Probabilistic Programming with Gaussian Process Memoization

Ulrich Schaechtle ULRICH.SCHAECHTLE@RHUL.AC.UK
Department of Computer Science
Royal Holloway, University of London

Ben Zinberg BZINBERG@MIT.EDU
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Alexey Radul AXOFCH@GMAIL.COM
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Kostas Stathis KOSTAS.STATHIS@RHUL.AC.UK
Department of Computer Science
Royal Holloway, University of London

Vikash K. Mansinghka VKM@MIT.EDU
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Editor: N.A.

Abstract

This paper describes the *Gaussian process memoizer*, a probabilistic programming technique that uses Gaussian processes to provide a statistical alternative to memorization. Memoizing a target procedure results in a self-caching wrapper that remembers previously computed values. Gaussian process memoization additionally produces a statistical emulator based on a Gaussian process whose predictions automatically improve whenever a new value of the target procedure becomes available. This paper also introduces an efficient implementation, named `gpmem`, that can use kernels given by a broad class of probabilistic programs. The flexibility of `gpmem` is illustrated via three applications: (i) GP regression with hierarchical hyper-parameter learning, (ii) Bayesian structure learning via compositional kernels generated by a probabilistic grammar, and (iii) a bandit formulation of Bayesian optimization with automatic inference and action selection. All applications share a single 50-line Python library and require fewer than 20 lines of probabilistic code each.

1. Introduction

Gaussian Processes (GP) are widely used tools in statistics (Barry, 1986), machine learning (Neal, 1995; Williams and Barber, 1998; Kuss and Rasmussen, 2005; Rasmussen and Williams, 2006; Damianou and Lawrence, 2013), robotics (Ferris et al., 2006), computer vision (Kemmler et al., 2013), and scientific computation (Kennedy and O'Hagan, 2001; Schneider et al., 2008; Kwan et al., 2013). They are also central to probabilistic numerics,

an emerging effort to develop more computationally efficient numerical procedures, and to Bayesian optimization, a family of meta-optimization techniques that are widely used to tune parameters for deep learning algorithms (Snoek et al., 2012; Gelbart et al., 2014). They have even seen use in artificial intelligence; for example, they provide the key technology behind a project that produces qualitative natural language descriptions of time series (Duvenaud et al., 2013; Lloyd et al., 2014).

This paper describes Gaussian process memoization, a technique for integrating GPs into a probabilistic programming language, and demonstrates its utility by re-implementing and extending state-of-the-art applications of the GP. Memoization is a classic programming technique in which a procedure is augmented with an input-output cache that is checked each time before the function is invoked. This prevents unnecessary recomputation, potentially saving time at the cost of increased storage requirements. Gaussian process memoization, implemented by the `gpmem()` procedure, generalizes this idea to include a statistical emulator that uses previously computed values as data in a statistical model that can accurately forecast probable outputs. The covariance function for the Gaussian process is also allowed to be an arbitrary probabilistic program.

This paper presents three applications of gpmem: (i) a replication of results Neal (1997) on outlier rejection via hyper-parameter inference; (ii) a fully Bayesian extension to the Automated Statistician project; and (iii) an implementation of Bayesian optimization via Thompson sampling. The first application can in principle be replicated in several other probabilistic languages embedding the proposal that is described in this paper. The remaining two applications rely on distinctive capabilities of Venture (Mansinghka et al., 2014): support for fully Bayesian structure learning and language constructs for inference programming. All applications share a single 50-line Python library and require fewer than 20 lines of probabilistic code each.

1.1 Related Work

A recent review on artificial intelligence and machine learning in Natur named GPs and probabilistic programming as amongst the most promising directions for future research (Ghahramani, 2015). A practitioner can find many software packages for GPs available for free (e.g. Rasmussen and Nickisch, 2010; The GPy authors, 2012–2015; Neumann et al., 2015). However, the context that a practitioner can provide with them is mostly limited to regression and classification tasks. Yet, GPs performed well in applications that go much further than these tasks. Neal shows an interesting application of Bayesian regression in the presence of outliers with a hierarchical system of hyper-priors Neal (1997). GPs have been applied to emulated complex computation such as in the case of Cosmic Calibration Framework (CCF) (Kwan et al., 2013). At the heart of this project is the CCF that exploits accuracy of large scale high-resolution cosmological simulations that are computationally expensive. Here, a sophisticated sampling scheme provides an optimal sampling strategy for the cosmological models to be simulated. The measurements from the simulations are then translated into functions that can be easily interpolated with GPs.

The discovery of symbolic relations and concepts has also been the focus of attention of Bayesian non-parametrics (e.g. Kemp et al., 2006). The Automatic Statistician introduces inductive learning of symbolic expressions by kernel structure learning, through performing

a greedy search over the space of possible GP kernel structure (Duvenaud et al., 2013; Lloyd et al., 2014). The resulting structure comes with symbolic interpretation that can be understood by humans. The engineering involved a significant work-around for the GPML-toolbox (Rasmussen and Nickisch, 2010) which, whilst providing a novel solution, still struggles to fulfill requirements to be fully Bayesian.

A further example of advanced use of GPs beyond regression and classification is the seemingly deterministic domain of optimization. To be Bayesian over the values of expensive functions, GPs were used as emulators, leading to efficient optimization of otherwise expensive machine learning algorithms (Snoek et al., 2012). Bayesian optimization updates the belief system of a Bayesian agent when new information about the true underlying function to optimize becomes available. A sampling scheme for performing Bayesian Optimization in very high dimensions has been introduced by (Mahendran et al., 2012).

Probabilistic programming systems on the other hand have pushed the boundaries of models usable for machine learning with general purpose inference machinery. Examples include picture, which is combining fast data-driven methods for image interpretation such as deep neural networks with generative modelling (Kulkarni et al., 2015). Classification with large, deep and evolving hierarchical models as needed for the identification and tracking of resident space objects was enabled with the Figaro language (Ruttenberg et al., 2014). Nitti and colleagues present a probabilistic programming approach to planning with Markov Decision Processes (MDP)s in hybrid domains, that is discrete and continuous-valued domains as well as domains with an unknown number of objects (Nitti et al., 2015). Bayesian Program induction outperforms deep neural networks in tasks with only one training example provided (Lake and Salakhutdinov, 2015). This kind of learning is particularly hard for machine learning while at the same time considerably easy for humans.

2. Background on Gaussian Processes

GPs are a Bayesian method for regression. We consider the regression input to be real-valued scalars x_i and the regression output $f(x_i) = y_i$ as the value of a function f at x_i. The complete training data will be denoted by column vectors $\mathbf{x}$ and $\mathbf{y}$. Unseen test input is denoted with $\mathbf{x}'$. GPs present a non-parametric way to express prior knowledge on the space of all possible functions f modeling a regression relationship. Formally, a GP is an infinite-dimensional extension of the multivariate Gaussian distribution.

The collection of random variables $\{f(x_i) = y_i\}$ (indexed by i) represents the values of the function f at each location x_i. We write $f \sim \mathcal{GP}(\mu, k \mid \boldsymbol{\theta}_{\text{mean}}, \boldsymbol{\theta})$, where μ is the *mean function* and k is the *covariance function* or *kernel*. That is, $\mu(x_i \mid \boldsymbol{\theta}_{\text{mean}})$ is the prior mean of the random variable y_i, and $k(x_i, x_j \mid \boldsymbol{\theta})$ is the prior covariance of the random variables y_i and y_j. The output of both mean and covariance function are conditioned on a few free hyper-parameters parameterizing k and μ. We refer to these hyper-parameters as $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_{\text{mean}}$ respectively. To simplify the calculation below, we will assume the prior mean μ is identically zero; once the derivation is done, this assumption can be easily relaxed via translation. We use upper case italic $K(\mathbf{x}, \mathbf{x}' \mid \boldsymbol{\theta})$ for a function that returns a matrix of dimension $I \times J$ with entries $k(x_i, x_j \mid \boldsymbol{\theta})$ and with $x_i \in \mathbf{x}$ and $x_j \in \mathbf{x}'$ where I and J indicate the length of the column vectors $\mathbf{x}$ and $\mathbf{x}'$. Throughout, we write $\mathbf{K}_{(\boldsymbol{\theta},\mathbf{x},\mathbf{x}')}$ for the prior covariance matrix that results from computing $K(\mathbf{x}, \mathbf{x}' \mid \boldsymbol{\theta})$. In the following, we

will sometimes drop the subscript $\mathbf{x}, \mathbf{x}'$, writing only $\mathbf{K}_{\boldsymbol{\theta}}$, for clarity. Note that we do this only in cases when both input vectors are identical and correspond to training input $\mathbf{x}$. We differentiate two different situations leading to different ways samples can be generated with this setup:

1. $\mathbf{y}'$ - the predictive posterior sample from a distribution conditioned on observed input $\mathbf{x}$ with observed output $\mathbf{y}$ and conditioned on $\boldsymbol{\theta}$.

2. $\mathbf{y}^*$ - a sample from the predictive prior. We will describe situations, where the GP has not seen any data $\mathbf{x}, \mathbf{y}$ yet. In this case, we sample from a Gaussian distribution with $\mathbf{y}^* \sim \mathcal{N}\big(0, K(\mathbf{x}^*, \mathbf{x}^* \mid \boldsymbol{\theta})\big)$; where the symbol $*$ indicates that no data has been observed yet.

We now show how to compute the predictive posterior distribution of test output $\mathbf{y}' := f(\mathbf{x}')$ conditioned on training data $\mathbf{y} := f(\mathbf{x})$. (Here $\mathbf{x}$ and $\mathbf{x}'$ are known constant vectors, and we are conditioning on an observed value of $\mathbf{y}$.) The predictive posterior can be computed by first forming the joint density when both training and test data are treated as randomly chosen from the prior, then fixing the value of $\mathbf{y}$ to a constant. To start, let

$$\Sigma := \begin{bmatrix} K(\mathbf{x}, \mathbf{x} \mid \boldsymbol{\theta}) & K(\mathbf{x}, \mathbf{x}' \mid \boldsymbol{\theta}) \\ K(\mathbf{x}', \mathbf{x} \mid \boldsymbol{\theta}) & K(\mathbf{x}', \mathbf{x}' \mid \boldsymbol{\theta}) \end{bmatrix} \text{ and } \Sigma^{-1} =: \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{bmatrix}.$$

We then have

$$P(\mathbf{y}, \mathbf{y}' \mid \boldsymbol{\theta}) \propto \exp\left\{ -\frac{1}{2} \begin{bmatrix} \mathbf{y}^\top & \mathbf{y}'^\top \end{bmatrix} \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{y}' \end{bmatrix} \right\}.$$

Treating $\mathbf{y}$ as a fixed constant, we obtain

$$P\left(\mathbf{y}' \mid \mathbf{y}, \boldsymbol{\theta}\right) \propto P(\mathbf{y}, \mathbf{y}' \mid \boldsymbol{\theta}) \propto \exp\left\{ -\frac{1}{2} \mathbf{y}'^\top \mathbf{M}_{22} \mathbf{y}' - \mathbf{h}^\top \mathbf{y}' \right\},$$

where $\mathbf{h} = M_{21}\mathbf{y}$ is a constant vector. Thus $P(\mathbf{y}'|\mathbf{y}, \boldsymbol{\theta})$ is Gaussian,

$$P\left(\mathbf{y}' \mid \mathbf{y}, \boldsymbol{\theta}\right) \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}, \mathbf{K}_{\boldsymbol{\theta}}^{\text{post}}), \tag{1}$$

with covariance matrix $\mathbf{K}_{\boldsymbol{\theta}}^{\text{post}} = \mathbf{M}_{22}^{-1}$. To find its mean $\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}$, we note that $P_{\mathbf{y}'|\mathbf{y},\boldsymbol{\theta}}(\mathbf{y}' + \boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}})$ is Gaussian with the same covariance as $P(\mathbf{y}'|\mathbf{y}, \boldsymbol{\theta})$, but its exponent has no linear term:

$$\begin{aligned} P_{\mathbf{y}'|\mathbf{y},\boldsymbol{\theta}}\left(\mathbf{y}' + \boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}} \;\middle|\; \mathbf{y}, \boldsymbol{\theta}\right) &\propto \exp\left\{ -\frac{1}{2}(\mathbf{y}' + \boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}})^\top \mathbf{M}_{22}(\mathbf{y}' + \boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}) - \mathbf{h}^\top(\mathbf{y}' + \boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}) \right\} \\ &\propto \exp\left\{ -\frac{1}{2}\mathbf{y}'^\top \mathbf{M}_{22}\mathbf{y}' - \underbrace{(\mathbf{h} + \mathbf{M}_{22}\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}})^\top}_{\text{must be 0}} \mathbf{y}' \right\}. \end{aligned}$$

Thus $\mathbf{h} = -\mathbf{M}_{22}\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}$ and $\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}} = -\mathbf{M}_{22}^{-1}\mathbf{h} = -\mathbf{M}_{22}^{-1}\mathbf{M}_{21}\mathbf{y}$.

The partioned inverse equations (Barnett and Barnett, 1979 following MacKay, 1998) give

$$\begin{aligned} \mathbf{M}_{22} &= \big(K(\mathbf{x}', \mathbf{x}' \mid \boldsymbol{\theta}) - K(\mathbf{x}', \mathbf{x} \mid \boldsymbol{\theta}) K(\mathbf{x}, \mathbf{x} \mid \boldsymbol{\theta})^{-1} K(\mathbf{x}, \mathbf{x}' \mid \boldsymbol{\theta})\big)^{-1}, \\ \mathbf{M}_{21} &= -\mathbf{M}_{22} K(\mathbf{x}', \mathbf{x} \mid \boldsymbol{\theta}) K(\mathbf{x}, \mathbf{x} \mid \boldsymbol{\theta})^{-1}. \end{aligned}$$

Substituting these in the above gives

$$\mathbf{K}_{\boldsymbol{\theta}}^{\text{post}} = K(\mathbf{x}', \mathbf{x}' \mid \boldsymbol{\theta}) - K(\mathbf{x}', \mathbf{x} \mid \boldsymbol{\theta}) K(\mathbf{x}, \mathbf{x} \mid \boldsymbol{\theta})^{-1} K(\mathbf{x}, \mathbf{x}' \mid \boldsymbol{\theta}), \tag{2}$$

$$\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}} = K(\mathbf{x}', \mathbf{x} \mid \boldsymbol{\theta}) K(\mathbf{x}, \mathbf{x} \mid \boldsymbol{\theta})^{-1} \mathbf{y}. \tag{3}$$

Together, $\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}$ and $\mathbf{K}_{\boldsymbol{\theta}}^{\text{post}}$ determine the computation of the predictive posterior with unseen input data (1).

Often one assumes the observed regression output is noisily measured, that is, one only sees the values of $\mathbf{y}_{\text{noisy}} = \mathbf{y} + \mathbf{w}$ where $\mathbf{w}$ is Gaussian white noise with variance σ^2_{noise}. This noise term can be absorbed into the covariance function $K(\mathbf{x}, \mathbf{x} \mid \boldsymbol{\theta})$. The log-likelihood of a GP can then be written as:

$$\log P(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}) = -\frac{1}{2} \mathbf{y}^\top \mathbf{K}_{\boldsymbol{\theta}}^{-1} \mathbf{y} - \frac{1}{2} \log |\mathbf{K}_{\boldsymbol{\theta}}| - \frac{n}{2} \log 2\pi \tag{4}$$

where n is the number of data points. Both log-likelihood and predictive posterior can be computed efficiently using a Stochastic Process (SP) in Venture (Mansinghka et al., 2014) with an algorithm that resorts to Cholesky factorization(Rasmussen and Williams, 2006, chap. 2). We write the Cholesky factorization as $\mathbf{L} \coloneqq \text{chol}(\mathbf{K}_{\boldsymbol{\theta}})$ when :

$$\mathbf{K}_{\boldsymbol{\theta}} = \mathbf{L}\mathbf{L}^\top \tag{5}$$

where $\mathbf{L}$ is a lower triangular matrix. This allows us to compute the inverse of a covariance matrix as

$$\mathbf{K}_{\boldsymbol{\theta}}^{-1} = (\mathbf{L}^{-1})^\top (\mathbf{L}^{-1}) \tag{6}$$

and its determinant as

$$det(\mathbf{K}_{\boldsymbol{\theta}}) = det(\mathbf{L})^2 \tag{7}$$

We compute (4) as

$$\log(P(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}) \coloneqq -\frac{1}{2} \mathbf{y}^\top \boldsymbol{\alpha} - \sum_i \log \mathbf{L}_{ii} - \frac{n}{2} \log 2\pi \tag{8}$$

where

$$\mathbf{L} \coloneqq \text{chol}(\mathbf{K}_{\boldsymbol{\theta}}) \tag{9}$$

and

$$\boldsymbol{\alpha} \coloneqq \mathbf{L}^\top \backslash (\mathbf{L} \backslash \mathbf{y}). \tag{10}$$

This results in a computational complexity for sampling in the number of data points of $O(n^3/6)$ for (9) an $O(n^2/2)$ for (10).

Above, we defined the GP prior as $\mathbf{y}^* \sim \mathcal{N}\big(0, K(\mathbf{x}_*, \mathbf{x}_* \mid \boldsymbol{\theta})\big)$. We see that this prior is fully determined by its covariance function.

2.1 Covariance Functions

The covariance function (or kernel) of a GP governs high-level properties of the observed data such as smoothness or linearity. The high-level properties are indicated with superscript on functions. A linear covariance can be written as:

$$k^{\text{linear}} = \sigma_1^2(xx'). \tag{11}$$

We can also express periodicity:

$$k^{\text{periodic}} = \sigma_2^2 \exp\left(\frac{2\sin^2(\pi(x-x')/p}{\ell^2}\right). \tag{12}$$

By changing these properties we get completely different prior behavior for sampling $\mathbf{y}^*$ from a GP with a linear kernel

$$\mathbf{y}^* \sim \mathcal{N}\big(0, K^{\text{linear}}(\mathbf{x}^*, \mathbf{x}^* \mid \sigma_1)\big)$$

as compared to sampling from the prior predictive with a periodic kernel (as depicted in Fig. 1 (c) and (d))

$$\mathbf{y}^* \sim \mathcal{N}\big(0, K^{\text{periodic}}(\mathbf{x}^*, \mathbf{x}^* \mid \sigma_2, \ell)\big).$$

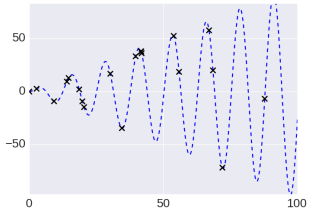
These high-level properties are compositional via addition and multiplication of different covariance functions. That means that we can also combine these properties. By using multiplication of kernels we can model a local interaction of two components, for example

$$k^{\text{linear}} \times k^{\text{periodic}} = \sigma_1^2(xx')\,\sigma_2^2 \exp\left(\frac{2\sin^2(\pi(x-x')/p}{\ell^2}\right) \tag{13}$$

This results in a combination of the higher level properties of linearity and periodicity. In Fig 1 (e) we depict samples for $\mathbf{y}^*$ that are periodic with linearly increasing amplitude. We consider this a local interaction because the actual interaction depends on the similarity of two data points. An addition of covariance functions models a global interaction, that is an interaction of two high-level components that is qualitatively not dependent on the input space. An example for this a periodic function with a linear trend.

For each kernel type, each $\boldsymbol{\theta}$ is different, that is, in (11) we have $\boldsymbol{\theta} = \{\sigma_1\}$, in (12) we have $\boldsymbol{\theta} = \{\sigma_2, p, \ell\}$ and in (13) we have $\boldsymbol{\theta} = \{\sigma_1, \sigma_2, p, \ell\}$. Adjusting these hyper-parameters changes lower level qualitative attributes such as length scales (ℓ) while preserving the higher level qualitative properties of the distribution such as linearity.

If we choose suitable set of hyper-parameters, for example by performing inference, we can capture the underlying dynamics of the data well (see Fig. 1 (f-h)) while sampling $\mathbf{y}'$. Note that goodness of fit is not only limited to the parameters. A too simple qualitative structure implies unsuitable behaviour, as for example in (Fig. 1 (g)) where additional recurring spikes are introduced to account for the changing amplitude of the true function that generated the data.

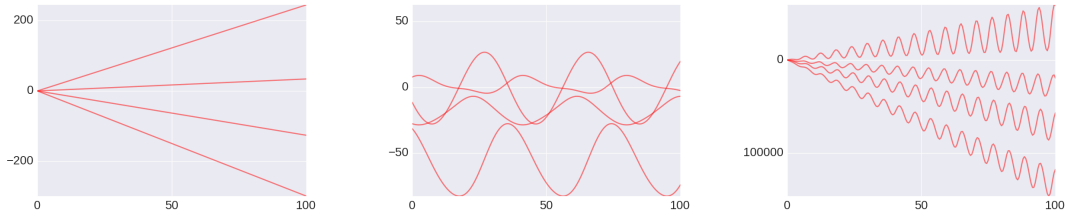


(a) Raw Data

$$k^{\text{linear}} = \sigma_1^2(xx')$$
$$k^{\text{periodic}} = \sigma_2^2 \exp\left(\frac{2\sin^2(\pi(x-x')/p}{\ell^2}\right)$$
$$k^{\text{linear}} \times k^{\text{periodic}} = \sigma_1^2(xx')\,\sigma_2^2 \exp\left(\frac{2\sin^2(\pi(x-x')/p}{\ell^2}\right)$$

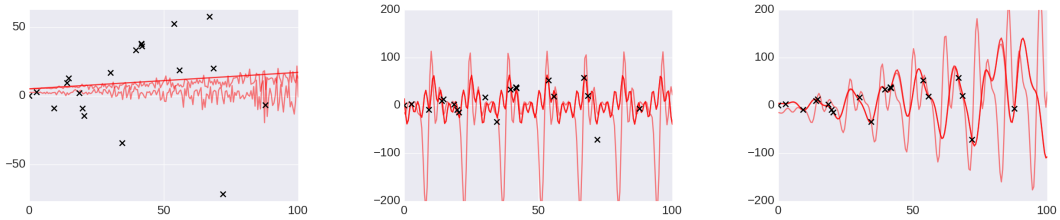
(b) Kernels

Prior predictive, $\mathbf{y}^* \sim \mathcal{N}\big(0, K(\mathbf{x}^*, \mathbf{x}^* \mid \boldsymbol{\theta})\big)$:



(c) K^{linear} (d) K^{periodic} (e) $K^{\text{linear}} \times K^{\text{periodic}}$

Posterior predictive $\mathbf{y}' \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}, \mathbf{K}_{\boldsymbol{\theta}}^{\text{post}})$:



(f) K^{linear} (g) K^{periodic} (h) $K^{\text{linear}} \times K^{\text{periodic}}$

Figure 1: We depict kernel composition. (a) shows raw data (black) generated with a sine function with linearly growing amplitude (blue). This data is used for all the plots (c-h). (b) shows the linear and the periodic base kernel in functional form as well as a composition of both. The multiplication of the two kernels indicates local interaction. The local interaction we account for in this case is the growing amplitude (a). For each column (c-h) $\boldsymbol{\theta}$ is different.(c-e) show samples from the prior prior predictive $\mathbf{y}^*$ where random parameters are used, that is, we sample before any data points are observed. (f-h) show samples from the predictive posterior $\mathbf{y}'$, after the data has been observed.

3. Gaussian Process Memoization in Venture

Memoization is the practice of storing previously computed values of a function so that future calls with the same inputs can be evaluated by lookup rather than re-computation. To transfer this idea to probabilistic programming, we now introduce a language construct called a *statistical memoizer*. Suppose we have a function `f` which can be evaluated but we wish to learn about the behavior of `f` using as few evaluations as possible. The statistical memoizer, which here we give the name `gpmem`, was motivated by this purpose. It produces two outputs:

$$\mathtt{f} \xrightarrow{\mathtt{gpmem}} (\mathtt{f}_{\text{compute}}, \mathtt{f}_{\text{emu}}).$$

The function $\mathtt{f}_{compute}$ calls `f` and stores the output in a memo table, just as traditional memoization does. The function $\mathtt{f}_{\text{emu}}$ is an online statistical emulator which uses the memo table as its training data. A fully Bayesian emulator, modelling the true function `f` as a random function $f \sim P(f)$, would satisfy

$$(\mathtt{f}_{\text{emu}}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_k) \sim P\left(f(\mathtt{x}_1), \ldots, f(\mathtt{x}_k) \mid f(\mathtt{x}) = (\mathtt{f}\ \mathtt{x}) \text{ for each } \mathtt{x} \text{ in memo table}\right).$$

Different implementations of the statistical memoizer can have different prior distributions $P(f)$; in this paper, we deploy a GP prior (implemented as `gpmem` below). Note that we require the ability to sample $\mathtt{f}_{\text{emu}}$ jointly at multiple inputs because the values of $f(\mathtt{x}_1), \ldots, f(\mathtt{x}_k)$ will in general be dependent.

We explain how `gpmem`, the statistical memoizer with GP-prior, works using a simple tutorial (Fig. 2). The top panel (Fig. 2, (a)) of this figure sketches the schematic of `gpmem`. `f` is the external process that we memoize. It can be evaluated using resources that potentially come from outside of Venture. We feed this function into `gpmem` alongside a parameterised kernel k. In this example, we make the qualitative assumption of f being smooth, and define k to be a squared-exponential covariance function:

$$k = k^{\text{se}} = \sigma^2 \exp(-\frac{(x - x')^2}{2\ell^2}).$$

The hyper-parameters $\boldsymbol{\theta}$ for this kernel are sampled from a prior distribution which is depicted in the top right box. Note that we annotate $\boldsymbol{\theta} = \{\mathtt{sf}, \mathtt{l}\}$ for subsequent inference as belonging to (i) the scope "hyper-parameter" and (ii) blocks 0 and 1 respectively.

`gpmem` implements a memoization table, where all previously computed function evaluations ($\{\mathbf{x}, \mathbf{y}\}$) are stored. We also initialize a GP-prior that will serve as our statistical emulator:

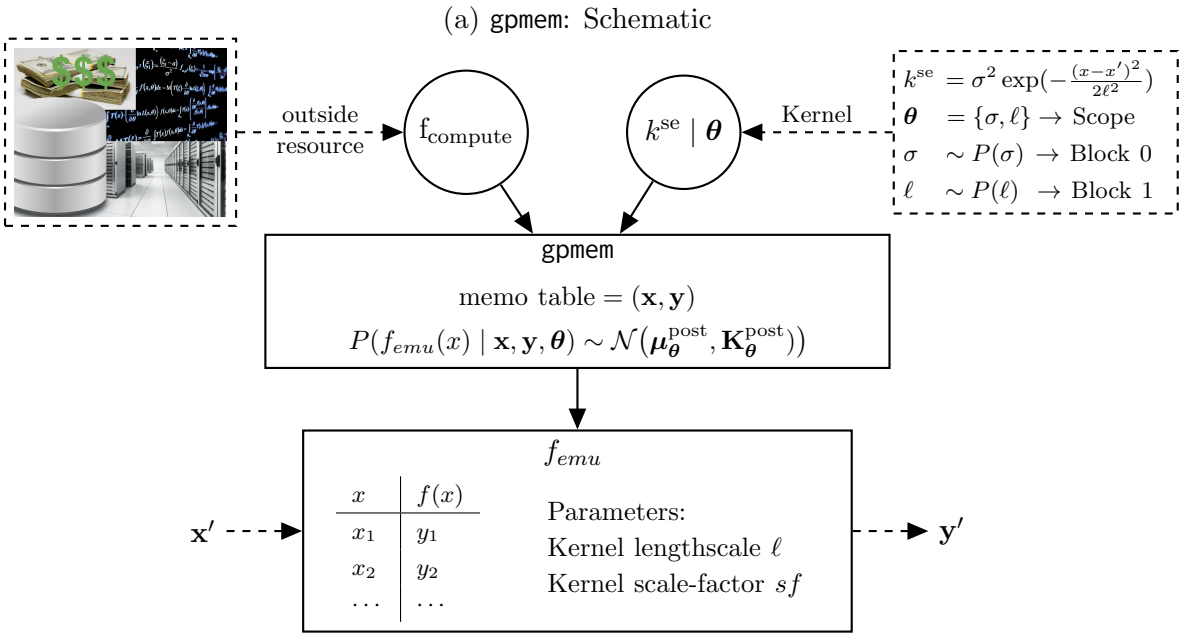
$$P(f_{emu}(x) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \sim \mathcal{N}\big(\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}, \mathbf{K}_{\boldsymbol{\theta}}^{\text{post}})\big)$$

where

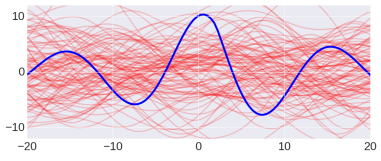
$$P(f_{emu}(x) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \mathbf{y}'$$

under the traditional GP perspective. All value pairs stored in the memoization table (memo table = $(\mathbf{x}, \mathbf{y})$) are incorporated as observations of the GP. We simply feed the regression input into the emulator and output a predictive posterior Gaussian distribution determined by the GP and the memoization table.

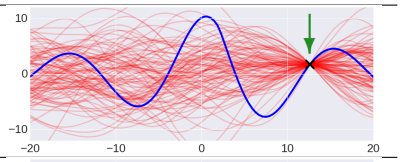
We can either define the function f that serves as as input for `gpmem` natively in Venture (as shown in the Fig. 2 (b)) or we interleave Venture with foreign code. This can be useful



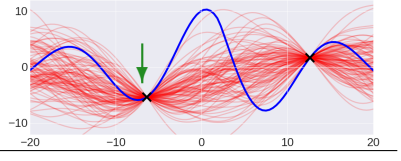
(b)
```
define f = proc(x) {
  exp(-0.1*abs(x-2)) * 10 * cos(0.4*x) + 0.2};
assume (f_compute, f_emu) = gpmem(f, kse);
sample f_emu(array(-20, ···, 20));
```



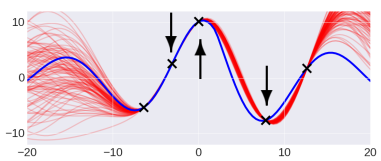
(c)
```
predict f_compute(12.6);
sample f_emu(array(-20, ···, 20));
```



(d)
```
predict f_compute(-6.4);
sample f_emu(array(-20, ···, 20));
```



(e)
```
observe f_emu(-3.1) =  2.60;
observe f_emu(7.8)  = -7.60;
observe f_emu(0.0)  = 10.19;
sample f_emu(array(-20, ···, 20));
```



(f)
```
infer mh(scope="hyper-parameter", block=one, steps=50);

sample f_emu(array(-20, ···, 20));
```

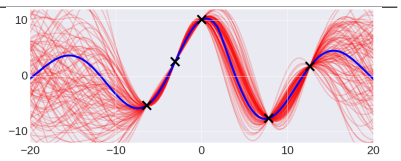


Figure 2: gpmem tutorial. The top shows a schematic of gpmem. f_compute probes an outside resource. This can be expensive (top left). Every probe is memoized and improves the emulator. Below the schematic we see the evolution of gpmem's state of believe of the world given certain Venture directives. On the right, we depict the true function (blue), samples from the emulator (red) and incorporated observations (black). We omid code for the hyper-parameters and kse, it can be found in Appendix D.

when `f` is computed with the help of outside resources. Before making any observations or calls to `f` we can sample from the prior at the inputs from -20 to 20 using the emulator:

```
assume (f_compute, f_emu) = gpmem(f, kse));

sample f_emu(array(-20, ..., 20));
```

where the second line corresponds to:

$$\mathbf{y}^* \sim \mathcal{N}\left(0, K^{\text{se}}\left(\begin{bmatrix}-20\\ \cdots \\ 20\end{bmatrix}, \begin{bmatrix}-20\\ \cdots \\ 20\end{bmatrix} \mid \boldsymbol{\theta} = \{\sigma, \ell\}\right)\right).$$

In Fig. 2 (c), we probe the external function `f` at point 12.6 and memoize it's result by calling

```
predict f_compute (12.6);
```

When we subsequently sample from the emulator, that is compute the $\mathbf{y}'$ at the input $\mathbf{x}' = \left[-20, \cdots, 20\right]^\top$. We see how the posterior which shifts from uncertainty to near certainty close to the input 12.6.

We can repeat the process at a different point (probing point -6.4 in Fig. 2 (d)) to see that we gain certainty about another part of the curve.

We can add information to $\texttt{f}_{\text{emu}}$ about presumable value pairs of `f` without calling $\texttt{f}_{\text{compute}}$ (Fig. 2 (e)). If a friend tells us the value of `f` we can call observe to store this information in the incorporated observations for $\texttt{f}_{\text{emu}}$ only:

```
observe f_emu( -3.1) = 2.60;
```

We have this value pair now available for the computation $\mathbf{y}'$. For sampling with the emulator, the effect is the same as calling predict with the $\texttt{f}_{\text{compute}}$. However, we can imagine at least one scenario where such as distinction in the treatment of observations is beneficial. Let us say we do not only have the real function available but also a domain expert with knowledge about this function. This expert could tell us what the value is at a given input. Potentially, the value provided by the expert could disagree with the value computed with `f` for example due to different levels of observation noise.

Finally, we can update our posterior by inferring the posterior over hyper-parameter values $\boldsymbol{\theta}$. For this we use the defined scopes, which denote a collection of related random choices, such as all hyper-parameters $\boldsymbol{\theta}$. Scopes may be further subdivided into blocks, on which block proposals can be made. In the program above we assign two blocks, 0 and 1 for each hyper-parameter. These tags are supplied to the inference program (in this case, MH) to specify on which random variables inference should be done:

```
infer mh(scope="hyper-parameters", block=one, steps=50);
```

In this case, we perform one Metropolis-Hastings (MH) transition over the scope hyper-parameters and choose a random block, that is we choose one hyper-parameter at random. We can also define custom inference actions. Let's define MH with Gaussian drift proposals.

```
define drift_kernel = proc(x) { normal(x, 1) };

define my_markov_chain =
mh_correct(
    on_subproblem(scope="hyper-parameters", block=2,
        symmetric_local_proposal(drift_kernel)));

infer my_markov_chain;
```

Note that this inference is not in the Figure. The important part of the above code snippet is `drift_kernel`, which is where we say that at each step of our Markov chain, we would like to propose a transition by sampling a new state from a unit normal distribution whose mean is the current state. We apply this inference action only for `block` 2 in the `scope` "hyper-parameters".

The newly inferred hyper-parameters allow us now adequately reflect uncertainty about the curve given all incorporated observations (compare Fig. 2, bottom panel (f) on the right with the samples before inference, one panel above (e)).

4. Applications

This paper illustrates the flexibility of `gpmem` by showing how it can concisely encode three different applications of GPs. The first is a standard example from hierarchical Bayesian statistics, where Bayesian inference over a hierarchical hyper-prior is used to provide a curve-fitting methodology that is robust to outliers. The second is a structure learning application from probabilistic artificial intelligence, where GPs are used to discover qualitative structure in time series data. The third is a reinforcement learning application, where GPs are used as part of a Thompson sampling formulation of Bayesian optimization for general real-valued objective functions with real inputs.

4.1 Nonlinear regression in the presence of outliers

We can apply `gpmem` for regression in a hierarchical Bayesian setting (Fig. 3). In a Bayesian treatment of hyper-parameter learning for GPs, we can write the posterior probability of the hyper-parameters of a GP (Fig. 3, (a)) given covariance function $\boldsymbol{\mathcal{K}} = k$ as:

$$P(\boldsymbol{\theta} = \{sf, \ell, \sigma\} \mid \mathbf{D}, \boldsymbol{\mathcal{K}}) = \frac{P(\mathbf{D} \mid \boldsymbol{\theta}, \boldsymbol{\mathcal{K}}) P(\boldsymbol{\theta} \mid \boldsymbol{\mathcal{K}})}{P(\mathbf{D} \mid \boldsymbol{\mathcal{K}})} \tag{14}$$

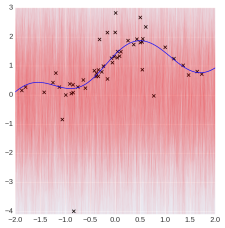
where $\mathbf{D} = \{\mathbf{x}, \mathbf{y}\}$ is a training data set and $\boldsymbol{\mathcal{K}}$ is treated as a random variable over covariance functions. Since we can apply `gpmem` to any process or procedure, it can be used in situations where only a data set is available via a look-up function `f_look_up`. In fact, we demonstrate `gpmem`'s application to regression using an example where the data was generated by a function which is not available, that is, we do not provide the synthetic function to `gpmem`

(a) $P(\boldsymbol{\theta} = \{\ell, sf, \sigma\} \mid \mathbf{D}, \boldsymbol{\mathcal{K}})$

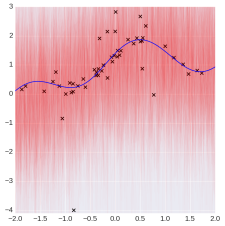
(b)
```
// Data and look-up function
define data = array(array(-1.87, 0.13), ..., array(1.67, 0.81));
assume f_look_up = proc(index) {lookup(data, index)};
```

(c)
```
assume sf = tag(scope="hyper", block=0, gamma(alpha_sf, beta_sf)));
assume l = tag(scope="hyper", block=1, gamma(alpha_l, beta_l)));
assume sigma = tag(scope="hyper", block=2, uniform_continuous(0, 2));
```

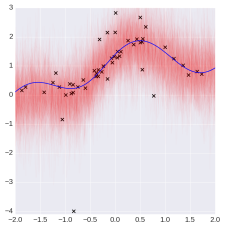
(d)
```
// The covariance function
assume se = make_squaredexp(sf, l);
assume wn = make_whitenoise(sigma);
assume composite_covariance = add_funcs(se, wn);
// Create a prober and emulator using gpmem
assume (f_compute, f_emu) =
    gpmem(f_look_up, composite_covariance);
sample f_emu(array(-2, ···, 2));
```



(e)
```
// Observe all data points
for (n = 0; n < size(data); n++){
    observe f_emu(first(lookup(data,n))) =
        second(lookup(data,n))};
// Or: probe all data points
for (n = 0; n < size(data); n++){
    predict f_compute(first(lookup(data,n)))};
sample f_emu(array(-2, ···, 2));
```



(f)
```
// Metropolis-Hastings
infer repeat(100, do(
    mh(scope="hyperhyper",block=one, steps=2),
    mh(scope="hyper",     block=one, steps=1)));
sample f_emu(array(-2, ···, 2));
```



(g)
```
// Optimization
infer gradient-ascent(scope="hyper",
                      block=all, steps=10);

sample f_emu(array(-2, ···, 2));
```

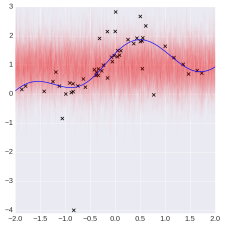


Figure 3: Regression with outliers and hierarchical prior structure. The code for the hyper-priors is omited here but provided in Appendix D

but only a data set (Fig. 3 (b)). This function, f_{true}, is taken from a paper on the treatment of outliers with hierarchical Bayesian hyper-priors for GPs (Neal, 1997):

$$f_{\text{true}}(x) = 0.3 + 0.4x + 0.5\sin(2.7x) + \frac{1.1}{(1+x^2)} + \eta \quad with \;\; \eta \sim \mathcal{N}(0, \sigma_{\text{noise}}). \tag{15}$$

We synthetically generate outliers by setting $\sigma_{\text{noise}} = 0.1$ in 95% of the cases and to $\sigma_{\text{noise}} = 1$ in the remaining cases. Instead of accessing the f_{true} directly, we are accessing the `data` in form of a a two dimensional `array` with `f_look_up`.

We set $\boldsymbol{\mathcal{K}} = k^{\text{se+wn}}$ and parameterize it with $\boldsymbol{\theta} = \{sf, \ell, \sigma\}$. For these hyper-parameters, Neals work suggests a hierarchical system for for hyper-parameterization. Here, we draw hyper-parameters from Γ distributions:

$$\ell \sim \Gamma(\alpha_1, \beta_1), \;\; \sigma \sim \Gamma(\alpha_2, \beta_2) \tag{16}$$

and in turn sample the α and β from Γ distributions as well:

$$\alpha_1 \sim \Gamma(\alpha_\alpha^1, \beta_\alpha^1), \;\; \alpha_2 \sim \Gamma(\alpha_\alpha^2, \beta_\alpha^2), \cdots \tag{17}$$

We model this in Venture as illustrated in Fig. 3 (c), using the build-in SP `gamma`. Note that we omit the prior distributions of (17) due limited space in the tutorial.

In Fig. 3, panel (d), we see that $k^{\text{se+wn}}$ is defined as a composite covariance function. It is the sum (`add_funcs`) of an squared exponential kernel (`make_squaredexp`) and a white noise (k^{wn}, Appendix A) kernel which is implemented with `make_whitenoise`[1]. We then initialize `gpmem` feeding it with `composite_covariance` and the data look-up function `f_look_up`. We sample from the prior $\mathbf{y}^*$ with random parameters `sf,l` and `sigma` and without any observations available. We depict those samples on the right (red), alongside the true function that generated the data (blue) and the data points we have available in the data set (black).

We can incorporate observations using both `observe` and `predict` (Fig. 3 (e)). When we subsequently sample $\mathbf{y}'$ from the emulator with $\mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}, \mathbf{K}_{\boldsymbol{\theta}}^{\text{post}})$, we can see that the GP posterior incorporates knowledge about the `data`. Yet, the hyper-parameters `sf,l` and `sigma` are still random, so the emulator does not capture the true underlying dynamics (f_{true}) of the `data` correctly.

Next, we demonstrate how we can capture these underlying dynamics within only 100 nested MH steps on the hyper-parameters to get a good approximation for their posterior $\mathbf{y}'$ (Fig. 3 (f)). We say nested because we first take two sweeps in the scope `hyperhyper` which characterizes (17) and then one sweep on the scope `hyper` which characterizes (16). This is repeated 100 using `repeat( 100, do(` $\cdots$. Note that Neal devises an additional noise model and performs a large number of Hybrid-Monte Carlo and Gibbs steps to achieve this, whereas inference in Venture with `gpmem` is merely one line of code.

Finally, we can change our inference strategy altogether. If we decide that instead of following a Bayesian sampling approach, we would like to perform empirical optimization, we do this by only changing one line of code, deploying `gradient-ascent` instead of `mh` (Fig. 3 (g)).

1. Note that in Neal's work (1997) the sum of an SE plus a constant kernel is used. We use a WN kernel for illustrative purposes instead.

4.2 Discovering qualitative structure from time series data

Inductive learning of symbolic expressions for continuous-valued time series data is a hard task which has recently been tackled using a greedy search over the approximate posterior of the possible kernel compositions for GPs (Duvenaud et al., 2013; Lloyd et al., 2014)[2].

With `gpmem` we can provide a fully Bayesian treatment of this, previously unavaible, using a stochastic grammar (see Fig. 4). This allows us to read an unstructured time series and automatically output a high-level, qualitative description of it. The stochastic grammar takes a set of primitive base kernels $\text{BK} = \{k^1_{\boldsymbol{\theta}^1}, \cdots, k^m_{\boldsymbol{\theta}^m}\}$ of size m, including their corresponding parametrization $\boldsymbol{\theta}^* = \{\boldsymbol{\theta}^1, \cdots, \boldsymbol{\theta}^m\}$ (Fig. 4 (a) and (b)) We depict the input for the stochastic grammar in Listing 1.

Listing 1: Initialize Base Kernels BK and $P(n)$

```
// Initialize hyper-parameters
assume theta_1 = tag(scope="hyper-parameters", block=1, gamma(5,1));
assume theta_2 = tag(scope="hyper-parameters", block=2, gamma(5,1));
assume theta_3 = tag(scope="hyper-parameters", block=3, gamma(5,1));
assume theta_4 = tag(scope="hyper-parameters", block=4, gamma(5,1));
assume theta_5 = tag(scope="hyper-parameters", block=5, gamma(5,1));
assume theta_6 = tag(scope="hyper-parameters", block=6, gamma(5,1));
assume theta_7 = tag(scope="hyper-parameters", block=7, gamma(5,1));

// Make kernels
assume lin = apply_function(make_linear, theta_1);
assume per = apply_function(make_periodic, theta_2, theta_3, theta_4);
assume se  = apply_function(make_squaredexp, theta_5, theta_6);
assume wn  = apply_function(make_noise, theta_7);

// Initialize the set of primitive base kernels BK
assume BK = list(lin, per, se, wn);
```

We sample a random subset S of the set of supplied base kernels. S is of size $n \leq m$. We write

$$\mathbf{S} = \{K^i_{\boldsymbol{\theta}^i}, \cdots, K^n_{\boldsymbol{\theta}^n}\} \sim P(\mathbf{S} = \{K^i_{\boldsymbol{\theta}^i}, \cdots, K^n_{\boldsymbol{\theta}^n}\} \mid \text{BK})$$

with

$$P(\mathbf{S} = \{K^i_{\boldsymbol{\theta}^i}, \cdots, K^n_{\boldsymbol{\theta}^n}\} \mid \text{BK}) = \frac{n!}{\mid \mathbf{S} = \{K^i_{\boldsymbol{\theta}^i}, \cdots, K^n_{\boldsymbol{\theta}^n}\} \mid !}.$$

BK is assumed to be fixed as the most general margin of our hypothesis space. In the following, we will drop it in the notation. The only building block that we are now missing is how to combine the sampled base kernels into a compositional covariance function (see Fig. 4 (b)). For each interaction i, we have to infer whether the data supports a local interaction or a global interaction, chosing between one out of two algebraic operators $\boldsymbol{\Omega}_i = \{+, \times\}$. The probability for all such decisions is given by a binomial distribution:

$$P(\boldsymbol{\Omega} \mid \mathbf{S}) = \binom{n}{r} p^r_{+\times}(1 - p_{+\times})^{n-r}. \tag{18}$$

2. http://www.automaticstatistician.com/

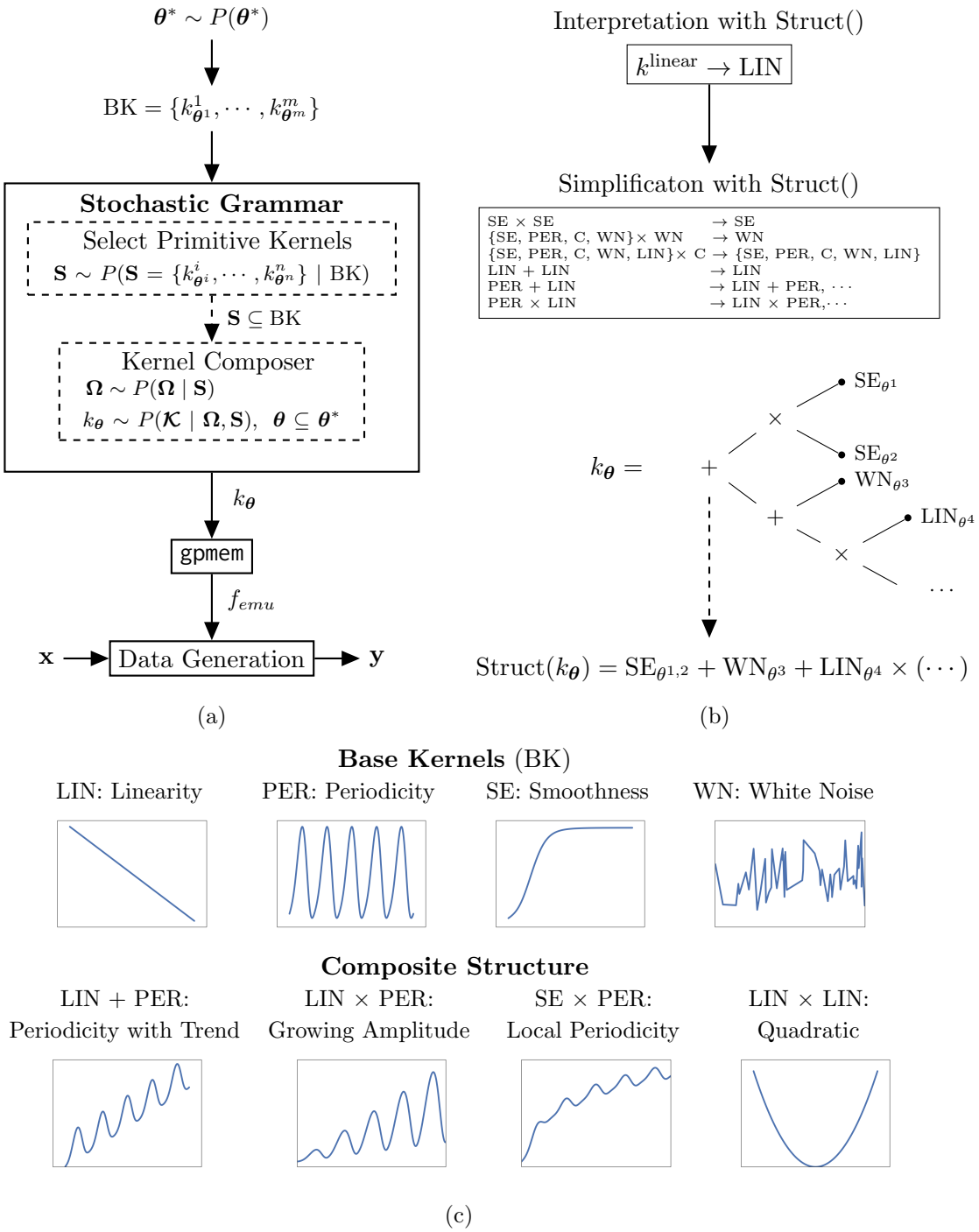


Figure 4: (a) Bayesian GP structure learning. A set of base kernels (BK) with priors on there hyper-parameters serve as hypothesis space and is supplied as input to the stochastic grammar. The stochastic grammar has two parts: (i) a sampler that selects a random set $\mathbf{S}$ of primitive kernels from BK and (ii) a kernel composer that combines the individual base kernels and generates a composite kernel function $k_{\boldsymbol{\theta}}$. This serves as input for `gpmem`. We observe value pairs $\mathbf{x}, \mathbf{y}$ of unstructured time series data on the bottom of the schematic. (b) An example of composite kernel structure. Struct$(k_{\boldsymbol{\theta}})$ takes as input a kernel $k_{\boldsymbol{\theta}}$ as sampled in (a) and interprets it symbolically and simplifies it. Rules for simplification can be found in Appendix B. We use addition and multiplication to combine base kernels. Base kernels and compositional kernels are shown in (c) alongside ther interpretation with Struct().

We can write the marginal probability of a kernel function as

$$P(\mathcal{K} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \iint_{\boldsymbol{\Omega}, \mathbf{S}} P(\mathcal{K} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}, \boldsymbol{\Omega}, \mathbf{S}) \times P(\boldsymbol{\Omega} \mid \mathbf{S}) \times P(\mathbf{S}) \, \mathbf{d}\boldsymbol{\Omega} \, \mathbf{dS} \tag{19}$$

with $\boldsymbol{\theta} \subseteq \boldsymbol{\theta}^*$ as implied by S. For structure learning with GP kernels, a composite kernel is sampled from $P(\mathcal{K})$ and fed into gpmem. The emulator generated by gpmem observes unstructured time series data. Venture code for the probabilistic grammar is shown in Listing 2, code for inference with gpmem in Listing 3.

Listing 2: Stochastic Grammar

```
// Select a random subset of the set possible primitive kernels (BK)
assume primitive_kernel_selection = tag( scope="grammar", block=0,
                                         select_primitive_kernels(BK));
// Construct kernel composition with a composer procedure
assume kernel_composer = proc(l) {
  if (size(l) <= 1) {
    first(l)
  } else {
      if (bernoulli()) {
          add_funcs(first(l),  kernel_composer(rest(l)))
      } else {
          mult_funcs(first(l), kernel_composer(rest(l)))
    }
  }
};

assume K = tag( scope="grammar", block=1,
                kernel_composer(primitive_kernel_selection));
```

Listing 3: gpmem inference for structure learning:

```
// Apply gpmem
assume (f_compute, f_emu) = gpmem(f_look_up, K);
// Probe all data points
for (n = 0; n < size(data); n++) {
      predict f_compute(first(lookup(data, n)))};
// Perform inference
infer repeat(200, do(
      mh(scope="grammar", block=one, steps=1),
      mh(scope="hyper-parameters", block=one, steps=2)));
```

Many equivalent covariance structures can be sampled due to covariance function algebra and equivalent representations with different parameterization (Lloyd et al., 2014). To inspect the posterior of these equivalent structures we convert each kernel expression into a sum of products and subsequently simplify. We introduce three different operators that work on kernel functions:

1. Simplify(k); this operators simplifies a kernel function k according to the simplifications that we present in Appendix B and Fig. 4 (b).
2. Struct(k); interprets the structure of a covariance function, for example Struct(k^{linear}) = LIN; and
3. Parse(k), an operator that parses a covariance function.

All base kernels can be found in Appendix A, rules for this simplification can be found in Appendix B. Kernel structure comes with symbolic interpretations. To read kernel function k and apply the simplifications described above, we apply, Struct(k). For examle, we write

$$\text{Struct}(k^{\text{linear}}) = \text{LIN},$$

which translates a function into a symbolic expression, see Appendix C. We defined a simple space of covariance structures in a way that allows us to produce results coherent with work presented in Automatic Statistician. The results are illustrated with two data sets.

Mauna Loa CO_2 data

We illustrate results in Fig 5. In Fig 5 (a) we depict the raw data. We see mean centered CO_2 measurements of the Mauna Loa Observatory, an atmospheric baseline station on Mauna Loa, on the island of Hawaii. A description of the data set can be found in Rasmussen and Williams, 2006, chapter 5. We use those raw data to compute a posterior on structure, parameters and GP samples. The latter are shown in Fig 5 (b) where we zoom in to show how the posterior captures the error bars adequately. This posterior of the GP is generated with a random sample from the parameters of the peak of the distribution on structure (Fig 5 (c)). We differentiate between a posterior distribution on kernel functions and on distribution on symbolic expressions describing different kernel structures. This allows us to compute the posterior of symbollically equivalent structures, such as Struct($k^{\text{linear}} + k^{\text{periodic}}$) = Struct($k^{\text{periodic}} + k^{\text{linear}}$). Both structures yield and addition of a linear kernel and a periodic kernel, that is LIN + PER. We coin the random variable over symbolic expressions for kernels as $\boldsymbol{\mathcal{K}}_{\text{Struct}}$ which we compute with Simplify(). The distribution peaks at:

$$\boldsymbol{\mathcal{K}}_{\text{Struct}} = \text{LIN} + \text{PER} + \text{SE} + \text{WN}. \tag{20}$$

We write this kernel equation out in Fig 5 (d). This kernel structure has a natural language interpretation that we spell out in Fig 5 (e), explaining that the posterior peaks at a kernel structure with four additive components. Each of which holds globally, that is there are no higher level, qualitative aspects of the data that vary with the input space. The additive components for this result are as follows:

- a linearly increasing function or trend;
- a periodic function;
- a smooth function; and
- white noise.

Previous work on automated kernel discovery (Duvenaud et al., 2013) illustrated the Mauna Loa data using an RQ kernel. We resort to the white noise kernel instead of RQ (similar to (Lloyd et al., 2014)).

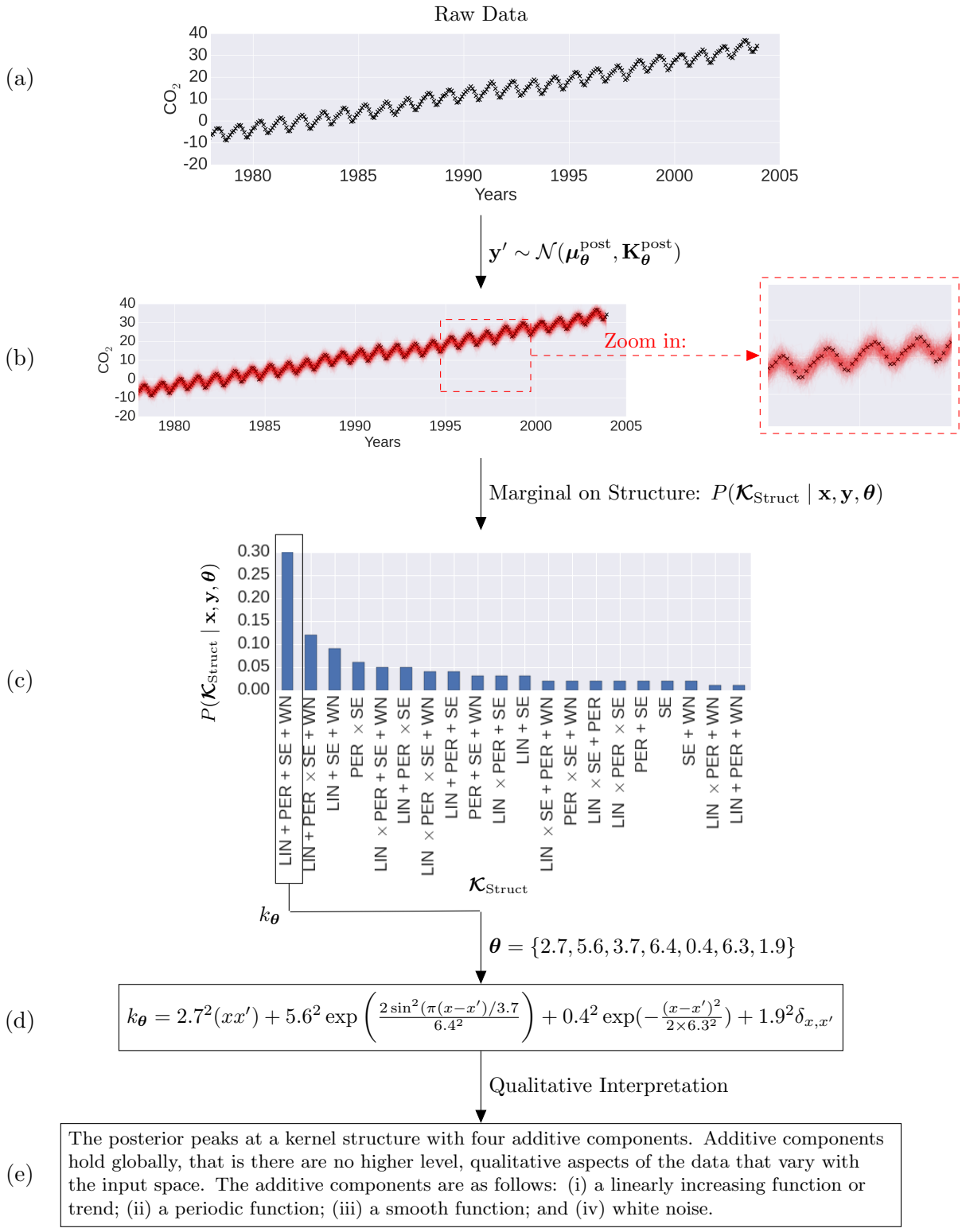


Figure 5: Structure Learning. Starting with raw data (a), we fit a GP (b) and compute the posterior distribution on structures (c). We take a sample of the peak of this distribution (LIN + PER + SE + WN) including its parameters and write it in functional form (d). We depict the human readable interpretation (e). We used (d) to plot (b).

Airline Data

The second data set (Fig. 6) we depict results for is the airline data set describing monthly totals of international airline passengers (Box et al., 1997, according to Duvenaud et al., 2013).

We illustrate results for this data set in Fig 6. In Fig 6 (a) we depict the raw data. Again, the data is mean centered and we use it to compute a posterior on structure, parameters and GP samples. The latter are shown in Fig 6 (b). This posterior of the GP is generated with a random sample from the parameters of the peak of the distribution on structure (Fig 6 (c)). The posterior over symbolic kernel expressions peaks at:

$$\boldsymbol{\mathcal{K}}_{\text{Struct}} = \text{LIN} + \text{SE} \times \text{PER} + \text{WN}. \tag{21}$$

We write this Kernel equation out in Fig 6 (d). This kernel structure has a natural language interpretation that we spell out in Fig 6 (e), explaining that the posterior peaks at a kernel structure with three additive components. Additive components hold globally, that is there are no higher level, qualitative aspects of the data that vary with the input space. The additive components are as follows:

- a linearly increasing function or trend;
- a approximate periodic function; and
- white noise.

Both datasets served as illustrations in the Automatic Statistician project.

Querying time series

With our Bayesian approach to structure learning we can gain valuable insights into time series data that were previously unavailable. This is due to our ability to estimate posterior marginal probabilities over the kernel structure. Over this marginal, we define boolean search operations that allow us to query the data for the probability of certain structures to hold true globally.

$$P(\boldsymbol{\mathcal{K}}_{\text{Struct}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T} f(\boldsymbol{\mathcal{K}}^t_{\text{Struct}}) \;\text{ where } f(\boldsymbol{\mathcal{K}}^t_{\text{Struct}}) = \begin{cases} 1, & \text{if } \boldsymbol{\mathcal{K}}_{\text{Struct}} \underset{global}{\in} \boldsymbol{\mathcal{K}}^t_{\text{Struct}}, \\ 0, & \text{otherwise.} \end{cases} \tag{22}$$

to ask whether it is true that a global structure $\boldsymbol{\mathcal{K}}_{\text{Struct}}$ is present. T is the number of all posterior samples for $\boldsymbol{\mathcal{K}}_{\text{Struct}}$ and $\boldsymbol{\mathcal{K}}^t_{\text{Struct}}$ is one such sample. We can now ask simple questions, for example:

Is there white noise in the data?

where we set $\boldsymbol{\mathcal{K}}_{\text{Struct}}$ =WN in (22). We can also formulate more sophisticated search operations using Boolean operators such as AND ($\wedge$) and OR ($\vee$). The AND operator is defined as follows:

$$P(\boldsymbol{\mathcal{K}}^a_{\text{Struct}} \wedge \boldsymbol{\mathcal{K}}^b_{\text{Struct}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{\mathcal{K}}^t_{\text{Struct}})$$

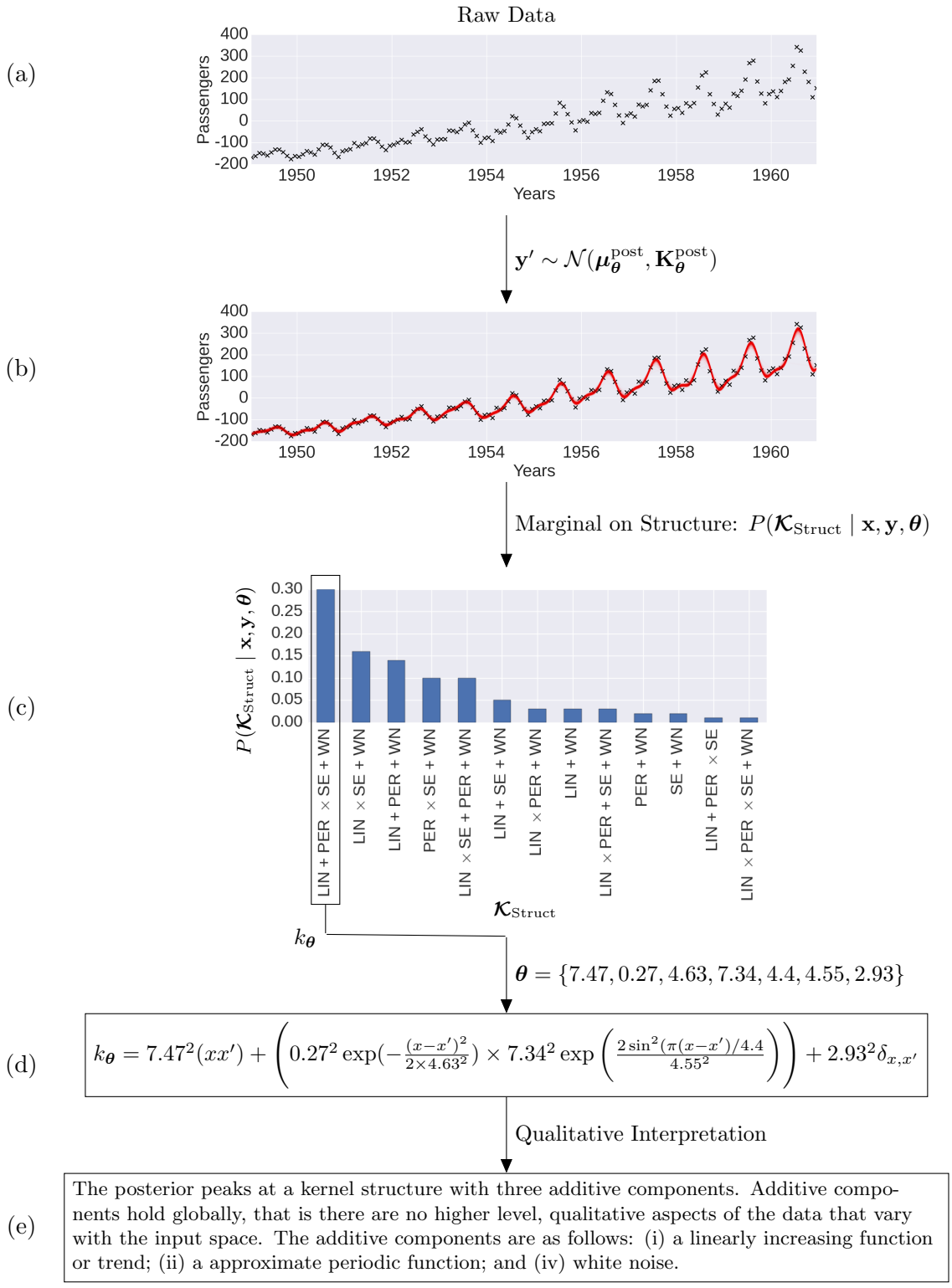


Figure 6: Structure Learning. Starting with raw data (a), we fit a GP (b) and compute the posterior distribution on structures (c). We take a sample of the peak of this distribution (LIN + PER × SE + WN) including its parameters and write it in functional form (d). We depict the human readable interpretation (e). We used (d) to plot (b).

where

$$f(\boldsymbol{\mathcal{K}}^t_{\text{Struct}}) = \begin{cases} 1, & \text{if } \boldsymbol{\mathcal{K}}^a_{\text{Struct}} \text{ and } \boldsymbol{\mathcal{K}}^b_{\text{Struct}} \underset{global}{\in} \boldsymbol{\mathcal{K}}^t_{\text{Struct}}, \\ 0, & \text{otherwise} \end{cases}.$$

By estimating $P(\text{LIN} \wedge \text{WN} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta})$ we can use this operator to ask questions such as

Is there a Linear component AND a white noise in the data?

Finally, we define the logical OR as

$$\begin{aligned} P(\boldsymbol{\mathcal{K}}^a_{\text{Struct}} \vee \boldsymbol{\mathcal{K}}^b_{\text{Struct}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = & P(\boldsymbol{\mathcal{K}}^a_{\text{Struct}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) + P(\boldsymbol{\mathcal{K}}^b_{\text{Struct}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \\ & - P(\boldsymbol{\mathcal{K}}^a_{\text{Struct}} \wedge \boldsymbol{\mathcal{K}}^b_{\text{Struct}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \end{aligned}$$

which allows us to ask questions about structures that are logically connected with OR, such as:

Is there white noise or heteroskedastic noise?

by estimating $P(\text{LIN} \times \text{WN} \;\vee\; \text{WN} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta})$. We know that noise can either be heteroskedastic or white, and we also know due to simple manipulations using kernel algebra that LIN $\times$ WN and WN are the only possible ways to construct noise with kernel composition, we see that we can generalize the question above to:

Is there noise in the data?

where we write the marginal posterior on qualitative structure for noise:

$$P(\boldsymbol{\mathcal{K}}^{\text{noise}}_{\text{Struct}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = P(\text{LIN} \times \text{WN} \;\vee\; \text{WN} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}). \tag{23}$$

Note that this allows us to start with general queries and subsequently formulate follow up queries that go into more detail. For example, we could start with a general query, such as:

What is the probability of a trend, a recurring pattern **and** noise in the data?

and then follow up with more detailed questions (Fig 7). This way of querying your data for their statistical implications is in stark contrast to what previous research in automatic kernel construction was able to provide. We could view our approach as a time series search engine which allows us to test whether or not certain structures can be found in an available time series. Another way to view this approach is as a new language to interact with the world. Real-world observations often come with time-stamps and in form of continuous valued sensor measurements. We provide the toolbox to query such observations in a similar manner as one would query a knowledge base in a logic programming language.

4.3 Bayesian optimization

The final application demonstrating the power of `gpmem` illustrates its use in Bayesian optimization. We introduce Thompson sampling, the basic solution strategy underlying the Bayesian optimization with `gpmem`. Thompson sampling Thompson (1933) is a widely used Bayesian framework for addressing the trade-off between exploration and exploitation in

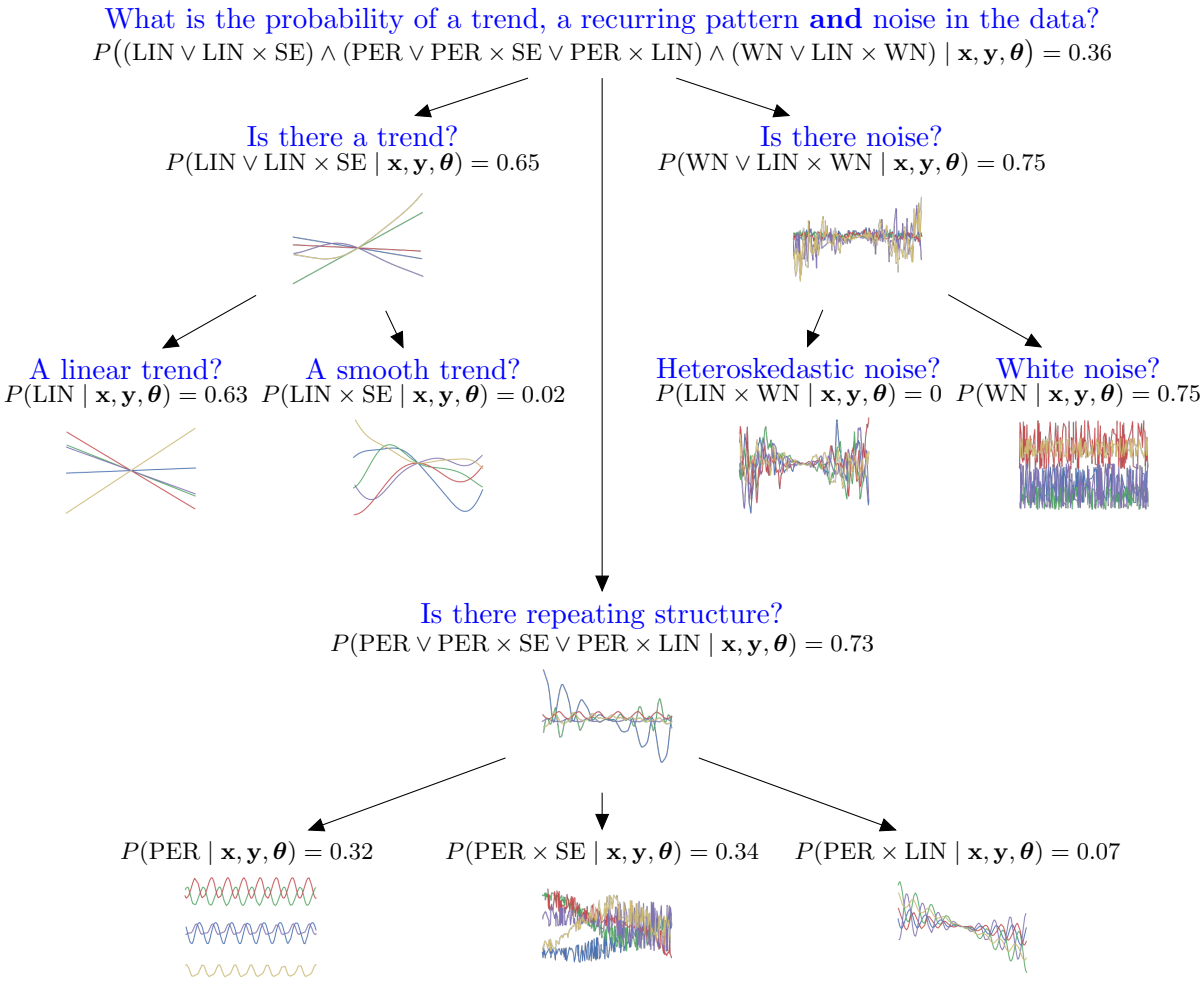


Figure 7: **Querying structural motifs in in time series using posterior inference over kernel structure.** The kernel structure serves as a way to formulate natural language questions about the data (blue). The initial question of interest (top) is a fairly general one: "What is the probability of a trend, a recurring pattern and noise in the data?" Below the natural language version of this question, the same question is formulated as an inference problem (black) over the marginal probability on kernels with Boolean operators AND ($\wedge$) and OR ($\vee$). To gain a deeper understanding of specific motifs in the time series more specific queries can be written. On the right, a query asks whether there is noise in the data (blue) by computing the disjunction of the marginal of a global white noise kernel and a multiplication between a linear and a white noise kernel (black). Samples from the predictive prior $\mathbf{y}_*$ of such kernels give an indication of the qualitative aspects that a kernel structure implies (coloured curves below the marginal). If the probability that there is noise in the data is high then it makes sense to drill even deeper asking more detailed questions. With regards to noise, this translates to querying whether or not the data supports the hypothesis that there is heteroskedastic noise or white noise. Queries for motifs of repeating structure are shown in the middle of the tree, queries related to trends on the left.

multi-armed (or continuum-armed) bandit problems. We cast the multi-armed bandit problem as a one-state Markov decision process (MDP), and describe how Thompson sampling can be used to choose actions for that MDP.

The MDP is as follows: An agent is to take a sequence of actions $x_1, x_2, \ldots$ from a (possibly infinite) set of possible actions $\mathcal{X}$. After each action, a reward $y \in \mathbb{R}$ is received, according to an unknown conditional distribution $P_{\text{true}}(y \mid x)$. The agent's goal is to maximize the total reward received for all actions in an online manner. In Thompson sampling, the agent accomplishes this by placing a prior distribution $P(\vartheta)$ on the possible "contexts" $\vartheta \in \Theta$. Here a context is a believed model of the conditional distributions $\{P(x \mid y)\}_{x \in \mathcal{X}}$, or at least, a believed statistic of these conditional distributions which is sufficient for deciding an action x. If actions are chosen so as to maximize expected reward, then one such sufficient statistic is the believed conditional mean $V(x \mid \vartheta) = \mathbb{E}[y \mid x; \vartheta]$, which can be viewed as a believed value function. For consistency with what follows, we will assume our context ϑ takes the form $(\boldsymbol{\theta}, \mathbf{x}, \mathbf{y})$ where $\mathbf{x}$ is the vector of past actions, $\mathbf{y}$ is the vector of their rewards, and $\boldsymbol{\theta}$ (the "semicontext") contains any other information that is included in the context.

In this setup, Thompson sampling has the following steps:

Algorithm 1 Thompson sampling.

Repeat as long as desired:

1. **Sample.** Sample a semicontext $\boldsymbol{\theta} \sim P(\boldsymbol{\theta})$.
2. **Search (and act).** Choose an action $x \in \mathcal{X}$ which (approximately) maximizes $V(x \mid \vartheta) = \mathbb{E}[y \mid x; \vartheta] = \mathbb{E}[y \mid x; \boldsymbol{\theta}, \mathbf{x}, \mathbf{y}]$.
3. **Update.** Let y_{true} be the reward received for action x. Update the believed distribution on $\boldsymbol{\theta}$, i.e., $P(\boldsymbol{\theta}) \leftarrow P_{\text{new}}(\boldsymbol{\theta})$ where $P_{\text{new}}(\boldsymbol{\theta}) = P(\boldsymbol{\theta} \mid x \mapsto y_{\text{true}})$.

Note that when $\mathbb{E}[y|x; \vartheta]$ (under the sampled value of $\boldsymbol{\theta}$ for some points x) is far from the true value $\mathbb{E}_{P_{\text{true}}}[y \mid x]$, the chosen action x may be far from optimal, but the information gained by probing action x will improve the belief ϑ. This amounts to "exploration." When $\mathbb{E}[y \mid x; \vartheta]$ is close to the true value except at points x for which $\mathbb{E}[y \mid x; \vartheta]$ is low, exploration will be less likely to occur, but the chosen actions x will tend to receive high rewards. This amounts to "exploitation." The trade-off between exploration and exploitation is illustrated in Figure 8. Roughly speaking, exploration will happen until the context ϑ is reasonably sure that the unexplored actions are probably not optimal, at which time the Thompson sampler will exploit by choosing actions in regions it knows to have high value.

Typically, when Thompson sampling is implemented, the search over contexts $\vartheta \in \Theta$ is limited by the choice of representation. In traditional programming environments, $\boldsymbol{\theta}$ often consists of a few numerical parameters for a family of distributions of a fixed functional form. With work, a mixture of a few functional forms is possible; but without probabilistic programming machinery, implementing a rich context space Θ would be an unworkably large technical burden. In a probabilstic programing language, however, the representation

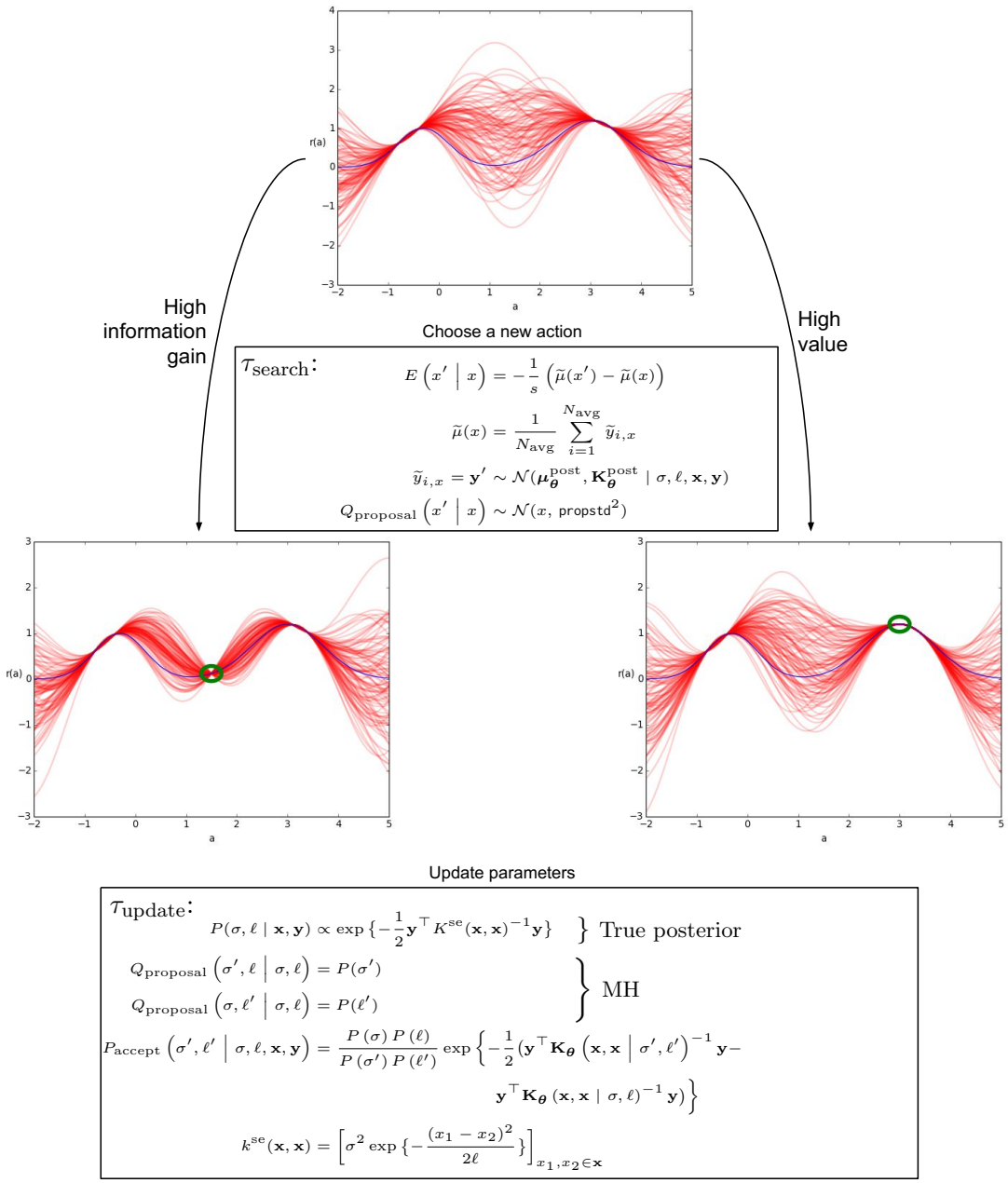


Figure 8: Two possible actions (in green) for an iteration of Thompson sampling. The believed distribution on the value function V is depicted in red. In this example, the true reward function is deterministic, and is drawn in blue. The action on the right receives a high reward, while the action on the left receives a low reward but greatly improves the accuracy of the believed distribution on V. The transition operators τ_{search} and τ_{update} are described in Section 4.3.

of heterogeneously structured or infinite-dimensional context spaces is quite natural. Any computable model of the conditional distributions $\{P(y \mid x)\}_{x \in \mathcal{X}}$ can be represented as a stochastic procedure $(\lambda(x) \ldots)$. Thus, for computational Thompson sampling, the most general context space $\widehat{\Theta}$ is the space of program texts. Any other context space Θ has a natural embedding as a subset of $\widehat{\Theta}$.

A Mathematical Specification

We now describe a particular case of Thompson sampling with the following properties:

- The regression function has a Gaussian process prior.
- The actions $x_1, x_2, \ldots \in \mathcal{X}$ are chosen by a Metropolis-like search strategy with Gaussian drift proposals.
- The hyperparameters of the Gaussian process are inferred using Metropolis–Hastings sampling after each action.

In this version of Thompson sampling, the contexts ϑ are Gaussian processes over the action space $\mathcal{X} = [-20, 20] \subseteq \mathbb{R}$. That is,

$$V \sim \mathcal{GP}(\mu, k),$$

where the mean μ is a computable function $\mathcal{X} \to \mathbb{R}$ and the covariance k is a computable (symmetric, positive-semidefinite) function $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$. This represents a Gaussian process $\{R_a\}_{a \in \mathcal{X}}$, where R_x represents the reward for action x. We write past actions as $\mathbf{x}$ and past rewards as $\mathbf{y}$. Computationally, we represent a context as a data structure

$$\vartheta = (\boldsymbol{\theta}, \mathbf{x}, \mathbf{y}) = (\mu, k, \boldsymbol{\theta}, \mathbf{x}, \mathbf{y}),$$

where $\boldsymbol{\mu}$ is a procedure to be used as the prior mean function and k is a procedure to be used as the prior covariance function, parameterized by $\boldsymbol{\theta}$. As above set $\mu \equiv 0$.

Note that the context space Θ is not a finite-dimensional parametric family, since the vectors $\mathbf{x}$ and $\mathbf{y}$ grow as more samples are taken. Θ is, however, representable as a computational procedure together with parameters and past samples, as we do in the representation $\vartheta = (\mu, k, \boldsymbol{\theta}, \mathbf{x}, \mathbf{y})$.

We combine the Update and Sample steps of Algorithm 1 by running a Metropolis–Hastings (MH) sampler whose stationary distribution is the posterior $P(\boldsymbol{\theta} \mid \mathbf{x}, \mathbf{y})$. The functional forms of μ and k are fixed in our case, so inference is only done over the parameters $\boldsymbol{\theta} = \{\sigma, \ell\}$; hence we equivalently write $P(\sigma, \ell \mid \mathbf{x}, \mathbf{y})$ for the stationary distribution. We make MH proposals to one variable at a time, using the prior as proposal distribution:

$$Q_{\text{proposal}}\left(\sigma', \ell \;\middle|\; \sigma, \ell\right) = P(\sigma')$$

and

$$Q_{\text{proposal}}\left(\sigma, \ell' \;\middle|\; \sigma, \ell\right) = P(\ell').$$

The MH acceptance probability for such a proposal is

$$P_{\text{accept}}\left(\sigma', \ell' \;\middle|\; \sigma, \ell\right) = \min\left\{1,\ \frac{Q_{\text{proposal}}\left(\sigma, \ell \mid \sigma', \ell'\right)}{Q_{\text{proposal}}\left(\sigma', \ell' \mid \sigma, \ell\right)} \cdot \frac{P(\mathbf{x}, \mathbf{y} \mid \sigma', \ell')}{P(\mathbf{x}, \mathbf{y} \mid \sigma, \ell)}\right\}$$

Because the priors on σ and ℓ are uniform in our case, the term involving Q_{proposal} equals 1 and we have simply

$$\begin{aligned} P_{\text{accept}}\left(\sigma', \ell' \mid \sigma, \ell\right) &= \min\left\{1,\ \frac{P\left(\mathbf{x}, \mathbf{y} \mid \sigma', \ell'\right)}{P\left(\mathbf{x}, \mathbf{y} \mid \sigma, \ell\right)}\right\} \\ &= \min\left\{1,\ \exp\left(-\frac{1}{2}\left(\mathbf{y}^T K\left(\mathbf{x}, \mathbf{x} \mid \sigma', \ell'\right)^{-1}\mathbf{y}\right.\right.\right. \\ &\qquad \left.\left.\left. - \mathbf{y}^T K\left(\mathbf{x}, \mathbf{x} \mid \sigma, \ell\right)^{-1}\mathbf{y}\right)\right)\right\}. \end{aligned}$$

The proposal and acceptance/rejection process described above define a transition operator τ_{update} which is iterated a specified number of times; the resulting state of the MH Markov chain is taken as the sampled semicontext $\boldsymbol{\theta}$ in Step 1 of Algorithm 1.

For Step 2 (Search) of Thompson sampling, we explore the action space using an MH-like transition operator τ_{search}. As in MH, each iteration of τ_{search} produces a proposal which is either accepted or rejected, and the state of this Markov chain after a specified number of steps is the new action x. The Markov chain's initial state is the most recent action, and the proposal distribution is Gaussian drift:

$$Q_{\text{proposal}}\left(x' \mid x\right) \sim \mathcal{N}(x, \texttt{propstd}^2),$$

where the drift width `propstd` is specified ahead of time. The acceptance probability of such a proposal is

$$P_{\text{accept}}\left(x' \mid x\right) = \min\left\{1,\ \exp\left(-E\left(x' \mid x\right)\right)\right\},$$

where the energy function $E\left(\bullet \mid a\right)$ is given by a Monte Carlo estimate of the difference in value from the current action:

$$E\left(x' \mid x\right) = -\frac{1}{s}\left(\widetilde{\mu}(x') - \widetilde{\mu}(x)\right)$$

where

$$\widetilde{\mu}(x) = \frac{1}{N_{\text{avg}}}\sum_{i=1}^{N_{\text{avg}}} \widetilde{y}_{i,x}$$

and

$$\widetilde{y}_{i,x} = \mathbf{y}' \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}, \mathbf{K}_{\boldsymbol{\theta}}^{\text{post}})$$

and $\{\widetilde{y}_{i,x}\}_{i=1}^{N_{\text{avg}}}$ are i.i.d. for a fixed x. (In the above, $\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}$ and $\mathbf{K}_{\boldsymbol{\theta}}^{\text{post}}$ are the mean and variance of a posterior sample at the single point $\mathbf{x}' = (x')$.) Here the temperature parameter $s \geq 0$ and the population size N_{avg} are specified ahead of time. Proposals of estimated value higher than that of the current action are always accepted, while proposals of estimated value lower than that of the current action are accepted with a probability that decays exponentially with respect to the difference in value. The rate of the decay is determined by the temperature parameter s, where high temperature corresponds to generous acceptance probabilities. For $s = 0$, all proposals of lower value are rejected; for $s = \infty$, all proposals are accepted. For points x at which the posterior mean $\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}$ is low but the posterior

variance $\mathbf{K}_{\boldsymbol{\theta}}^{\text{post}}$ is high, it is possible (especially when N_{avg} is small) to draw a "wild" value of $\widetilde{\mu}(x)$, resulting in a favorable acceptance probability.

Indeed, taking an action x with low estimated value but high uncertainty serves the useful function of improving the accuracy of the estimated value function at points near x (see Figure 8).[3,4] We see a comlete probabilistic program with `gpmem` implementing Bayesian optimization with Thompson Sampling below (Listing 4).

Listing 4: Bayesian optimization using `gpmem`

```
assume sf = tag(scope="hyper", block=0, uniform_continuous(0, 10));
assume l = tag(scope="hyper", block=1, uniform_continuous(0, 10));
assume se = make_squaredexp(sf, l);
assume blackbox_f = get_bayesopt_blackbox();
assume (f_compute, f_emulate) = gpmem(blackbox_f, se);

// A naive estimate of the argmax of the given function
define mc_argmax = proc(func) {
  candidate_xs = mapv(proc(i) {uniform_continuous(-20, 20)},
                      arange(20));
  candidate_ys = mapv(func, candidate_xs);
  lookup(candidate_xs, argmax_of_array(candidate_ys))
};

// Shortcut to sample the emulator at a single point without packing
// and unpacking arrays
define emulate_pointwise = proc(x) {
  run(sample(lookup(f_emulate(array(unquote(x))), 0)))
};

// Main inference loop
infer repeat(15, do(pass,
  // Probe V at the point mc_argmax(emulate_pointwise)
  predict(f_compute(unquote(mc_argmax(emulate_pointwise)))),
  // Infer hyperparameters
  mh(scope="hyper", block=one, steps=50)));
```

In Fig. 9 we show results for our implementation of Bayesian Optimization with Thompson sampling. We compare two different proposal distributions, namely uniform proposals and Gaussian drift proposals. We see that in this experiment, Gaussian drift is starting near the global optimum and drifts quickly towards it. (red curve, top panel of Fig. 9). Uniform proposals take longer to find the global optimum (blue curve, top panel of Fig. 9) but we see that it can surpass the local optima of the curve[5]. The bottom panel of Fig. 9 depicts

3. At least, this is true when we use a smoothing prior covariance function such as the squared exponential.
4. For this reason, we consider the sensitivity of $\widehat{\mu}$ to uncertainty to be a desirable property; indeed, this is why we use $\widehat{\mu}$ rather than the exact posterior mean μ.
5. In fact, repeated experiments have shown that when the Gaussian drift proposals starts near a local optimum, it gets stuck there. Uniform proposals do not.

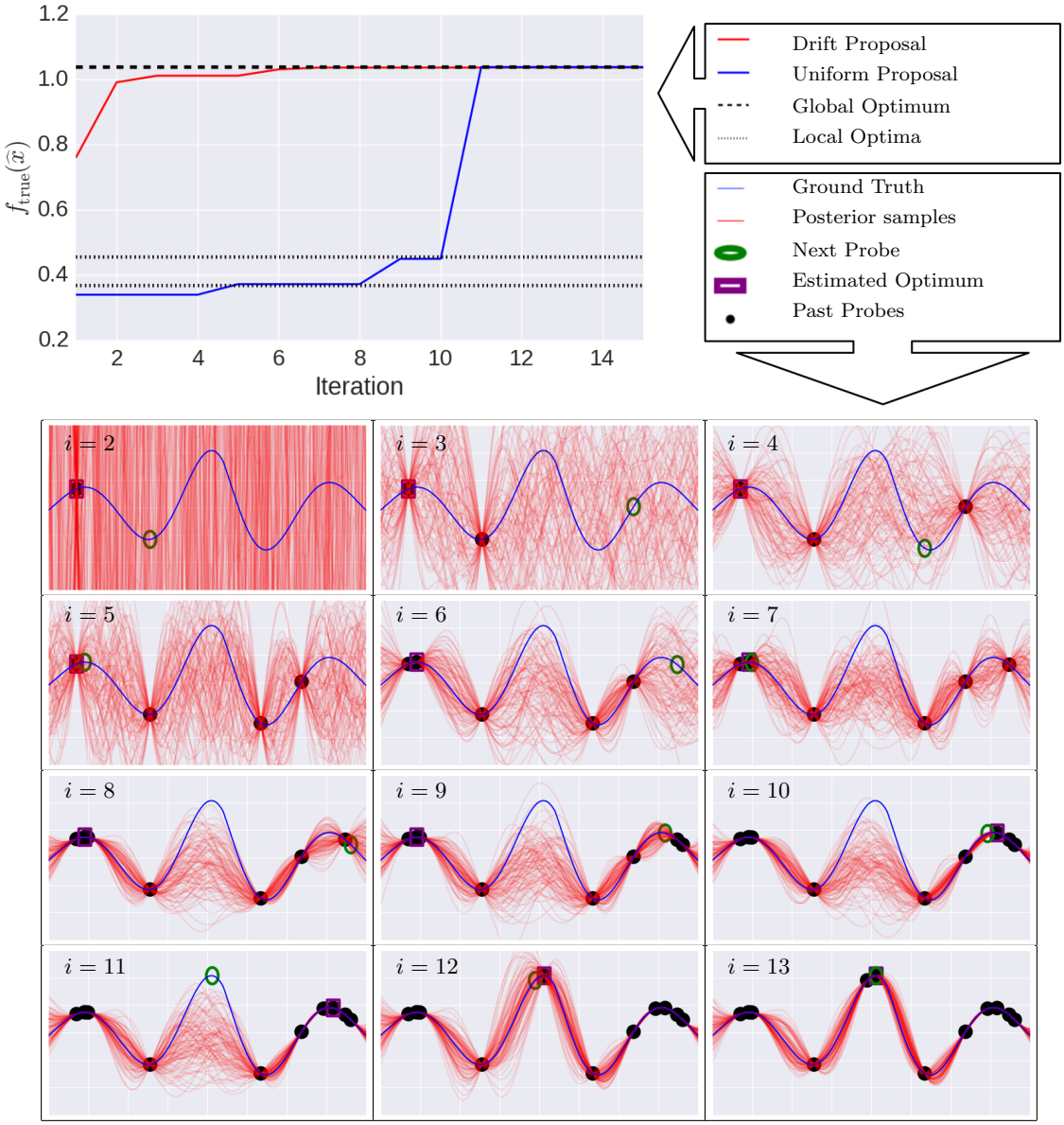


Figure 9: Top: the estimated optimum over time. Blue and Red represent optimization with uniform and Gaussian drift proposals. Black lines indicate the local optima of the true functions. Bottom: a sequence of actions. Depicted are iterations 7-12 with uniform proposals.

a sequence of actions using uniform proposals. The sequence illustrates the exploitation exploration trade-off that the implementation overcomes. We start with complete uncertainty ($i = 2$). The Bayesian agent performs exploration until it gets a (wrong!) idea of where the optimum could be (exploiting the local optima $i = 5$ to $i = 10$). $i = 11$ shows a change in tactic. The Bayesian agent, having exploited the local optima in previous steps, is now reducing uncertainty in area it knows nothing about, eventually finding the global optimum.

5. Discussion

We provided `gpmem`, an elegant linguistic framework for function learning-related tasks such as Bayesian optimization and GP kernel structure learning. We highlighted how `gpmem` overcomes shortcomings of the notations currently used in statistics, and how language constructs from programming allow the expression of models which would be cumbersome (prohibitively so, in some cases) to express in statistics notation. We evaluated our contribution on a range of hard problems for state-of-the-art Bayesian nonparametrics.

Appendix

A Covariance Functions

$$k^{\text{se}} = \sigma^2 \exp(-\frac{(x-x')^2}{2\ell^2}) \tag{24}$$

$$k^{\text{linear}} = \sigma^2(xx') \tag{25}$$

$$k^{\text{constant}} = \sigma^2 \tag{26}$$

$$k^{\text{wn}} = \sigma^2\delta_{x,x'} \tag{27}$$

$$k^{\text{rational quadratic}} = \sigma^2\left(1 + \frac{(x-x')^2}{2\alpha\ell^2}\right)^{-\alpha} \tag{28}$$

$$k^{\text{periodic}} = \sigma^2 \exp\left(\frac{2\sin^2(\pi(x-x')/p}{\ell^2}\right). \tag{29}$$

From top to bottom: the squared-exponential covariance function (24), also know as smoothing kernel; the linear kernel (25); the constant kernel (26); the white noise kernel (27); the rational quadratic kernel (28); and the periodic kernel (29).

B Covariance Simplification

`SE`	$\times$ `SE`	$\rightarrow$ `SE`
`{SE,PER,C,WN}`	$\times$ `WN`	$\rightarrow$ `WN`
`LIN`	+ `LIN`	$\rightarrow$ `LIN`
`{SE,PER,C,WN,LIN}`	$\times$ `C`	$\rightarrow$ `{SE,PER,C,WN,LIN}`

Rule 1 is derived as follows:

$$\begin{aligned}
\sigma_c^2 \exp(-\frac{(x-x')^2}{2\ell_c^2}) &= \sigma_a^2 \exp(-\frac{(x-x')^2}{2\ell_a^2}) \times \sigma_b^2 \exp(-\frac{(x-x')^2}{2\ell_b^2}) \\
&= \sigma_c^2 \exp(-\frac{(x-x')^2}{2\ell_a^2}) \times \exp(-\frac{(x-x')^2}{2\ell_b^2}) \\
&= \sigma_c^2 \exp\left(-\frac{(x-x')^2}{2\ell_a^2} - \frac{(x-x')^2}{2\ell_b^2}\right) \\
&= \sigma_c^2 \exp\left(-\frac{(x-x')^2}{2\ell_c^2}\right)
\end{aligned} \tag{30}$$

For stationary kernels that only depend on the lag vector between x and x' it holds that multiplying such a kernel with a WN kernel we get another WN kernel (Rule 2). Take for example the SE kernel:

$$\sigma_a^2 \exp\left(-\frac{(x-x')^2}{2\ell_c^2}\right) \times \sigma_b\delta_{x,x'} = \sigma_a\sigma_b\delta_{x,x'} \tag{31}$$

Rule 3 is derived as follows:

$$\theta_c(x \times x') = \theta_a(x \times x') + \theta_b(x \times x') \tag{32}$$

Multiplying any kernel with a constant obviously changes only the scale parameter of a kernel (Rule 4).

C The Struct-Operator

$$\begin{aligned}
\text{Struct}(k^{\text{linear}}) &= \text{LIN} \\
\text{Struct}(k^{\text{periodic}}) &= \text{PER} \\
\text{Struct}(k^{\text{se}}) &= \text{SE} \\
\text{Struct}(k^{\text{wn}}) &= \text{WN} \\
\text{Struct}(k^{\text{linear}+\text{periodic}}) &= \text{LIN} + \text{PER} \\
\text{Struct}(k^{\text{linear}\times\text{periodic}}) &= \text{LIN} \times \text{PER}
\end{aligned}$$

D Additional Venture Code

Squared-exponential Kernel in Venture

```
assume sf = tag(scope="hyper-parameters", block=1, gamma(5,1));
assume l = tag("scope="hyper-parameters", block=2, gamma(5,1));
assume kse  = apply_function(make_squaredexp, sf, l);
```

Hierarchical Hyper-priors

```
assume alpha_sf = tag(scope="hyperhyper", block=1, gamma(5,1));
assume beta_sf = tag("scope="hyperhyper", block=2, gamma(5,1));

assume alpha_l = tag(scope="hyperhyper", block=1, gamma(5,1));
assume beta_l = tag("scope="hyperhyper", block=2, gamma(5,1));
```

E Glossary

$\mathcal{N}$	(Multivariate-) Gaussian
$\mathcal{GP}$	Gaussian Process
$\mathbb{E}$	Expectation
x, x_i	Scalar, possibly indexed with i
$\mathbf{x}$	Column vector, training data: regression input (also actions in section 4.3)
$\mathbf{y}$	Column vector, training data: regression output (also rewards in section 4.3)
$\mathcal{X}$	A set of possible actions
$\mathbf{x}'$	Column vector, unseen test input: regression input
$\mathbf{y}'$	Column vector, sample from predictive posterior, that is a sample from $\mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}, \mathbf{K}_{\boldsymbol{\theta}}^{\text{post}})$
$\mathbf{x}^*$	Column vector, unseen test input: regression input before any data has been observed
$\mathbf{y}^*$	Column vector, sample from the predictive prior conditioned on $\boldsymbol{\theta}$ and unseen test input $\mathbf{x}^*$
$\mathbf{D}$	Data matrix $[\mathbf{x}\ \mathbf{y}]$
$\mu(x)$	Mean function
$\boldsymbol{\theta}_{\text{mean}}$	hyper-parameters for a mean function
k or $k(x_i, x_j)$	a covariance function or kernel, that is a function that takes two scalars as input
$\boldsymbol{\theta}$	hyper-parameters for a kernel/covariacne function (also semicontext in section 4.3)
$k(x_i, x_j \mid \boldsymbol{\theta})$	a kernel conditioned on its hyper-parameters
$K(\mathbf{x}, \mathbf{x}' \mid \boldsymbol{\theta})$	Function outputting a matrix of dimension $I \times J$ with entries $k(x_i, x_j \mid \boldsymbol{\theta})$; with $x_i \in \mathbf{x}$ and $x_j \in \mathbf{x}'$ where I and J indicate the length of the column vectors $\mathbf{x}$ and $\mathbf{x}'$
$k_{\boldsymbol{\theta}}$	a covariance function parameterized with $\boldsymbol{\theta}$
$\mathbf{K}_{\boldsymbol{\theta}}$ or $\mathbf{K}_{(\boldsymbol{\theta},\mathbf{x},\mathbf{x})}$	covariance matrix computed with by $K(\mathbf{x}, \mathbf{x} \mid \boldsymbol{\theta})$
$\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}$	Posterior mean vector for $\mathbf{y}' \mid \mathbf{x}, \mathbf{x}', \mathbf{y}, \boldsymbol{\theta}$
$\mathbf{K}_{\boldsymbol{\theta}}^{\text{post}}$	Posterior covariance matrix for $\mathbf{y}' \mid \mathbf{x}, \mathbf{x}', \mathbf{y}, \boldsymbol{\theta}$
$\mathbf{L}$	lower triangular matrix, given by the Cholesky factorization as $\mathbf{L} \coloneqq \text{chol}(\mathbf{K}_{\boldsymbol{\theta}})$
k^{se}	Squared exponential covariance function
k^{linear}	Linear covariance function
k^{constant}	Constant covariance function
k^{wn}	White noise covariance function
$k^{\text{rational quadratic}}$	Rational quadratic covariance function
k^{periodic}	Periodic covariance function
SE	Symbolic expression for the squared exponential covariance function
LIN	Symbolic expression for the linear covariance function
PER	Symbolic expression for the periodic covariance function
RQ	Symbolic expression for the rational quadratic covariance function
C	Symbolic expression for the constant covariance function
WN	Symbolic expression for the white noise covariance function

$\wedge$	Logical and
$\vee$	Logical or
$\boldsymbol{\mathcal{K}}$	Random variable over kernel functions
$\boldsymbol{\mathcal{K}}_{\text{Struct}}$	Random variable over symbolic interpretations for kernel functions
$\text{Struct}(k)$	Symbolic interpretation and simplification for kernel k
BK	A set of base kernels
$\mathbf{S}$	A subset of BK, randomly selected
$\boldsymbol{\Omega}$	Random variable for composition operators, in our case that is kernel addition and multiplication $\{+, \times\}$
$\Gamma(\alpha, \beta)$	Gamma distribution with shape parameter α and rate β
ℓ	Length-scale parameter for k^{se}
sf	Scale factor parameter
$\vartheta \in \Theta$	Context in Thompson sampling
Θ	Context space
V	Value function
Q_{proposal}	Proposal distribution
E	Energy function
s	Temperature parameter

References

Stephen Barnett and Stephen Barnett. *Matrix methods for engineers and scientists*. McGraw-Hill, 1979.

D. Barry. Nonparametric bayesian regression. *The Annals of Statistics*, 14(3):934–953, 1986.

G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time series analysis: forecasting and control*. 1997.

A. Damianou and N. Lawrence. Deep gaussian processes. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 207–215, 2013.

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

B. Ferris, D. Haehnel, and D. Fox. Gaussian processes for signal strength-based location estimation. In *Proceedings of the Conference on Robotics Science and Systems*. Citeseer, 2006.

M. A. Gelbart, J. Snoek, and R. P. Adams. Bayesian optimization with unknown constraints. *arXiv preprint arXiv:1403.5607*, 2014.

Z. Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553): 452–459, 2015.

M. Kemmler, E. Rodner, E. Wacker, and J. Denzler. One-class classification with gaussian processes. *Pattern Recognition*, 46(12):3507–3518, 2013.

C. Kemp, J. B. Tenenbaum, T. L. Griffiths, T.i Yamada, and N. Ueda. Learning systems of concepts with an infinite relational model. In *AAAI*, volume 3, page 5, 2006.

M. C. Kennedy and A. O'Hagan. Bayesian calibration of computer models. *Journal of the Royal Statistical Society. Series B, Statistical Methodology*, pages 425–464, 2001.

T. D. Kulkarni, P. Kohli, J. B. Tenenbaum, and V. Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4390–4399, 2015.

M. Kuss and C. E. Rasmussen. Assessing approximate inference for binary gaussian process classification. *The Journal of Machine Learning Research*, 6:1679–1704, 2005.

J. Kwan, S. Bhattacharya, K. Heitmann, and S. Habib. Cosmic emulation: The concentration-mass relation for wcdm universes. *The Astrophysical Journal*, 768(2):123, 2013.

B.M. Lake and J.B. Salakhutdinov, R.and Tenenbaum. A global geometric framework for nonlinear dimensionality reduction. *Science*, 350(6266):1332–1338, 2015.

J. R. Lloyd, D. Duvenaud, R. Grosse, J. Tenenbaum, and Z. Ghahramani. Automatic construction and natural-language description of nonparametric regression models. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2014.

David JC MacKay. Introduction to gaussian processes. *NATO ASI Series F Computer and Systems Sciences*, 168:133–166, 1998.

Nimalan Mahendran, Ziyu Wang, Firas Hamze, and Nando D Freitas. Adaptive mcmc with bayesian optimization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 751–760, 2012.

V. K. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

R. M. Neal. *Bayesian Learning for Neural Networks*. PhD thesis, University of Toronto, 1995.

R. M. Neal. Monte carlo implementation of gaussian process models for bayesian regression and classification. *arXiv preprint physics/9701026*, 1997.

M. Neumann, S Huang, D. Marthaler, and K. Kersting. pygps - a package for gaussian processe, December 2015. URL `http://www-ai.cs.uni-dortmund.de/weblab/static/api_docs/pyGPs/1`.

Davide Nitti, Vaishak Belle, and Luc De Raedt. Planning in discrete and continuous markov decision processes by probabilistic programming. In *Machine Learning and Knowledge Discovery in Databases*, pages 327–342. Springer, 2015.

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

C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2006.

Brian E Ruttenberg, Matthew P Wilkins, and Avi Pfeffer. Hierarchical reasoning with probabilistic programming. In *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

M. D. Schneider, L. Knox, S. Habib, K. Heitmann, D. Higdon, and C. Nakhleh. Simulations and cosmological inference: A statistical model for power spectra means and covariances. *Physical Review D*, 78(6):063529, 2008.

J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2951–2959, 2012.

The GPy authors. GPy: A gaussian process framework in python. `http://github.com/SheffieldML/GPy`, 2012–2015.

W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, pages 285–294, 1933.

C. K. I. Williams and D. Barber. Bayesian classification with gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1342–1351, 1998.