

Probabilistic Programming with Gaussian Process Memoization

Ulrich Schaechtle

Department of Computer Science

Royal Holloway, University of London

ULRICH.SCHAECHTLE@RHUL.AC.UK

Ben Zinberg

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

BZINBERG@MIT.EDU

Alexey Radul

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

AXOFCH@GMAIL.COM

Kostas Stathis

Department of Computer Science

Royal Holloway, University of London

KOSTAS.STATHIS@RHUL.AC.UK

Vikash K. Mansinghka

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

VKM@MIT.EDU

Editor: N.A.

Abstract

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

1. Introduction

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

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

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

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

2. Background on Gaussian Processes

GPs are a Bayesian method for regression. We consider the regression input to be real-valued scalars x_i and the regression output as the value of a function f at x_i . The complete training data will be denoted by column vectors \mathbf{x} and \mathbf{f} . Unseen test data is denoted with $\hat{\mathbf{x}}$ and $\hat{\mathbf{f}}$. 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.

Definition 1 *A Gaussian process (GP) is a collection of random variables, any finite number of which have a joint Gaussian distribution (Rasmussen and Williams, 2006, chapter 2).*

The collection of random variables $\{f(x)\}$ (indexed by x) represents the values of the function f at each location x . We write $f \sim \text{GP}(m, k)$, where m is the *mean function* and k is the *covariance function* or *kernel*. That is, $m(x)$ is the prior mean of the random variable $f(x)$, and $k(x, x')$ is the prior covariance of the random variables $f(x)$ and $f(x')$. Throughout, we write $\mathbf{K}(\mathbf{x}, \mathbf{x}')$ for the prior covariance matrix determined by \mathbf{x} and \mathbf{x}' , that is, the covariance between the random vectors $\{f(x)\}_{x \in \mathbf{x}}$ and $\{f(x')\}_{x' \in \mathbf{x}'}$.

We now compute the predictive posterior distribution of test output $\hat{\mathbf{f}} := f(\hat{\mathbf{x}})$ conditioned on training data $\mathbf{f} := f(\mathbf{x})$. (Here \mathbf{x} and $\hat{\mathbf{x}}$ are known constant vectors, and we are conditioning on an observed value of \mathbf{f} .) To simplify the calculation, we will assume the prior mean m is identically zero; once the derivation is done, this assumption can be easily relaxed via translation.

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{f} to a constant. To start, let

$$\Sigma := \begin{bmatrix} K(\mathbf{x}, \mathbf{x}) & K(\mathbf{x}, \hat{\mathbf{x}}) \\ K(\hat{\mathbf{x}}, \mathbf{x}) & K(\hat{\mathbf{x}}, \hat{\mathbf{x}}) \end{bmatrix} \text{ and } \Sigma^{-1} =: \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}.$$

We then have

$$P(\mathbf{f}, \hat{\mathbf{f}}) \propto \exp \left\{ -\frac{1}{2} [\mathbf{f}^\top \quad \hat{\mathbf{f}}^\top] \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \begin{bmatrix} \mathbf{f} \\ \hat{\mathbf{f}} \end{bmatrix} \right\}.$$

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

$$P(\hat{\mathbf{f}} \mid \mathbf{f}) \propto P(\mathbf{f}, \hat{\mathbf{f}}) \propto \exp \left\{ -\frac{1}{2} \hat{\mathbf{f}}^\top M_{22} \hat{\mathbf{f}} - \mathbf{h}^\top \hat{\mathbf{f}} \right\},$$

where $\mathbf{h} = M_{21}\mathbf{f}$ is a constant vector. Thus $P(\hat{\mathbf{f}} \mid \mathbf{f})$ is Gaussian,

$$P(\hat{\mathbf{f}} \mid \mathbf{f}) \sim \mathcal{N}(\hat{\boldsymbol{\mu}}, \hat{\mathbf{K}}), \tag{1}$$

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

$$\begin{aligned} P_{\hat{\mathbf{f}}|\mathbf{f}}(\hat{\mathbf{f}} + \hat{\boldsymbol{\mu}} \mid \mathbf{f}) &\propto \exp \left\{ -\frac{1}{2} (\hat{\mathbf{f}} + \hat{\boldsymbol{\mu}})^\top M_{22} (\hat{\mathbf{f}} + \hat{\boldsymbol{\mu}}) - \mathbf{h}^\top (\hat{\mathbf{f}} + \hat{\boldsymbol{\mu}}) \right\} \\ &\propto \exp \left\{ -\frac{1}{2} \hat{\mathbf{f}}^\top M_{22} \hat{\mathbf{f}} - \underbrace{(\mathbf{h} + M_{22}\hat{\boldsymbol{\mu}})^\top \hat{\mathbf{f}}}_{\text{must be 0}} \right\}. \end{aligned}$$

Thus $\mathbf{h} = -M_{22}\hat{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\mu}} = -M_{22}^{-1}\mathbf{h} = -M_{22}^{-1}M_{21}\mathbf{f}$.

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

$$\begin{aligned} M_{22}^{-1} &= K(\hat{\mathbf{x}}, \hat{\mathbf{x}}) - K(\hat{\mathbf{x}}, \mathbf{x})K(\mathbf{x}, \mathbf{x})^{-1}K(\mathbf{x}, \hat{\mathbf{x}}), \\ M_{21} &= -M_{22}K(\hat{\mathbf{x}}, \mathbf{x})K(\mathbf{x}, \mathbf{x})^{-1}. \end{aligned}$$

Substituting these in the above gives

$$\hat{\mathbf{K}} = K(\hat{\mathbf{x}}, \hat{\mathbf{x}}) - K(\hat{\mathbf{x}}, \mathbf{x})K(\mathbf{x}, \mathbf{x})^{-1}K(\mathbf{x}, \hat{\mathbf{x}}), \tag{2}$$

$$\hat{\boldsymbol{\mu}} = K(\hat{\mathbf{x}}, \mathbf{x})K(\mathbf{x}, \mathbf{x})^{-1}\mathbf{f}. \tag{3}$$

Often one assumes the observed regression output is noisily measured, that is, one only sees the values of $\mathbf{y}_{\text{noisy}} = \mathbf{f} + \mathbf{w}$ where \mathbf{w} is Gaussian white noise with variance σ_{noise}^2 . This

noise term can be absorbed into the covariance matrix $\mathbf{K}(\mathbf{x}, \mathbf{x})$ which in the following, we will write as \mathbf{K} for readability. The log-likelihood of a GP can then be written as:

$$\log P(\mathbf{f} | \mathbf{x}) = -\frac{1}{2}\mathbf{y}^\top \mathbf{K}^{-1} \mathbf{y} - \frac{1}{2} \log |\mathbf{K}| - \frac{n}{2} \log 2\pi \quad (4)$$

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

$$\mathbf{K} = \mathbf{L}\mathbf{L}^\top \quad (5)$$

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

$$\mathbf{K}^{-1} = (\mathbf{L}^{-1})^\top (\mathbf{L}^{-1}) \quad (6)$$

and its determinant as

$$\det(\mathbf{K}) = \det(\mathbf{L})^2 \quad (7)$$

We compute (4) as

$$\log(P(\mathbf{f} | \mathbf{x})) := -\frac{1}{2}\mathbf{f}^\top \boldsymbol{\alpha} - \sum_i \log \mathbf{L}_{ii} - \frac{n}{2} \log 2\pi \quad (8)$$

where

$$\mathbf{L} := \text{chol}(\mathbf{K}) \quad (9)$$

and

$$\boldsymbol{\alpha} := \mathbf{L}^\top \backslash (\mathbf{L} \backslash \mathbf{f}). \quad (10)$$

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

The covariance function (or kernel) of a GP governs high-level properties of the observed data such as linearity, periodicity and smoothness. It comes with few free parameters that we call hyper-parameters. Adjusting the hyper-parameters changes non-qualitative attributes such as length scales while preserving the qualitative properties of the distribution. These high-level properties are compositional via addition and multiplication of different covariance functions. An addition 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. A multiplication models a local interaction of two components. An example for this is a periodic function with a linearly increasing amplitude. We demonstrate kernel composition with local interactions in the tutorial in Fig. 1.

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.

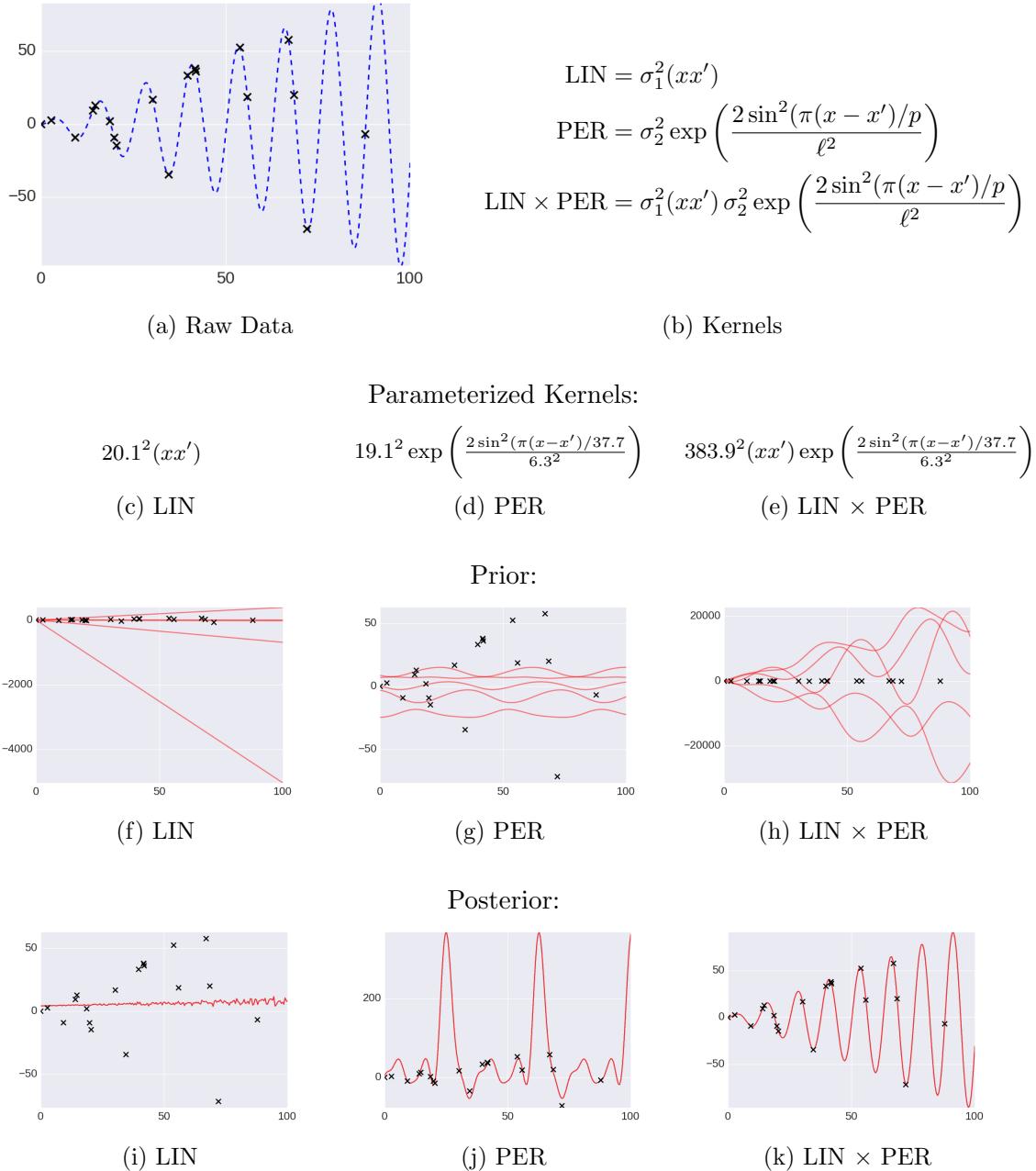


Figure 1: We depict kernel composition. (a) shows raw data (black) generated with a sine function with linearly growing amplitude (blue). (b) shows the linear and the periodic base kernel as well as a composition of both. The multiplication of the two kernels indicates local interaction. The local interaction we account for in this case is the growing amplitude (a). (c-e) show the parameterized kernels that we introduced in (b). The parameters are sampled from the posterior distribution on parameters for a GP with kernel $\mathbf{K} = \text{LIN} \times \text{PER}$ where the data points from (a) are observed. (f-h) show samples from the prior where the parametrization from (c-e) is used, that is, before any data points are observed. (i-k) show samples from the posterior, after the data has been observed. Note that the same data is used for all of the plots.

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:

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

The function f_{compute} calls f and stores the output in a memo table, just as traditional memoization does. The function f_{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

$$(f_{\text{emu}}(x_1, \dots, x_k) \sim P(f(x_1), \dots, f(x_k) \mid f(x) = (f_{\text{compute}}(x)) \text{ for each } x \text{ in memo table}).$$

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 f_{emu} jointly at multiple inputs because the values of $f(x_1), \dots, f(x_k)$ 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 of this figure sketches the schematic of `gpmem`. f is the aforementioned function which can be evaluated using resources that potentially come from outside of Venture. We feed this function into `gpmem` alongside a parameterised kernel \mathbf{K} . The hyper-parameters for this kernel are sampled from a prior distribution which is depicted in the top right box. In Venture, we can express this as:

```
assume sf = tag( 'hyper-parameter', 0, gamma(1,1))
assume l  = tag( 'hyper-parameter', 1, gamma(1,1))
assume K  = make_squaredexp( sf, l ).
```

Note that the hyper-priors over hyper-parameters follows a gamma distribution and we annotate `sf` and `l` as belonging to (i) the scope "hyper-parameter" and (ii) blocks 0 and 1 respectively. The squared-exponential kernel (SE) is defined in appendix A.

`gpmem` implements a memoization table, where all previously computed function evaluations are stored. We also initialize a GP-prior that will serve as our statistical emulator. All value pairs stored in the memoization table are incorporated as observations of the GP. The emulator allows us to make probabilistic predictions on function evaluations. 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 second panel Fig. 2) or we interleave Venture with foreign code. This can be useful when f is computed with the help of outside resources. Before making any observations or calls to f we can sample from the prior at the inputs from -20 to 20 using the emulator:

```
assume (f_compute f_emu) = gpmem( f, K )
sample f_emu ( array( -20, ..., 20 ) )
```

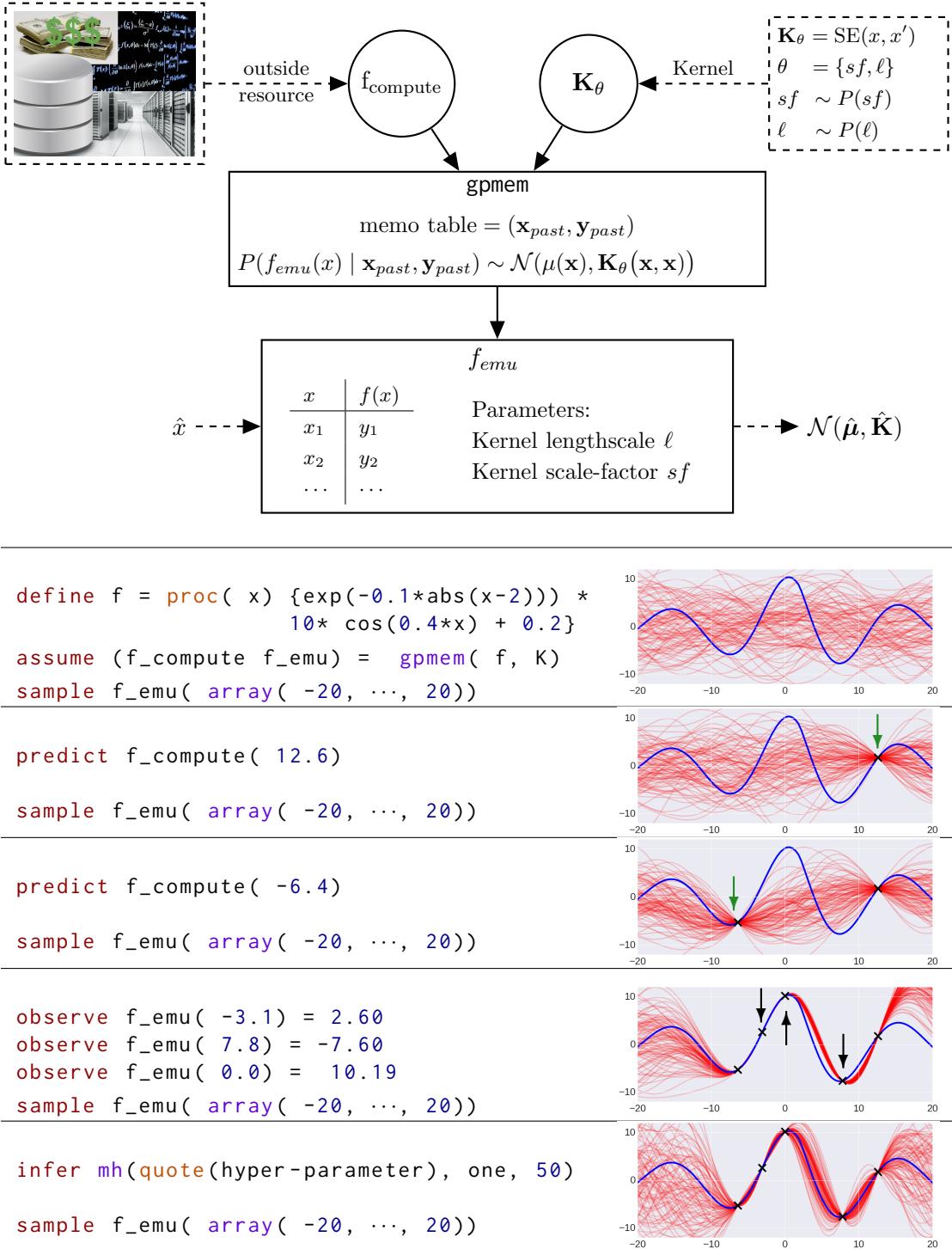


Figure 2: gpmem tutorial. The top shows a schematic of gpmem. f_{compute} probes an outside resource. This can be expensive (top left). Every probe is memoized and improves the emulator. Below the schematic we see the evolution of gpmem's state of believe of the world given certain Venture directives. On the right, we depict the true function (blue), samples from the emulator (red) and incorporated observations (black).

In the following panel of Fig. 2, 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, we now compute the GP posterior which shifts from uncertainty to near certainty close to the input 12.6.

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

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

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

We have this value pair now available for the computation of the GP posterior. For sampling with the emulator, the effect is the same as calling `predict` with the f_{compute} . However, we can imagine at least one scenario where such a 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. We can take 50 Metropolis-Hastings (MH) steps by calling that where we take an MH steps for either of the two blocks for the scope "hyper-parameter":

```
infer mh( 'hyper-parameter', one, 50 ).
```

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

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

In a Bayesian treatment of hyper-parameter learning for GPs, we can write the probability of the hyper-parameters of a GP as defined above, given covariance function \mathbf{K} as:

$$P(\boldsymbol{\theta} | \mathbf{D}, \mathbf{K}) = \frac{P(\mathbf{D} | \boldsymbol{\theta}, \mathbf{K})P(\boldsymbol{\theta} | \mathbf{K})}{P(\mathbf{D} | \mathbf{K})}. \quad (11)$$

Let \mathbf{K} be the sum of a smoothing and a white noise (WN) kernel. For this case, Neal (1997) suggested the problem of outliers in data as a use-case for a hierarchical Bayesian treatment of Gaussian processes¹. We can tackle this problem using `gpmem`, as illustrated with the tutorial in Fig. 3. The main aim here is to estimate the posterior of (11) which we depict in panel (a) of Fig. 3.

To arrive at this posterior, we must first have access to the data used in this problem. We define the data generating function as in Neal’s paper. Let f be the underlying function that generates the data:

$$f(x) = 0.3 + 0.4x + 0.5 \sin(2.7x) + \frac{1.1}{(1+x^2)} + \eta \quad \text{with } \eta \sim \mathcal{N}(0, \sigma) \quad (12)$$

We synthetically generate outliers by setting $\sigma = 0.1$ in 95% of the cases and to $\sigma = 1$ in the remaining cases. This data set is accessed by a look-up function in Venture (Fig. 3 (b)).

Neals work suggests a hierarchical system of hyper-parameterization. Here, we draw hyper-parameters from Γ distributions:

$$\ell^{(t)} \sim \Gamma(\alpha_1, \beta_1), \sigma^{(t)} \sim \Gamma(\alpha_2, \beta_2) \quad (13)$$

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

$$\alpha_1^{(t)} \sim \Gamma(\alpha_\alpha^1, \beta_\alpha^1), \alpha_2^{(t)} \sim \Gamma(\alpha_\alpha^2, \beta_\alpha^2), \dots \quad (14)$$

We model this in Venture as illustrated in Fig. 3 (c). Note that we omit the prior distributions of (14) due limited space in the tutorial.

In Fig. 3, panel (c), we first construct a composite Kernel by adding a squared-exponential covariance function to a white noise covariance function (SE and WN, Appendix A). We then initialize `gpmem` feeding it with the composite covariance and the data look-up function. We sample from the prior with random parameters `sf, 1` and `sigma` and without any observations available. We depict those samples on the right (red), alongside the true function that generated the data (blue) and the data points we have available in the data set (black).

We can incorporate observations using both `observe` and `predict` (Fig. 3 (e)). When we subsequently sample from the emulator, we can now see that the GP posterior incorporates knowledge about the data points. Yet, the hyper-parameters are still random, so the emulator does not capture the true underlying dynamics of the data correctly.

Next, we demonstrate how we can capture these underlying dynamics within only 100 MH steps on the hyper-parameters to get a good approximation for their posterior (Fig.

1. In Neal’s work (1997) the sum of an SE plus a constant kernel is used. We keep the WN kernel for illustrative purposes.

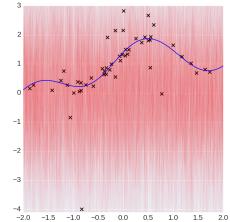
(a) Hyper-Parameters for the Kernel:



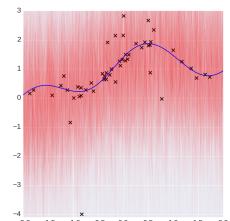
```
(b) // Data and look-up function
define data = array(array(-1.87,0.13),..., array(1.67,0.81))
assume f_look_up = proc(index) {lookup( data, index)}
```

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

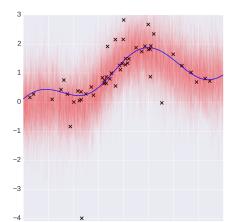
```
// The covariance function
assume se = make_squaredexp(sf, l)
assume wn = make_whitenoise(sigma)
assume composite_covariance = add_funcs(se, wn)
(d) // Create a prober and emulator using gpmem
assume (f_compute, f_emu)
= gpmem(f_look_up, composite_covariance)
sample f_emu( array( -2, ..., 2))
```



```
// Observe all data points
for n ... N
    observe f_emu(first(lookup(data,n)))
        = second(lookup(data,n))
// Or: probe all data points
for n ... N
    predict f_compute(first(lookup(data,n)))
sample f_emu( array( -2, ..., 2))
```



```
// Metropolis-Hastings
(f) infer repeat( 100, do(
    mh( quote(hyperhyper), one, 2),
    mh( quote(hyper), one, 1)))
sample f_emu( array( -2, ..., 2))
```



```
// Optimization
(g) infer map( quote(hyper), all, 0.01, 15)
sample f_emu( array( -2, ..., 2))
```

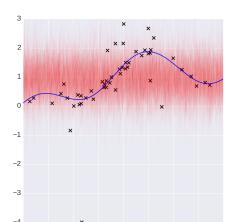


Figure 3: Regression with outliers and hierarchical prior structure

3 (f)). 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 (Fig. 3 (g)).

4.2 Discovering qualitative structure from time series data

Inductive learning of symbolic expression 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).

The stochastic grammar models a prior on different structural compositions of the covariance function. An input of non-composite kernels (base kernels) is supplied to generate a posterior distributions of composite structure to express local and global aspects of the data.

We approximate the following intractable integrals of the expectation for the prediction:

$$\mathbb{E}[\hat{f} | \hat{x}, \mathbf{D}, \mathbf{K}] = \iint f(\hat{x}, \boldsymbol{\theta}, \mathbf{K}) P(\boldsymbol{\theta} | \mathbf{D}, \mathbf{K}) P(\mathbf{K} | \boldsymbol{\Omega}, s, n) d\boldsymbol{\theta} d\mathbf{K}. \quad (15)$$

This is done by sampling from the posterior probability distribution of the hyperparameters and the possible kernel:

$$\hat{f} \approx \frac{1}{T} \sum_{t=1}^T f(\hat{x} | \boldsymbol{\theta}^{(t)}, \mathbf{K}^{(t)}). \quad (16)$$

In order to provide the sampling of the kernel, we introduce a stochastic process that simulates the grammar for algebraic expressions of covariance function algebra:

$$\mathbf{K}^{(t)} \sim P(\mathbf{K} | \boldsymbol{\Omega}, s, n) \quad (17)$$

Here, we start with the set of given base kernels and draw a random subset. For this subset of size n , we sample a set of possible operators $\boldsymbol{\Omega}$ combining base kernels. The marginal probability of a composite structure

$$P(\mathbf{K} | \boldsymbol{\Omega}, s, n) = P(\boldsymbol{\Omega} | s, n) \times P(s | n) \times P(n), \quad (18)$$

is characterized by the prior $P(n)$ on the number of base kernels used, the probability of a uniformly chosen subset of the set of n possible covariance functions

$$P(s | n) = \frac{n!}{|s|!}, \quad (19)$$

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

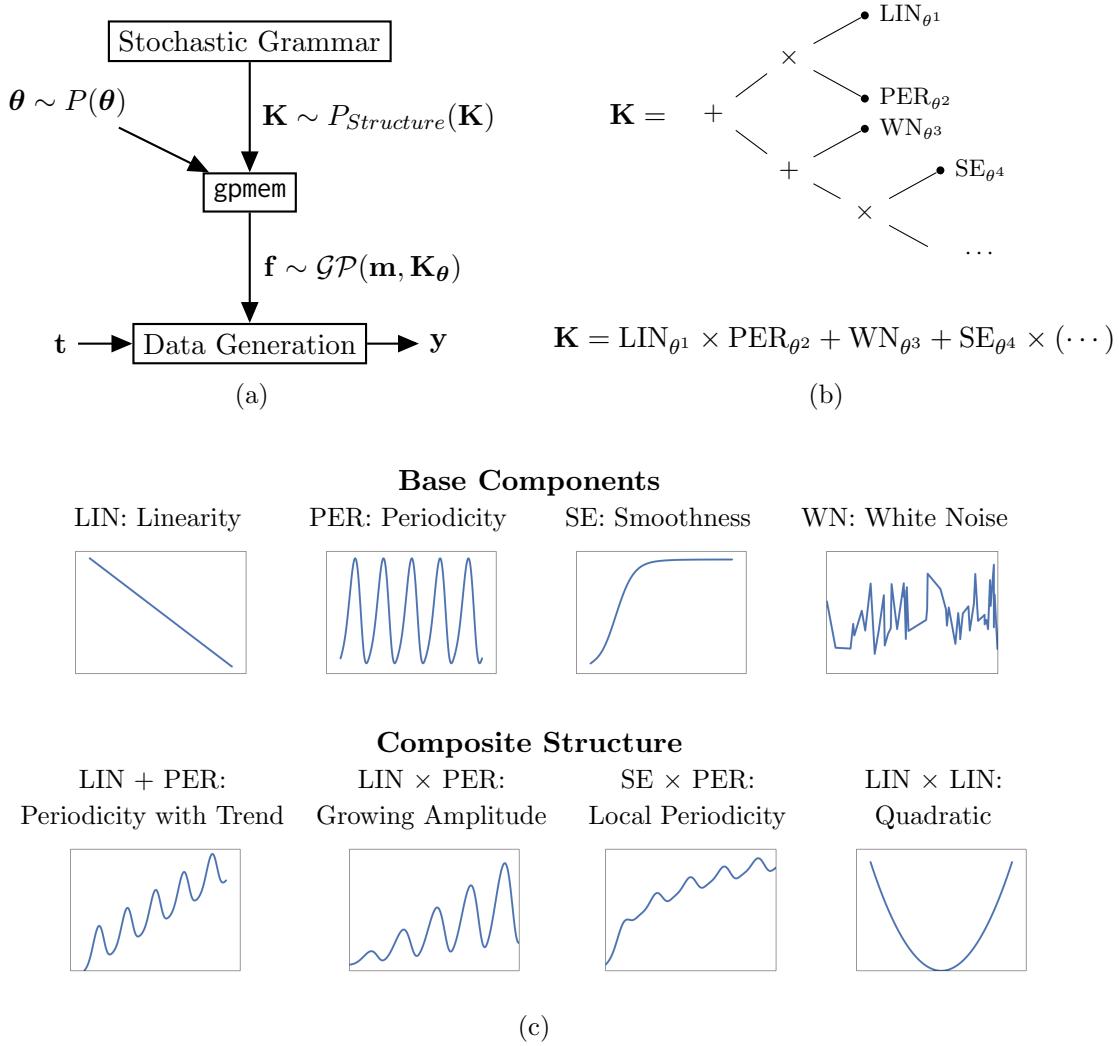


Figure 4: (a) Graphical description of Bayesian GP structure learning. (b) Composite structure. (c) The natural language interpretation of the structure.

and the probability of sampling a global or a local structure, which is given by a binomial distribution:

$$P(\Omega | s, n) = \binom{n}{r} p_{+ \times}^r (1 - p_{+ \times})^{n-r}. \quad (20)$$

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. All base kernels can be found in Appendix A, rules for this simplification can be found in appendix B. The code for learning of kernel structure is as follows:

```

// GRAMMAR FOR KERNEL STRUCTURE
1 assume kernels = list(se, wn, lin, per, rq) // defined as above

2 // prior on the number of kernels
3 assume p_number_k = uniform_structure(n)
4 assume subset_kernels = tag(quote(grammar), 0,
                           subset(kernels, p_number_k))

// kernel composition
5 assume composition = proc(l) {
6   if (size(l) <= 1)
7     { first(l) }
8   else { if (bernoulli())
9         { add_funcs(first(l), composition(rest(l))) }
10        else { mult_funcs(first(l), composition(rest(l))) }
11      }
12 }

13 assume K = tag(quote(grammar), 1, composition(subset_kernels))

14 assume (f_compute f_emu) = gpmem(f_look_up, K)

// Probe all data points
15 for n ... N
16   predict f_compute(get_data_xs(n))

// PERFORMING INFERENCE
17 infer repeat(200, do(
18   mh(quote(grammar), one, 1),
19   for kernel in K:
20     mh(quote(parameters_kernel), one, 1)))

```

We defined a simple space of covariance structures in a way that allows us to produce results coherent with work presented in Automatic Statistician. We illustrate results in Fig. 5 and 6 using the Mauna Loa CO₂ data set (see Rasmussen and Williams, 2006 for a description) and the airline data set describing monthly totals of international airline

passengers (Box et al., 1997, according to Duvenaud et al., 2013). Both datasets served as illustrations in the Automatic Statistician project. 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)).

In contrast to what previous work has presented, we compute a posterior on structures. This allows us to gain valuable insight into the data that was previously unavailable. We can query the data for the probability of certain structures to hold true. For example, we could be interested in whether or not a trend is present in the data. We can also formulate queries using logical operators such as AND and OR. This provides us with a rich language to ask questions about our time series data and let the data speak for itself for finding the answers. We demonstrate this in Fig 7.

4.3 Bayesian optimization

The final application demonstrating the power of `gpmem` demonstrates 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, and describe how Thompson sampling can be used to choose actions for that Markov Decision Processes (MDP).

Here, an agent is to take a sequence of actions a_1, a_2, \dots from a (possibly infinite) set of possible actions \mathcal{A} . After each action, a reward $r \in \mathbb{R}$ is received, according to an unknown conditional distribution $P_{\text{true}}(r | a)$. 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 $\{P(r | a)\}_{a \in \mathcal{A}}$, or at least, a believed statistic of these conditional distributions which is sufficient for deciding an action a . If actions are chosen so as to maximize expected reward, then one such sufficient statistic is the believed conditional mean $V(a | \vartheta) = \mathbb{E}[r | a; \vartheta]$, which can be viewed as a believed value function. For consistency with what follows, we will assume our context ϑ takes the form $(\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$ where \mathbf{a}_{past} is the vector of past actions, \mathbf{r}_{past} is the vector of their rewards, and θ (the "semicontext") contains any other information that is included in the context.

In this setup, Thompson sampling has the following steps:

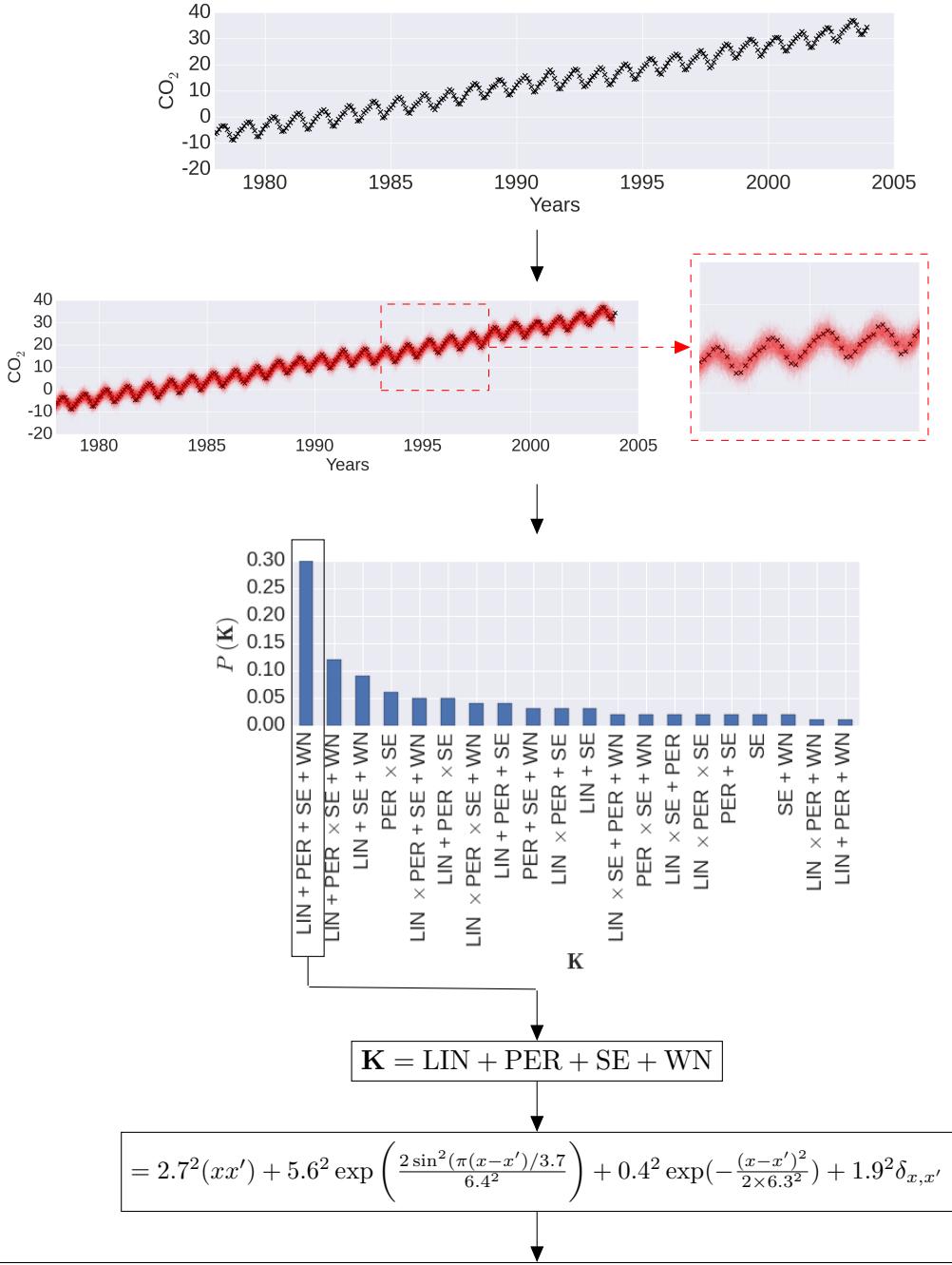
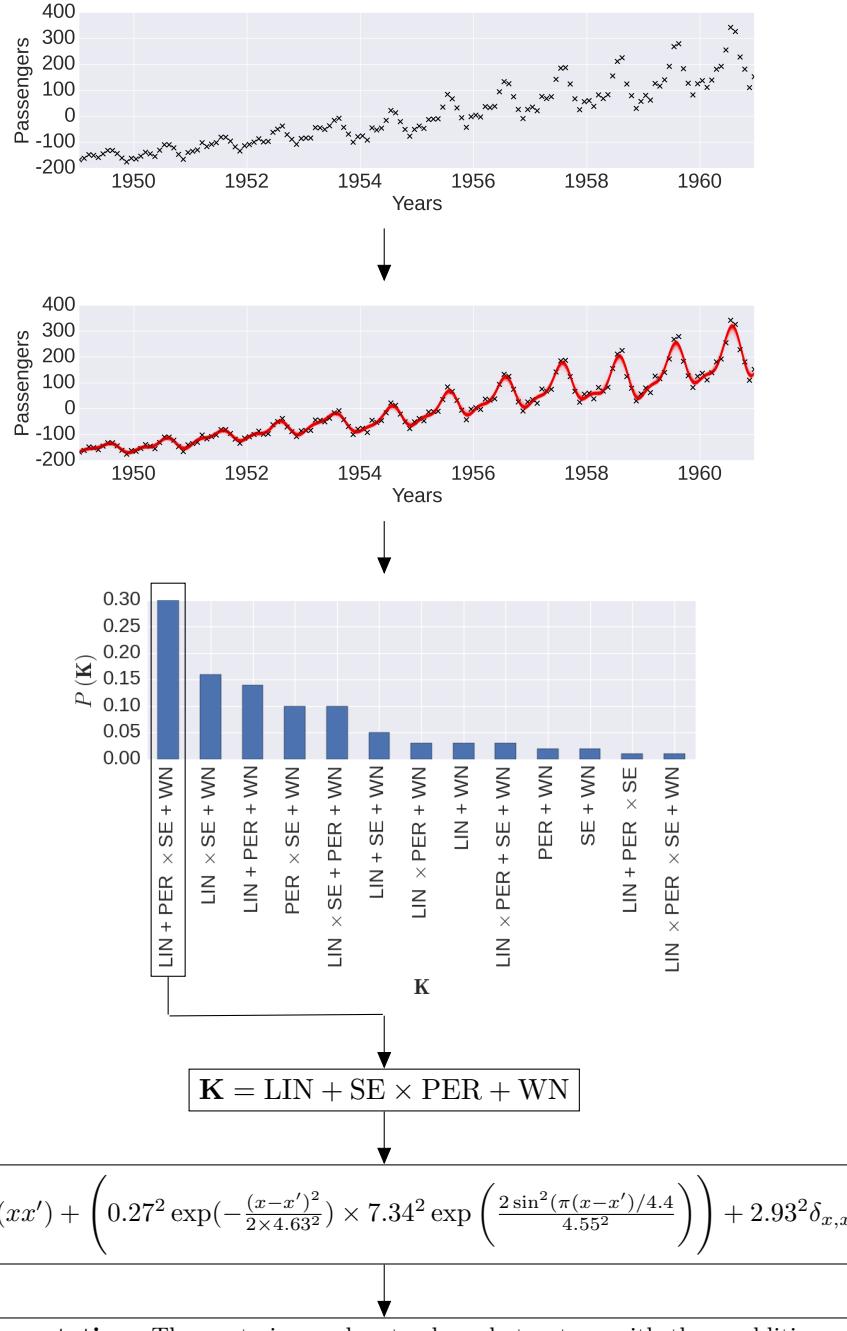


Figure 5: Posterior of structure and qualitative, human interpretable reading. We take the raw data (top), compute a posterior distribution on structures (red samples and bar plot). We take the peak of this distribution ($\text{LIN} + \text{PER} + \text{SE} + \text{WN}$) with the sampled parameters used to generate the samples for the second plot from the top and write it in functional form with parameters. We depict the human readable interpretation of the equation on the bottom.



Qualitative Interpretation: 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) a approximate periodic function; and (iv) white noise.

Figure 6: Posterior of structure and qualitative, human interpretable reading. We take the raw data (top), compute a posterior distribution on structures (red samples and bar plot). We take the peak of this distribution ($\text{LIN} + \text{PER} \times \text{SE} + \text{WN}$) with the sampled parameters used to generate the samples for the second plot from the top and write it in functional form with parameters. We depict the human readable interpretation of the equation on the bottom.

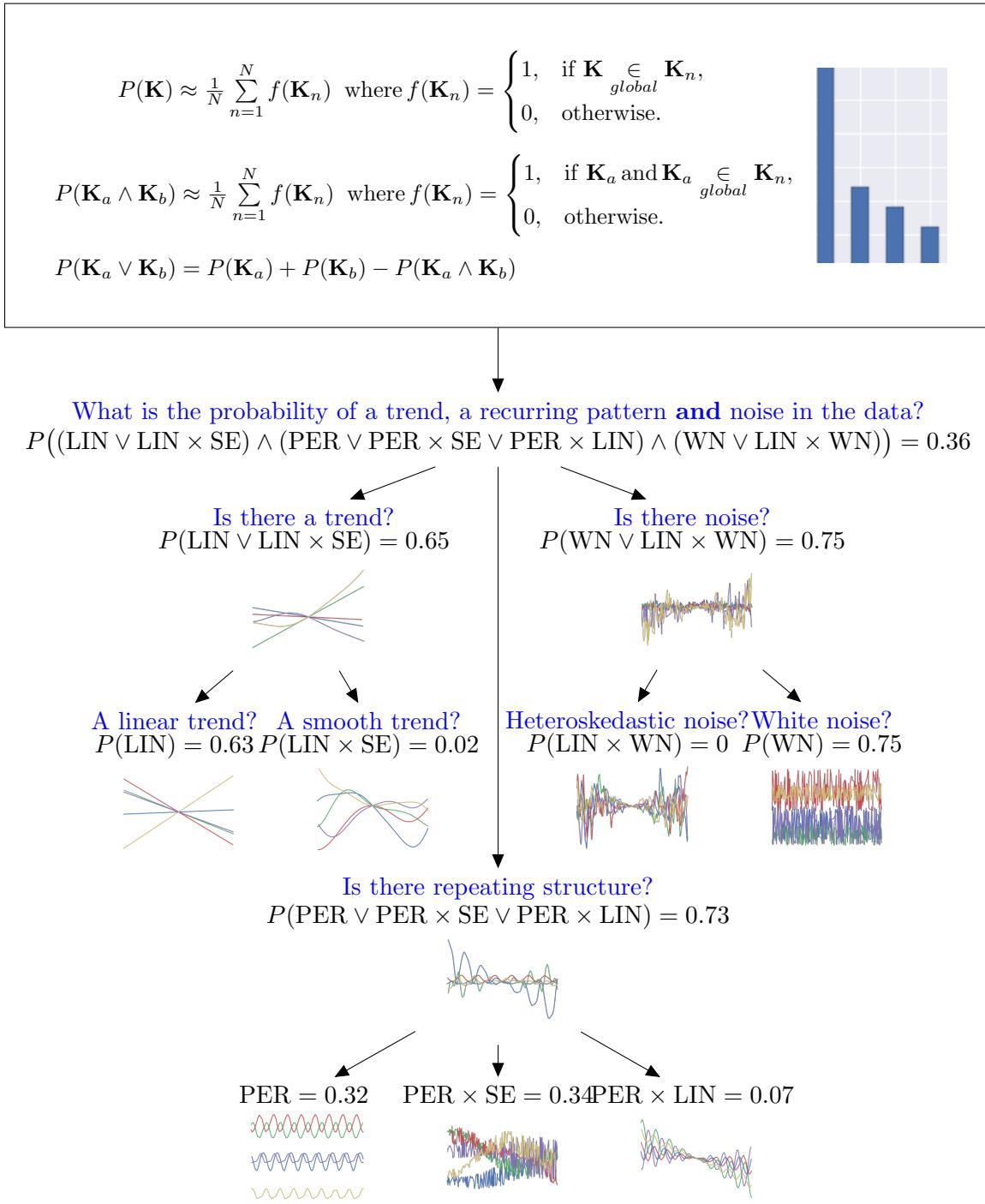


Figure 7: We can query the data if some logical statements are probable to be true, for example, is it true that there is a trend?

Algorithm 1 Thompson sampling.

Repeat as long as desired:

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

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

Typically, when Thompson sampling is implemented, the search over contexts $\vartheta \in \Theta$ is limited by the choice of representation. In traditional programming environments, θ 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 programming language, however, the representation of heterogeneously structured or infinite-dimensional context spaces is quite natural. Any computable model of the conditional distributions $\{P(r | a)\}_{a \in \mathcal{A}}$ can be represented as a stochastic procedure $(\lambda(a) \dots)$. 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

Our mathematical specification assert the following properties:

- The regression function has a Gaussian process prior.
- The actions $a_1, a_2, \dots \in \mathcal{A}$ 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 $\mathcal{A} = [-20, 20] \subseteq \mathbb{R}$. That is,

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

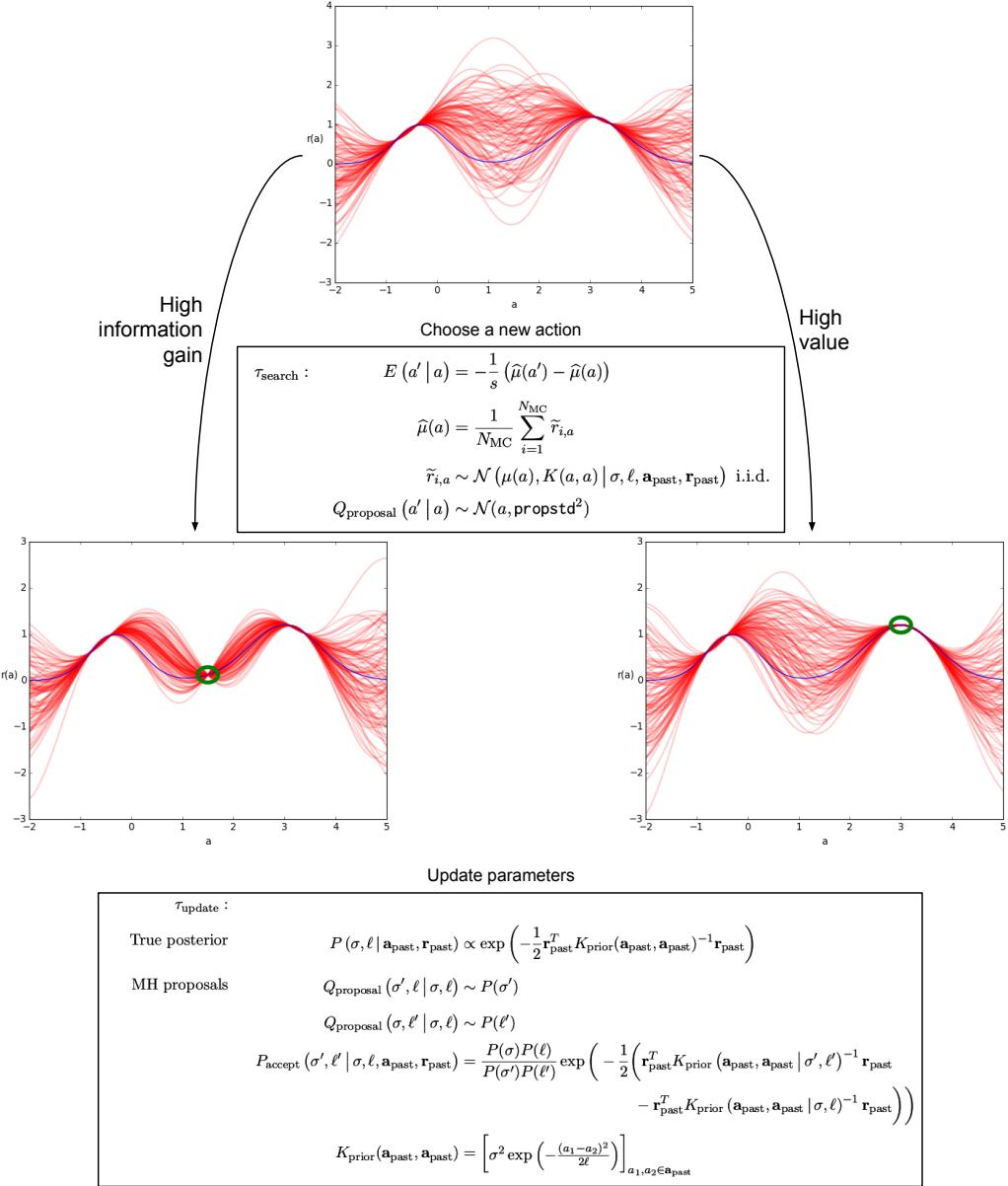


Figure 8: Two possible actions (in green) for an iteration of Thompson sampling. The believed distribution on the value function V is depicted in red. In this example, the true reward function is deterministic, and is drawn in blue. The action on the right receives a high reward, while the action on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

where the mean μ is a computable function $\mathcal{A} \rightarrow \mathbb{R}$ and the covariance K is a computable (symmetric, positive-semidefinite) function $\mathcal{A} \times \mathcal{A} \rightarrow \mathbb{R}$. This represents a Gaussian process $\{R_a\}_{a \in \mathcal{A}}$, where R_a represents the reward for action a . Computationally, we represent a context as a data structure

$$\vartheta = (\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}) = (\mu_{\text{prior}}, K_{\text{prior}}, \eta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}),$$

where μ_{prior} is a procedure to be used as the prior mean function. w.l.o.g. we set $\mu_{\text{prior}} \equiv 0$. K_{prior} is a procedure to be used as the prior covariance function, parameterized by η .

The posterior mean and covariance for such a context ϑ are gotten by the usual conditioning formulas (assuming, for ease of exposition as above, that the prior mean is zero):³

$$\begin{aligned}\mu(\mathbf{a}) &= \mu(\mathbf{a} \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}) \\ &= K_{\text{prior}}(\mathbf{a}, \mathbf{a}_{\text{past}}) K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1} \mathbf{r}_{\text{past}} \\ K(\mathbf{a}, \mathbf{a}) &= K(\mathbf{a}, \mathbf{a} \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}) \\ &= K_{\text{prior}}(\mathbf{a}, \mathbf{a}) - K_{\text{prior}}(\mathbf{a}, \mathbf{a}_{\text{past}}) K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1} K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}).\end{aligned}$$

Note that the context space Θ is not a finite-dimensional parametric family, since the vectors \mathbf{a}_{past} and \mathbf{r}_{past} 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 $\vartheta = (\mu_{\text{prior}}, K_{\text{prior}}, \eta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$.

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

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

and

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

The MH acceptance probability for such a proposal is

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

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

$$\begin{aligned}P_{\text{accept}}(\sigma', \ell' \mid \sigma, \ell) &= \min \left\{ 1, \frac{P(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma', \ell')}{P(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma, \ell)} \right\} \\ &= \min \left\{ 1, \exp \left(-\frac{1}{2} \left(\mathbf{r}_{\text{past}}^T K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}} \mid \sigma', \ell')^{-1} \mathbf{r}_{\text{past}} \right. \right. \right. \\ &\quad \left. \left. \left. - \mathbf{r}_{\text{past}}^T K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}} \mid \sigma, \ell)^{-1} \mathbf{r}_{\text{past}} \right) \right) \right\}.\end{aligned}$$

3. Here, for vectors $\mathbf{a} = (a_i)_{i=1}^n$ and $\mathbf{a}' = (a'_i)_{i=1}^{n'}$, $\mu(\mathbf{a})$ denotes the vector $(\mu(a_i))_{i=1}^n$ and $K(\mathbf{a}, \mathbf{a}')$ denotes the matrix $[K(a_i, a'_j)]_{1 \leq i \leq n, 1 \leq j \leq n'}$.

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 θ 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 a . The Markov chain's initial state is the most recent action, and the proposal distribution is Gaussian drift:

$$Q_{\text{proposal}}(a' | a) \sim \mathcal{N}(a, \text{propstd}^2),$$

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

$$P_{\text{accept}}(a' | a) = \min \{1, \exp(-E(a' | a))\},$$

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

$$E(a' | a) = -\frac{1}{s} (\hat{\mu}(a') - \hat{\mu}(a))$$

where

$$\hat{\mu}(a) = \frac{1}{N_{\text{avg}}} \sum_{i=1}^{N_{\text{avg}}} \tilde{r}_{i,a}$$

and

$$\tilde{r}_{i,a} \sim \mathcal{N}(\mu(a), K(a, a))$$

and $\{\tilde{r}_{i,a}\}_{i=1}^{N_{\text{avg}}}$ are i.i.d. for a fixed a . 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 a at which the posterior mean $\mu(a)$ is low but the posterior variance $K(a, a)$ is high, it is possible (especially when N_{avg} is small) to draw a “wild” value of $\hat{\mu}(a)$, resulting in a favorable acceptance probability.

Indeed, taking an action a with low estimated value but high uncertainty serves the useful function of improving the accuracy of the estimated value function at points near a (see Figure 8).^{4,5} We see a complete probabilistic program with `gpmem` implementing Bayesian optimization with Thompson Sampling below (Listing 1).

4. At least, this is true when we use a smoothing prior covariance function such as the squared exponential.

5. For this reason, we consider the sensitivity of $\hat{\mu}$ to uncertainty to be a desirable property; indeed, this is why we use $\hat{\mu}$ rather than the exact posterior mean μ .

Listing 1: Bayesian optimization using `gpmem`

```

1  assume sf = tag(quote(hyper), 0, uniform_continuous(0, 10))
2  assume l = tag(quote(hyper), 1, uniform_continuous(0, 10))
3  assume se = make_squaredexp(sf, l)
4  assume blackbox_f = get_bayesopt_blackbox()
5  assume (f_compute, f_emulate) = gpmem(blackbox_f, se)

// A naive estimate of the argmax of the given function
6  define mc_argmax = proc(func) {
7    candidate_xs = mapv(proc(i) {uniform_continuous(-20, 20)},
8                        arange(20));
9    candidate_ys = mapv(func, candidate_xs);
10   lookup(candidate_xs, argmax_of_array(candidate_ys))
11};

// Shortcut to sample the emulator at a single point without packing
// and unpacking arrays
12 define emulate_pointwise = proc(x) {
13   run(sample(lookup(f_emulate(array(unquote(x))), 0)))
14};

// Main inference loop
15 infer repeat(15, do(pass,
16   // Probe V at the point mc_argmax(emulate_pointwise)
17   predict(f_compute(unquote(mc_argmax(emulate_pointwise))),
18   // Infer hyperparameters
19   mh(quote(hyper), one, 50)));

```

5. Discussion

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

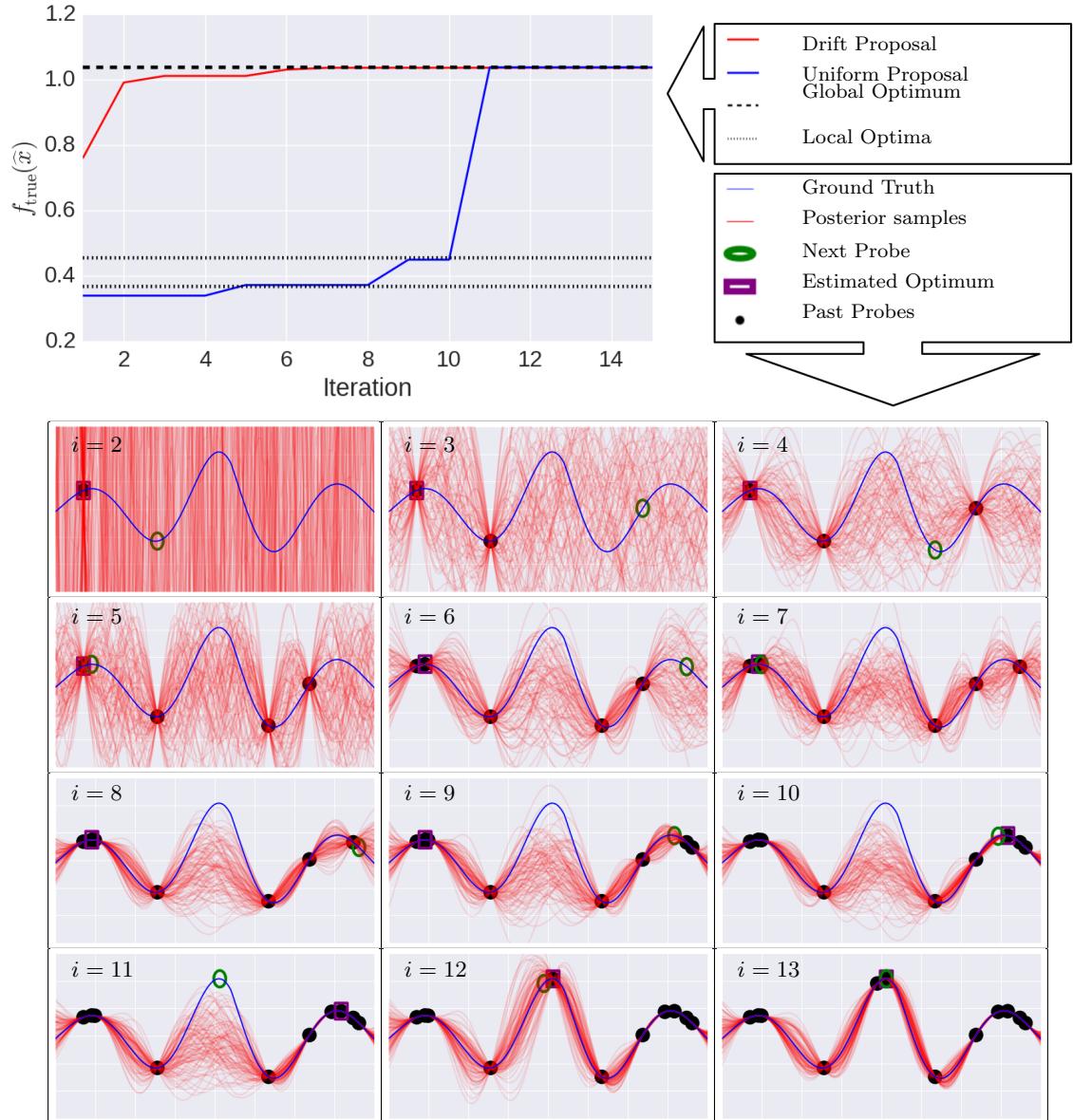


Figure 9: Top: the estimated optimum over time. Blue and Red represent optimization with uniform and Gaussian drift proposals. Black lines indicate the local optima of the true functions. Bottom: a sequence of actions. Depicted are iterations 7-12 with uniform proposals.

Appendix

A Covariance Functions

$$\text{SE} = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right) \quad (21)$$

$$\text{LIN} = \sigma^2(x x') \quad (22)$$

$$\text{C} = \sigma^2 \quad (23)$$

$$\text{WN} = \sigma^2 \delta_{x,x'} \quad (24)$$

$$\text{RQ} = \sigma^2 \left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha} \quad (25)$$

$$\text{PER} = \sigma^2 \exp\left(\frac{2 \sin^2(\pi(x - x')/p)}{\ell^2}\right). \quad (26)$$

B Covariance Simplification

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

Rule 1 is derived as follows:

$$\begin{aligned} \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) &= \sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \sigma_b^2 \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2} - \frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \end{aligned} \quad (27)$$

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:

$$\sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \times \sigma_b \delta_{x,x'} = \sigma_a \sigma_b \delta_{x,x'} \quad (28)$$

Rule 3 is derived as follows:

$$\theta_c(x \times x') = \theta_a(x \times x') + \theta_b(x \times x') \quad (29)$$

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

C Glossary

x	Scalar
X	Random variable
\mathbf{x}	Column vector
\mathbf{X}	Matrix
\mathcal{D}	Data matrix
\mathbb{E}	Expectation
\mathcal{N}	(Multivariate-) Gaussian
\mathcal{GP}	Gaussian Process
μ	Mean
$\boldsymbol{\mu}$	Mean vector
σ	Standard deviation
$m(x)$	Mean function
$k(x, x')$	Kernel or covariance function
$\mathbf{K} = \mathbf{K}(\mathbf{x}, \mathbf{x})$	Prior Kernel or covariance function given a data vector
$\hat{\mathbf{x}}$	unseen input data
$\mathbf{K}(\mathbf{x}, \hat{\mathbf{x}})$	Cross covariance
$f = f(x)$	One-dimensional sample from a GP
\mathbf{f}	Sample from a GP prior \ regression target, i.e. observed output
$\hat{\mathbf{f}}$	Sample from a predictive posterior, that is regression output for unseen data $\hat{\mathbf{x}}$
$\hat{\mu}$	Posterior mean for $\hat{\mathbf{f}}$
$\hat{\mathbf{K}}$	Posterior covariance for $\hat{\mathbf{f}}$
SE	Squared exponential covariance function on a data vector
LIN	Linear covariance function on a data vector
PER	Periodic covariance function on a data vector
RQ	Rational quadratic covariance function on a data vector
C	Constant covariance function on a data vector
WN	White noise covariance function on a data vector
\wedge	Logical and
\vee	Logical or

References

- Stephen Barnett and Stephen Barnett. *Matrix methods for engineers and scientists*. McGraw-Hill, 1979.
- D. Barry. Nonparametric bayesian regression. *The Annals of Statistics*, 14(3):934–953, 1986.
- G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time series analysis: forecasting and control*. 1997.
- A. Damianou and N. Lawrence. Deep gaussian processes. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 207–215, 2013.
- D. Duvenaud, J. R. Lloyd, R. Grosse, J. Tenenbaum, and Z. Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1166–1174, 2013.
- B. Ferris, D. Haehnel, and D. Fox. Gaussian processes for signal strength-based location estimation. In *Proceedings of the Conference on Robotics Science and Systems*. Citeseer, 2006.
- M. A. Gelbart, J. Snoek, and R. P. Adams. Bayesian optimization with unknown constraints. *arXiv preprint arXiv:1403.5607*, 2014.
- 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.
- 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.
- 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.
- 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.