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@ALUM.MIT.EDU
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Alexey Radul AXCH@MIT.EDU
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

Gaussian Processes (GPs) are widely used tools in statistics, machine learning, robotics, computer vision, and scientific computation. However, despite their popularity, they can be difficult to apply; all but the simplest classification or regression applications require specification and inference over complex covariance functions that do not admit simple analytical posteriors. Probabilistic programming shows potential for reducing the latter barrier, if the computational surface of a GP can be suitably packaged for cooperating with available generic inference tactics. This paper shows how to embed Gaussian processes in any higher-order probabilistic programming language, using an idiom based on memoization, and demonstrates its utility by implementing and extending classic and state-of-the-art GP applications. The interface to Gaussian processes, called `gpmem`, takes an arbitrary real-valued computational process as input and returns a statistical emulator that automatically improves as the original process is invoked and its input-output behavior is recorded. The flexibility of `gpmem` is illustrated via three applications: (i) robust GP regression with hierarchical hyper-parameter learning, (ii) discovering symbolic expressions from time-series data by fully Bayesian structure learning over kernels generated by a stochastic grammar, and (iii) a bandit formulation of Bayesian optimization with automatic inference and action selection. All applications share a single 100-line Python library and require fewer than 20 lines of probabilistic code each.

Keywords: Probabilistic Programming, Gaussian Processes, Structure Learning, Bayesian Optimization

1. Introduction

Gaussian Processes (GPs) 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, by searching over structured kernels generated by a stochastic grammar, the "Automated Statistician" system can produce symbolic descriptions of time series (Duvenaud et al., 2013) that can be translated into natural language (Lloyd et al., 2014).

This paper shows how to integrate GPs into higher-order probabilistic programming languages and illustrates the utility of this integration by implementing it for the Venture platform. The key idea is to use GPs to implement a kind of "statistical" or "generalizing" memoization. The resulting higher-order procedure, called `gpmem`, takes a kernel function and a source function and returns a GP-based statistical emulator for the source function that can be queried at locations where the source function has not yet been evaluated. When the source function is invoked, new datapoints are incorporated into the emulator. In principle, the covariance function for the GP is also allowed to be an arbitrary probabilistic program. This simple packaging covers the full range of uses of the GP described above, including both statistical applications and applications to scientific computation and uncertainty quantification.

This paper illustrates `gpmem` by embedding it in Venture, a general-purpose, higher-order probabilistic programming platform (Mansinghka et al., 2014). Venture has several distinctive capabilities that are needed for the applications in this paper. First, it supports a flexible foreign interface for modeling components that supports the efficient rank-1 updates required by standard GP implementations. Second, it provides inference programming constructs that can be used to describe custom inference strategies that combine elements of gradient-based, Monte Carlo, and variational inference techniques. This level of control over inference is key to state-of-the-art applications of GPs. Third, it supports models with stochastic recursion, a priori unbounded support sets, and higher-order procedures; together, these enable the combination of stochastic grammars with a fast GP implementation, needed for structure learning. Fourth, Venture permits nesting of modeling and inference, which is needed for the use of GPs in Bayesian optimization over general objective functions that may in general themselves be derived from modeling and inference.

To the best of our knowledge, this is the first general-purpose integration of GPs into a probabilistic programming language. Unlike software libraries such as GPy (The GPy authors, 2012–2015), our embedding allows uses of GPs that go beyond classification and regression to include state-of-the art applications in structure learning and meta-optimization.

This paper presents three applications of gpmem: (i) a replication of results by 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: support for fully Bayesian structure learning and language constructs for inference programming. All applications share a single 100-line Python library and require fewer than 20 lines of probabilistic code each.

2. Background on Gaussian Processes

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

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

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_{noise}^2. 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. Efficient computation of both the log-likelihood and the predictive posterior can be packaged for Venture as a Stochastic Procedure (SP) (Mansinghka et al., 2014). Our algorithm of choice 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

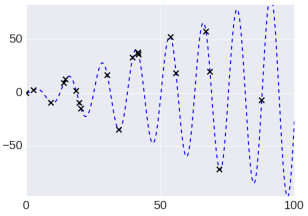
$$\mathbf{y}^* \sim \mathcal{N}\big(0, K^{\text{linear}}(\mathbf{x}^*, \mathbf{x}^* \mid \sigma_1)\big)$$

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

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



(a) Raw Data

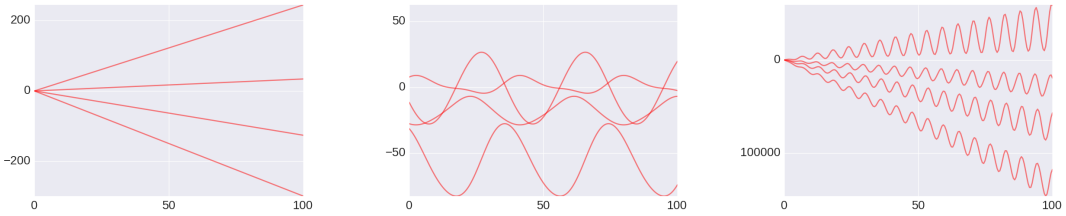
$$k^{\text{linear}} = \sigma_1^2(xx')$$

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

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

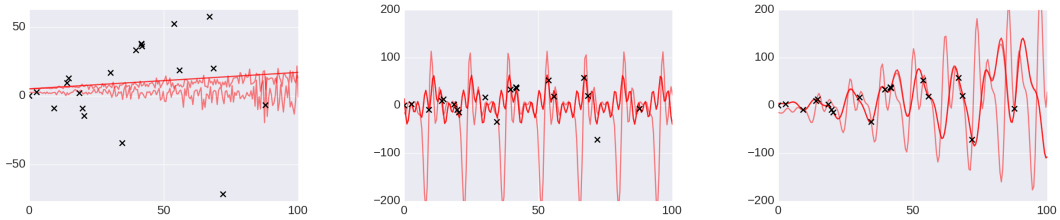
(b) Kernels

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 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.

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 a 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.

2.2 Packaging Gaussian Processes for Venture

Smoothly integrating Gaussian processes into a probabilistic programming system calls for several facilities. One natural embedding of a GP is as a higher-order stochastic function that accepts the mean and covariance kernel functions and returns a function from inputs to outputs:

```
assume gp = make_gp(mu, K);
// Jointly sample values for the inputs 3, 4, 5
predict gp(3);
predict gp(4);
predict gp(5);
```

The different invocations of `gp` above are not independent (unless `K` gives zero covariance for those pairs of inputs), though they are exchangeable. Furthermore, it is not possible to construct a finite object conditioned on which all future invocations of `gp` become independent, as it would need to contain an infinite amount of information. Consequently, those applications need to refer to some shared state that is updated in response to applications appearing (and, if they occur in conditional control branches, disappearing).

In addition to sampling, the GP library must also expose the ability to report the probability density function of any given application of `gp` (conditioned on the results of all the others). Finally, since they have to maintain sufficient statistics to implement exchangeable coupling anyway, it makes sense to expose to the runtime system the ability to evaluate the joint probability density of all extant applications of `gp` without having to query each individually, for example for doing inference on (the parameters of) the covariance kernel.

We wanted to take advantage of optimized linear algebra subroutines for the covariance matrix manipulations, so we implemented the Gaussian process for Venture as a (higher-order) SP. Our GP implementation thus exercises all the major components of the Venture SP interface:

- `make_gp` is an SP that accepts a representation of the mean and covariance kernels and returns a freshly made `gp` procedure.
- a given made `gp` is another SP that is to accept an input point and return the value of the GP at that point. In detail:
- When a `gp` is created, Venture invokes its `constructSPAux` method ("aux" for "auxiliary information"), which creates an (initially empty) store of seen applications of that GP.
- Venture uses the `simulate` and `logDensity` methods of the `gp` SP to sample from or evaluate the probability density function of the Gaussian distribution that describes the value of the GP at that input point. These distributions are conditioned on other extant applications of the `gp` because the `simulate` and `logDensity` methods have access to the auxiliary store of seen input-output pairs.
- When applications of `gp` are created (resp. destroyed), Venture invokes that SP's `incorporate` (resp. `unincorporate`) method with the input-output pair. For a `gp`, that adds (resp. removes) that point from the auxiliary store.
- `gp` sets a flag that indicates to Venture that it can evaluate the joint probability density of all extant applications of itself by examining its auxiliary store. Therefore, when Venture evaluates proposals to, for example, the parameters of a GP covariance kernel, it can compute that GP's applications' contribution to any relevant acceptance ratios by invoking the `logDensityOfCounts` method, instead of calling `logDensity` for each application separately.
- The GP implementation supports gradient inference methods (gradient optimization, Hamiltonian Monte Carlo, etc.) by defining the `gradientOfLogDensityOfCounts` method appropriately.

In the rest of the paper, we do not use `make_gp` directly, but wrap it in the generalizing memoizer `gpmem`, introduced in the next section. As far as implementation in Venture goes, it suffices to arrange for the `f_compute` that `gpmem` returns to insert computed input-output pairs into the auxiliary store constructed for the `f_emu` that `gpmem` also returns.

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:

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

The function $\mathtt{f}_{compute}$ calls `f` and stores the output in a memo table, just as traditional memoization does. The function $\mathtt{f}_{\mathrm{emu}}$ 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:

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

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 the scope "hyper-parameters".

`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:

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

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.

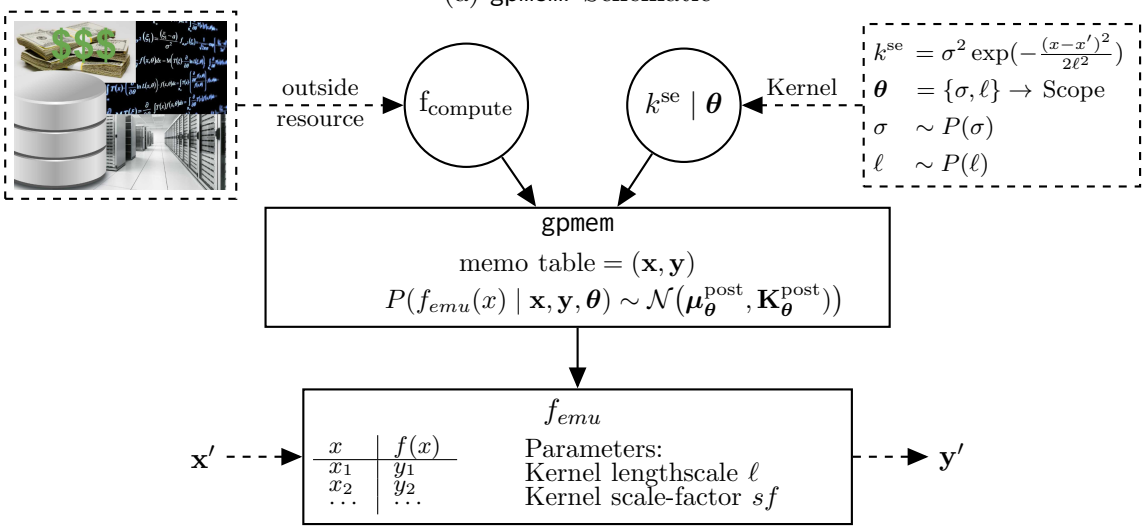
We can either define the function f that serves as as input for `gpmem` natively in Venture (as shown in the Fig. 2 (b)) or we interleave Venture with foreign code. This can be useful when `f` is computed with the help of outside resources. We define and parameterize a squared-exponential kernel (b) which we then supply to `gpmem` (Fig. 2 (c)). 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));
```

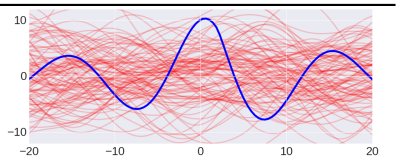
where the second line corresponds to:

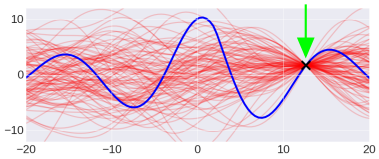
$$\mathbf{y}^* \sim \mathcal{N}\left(0, K^{\text{se}}\left(\begin{bmatrix} -20 \\ \cdots \\ 20 \end{bmatrix}, \begin{bmatrix} -20 \\ \cdots \\ 20 \end{bmatrix} \mid \boldsymbol{\theta} = \{\sigma, \ell\}\right)\right).$$

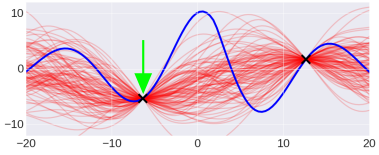
(a) gpmem: Schematic

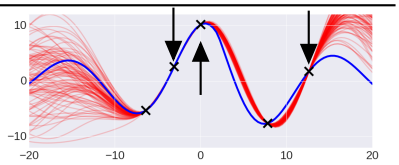


(b)
```
define f   = proc(x) {exp(-0.1*abs(x-2)) * 10 * cos(0.4*x) + 0.2};
assume sf  ~ gamma(5,1)) #"hyper-parameters";
assume l   ~ gamma(5,1)) #"hyper-parameters";
assume kse = gp_cov_scale(sf, gp_cov_se(l));
```

(c)
```
assume (f_compute, f_emu) = gpmem(f, kse);
sample f_emu(array(-20, ···, 20));
```


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


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


(f)
```
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));
```


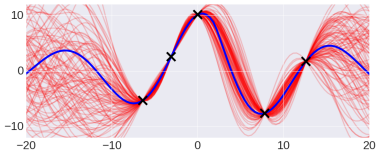
(g)
```
infer mh(#"hyper-parameters", steps=50);
sample f_emu(array(-20, ···, 20));
```


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 belief given certain Venture directives. On the right, we depict the true function (blue), samples from the emulator (red) and incorporated observations (black).

Note: Each red curve in each right-hand panel in Fig. 2 corresponds to a distinct invocation of the corresponding "sample" directive. For brevity, we show only one invocation in the left-hand panel.

In Fig. 2 (d), we probe the external function `f` at point 12.6 and memoize its 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 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 (e)) to see that we gain certainty about another part of the curve.

We can add information to $\mathtt{f}_{\text{emu}}$ about presumable value pairs of `f` without calling $\mathtt{f}_{\text{compute}}$ (Fig. 2 (f)). If a friend tells us the value of `f` we can call `observe` to store this information in the incorporated observations for $\mathtt{f}_{\text{emu}}$ 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 $\mathtt{f}_{\text{compute}}$. 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 tag a collection of related random choices, such as all hyper-parameters $\boldsymbol{\theta}$. These tags are supplied to the inference program to specify on which random variables inference should be done:

```
infer mh(#"hyper-parameters", steps=50);
```

In this case, we perform one Metropolis-Hastings (MH) transition over the scope hyper-parameters and choose a random member of this scope, that is we choose one hyper-parameter at random.

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

It bears mention that Venture supports custom inference actions. For example, MH with Gaussian drift proposals looks like this:

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

define my_markov_chain =
```

```
        apply_mh_correction(
            subproblem=choose_by_scope(#"hyper-parameters"),
            proposal=symmetric_local_proposal_from_chain(drift_kernel))

    infer my_markov_chain;
```

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. As written, this kernel will be applied to one hyper parameter at a time. The behavior of this inference program is not displayed in the Figure.

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). Here, we show hyper-parameter inference for regression using an example taken from a paper on the treatment of outliers with hierarchical Bayesian hyper-priors for GPs (Neal, 1997):

$$f_{\text{true}}(x) = 0.3 + 0.4x + 0.5\sin(2.7x) + \frac{1.1}{(1+x^2)} + \eta \quad \textit{with} \quad \eta \sim \mathcal{N}(0, \sigma_{\text{noise}}). \tag{14}$$

We emphasize that `gpmem` can be used with externally collected data sets whose generating process may not be available at all, by coding the "true function" argument to `gpmem` as a lookup in a fixed data set (Fig. 3 (b)). That data was synthesized offline by applying the f_{true} of (14) to a fixed set of inputs, with outliers given by setting $\sigma_{\text{noise}} = 1$ in 5% of the cases (the remaining 95% "inliers" have $\sigma_{\text{noise}} = 0.1$).

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

$$\ell \sim \Gamma(\alpha_1, \beta_1),\ \sigma \sim \Gamma(\alpha_2, \beta_2) \tag{15}$$

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

$$\alpha_1 \sim \Gamma(\alpha_\alpha^1, \beta_\alpha^1),\ \beta_1 \sim \Gamma(\alpha_\beta^1, \beta_\beta^1),\ \alpha_2 \sim \Gamma(\alpha_\alpha^2, \beta_\alpha^2),\ \beta_2 \sim \Gamma(\alpha_\beta^2, \beta_\beta^2). \tag{16}$$

We model this in Venture as illustrated in Fig. 3 (c), using the built-in SP `gamma`.

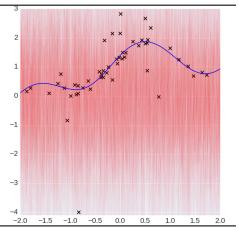
(a) $P(\boldsymbol{\theta} = \{\ell, sf, \sigma\} \mid \mathbf{D}, \boldsymbol{\mathcal{K}})$

(b)
```
// Define 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)};
```

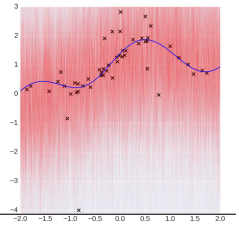
(c)
```
// Initialize hyper-priors
assume alpha_sf ~ gamma(5,1)                    #"hyperhyper";
assume beta_sf  ~ gamma(5,1)                    #"hyperhyper";
assume alpha_l  ~ gamma(5,1)                    #"hyperhyper";
assume beta_l   ~ gamma(5,1)                    #"hyperhyper";

assume sf       ~ gamma(alpha_sf, beta_sf)))    #"hyper";
assume l        ~ gamma(alpha_l, beta_l)))      #"hyper";
assume sigma    ~ gamma(5,1))                   #"hyper";
```

(d)
```
// Initialize covariance function
assume se = gp_cov_scale(sf, gp_cov_se(l));
assume wn = gp_cov_scale(sigma, gp_cov_delta());
assume composite_covariance = gp_cov_sum(se, wn);
```

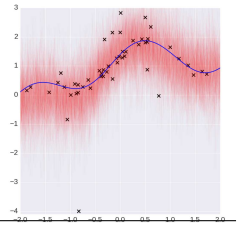
(e)
```
// Create a prober and emulator using gpmem
assume (f_compute, f_emu) =
    gpmem(f_look_up, composite_covariance);
sample f_emu(array(-2, ···, 2));
```



(f)
```
    // Observe all data points
    for (x,y) in data {
                        observe f_emu(x) = y};
    // Or: probe all data points
    for (x,_) in data {
                        predict f_compute(x)};
    sample f_emu(array(-2, ···, 2));
```



(g)
```
// Metropolis-Hastings
infer repeat(100, do(
              mh(#"hyperhyper", steps=1),
              mh(#"hyper",      steps=3)));
sample f_emu(array(-2, ···, 2));
```



(h)
```
// Optimization
infer gradient-ascent(#"hyper", steps=10);

sample f_emu(array(-2, ···, 2));
```

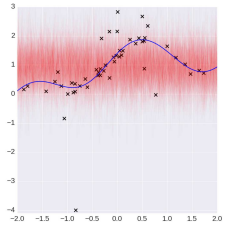


Figure 3: Regression with outliers and hierarchical prior structure.

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 a 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`, giving it `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 (Fig. 3, panel (e)). 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 (f)). 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 (g)). We say nested because we first take two sweeps in the scope `hyperhyper` which characterizes (16) and then one sweep on the scope `hyper` which characterizes (15). This is repeated 100 times 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 (h)).

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 unavailable, 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 $\text{BK} = \{k^1_{\boldsymbol{\theta}^1}, \cdots, k^m_{\boldsymbol{\theta}^m}\}$ with $\boldsymbol{\theta}^* = \{\boldsymbol{\theta}^1, \cdots, \boldsymbol{\theta}^m\}$ (Fig. 4 (a) and (b)). We depict the input for the stochastic grammar in Listing 2.

We sample a random subset $\mathbf{S}$ of the set of supplied base kernels. $\mathbf{S}$ 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

1. Neal (1997) uses the sum of an SE plus a constant kernel. For illustration, we use a WN kernel instead.
2. http://www.automaticstatistician.com/

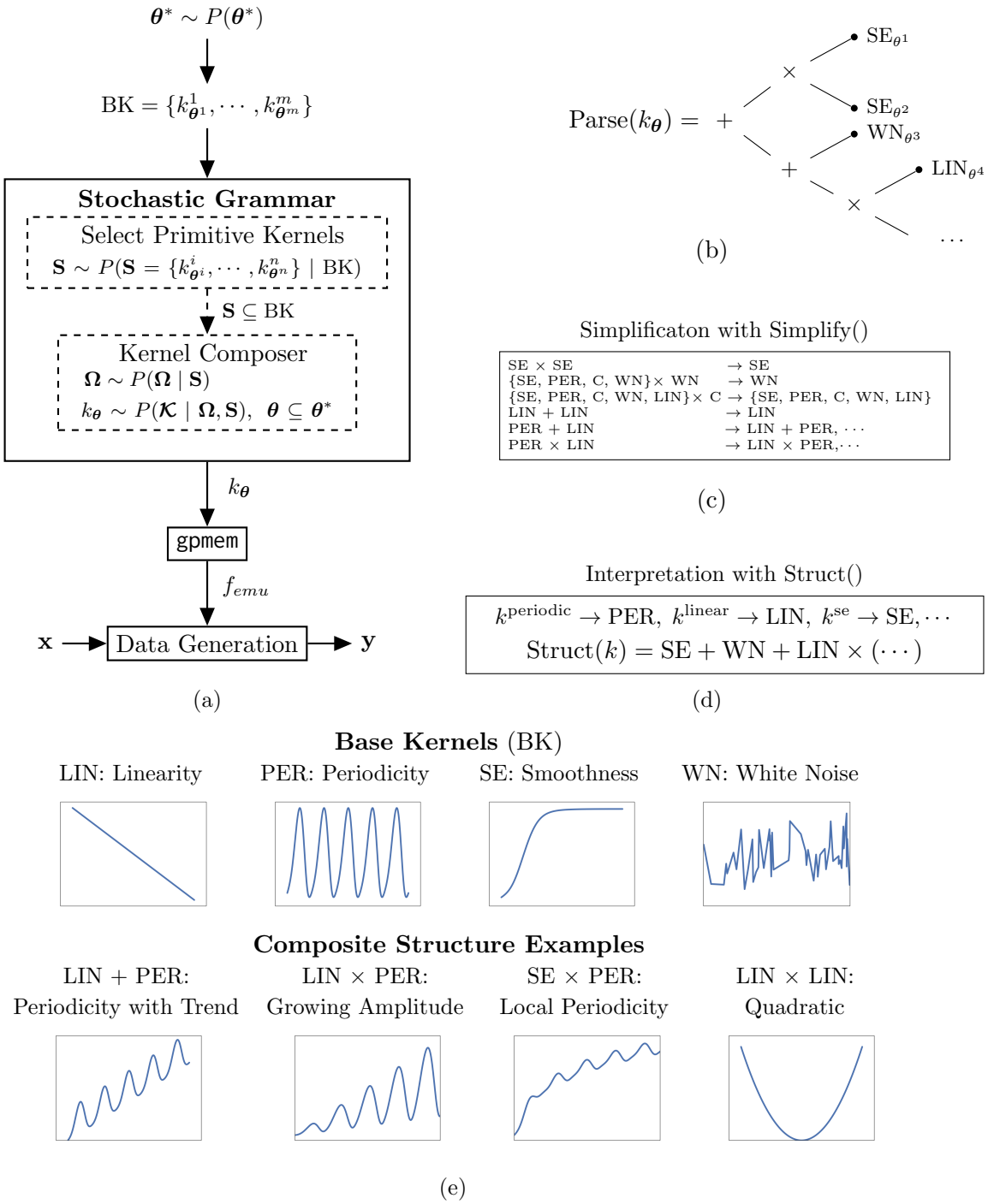


Figure 4: (a) Bayesian GP structure learning. A set of base kernels (BK) with priors on their hyper-parameters serves as hypothesis space and is supplied as input to the stochastic grammar. The stochastic grammar has two parts: (i) a sampler that selects a random set $\mathbf{S}$ of primitive kernels 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) We use Parse($k_{\boldsymbol{\theta}}$) to parse a structure. (c) kernel functions are simplified with the Simplify()-operator. The simplifed $k_{\boldsymbol{\theta}}$ is used as input for Struct() (d) which interprets it symbolically. Base kernels and compositional kernels are shown in (e) alongside their interpretation with Struct().

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 $\mathbf{\Omega}_i = \{+, \times\}$. The probability for all such decisions is given by a binomial distribution:

$$P(\mathbf{\Omega} \mid \mathbf{S}) = \binom{n}{r} p_{+\times}^{r} (1 - p_{+\times})^{n-r}. \tag{17}$$

We can write the marginal probability of a kernel function as

$$P(\boldsymbol{\mathcal{K}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \iint_{\mathbf{\Omega},\mathbf{S}} P(\boldsymbol{\mathcal{K}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}, \mathbf{\Omega}, \mathbf{S}) \times P(\mathbf{\Omega} \mid \mathbf{S}) \times P(\mathbf{S}) \, \mathbf{d\Omega} \, \mathbf{dS} \tag{18}$$

with $\boldsymbol{\theta} \subseteq \boldsymbol{\theta}^*$ as implied by $\mathbf{S}$. For structure learning with GP kernels, a composite kernel is sampled from $P(\boldsymbol{\mathcal{K}})$ and fed into `gpmem`. The emulator generated by `gpmem` observes unstructured time series data. Venture code for the probabilistic grammar is shown in Listing 1, code for inference with `gpmem` in Listing **??**.

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

All base kernels relevant for this work can be found in Appendix A.

Listing 1: Stochastic Grammar

```
assume choose_primitive = proc(){
    rolled_dice ~  uniform_discrete(0,5);
    cond(
      (rolled_dice == 0)(list("WN",list())),
      (rolled_dice == 1)(list("C",list(gamma(5,1)))),
      (rolled_dice == 2)(list("LIN",list(gamma(5,1)))),
      (rolled_dice == 3)(list("SE",list(gamma(5,1)))),
      (rolled_dice == 4)(list("PER",list(gamma(5,1), gamma(5,1)))),
      else("error"))
  };



assume choose_operator = proc(){
    if (flip())
          { "+" }
    else
          { "*" }
    };

assume generate_abstract_syntax_tree = proc(){
    if (flip(0.7))
          { choose_primitive() }
    else
          { list(choose_operator(), generate_abstract_syntax_tree(),
                                     generate_abstract_syntax_tree())}
    };
```

Listing 2: Compilation

```
assume produce_covariance = proc(ast){
    cond(
      (first(ast) == "WN")(gp_cov_delta(tolerance_constant)),
      (first(ast) == "C")(gp_cov_const(first(second(ast)))),
      (first(ast) == "LIN")(gp_cov_linear(first(second(ast)))),
      (first(ast) == "SE")(gp_cov_se(first(second(ast)))),
      (first(ast) == "PER")(gp_cov_periodic(first(second(ast)),
                                            second(second(ast)))),
      (first(ast) =="+")(gp_cov_sum(
                                    produce_covariance(second(ast)),
                                    produce_covariance(third(ast))
                          )),
      (first(ast) =="*")(gp_cov_product(
                                    produce_covariance(second(ast)),
                                    produce_covariance(third(ast))
                          )),
      else("error"))
  };

assume simplify = proc(ast){ast};

assume produce_binary = proc(ast){
  // baseline noise for stability
  noise = gp_cov_scale(0.1,gp_cov_delta(tolerance_constant));
  covariance_kernel = gp_cov_sum(produce_covariance(ast), noise);
  //covariance_kernel = produce_covariance(ast);
  make_gp(gp_mean_const(0.), covariance_kernel)
};

assume compile = proc(ast) {
  simplified_ast = simplify(ast);
  produce_binary(simplified_ast)
 };

assume ast  ~ generate_abstract_syntax_tree();

assume executable = compile(ast);
```

We defined a simple space of covariance structures in a way that allows us to produce results coherent with work presented in Automatic Statistician (Duvenaud et al., 2013; Lloyd et al., 2014). 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. These are 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 of kernel functions and a posterior distribution of symbolic expressions describing different kernel structures. This allows us to compute the posterior of symbollically equivalent structures, such as $\text{Struct}(k^{\text{linear}} + k^{\text{periodic}}) = \text{Struct}(k^{\text{periodic}} + k^{\text{linear}})$. Both structures yield a sum of a linear kernel and a periodic kernel, that is LIN + PER. Therefore, we parse k with $\text{Parse}(k)$, we simplify an expression with $\text{Simplify}(k)$ and then compute $\text{Struct}(k)$. For the Mauna Loa Co_2 data, this distribution peaks at:

$$\text{Struct}(\boldsymbol{\mathcal{K}} = k) = \text{LIN} + \text{PER} + \text{SE} + \text{WN}. \tag{19}$$

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)).

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

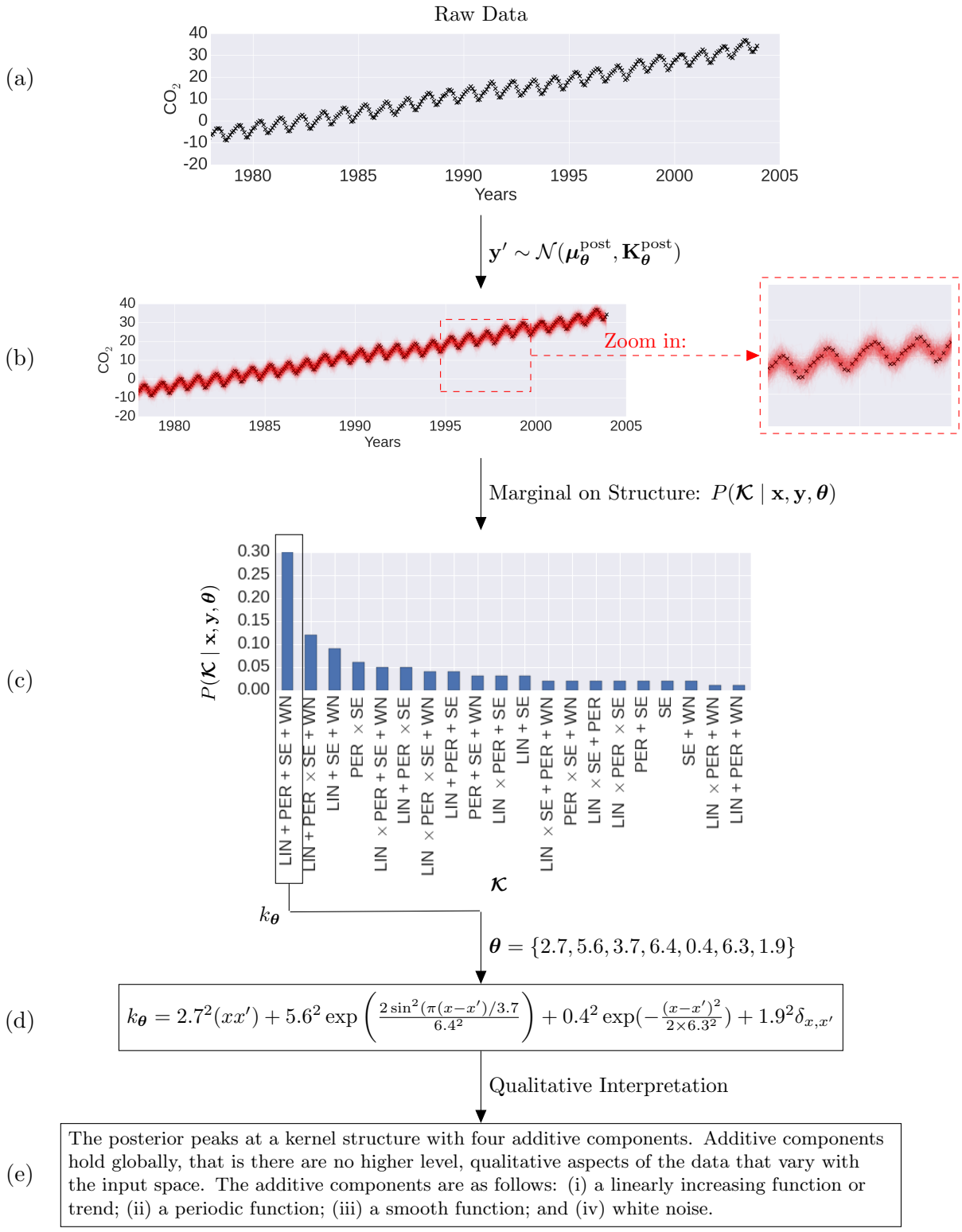
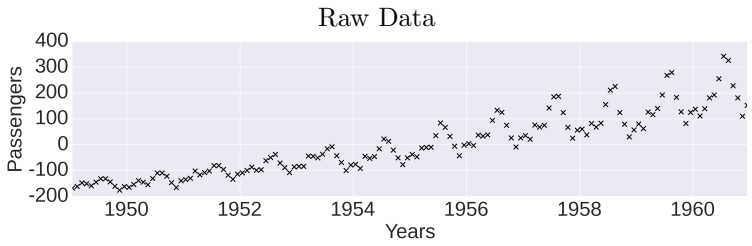


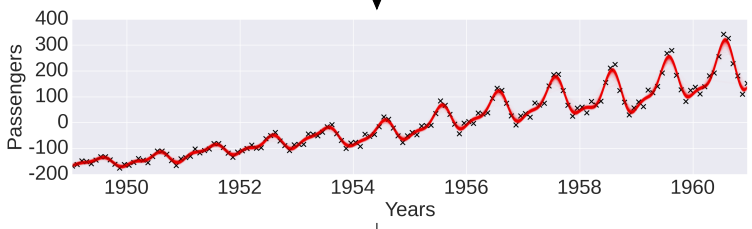
Figure 5: Structure Learning. Starting with raw data (a), we fit a GP (b) and compute the posterior distribution on structures (c). We take a sample of the peak of this distribution (LIN + PER + SE + WN) including its parameters and write it in functional form (d). We depict the human readable interpretation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

(a)



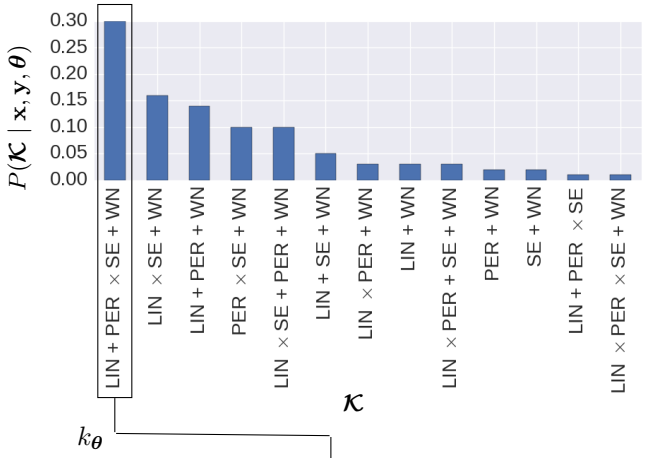
$\mathbf{y}' \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}^{\text{post}}, \mathbf{K}_{\boldsymbol{\theta}}^{\text{post}})$

(b)



Marginal on Structure: $P(\boldsymbol{\mathcal{K}} \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta})$

(c)



$k_{\boldsymbol{\theta}}$

$\boldsymbol{\theta} = \{7.47, 0.27, 4.63, 7.34, 4.4, 4.55, 2.93\}$

(d)

$$k_{\boldsymbol{\theta}} = 7.47^2(xx') + \left(0.27^2 \exp(-\tfrac{(x-x')^2}{2\times 4.63^2}) \times 7.34^2 \exp\left(\tfrac{2\sin^2(\pi(x-x')/4.4}{4.55^2}\right)\right) + 2.93^2\delta_{x,x'}$$

Qualitative Interpretation

(e)

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: (i) a linearly increasing function or trend; (ii) an approximate periodic function; and (iv) white noise.

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).

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:

$$\text{Struct}(\boldsymbol{\mathcal{K}} = k) = \text{LIN} + \text{SE} \times \text{PER} + \text{WN}. \tag{20}$$

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;
- an approximate periodic function; and
- white noise.

Both datasets served as illustrations in the Automatic Statistician project.

Querying time series

Our Bayesian approach to structure learning permits valuable insights into time series data that were previously unavailable. The critical capability is estimating the posterior marginal probability distribution over the kernel structure. For instance, we can define boolean search operations over this marginal that allow one to query the data for the probability that certain structures are present globally.

To wit, we can formulate the question "is a structure $\boldsymbol{\mathcal{K}} = k$ present globally in the data?" as the probability

$$P(\text{Struct}(\boldsymbol{\mathcal{K}} = k) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \approx \frac{1}{T} \sum_{t=1}^{T} \text{Contains}(k, k^t) \tag{21}$$

$$\text{where} \quad \text{Contains}(k, k^t) = \begin{cases} 1, & \text{if } k \underset{global}{\in} k^t, \\ 0, & \text{otherwise.} \end{cases} \tag{22}$$

T is the number of all posterior samples for $\boldsymbol{\mathcal{K}}$ and k^t is one such sample. $\underset{global}{\in} k^t$ reads as "is one of k^t's global functional components". We can now ask simple questions, for example:

> Is there white noise in the data?

To answer this question we set $\text{Struct}(\boldsymbol{\mathcal{K}})$ =WN in (21). We write this in shorthand as $\text{WN}(\boldsymbol{\mathcal{K}})$. Similarly, we write $\text{LIN}(\boldsymbol{\mathcal{K}})$, $\text{PER}(\boldsymbol{\mathcal{K}})$ and $\text{SE}(\boldsymbol{\mathcal{K}})$. 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:

$$P(\text{Struct}(\boldsymbol{\mathcal{K}}^a = k^a) \wedge \text{Struct}(\boldsymbol{\mathcal{K}}^b = k^b) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \text{Contains}(k^a, k^t) \wedge \text{Contains}(k^b, k^t)$$

where

$$\text{Contains}(k^a, k^t) \wedge \text{Contains}(k^b, k^t) = \begin{cases} 1, & \text{if } k^a \text{ and } k^b \underset{global}{\in} k^t, \\ 0, & \text{otherwise} \end{cases}.$$

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

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

Finally, we define the logical OR as

$$\begin{aligned} P(\text{Struct}(\boldsymbol{\mathcal{K}}^a = k^a) \vee \text{Struct}(\boldsymbol{\mathcal{K}}^b = k^b) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \\ = P(\text{Struct}(\boldsymbol{\mathcal{K}}^a = k^a) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \\ + P(\text{Struct}(\boldsymbol{\mathcal{K}}^b = k^b) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \\ - P(\text{Struct}(\boldsymbol{\mathcal{K}}^a = k^a) \wedge \text{Struct}(\boldsymbol{\mathcal{K}}^b = k^b) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta}). \end{aligned}$$

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

Is there white noise or heteroskedastic noise?

by estimating $P(\text{LIN}(\boldsymbol{\mathcal{K}}) \times \text{WN}(\boldsymbol{\mathcal{K}}) \ \vee \ \text{WN}(\boldsymbol{\mathcal{K}}) \mid \mathbf{x}, \mathbf{y}, \boldsymbol{\theta})$. 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. This allows us to generalize the question above to:

Is there noise in the data?

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

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

From a methodological perspective, 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 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.

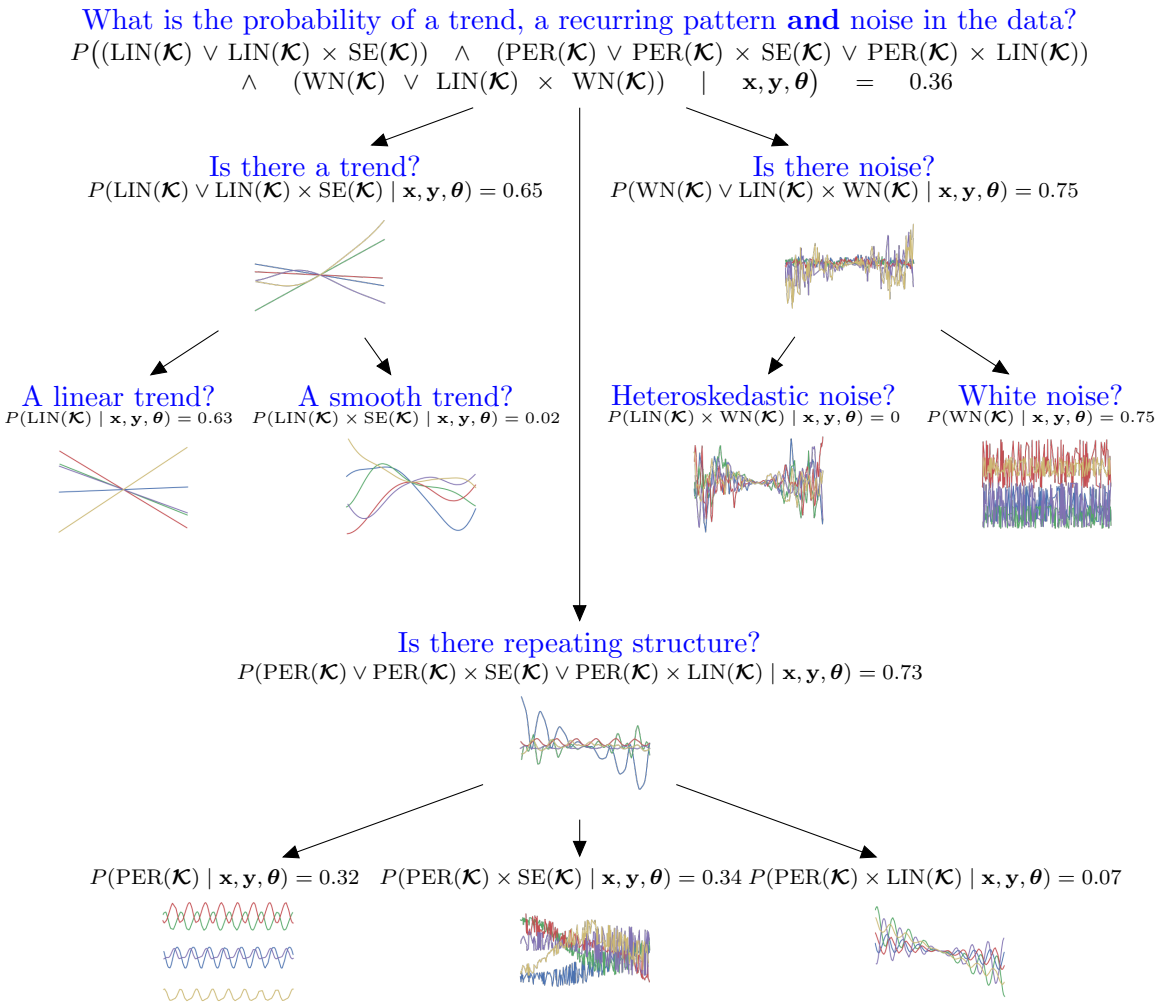


Figure 7: **Querying structural motifs in time series using posterior inference over kernel structure.** The kernel structure serves as a way to formulate natural language questions about the data (blue). The initial question of interest (top) is a fairly general one: "What is the probability of a trend, 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, 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.

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 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 can be described 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.

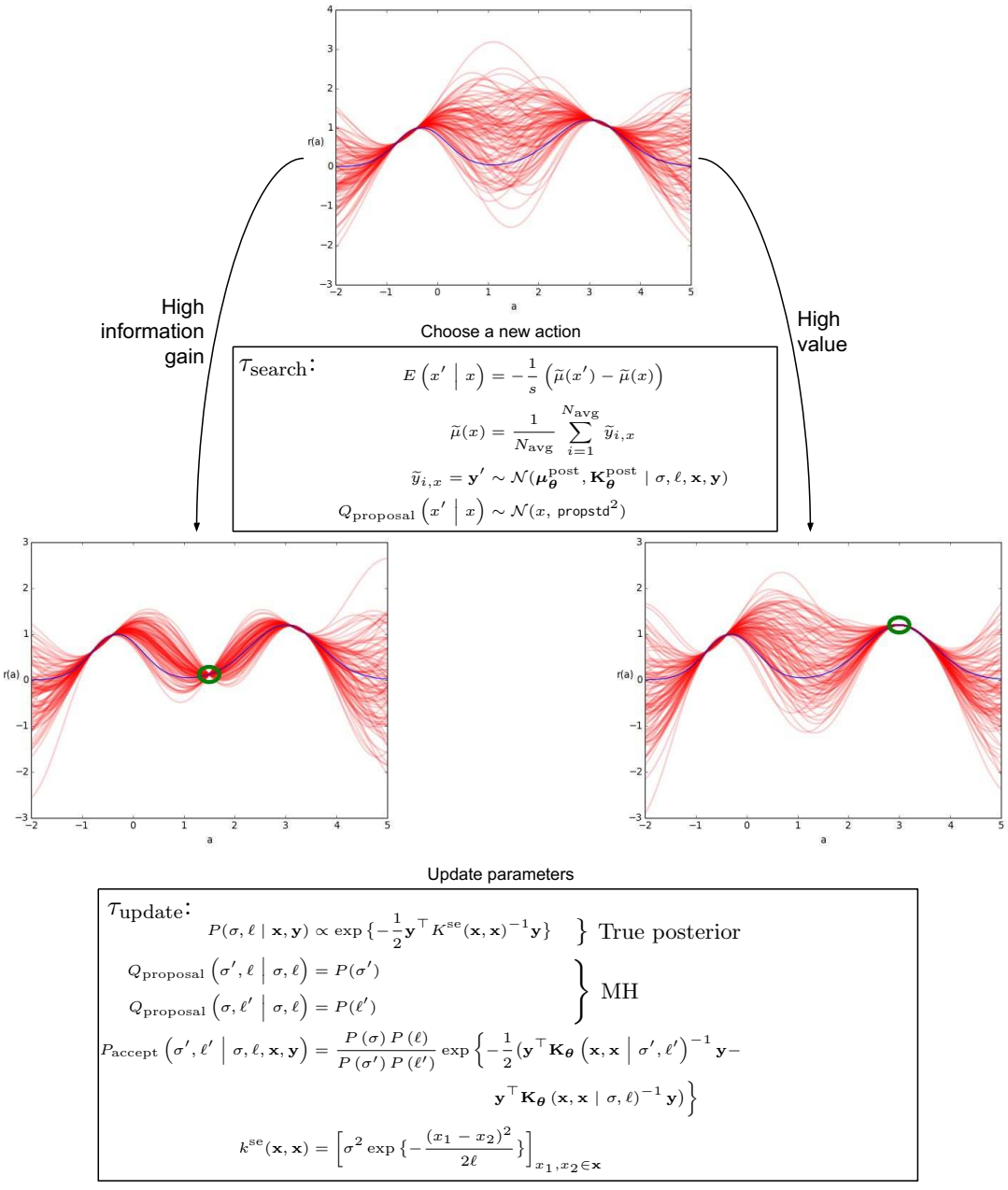


Figure 8: Two possible actions (in green) for an iteration of Thompson sampling. The believed distribution on the value function V is depicted in red. In this example, the true reward function is deterministic, and is drawn in blue. The action on the right receives a high reward, while the action on 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.

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 probabilistic programing language, however, the representation 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

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

The MH acceptance probability for such a proposal is

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

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 complete probabilistic program with `gpmem` implementing Bayesian optimization with Thompson Sampling and both, uniform proposals and drift proposals below (Listing 3,4 and 5).

Listing 3: Initialize `gpmem` for Bayesian optimization

```
1 assume sf ~ uniform_continuous(0, 10) #"hyper";
2 assume l  ~ uniform_continuous(0, 10) #"hyper";
3 assume se = gp_cov_scale(sf, gp_cov_se(l));
4 assume blackbox_f = get_bayesopt_blackbox();
5 assume (f_compute, f_emulate) = gpmem(blackbox_f, se);
```

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.

Listing 4: Bayesian optimization with uniformly distributed proposals

```
// 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 hyper-parameters
  mh(#"hyper", steps=50)));
```

Listing 5: Bayesian optimization with Gaussian drift proposals

```
// A naive estimate of the argmax of the given function
define mc_argmax = proc(func) {
  candidate_xs = mapv(proc(x) {normal(x, 1)},
                      fill(20,last));
  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)))
};

// Initialize helper variables
assume previous_point = uniform_continuous(-20,20);
run(observe(previous_point ,run(sample(previous_point)),prev));

// Main inference loop
infer repeat(15, do(pass,
  // find the next point with mc argmax
  next_point <- action(mc_argmax(
                         emu_pointwise,run(sample(previous_point)))),
  // Probe V at the point mc_argmax(emu_pointwise)
  predict(first(package)(unquote(next_point))),
  // Clear the previous point
  forget(quote(prev)),
  // Remember the current probe as the previous one for the next iter.
  observe(previous_point, next_point,prev),
  // Infer hyper-parameters
  mh(#"hyper", 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 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,

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.

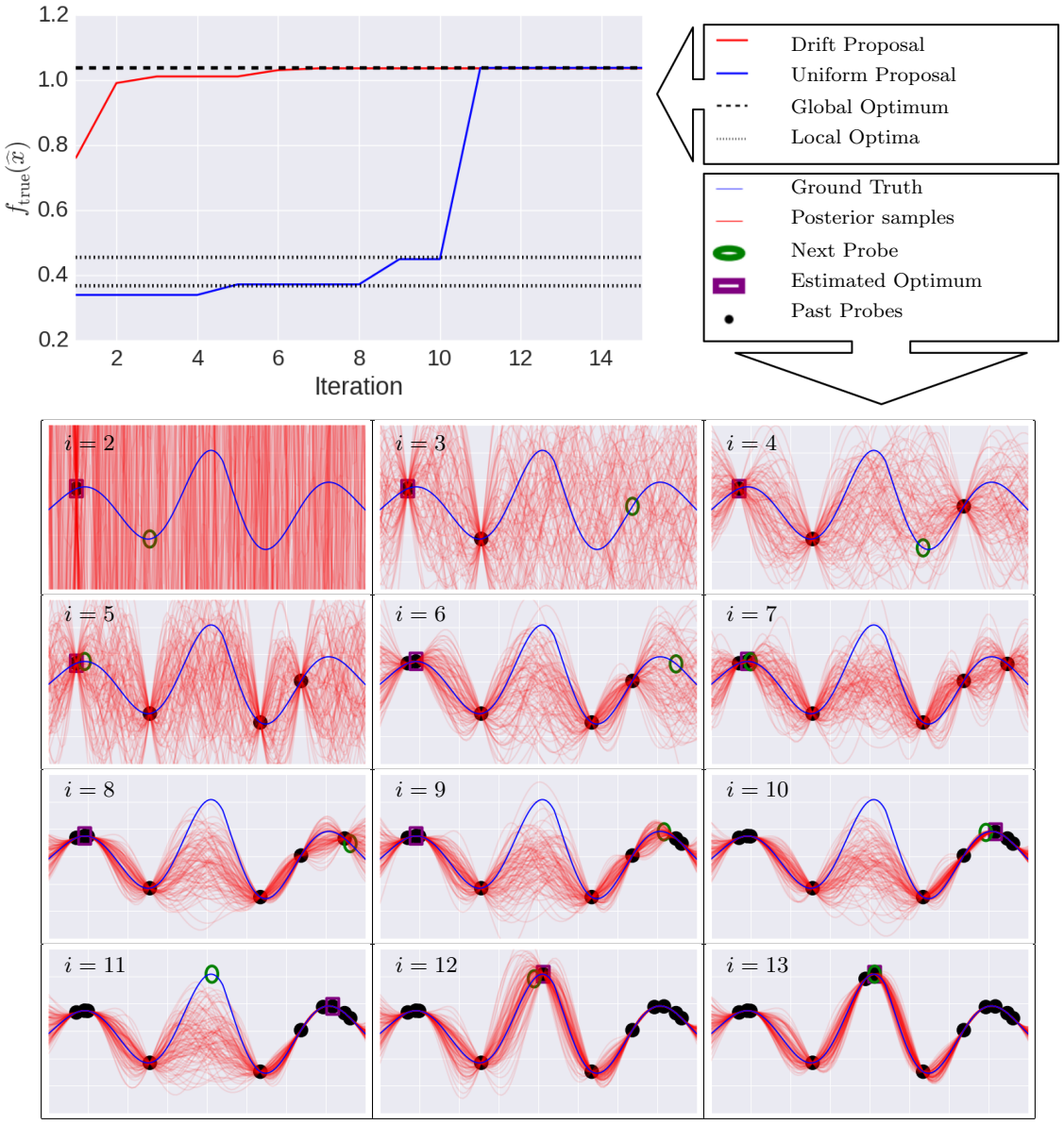


Figure 9: Top: the estimated optimum over time. Blue and Red represent optimization with uniform and Gaussian drift proposals. Black lines indicate the local optima of the true functions. Bottom: a sequence of actions. Depicted are iterations 7-12 with uniform proposals.

is now reducing uncertainty in area it knows nothing about, eventually finding the global optimum.

5. Discussion

This paper has shown that it is feasible and useful to embed Gaussian processes in higher-order probabilistic programming languages by treating them as a kind of statistical memoizer. It has described classic GP regression with both fully Bayesian and MAP inference in a hierarchical hyperprior, as well as state-of-the-art applications to discovering symbolic structure in time series and to Bayesian optimization. All the applications share a common 100-line Python GP library and require fewer than 20 lines of probabilistic code each.

These results suggest several research directions. First, it will be important to develop versions of `gpmem` that are optimized for larger-scale applications. Possible approaches include the standard low-rank approximations to the kernel matrix that are popular in machine learning (Bui and Turner, 2014) as well as more sophisticated sampling algorithms for approximate conditioning of the GP (Lawrence et al., 2009). Second, it seems fruitful to abstract the notion of a "generalizing" memoizer from the specific choice of a Gaussian process model as the mechanism for generalization. "Generalizing" or statistical memoizers with custom regression techniques could be broadly useful in performance engineering and scheduling systems. The timing data from performance benchmarks could be run through a generalizing memoizer by default. This memoizer could be queried (and its output error bars examined) to inform the best strategy for performing the computation or predict the likely runtime of long-running jobs. Third, the structure learning application suggests follow-on research in information retrieval for structured data. It should be possible to build a time series search engine that can handle search predicates such as "has a rising trend starting around 1988" or "is perodic during the 1990s". The variation on the Automated Statistician presented in this paper can provide ranked result sets for these sorts of queries because it tracks posterior uncertainty over structure and also because the space of structural patterns that it can handle is easy to modify by making small changes to a short VentureScript program.

The field of Bayesian nonparametrics offers a principled, fully Bayesian response to the empirical modeling philosophy in machine learning (Ghahramani, 2012), where Bayesian inference is used to encode a state of broad ignorance rather than a bias stemming from strong prior knowledge. It is perhaps surprising that two key objects from Bayesian nonparametrics, Dirichlet processes and GPs, fit naturally in probabilistic programming as variants of memoization (Roy et al., 2008). It is not yet clear if the same will be true for other processes, e.g. Wishart processes, or hierarchical Beta processes. We hope that the results in this paper encourage the development of other nonparametric libraries for higher-order probabilistic programming languages.

Acknowledgements
This research was supported by DARPA (under the XDATA and PPAML programs), IARPA (under research contract 2015-15061000003), the Office of Naval Research (under research contract N000141310333), the Army Research Office (under agreement number W911NF-13-1-0212), the Bill & Melinda Gates Foundation, and gifts from Analog Devices and Google.

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 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
kernel functions	
Parse(k)	Parse the structure for kernel k
Simplify(k)	Simplify the functional expression for kernel k
Struct(k)	Symbolic interpretation for kernel k
Contains(k, k^t)	A kernel k^t contains the global kernel structure k
SE($\boldsymbol{\mathcal{K}}$)	Operator to check if Struct($\boldsymbol{\mathcal{K}} = k$) = SE
LIN($\boldsymbol{\mathcal{K}}$)	Operator to check if Struct($\boldsymbol{\mathcal{K}} = k$) = LIN
PER($\boldsymbol{\mathcal{K}}$)	Operator to check if Struct($\boldsymbol{\mathcal{K}} = k$) = PER
WN($\boldsymbol{\mathcal{K}}$)	Operator to check if Struct($\boldsymbol{\mathcal{K}} = k$) = WN
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.

T. D. Bui and R. E. Turner. Tree-structured gaussian process approximations. In *Advances in Neural Information Processing Systems*, pages 2213–2221, 2014.

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. Bayesian non-parametrics and the probabilistic approach to modelling. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 371(1984):20110553, 2012.

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

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.

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.

N. D. Lawrence, M. Rattray, and M. K. Titsias. Efficient sampling for gaussian process inference using control variables. In *Advances in Neural Information Processing Systems*, pages 1681–1688, 2009.

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.

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.

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

D. M. Roy, V. K. Mansinghka, N. D. Goodman, and J. B. Tenenbaum. A stochastic programming perspective on nonparametric bayes. In *Nonparametric Bayesian Workshop, Int. Conf. on Machine Learning*, volume 22, page 26, 2008.

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.