent function, conditional on observed data $p(\mathbf{f} \mid \mathcal{D}, \boldsymbol{\theta})$, 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, 1998) 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 parameterisation as given in Murray and Adams (2010); Filippone et al. (2013); Hensman et al. (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 (1) we have,

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

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 $p(\mathbf{f} \mid \mathcal{D}, \boldsymbol{\theta})$, or in the transformed setting, $p(\mathbf{v} \mid \mathcal{D}, \boldsymbol{\theta})$ 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

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

Numerous MCMC algorithms have been proposed to sample from the Gaussian process posterior (see Titsias et al. (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 (9), calculating the gradient of the posterior requires the derivative of the Cholesky factor $L_{\boldsymbol{\theta}}$. In the package, this derivative is calculated 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 $p(\boldsymbol{\theta}, \mathbf{v} \mid \mathcal{D})$. 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 (4),

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

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

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`. Julia will also install all of the required dependency packages. Help files for the package, like other Julia help files, can be accessed by first typing `?` prior to searching for the function of interest.

```
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 optmized
  * ‘kern::Bool‘: Kernel function hyperparameters should be optmized
  * ‘noise::Bool‘: Observation noise hyperparameter should be optimized (GPE only)
  * ‘lik::Bool‘: Likelihood hyperparameters should be optimized (GPMC 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.

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

The GP function will, in the background, call either GPE or GPMC for exact or Monte Carlo 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 (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.

```
using GaussianProcesses, Random

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

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
mZero = MeanZero()                # Zero mean function
kern = SE(0.0,0.0)                # Sqaured exponential kernel
logObsNoise = -1.0                # log standard deviation of observation noise

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 (8).

```
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,-0.875402,0.92976, ...
  Variance of observation noise = 0.1353352832366127
  Marginal Log-Likelihood = -6.719
```

Once the GP function has been fit 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 derive 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 (7) (or variance vector if full_cov=false).

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

Plotting one and two-dimensional GPs is straightforward and utilises the recipes approach to plotting graphs from the **Plots.jl**[1] package. **Plots.jl** provides a general interface for plotting with several different backends including **PyPlot.jl**, **Plotly.jl** and **GR.jl**. 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 `;`.

```
using Plots
pyplot()  # Optionally select a plotting backend

plot(gp; xlabel="x", ylabel="y", title="Gaussian process", legend=false, fmt=:png) # Plot the GP
```

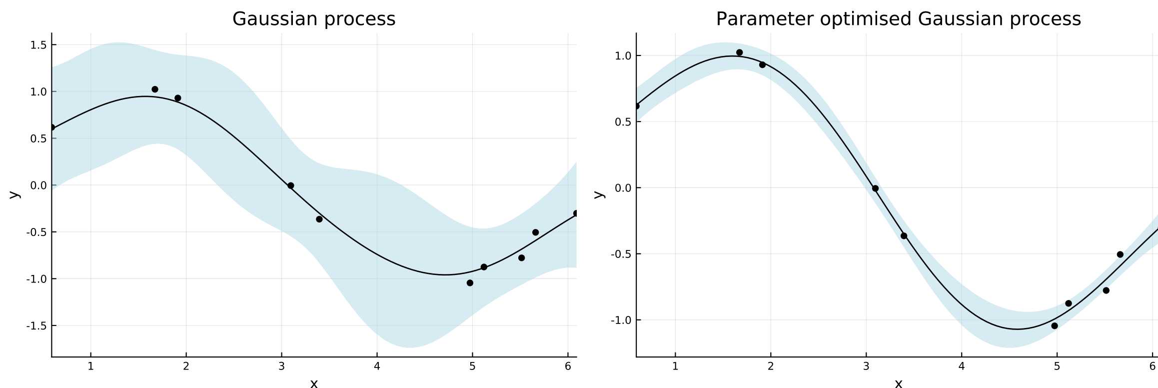


Figure 2: One dimensional Gaussian process regression with initial kernel parameters (left) and optimised parameters (right).

The parameters $\boldsymbol{\theta}$ are optimised using the **Optim.jl** package (see 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.

```
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
```

[1] http://docs.juliaplots.org/latest/

Parameters can be estimated using a Bayesian approach, where instead of maximising the log-likelihood function, we can approximate the marginal posterior distribution $p(\boldsymbol{\theta}, \sigma|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta}, \sigma)p(\boldsymbol{\theta}, \sigma)$. 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**[2] package. Further details on the MCMC sampling of the package is given in Section 4.1.

```
using Distributions

set_priors!(kern, [Normal(0,1), Normal(0,1)]) # p(θ) ∝ 1 is the default if priors are not specified
chain = mcmc(gp)
plot(chain', label=["Noise", "SE log length", "SE log scale"])
```

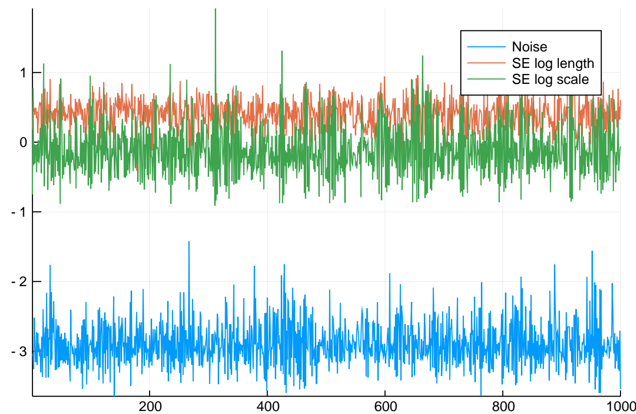


Figure 3: Trace plots of the MCMC output for the posterior samples

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 2D Gaussian process
n = 10
X = 2 *rand(2, n)
y = sin.(X[1,:]) .* cos.(X[2,:]) + 0.5*rand(n)

kern = Matern(5/2,[0.0,0.0],0.0) + SE(0.0,0.0) # sum of two kernels

gp2 = GP(X,y,MeanZero(),kern)
```

[2]https://github.com/JuliaStats/Distributions.jl

```
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
p1 = plot(gp2; title="Mean of GP")
p2 = plot(gp2; var=true, title="Variance of GP", fill=true)
plot(p1, p2; fmt=:pdf)
```

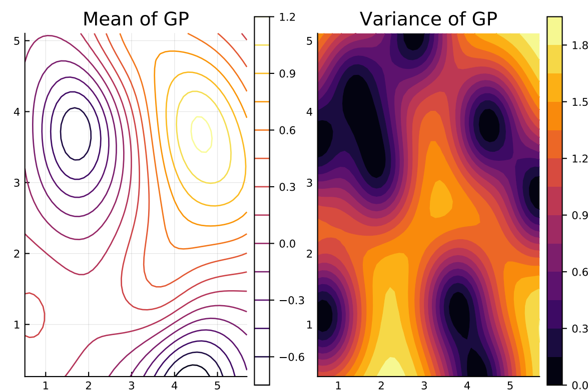


Figure 4: GP mean and variance from the two-dimensional process.

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 flexibility of this package also provides a richer array of plotting functions, such as contour, surface and heatmap plots.

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

4 Demos

So far we have considered Gaussian processes where the data $\mathbf{y}$ are assumed to follow a Gaussian distribution centred around the latent Gaussian process function $\mathbf{f}$ (5). As highlighted in Section 2.2, Gaussian processes

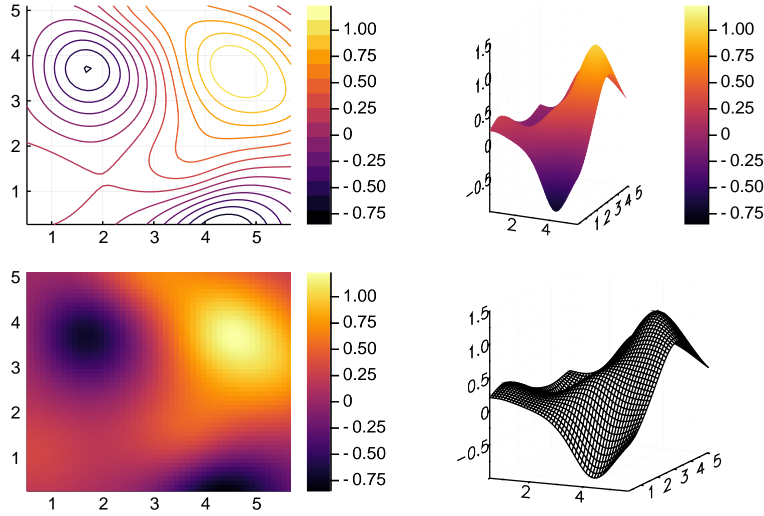


Figure 5: Two-dimensional plot of the GP mean with a range of available plotting options. Clockwise from the top left: contour, surface, wireframe and heatmap plots

can easily be extended to model non-Gaussian 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 (REF) and in geostatistics for spatial point process modelling (REF). 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.

4.1 Binary Classification

In this example we show how the GP Monte Carlo function can be used for supervised learning classification. 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 model parameters $\boldsymbol{\theta}$ from the training data $\{\mathbf{X}, \mathbf{y}\}$.

```
using GaussianProcesses, RDatasets, Random

Random.seed!(113355)

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

train = crabs[1:div(end,2),:];         # training data

y = Array{Bool}(undefsize(train)[1]);  # response
y[train[:Sp].=="B"]=0;                 # convert characters to booleans
y[train[:Sp].=="O"]=1;

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 $\Phi : \mathbb{R} \rightarrow [0,1]$ is the cumulative distribution function of a standard Gaussian and acts as a link function that mapping 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
mZero = MeanZero();              # Zero mean function
kern = Matern(3/2,zeros(5),0.0); # Matern 3/2 ARD kernel
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 `GPMC` function, which is used to generate Monte Carlo approximations of the latent function when the likelihood is non-Gaussian.

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

GP Monte Carlo 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; 15.4 9.4   10.4 16.6]
  Output observations = Bool[false,false,false,false,true,true,false,true,true,true
       false,true,false,false,false,true,false,false,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.

```
  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).

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

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

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

```
yTest[test[:Sp].=="O"]=1;
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 y^* (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.

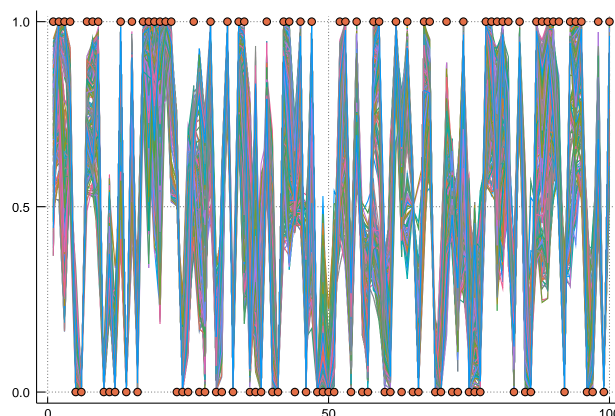
```
ymean = Array{Float64}(undef,size(samples,2),size(xTest,1));

for i in 1:size(samples,2)
    set_params!(gp,samples[:,i])      # Set the GP parameters to the 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 (Figure 4.1) the predicted observation y^* (given as lines) and overlay the true observations from the held-out data (circles).

```
using Plots
gr()

plot(ymean',leg=false)
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.

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).

```
using GaussianProcesses, DelimitedFiles

data = readdlm("data/CO2_data.csv",',')

year = data[:,1]; co2 = data[:,2];
#Split the data into training and testing data
xtrain = year[year.<2004]; ytrain = co2[year.<2004];
xtest = year[year.>=2004]; ytest = co2[year.>=2004];

#Kernel is represented as a sum of kernels
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);

gp = GP(xtrain,ytrain,MeanZero(),kernel,-2.0) #Fit the GP

optimize!(gp) #Estimate the parameters through maximum likelihood estimation

μ, Σ = predict_y(gp,xtest);

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

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

The predictive accuracy of the Gaussian process is plotted in Figure 6. 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 et al., 2013) could be used to optimise the modelling process.

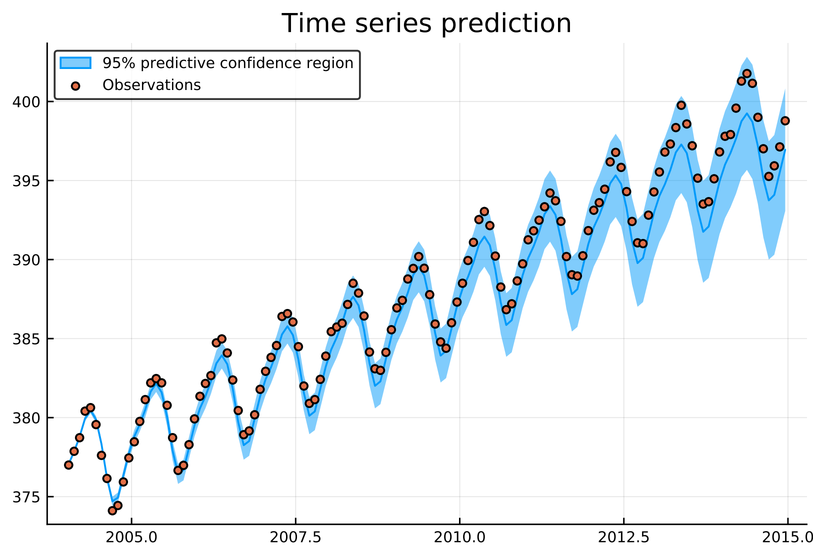


Figure 6: Predictive mean and 95% confidence interval for CO_2 measurements at the Mauna Loa observatory from 2004 to 2015

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.

In this example we will consider the dataset of recorded annual British coal mining disasters between 1851 and 1962. These data have been analysed previously (Adams et al., 2009; Lloyd et al., 2015) and it has been shown that they follow a non-homogeneous Poisson process. These data have also been extensively analysed in the changepoint literature Carlin et al. (1992); Fearnhead (2006) to identify the year in which there is a structural change to the data, i.e. change in Poisson intensity. We fit a Gaussian process to the coal mining data using a Poisson likelihood function with a Matérn 3/2 kernel function. MCMC is used to sample from the posterior distribution of the latent function and kernel parameters.

```
using GaussianProcesses, Distributions, Plots
pyplot()

coal = readcsv("notebooks/data/coal.csv")

X = coal[:,1]; Y = convert(Array{Int},coal[:,2]);

#GP set-up
k = Matern(3/2,0.0,0.0)      # Matern 3/2 kernel
l = PoisLik()                # Poisson likelihood
gp = GP(X, Y, MeanZero(), k, l) # Fit the GP

#Set the priors and sample from the posterior
set_priors!(gp.kernel,[Normal(-2.0,4.0),Normal(-2.0,4.0)])
samples = mcmc(gp; =0.08, nIter=11000,burn=1000,thin=10);

#Sample posterior function realisations
x = linspace(minimum(gp.X),maximum(gp.X),50);
fsamples = Array{Float64}(undef,size(samples,2), length(x));
for i in 1:size(samples,2)
    set_params!(gp,samples[:,i])
    update_target!(gp)
    fsamples[i,:] = rand(gp, x)
end
```

A visual inspection, given in Figure 7, of the latent function reveals a change in the log-intensity of the Poisson process after the year 1876. This change corresponds with several pieces of parliamentary legislation in the UK between 1870-1890 intended to improve the safety standards in British coal mines. Additionally, there appears to be a further decline in the number of accidents after 1935.

4.4 Bayesian Optimization

This section introduces the **BayesianOptimization.jl**[3] package, which requires **GaussianProcesses.jl** as a dependency, and where we highlight how Gaussian processes can be applied to optimize noisy or costly black-box objective functions Shahriari et al. (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

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

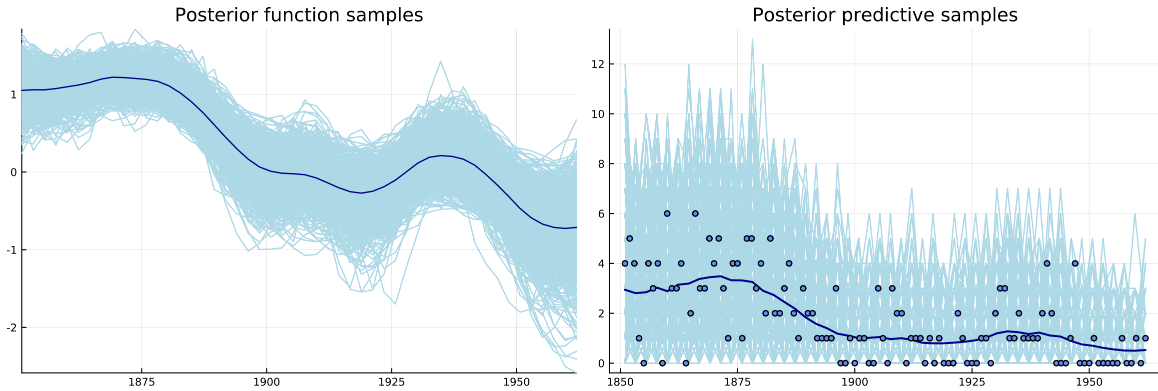


Figure 7: Samples from the posterior (left) and predictive distributions (right) of the Gaussian process with a Poisson observation model. The points show the recorded number of incidents and the dark blue lines represent the mean posterior and predictive functions, respectively.

fitted to these observations $\mathcal{D}_t = \{(\mathbf{x}_i, y_i)\}_{i=1,\dots,t}$ 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.

```
gp = ElasticGPE(2,                        # 2 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

append!(gp, [1., 2.], 0.4)   # append observation x = [1., 2.] and y = 0.4
GP Exact object:
  Dim = 2
  Number of observations = 2
  Mean function:
    Type: MeanConst, Params: [0.0]
  Kernel:
    Type: SEArd{Float64}, Params: [-0.0, -0.0, 5.0]
  Input observations =
```

```
[1.0 1.0; 2.0 2.0]
  Output observations = [0.4, 0.4]
  Variance of observation noise = 1.0
  Marginal Log-Likelihood = -7.184
```

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 two-dimensional objective function, $f(x) = \sum_{i=1}^{d}(x_i - 1)^2 + \epsilon$, where $\epsilon \sim \mathcal{N}(0,1)$, which is illustrated in Figure 8.

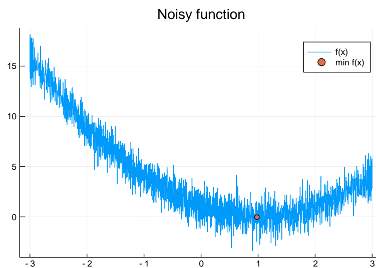


Figure 8: One-dimensional representation of $f(x) = \sum_{i=1}^{d}(x_i - 1)^2 + \epsilon$, where $\epsilon \sim \mathcal{N}(0,1)$.

```
using BayesianOptimization, GaussianProcesses

f(x) = sum((x .- 1).^2) + randn()            # noisy function to minimize

# Choose as a model an elastic GP with input dimensions 2.
model = ElasticGPE(2, mean = MeanConst(0.), kernel = SEArd([0., 0.], 5.), logNoise = 0.)


# Optimize the hyperparameters of the GP using maximum likelihood (ML) estimates every 50 steps
modeloptimizer = MLGPOptimizer(every = 50, noisebounds = [-4, 3], # bounds of the logNoise
                                kernbounds = [[-1, -1, 0], [4, 4, 10]], # bounds of the 3
                                    parameters GaussianProcesses.get_param_names(model.kernel)
                                maxeval = 40)
opt = BOpt(f,
           model,
           ExpectedImprovement(),              # type of acquisition function
           modeloptimizer,
           [-5., -5.], [5., 5.],               # lowerbounds, upperbounds
           maxiterations = 500,                # evaluate the objective function 500 times
           sense = Min,                        # minimize the objective function
           gradientfree = false,               # use gradient information
           verbosity = Silent)

result = boptimize!(opt)
(observerd_optimum = -2.002286162842888, observed_optimizer = [0.460792, 1.18039], model_optimum =
    -0.028814819133003766, model_optimizer = [1.00621, 0.952885])
```

With `gradientfree = false`, **BayesianOptimization.jl** uses automatic differentiation tools in **ForwardDiff.jl** to compute gradients of the acquisition function (`ExpectedImprovement()` in the example above). After convergence, we can see from Figure 8 that the Bayesian optimizer has located the function minima.

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 1: 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, 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 1, 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[4]. The following are package enhancements currently under development:

- Variational approximations - Currently the package uses MCMC for inference with non-Gaussian likelihoods. MCMC has good theoretical convergence properties, but can be slow for large data sets. Variational approximations Opper and Winther (2000) using, for example, mean-field, have been widely

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

used in the Gaussian process literature, and while not exact, can produce highly accurate posterior approximations.

- Sparse Gaussian processes - The computational cost of fitting a standard GP is $\mathcal{O}(n^3)$, which can be very expensive for large data sets. This has been a well-studied issue in the literature with a number of solutions proposed. Most solutions to the scalability problem of GPs have led to the development of *sparse GPs* (Quiñonero-Candela and Rasmussen, 2005), which reduce the computational cost to $\mathcal{O}(nm^2)$, where $m << n$ is the number of pseudo-input points.

- 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 function, which also needing to hand-code the corresponding derivatives.

References

Adams, R. P., Murray, I., and MacKay, D. J. C. (2009). Tractable nonparametric Bayesian inference in Poisson processes with Gaussian process intensities. *Proceedings of the 26th Annual International Conference on Machine Learning - ICML*, pages 9—-16.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98.

Carlin, B. P., Gelfand, A. E., and Smith, A. F. M. (1992). Hierarchical Bayesian Analysis of Changepoint Problems. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 41(2):389–405.

Deisenroth, M. P., Fox, D., and Rasmussen, C. E. (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., and 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, D. K. (2014). *Automatic Model Construction with Gaussian Processes*. PhD thesis, University of Cambridge.

Fearnhead, P. (2006). Exact and efficient Bayesian inference for multiple changepoint problems. *Statistics and Computing*, 16(2):203–213.

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

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

Gonzalez, J. P., Cook, S. E., Oberthür, T., Jarvis, A., Bagnell, J. A., and Dias, M. B. (2007). Creating low-cost soil maps for tropical agriculture using gaussian processes.

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

Hensman, J., Matthews, A., Filippone, M., and Ghahramani, Z. (2015). MCMC for Variationally Sparse Gaussian Processes. *arXiv preprint arXiv:1506.04000*.

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

Lloyd, C., Gunter, T., Osborne, M. A., and Roberts, S. J. (2015). Variational Inference for Gaussian Process Modulated Poisson Processes. In *International Conference on Machine Learning*, pages 1814—-1822.

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

Matthews, A. G. d. G., van der Wilk, M., Nickson, T., Fujii, K., Boukouvalas, A., León-Villagrá, P., Ghahramani, Z., and Hensman, J. (2017). GPflow: A Gaussian process library using TensorFlow. *Journal of Machine Learning Research*, 18(40):1–6.

Minka, T. P. (2001). *A family of algorithms for approximate bayesian inference*. PhD thesis.

Mogensen, P. K. and Riseth, A. N. (2018). Optim: A mathematical optimization package for Julia Usage in research and industry. *Journal of Open Source Software*, 3(24).

Murray, I. (2016). Differentiation of the Cholesky decomposition.

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

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

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

Opper, M. and Winther, O. (2000). Gaussian Processes for Classification: Mean Field Algorithms. In *Neural Computation 12(11)*, pages 2655–2684.

Osborne, M. A., Roberts, S. J., Rogers, A., and Jennings, N. R. (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. and Rasmussen, C. E. (2005). A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research*, 6:1935–1959.

Rasmussen, C. E. and Nickisch, H. (2010). Gaussian processes for machine learning (gpml) toolbox. *Journal of machine learning research*, 11(Nov):3011–3015.

Rasmussen, C. E. and Nickisch, H. (2017). Gpml v4.1. Matlab/Octave package. Last accessed June 2018.

Rasmussen, C. E. and Williams, C. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

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

Robert, C. P. (2004). *Monte carlo methods*. Wiley Online Library.

Roberts, G. O. and Rosenthal, J. S. (1998). Optimal scaling of discrete approximations to Langevin diffusions. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 60(1):255–268.

Roberts, G. O. and Rosenthal, J. S. (2004). General state space Markov chains and MCMC algorithms. *Probability Surveys*, 1(0):20–71.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and de Freitas, N. (2016). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148175.

Titsias, M. K., Lawrence, N., and Rattray, M. (2008). Markov chain monte carlo algorithms for gaussian processes. *Inference and Estimation in Probabilistic Time-Series Models*, 9.

Wang, L., Durante, D., Jung, R. E., and Dunson, D. B. (2017). Bayesian networkresponse regression. *Bioinformatics*, 33(12):1859–1866.

Williams, C. K. I. and Barber, D. (1998). Bayesian Classification with Gaussian Processes. *IEEE Trans Pattern Analysis and Machine Intelligence*, 20(12):1342–1351.

Wilson, A. G., Dann, C., and Nickisch, H. (2015). Thoughts on Massively Scalable Gaussian Processes. *ArXiv e-prints*.

A Available functions

Function	Description	$k_\theta(\mathbf{x}, \mathbf{x}^*) =$	$\boldsymbol{\theta}$
`Const`	Constant	σ^2	$\log \sigma$
`Lin`	Linear ARD	$\mathbf{x}^\top L^{-2} \mathbf{x}^*, \quad L = \mathrm{diag}(\ell_1, \ldots, \ell_d)$	$(\log \ell_1, \ldots, \log \ell_d)$
`Lin`	Linear Iso	$\mathbf{x}^\top \mathbf{x}^* / \ell^2$	$\log \ell$
`Matern(1/2,...)`	Matérn ARD (1/2)	$\sigma^2 \exp(-\lvert\mathbf{x} - \mathbf{x}^*\rvert / L), \quad L = \mathrm{diag}(\ell_1, \ldots, \ell_d)$	$(\log \ell_1, \ldots, \log \ell_d, \log \sigma)$
`Matern(3/2,...)`	Matérn ARD (3/2)	$\sigma^2 (1 + \sqrt{3} \lvert\mathbf{x} - \mathbf{x}^*\rvert / L) \exp(-\sqrt{3} \lvert\mathbf{x} - \mathbf{x}^*\rvert / L)$	$(\log \ell_1, \ldots, \log \ell_d, \log \sigma)$
`Matern(5/2,...)`	Matérn ARD (5/2)	$\sigma^2 (1 + \sqrt{5} \lvert\mathbf{x} - \mathbf{x}^*\rvert / L + 5 \lvert\mathbf{x} - \mathbf{x}^*\rvert^2 / 3L^2) \exp(-\sqrt{5} \lvert\mathbf{x} - \mathbf{x}^*\rvert / L)$	$(\log \ell_1, \ldots, \log \ell_d, \log \sigma)$
`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	$\sigma^2 \exp(-(\mathbf{x} - \mathbf{x}^*)^\top L^{-2} (\mathbf{x} - \mathbf{x}^*)/2) \quad L = \mathrm{diag}(\ell_1, \ldots, \ell_d)$	$(\log \ell_1, \ldots, \log \ell_d, \log \sigma)$
	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	$\sigma^2 (1 + (\mathbf{x} - \mathbf{x}^*)^\top L^{-2} (\mathbf{x} - \mathbf{x}^*)/2\alpha)^{-\alpha} \quad L = \mathrm{diag}(\ell_1, \ldots, \ell_d)$	$(\log \ell_1, \ldots, \log \ell_d, \log \sigma, \log \alpha)$
`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 $i \subseteq \{1, , \ldots, d\}$	$k_\theta(\mathbf{x}_i, \mathbf{x}_i^*)$	$\emptyset$
`*`	Product kernels	$\prod_i k_\theta(\mathbf{x}, \mathbf{x}^*)$	$\emptyset$
`+`	Sum kernels	$\sum_i k_\theta(\mathbf{x}, \mathbf{x}^*)$	$\emptyset$

Table 2: 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 3: 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 4: List of available mean functions