

---

# Probabilistic Programming with Gaussian Process Memoization

---

009                   **Anonymous Author(s)**

010                   Affiliation

011                   Address

012                   email

## 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

Probabilistic programming could be revolutionary for machine intelligence due to universal inference engines and the rapid prototyping for novel models (Ghahramani, 2015). Probabilistic programming languages aim to provide a formal language to specify probabilistic models in the style of computer programming and can represent any computable probability distribution as a program. We can write such programs in Venture (Mansinghka et al., 2014), the first probabilistic programming language suitable for general purpose use. Venture comes with scalable performance on hard problems and with a general purpose inference engine.

A family of statistical distributions rarely treated in probabilistic programming languages are Gaussian Process (GP). GPs are gaining increasing attention for representing unknown functions by posterior probability distributions in various fields such as machine learning (Rasmussen and Williams, 2006), signal processing (Clifton et al., 2013), computer vision (Kemmler et al., 2013) and biomedical data analysis (Shepherd and Owenius, 2012). When incorporated in probabilistic programming, GPs allow us to treat a wide range of problems naturally within the same unified semantic framework. Examples for hard problems are hierarchical prior construction (Neal, 1997), systems for inductive learning of symbolic expressions such as the one introduced in the Automated Statistician project (Duvenaud et al., 2013; Lloyd et al., 2014) and Bayesian Optimization (Snoek et al., 2012).

In the following, we will present GP memoization (`gpmem`) as a novel probabilistic programming technique that solves such hard problems and provides the necessary semantic framework. `gpmem` introduces a statistical alternative to standard memoization. Our contribution is threefold. Firstly, we introduce an efficient implementation of `gpmem` in form of a self-caching wrapper that remembers

054 previously computed values. Secondly, we illustrate the statistical emulator that gpmem produces and  
 055 how it improves with every data point that becomes available. Finally, we show how one can solve  
 056 hard problems of state-of-the-art machine learning related to GP using gpmem. In the following,  
 057 we will introduce GP memoization. We will start by introducing gpmem. We then elaborate on its  
 058 use at hand of the problems mentioned above. To evaluate the performance we use synthetic and  
 059 real-world data.

## 2 Gaussian Processes Memoization

063 Let the data be pairs of real-valued scalars  $\{(x_i, y_i)\}_{i=1}^n$  (complete data will be denoted by column  
 064 vectors  $\mathbf{x}, \mathbf{y}$ ). GPs present a non-parametric way to express prior knowledge on the space of possible  
 065 functions  $f$  modeling a regression relationship. Formally, a GP is an infinite-dimensional extension  
 066 of the multivariate Gaussian distribution. Often one assumes the values  $\mathbf{y}$  are noisily measured, that  
 067 is, one only sees the values of  $\mathbf{y}_{\text{noisy}} = \mathbf{y} + \mathbf{w}$  where  $\mathbf{w}$  is Gaussian white noise with variance  $\sigma_{\text{noise}}^2$ .  
 068 In that case, the log-likelihood of a GP is

$$069 \log p(\mathbf{y}_{\text{noisy}} | \mathbf{x}) = -\frac{1}{2}\mathbf{y}^\top(\Sigma + \sigma_{\text{noise}}^2\mathbf{I})^{-1}\mathbf{y} - \frac{1}{2}\log|\Sigma + \sigma_{\text{noise}}^2\mathbf{I}| - \frac{n}{2}\log 2\pi \quad (1)$$

070 where  $n$  is the number of data points. Both log-likelihood and predictive posterior can be computed  
 071 efficiently in a Venture SP with an algorithm that resorts to Cholesky factorization(Rasmussen and  
 072 Williams, 2006, chap. 2) resulting in a computational complexity of  $\mathcal{O}(n^3)$  in the number of data  
 073 points.  
 074

075 The covariance function (or kernel) of a GP governs high-level properties of the observed data such  
 076 as linearity, periodicity and smoothness. It comes with few free parameters that we call hyper-  
 077 parameters. Adjusting these results in minor changes, for example with regards to when two data  
 078 points are treated similar. More drastically different covariance functions are achieved by changing  
 079 the structure of the covariance function itself. Note that covariance function structures are compo-  
 080 sitional: adding or multiplying two valid covariance functions results in another valid covariance  
 081 function.  
 082

083 Venture includes the primitive `make_gp`, which takes as arguments a unary function `mean` and a  
 084 binary (symmetric, positive-semidefinite) function `cov` and produces a function `g` distributed as a  
 085 Gaussian process with the supplied mean and covariance. For example, a function  $g \sim \mathcal{GP}(0, \text{SE})$ ,  
 086 where `SE` is a squared-exponential covariance

$$087 \text{SE}(x, x') = \sigma^2 \exp\left(\frac{(x - x')^2}{2\ell}\right)$$

088 with  $\sigma = 1$  and  $\ell = 1$ , can be instantiated as follows:  
 089

```
090 assume zero = make_const_func( 0.0 )
091 assume se = make_squaredexp( 1.0, 1.0 )
092 assume g = make_gp( zero, se )
```

093 There are two ways to view `g` as a “random function.” In the first view, the `assume` directive that  
 094 instantiates `g` does not use any randomness—only the subsequent calls to `g` do—and coherence  
 095 constraints are upheld by the interpreter by keeping track of which evaluations of `g` exist in the current  
 096 execution trace. Namely, if the current trace contains evaluations of `g` at the points  $x_1, \dots, x_N$  with  
 097 return values  $y_1, \dots, y_N$ , then the next evaluation of `g` (say, jointly at the points  $x_{N+1}, \dots, x_{N+n}$ )  
 098 will be distributed according to the joint conditional distribution  
 099

$$100 P((g x_{N+1}), \dots, (g x_{N+n}) | (g x_i) = y_i \text{ for } i = 1, \dots, N).$$

101 In the second view, `g` is a randomly chosen deterministic function, chosen from the space of all  
 102 deterministic real-valued functions; in this view, the `assume` directive contains *all* the randomness,  
 103 and subsequent invocations of `g` are deterministic. The first view is procedural and is faithful to the  
 104 computation that occurs behind the scenes in Venture. The second view is declarative and is faithful  
 105 to notations like “ $g \sim P(g)$ ” which are often used in mathematical treatments. Because a model  
 106 program could make arbitrarily many calls to `g`, and the joint distribution on the return values of the  
 107

108 calls could have arbitrarily high entropy, it is not computationally possible in finite time to choose  
 109 the entire function  $g$  all at once as in the second view. Thus, it stands to reason that any computa-  
 110 tionally implementable notion of “nonparametric random functions” must involve incremental random  
 111 choices in one way or another, and Gaussian processes in Venture are no exception.  
 112

## 113 2.1 Memoization

114  
 115 Memoization is the practice of storing previously computed values of a function so that future calls  
 116 with the same inputs can be evaluated by lookup rather than recomputation. Although memoiza-  
 117 tion does not change the semantics of a deterministic program, it does change that of a stochastic  
 118 program (Goodman et al., 2008). The authors provide an intuitive example: let  $f$  be a function  
 119 that flips a coin and return “head” or “tails”. The probability that two calls of  $f$  are equivalent is  
 120 0.5. However, if the function call is memoized, it is 1. In fact, there is an infinite range of possible  
 121 caching policies (specifications of when to use a stored value and when to recompute), each poten-  
 122 tially having a different semantics. Any particular caching policy can be understood by random  
 123 world semantics (Poole, 1993; Sato, 1995) over the stochastic program: each possible world corre-  
 124 sponds to a mapping from function input sequence to function output sequence (McAllester et al.,  
 125 2008). In Venture, these possible worlds are first-class objects, and correspond to the *probabilistic*  
 126 *execution traces* (Mansingka et al., 2014).

127 To transfer this idea to probabilistic programming, we now introduce a language construct called a  
 128 *statistical memoizer*. Suppose we have a function  $f$  which can be evaluated but we wish to learn  
 129 about the behavior of  $f$  using as few evaluations as possible. The statistical memoizer, which here  
 130 we give the name `gpmem`, was motivated by this purpose. It produces two outputs:  
 131

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

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

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

136 Different implementations of the statistical memoizer can have different prior distributions  $P(f)$ ; in  
 137 this paper, we deploy a Gaussian process prior (implemented as `gpmem` below). Note that we require  
 138 the ability to sample  $f_{\text{emu}}$  jointly at multiple inputs because the values of  $f(x_1), \dots, f(x_k)$  will in  
 139 general be dependent. To illustrate the linguistic power of `gpmem` consider a simple model program  
 140 that uses `gpmem` procedure abstraction, and a loop:  
 141

```

1 define choose_next_point = proc(em) { gridsearch_argmax( em ) }
2
3 // Here stats( em ) is the memo table {(x_i, y_i)}.
4 // Take the (x,y) pair with the largest y.
5 define extract_answer = proc(em) { first(max( stats( em, second ))) }
6
7 // Hyper-parameter
8 assume theta = tag( quote( params ), 0, gamma(1,1))
9 assume (f_probe f_emu) = gpmem(f, make_squaredexp(1.0, theta))
10
11 for t...T:
12   f_probe( choose_next_point( f_emu ))
13   extract_answer( f_emu )
14
  
```

155 An equivalent model in typical statistics notation, written in a way that attempts to be as linguistically  
 156 faithful as possible, is

$$\begin{aligned}
 158 \quad & f^{(0)} \sim P(f) \\
 159 \quad & x^{(t)} = \arg \max_x f^{(t)}(x) \\
 160 \quad & f^{(t+1)} \sim P \left( f^{(t)} \mid f^{(t)}(x^{(t)}) = (f x^{(t)}) \right).
 \end{aligned}$$

162 The linguistic constructs for abstraction are not present in statistics notation. The equation  $x^{(t)} =$   
 163  $\arg \max_x f^{(t)}(x)$  has to be inlined in statistics notation: while something like  
 164

$$x^{(t)} \sim F(P^{(t)})$$

165 (where  $P^{(t)}$  is a probability measure, and  $F$  here plays the role of `choose_next_point`) is mathematically coherent, it is not what statisticians would ordinarily write. This lack of abstraction then  
 166 makes modularity, or the easy digestion of large but finely decomposable models, more difficult in  
 167 statistics notation.  
 168

169 Adding a single line to the program, such as  
 170

```
171     infer mh( quote( params ), one, 50 )
```

172 after line 12 to infer the parameters of the emulator, would require a significant refactoring and  
 173 elaboration of the statistics notation. One basic reason is that probabilistic programs are written pro-  
 174 cedurally, whereas statistical notation is declarative; reasoning declaratively about the dynamics of a  
 175 fundamentally procedural inference algorithm is often unwieldy due to the absence of programming  
 176 constructs such as loops and mutable state.  
 177

178 We implement `gpmem` by memoizing a target procedure in a wrapper that remembers previously  
 179 computed values. This comes with interesting implications: from the standpoint of computation, a  
 180 data set of the form  $\{(x_i, y_i)\}$  can be thought of as a function  $y = f_{\text{restr}}(x)$ , where  $f_{\text{restr}}$  is restricted  
 181 to only allow evaluation at a specific set of inputs  $x$  (Alg. 1). Modelling the data set with a GP then  
 182

---

#### Algorithm 1 Restricted Function

---

```
1: function  $f_{\text{RESTR}}(x)$ 
2:   if  $x \in \mathcal{D}$  then
3:     return  $\mathcal{D}[x]$ 
4:   else
5:     raise Exception('Illegal input')
6:   end if
7: end function
```

---

196  
 197 amounts to trying to learn a smooth function  $f_{\text{emu}}$  ("emu" stands for "emulator") which extends  $f$   
 198 to its full domain. Indeed, if  $f_{\text{restr}}$  is a foreign procedure made available as a black-box to Venture,  
 199 whose secret underlying pseudo code is in Alg. 1, then the `observe` code can be rewritten using  
 200 `gpmem` as in Listing 1 (where here the data set  $D$  has keys  $x[1], \dots, x[n]$ ):  
 201

Listing 1: Observation with `gpmem`

```
202
203
204 1  assume (f_compute f_emu) = gpmem( f_restr )
205 2  for i=1 to n:
206 3    predict f_compute( x[i] )
207 4    infer mh(quote(hyper-parameters), one, 100)
208 5    sample (f_emu( array( 1, 2, 3 ))
```

210  
 211 This rewriting has at least two benefits: (i) readability (in some cases), and (ii) amenability to active  
 212 learning. As to (i), the statistical code of creating a Gaussian process is replaced with a memoization-  
 213 like idiom, which will be more familiar to programmers. As to (ii), when using `gpmem`, it is quite  
 214 easy to decide incrementally which data point to sample next: for example, the loop from  $x[1]$  to  
 215  $x[n]$  could be replaced by a loop in which the next index  $i$  is chosen by a supplied decision rule. In  
 this way, we could use `gpmem` to perform online learning using only a subset of the available data.

```

216
217
218 assume (f_compute f_emu) = gpmem( f )
219 sample (f_emu( array( -20, ..., 20))
220
221  ~ N(mu, K)
222
223
224 predict f_compute( 12.6)
225
226 sample (f_emu( array( -20, ..., 20))
227
228
229
230 predict f_compute( -6.4)
231
232 sample (f_emu( array( -20, ..., 20))
233
234
235
236
237
```

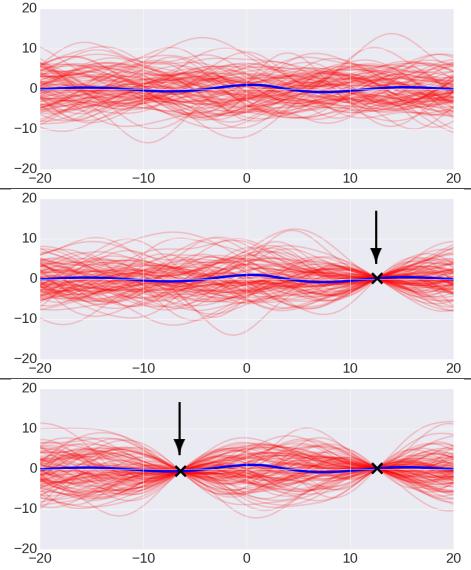


Figure 1: gpmem tutorial

### 3 Example Applications

gpmem is an elegant linguistic framework for function learning-related tasks. The technique allows language constructs from programming to help express models which would be cumbersome to express in statistics notation. We will now illustrate this with three example applications.

#### 3.1 Hyperparameter estimation with structured hyper-prior

The probability of the hyper-parameters of a GP as defined above and given covariance function structure  $\mathbf{K}$  is:

$$P(\boldsymbol{\theta} | \mathbf{D}, \mathbf{K}) = \frac{P(\mathbf{D} | \boldsymbol{\theta}, \mathbf{K})P(\boldsymbol{\theta} | \mathbf{K})}{P(\mathbf{D} | \mathbf{K})}. \quad (2)$$

Let the  $\mathbf{K}$  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 processes<sup>1</sup>. The work suggests a hierarchical system of hyper-parameterization. Here, we draw hyper-parameters from a  $\Gamma$  distributions:

$$\ell^{(t)} \sim \Gamma(\alpha_1, \beta_1), \sigma^{(t)} \sim \Gamma(\alpha_2, \beta_2) \quad (3)$$

and in turn sample the  $\alpha$  and  $\beta$  from  $\Gamma$  distributions as well:

$$\alpha_1^{(t)} \sim \Gamma(\alpha_\alpha^1, \beta_\alpha^1), \alpha_2^{(t)} \sim \Gamma(\alpha_\alpha^2, \beta_\alpha^2), \dots \quad (4)$$

One can represent this kind of model using gpmem (Listing 2). Neal provides a custom inference algorithm setting and evaluates it using the following synthetic data problem. Let  $f$  be the underlying function that generates the data:

$$f(x) = 0.3 + 0.4x + 0.5 \sin(2.7x) + \frac{1.1}{(1+x^2)} + \eta \quad \text{with } \eta \sim \mathcal{N}(0, \sigma) \quad (5)$$

We synthetically generate outliers by setting  $\sigma = 0.1$  in 95% of the cases and to  $\sigma = 1$  in the remaining cases. gpmem can capture the true underlying function within only 100 MH steps on the hyper-parameters to get a good approximation for their posterior (see Fig. 2). Note that Neal devises an additional noise model and performs a large number of Hybrid-Monte Carlo and Gibbs steps.

<sup>1</sup>In (Neal, 1997) the sum of an SE plus a constant kernel is used. We keep the WN kernel for illustrative purposes.

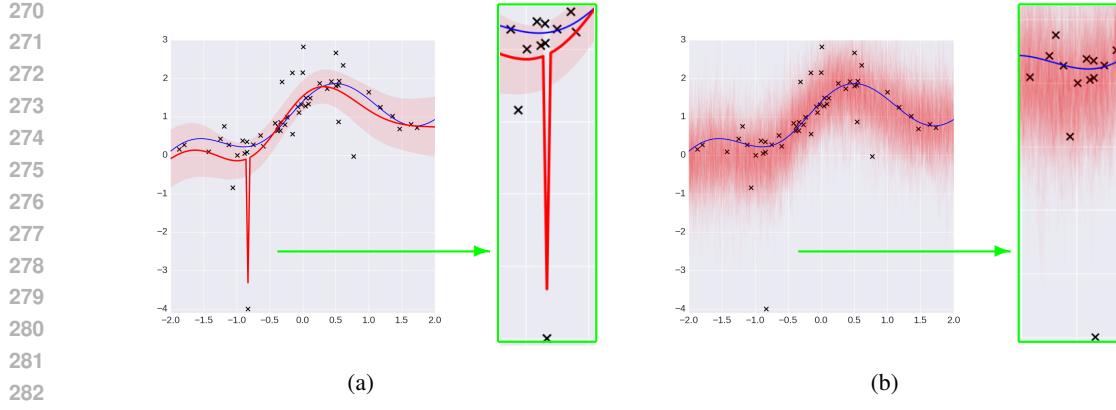


Figure 2: (a) shows deterministic inference, that is optimization for an example of Neal's synthetic function with an extreme outlier (green).

We illustrate the hyper-parameters by showing a comparison between  $\sigma$  (Fig. 2). We see that gpmem learns the posterior distribution well, the posterior even exhibits a bimodal histogram when sampling  $\sigma$  100 times reflecting the two modes of data generation, that is normal noise and outliers.

Listing 2: Hyper-Prior Learning

```

299
300 // SETTING UP THE MODEL
301 assume alpha_sf = tag(quote(hyperhyper), 0, gamma(7, 1))
302 assume beta_sf = tag(quote(hyperhyper), 1, gamma(7, 1))
303 assume alpha_l = tag(quote(hyperhyper), 2, gamma(7, 1))
304 assume beta_l = tag(quote(hyperhyper), 3, gamma(7, 1))

305 // Parameters of the covariance function
306 assume sf = tag(quote(hyper), 0, gamma(alpha_sf, beta_sf)))
307 assume l = tag(quote(hyper), 1, gamma(alpha_l, beta_l)))
308 assume sigma = tag(quote(hyper), 2, uniform_continuous(0, 2))

309 // The covariance function
310 assume se = make_squaredexp(sf, l)
311 assume wn = make_whitenoise(sigma)
312 assume composite_covariance = add_funcs(se, wn)

313 /// PERFORMING INFERENCE
314 // Create a prober and emulator using gpmem
315 assume f_restr = get_neal_blackbox()
316 assume (f_compute, f_emu) = gpmem(f_restr, composite_covariance)

317 // Probe all data points
318 for n ... N
319     predict f_compute(get_neal_data_xs(n))

320 // Infer hypers and hyperhypers
321 infer repeat(100, do(
322     mh(quote(hyperhyper), one, 2),
323     mh(quote(hyper), one, 1)))

```

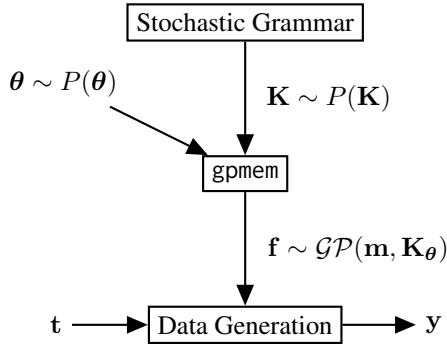


Figure 3: Graphical description of Bayesian GP structure learning.

### 3.2 Discovering symbolic models for time series

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)<sup>2</sup>.

With gpmem we can provide a fully Bayesian treatment of this, previously unavailable, using a stochastic grammar (see Fig. 3).

We deploy a probabilistic context free grammar for our prior on structures. 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:

$$\mathbb{E}[y^* | x^*, \mathbf{D}, \mathbf{K}] = \int \int f(x^*, \theta, \mathbf{K}) P(\theta | \mathbf{D}, \mathbf{K}) P(\mathbf{K} | \Omega, s, n) d\theta d\mathbf{K}. \quad (6)$$

This is done by sampling from the posterior probability distribution of the hyper-parameters and the possible kernel:

$$y^* \approx \frac{1}{T} \sum_{t=1}^T f(x^* | \theta^{(t)}, \mathbf{K}^{(t)}). \quad (7)$$

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:

$$\mathbf{K}^{(t)} \sim P(\mathbf{K} | \Omega, s, n) \quad (8)$$

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  $\Omega$  combining base kernels. The marginal probability of a composite structure

$$P(\mathbf{K} | \Omega, s, n) = P(\Omega | s, n) \times P(s | n) \times P(n), \quad (9)$$

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

$$P(s | n) = \frac{n!}{|s|!}, \quad (10)$$

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

$$P(\Omega | s, n) = \binom{n}{r} p_{+ \times}^k (1 - p_{+ \times})^{n-k}. \quad (11)$$

---

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

378 Many equivalent covariance structures can be sampled due to covariance function algebra and equiv-  
 379 alent representations with different parameterization (Lloyd et al., 2014). To inspect the posterior  
 380 of these equivalent structures we convert each kernel expression into a sum of products and sub-  
 381 sequently simplify. All base kernels can be found in Appendix A, rules for this simplification can be  
 382 found in appendix B. The code for learning of kernel structure is as follows:  
 383

```

384 // GRAMMAR FOR KERNEL STRUCTURE
385 assume kernels = list(se, wn, lin, per, rq) // defined as above
386
387 // prior on the number of kernels
388 2 assume p_number_k = uniform_structure(n)
389 3 assume sub_s = tag(quote(grammar), 0,
390 4                           subset(kernels, p_number_k))
391
392 5 assume grammar = proc(l) {
393 6   // kernel composition
394 7   if (size(l) <= 1)
395 8     { first(l) }
396 9   else { if (bernoulli())
397 10      { add_funcs(first(l), grammar(rest(l))) }
398 11      else { mult_funcs(first(l), grammar(rest(l))) }
399 12    }
400
401 13 assume K = tag(quote(grammar), 1, grammar(sub_s))
402
403 14 assume (f_compute f_emu) = gpmem(f_restr, K)
404
405 // Probe all data points
406 15 for n ... N
407 16   predict f_compute(get_data_xs(n))
408
409 // PERFORMING INFERENCE
410 17 infer repeat(2000, do(
411 18   mh(quote(grammar), one, 1),
412 19   for kernel ∈ K
413 20     mh(quote(hyperkernel), one, 1)))
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
  
```

412 We defined the space of covariance structures in a way that allows us to produce results coherent with  
 413 work presented in Automatic Statistician. For example, for the airline data set describing monthly  
 414 totals of international airline passengers (Box et al., 1997, according to Duvenaud et al., 2013). Our  
 415 most frequent sample is identical with the highest scoring result reported in previous work using a  
 416 search-and-score method (Duvenaud et al., 2013) for the CO<sub>2</sub> data set (see Rasmussen and Williams,  
 417 2006 for a description) and the predictive capability is comparable. However, the components factor  
 418 in a different way due to different parameterization of the individual base kernels. We see that the  
 419 most probable alternatives for a structural description both recover the data dynamics (Fig. 8 for the  
 420 airline data set).

421 Confident about our results, we can now query the data for certain structures being present. We  
 422 illustrate this using the Mauna Loa data used in previous work on automated kernel discovery (Du-  
 423 venaud et al., 2013). We assume a relatively simple hypothesis space consisting of only four kernels,  
 424 a linear, a smoothing, a periodic and a white noise kernel. In this experiment, we resort to the white  
 425 noise kernel instead RQ (similar to (Lloyd et al., 2014)). We can now run the algorithm, compute  
 426 a posterior of structures. We can also query this posterior distribution for the marginal of certain  
 427 simple structures to occur. We demonstrate this in Fig. 4

### 3.3 Bayesian optimization via Thompson sampling

428 We introduce Thompson sampling, the basic solution strategy underlying the Bayesian optimization  
 429 with gpmem. Thompson sampling Thompson (1933) is a widely used Bayesian framework for ad-  
 430 dressing the trade-off between exploration and exploitation in multi-armed (or continuum-armed)

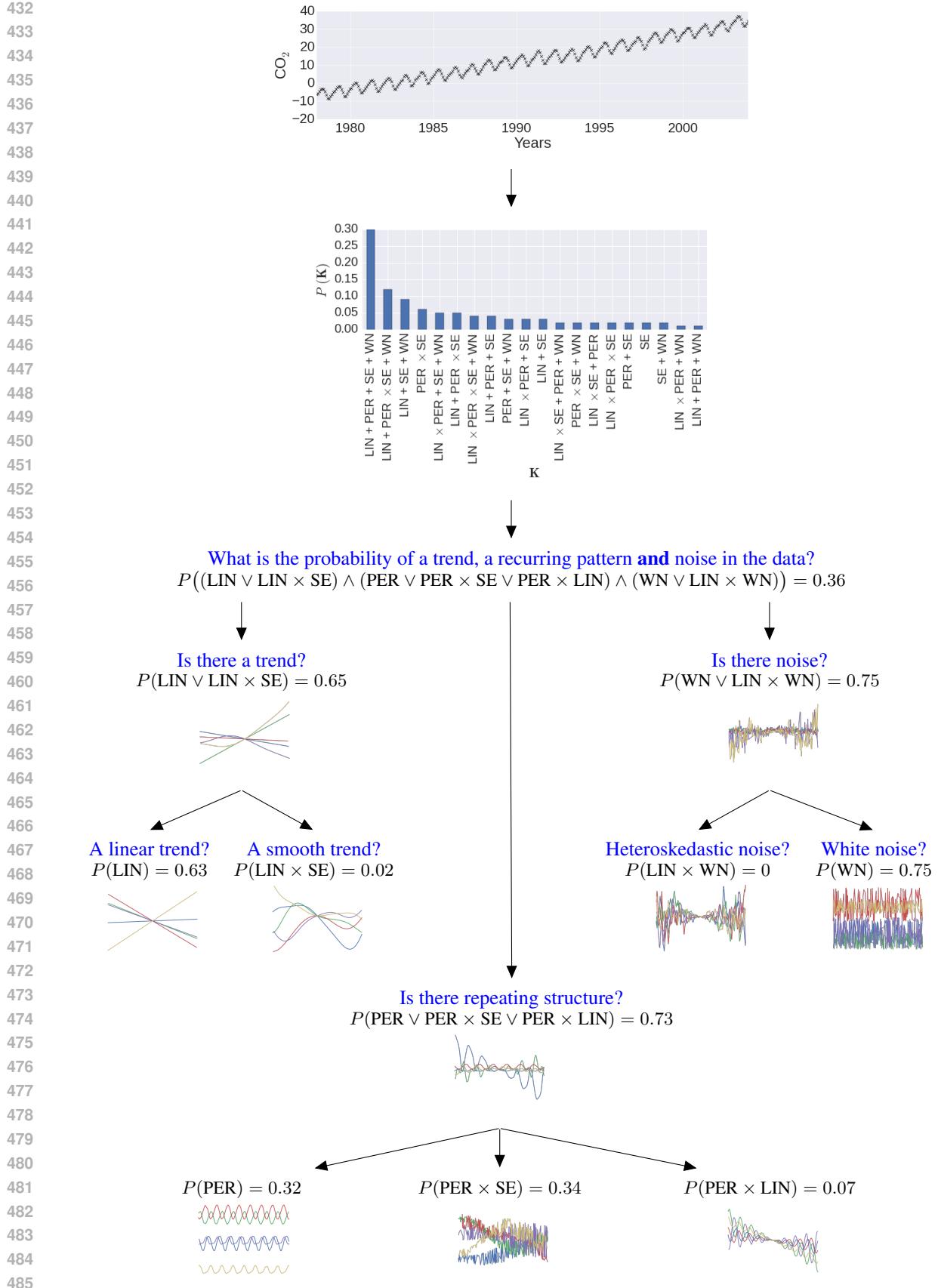


Figure 4: We can query the data if some logical statements are probable to be true, for example, is it true that there is a trend?

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  $\vartheta$  takes the form  $(\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$  where  $\mathbf{a}_{\text{past}}$  is the vector of past actions,  $\mathbf{r}_{\text{past}}$  is the vector of their rewards, and  $\theta$  (the "semicontext") contains any other information that is included in the context.

In this setup, Thompson sampling has the following steps:

---

**Algorithm 2** 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  $r_{\text{true}}$  be the reward received for action  $a$ . Update the believed distribution on  $\theta$ , 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  $\theta$  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  $\vartheta$ . 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 5. Roughly speaking, exploration will happen until the context  $\vartheta$  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.

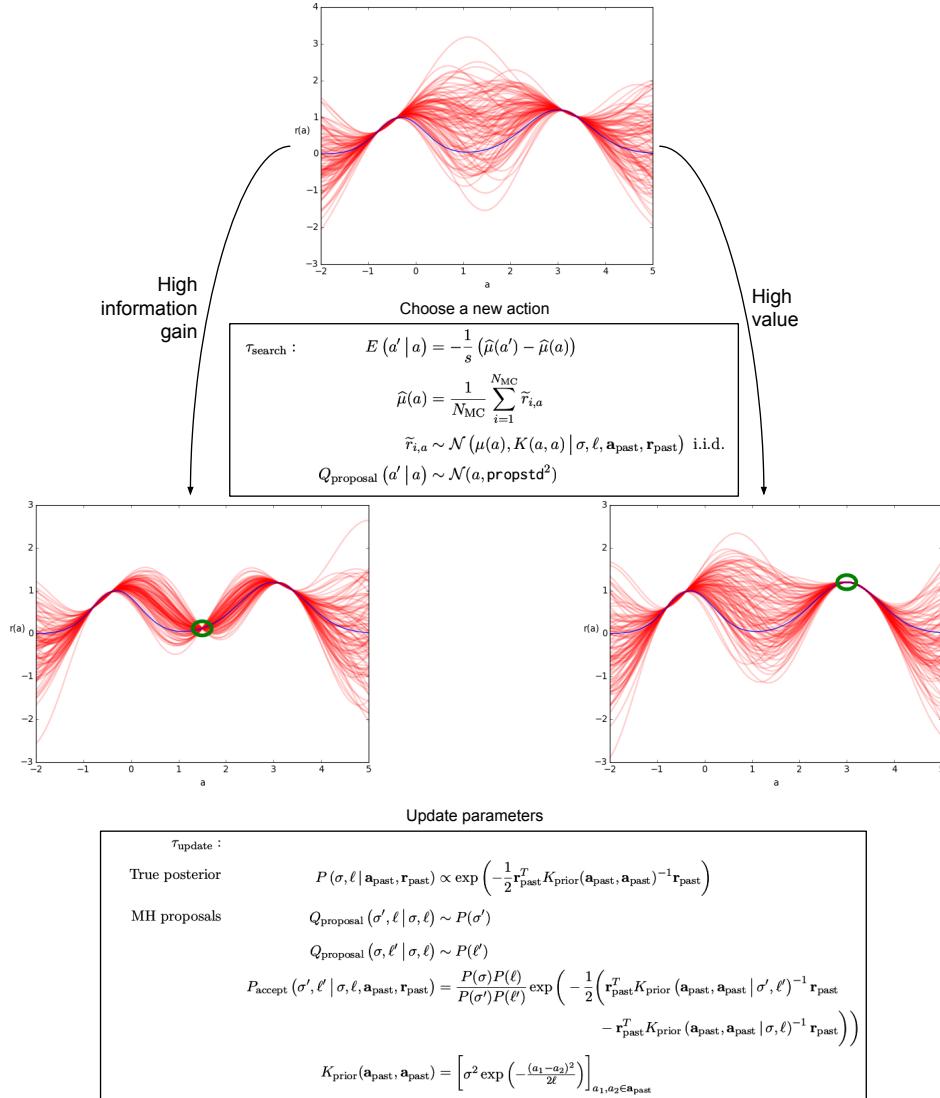
Typically, when Thompson sampling is implemented, the search over contexts  $\vartheta \in \Theta$  is limited by the choice of representation. In traditional programming environments,  $\theta$  often consists of a few numerical parameters for a family of distributions of a fixed functional form. With work, a mixture of a few functional forms is possible; but without probabilistic programming machinery, implementing a rich context space  $\Theta$  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  $\Theta$  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.

540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562



583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593

Figure 5: 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  $\tau_{\text{search}}$  and  $\tau_{\text{update}}$  are described in Section 3.3.

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

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

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

The posterior mean and covariance for such a context  $\vartheta$  are gotten by the usual conditioning formulas (assuming, for ease of exposition as above, that the prior mean is zero):<sup>3</sup>

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

Note that the context space  $\Theta$  is not a finite-dimensional parametric family, since the vectors  $\mathbf{a}_{\text{past}}$  and  $\mathbf{r}_{\text{past}}$  grow as more samples are taken.  $\Theta$  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 2 by running a Metropolis–Hastings (MH) sampler whose stationary distribution is the posterior  $P(\theta | \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$ . The functional forms of  $\mu_{\text{prior}}$  and  $K_{\text{prior}}$  are fixed in our case, so inference is only done over the parameters  $\eta = \{\sigma, \ell\}$ ; hence we equivalently write  $P(\sigma, \ell | \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$  for the stationary distribution. We make MH proposals to one variable at a time, using the prior as proposal distribution:

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

and

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

The MH acceptance probability for such a proposal is

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

Because the priors on  $\sigma$  and  $\ell$  are uniform in our case, the term involving  $Q_{\text{proposal}}$  equals 1 and we have simply

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

The proposal and acceptance/rejection process described above define a transition operator  $\tau_{\text{update}}$  which is iterated a specified number of times; the resulting state of the MH Markov chain is taken as the sampled semicontext  $\theta$  in Step 1 of Algorithm 2.

For Step 2 (Search) of Thompson sampling, we explore the action space using an MH-like transition operator  $\tau_{\text{search}}$ . As in MH, each iteration of  $\tau_{\text{search}}$  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  $a$ .

<sup>3</sup>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  $[K(a_i, a'_j)]_{1 \leq i \leq n, 1 \leq j \leq n'}$ .

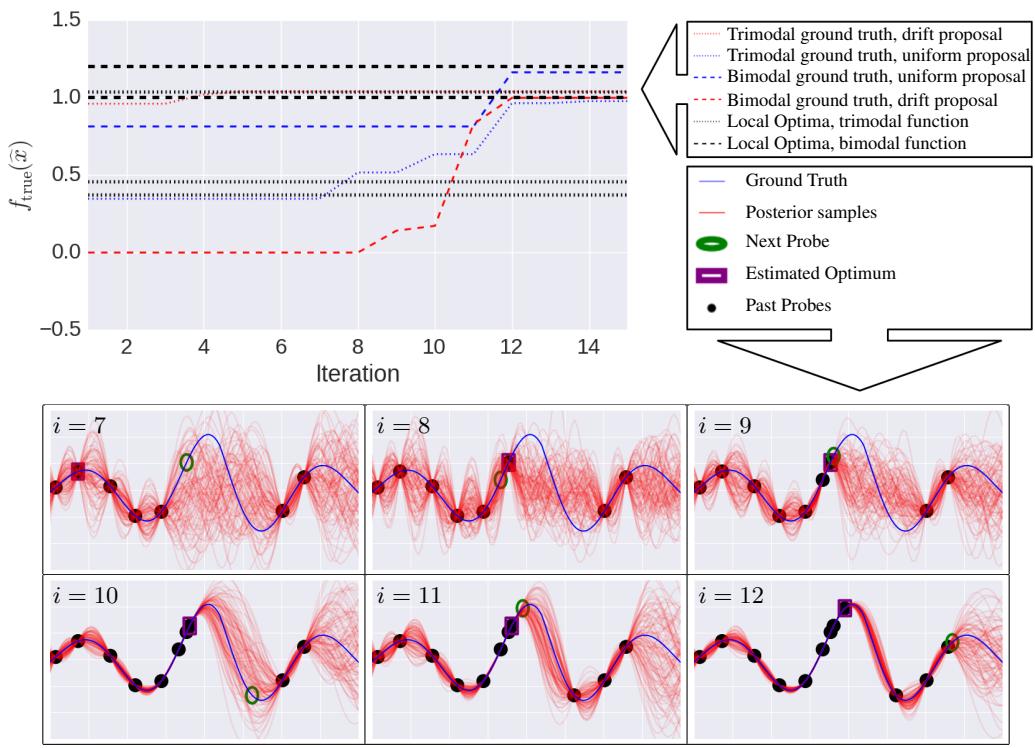


Figure 6: Top: the estimated optimum over time. Blue and Red represent optimization with uniform and Gaussian drift proposals. Black lines indicate the local optima of the true functions. Bottom: a sequence of actions. Depicted are iterations 7-12 with uniform proposals.

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  $N_{\text{avg}}$  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  $N_{\text{avg}}$  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 5).<sup>4,5</sup> We see a complete probabilistic program with gpmem implementing Bayesian optimization with Thompson Sampling below (Listing 3).

Listing 3: 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)

714 // A naive estimate of the argmax of the given function
715 define mc_argmax = proc(func) {
716     candidate_xs = mapv(proc(i) {uniform_continuous(-20, 20)},
717                          arange(20));
718     candidate_ys = mapv(func, candidate_xs);
719     lookup(candidate_xs, argmax_of_array(candidate_ys))
720 };

721 // Shortcut to sample the emulator at a single point without packing
722 // and unpacking arrays
723 define emulate_pointwise = proc(x) {
724     run(sample(lookup(f_emulate(array(unquote(x))), 0)))
725 };

726 // Main inference loop
727 infer repeat(15, do(pass,
728     // Probe V at the point mc_argmax(emulate_pointwise)
729     predict(f_compute(unquote(mc_argmax(emulate_pointwise))),
730     // Infer hyperparameters
731     mh(quote(hyper), one, 50)));
732
733
734 4 Discussion
735
736 We provided gpmem, an elegant linguistic framework for function learning-related tasks such as
737 Bayesian optimization and GP kernel structure learning. We highlighted how gpmem overcomes
738 shortcomings of the notations currently used in statistics, and how language constructs from pro-
739 gramming allow the expression of models which would be cumbersome (prohibitively so, in some
740 cases) to express in statistics notation. We evaluated our contribution on a range of hard problems
741 for state-of-the-art Bayesian nonparametrics.
742
743
744
745
746
747
748
749
750
751
752
753

```

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

<sup>5</sup>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  $\mu$ .

756      **Appendix**

758      **A Covariance Functions**

760  
761       $SE = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right)$       (12)  
762

763       $LIN = \sigma^2(xx')$       (13)  
764

765       $C = \sigma^2$       (14)  
766

767       $WN = \sigma^2 \delta_{x,x'}$       (15)  
768

769       $RQ = \sigma^2 \left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha}$       (16)  
770

771       $PER = \sigma^2 \exp\left(\frac{2 \sin^2(\pi(x - x')/p)}{\ell^2}\right).$       (17)

772      **B Covariance Simplification**  
773

774      

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

775      Rule 1 is derived as follows:  
776

777      
$$\begin{aligned} \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) &= \sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \sigma_b^2 \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2} - \frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \end{aligned} \quad (18)$$

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

780      
$$\sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \times \sigma_b \delta_{x,x'} = \sigma_a \sigma_b \delta_{x,x'} \quad (19)$$

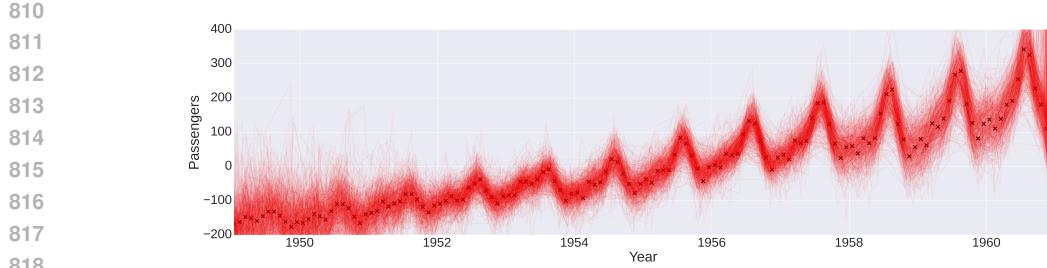
781      Rule 3 is derived as follows:  
782

783      
$$\theta_c(x \times x') = \theta_a(x \times x') + \theta_b(x \times x') \quad (20)$$

784      Multiplying any kernel with a constant obviously changes only the scale parameter of a kernel (Rule  
785      4).

800      **C Additional Structure Learning Results**  
801

802  
803  
804  
805  
806  
807  
808  
809



819 Figure 7: Data (black x) and posterior samples (red) for the airline data set.  
820



826 Figure 8: We see two possible hypotheses for the composite structure of  $\mathbf{K}$ . (a) Most frequent sample  
827 drawn from the posterior on structure. We have found two global components. First, a smooth trend  
828 (LIN  $\times$  SE) with a non-linear increasing slope. Second, a periodic component with increasing  
829 variation and noise. (b) Second most frequent sample drawn from the posterior on structure. We  
830 found one global component. It is comprised of local changes that are periodic and with changing  
831 variation.

## References

- 832  
833  
834
- 835 Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. (1997). *Time series analysis: forecasting and*  
836 *control*.
- 837 Clifton, L., Clifton, D. A., Pimentel, M. A. F., Watkinson, P. J., and Tarassenko, L. (2013). Gaussian  
838 processes for personalized e-health monitoring with wearable sensors. *Biomedical Engineering,*  
839 *IEEE Transactions on*, 60(1):193–197.
- 840 Duvenaud, D., Lloyd, J. R., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2013). Structure  
841 discovery in nonparametric regression through compositional kernel search. In *Proceedings of*  
842 *the International Conference on Machine Learning (ICML)*, pages 1166–1174.
- 843 Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*,  
844 521(7553):452–459.
- 845 Goodman, N. D .and Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. (2008). Church:  
846 A language for generative models. In *Proceedings of the Conference on Uncertainty in Artificial*  
847 *Intelligence (UAI)*, pages 220–229.
- 848 Kemmler, M., Rodner, E., Wacker, E., and Denzler, J. (2013). One-class classification with gaussian  
849 processes. *Pattern Recognition*, 46(12):3507–3518.
- 850 Lloyd, J. R., Duvenaud, D., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2014). Automatic con-  
851 struction and natural-language description of nonparametric regression models. In *Proceedings*  
852 *of the Conference on Artificial Intelligence (AAAI)*.
- 853 Mansinghka, V. K., Selsam, D., and Perov, Y. (2014). Venture: a higher-order probabilistic pro-  
854 gramming platform with programmable inference. *arXiv preprint arXiv:1404.0099*.
- 855 McAllester, D., Milch, B., and Goodman, N. D. (2008). Random-world semantics and syntactic  
856 independence for expressive languages. Technical report.
- 857 Neal, R. M. (1997). Monte carlo implementation of gaussian process models for bayesian regression  
858 and classification. *arXiv preprint physics/9701026*.
- 859 Poole, D. (1993). Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*,  
860 64(1):81–129.
- 861 Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning (Adap-*  
862 *tive Computation and Machine Learning)*. The MIT Press.

864 Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *In  
865 Proceedings of the International Conference on Logic Programming*. Citeseer.  
866

867 Shepherd, T. and Owenius, R. (2012). Gaussian process models of dynamic pet for functional  
868 volume definition in radiation oncology. *Medical Imaging, IEEE Transactions on*, 31(8):1542–  
869 1556.

870 Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine  
871 learning algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2951–  
872 2959.

873 Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view  
874 of the evidence of two samples. *Biometrika*, pages 285–294.  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917