

---

# 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 con-  
 095 straints 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* (Mansinghka 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
```

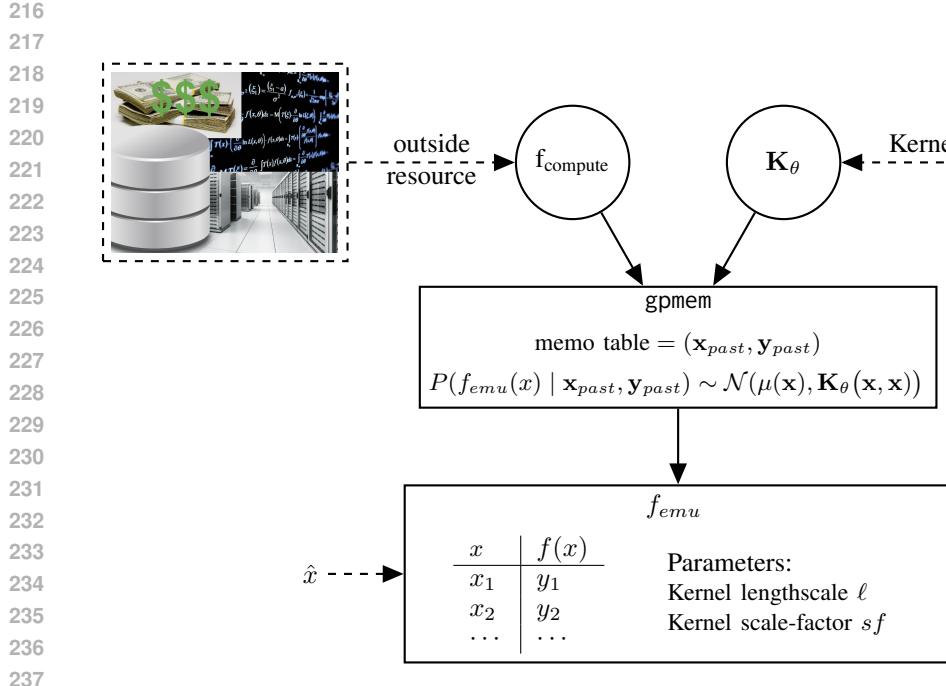
---

179 amounts to trying to learn a smooth function  $f_{\text{emu}}$  ("emu" stands for "emulator") which extends  $f$   
 180 to its full domain. Indeed, if  $f_{\text{restr}}$  is a foreign procedure made available as a black-box to Venture,  
 181 whose secret underlying pseudo code is in Alg. 1, then the `observe` code can be rewritten using  
 182 `gpmem` as in Listing 1 (where here the data set  $D$  has keys  $x[1], \dots, x[n]$ ):  
 183

Listing 1: Observation with `gpmem`

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

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.



```

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

predict f_compute( 12.6)
sample f_emu( array( -20, ..., 20))

predict f_compute( -6.4)
sample f_emu( array( -20, ..., 20))

observe f_emu( -3.1) = 2.60
observe f_emu( 7.8) = -7.60
observe f_emu( 0.0) = 10.19

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

infer mh(quote(hyper-parameter), one, 50)

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

```

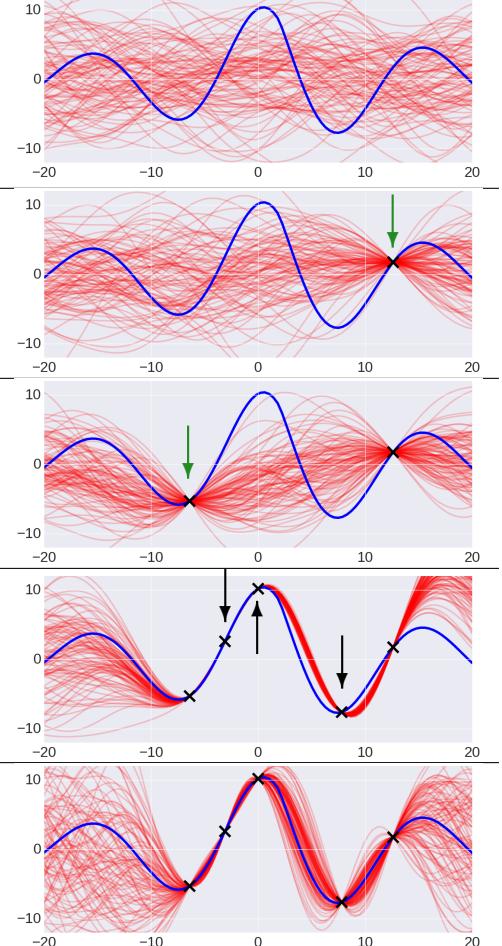
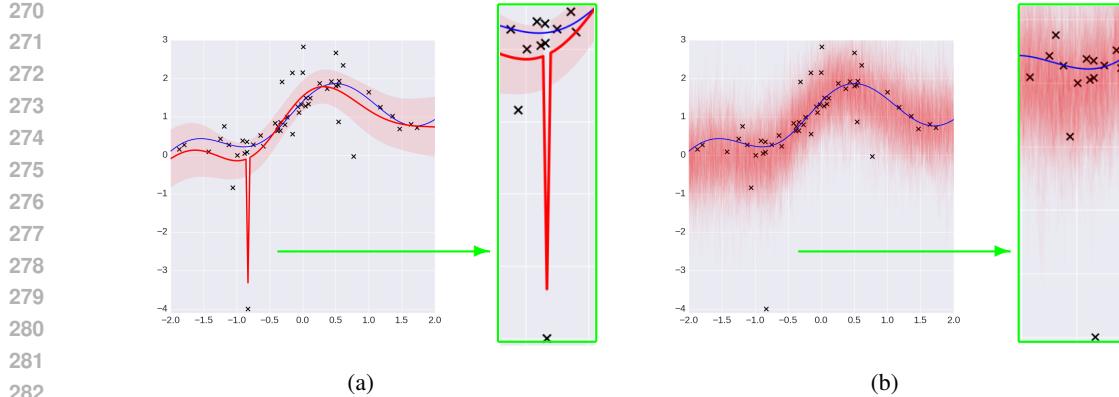


Figure 1: gpmem tutorial. The top shows a schematic of gpmem.  $f_{com} = f_{compute}$  probes an outside resource. This can be expensive (top left). Every probe is memoized and improves the GP-based emulator. Below the schematic we see a movie of the evolution of gpmem's state of believe of the world given certain Venture directives.



270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323

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

### 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 ??). 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.

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.

<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.

```

324
325
326
327
328
329
330
331
332
333 // Data and look-up function
334 define data = array(array(-1.87,0.13),..., array(1.67,0.81))
335 assume f_look_up = proc(index) {lookup( data, index)}
336
337
338 assume sf = tag(quote(hyper), 0, gamma(alpha_sf, beta_sf)))
339 assume l = tag(quote(hyper), 1, gamma(alpha_l, beta_l)))
340 assume sigma = tag(quote(hyper), 2, uniform_continuous(0, 2))
341
342 // The covariance function
343 assume se = make_squaredexp(sf, l)
344 assume wn = make_whitenoise(sigma)
345 assume composite_covariance = add_funcs(se, wn)
346
347 // Create a prober and emulator using gpmem
348 assume (f_compute, f_emu)
349 = gpmem(f_look_up, composite_covariance)
350
351 sample f_emu( array( -2, ..., 2))
352
353 // Observe all data points
354 for n ... N
355 observe f_emu(first(lookup(data,n)))
356 = second(lookup(data,n))
357 // Or: probe all data points
358 for n ... N
359 predict f_compute(first(lookup(data,n)))
360
361 sample f_emu( array( -2, ..., 2))
362
363 // Metropolis-Hastings
364 infer repeat( 100, do(
365 mh( quote(hyperhyper), one, 2),
366 mh( quote(hyper), one, 1)))
367
368 sample f_emu( array( -2, ..., 2))
369
370
371 // Optimization
372
373 infer map( quote(hyper), all, 0.01, 15)
374
375 sample f_emu( array( -2, ..., 2))
376
377
```

Hyper-Parameters for the Kernel:

---



---



---



---



---



---



---

Figure 3: Hyper-parameterization

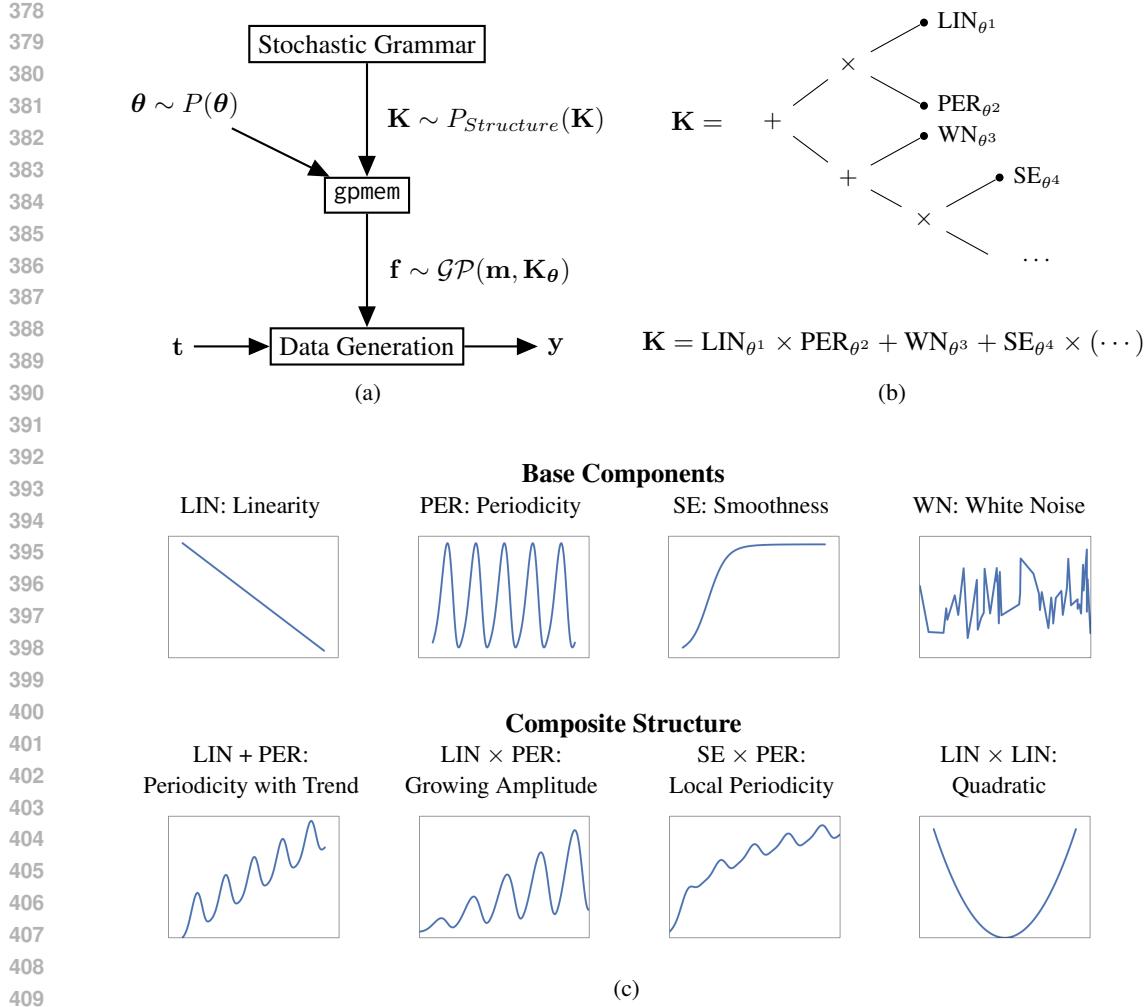


Figure 4: (a) Graphical description of Bayesian GP structure learning. (b) Composite structure. (c) The natural language interpretation of the structure.

### 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. 4).

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:

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

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

432 This is done by sampling from the posterior probability distribution of the hyper-parameters and the  
 433 possible kernel:  
 434

$$435 \quad y^* \approx \frac{1}{T} \sum_{t=1}^T f(x^* | \boldsymbol{\theta}^{(t)}, \mathbf{K}^{(t)}). \quad (7)$$

437 In order to provide the sampling of the kernel, we introduce a stochastic process that simulates the  
 438 grammar for algebraic expressions of covariance function algebra:  
 439

$$440 \quad \mathbf{K}^{(t)} \sim P(\mathbf{K} | \boldsymbol{\Omega}, s, n) \quad (8)$$

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

$$445 \quad P(\mathbf{K} | \boldsymbol{\Omega}, s, n) = P(\boldsymbol{\Omega} | s, n) \times P(s | n) \times P(n), \quad (9)$$

446 is characterized by the prior  $P(n)$  on the number of base kernels used, the probability of a uniformly  
 447 chosen subset of the set of  $n$  possible covariance functions  
 448

$$449 \quad P(s | n) = \frac{n!}{|s|!}, \quad (10)$$

451 and the probability of sampling a global or a local structure, which is given by a binomial distribu-  
 452 tion:  
 453

$$454 \quad P(\boldsymbol{\Omega} | s, n) = \binom{n}{r} p_{+ \times}^k (1 - p_{+ \times})^{n-k}. \quad (11)$$

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

461  
 462  
 463  
 464  
 465  
 466  
 467  
 468  
 469  
 470  
 471  
 472  
 473  
 474  
 475  
 476  
 477  
 478  
 479  
 480  
 481  
 482  
 483  
 484  
 485

```

486
487 // GRAMMAR FOR KERNEL STRUCTURE
488 1 assume kernels = list(se, wn, lin, per, rq) // defined as above
489
490 // prior on the number of kernels
491 2 assume p_number_k = uniform_structure(n)
492 3 assume sub_s = tag(quote(grammar), 0,
493                      subset(kernels, p_number_k))
494
495 5 assume grammar = proc(l) {
496     // kernel composition
497     if (size(l) <= 1)
498         { first(l) }
499     else { if (bernoulli())
500           { add_funcs(first(l), grammar(rest(l))) }
501           else { mult_funcs(first(l), grammar(rest(l))) }
502     }
503
504 13 assume K = tag(quote(grammar), 1, grammar(sub_s))
505
506 // Probe all data points
507 15 for n ... N
508     predict f_compute(get_data_xs(n))
509
510 // PERFORMING INFERENCE
511 17 infer repeat(2000, do(
512     mh(quote(grammar), one, 1),
513     for kernel ∈ K
514         mh(quote(hyperkernel), one, 1)))
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539

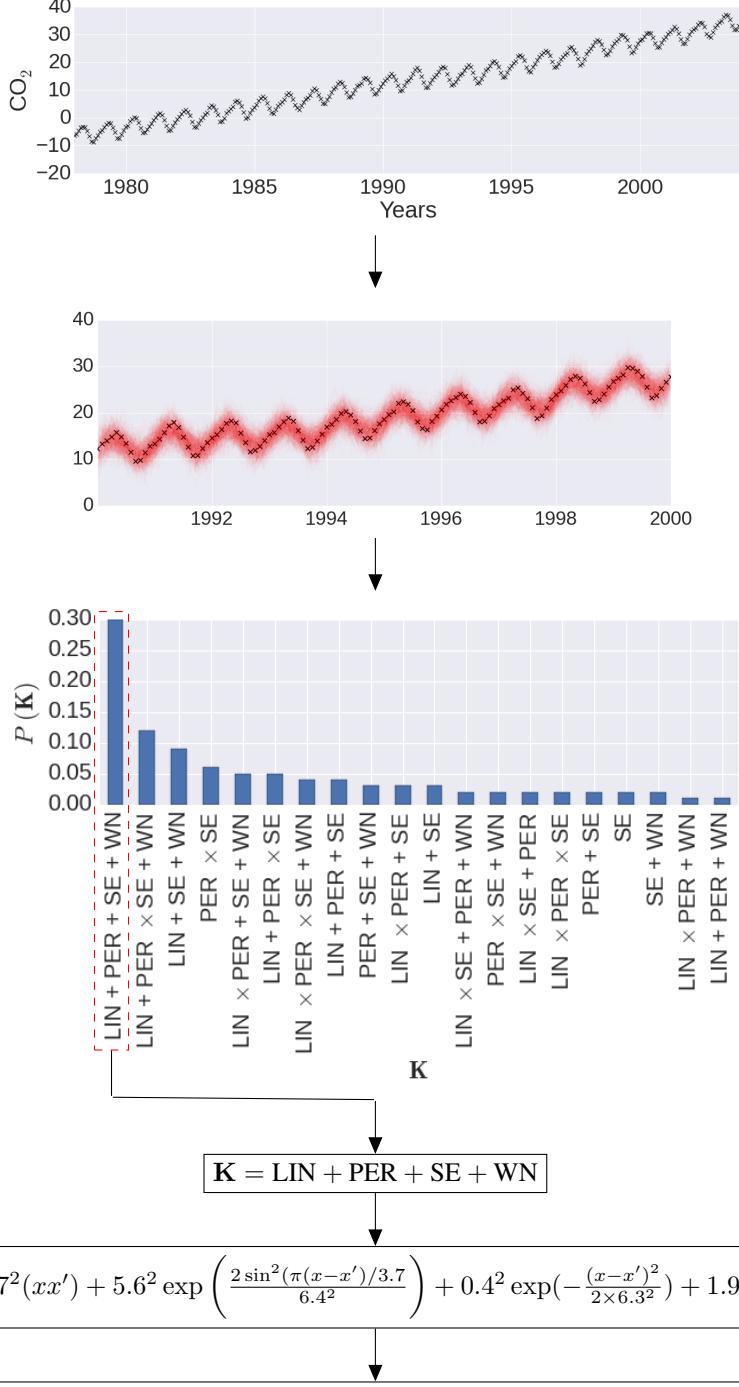
```

We defined the space of covariance structures in a way that allows us to produce results coherent with work presented in Automatic Statistician. For example, for the airline data set describing monthly totals of international airline passengers (Box et al., 1997, according to Duvenaud et al., 2013). Our most frequent sample is identical with the highest scoring result reported in previous work using a search-and-score method (Duvenaud et al., 2013) for the CO<sub>2</sub> data set (see Rasmussen and Williams, 2006 for a description) and the predictive capability is comparable. However, the components factor in a different way due to different parameterization of the individual base kernels. We see that the most probable alternatives for a structural description both recover the data dynamics (Fig. 12 for the airline data set).

Confident about our results, we can now query the data for certain structures being present. We illustrate this using the Mauna Loa data used in previous work on automated kernel discovery (Duvenaud et al., 2013). We assume a relatively simple hypothesis space consisting of only four kernels, a linear, a smoothing, a periodic and a white noise kernel. In this experiment, we resort to the white noise kernel instead RQ (similar to (Lloyd et al., 2014)). We can now run the algorithm, compute a posterior of structures (see Fig. 5). We can also query this posterior distribution for the marginal of certain simple structures to occur. We demonstrate this in Fig. 6

### 3.3 Bayesian optimization via Thompson sampling

In this section we present a probabilistic meta-program for Bayesian optimization, along with optimization routines implemented as concrete applications of the meta-program. Figure 7 shows a probabilistic meta-program for Bayesian optimization. We use the term “meta-program” because this program defines a generic rule for combining abstract components to perform optimization. The components are themselves subprograms, whose implementations may vary. A traditional example of generic programming is the tree search meta-program of Norvig (1992), written in LISP, which we present alongside the optimization meta-program for comparison.



**Qualitative Interpretation:** The the posterior peaks at a kernel structure with four 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 periodic function; (iii) a smooth function; and (iv) white noise.

Figure 5: Posterior of structure and qualitative, human interpretable reading. We take the raw data (top), compute a posterior distribution on structures (bar plot). We take the peak of this distribution ( $\text{LIN} + \text{PER} + \text{SE} + \text{WN}$ ) and show its human readable interpretation (left of bar plot). Below the bar plot show one sample of this structure with corresponding parameters. On the bottom, we sample from a GP with this kernel-hyper-parameter combination.

594  
595  
596  
597  
598  
599

600       $P(\mathbf{K}) \approx \frac{1}{N} \sum_{n=1}^N f(\mathbf{K}_n)$  where  $f(\mathbf{K}_n) = \begin{cases} 1, & \text{if } \mathbf{K}_{global} \in \mathbf{K}_n, \\ 0, & \text{otherwise.} \end{cases}$



601       $P(\mathbf{K}_a \wedge \mathbf{K}_b) \approx \frac{1}{N} \sum_{n=1}^N f(\mathbf{K}_n)$  where  $f(\mathbf{K}_n) = \begin{cases} 1, & \text{if } \mathbf{K}_a \text{ and } \mathbf{K}_b \in \mathbf{K}_n, \\ 0, & \text{otherwise.} \end{cases}$

602       $P(\mathbf{K}_a \vee \mathbf{K}_b) = P(\mathbf{K}_a) + P(\mathbf{K}_b) - P(\mathbf{K}_a \wedge \mathbf{K}_b)$

603  
604  
605  
606  
607  
608

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

610       $P((\text{LIN} \vee \text{LIN} \times \text{SE}) \wedge (\text{PER} \vee \text{PER} \times \text{SE} \vee \text{PER} \times \text{LIN}) \wedge (\text{WN} \vee \text{LIN} \times \text{WN})) = 0.36$

611  
612  
613

614      Is there a trend?  
615       $P(\text{LIN} \vee \text{LIN} \times \text{SE}) = 0.65$



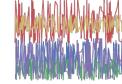
616      Is there noise?  
617       $P(\text{WN} \vee \text{LIN} \times \text{WN}) = 0.75$



618      A linear trend?  
619       $P(\text{LIN}) = 0.63$   
620      A smooth trend?  
621       $P(\text{LIN} \times \text{SE}) = 0.02$

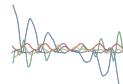


622      Heteroskedastic noise?  
623       $P(\text{LIN} \times \text{WN}) = 0$   
624      White noise?  
625       $P(\text{WN}) = 0.75$



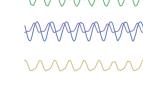
626  
627  
628

629      Is there repeating structure?  
630       $P(\text{PER} \vee \text{PER} \times \text{SE} \vee \text{PER} \times \text{LIN}) = 0.73$

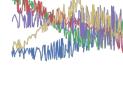


631  
632  
633

634      PER = 0.32



635      PER × SE = 0.34



636      PER × LIN = 0.07



637  
638  
639

640      Figure 6: We can query the data if some logical statements are probable to be true, for example, is it  
641      true that there is a trend?

642  
643  
644  
645  
646  
647

```

648
649
650
651
652
653
654
655
656
657
658
659
660
661
662 (defun tree-search (states goal-p successors combiner)
663   "Find a state that satisfies goal-p. Start with states,
664   and search according to successors and combiner."
665   (cond ((null states) fail)
666         ((funcall goal-p (first states)) (first states))
667         (t (tree-search
668             (funcall combiner
669                 (funcall successors (first states))
670                 (rest states))
671                 goal-p successors combiner))))
```

```

672
673
674 (define optimize
675   (lambda (probe do-search search-state-box
676             post-probe-inference extract-answer finished?)
677     (let loop ()
678       (if (finished?)
679           (extract-answer)
680           (begin
681             (do-search)
682             (if (contents-changed? search-state-box)
683                 (predict (probe ,(contents search-state-box))))
684                 (post-probe-inference)
685                 (loop)))))))
```

Figure 7: Top: Norvig's LISP meta-program for tree search. Bottom: A probabilistic meta-program for Bayesian optimization, written in VentureScheme, an abstract, Scheme-like syntax that can be used for Venture.

```

690
691
692
693
694
695
696
697
698
699
700
701
```

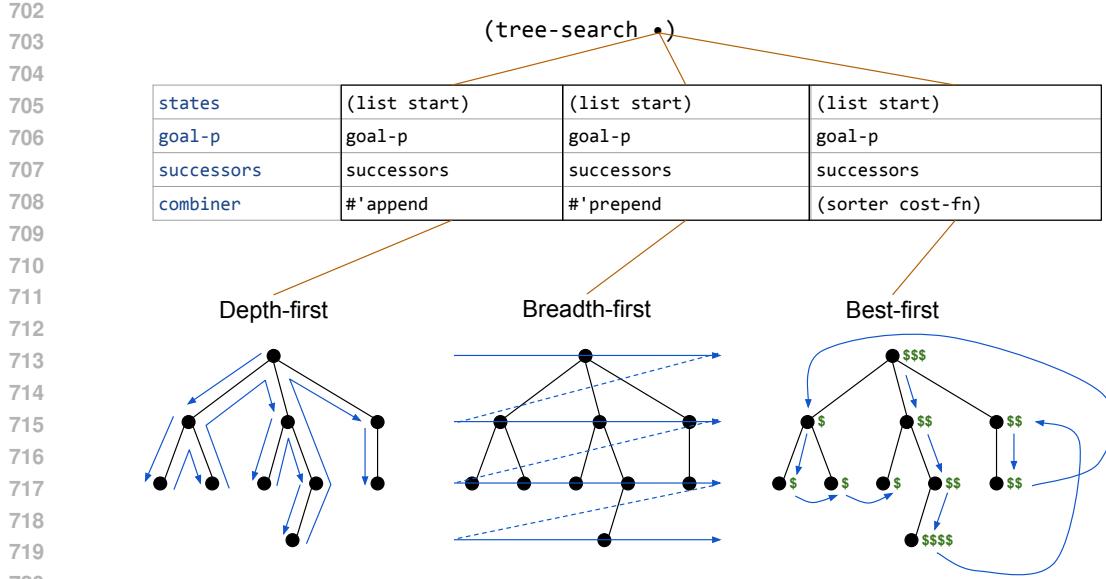


Figure 8: Invocations of Norvig’s tree search meta-procedure with different arguments to perform a variety of different tree searches.

Figure 8 shows the flexibility of Norvig’s meta-program. The procedure `tree-search` can be turned into a concrete implementation of depth-first, breadth-first or best-first search (among others) by simply supplying the appropriate combiner and set of initial states. Similarly, the meta-procedure `optimize` can make use of a variety of statistical emulators (e.g., GP-based emulators or polynomial regressions), a variety of acquisition heuristics (e.g., randomized grid search or Gaussian drift) and even a variety of rules for choosing an optimum based on the final state of the program (e.g., return the best probe so far, or return the analytically determined optimum of the emulator).

The specifications for the arguments to the meta-procedure `optimize` in Figures 7 are as follows:

- **probe:** A procedure for querying the true function  $f$  (and storing its value in the appropriate table, if caching is desired).
- **do\_search:** A procedure to search for a new probe point and, if one is found, store it in `search_state_box`.
- **search\_state\_box:** A container for the next value to be probed. In each iteration of the loop, if the contents of `search_state_box` have changed, a new probe is performed.
- **post\_probe\_inference:** Any inference instructions which should be run after each probe, such as inferring hyperparameters.
- **extract\_answer:** A procedure to produce an approximate optimum, based on the probes and inference that have been done so far.
- **finished?:** A procedure to decide whether optimization should be truncated here or should continue.

The meta-program is quite simple: Until `finished?` decides that optimization is finished, we choose new probe points and call `probe` on them. After each probe, we perform post-probe inference. Once optimization is finished, we call `extract_answer` to obtain an approximate optimum.

### Thompson Sampling

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,

756 and describe how Thompson sampling can be used to choose actions for that Markov Decision  
757 Processes (MDP).  
758

759 Here, an agent is to take a sequence of actions  $a_1, a_2, \dots$  from a (possibly infinite) set of possible  
760 actions  $\mathcal{A}$ . After each action, a reward  $r \in \mathbb{R}$  is received, according to an unknown conditional  
761 distribution  $P_{\text{true}}(r | a)$ . The agent's goal is to maximize the total reward received for all actions in  
762 an online manner. In Thompson sampling, the agent accomplishes this by placing a prior distribution  
763  $P(\vartheta)$  on the possible "contexts"  $\vartheta \in \Theta$ . Here a context is a believed model of the conditional  
764 distributions  $\{P(r | a)\}_{a \in \mathcal{A}}$ , or at least, a believed statistic of these conditional distributions which  
765 is sufficient for deciding an action  $a$ . If actions are chosen so as to maximize expected reward, then  
766 one such sufficient statistic is the believed conditional mean  $V(a | \vartheta) = \mathbb{E}[r | a; \vartheta]$ , which can be  
767 viewed as a believed value function. For consistency with what follows, we will assume our context  
768  $\vartheta$  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  
769 rewards, and  $\theta$  (the "semicontext") contains any other information that is included in the context.  
770

In this setup, Thompson sampling has the following steps:

---

771 **Algorithm 2** Thompson sampling.

---

772 Repeat as long as desired:

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

781  
782 Note that when  $\mathbb{E}[r | a; \vartheta]$  (under the sampled value of  $\theta$  for some points  $a$ ) is far from the true value  
783  $\mathbb{E}_{P_{\text{true}}}[r | a]$ , the chosen action  $a$  may be far from optimal, but the information gained by probing  
784 action  $a$  will improve the belief  $\vartheta$ . This amounts to "exploration." When  $\mathbb{E}[r | a; \vartheta]$  is close to the  
785 true value except at points  $a$  for which  $\mathbb{E}[r | a; \vartheta]$  is low, exploration will be less likely to occur, but  
786 the chosen actions  $a$  will tend to receive high rewards. This amounts to "exploitation." The trade-off  
787 between exploration and exploitation is illustrated in Figure 9. Roughly speaking, exploration will  
788 happen until the context  $\vartheta$  is reasonably sure that the unexplored actions are probably not optimal,  
789 at which time the Thompson sampler will exploit by choosing actions in regions it knows to have  
790 high value.

791 Typically, when Thompson sampling is implemented, the search over contexts  $\vartheta \in \Theta$  is limited  
792 by the choice of representation. In traditional programming environments,  $\theta$  often consists of a few  
793 numerical parameters for a family of distributions of a fixed functional form. With work, a mixture of  
794 a few functional forms is possible; but without probabilistic programming machinery, implementing  
795 a rich context space  $\Theta$  would be an unworkably large technical burden. In a probabilistic programming  
796 language, however, the representation of heterogeneously structured or infinite-dimensional context  
797 spaces is quite natural. Any computable model of the conditional distributions  $\{P(r | a)\}_{a \in \mathcal{A}}$  can  
798 be represented as a stochastic procedure  $(\lambda(a) \dots)$ . Thus, for computational Thompson sampling,  
799 the most general context space  $\hat{\Theta}$  is the space of program texts. Any other context space  $\Theta$  has a  
800 natural embedding as a subset of  $\hat{\Theta}$ .  
801

802 **A Mathematical Specification**

803 Our mathematical specification assert the following properties:  
804

- 805 • The regression function has a Gaussian process prior.  
806 • The actions  $a_1, a_2, \dots \in \mathcal{A}$  are chosen by a Metropolis-like search strategy with Gaussian  
807 drift proposals.  
808 • The hyperparameters of the Gaussian process are inferred using Metropolis–Hastings sam-  
809 pling after each action.

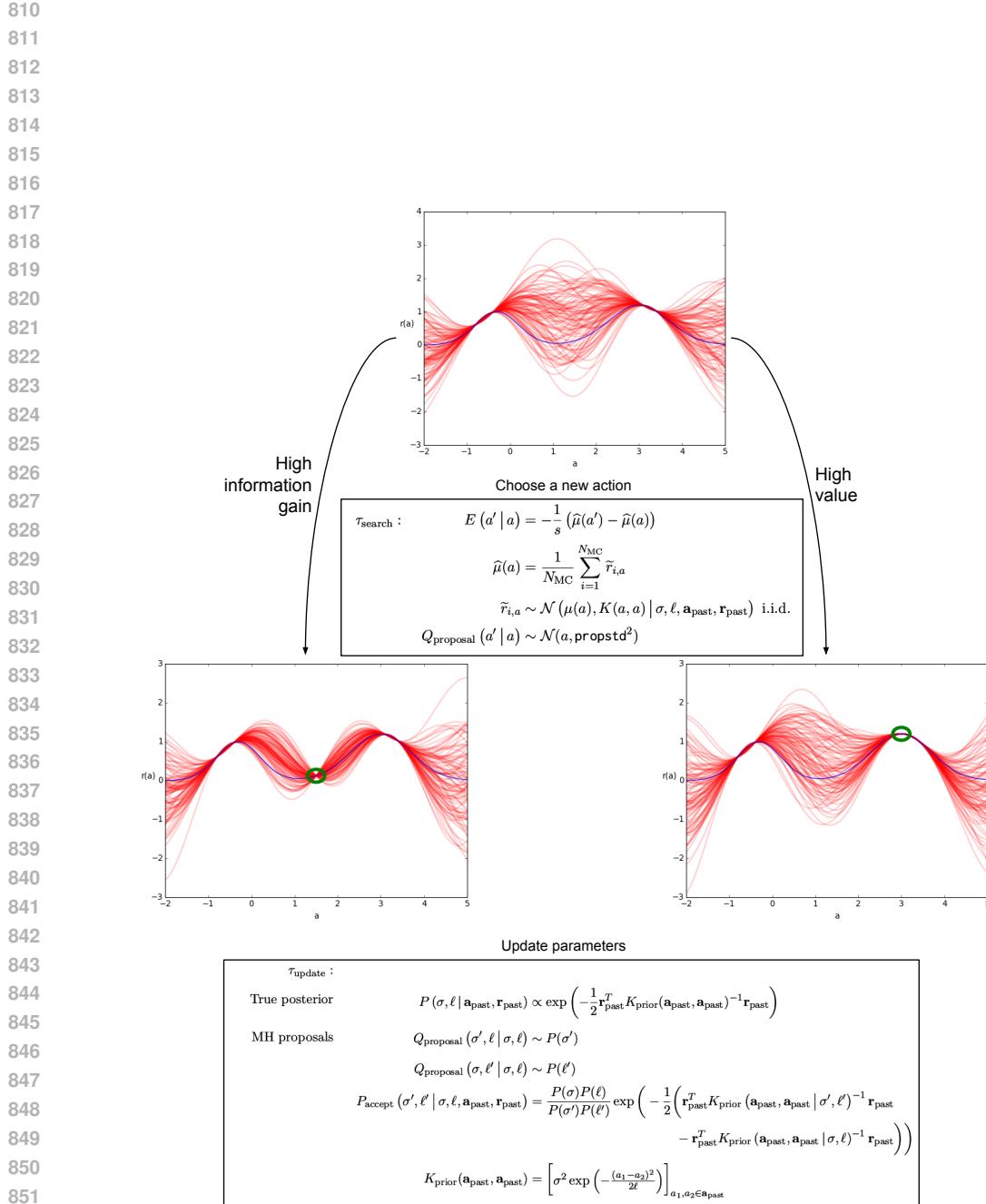


Figure 9: Two possible actions (in green) for an iteration of Thompson sampling. The believed distribution on the value function  $V$  is depicted in red. In this example, the true reward function is deterministic, and is drawn in blue. The action on the right receives a high reward, while the action 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  $[K(a_i, a'_j)]_{1 \leq i \leq n, 1 \leq j \leq n'}$ .

918 The Markov chain's initial state is the most recent action, and the proposal distribution is Gaussian  
 919 drift:

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

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

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

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

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

924 where

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

925 and

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

926 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  
 927  $N_{\text{avg}}$  are specified ahead of time. Proposals of estimated value higher than that of the current action  
 928 are always accepted, while proposals of estimated value lower than that of the current action are  
 929 accepted with a probability that decays exponentially with respect to the difference in value. The  
 930 rate of the decay is determined by the temperature parameter  $s$ , where high temperature corresponds  
 931 to generous acceptance probabilities. For  $s = 0$ , all proposals of lower value are rejected; for  
 932  $s = \infty$ , all proposals are accepted. For points  $a$  at which the posterior mean  $\mu(a)$  is low but the  
 933 posterior variance  $K(a, a)$  is high, it is possible (especially when  $N_{\text{avg}}$  is small) to draw a “wild”  
 934 value of  $\hat{\mu}(a)$ , resulting in a favorable acceptance probability.

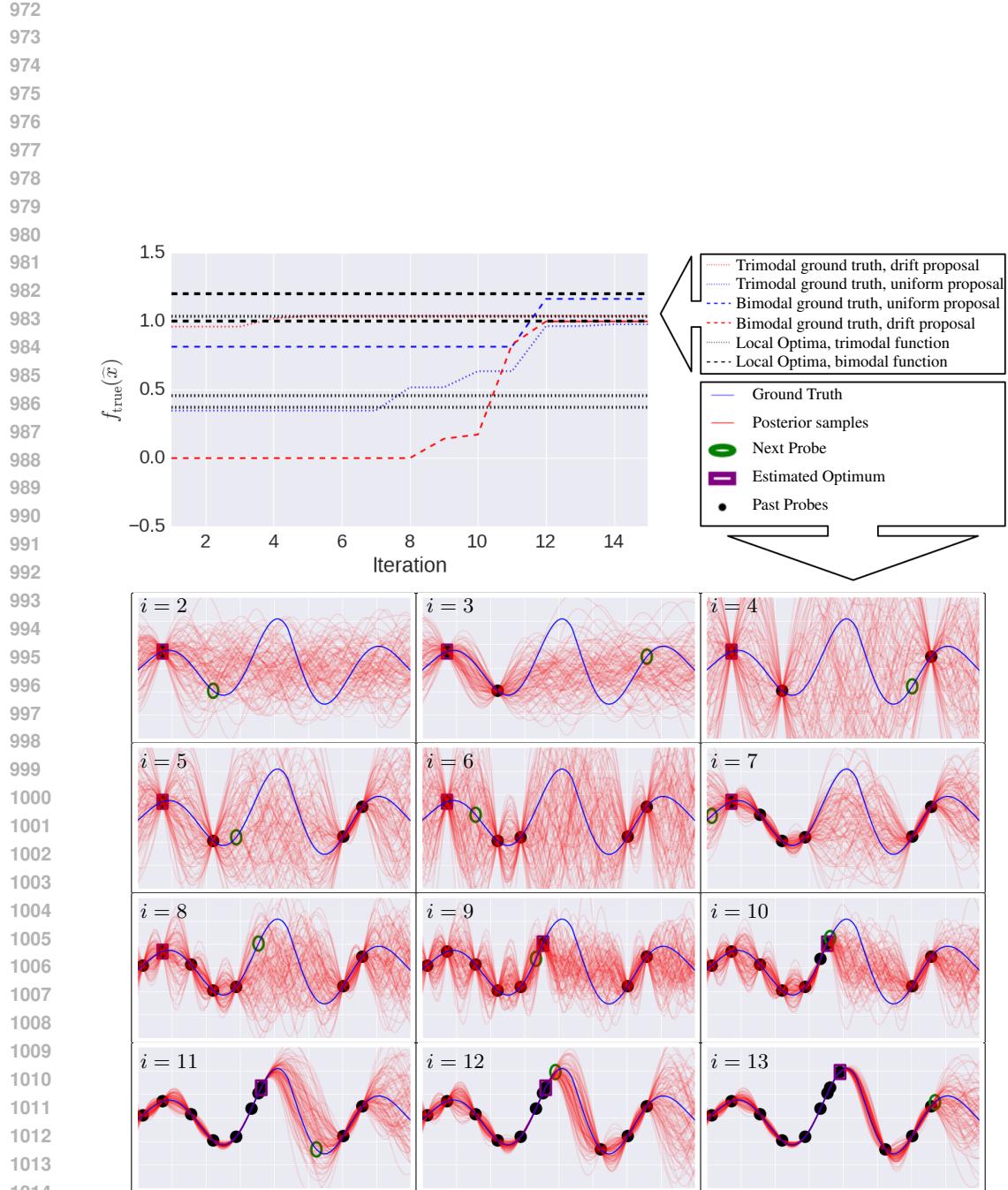
935 Indeed, taking an action  $a$  with low estimated value but high uncertainty serves the useful function  
 936 of improving the accuracy of the estimated value function at points near  $a$  (see Figure 9).<sup>4,5</sup> We see  
 937 a complete probabilistic program with `gpmem` implementing Bayesian optimization with Thompson  
 938 Sampling below (Listing 2).

939  
 940  
 941  
 942  
 943  
 944  
 945  
 946  
 947  
 948  
 949  
 950  
 951  
 952  
 953  
 954  
 955  
 956  
 957  
 958  
 959  
 960  
 961  
 962  
 963  
 964  
 965  
 966  
 967  
 968  
 969

---

<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  $\mu$ .



1015 Figure 10: Top: the estimated optimum over time. Blue and Red represent optimization with uniform  
1016 and Gaussian drift proposals. Black lines indicate the local optima of the true functions. Bottom: a  
1017 sequence of actions. Depicted are iterations 7-12 with uniform proposals.

1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025

```

1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079

    Listing 2: 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, 1)
4 assume blackbox_f = get_bayesopt_blackbox()
5 assume (f_compute, f_emulate) = gpmem(blackbox_f, se)

6 // A naive estimate of the argmax of the given function
7 define mc_argmax = proc(func) {
8     candidate_xs = mapv(proc(i) {uniform_continuous(-20, 20)},
9                          arange(20));
10    candidate_ys = mapv(func, candidate_xs);
11    lookup(candidate_xs, argmax_of_array(candidate_ys))
12};

13 // Shortcut to sample the emulator at a single point without packing
14 // and unpacking arrays
15 define emulate_pointwise = proc(x) {
16     run(sample(lookup(f_emulate(array(unquote(x))), 0)))
17};

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

```

## 4 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.

1080                   **Appendix**  
 1081

1082                   **A Covariance Functions**  
 1083

1084  
 1085                    $\text{SE} = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right)$                    (12)  
 1086

1087                    $\text{LIN} = \sigma^2(xx')$                    (13)  
 1088

1089                    $\text{C} = \sigma^2$                    (14)  
 1090

1091                    $\text{WN} = \sigma^2 \delta_{x,x'}$                    (15)  
 1092

1093                    $\text{RQ} = \sigma^2 \left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha}$                    (16)  
 1094

1095                    $\text{PER} = \sigma^2 \exp\left(\frac{2\sin^2(\pi(x - x')/p)}{\ell^2}\right).$                    (17)

1096                   **B Covariance Simplification**  
 1097

1098                    $\begin{array}{ll} \text{SE} \times \text{SE} & \rightarrow \text{SE} \\ \{ \text{SE}, \text{PER}, \text{C}, \text{WN} \} \times \text{WN} & \rightarrow \text{WN} \\ \text{LIN} + \text{LIN} & \rightarrow \text{LIN} \\ \{ \text{SE}, \text{PER}, \text{C}, \text{WN}, \text{LIN} \} \times \text{C} & \rightarrow \{ \text{SE}, \text{PER}, \text{C}, \text{WN}, \text{LIN} \} \end{array}$

1102                   Rule 1 is derived as follows:  
 1103

1104                   
$$\begin{aligned} \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) &= \sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \sigma_b^2 \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2}\right) \times \exp\left(-\frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_a^2} - \frac{(x - x')^2}{2\ell_b^2}\right) \\ &= \sigma_c^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \end{aligned} \quad (18)$$

1113                   For stationary kernels that only depend on the lag vector between  $x$  and  $x'$  it holds that multiplying  
 1114 such a kernel with a WN kernel we get another WN kernel (Rule 2). Take for example the SE kernel:  
 1115

1116                   
$$\sigma_a^2 \exp\left(-\frac{(x - x')^2}{2\ell_c^2}\right) \times \sigma_b \delta_{x,x'} = \sigma_a \sigma_b \delta_{x,x'} \quad (19)$$
  
 1117

1118                   Rule 3 is derived as follows:  
 1119

1120                   
$$\theta_c(x \times x') = \theta_a(x \times x') + \theta_b(x \times x')$$
                   (20)  
 1121

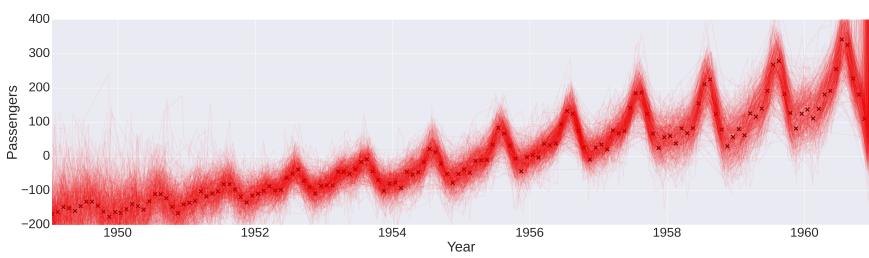
1122                   Multiplying any kernel with a constant obviously changes only the scale parameter of a kernel (Rule  
 1123 4).  
 1124

1125                   **C Additional Structure Learning Results**  
 1126

1127                   Below we depict additional results for the airline data set using a hypothesis space of LIN, PER, SE  
 1128 and RQ covariance functions as base kernels for the composition. Fig 11 shows the training data  
 1129 and posterior samples drawn from the GP with posterior composite structure.  
 1130

Fig. 12 shows why it advantageous to be Bayesian here. We see two possible hypotheses for the  
 1131 composite structure of  $\mathbf{K}$ .  
 1132

1133



1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143 Figure 11: Data (black x) and posterior samples (red) for the airline data set.  
1144



1150  
1151  
1152  
1153  
1154  
1155 Figure 12: We see two possible hypotheses for the composite structure of  $\mathbf{K}$ . (a) Most frequent sample drawn from the posterior on structure. We have found two global components. First, a smooth trend ( $\text{LIN} \times \text{SE}$ ) with a non-linear increasing slope. Second, a periodic component with increasing variation and noise. (b) Second most frequent sample drawn from the posterior on structure. We found one global component. It is comprised of local changes that are periodic and with changing variation.

## References

- 1156  
1157  
1158  
1159 Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. (1997). *Time series analysis: forecasting and control*.
- 1160  
1161 Clifton, L., Clifton, D. A., Pimentel, M. A. F., Watkinson, P. J., and Tarassenko, L. (2013). Gaussian processes for personalized e-health monitoring with wearable sensors. *Biomedical Engineering, IEEE Transactions on*, 60(1):193–197.
- 1162  
1163  
1164 Duvenaud, D., Lloyd, J. R., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2013). Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1166–1174.
- 1165  
1166 Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459.
- 1167 Goodman, N. D .and Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. (2008). Church: A language for generative models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 220–229.
- 1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960

- 1188 Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning (Adaptive  
1189 Computation and Machine Learning)*. The MIT Press.
- 1190
- 1191 Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *In  
Proceedings of the International Conference on Logic Programming*. Citeseer.
- 1192
- 1193 Shepherd, T. and Owenius, R. (2012). Gaussian process models of dynamic pet for functional  
1194 volume definition in radiation oncology. *Medical Imaging, IEEE Transactions on*, 31(8):1542–  
1195 1556.
- 1196 Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine  
1197 learning algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2951–  
1198 2959.
- 1199 Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view  
1200 of the evidence of two samples. *Biometrika*, pages 285–294.
- 1201
- 1202
- 1203
- 1204
- 1205
- 1206
- 1207
- 1208
- 1209
- 1210
- 1211
- 1212
- 1213
- 1214
- 1215
- 1216
- 1217
- 1218
- 1219
- 1220
- 1221
- 1222
- 1223
- 1224
- 1225
- 1226
- 1227
- 1228
- 1229
- 1230
- 1231
- 1232
- 1233
- 1234
- 1235
- 1236
- 1237
- 1238
- 1239
- 1240
- 1241