Bayesian Optimization as a Probabilistic Meta-Program

by Ben Zinberg

S.B. in Mathematics and Electrical Engineering & Computer Science, M.I.T., 2014

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachvsetts Institvte of Technology

September 2015

© Massachvsetts Institvte of Technology, 2015. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
September 7, 2015

Certified by ....
Vikash K. Mansinghka, Research Scientist, Thesis Supervisor
September 7, 2015

Accepted by ....
Dr. Christopher Terman, Chairman, Masters of Engineering Thesis Committee

Bayesian Optimization as a Probabilistic Meta-Program

by

Ben Zinberg

Submitted to the Department of Electrical Engineering and Computer Science
September 7, 2015
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Thesis Supervisor: Vikash K. Mansinghka, Research Scientist

Abstract

This thesis answers two questions: 1. How should probabilistic programming languages incorporate Gaussian processes? and 2. Is it possible to write a probabilistic meta-program for Bayesian optimization, a probabilistic meta-algorithm that can combine regression frameworks such as Gaussian processes with a broad class of parameter estimation and optimization techniques? We answer both questions affirmatively, presenting both an implementation and informal semantics for Gaussian process models in probabilistic programming systems, and a probabilistic meta-program for Bayesian optimization. The meta-program exposes modularity common to a wide range of Bayesian optimization methods in a way that is not apparent from their usual treatment in statistics.

Acknowledgments

This thesis is the visible output of my collaboration with a number of outstanding colleagues and mentors. I extend my sincerest thanks to my thesis supervisor Vikash Mansinghka, who has been a terrific mentor both inside and outside the academic realm. I also thank my colleagues in the Probabilistic Computing Group, who fixed countless confusions of mine, and also received (rightly!) countless new confusions as a result of collaborating with me. In other words, I would like to thank my colleagues for their patience and for sharing their perspectives through many interesting conversations. Additional thanks go to Ulli Schaechtle, who not only was my colleague in the Probabilistic Computing Group, but also was my roommate for a seemingly endless stay at the Embassy Suites Airport Hotel in Portland, and was a very supportive collaborator on the work that led directly to this thesis.

I would also like to thank Prof. Gerald Sussman for many stimulating conversations, reading recommendations, and—regarding the experience of adding a second major during senior year of undergrad—for being the "way" in the expression "where there's a will there's a way."

It is a relief that I do not have to fully articulate in this note how indebted I am to my parents and to my friends at MIT. That would be impossible. Suffice it to say that it was only through an incredible amount of consistent support, especially during the not-so-few periods of crisis, that I was able to reach this point.

Finally, I extend my thanks to DARPA's PPAML program for funding this research and my appointment as a research assistant.

Contents

1 Background **5**
- 1.1 Bayesian Optimization . . . 5
- 1.2 Background on Probabilistic Programming . . . 6
- 1.3 Traces and Interactivity . . . 10

2 Gaussian Processes As Statistical Memoizers **11**
- 2.1 Gaussian Processes In Probabilistic Programs . . . 11
- 2.2 Statistical Memoizers . . . 14

3 Bayesian Optimization Using Thompson Sampling **16**
- 3.1 Framework . . . 16
- 3.2 Thompson Sampling with a Statistical Memoizer . . . 18
- 3.3 A Mathematical Specification . . . 20
  - 3.3.1 Representations . . . 22
  - 3.3.2 Approximation Strategies . . . 23

4 Meta-Program **25**
- 4.1 The Meta-Program . . . 25
- 4.2 Concrete Version 1: Random Exploration of GP Emulator . . . 28
- 4.3 Demonstrations . . . 32
- 4.4 Concrete Version 2: Exact Optimization of Locally Quadratic Emulator . . . 34
- 4.5 Concrete Version 3: Stochastic Local Search with GP Emulator . . . 36
- 4.6 Computational Cost . . . 41

5 Conclusion **43**

A Additional Figures **44**

B Time Complexities of the Concrete Implementations **47**

C Attachments **48**

References **50**

1 Background

We begin by reviewing some relevant background on Bayesian optimization and probabilistic programming. We then provide further discussion of the concept of execution traces, which play a central role in our probabilistic programming language.

1.1 Bayesian Optimization

Bayesian optimization of an unknown function f consists of positing a prior distribution on f and treating all values depending on f, such as its values at specific points and its optimum, as random variables. In this Bayesian approach, there is in principle a well-defined correct (soft) answer to any query that could be asked about f, which depends only on the prior $P(f)$ and the observed data D on which the answers are conditioned: the posterior distribution $P(\bullet|D)$. It is up to the user (or domain expert) to specify a good prior $P(f)$, and to find an algorithm for efficiently computing an approximation to the posterior. Once these costs have been paid, however, the result has the desirable property that all available information is used at every stage of inference (as opposed to, e.g., hill-climbing methods which may be more prone to getting stuck at local optima due to their use of only local information). It has also been found that fully Bayesian methods can arrive at accurate answers with less acquired data than other methods [3, 11].

If f has domain $\mathcal{A}$ and target $\mathcal{B}$, then a prior on f can be viewed as a random field of $\mathcal{B}$-valued random variables $\{f(a)\}_{a\in\mathcal{A}}$ indexed by $\mathcal{A}$. In practice often $\mathcal{A} = \mathbb{R}^d$ (or, to facilitate optimization, one often takes $\mathcal{A}$ a bounded subset of $\mathbb{R}^d$) and $\mathcal{B} = \mathbb{R}$. A major class of examples comes from machine learning, in which the elements $a \in \mathcal{A}$ are settings of the parameters for a machine learning algorithm and $f(a)$ is a measure of the goodness of parameter values a in experiments [10]. Snoek et al. [10] demonstrate that fully Bayesian treatment of the unknown function f can lead to parameter settings which match or outperform those obtained by state-of-the-art parameter optimization methods with significantly fewer evaluation resources, on three notoriously hard machine learning problems:

- Online latent Dirichlet allocation, in which the number of topics, Dirichlet hyperparameters for each topic, and Dirichlet parameters for the overall topic distribution must be tuned

- Max-margin min-entropy (M3E) models [7] for binary classification of DNA sequences, which include the parameters of a latent-structured support vector machine as well as an entropy term
- A three-layer convolutional neural network for learning the CIFAR-10 benchmark data set for image classification.

In the case of the convolutional neural network, the Bayesian optimization methods of Snoek et al. outperformed the state of the art: 15% test error, improving upon the previous record of 18% test error which used parameter values chosen by experts.

The methods of Snoek et al. employ a Gaussian process (GP) prior on f.[1] This affords them a handful of efficiently computable acquisition heuristics: probability of improvement, expected improvement, and GP upper confidence bound [2, §2.3]. In principle, non-GP priors are possible, and below we initially approach Bayesian optimization from the perspective of an arbitrary prior, though our implementations and exposition focus primarily on GP priors. We do note, however, that using a different class of priors would likely require the invention of new acquisition heuristics that are efficiently computable under the new prior.

1.2 Background on Probabilistic Programming

A *probabilistic programming language* is a language for defining probabilistic generative models and inference on those models. For us, a *model program* means an executable description of a generative model. An *inference program* is an executable description of an inference algorithm, which may also make use of randomness. In this paper, our examples will use the Venture probabilistic programming language [6], but the concepts illustrated apply to probabilistic programming in general. Notably, in Venture, the sample space for all random variables occurring in a model is the set of all possible execution traces of the model program (see [6, §4] and Section 1.3). In fact, we take this as a definition: for a probabilistic program written in Venture, we define the *model program* to consist of all instructions (synonymously, *directives*) which enter the execution trace, and the

[1] For an introduction to Gaussian processes, see [9], especially §2.2.

inference program to consist of all directives, including untraced directives which manipulate the existing trace.

The range of models describable as probabilistic programs is very broad, containing graphical models (including plate notation) as a special case. There is a precise sense in which any computable generative model can be expressed as a probabilistic program [1]. The power of the modelling language comes not from executability of models—in other words, not from the ability to perform *forward-sampling*—but from the rich additional layers of description which allow automatic reasoning (such as automatic dependency tracking) and semi-automatic reasoning (such as exploratory inference programming) about models.

The range of inference algorithms describable as probabilistic programs is also very broad. General-purpose versions of rejection sampling, Gibbs sampling, Metropolis–Hastings, mean-field, and slice sampling are built into Venture as primitives [6]. Custom inference algorithms can be written as programs that manipulate traces: below we implement Thompson sampling as a custom inference program.

Listing 1.1 shows a simple probabilistic program containing only modelling instructions:

```
(assume rain (flip 0.3))

(assume sprinkler (if rain
                    false
                    (flip 0.8)))

(assume grass_wet (flip (if rain
                          (if sprinkler
                            0.99
                            0.95)
                          (if sprinkler
                            0.6
                            0.1))))

(assume humidity (if rain
                   (normal 95 2)
                   (normal 50 20)))
```

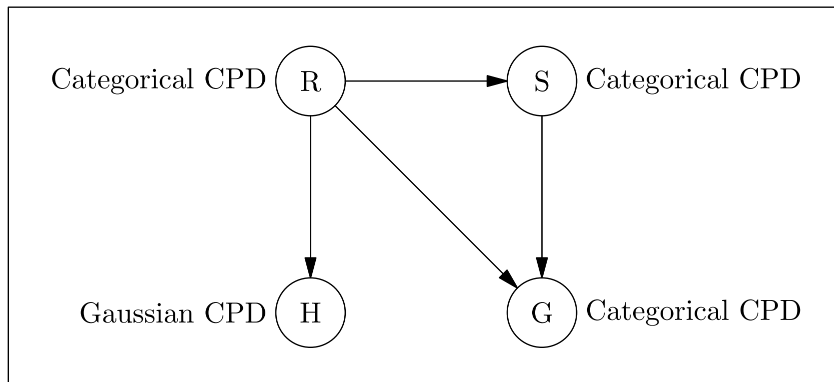
Listing 1.1. A simple probabilistic program (written in Venture) containing only modelling instructions.

This program reads as follows: It is raining (R) with probability 0.3. If it is raining, then the sprinkler (S) is never on; if it is not raining, then the sprinkler is on with probability 0.8. The probability that the grass is wet (G) depends on both whether it is raining and whether the sprinkler is on, according to the table

	R	¬R
S	0.99	0.6
¬S	0.95	0.1

Finally, the humidity is a continuous random variable whose parameters depend on whether it is raining: $\mathcal{N}(95, 2)$ (normal with mean 95 and variance 2) if it is raining, and $\mathcal{N}(50, 20)$ if it is not raining.

The program in Listing 1.1 induces the following directed graphical model:



Not all model programs can be described by a graphical model: due to branching and recursion, the set of random variables may vary from execution to execution, as may the dependency structure of even a fixed set of variables. In fact, the class of computable distributions that can be described by a graphical model is precisely the class of models that can be written as a probabilistic program with fixed control flow. Note that the program in Listing 1.1, while it does not have fixed control flow as written, has a finite number of possible control paths and thus can be rewritten with fixed control flow by table lookup into an enumeration of all possible paths. There are other cases in which the class of models expressible in an existing framework can be seen as those arising from a restricted class of probabilistic programs; one such case is plate notation in graphical models, which corresponds to probabilistic programs with fixed control flow which employ memoization via `mem`

(see [5, §2.1]) and treat the value of the memoized procedure at each input as a random variable. However, the full class of models expressible as probabilistic programs does not correspond to an existing notational framework. This is not surprising: we would expect that a computationally universal class of models requires a computational language to describe.[2]

Suppose the value of a random variable is observed—say, $\mathrm{H} = 85$—and we wish to infer values of other random variables. Rejection sampling of full executions of the model program, which in this context we call *global rejection sampling*, produces joint samples from the posterior distribution $P(\mathrm{R}, \mathrm{S}, \mathrm{G} \mid \mathrm{H} = 85)$; it is achieved in Venture by the directive

```
(infer (rejection default all 1))
```

Global rejection sampling is essentially the only inference program that can be employed on a black-box model program with no accompanying source code or inspection infrastructure. Our model, however, is not a black box; due to the richness of the description language, a wide range of inference programs are readily implementable. Rejection sampling can be performed on one random variable at a time, conditioned on the current values of the other variables, resulting in a Markov chain Monte Carlo (MCMC) inference routine whose state is a random tuple $(\mathrm{R}, \mathrm{S}, \mathrm{G})$ whose distribution, after sufficiently many transitions, is approximately the posterior. Note that inference on one variable at a time allows the programmer to choose the order in which variables are inferred, or to choose not to infer the values of variables on which the variable of interest does not depend. This need not be done manually: Venture dynamically tracks dependencies between random variables, which, for example, allows one to run Metropolis–Hastings inference on a single random variable X in which proposals only modify the values of those variables that depend on X:

```
(infer (mh default one 1))
```

(Here the keywords `default one` mean that X is chosen randomly from the set of random variables in the model, but it is also possible to specify an explicit choice of X by supplying different

[2] The language of probabilistic Turing machines is also "universal" in the sense that it could describe a program for forward-sampling at the machine level. But such a description lacks many of the useful characteristics of a probabilistic program: semantics for what the random variables are and what they depend on, and abstraction and modularity to allow for local changes to the model program.

arguments to `mh`.) While standard inference routines such as rejection sampling and Metropolis–Hastings are built into Venture, inference in general can be (and in Venture, is) fully programmable, allowing for composite inference strategies which use different inference subroutines on different parts of the model at different times, as well as dynamic choices of inference strategy which depend on state accrued during the execution of the model program and inference thereupon.

1.3 Traces and Interactivity

The output Y of a generative model can be thought of as the projection of a random variable J to just the non-latent parts. That is,

$$Y : \omega \in \Omega \overset{J}{\mapsto} (x, y) \overset{\text{proj}}{\longmapsto} y,$$

where Ω is the sample space; x is referred to as the "latent" data for this sample. If our generative model J is computable, and is embodied by the model program `J`, then as we mentioned in Section 1.2, without loss of generality we can take Ω to simply be the set of all possible executions of `J`, as is done in Venture.

Just as a generative model J can be extended by making additional random choices $J' \sim P(J'|J)$ conditioned on the value of J, an execution trace can be extended by executing more model directives. Consequently, the state of a model program can be inspected interactively, without making a commitment as to whether there is more model code to be executed later. In Venture, the current trace can be extended using the `predict` directive (or `assume`, which does the same thing and then assigns the resulting value to a named variable; see [12] and [6] for a detailed treatment of all the Venture directives). The hypothetical result of evaluating an expression `e` inside the model can be queried effectlessly using `sample`, which is equivalent to `predict` followed by `forget`: an evaluation of `e` is added to the current trace and then immediately deleted, and the value is returned. `predict` and `sample` are not interchangeable; that is to say, the meaning of a program (and the distribution of its possible traces) can depend on which evaluations are added to the trace and which are forgotten. We will see this below in Section 2.1.

2 Gaussian Processes As Statistical Memoizers

In this section, we first discuss the implementation of Gaussian processes in Venture. This serves two purposes: it brings to light certain properties that any implementation of "nonparametric random functions" in a probabilistic programming system must have, and it sets the stage for the probabilistic programs presented in Sections 3 and 4 which use Gaussian process models. Next, we introduce the statistical memoizer, a novel language construct for probabilistic programs which codifies the relationship between data acquisition and online regression. Statistical memoizers will be used to implement Thompson sampling in Section 4.

2.1 Gaussian Processes In Probabilistic Programs

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

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

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

```
(assume zero (make_const_func 0.0))
(assume se (make_squaredexp 1.0 1.0))
(assume g (make_gp zero se))
```

There are two ways two view `g` as a "random function." In the first view, the `assume` directive that instantiates `g` does not use any randomness—only the subsequent calls to `g` do—and coherence constraints are upheld by the interpreter by keeping track of which evaluations of `g` exist in the current trace. Namely, if the current trace contains evaluations of `g` at the points $x_1, \ldots, x_N$ with return values $y_1, \ldots, y_N$, then the next evaluation of `g` (say, jointly at the points $x_{N+1}, \ldots, x_{N+n}$)

will be distributed according to the joint conditional distribution

$$P\left((\mathtt{g}\ x_{N+1}),\ldots,(\mathtt{g}\ x_{N+n})\,\middle|\,(\mathtt{g}\ x_i)=y_i\text{ for }i=1,\ldots,N\right).$$

In the second view, `g` is a randomly chosen deterministic function, chosen from the space of all deterministic real-valued functions; in this view, the `assume` directive contains *all* the randomness, and subsequent invocations of `g` are deterministic. The first view is procedural and is faithful to the computation that occurs behind the scenes in Venture. The second view is declarative and is faithful to notations like "$g \sim P(g)$" which are often used in mathematical treatments. Because a model program could make arbitrarily many calls to `g`, and the joint distribution on the return values of the calls could have arbitrarily high entropy, it is not computationally possible in finite time to choose the entire function `g` all at once as in the second view. Thus, it stands to reason that any computationally implementable notion of "nonparametric random functions" must involve incremental random choices in one way or another, and Gaussian processes in Venture are no exception.

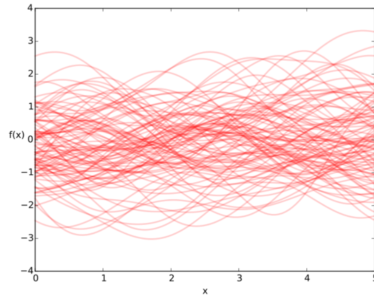
Behind the scenes, Venture attaches a state $D=(\mathbf{x}_{\text{past}},\mathbf{y}_{\text{past}})$ to each procedure created by `make_gp`, where $\mathbf{x}_{\text{past}}$ is a vector of all past inputs to `g` in the current trace, and $\mathbf{y}_{\text{past}}$ is the corresponding vector of outputs. (The state D is updated each time an invocation of `g` is added to or removed from the current trace.) Thus, the incremental random choices made by a GP `g` in Venture are simply samples from the conditional distribution $P(\mathtt{g}\,|\,D)$. That is, if `g` was initialized as `(make_gp mean cov)` and has accrued state $D=(\mathbf{x}_{\text{past}},\mathbf{y}_{\text{past}})$, then

$$(\mathtt{g}\ \mathbf{x})\sim P\left((\mathtt{g}_0\ \mathbf{x})\,|\,(\mathtt{g}_0\ \mathbf{x}_{\text{past}})=\mathbf{y}_{\text{past}}\right),\qquad\text{where }\mathtt{g}_0\sim\mathtt{(make_gp\ mean\ cov)}.$$

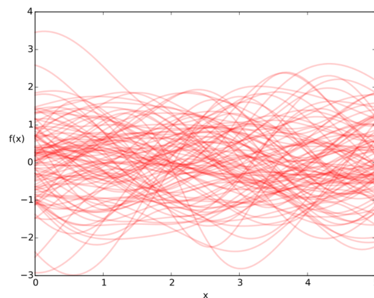
Statefully represented random functions highlight the difference between `predict` and `sample` (see Section 1.3). To illustrate, consider the following program:

```
(assume zero (make_const_func 0.0))
(assume se (make_squaredexp 1.0 1.0))
```

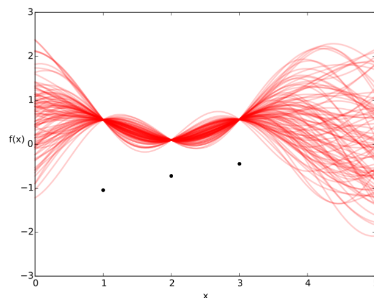
```
(assume g (make_gp zero se))

(call_back draw_gp_curves g ...)
```




```
(sample (g '(1 2 3)))
;; [-1.04331626 -0.72016875 -0.44525212]
(call_back draw_gp_curves g ...)
```




```
(predict (g '(1 2 3)))
;; [ 0.55736593  0.09784888  0.56553757]
(call_back draw_gp_curves g ...)
```



(The callback `draw_gp_curves` draws a cloud of curves sampled from `g`.) The `sample` directive does not affect the state of `g`, so the distribution of `g` after the `sample` is the same as the prior. The `predict`, however, does affect the state; the distribution of `g` after the `predict` directive is conditioned on the values returned by `g` inside the directive. This explains why, in the final plot, all curves go through the points returned by the previous `predict` directive, and do not go through the points returned by the `sample` directive (which are marked with black dots for reference).

2.2 Statistical Memoizers

We now introduce a language construct for probabilistic programming called a *statistical memoizer*. In addition to providing an elegant linguistic framework for function learning–related tasks such as Bayesian optimization, the statistical memoizer highlights 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.

Suppose we have a function `f` which can be evaluated, but as in Section 1.1, we wish to learn about the behavior of `f` using as few evaluations as possible. The statistical memoizer, which here we give the name `mem&em`, was motivated by this purpose. It produces two outputs:

$$\mathtt{f} \xrightarrow{\mathtt{mem\&em}} (\mathtt{f}_{\text{probe}}, \mathtt{f}_{\text{emu}}).$$

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

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

Non-Bayesian emulators are possible as well, but here we will focus on the Bayesian case. Different implementations of the statistical memoizer can have different prior distributions $P(f)$; our main example is a Gaussian process prior (implemented as `gpmem` below). Note that we require the ability to sample $\mathtt{f}_{\text{emu}}$ jointly at multiple inputs because the values of $f(\mathtt{x}_1), \ldots, f(\mathtt{x}_k)$ will in general be dependent.

To illustrate the linguistic power of `mem&em`, consider a simple model program that uses `mem&em`, procedure abstraction, and a loop:

```
1 (define (choose_next_point em)
2   (gridsearch_argmax em))
3 (define (extract_answer em)
```

```
4   ;; Here (stats em) is the memo table {(x_i, y_i)}.
5   ;; Take the (x,y) pair with the largest y.
6   (first (max (stats em) second)))
7 (assume theta (tag 'params ⟨parameters for the statistical emulator⟩))
8
9 (assume_list (f_probe f_emu) (mem&em f theta))
10 (let loop ((count 15))
11   (if (= count 0)
12     (extract_answer f_emu)
13     (begin
14       (f_probe (choose_next_point f_emu))
15       (loop (- count 1)))))
```

This program is written in a Venture-flavored pseudocode language that we call "pseudo-Venture." Most of the code we present, including Figure 4.1 and Listings 4.1, 4.2, and 4.3, will be in pseudo-Venture in order to simplify the presentation. The runnable Venture code is essentially the same, except for some rough edges of syntax and other inessential complications; see Appendix C for more information.

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

$$f^{(0)} \sim P(f)$$
$$\mathsf{x}^{(t)} = \arg\max_x f^{(t)}(x)$$
$$f^{(t+1)} \sim P\left(f^{(t)} \,\middle|\, f^{(t)}(\mathsf{x}^{(t)}) = \texttt{(f x}^{(t)}\texttt{)}\right).$$

We highlight two points:

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

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

(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 makes modularity, or the easy digestion of large but finely decomposable models, more difficult in statistics notation.

2. Adding a single line to the program, such as

```
(infer (mh 'params one 50))
```

after line 14 to infer the parameters of the emulator[3], would require a significant refactoring and elaboration of the statistics notation. One basic reason is that probabilistic programs are written procedurally, whereas statistical notation is declarative; reasoning declaratively about the dynamics of a fundamentally procedural inference algorithm is often unwieldy due to the absence of programming constructs such as loops and mutable state.

3 Bayesian Optimization Using Thompson Sampling

In this section, we introduce Thompson sampling, the basic solution strategy underlying the Bayesian optimization implementations in Section 4. First, we cast our Bayesian optimization task as a Markov decision process (MDP) and give a high-level algorithmic description of how Thompson sampling can be used to choose actions in an approximately optimal manner for this MDP. We note that probabilistic programming has the expressive power to support exploration of much richer context spaces than are typically used in computational Thompson sampling. Next, we present a code template for expressing Thompson sampling as a probabilistic program using a statistical memoizer. Finally, we give a full mathematical specification of one particular application of Thompson sampling, which (among others) will be implemented in Section 4.

3.1 Framework

Thompson sampling [13] is a widely used Bayesian framework for addressing the trade-off between exploration and exploitation in the so-called multi-armed (or continuum-armed) bandit problem.

[3] The definition of these parameters depends on the particular implementation of `mem&em`, and could for example be the hyperparameters of the covariance function of a GP.

In this subsection, 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 MDP.

The setup is as follows: An agent is to take a sequence of actions $a_1, a_2, \ldots$ 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 $P_{\text{true}}(r \mid a)$. 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 $\{P(r \mid a)\}_{a \in \mathcal{A}}$, 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 $V(a \mid \vartheta) = \mathbb{E}[r \mid a; \vartheta]$, which can be viewed as a believed value function. For consistency with what follows, we will assume our context ϑ takes the form $(\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$ where $\mathbf{a}_{\text{past}}$ is the vector of past actions, $\mathbf{r}_{\text{past}}$ is the vector of their rewards, and θ (the "semicontext") contains any other information that is included in the context.

In this setup, Thompson sampling has the following steps:

Algorithm 3.1: Thompson sampling.

Repeat as long as desired:

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

Note that when $\mathbb{E}[r|a;\vartheta]$ (under the sampled value of θ for some points a) is far from the true value $\mathbb{E}_{P_{\text{true}}}[r \mid a]$, the chosen action a may be far from optimal, but the information gained by probing action a will improve the belief ϑ. This amounts to "exploration." When $\mathbb{E}[r \mid a; \vartheta]$ is close to the true value except at points a for which $\mathbb{E}[r \mid a; \vartheta]$ 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 3.1. Roughly speaking, exploration will happen until the context ϑ is reasonably sure that the unexplored actions are probably not optimal, at which time the Thompson sampler will exploit by choosing actions in regions it knows to have high value.

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

3.2 Thompson Sampling with a Statistical Memoizer

Thompson sampling as described above can be expressed compactly in terms of a statistical memoizer, as in Listing 3.1 (see also Figure 3.2). Here and in what follows, to simplify the treatment, we assume the true reward function is deterministic, i.e., for each fixed a, the conditional distribution $P_{\text{true}}(r \mid a)$ is a delta distribution. We thus treat the notations $a, r, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}$ as synonyms for $x, y, \mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}}$, respectively, differing only in that they carry the connotations of "action" and "reward."

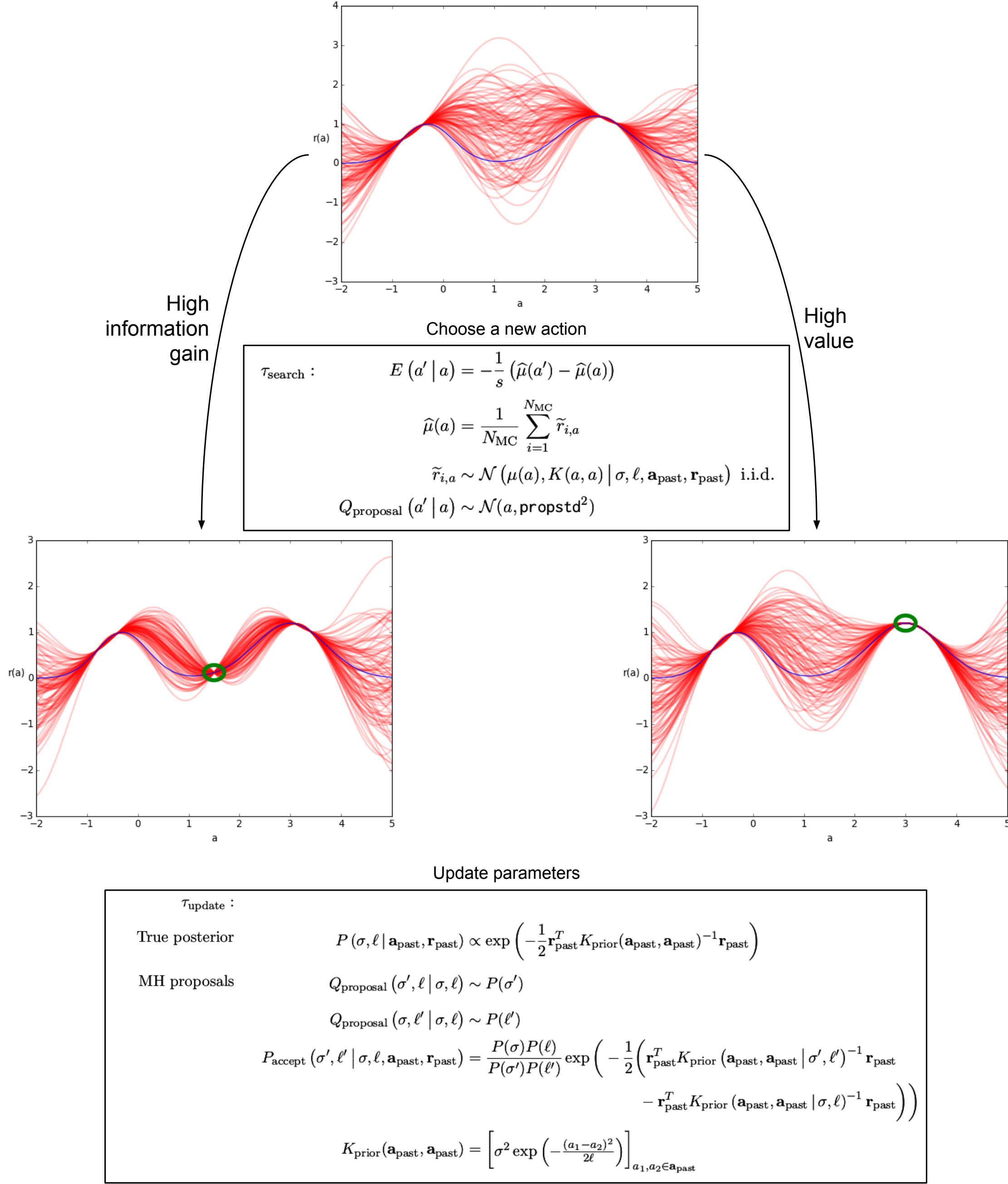


Figure 3.1. Two possible actions (in green) for an iteration of Thompson sampling. The believed distribution on the value function V is depicted in red. In this example, the true reward function is deterministic, and is drawn in blue. The action on the right receives a high reward, while the action on the left receives a low reward but greatly improves the accuracy of the believed distribution on V. The transition operators τ_{search} and τ_{update} are described in Section 3.3.

```
(assume theta (tag 'theta (prior_on_semicontexts)))
(assume_list (r_probe r_emulate)
             (mem&em do_action theta))
(let loop ((a (argmax r_emulate)))
  (r_probe a)
  (inference_on 'theta)
  (loop (argmax r_emulate)))
```

Listing 3.1. Code template for Thompson sampling in pseudo-Venture using a statistical memoizer. The choice of statistical memoizer (`mem&em`), prior on semicontexts (`prior_on_semicontexts`), and approximation strategy for maximizing the emulator (`argmax`) are not included.

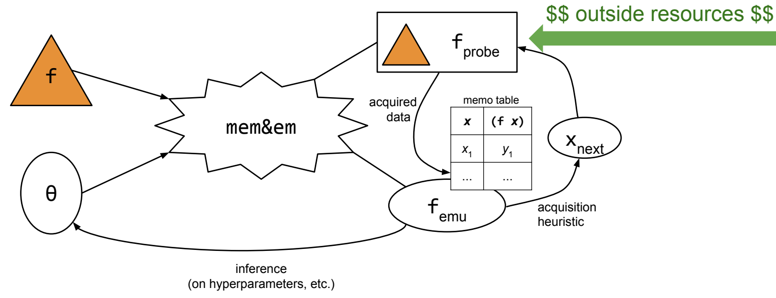
In Listing 3.1, `do_action` is a procedure which interfaces with the outside environment and returns the reward received. `do_action` is wrapped in the statistical memoizer `mem&em`, with parameters given by the semicontext `theta` (see Setion 3.1). `theta` is given a prior distribution, and as more actions `a` are probed, inference is performed on `theta`, and subsequent evaluations of the emulator `r_emulate` are affected accordingly. The action `a` chosen at each step is determined by the procedure `argmax`; for purposes of Thompson sampling, `argmax` (which takes a possibly stochastic procedure as its argument) should be implemented so that `(argmax func)` returns an approximation to $\arg\max_{\mathtt{a}} \mathbb{E}\left[\mathtt{(func\ a)}\right]$.

3.3 A Mathematical Specification

Below, we give the mathematical specification of a particular application of Thompson sampling. This application has the following properties:

- The regression function has a Gaussian process prior.
- The actions $a_1, a_2, \ldots \in \mathcal{A}$ 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.

Details of both the model and the inference algorithm are given in the subsections below. An implementation will be presented in Section 4.5.



In general:

$$P\left((\mathtt{f}_{\mathrm{emu}}\ \mathbf{x})\,\middle|\,\text{memo table} = (\mathbf{x}_{\mathrm{past}}, \mathbf{y}_{\mathrm{past}})\right) \sim P\left(f(\mathbf{x})\,\middle|\,f(\mathbf{x}_{\mathrm{past}}) = \mathbf{y}_{\mathrm{past}}\right),$$
$$\text{where } f \sim P\left(\mathtt{f}_{\mathrm{emu}}\,\middle|\,\text{memo table} = (\emptyset, \emptyset)\right)$$

For $\mathcal{GP}(\mu_{\mathrm{prior}}, K_{\mathrm{prior}})$ prior on $\mathtt{f}_{\mathrm{emu}}$:

$$P\left((\mathtt{f}_{\mathrm{emu}}\ \mathbf{x})\,|\,\mathbf{x}_{\mathrm{past}}, \mathbf{y}_{\mathrm{past}}\right) \sim \mathcal{N}\left(\mu(\mathbf{x}), K(\mathbf{x}, \mathbf{x})\right)$$
$$\mu(\mathbf{x}) = \mu_{\mathrm{prior}}(\mathbf{x}) + K_{\mathrm{prior}}(\mathbf{x}, \mathbf{x}_{\mathrm{past}})\, K_{\mathrm{prior}}(\mathbf{x}_{\mathrm{past}}, \mathbf{x}_{\mathrm{past}})^{-1} \left(\mathbf{y}_{\mathrm{past}} - \mu_{\mathrm{prior}}(\mathbf{x}_{\mathrm{past}})\right)$$
$$K(\mathbf{x}, \mathbf{x}) = K_{\mathrm{prior}}(\mathbf{x}, \mathbf{x}) - K_{\mathrm{prior}}(\mathbf{x}, \mathbf{x}_{\mathrm{past}})\, K_{\mathrm{prior}}(\mathbf{x}_{\mathrm{past}}, \mathbf{x}_{\mathrm{past}})^{-1} K_{\mathrm{prior}}(\mathbf{x}_{\mathrm{past}}, \mathbf{x})$$

If $\mathtt{f}$ is (nearly) smooth, then as $|\mathbf{x}_{\mathrm{past}}| \to \infty$, $\mathtt{f}_{\mathrm{emu}} \approx \mathtt{f}$.

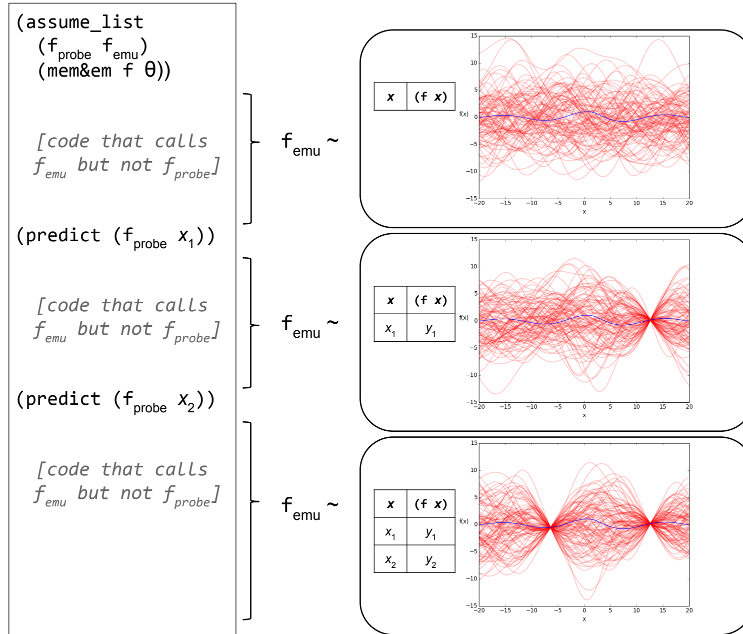


Figure 3.2. Top: Diagrammatic representation of Thompson sampling using the statistical memoizer `mem&em`. The emulator $\mathtt{f}_{\mathrm{emu}}$ is used to choose a point $\mathtt{x}_{\mathrm{next}}$ to acquire; the acquired data are incorporated into the memo table and the parameters θ are updated according to this new information. Middle: Mathematical specification of the statistical emulator $\mathtt{f}_{\mathrm{emu}}$. Bottom: Depiction of the state of the emulator after zero, one, and two probes have been taken. Here the true value function V is drawn in blue; the believed distribution on V is drawn in red. Between probes, $\mathtt{f}_{\mathrm{emu}}$ can be `sample`d but not `predict`ed, as its state (see Section 2.1) must not be changed. The state (i.e., the memo table) must only be changed when $\mathtt{f}_{\mathrm{probe}}$ is called.

3.3.1 Representations

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

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

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

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

where:

- μ_{prior} is a procedure to be used as the prior mean function. To simplify the treatment, we take $\mu_{\text{prior}} \equiv 0$.

- K_{prior} is a procedure to be used as the prior covariance function. In this example, we use a fixed functional form[4] $K_{\text{prior}} = K_{\text{prior}}\left(\bullet \,|\, \sigma, \ell\right)$, the squared exponential

$$K_{\text{prior}}(a, a') = \sigma^2 \exp\left(-\frac{(a - a')^2}{2\ell}\right).$$

- η is a set of (hyper)parameters for the mean and covariance functions; in this case $\eta = \{\sigma, \ell\}$.

[4] However, within this framework there is no reason why the functional form of K_{prior} cannot itself be random and be chosen by probabilistic inference. The support of $P(K_{\text{prior}})$ could, for example, be the search space of composite kernel structures explored in Duvenaud et al. [4].

The posterior mean and covariance for such a context ϑ are gotten by the usual conditioning formulas (assuming, for ease of exposition as above, that the prior mean is zero):[5]

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

Note that the context space Θ is not a finite-dimensional parametric family, since the vectors $\mathbf{a}_{\text{past}}$ and $\mathbf{r}_{\text{past}}$ grow as more samples are taken. Θ is, however, quite easily representable as a computational procedure together with parameters and past samples, as we do in the representation $\vartheta = (\mu_{\text{prior}}, K_{\text{prior}}, \eta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$.

3.3.2 Approximation Strategies

We combine the Update and Sample steps of Algorithm 3.1 by running a Metropolis–Hastings (MH) sampler whose stationary distribution is the posterior $P\left(\theta \,|\, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right)$. The functional forms of μ_{prior} and K_{prior} are fixed in our case, so inference is only done over the parameters $\eta = \{\sigma, \ell\}$; hence we equivalently write $P\left(\sigma, \ell \,|\, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right)$ for the stationary distribution. We make MH proposals to one variable at a time, using the prior as proposal distribution:

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

and

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

The MH acceptance probability for such a proposal is

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

[5] 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 $\left[K(a_i, a_j')\right]_{1 \le i \le n, 1 \le j \le n'}$.

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

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

The proposal and acceptance/rejection process described above define a transition operator τ_{update} which is iterated a specified number of times; the resulting state of the MH Markov chain is taken as the sampled semicontext θ in Step 1 of Algorithm 3.1.

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

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

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

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

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

$$E\left(a' \,\middle|\, a\right) = -\frac{1}{s}\left(\widehat{\mu}(a') - \widehat{\mu}(a)\right)$$

where

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

and

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

and $\{\widetilde{r}_{i,a}\}_{i=1}^{N_{\text{avg}}}$ are i.i.d. for a fixed a. Here the temperature parameter $s \geq 0$ and the population size N_{avg} are specified ahead of time. Note the following properties of the above acceptance probability:

- Proposals of estimated value higher than that of the current action are always accepted, while proposals of estimated value lower than that of the current action are accepted with a probability that decays exponentially with respect to the difference in value.

- The rate of the decay is determined by the temperature parameter s, where high temperature corresponds to generous acceptance probabilities. For $s = 0$, all proposals of lower value are rejected; for $s = \infty$, all proposals are accepted.

- For points a at which the posterior mean $\mu(a)$ is low but the posterior variance $K(a, a)$ is high, it is possible (especially when N_{avg} is small) to draw a "wild" value of $\widehat{\mu}(a)$, 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 3.1).[6,7]

4 Meta-Program

In this section we present a probabilistic meta-program for Bayesian optimization, along with three optimization routines implemented as concrete applications of the meta-program. We discuss the properties of each routine and provide plots of each routine running on synthetic examples. Afterwards, we briefly analyze and discuss the computational cost of the meta-program in terms of the costs of its components.

4.1 The Meta-Program

Figure 4.1 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

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

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

optimization. The components are themselves subprograms, whose implementations may vary. A traditional example of generic programming is the tree search meta-program of Norvig [8], written in LISP, which we present alongside the optimization meta-program for comparison.

Figure 4.2 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 flexibility of `optimize` is illustrated in Figure 4.3, where three concrete implementations (namely, those of Sections 4.2, 4.4, and 4.5) are shown.

The specifications for the arguments to the meta-procedure `optimize` in Figures 4.1 and 4.3 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.

```
(defun tree-search (states goal-p successors combiner)
  "Find a state that satisfies goal-p.  Start with states,
  and search according to successors and combiner."
  (cond ((null states) fail)
        ((funcall goal-p (first states)) (first states))
        (t (tree-search
             (funcall combiner
                      (funcall successors (first states))
                      (rest states))
             goal-p successors combiner))))
```

```
(define optimize
  (lambda (probe do_search search_state_box
           post_probe_inference extract_answer finished?)
    (let loop ()
      (if (finished?)
        (extract_answer)
        (begin
          (do_search)
          (if (contents_changed? search_state_box)
            (predict (probe ,(contents search_state_box))))
          (post_probe_inference)
          (loop))))))
```

Figure 4.1. Top: Norvig's LISP meta-program for tree search. Bottom: A probabilistic meta-program for Bayesian optimization, written in pseudo-Venture.

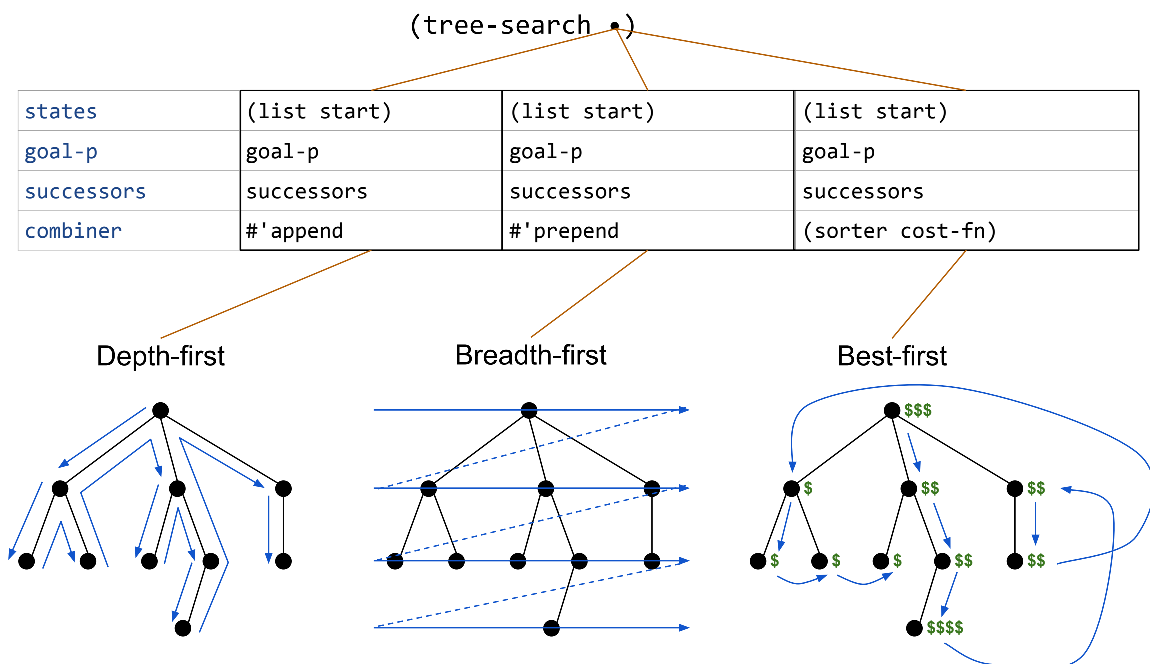


Figure 4.2. Invocations of Norvig's tree search meta-procedure with different arguments to perform a variety of different tree searches.

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.

4.2 Concrete Version 1: Random Exploration of GP Emulator

Listing 4.1 shows an implementation of Strategy 1 from Figure 4.3 as a set of concrete implementations for the components of `optimize`. The behavior of these components is as follows:

- The supplied function `f` (to be optimized) is wrapped in the statistical memoizer `gpmem`, which is backed by a Gaussian process prior. We supply the GP with prior mean zero and prior covariance function the squared exponential

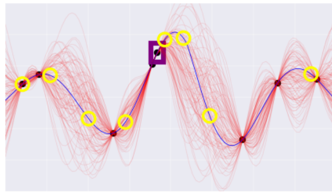
$$K_{\text{prior}}(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell}\right).$$

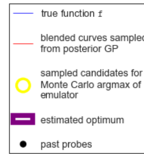
  Here σ and ℓ are continuous random variables, given a Uniform$([0, 10])$ prior, and will be

```
(optimize •)
```

	Strategy 1	Strategy 2	Strategy 3
probe	f_{probe}	f_{probe}	f_{probe}
do_search	(use_empirical_emu_argmax 20)	use_exact_parabola_argmax	(lambda () (gaussian_drift driftsteps_per_iter))
search_state_box	search_state_box	search_state_box	search_state_box
post_probe_inference	(mh_on_hypers N_MH)	(mh_on_parameters N_MH)	(mh_on_hypers N_MH)
extract_answer	best_probe_so_far	exact_parabola_max	best_probe_so_far
finished?	(probe_counter numprobes)	(probe_counter num_acquisitions)	(driftstep_counter (* num_iters driftsteps_per_iter))

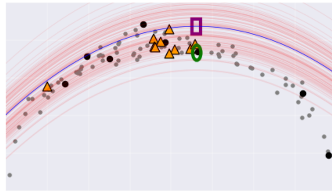
Strategy 1: GP regression, empirical argmax search strategy with uniform proposals

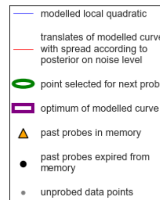




$$\begin{aligned} x_{\text{next}} &= \tilde{x}_{i^*} \\ i^* &= \arg\max_i \tilde{y}_i \\ \tilde{y}_i &\sim P\left((\mathsf{f}_{\text{emu}}\ \tilde{x}_i) \mid \theta, \mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}}\right) \\ \tilde{x}_i &\sim \text{Uniform}([-20, 20]) \text{ i.i.d.} \end{aligned}$$

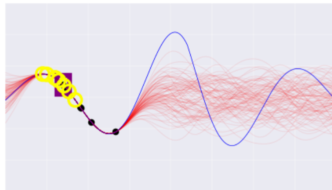
Strategy 2: Local quadratic regression with limited acquisition from a large data set, search by exact optimization

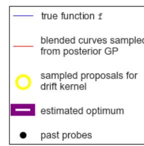




$$\begin{aligned} x_{\text{next}} &= \arg\min_{x \in D} |x - \hat{x}| \\ \hat{x} &= \underbrace{\arg\max_{x \in [-20,20]} \mathbb{E}\left[(\mathsf{f}_{\text{emu}}\ x) \mid \vartheta\right]}_{\text{exact optimization}} \end{aligned}$$

Strategy 3: GP regression, Metropolis search strategy with Gaussian drift proposals





$$\begin{aligned} x_{\text{next}} &= \tau_{\text{search}} \circ \cdots \circ \tau_{\text{search}}(x_{\text{current}}) \\ E\left(x' \mid x\right) &= -\frac{1}{s}\left(\hat{\mu}(x') - \hat{\mu}(x)\right) \\ \hat{\mu}(x) &= \frac{1}{N_{\text{avg}}} \sum_{i=1}^{N_{\text{avg}}} \tilde{y}_{i,x} \\ \tilde{y}_{i,x} &\sim \mathcal{N}\left(\mu(x), K(x,x) \mid \theta, \mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}}\right) \\ Q_{\text{proposal}}\left(x' \mid x\right) &\sim \mathcal{N}\left(x, \texttt{propstd}^2\right) \end{aligned}$$

Figure 4.3. Invocations of `optimize` with different arguments to perform a variety of different optimization routines.

```
(define (gp_uniform_search_mh_hypers fname numprobes)

  (assume sigma
    (tag 'hyper (uniform_continuous 0 10)))
  (assume l
    (tag 'hyper (uniform_continuous 0 10)))

  (assume_list (f_probe f_emu)
               (gpmem ,fname zero (make_squaredexp sigma l)))

  (define initial_state (uniform_continuous -20 20))
  (define search_state_box (make_box initial_state))

  (define (best_probe_so_far)
    (max (stats f_emu) second))

  (define (mh_on_hypers N_MH)
    (lambda () (mh 'hyper one N_MH)))

  (define (use_empirical_emu_argmax N_MC)
    (lambda ()
      (set_contents
        search_state_box
        (mc_argmax (joint->pointwise
                     (lambda (xs) (sample (f_emu xs))))
                   (iid_uniform_sampler -20 20)
                   N_MC))))

  (define (joint->pointwise func)
    (lambda (x)
      (first (func (array x)))))

  (define (probe_counter max)
    (lambda ()
      (= (size (stats f_emu))
         max)))

  (list f_probe                        ; probe
        (use_empirical_emu_argmax 20)  ; do_search
        search_state_box               ; search_state_box
        (mh_on_hypers 50)              ; post_probe_inference
        best_probe_so_far              ; extract_answer
        (probe_counter numprobes)))    ; finished?
```

Listing 4.1. Pseudo-Venture for Strategy 1 as a special case of `optimize`.

inferred as data are accumulated.

- Like `mem&em`, `gpmem` returns two procedures: a prober $\mathsf{f}_{\text{probe}}$ and an emulator f_{emu}. $\mathsf{f}_{\text{probe}}$ is supplied as the `probe` argument of `optimize`; f_{emu} is used as part of the search strategy `use_empirical_emu_argmax`, described below.

- `use_empirical_emu_argmax` chooses the next probe point by estimating the "arg max" of f_{emu}. Since the procedure f_{emu} is itself stochastic (it is a sampler from the conditional distribution $P\left(\texttt{(}\mathsf{f}_{\text{emu}}\ \bullet\texttt{)} \,|\, \text{memo table}\right)$, the ordinary notion of arg max does not directly apply. Instead, we take a Monte Carlo maximum (hence the name of the helper procedure `mc_argmax`) as follows: sample N_{MC} candidate points $\widetilde{x}_1, \ldots, \widetilde{x}_{N_{\text{MC}}}$ at random, and let $\widetilde{y}_i = \texttt{(}\mathsf{f}_{\text{emu}}\ \widetilde{x}_i\texttt{)}$. (Note that $\widetilde{y}_i$ is random, even when $\widetilde{x}_i$ is fixed.) Here the pairs $\{(\widetilde{x}_i, \widetilde{y}_i)\}$ are jointly independent, because a separate `sample` directive containing a call to f_{emu} is made for each $\widetilde{x}_i$ (see Section 1.3). In statistics notation, we have

$$\begin{aligned} \widetilde{x}_i &\sim P_{\text{MC}}(\widetilde{x}), \quad \text{i.i.d.,} \\ \widetilde{y}_i &\sim P\left(f(\widetilde{x}_i) \,|\, \theta \text{ and all previously gathered probe data}\right). \end{aligned}$$

  The population size N_{MC} and candidate distribution P_{MC} can be chosen freely, and in this implementation we take a population size of 20 and a Uniform$([-20, 20])$ candidate distribution. Then, the next chosen probe point is

$$x_{\text{next}} = \widetilde{x}_{i^*}, \quad \text{where } i^* = \arg\max_i \widetilde{y}_i.$$

- `mh_on_hypers` runs N_{MH} steps of Metropolis–Hastings inference on the hyperparameters σ and ℓ (here we have set $N_{\text{MH}} = 50$). The stationary distribution of this MH Markov chain is the

posterior distribution

$$
\begin{aligned}
&P\left(\sigma, \ell \,|\, \text{memo table accumulated from probes so far}\right) \\
&\propto P\left(\sigma, \ell, \mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}}\right) \\
&\propto \exp\left(-\frac{1}{2}\mathbf{y}_{\text{past}}^T K(\mathbf{x}_{\text{past}}, \mathbf{x}_{\text{past}})^{-1}\mathbf{y}_{\text{past}}\right).
\end{aligned}
$$

- `best_probe_so_far` simply returns the best probe point chosen so far. To unpack the notation, `(stats f_emu)` returns the list of pairs $\{(x, y)\}$, where x are the probe points and $y =$ `(f` x`)`; in functional programming terms, `(stats f_emu)` $= zip(\mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}})$. We take the pair with the largest the largest y.

- `(probe_counter numprobes)` simply decides we are finished after `numprobes` probes, with no dependence on program state.

4.3 Demonstrations

Below, we demonstrate three test cases for the concrete implementation of `optimize` in Section 4.2. In the first case, learning is essentially impossible; in the second case, learning is easy; in the third case, learning is more difficult due to the multimodality of the curve.

- **Delta function.** Let f be the approximate Kronecker delta function

$$
f(x) = \begin{cases} 1 & \text{if } |x - 5.2| < \epsilon \\ -1 & \text{otherwise,} \end{cases}
$$

  where ϵ is very small. Figure 4.4 shows a snapshot of the algorithm of Section 4.2 on f. Note that as long as no probes are taken within distance ϵ of the maximum, this function is indistinguishable from a constant function. As such, learning is impossible until the algorithm is "lucky" enough to choose the right probe, which, if ϵ is very small, will never happen. That is, as $\epsilon \to 0$, learning and optimization become impossible.

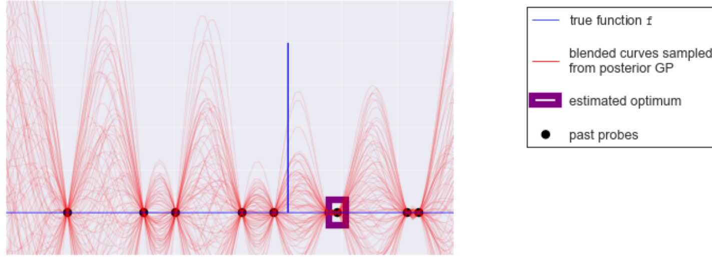


Figure 4.4. Trying to optimize a delta function with Strategy 1. No progress is made because, absent a very lucky choice of probe that is not at all suggested by previously accrued information, the function is indistinguishable from a constant function.

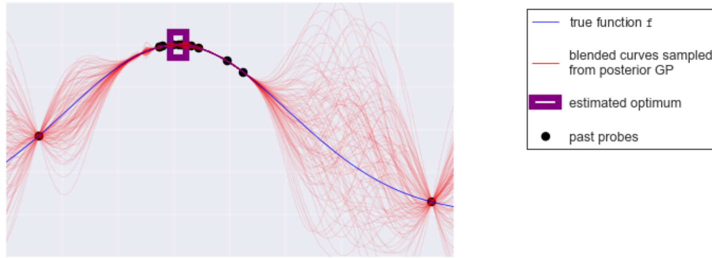


Figure 4.5. Optimizing a bell curve with Strategy 1.

- **Wide unimodal curve.** Let f be the bell curve

$$f(x) = -1 + 2\exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

with $\mu = -4.7$ and $\sigma = 10$. Figure 4.5 shows a snapshot of the algorithm of Section 4.2 on f. Unlike the delta function, this curve is smooth, and the local geometry of the curve provides hints as to where the mode lies. The algorithm is able to optimize this curve easily, though hill-climbing methods such as gradient ascent and drift kernels (see Section 4.5) might perform even better.

- **Trimodal curve.** Let f be the curve

$$f(x) = 0.2 + \exp(-0.1 \cdot |x-2|)\cos(0.4x),$$

which is trimodal over the domain of optimization $\mathcal{A} = [-20, 20]$. Figure 4.5 shows a snapshot

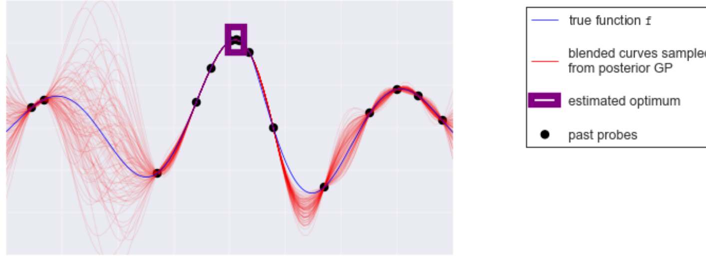


Figure 4.6. Optimizing a trimodal curve with Strategy 1. Unlike local search strategies like gradient ascent or drift kernels, Strategy 1 tends to eventually explore all three modes.

of Strategy 1 on f; a full trajectory of the algorithm can be seen in Figure A.1. Note that here hill-climbing methods might have difficulty finding the global maximum ($x \approx 0.61$) due to the existence of other local maxima. The `use_empirical_emu_argmax` search strategy tends to explore all three maxima[8], and indeed can be seen as a limiting case of annealing at high temperature.

4.4 Concrete Version 2: Exact Optimization of Locally Quadratic Emulator

Listing 4.2 presents a second set of implementations for the components of `optimize`. The resulting optimizer follows Strategy 2 (see Figure 4.3) with a noisy quadratic model:

$$f(x) \sim \mathcal{N}(f_{\text{noiseless}}(x), \texttt{noisestd}^2),$$

where $f_{\text{noiseless}} \sim P(f_{\text{noiseless}})$ is an unknown quadratic function. The prior on $f_{\text{noiseless}}$ in our case is induced by priors on the parameters:

$$f_{\text{noiseless}}(x) = -\mathsf{a}(x - \mathsf{b})^2 + \mathsf{c},$$

[8] Though, one still sees a higher concentration of probes near each mode, because points of high believed value are favored and each of the three modes has a relatively high value. However, the uniform distribution on candidate next probes $\widetilde{x}_i$ results in a greater propensity to explore remote areas of the search space than a localized search strategy like the Gaussian drift of Strategy 3.

where a, b and c, as well as the noise standard deviation noisestd, are treated as unknown and given independent uniform priors.[9] Learning is done actively, by acquiring a limited number of data points sequentially from a large data set. Regression is performed only on the L most recent acquired data points (where L is a supplied parameter) to allow for accurate local modelling of functions with varying local properties.

Figure A.2 shows a full trajectory of Strategy 2 on a synthetic data set. The test data were generated according to the process

$$\begin{aligned} x_i &\sim \text{Uniform}([-20, 20]) \\ y_i &\sim f_{\text{true}}(x_i) + \epsilon_i \\ \epsilon_i &\sim \mathcal{N}(0,\, 0.2^2) \text{ i.i.d.} \\ f_{\text{true}}(x) &= \log(20 - |x|) \end{aligned}$$

(see Figure A.3).

- The noisy quadratic model is encapsulated in the statistical memoizer quadmem, which takes a, b, c, and noisestd (in addition to the function f) as arguments. The posterior on these parameters is

$$\begin{aligned} &P\left(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{noisestd} \,|\, \mathsf{f}, \mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}}\right) \\ \propto &P\left(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{noisestd}, \mathbf{x}_{\text{past}}, \mathbf{y}_{\text{past}} \,|\, \mathsf{f}\right) \\ = &\prod_{i=1}^{n} \mathcal{N}\left(y_i; (\mathsf{f}\ x_i), \mathsf{noisestd}^2\right) \\ = &(2\pi\ \mathsf{noisestd}^2)^{-n/2} \exp\left(\frac{-1}{2\,\mathsf{noisestd}^2} \sum_{i=1}^{n} \left(y_i - (\mathsf{f}\ x_i)\right)^2\right). \end{aligned}$$

- The function f in this case is simply table lookup from a large data set. Thus, emulating the

[9] The priors on a, b and c are Uniform([0, 0.01]), Uniform([−20, 20]), and Uniform([−10, 10]), respectively. These ranges were chosen so that modelled curves would roughly fit inside the plot window of our figures, which we arbitrarily chose to be $[-20, 20] \times [-1, 4]$. The prior on noisestd is Uniform([0, 10]).

function `f` is the same as doing regression. That is, for a data set D,

$$\text{Regression on data set } D$$

$$\Updownarrow$$

$$\text{Statistical emulation of } f_D(x) = \begin{cases} D[x] & \text{if } x \in D \\ \text{Error} & \text{otherwise.} \end{cases}$$

- Conditioned on specific values of the parameters `b` and `c`, the maximum of the parabolic model occurs at $x =$ `b` and has value `c`. The search strategy `use_exact_parabola_argmax` simply chooses the closest data point to this maximum that has not been probed yet. Note that, unlike `best_probe_so_far` in Section 4.2, this answer may not be close to any points that have been probed—the model may predict an optimum in a region far away from the explored data. This may be useful, e.g., in situations where the data set is globally quadratic but all available data points are far away from the true optimum.

4.5 Concrete Version 3: Stochastic Local Search with GP Emulator

Listing 4.3 shows an implementation of Strategy 3, which was described in detail in Section 3.3. This strategy uses Metropolis search with Gaussian drift proposals:

$$Q_{\text{proposal}}\left(x' \,\middle|\, x\right) = \mathcal{N}(x'; x, \texttt{propstd}^2),$$

where the proposal width `propstd` is supplied as a parameter. The acceptance probability for such a proposal is

$$P_{\text{accept}}\left(x' \,\middle|\, x\right) = \min\left\{1,\ \exp\left(\frac{1}{s}\left(\texttt{(emu_avg}\ x'\ N_{\text{avg}}\texttt{)} - \texttt{(emu_avg}\ x\ N_{\text{avg}}\texttt{)}\right)\right)\right\},$$

where the temperature s and the sample size N_{avg} are provided as parameters. Here $\texttt{(emu_avg}\ x\ N_{\text{avg}}\texttt{)}$ (called $\widehat{\mu}(x)$ in the notation of Section 3.3) is a Monte Carlo estimate of $\mathbb{E}\left[\texttt{(f}_{\text{emu}}\ x\texttt{)}\right]$ over N_{avg}

```
(define (quad_exactmax_search_mh_params
          fname data_xs numprobes L N_MH)
  (assume a (tag 'param (uniform_continuous 0 0.01)))
  (assume b (tag 'param (uniform_continuous -20 20)))
  (assume c (tag 'param (uniform_continuous -10 10)))
  (assume noisestd (tag 'param (uniform_continuous 0 10)))

  (assume_list (f_probe f_emu)
               (quadmem ,fname a b c noisestd ,L))

  (define initial_state (uniform_continuous -20 20))
  (define search_state_box (make_box initial_state))

  (define (past_probes)
    (map first (stats f_emu)))
  (define (active_data_xs)
    (tail (past_probes) L))
  (define (closest_valid_input x)
    (closest_point x (exclude (active_data_xs)
                              data_xs)))

  (define (use_exact_parabola_argmax)
    (set_contents search_state_box
                  (closest_valid_input (sample b))))

  (define (exact_parabola_max)
    (sample (list b c)))

  (define (mh_on_parameters N_MH)
    (lambda () (mh 'param N_MH)))

  (define (probe_counter max)
    (lambda ()
      (= (size (past_probes))
         max)))

  (list f_probe                          ; probe
        use_exact_parabola_argmax        ; do_search
        search_state_box                 ; search_state_box
        (mh_on_parameters N_MH)          ; post_probe_inference
        exact_parabola_max               ; extract_answer
        (probe_counter numprobes)))      ; finished?
```

Listing 4.2. Pseudo-Venture for Strategy 2 as a special case of `optimize`.

samples. Roughly speaking, proposals should be accepted in accordance with how much higher the emulator thinks their values are than the current state's value.

Note that in bad cases, few drift proposals will be accepted and it may take many tries for the drift kernel to find a new action to probe. Therefore, the end condition of this implementation is based on the total number of iterations through the loop, rather than the number of probes taken.

Figure 4.7 shows a trajectory of Strategy 3 on the trimodal curve of Section 4.3, as well as time series of the learned optima over several separate invocations. As we would expect, compared to Strategy 1, this strategy has a greater tendency to choose probes near already explored actions. Consequently, each invocation finds and stays at one of the curve's three local optima. Because the width of the drift proposals (in the case pictured, `propstd` = 0.5) is significantly less than the distance between the local optima (approximately 16), it is quite difficult for a trajectory to leave a local optimum once it arrives, and so performance is extremely dependent on the choice of initialization. This is not the case with Strategy 1: we can see in Figure A.1 that even trajectories that get stuck at one of the smaller local optima eventually find their way to the global optimum.

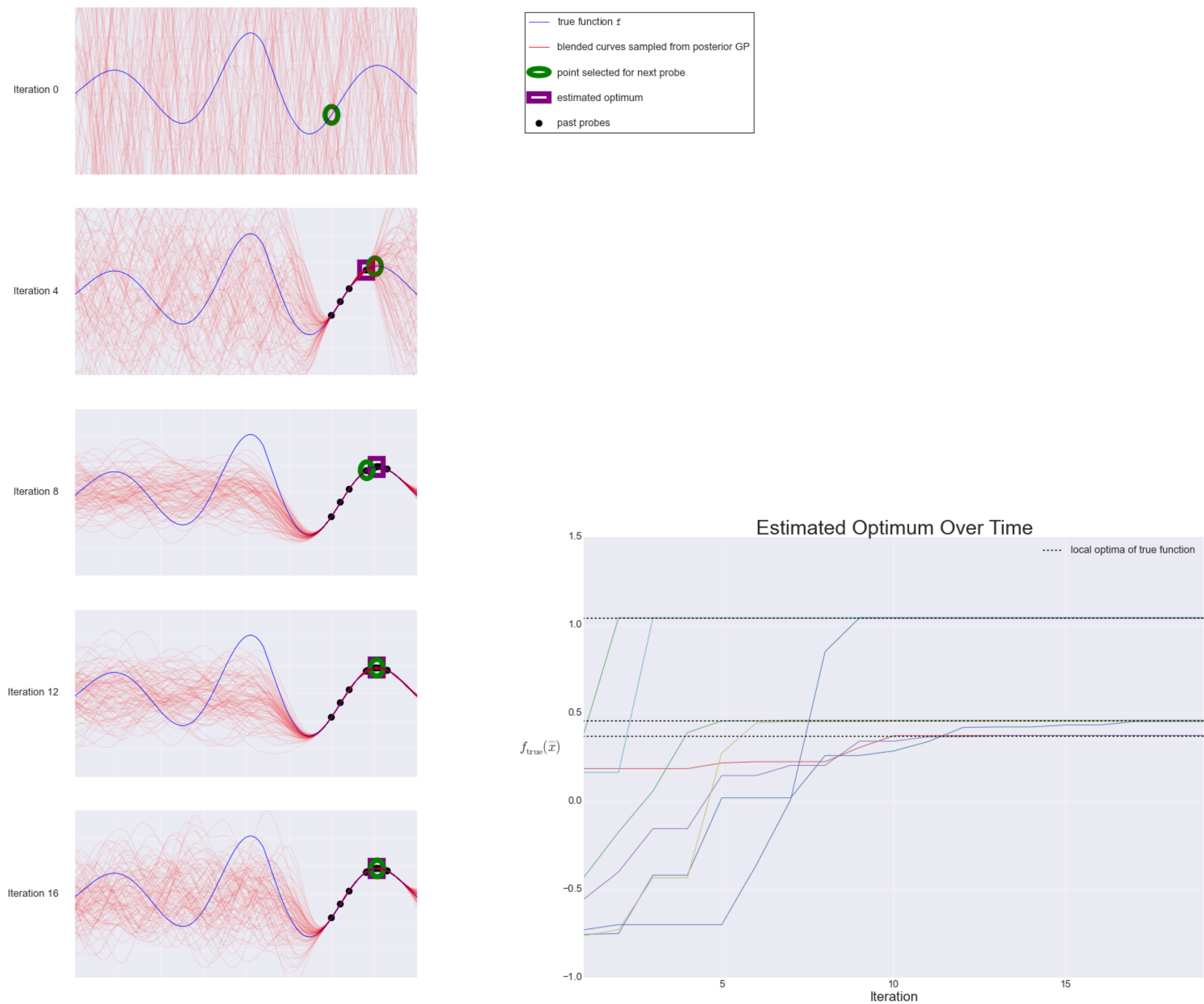


Figure 4.7. Trajectory of Strategy 3 on a trimodal curve, along with line plots of the learned optima $\widehat{x}$ over several separate invocations.

```
(define (gp_gaussian_drift_mh_hypers
           fname num_iters driftsteps_per_iter
           propstd s initial_state N_avg N_MH)
  (assume sigma (tag 'hyper (uniform_continuous 0 10)))
  (assume l (tag 'hyper (uniform_continuous 0 10)))
  (assume_list (f_probe f_emu)
               (gpmem ,fname zero (make_squaredexp sigma l)))

  (define emu_at_point
    (joint->pointwise (lambda (xs) (sample (f_emu ,xs)))))

  (define (emu_avg x K)
    (let ((dct (alist_to_dict
                 (stats f_emu))))
      (if (contains dct x)
        (lookup dct x)
        (let ((samps (repeat0 K (lambda () (emu_at_point x)))))
          (/ (apply + samps) K)))))

  (define drift_stepcount (make_box 0))
  (define search_state_box (make_box initial_state))
  (mark_recently_changed search_state_box)

  (define (best_probe_so_far)
    (max (stats f_emu) second))

  (define (mh_on_hypers N_MH)
    (lambda () (mh 'hyper one N_MH)))

  (define (generate_proposal_from x)
    (confine (normal x propstd) -20 20))

  (define (gaussian_drift driftsteps)
    (let loop ((steps_remaining driftsteps))
      (if (> steps_remaining 0)
        (let ((proposed_state (generate_proposal_from
                                (peek search_state_box)))
              (acc_ratio
                (min 1.0
                     (exp (* (/ 1 s)
                             (- (emu_avg proposed_state N_avg)
                                (emu_avg current_state N_avg)))))))
          (if (flip acc_ratio)
            (set_contents search_state_box proposed_state))
          (increment drift_stepcount)
```

```
        (loop (- steps_remaining 1))))))

(define (choose_next_point)
  current_state)

(define (driftstep_counter max)
  (lambda ()
    (>= (peek drift_stepcount) max)))

(list f_probe                                 ; probe
      (lambda () (gaussian_drift              ;
                   driftsteps_per_iter))      ; do_search
      search_state_box                        ; search_state_box
      (mh_on_hypers N_MH)                     ; post_probe_inference
      best_probe_so_far                       ; extract_answer
      (driftstep_counter                      ;
        (* num_iters                          ;
           driftsteps_per_iter))))            ; finished?
```

Listing 4.3. Pseudo-Venture for Strategy 3 as a special case of `optimize`.

4.6 Computational Cost

We now analyze the computational cost of the optimization meta-program in Figure 4.1 in terms of the costs of its components. After some remarks on these costs in general, we specialize to estimate the time complexity of each of the three concrete implementations in Section 4.

We denote the cost of component `cmp` as $T_{\texttt{cmp}}$, or $T^{(i)}_{\texttt{cmp}}$ for the ith iteration of `cmp` if multiple iterations occur. As the choice of letter T suggests, our primary focus will be time complexity.

Let $N+1$ be the total number of times `loop` is called during an execution of `optimize` in Figure 4.1, starting at iteration $i=0$ and ending at iteration $i=N$ during which `extract_answer` is called. Note that N may be random, e.g., by depending (randomly or deterministically) on the state of the optimizer which is itself stochastic. We have

$$T_{\texttt{optimize}} = \sum_{i=0}^{N} T^{(i)}_{\texttt{finished?}} + \sum_{i=0}^{N-1} T^{(i)}_{\texttt{do_search}} + \sum_{i=0}^{N-1} T^{(i)}_{\texttt{probe}} + \sum_{i=0}^{N-1} T^{(i)}_{\texttt{post_probe_inference}} + T_{\texttt{extract_answer}} + \sum_{i=0}^{N} O(1).$$

We suggest that each component could, in some reasonable paradigm, be either cheap or costly.

- In our three concrete implementations, `finished?` is constant-time. In other implementations,

finished? could be written to end the loop when inference quality reaches a certain level, according to some inference quality heuristic. Depending on the heuristic, the computational cost could be arbitrarily high (e.g., checking whether two computable distributions are equal is uncomputable [1]). Heuristics may also make use of resources other than time, such as outside data sources (e.g., validation on a test set) or human expertise (e.g., hand-inspection of the model by an expert).

- do_search embodies the rule for acquiring new data points. It stands to reason that the cost of do_search is non-negligible only in implementations that learn actively. In our GP-based implementations, the acquisition heuristic involves sampling the current posterior GP, and so the ith iteration involves inverting an $i \times i$ matrix. Other implementations could conceivably include acquisition heuristics that make fine-grained trade-offs between quality of probe points and resources spent choosing them, so that do_search does not use an undesirably large amount of computation as i becomes large.

- probe is where the expensive function f is called. If f is costly in terms of time, then the time complexity of optimize may be dominated by the calls to probe. If f is costly in terms of another resource, such as human time or outside data sources, then we can view Bayesian optimization as trading off computational resources for these other resources.

- post_probe_inference is open-ended, containing any inference required to take a newly acquired data point into account. In the three implementations above, this consisted solely of inference on parameters. In other implementations, post_probe_inference may include further subroutines, such as inference on the structure of the covariance function of a GP.

- In our three concrete implementations, extract_answer is cheap (either constant-time or a linear search of size N). In other implementations, extract_answer could perform optimization on a model of the function f that was trained during previous iterations of the loop. For example, in the GP-based implementations, the symbolic forms of the prior mean and covariance are known, so a symbolic form of the posterior mean can be computed, and extract_answer could output the maximum of the posterior mean, computed either numerically or symboli-

cally.

In Appendix B, we apply the above equation to estimate the time complexity of each of the concrete implementations in Sections 4.2, 4.4, and 4.5.

5 Conclusion

We have presented a probabilistic meta-program for Bayesian optimization. By applying the meta-program to three diverse sets of component subprograms, we have demonstrated that our meta-algorithm is general enough to include a wide range of search strategies and regression frameworks. In particular, we have described the informal semantics of Gaussian process models in Venture, and have demonstrated their use as statistical memoizers. By using Gaussian processes to implement Thompson sampling as a probabilistic program, we have been able to call attention to ways in which the expressivity of probabilistic programming languages can potentially expand the range and complexity of available modelling and inference techniques beyond those typically considered in traditional statistics.

A Additional Figures

In this appendix, we present several figures that were omitted from the main text for reasons of flow.

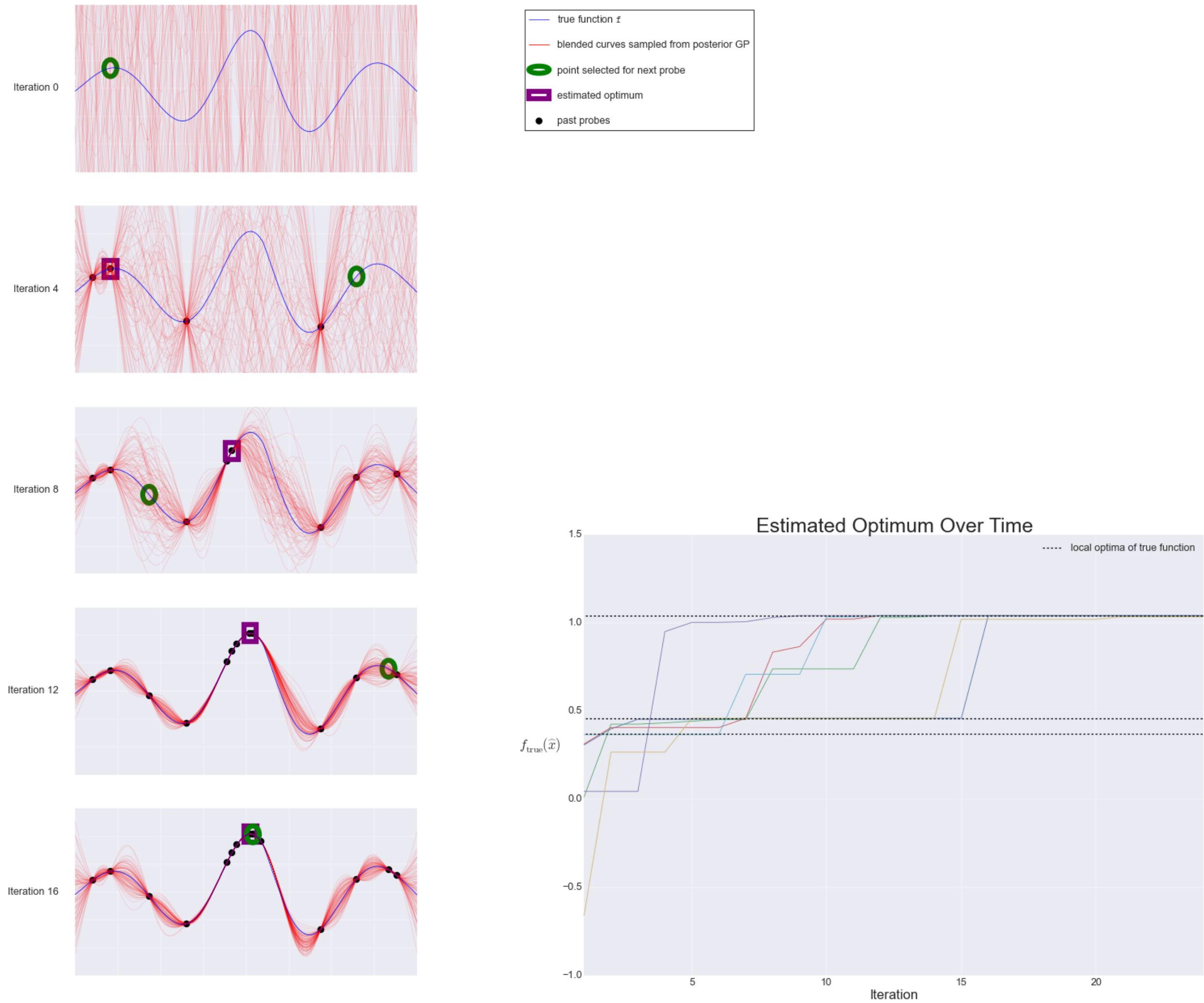


Figure A.1. Trajectory of Strategy 1 on a trimodal curve, along with line plots of the learned optima $\widehat{x}$ over several separate invocations. Unlike hill-climbing methods such as gradient ascent or drift kernels, the `use_empirical_emu_argmax` search strategy tends to explore all three local optima.

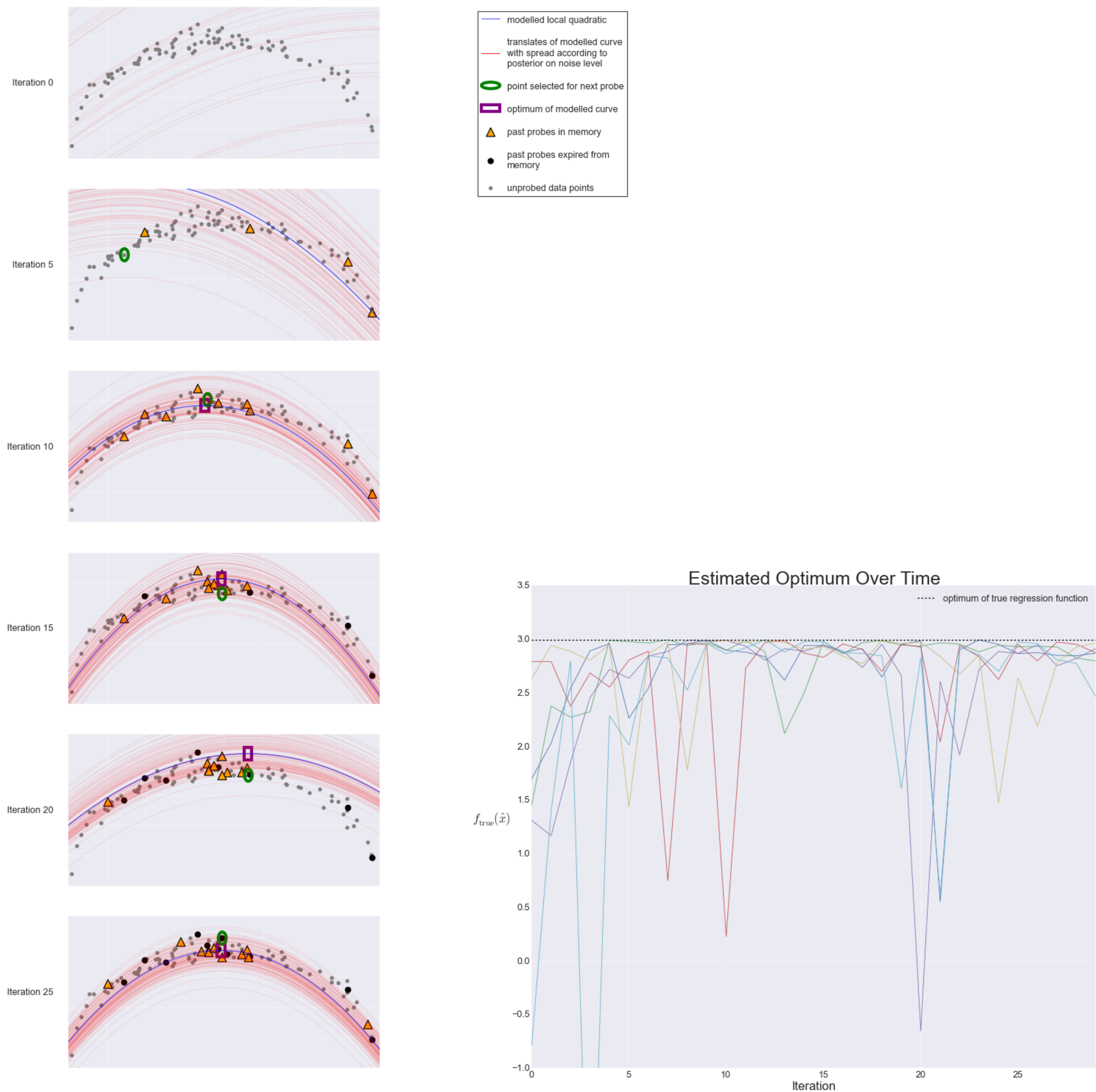


Figure A.2. Trajectory of Strategy 2 on the data set described in Section 4.4, along with line plots of the learned optima over several separate invocations. Regression is only done on the most recent L probes, where here $L = 10$. This limited-memory setup is part of the reason that the learned optimum $\widehat{x}$ occasionally becomes poor for a brief time before becoming accurate again (the other part being occasional draws from unlikely regions of the approximate posterior on parameters). The value of $\widehat{x}$ would be more stable (after suitably many probes) if L were made larger.

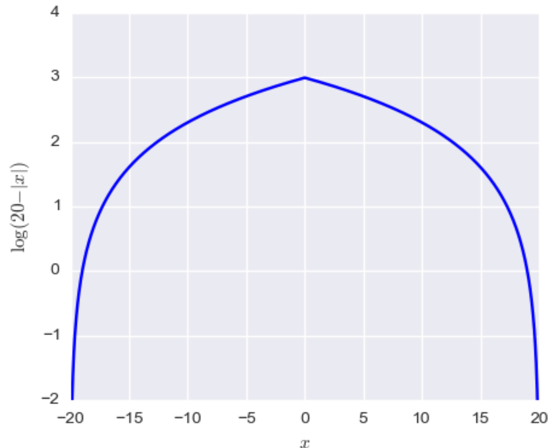


Figure A.3. True regression function for the process that generated the test data in Figure A.2 (see Section 4.4 for a description of the process): $f_{\text{true}}(x) = \log(20 - |x|)$.

B Time Complexities of the Concrete Implementations

In this appendix, we specialize the framework of Section 4.6 to analyze the asymptotic time complexity (excluding any time taken by evaluations of $\texttt{f}_{\text{probe}}$) of the concrete implementations of `optimize` presented in Sections 4.2, 4.4, and 4.5. The notations used in Section 4.6 (e.g., definitions of N and i) will be used here.

- Strategy 1: The most expensive operations are matrix inversion (i.e., computing $K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1}$) and matrix-times-vector multiplications for sampling (during `mc_argmax`) and computing the log-likelihood of new hyperparameters (during `mh_on_hypers`). At step i, the size of $K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})$ is $i \times i$, so the time complexity of this step is $O\left(i^{\alpha} + (N_{\text{MC}} + N_{\text{MH}})i^2\right)$, where α is the exponent of matrix inversion (e.g., for the Strassen algorithm $\alpha \approx 2.8$).[10] Thus, the total time complexity is

$$\sum_{i=1}^{N-1} O\left(i^{\alpha} + (N_{\text{MC}} + N_{\text{MH}})\, i^2\right) = O\left(N^{\alpha+1} + (N_{\text{MC}} + N_{\text{MH}})N^3\right).$$

- Strategy 2: All operations are quite easy, except the MH inference.[11] If there are L past probes $\{(x_1, y_1), \ldots, (x_L, y_L)\}$ in memory, then the log-likelihood of parameters $\texttt{a}, \texttt{b}, \texttt{c}, \texttt{noisestd}$ is

$$-L\left(\log \texttt{noisestd} + \frac{1}{2}\log 2\pi\right) - \frac{1}{2\,\texttt{noisestd}^2}\sum_{i=1}^{L}(y_i - f_{\text{reg}}\left(x_i;\, \texttt{a}, \texttt{b}, \texttt{c}\right))^2,$$

[10] As written, the algorithm inverts $K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})$ multiple times per iteration, with the result that the running time has a worse dependence on N_{MC} and N_{MH}. However, for purposes of analysis we assume that the simple optimization of caching the value of $K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1}$ is used.

[11] Perhaps, if `closest_point` performs a linear search over a large data set, `closest_point` could have significant cost. However, for this analysis, we assume that access to the data is mediated by an external interface. We further assume that the external interface evaluates `closest_point` for us, and we discount the time taken by these queries.

where $f_{\text{reg}}(x; \mathsf{a}, \mathsf{b}, \mathsf{c}) = -\mathsf{a}(x - \mathsf{b})^2 + \mathsf{c}$ is the parametrized regression function. Assuming constant-time arithmetic and logs, this log-likelihood can be computed in $O(L)$ time. Thus the total running time of this strategy (aside from data acquisition) is $O(N_{\text{MH}} N L)$.

- Strategy 3: The analysis is similar to that of Strategy 1. Now, however, N_{avg} point samples are taken during each drift step, each requiring matrix-times-vector multiplications, so `do_search` incurs a cost of $O\left(i^2 \cdot N_{\text{avg}} \cdot \mathtt{driftsteps_per_iter}\right)$ aside from matrix inversion, for a total time cost of

$$\sum_{i=1}^{N-1} O\left(i^\alpha + \left(N_{\text{avg}} \cdot \mathtt{driftsteps_per_iter} + N_{\text{MH}}\right) i^2\right)$$
$$= O\left(N^{\alpha+1} + (N_{\text{avg}} \cdot \mathtt{driftsteps_per_iter} + N_{\text{MH}}) N^3\right).$$

C Attachments

Several of the code samples given in the above sections are written in "pseudo-Venture," a pseudocode language that is essentially the same as Venture but elides some of the rough edges of syntax and looks more similar to Scheme (which the earlier versions of Venture were entirely based on, due to its descendance from Church [5]). The primary differences between Venture and pseudo-Venture are as follows:

- The existence of Haskell-inspired "inference actions," particularly `do`-blocks. All sequences of expressions (including what in pseudo-Venture are `begin` and the bodies of `lambda`s) are written as do-blocks; internal `define`s are replaced by lines of the form `(name <- value)` inside `do`-blocks. We also omit from pseudocode the command `run`, which executes a supplied inference action.

- The lack of certain syntactic sugars:

Sugared	*Unsugared*
`(define (f arg1 arg2) body)`	`[define f (lambda (arg1 arg2) body)]`

`(assume_list (a b) e)`	`[assume package e]`
	`[assume a (first package)]`
	`[assume b (second package)]`

As a point of comparison, Figure C.1 shows both pseudo-Venture and Venture versions of the Bayesian optimization meta-program from Figure 4.1.

Figure 4.1 and Listings 4.1, 4.2, and 4.3 contain pseudo-Venture. The corresponding runnable Venture code is located in `bayesopt.vnt`. Instructions for running the optimization routines and plotting the results are included in `README.md`.

```
(define optimize
  (lambda (probe do_search search_state_box
           post_probe_inference extract_answer finished?)
    (let loop ()
      (if (finished?)
        (extract_answer)
        (begin
          (do_search)
          (if (contents_changed? search_state_box)
            (predict (probe ,(contents search_state_box))))
          (post_probe_inference)
          (loop))))))
```

```
[define optimize
  (lambda (probe do_search search_state_box
           post_probe_inference extract_answer check_finished)
    (letrec
     ((loop (lambda ()
              (if (check_finished)
                (extract_answer)
                (do
                  (do_search)
                  (if (contents_changed search_state_box)
                    (probe (contents search_state_box))
                    (return (print "Search state unchanged")))
                  (post_probe_inference)
                  (loop))))))
      (loop)))]
```

Figure C.1. Top: Pseudo-Venture for the Bayesian optimization meta-program. Bottom: Venture for the same meta-program.

References

[1] Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. On the computability of conditional probability. *arXiv:1005.3014v2*, 2011.

[2] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010.

[3] Adam D. Bull. Convergence rates of efficient global optimization algorithms. *J. Mach. Learn. Res.*, 12:2879–2904, November 2011.

[4] D. Duvenaud, J. R. Lloyd, R. Grosse, J. Tenenbaum, and Z. Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1166–1174, 2013.

[5] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008.

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

[7] Kevin Miller, M. P. Kumar, Benjamin Packer, Danny Goodman, and Daphne Koller. Max-margin min-entropy models. In Neil D. Lawrence and Mark A. Girolami, editors, *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (AISTATS-12)*, volume 22, pages 779–787, 2012.

[8] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

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

[10] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[11] Niranjan Srinivas, Andreas Krause, Matthias Seeger, and Sham M. Kakade. Gaussian process optimization in the bandit setting: No regret and experimental design. In Johannes Frnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1015–1022. Omnipress, 2010.

[12] Venture Team. VentureScript reference manual. http://probcomp.csail.mit.edu/venture/edge/reference/index.html, 2015.

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

README.md

````
Unfortunately, MIT's DSpace repository, despite being hosted on the Web, does
not allow the submission of any supplementary files other than a single PDF.  I
will therefore include the most conceptually important code directly into my
thesis, and exclude all predictable or uninteresting code, such as plotting
code (of which, trust me, there is a lot).  If you would like to reproduce the
plots or use this code for anything other than reading, please contact me at
bzinberg@alum.mit.edu and I will send non-PDFs to you.

The code in `bayesopt.vnt` (included below) is "the point," and detailed
pseudocode for it has already been given in the text.  As far as supporting
code, I have included:

* `gpexample_plugin.py` -- mostly just because it contains the definitions of
  the example functions (`trimodal`, `wide_unimodal`, `spike`, `bimodal`).  It
  also shows an example of how callbacks are created for collecting data and
  dumping it into files.
* `gpmem.py` and `quadmem.py` -- the statistical memoizers used in
  `bayesopt.vnt`.  This code will not make sense unless you are already
  familiar with the internals of Venture.

To give you an idea of what sane arguments to the procedures in `bayesopt.vnt`
look like, here are some examples:

The trajectory figure for Strategy 1 was created using data from the following
command:

```
$ venture --a -f bayesopt.vnt -e "(gp_uniform_main 'trimodal 25 'testrun)"
```

The trajectory figure for Strategy 2 was created using data from the following
command:

```
$ venture --a -f bayesopt.vnt -e "(quad_main 'LogAbs 30 10 200 -18.0 'testrun)"
```

The trajectory figure for Strategy 3 was created using data from the following
command:

```
$ venture --a -f bayesopt.vnt -e "(gp_drift_main 'trimodal 20 10 0.5 0.125 \
  'choose_for_me 10 50 'testrun)"
```

These were tested on commit 83963ce949430a5fe86d0a7bfaeb29cea703e4e5 of the
Venture software, but should also work on a recent release such as Developer
Alpha v0.4.1, available [here][1] ([direct link][2], [tutorial][3],
[documentation][4]).  Beware that the tutorial and documentation are written in
the JavaScript-like syntax, while all my code is written in the Scheme-like
syntax (hence I invoke Venture with the `--abstract-syntax` option).

[1]: http://probcomp.csail.mit.edu/venture/
[2]: http://probcomp.csail.mit.edu/venture/release-0.4.1/venture-0.4.1.tgz
````

[3]: http://probcomp.csail.mit.edu/venture/release-0.4.1/tutorial/
[4]: http://probcomp.csail.mit.edu/venture/release-0.4.1/reference/index.html

---

```
[infer (load_plugin "gpexample_plugin.py")]
[infer (load_plugin "quadexample_plugin.py")]
[infer (load_plugin "box.py")]
[infer (load_plugin "exclude.py")]

;;; The meta-program
[define optimize
  (lambda (probe do_search search_state_box
           post_probe_inference extract_answer check_finished)
    (letrec
     ((loop (lambda ()
              (if (check_finished)
                (extract_answer)
                (do
                  (do_search)
                  (if (contents_changed search_state_box)
                    (probe (contents search_state_box))
                    (return (print "Search state unchanged")))
                  (post_probe_inference)
                  (loop))))))
      (loop)))]

;;; Strategy 1
[define gp_uniform_search_mh_hypers
   (lambda (fname numprobes func_id datafile_prefix) (do
     (assume sigma (tag 'hyper 0 (uniform_continuous 0 10)))
     (assume l (tag 'hyper 1 (uniform_continuous 0 10)))
     (assume package ((allocate_gpmem) ,fname zero (make_squaredexp sigma l)))

     (initial_state <- (return (uniform_continuous -20 20)))
     (search_state_box <- (return (make_box initial_state)))

     (best_probe_so_far <- (return (lambda ()
       (return (keyed_max (run (extract_stats (second package)))
                          get2nd)))))

     (mh_on_hypers <- (return (lambda (N_MH)
          (lambda () (mh 'hyper one N_MH)))))

     (use_empirical_emu_argmax <- (return (lambda (N_MC)
       (lambda () (return
         (set_contents
           search_state_box
           (mc_argmax
             (joint_to_pointwise
               (lambda (x) (run (sample ((second package) ',x)))))
             (iid_uniform_sampler -20 20)
             N_MC)))))))

     (probe_counter <- (return (lambda (max)
       (lambda ()
         (= (size (run (extract_stats (second package))))
            max)))))
```

```
    ;; For data collection and plotting
    (collect_stats <- (return (lambda () (do
            (stats <- (extract_stats (second package)))
            (call_back add_gp_plot_data sigma l ',stats)))))

    (return (array
          (lambda (x) (predict ((first package) ,x)))
          (run (do
            ;; First collect data for plots, then do the actual searching
            (do_search <- (return (use_empirical_emu_argmax 20)))
            (return (lambda () (do
              (collect_stats)
              (do_search))))))
          search_state_box
          (mh_on_hypers 50)
          (lambda () (do
            ;; First dump the collected data to a file, then extract answer
            (call_back dump_gp_plot_data 'uniform func_id ',datafile_prefix)
            (best_probe_so_far)))
          (probe_counter numprobes)))))]

;;; Strategy 2
[define quad_exactmax_search_mh_params
   (lambda (fname data_xs numprobes L N_MH initial_state data_generator_name
           datafile_prefix) (do
     (assume a (tag 'param 0 (uniform_continuous 0 0.01)))
     (assume b (tag 'param 1 (uniform_continuous -20 20)))
     (assume c (tag 'param 2 (uniform_continuous -10 10)))
     (assume noisestd (tag 'param 3 (uniform_continuous 0 10)))
     (assume package ((allocate_quadmem) ,fname a b c noisestd ,L))

     (search_state_box <- (return (make_box initial_state)))

     (past_probes <- (return (lambda ()
        (mapv get1st (run (extract_stats (second package)))))))
     (active_data_xs <- (return (lambda ()
          (tail (past_probes) L))))
     (closest_valid_input <- (return (lambda (x)
       (closest_point x (exclude
                         (active_data_xs)
                         data_xs)))))

     (use_exact_parabola_argmax <- (return (lambda () (return
       (set_contents search_state_box
             (closest_valid_input (run (sample b))))))))

     (exact_parabola_max <- (return (lambda () (do
         (sample (array b c))))))

     (mh_on_parameters <- (return (lambda (N_MH) (lambda ()
       (mh 'param one N_MH)))))

     (probe_counter <- (return (lambda (max)
                         (lambda ()
                           (= (size (past_probes))
                              max)))))
```

```
    ;; Use an explicit priority counter to determine which past probes
    ;; stay in the emulator's memory -- can't use clock time as the
    ;; priority because that would screw up resimulation-based inference
    ;; methods
    (pri_counter <- (return (make_box 0)))

    (return (array
          (lambda (x)
            (predict ((first package) ,x ,(increment pri_counter))))
          (lambda () (do
            ;; First collect data for plots, then search
            (call_back add_quad_plot_data
                       a b c noisestd ,L ',data_xs ys ',(active_data_xs))
            (use_exact_parabola_argmax)))
          search_state_box
          (mh_on_parameters N_MH)
          (lambda () (do
            ;; First dump data to a file, then extract answer
            (call_back dump_quad_plot_datas ',data_generator_name ',datafile_prefix)
            (exact_parabola_max)))
          (probe_counter numprobes)))))]

;;; Strategy 3
[define gp_gaussian_drift_mh_hypers
   (lambda (fname num_iters driftsteps_per_iter propstd s initial_state N_avg
            N_MH func_id datafile_prefix) (do
    (assume sigma (tag 'hyper 0 (uniform_continuous 0 10)))
    (assume l (tag 'hyper 1 (uniform_continuous 0 10)))
    (assume package ((allocate_gpmem) ,fname zero (make_squaredexp sigma l)))

    (emu_at_point <- (return
      (joint_to_pointwise
        (lambda (xs) (run (sample ((second package) ',xs)))))))

    (emu_avg <- (return (lambda (x K)
      (let ((dct (alist_to_dict
                   (run (extract_stats (second package))))))
        (if (contains dct x)
          (lookup dct x)
          (run (do
            (samps <- (return (repeatO  K (lambda () (emu_at_point x)))))
            (return (/ (apply + samps) K)))))))))

    (drift_stepcount <- (return (make_box 0)))
    (search_state_box <- (return (make_box
                               (if (= initial_state 'choose_for_me)
                                 (uniform_continuous -20 20)
                                 initial_state))))
    (return (mark_recently_changed search_state_box))

    (best_probe_so_far <- (return (lambda () (return
      (keyed_max (run (extract_stats (second package)))
                 get2nd)))))

    (mh_on_hypers <- (return (lambda (N_MH) (lambda ()
```

```
          (mh 'hyper one N_MH)))))

      (gaussian_drift <- (return (letrec
        ((generate_proposal_from (lambda (x)
                                   (confine (normal x propstd) -20 20)))
         (drift (lambda (steps_remaining)
           (if (= steps_remaining 0)
             pass
             (do
               (proposed_state <- (return (generate_proposal_from
                                             (peek search_state_box))))
               (acc_ratio <- (return
                   (min 1.0
                        ;; Numerical badness in exchange for clarity
                        (exp (* (/ 1 s)
                                (- (emu_avg proposed_state N_avg)
                                   (emu_avg (bget search_state_box) N_avg)))))))
               (if (flip acc_ratio)
                 (return (set_contents search_state_box proposed_state))
                 pass)
               (return (increment drift_stepcount))
               (drift (- steps_remaining 1)))))))
        drift)))

      (driftstep_counter <- (return
        (lambda (max)
          (lambda ()
            (>= (peek drift_stepcount)
                max)))))

      (collect_stats <- (return (lambda () (do
        (stats <- (extract_stats (second package)))
        (call_back add_gp_plot_data sigma l ',stats)))))

      (return (array
            (lambda (x) (predict ((first package) ,x)))
            (lambda () (do
              ;; First collect data for plots, then search
              (collect_stats)
              (gaussian_drift driftsteps_per_iter)))
            search_state_box
            (mh_on_hypers N_MH)
            (lambda () (do
              (call_back dump_gp_plot_data 'drift func_id ',datafile_prefix)
              (best_probe_so_far)))
            (driftstep_counter (* num_iters driftsteps_per_iter))))))]

;;; Helpers
[define keyed_max
   (lambda (arr get_key)
     (let ((keys (mapv get_key arr))
           (i (argmax_of_array keys)))
       (lookup arr i)))]

[define keyed_min
   (lambda (arr get_key)
```

```
    (keyed_max arr (lambda (x)
                     (- 0 (get_key x)))))]

[define min
   (lambda (x y)
     (if (< x y) x y))]

[define max
   (lambda (x y)
     (if (> x y) x y))]

[define confine
   (lambda (x lower upper)
     (min upper (max lower x)))]

[define get1st
   (lambda (arr)
     (lookup arr 0))]

[define get2nd
   (lambda (arr)
     (lookup arr 1))]

[define iid_uniform_sampler
   (lambda (min max)
     (lambda (count)
       (repeatO count
                (lambda () (uniform_continuous min max)))))]

[define repeatO
   (lambda (count func)
     (mapv (lambda (i) (func))
           (sentinels count)))]

[define sentinels
   (lambda (count)
     (linspace 0 0 count))]

[define mc_argmax
   (lambda (func iid_U count)
     (let ((candidate_xs (iid_U count)))
       (keyed_max candidate_xs func)))]

[define joint_to_pointwise
   (lambda (func)
     (lambda (x)
       (get1st (func (array x)))))]

[define contents_changed
   (lambda (box)
     (> (changecount box) 0))]

[define contents
   (lambda (box)
     (run (do
       (return (reset_changecount box))
```

```
       (return (bget box)))))]

[define set_contents bset]

[define peek bget]

[define increment
   (lambda (box)
     (bset box (+ (bget box) 1)))]

[define mark_recently_changed
   (lambda (box)
     ;; do a dummy change, just to have positive changecount
     (bset box (bget box)))]

[define closest_point
   (lambda (needle haystack)
     (keyed_min haystack
                (lambda (x)
                  (abs (- x needle)))))]

[define tail
   (lambda (arr max_tsize)
     (let ((tsize (min (size arr) max_tsize))
           (start_index (- (size arr) tsize)))
       (mapv (lambda (i) (lookup arr (+ start_index i)))
             (linspace 0 (- tsize 1) tsize))))]

[define alist_to_dict
   (lambda (alist)
     (if (= (size alist) 0)
       (dict '() '())
       (apply dict (apply zip alist))))]

[assume zero (make_const_func 0.0)]


;;; Starter functions
[define gp_uniform_main
  (lambda (func_id numprobes datafile_prefix) (do
    (assume func_id ',func_id)
    (assume V (make_audited_expensive_function "V" func_id))
    (apply optimize (run (gp_uniform_search_mh_hypers
                            'V numprobes func_id datafile_prefix)))))]

[define quad_main
  (lambda (data_generator_name numprobes L per_iter_mhsteps initial_state
           datafile_prefix) (do
    (assume D ',(generate_quad_data data_generator_name))
    (assume xs (lookup D 0))
    (assume ys (lookup D 1))
    (assume dct (dict xs ys))
    (assume f (lambda (x) (lookup dct x)))
    (apply optimize (run (quad_exactmax_search_mh_params
                           'f (run (sample xs)) numprobes L
                           per_iter_mhsteps initial_state
```

```
                           data_generator_name datafile_prefix)))))]

[define gp_drift_main
  (lambda (func_id num_iters driftsteps_per_iter propstd s initial_state
           N_avg N_MH datafile_prefix) (do
    (assume func_id ',func_id)
    (assume V (make_audited_expensive_function "V" func_id))
    (apply optimize (run (gp_gaussian_drift_mh_hypers
                           'V num_iters driftsteps_per_iter
                           propstd s initial_state N_avg N_MH
                           func_id datafile_prefix)))))]
```

gpexample_plugin.py

```
import sys
sys.path.append('.')

import numpy as np
from numpy.random import random as rand
import scipy.spatial.distance as spdist

import venture.lite.types as t
import venture.lite.value as v
import venture.lite.sp as sp
from venture.lite.function import VentureFunction
from venture.lite.sp_help import deterministic_typed
import gpmem
import collections
from datetime import datetime
import pickle
import os

BayesOptPlotData = collections.namedtuple('BayesOptPlotData', ['sigma', 'l', 'Xseen', 'Yseen'])

FSD = v.VentureValue.fromStackDict
def getNumber(n):
    return n.getNumber()
def getArray(n):
    return n.getArray()

def unnormalized_bell(x, mean, stdev):
    return np.exp(-(x - mean)**2 / (2 * stdev**2))

# Library of true functions for the GP Bayesian optimization example
F_TRUES = {
        'trimodal': (lambda x: (0.2 + np.exp(-0.1*abs(x-2))) * np.cos(0.4*x)),
        'wide_unimodal': (lambda x: -1 + 2 * np.exp(-(x + 4.7)**2 / (2 * 10**2))),
        'spike': (lambda x: -1 + 2 * float(abs(x-5.2) < 0.01)),
        'bimodal': (lambda x: 1.0 * unnormalized_bell(x, -0.3, 0.5)
                                + 1.2 * unnormalized_bell(x, 3.0, 0.7)),
        }

TRUE_LOCAL_OPTIMA_XS = {
        'trimodal': [
            -5 * np.arctan(4 - np.sqrt(17)),
            -5 * np.arctan(4 - np.sqrt(17)) - 5 * np.pi,
            5 * np.arctan(4 - np.sqrt(17)) + 5 * np.pi,
            ],
        'wide_unimodal': [-4.7],
        'spike': [5.2],
        'bimodal': [2.99988, -0.29997],
        }

def get_ftrue(func_id):
    return F_TRUES[func_id]
def get_true_local_optima(func_id):
    local_optima_xs = TRUE_LOCAL_OPTIMA_XS[func_id]
    f = F_TRUES[func_id]
```

```
    return [(x, f(x)) for x in local_optima_xs]
def get_true_optimum(func_id):
    return max(get_true_local_optima(func_id), key=lambda xy: xy[1])

# Covariance functions
def squared_exponential(sigma, l):
    def f(x1, x2):
        #A = spdist.cdist([[x1/l]],[[x2/l]],'sqeuclidean')
        #ans = sf * np.exp(-0.5*A)[0,0]
        #return ans
        return sigma**2 * np.exp(-(x1-x2)**2 / (2*l))
    return f

covType = sp.SPType([t.NumberType(), t.NumberType()], t.NumberType())

def __venture_start__(ripl, *args):

    # External SPs
    argmaxSP = deterministic_typed(np.argmax, [t.HomogeneousArrayType(t.NumberType())],
            t.NumberType())
    absSP = deterministic_typed(abs, [t.NumberType()], t.NumberType())
    make_se_SP = deterministic_typed(lambda sf, l:
                VentureFunction(squared_exponential(sf, l),
                    name="SE", parameter=[sf,l], sp_type=covType),
            [t.NumberType(), t.NumberType()], t.AnyType("VentureFunction"))
    make_const_func_SP = deterministic_typed(lambda c:
            VentureFunction(lambda x: c,
                sp_type = sp.SPType([], t.NumberType())),
            [t.NumberType()], t.AnyType("VentureFunction"))

    ripl.bind_foreign_sp('allocate_gpmem', gpmem.allocateGPmemSP)
    ripl.bind_foreign_inference_sp('argmax_of_array', argmaxSP)
    ripl.bind_foreign_sp('abs', absSP)
    ripl.bind_foreign_sp('make_squaredexp', make_se_SP)
    ripl.bind_foreign_sp('make_const_func', make_const_func_SP)


    # Gpmem example
    def make_audited_expensive_function(name, id_of_preset):
        f_true = F_TRUES[id_of_preset]
        def expensive_f(x):
            expensive_f.count += 1 # A tracker for how many times I am called
            ans = f_true(x)
            print "[PROBE %s] Probe #%d: %s(%f) = %f" % (
                    expensive_f.name, expensive_f.count, expensive_f.name, x, ans)
            return ans
        expensive_f.count = 0
        expensive_f.name = name
        audited_sp = deterministic_typed(expensive_f, [t.NumberType()], t.NumberType())
        return sp.VentureSPRecord(audited_sp)

    ripl.bind_foreign_sp('make_audited_expensive_function', deterministic_typed(
        make_audited_expensive_function, [t.StringType(), t.SymbolType()],
        sp.SPType([t.NumberType()], t.NumberType())))

    # Accumulator for plot datas
```

```
PLOT_DATAS = []
class AddPlotDataCallback(object):
    def __call__(self, inferrer, sigma_, l_, stats_):
        sigma = FSD(sigma_[0]).getNumber()
        l = FSD(l_[0]).getNumber()
        all_pairs = [map(getNumber, p) for p in map(getArray, FSD(stats_[0]).getArray())]
        (Xseen, Yseen) = zip(*all_pairs) if len(all_pairs) > 0 else ([], [])
        plot_data = BayesOptPlotData(sigma, l, Xseen, Yseen)
        PLOT_DATAS.append(plot_data)

class DumpPlotDataCallback(object):
    def __call__(self, inferrer, strategy_name_, func_id_, user_prefix_):
        strategy_name = FSD(strategy_name_[0]).getSymbol()
        func_id = FSD(func_id_[0]).getSymbol()
        user_prefix = FSD(user_prefix_[0]).getSymbol()
        assert isinstance(func_id, str)
        date_fmt = '%Y%m%d_%H%M%S'
        directory = 'bayesopt_output'
        def j(fname):
            return os.path.join(directory, fname)
        log_fname = 'plot_data_%s_%s_%s_%s.pkl' % (
                func_id,
                strategy_name,
                user_prefix,
                datetime.now().strftime(date_fmt),)
        recents_fname = 'MOST_RECENT'
        print "Logging to %s" % (j(log_fname),)
        with open(j(log_fname), 'wb') as f:
            pickle.dump([func_id, PLOT_DATAS], f)
        with open(j(recents_fname), 'wb') as f:
            print >> f, log_fname
        print "Done."

ripl.bind_callback("add_gp_plot_data", AddPlotDataCallback())
ripl.bind_callback("dump_gp_plot_data", DumpPlotDataCallback())
```

gpmem.py

```
from venture.lite.psp import DeterministicPSP
from venture.lite.sp import SP, VentureSPRecord
from venture.lite.lkernel import SimulationAAALKernel
from venture.lite.env import VentureEnvironment
from venture.lite.request import Request,ESR
from venture.lite.address import emptyAddress
from venture.lite.sp_help import no_request
import venture.lite.types as t
from venture.lite.gp import GPSP, GPSPAux
import collections

class MakeMakeGPMSPOutputPSP(DeterministicPSP):
    def simulate(self, args):
        return VentureSPRecord(no_request(MakeGPMSPOutputPSP()))

class MakeGPMSPOutputPSP(DeterministicPSP):
    """
    This class (except its simulate method) is based on DeterministicAAAMakerPSP.
    """
    def __init__(self):
        self.shared_aux = GPSPAux(collections.OrderedDict())

    def simulate(self, args):
        f_node = args.operandNodes[0]
        prior_mean_function = args.operandValues()[1]
        prior_covariance_function = args.operandValues()[2]
        f_compute = VentureSPRecord(
                SP(GPMComputerRequestPSP(f_node), GPMComputerOutputPSP()))
        f_emu = VentureSPRecord(GPSP(prior_mean_function, prior_covariance_function))
        f_emu.spAux = self.shared_aux
        f_compute.spAux = self.shared_aux
        return t.pythonListToVentureList([f_compute, f_emu])

    def childrenCanAAA(self): return True
    def getAAALKernel(self): return GPMDeterministicMakerAAALKernel(self)

class GPMDeterministicMakerAAALKernel(SimulationAAALKernel):
    """
    This is based on DeterministicMakerAAALKernel.
    """
    def __init__(self,makerPSP): self.makerPSP = makerPSP
    def simulate(self, _trace, args):
        return self.makerPSP.simulate(args)
    def weight(self, _trace, newValue, _args):
        (_f_compute, f_emu) = newValue.asPythonList()
        answer = f_emu.sp.outputPSP.logDensityOfCounts(self.makerPSP.shared_aux)
        # print "gpmem LKernel weight = %s" % answer
        return answer

allocateGPmemSP = no_request(MakeMakeGPMSPOutputPSP())

class GPMComputerRequestPSP(DeterministicPSP):
    def __init__(self, f_node):
        self.f_node = f_node
```

```
    def simulate(self, args):
        id = str(args.operandValues())
        exp = ["gpmemmedSP"] + [["quote",val] for val in args.operandValues()]
        env = VentureEnvironment(None,["gpmemmedSP"],[self.f_node])
        return Request([ESR(id,exp,emptyAddress,env)])

# TODO Perhaps this could subclass ESRRefOutputPSP to correctly handle
# back-propagation?
class GPMComputerOutputPSP(DeterministicPSP):
    def simulate(self,args):
        assert len(args.esrNodes()) ==  1
        return args.esrValues()[0]

    def incorporate(self, value, args):
        x = args.operandValues()[0].getNumber()
        y = value.getNumber()
        args.spaux().samples[x] = y

    def unincorporate(self, value, args):
        x = args.operandValues()[0].getNumber()
        samples = args.spaux().samples
        if x in samples:
            del samples[x]
```

quadmem.py

```
from venture.lite.psp import DeterministicPSP, RandomPSP, TypedPSP, NullRequestPSP
from venture.lite.sp import SP, VentureSPRecord, SPType, SPAux
from venture.lite.lkernel import SimulationAAALKernel
from venture.lite.env import VentureEnvironment
from venture.lite.request import Request,ESR
from venture.lite.address import emptyAddress
from venture.lite.sp_help import no_request, typed_nr
import venture.lite.types as t
import venture.lite.value as v
import numpy as np
from SortedCollection import SortedCollection
import copy

class QuadOutputPSP(RandomPSP):
    def __init__(self, a, b, c, noisestd, L):
        self.a = a
        self.b = b
        self.c = c
        self.noisestd = noisestd
        self.L = L
        assert isinstance(self.L, int)

    def get_noiseless(self):
        def f(xs):
            return -self.a * (xs - self.b)**2 + self.c
        return f

    def simulate(self, args):
        xs = np.array(args.operandValues()[0])
        pris = np.array(args.operandValues()[1])
        assert len(xs) == len(pris)
        f = self.get_noiseless()
        os = f(xs) + np.random.normal(0, scale=self.noisestd, size=xs.shape)
        return os

    def logDensity(self, os, args):
        raise Exception("This should never be called")

    def logDensityOfCounts(self, aux):
        fresh_samples = aux.samples[-self.L:]
        N = len(fresh_samples)
        f_true = self.get_noiseless()
        tot_sqerr = sum((y - f_true(x))**2 for (pri, x, y) in fresh_samples)
        ans = (-N * (np.log(self.noisestd) + 0.5*np.log(2*np.pi)) -
            tot_sqerr / (2.0 * self.noisestd**2))
        # print "LDOC = ", ans
        return ans

    def incorporate(self, os, args):
        samples = args.spaux().samples
        xs = args.operandValues()[0]
        pris = args.operandValues()[1]

        for pri, x, o in zip(pris, xs, os):
```

```
            samples.insert((pri, x, o))

    def unincorporate(self, os, args):
        samples = args.spaux().samples
        xs = args.operandValues()[0]
        pris = args.operandValues()[1]
        for pri, x, o in zip(pris, xs, os):
            samples.remove((pri, x, o))

quadType = SPType(2*[t.ArrayUnboxedType(t.NumberType())], t.ArrayUnboxedType(t.NumberType()))

class QuadSPAux(SPAux):
    def __init__(self, samples):
        self.samples = samples
    def copy(self):
        return QuadSPAux(copy.copy(self.samples))
    def asVentureValue(self):
        def encode(xy):
            # (x,y) = xy
            return v.VentureArray(map(v.VentureNumber, xy))
        return v.VentureArray([encode((x,y)) for (pri,x,y) in self.samples])

class QuadSP(SP):
    def __init__(self, a, b, c, noisestd, L):
        self.a = a
        self.b = b
        self.c = c
        self.noisestd = noisestd
        self.L = L
        assert isinstance(self.L, int)
        output = TypedPSP(QuadOutputPSP(a, b, c, noisestd, L), quadType)
        super(QuadSP, self).__init__(NullRequestPSP(), output)

    def constructSPAux(self): return QuadSPAux(SortedCollection())

class MakeMakeQuadMSPOutputPSP(DeterministicPSP):
    def simulate(self, args):
        return VentureSPRecord(typed_nr(MakeQuadMSPOutputPSP(),
            [t.AnyType()] + 5*[t.NumberType()], t.AnyType()))

class MakeQuadMSPOutputPSP(DeterministicPSP):
    """"
    This class (except its simulate method) is based on DeterministicAAAMakerPSP.
    """
    def __init__(self):
        # We need to keep the aux ourselves, because proposals to the arguments
        # of this function will cause f_compute and f_emu to be re-created,
        # with initially blank auxs that we need to replace with our stored
        # aux.
        self.shared_aux = QuadSPAux(SortedCollection())

    def simulate(self, args):
        assert len(args.operandValues()) == 6
        f_node = args.operandNodes[0]
        a, b, c, noisestd, L_ = args.operandValues()[1:6]
        L = int(L_)
```

```
        f_compute = VentureSPRecord(
                SP(QuadMComputerRequestPSP(f_node), QuadMComputerOutputPSP()))
        f_emu = VentureSPRecord(QuadSP(a, b, c, noisestd, L))
        f_emu.spAux = self.shared_aux
        f_compute.spAux = self.shared_aux
        return t.pythonListToVentureList([f_compute, f_emu])

    def childrenCanAAA(self): return True
    def getAAALKernel(self): return QuadMDeterministicMakerAAALKernel(self)

class QuadMDeterministicMakerAAALKernel(SimulationAAALKernel):
    """
    This is based on DeterministicMakerAAALKernel.
    """
    def __init__(self,makerPSP): self.makerPSP = makerPSP
    def simulate(self, _trace, args):
        return self.makerPSP.simulate(args)

    def weight(self, _trace, newValue, _args):
        # Using newValue.spAux here because args.madeSPAux is liable to be
        # None when detaching. This has something to do with when the Args
        # object is constructed relative to other things that happen
        # during detach/regen. TODO: fix it so that this code is less
        # fragile.

        (_f_compute, f_emu) = newValue.asPythonList()
        answer = f_emu.sp.outputPSP.logDensityOfCounts(self.makerPSP.shared_aux)
        # print "quadmem LKernel weight = %s" % answer
        return answer

allocateQuadmemSP = no_request(MakeMakeQuadMSPOutputPSP())

class QuadMComputerRequestPSP(DeterministicPSP):
    def __init__(self, f_node):
        self.f_node = f_node

    def simulate(self, args):
        id = str(args.operandValues())
        x = args.operandValues()[0]
        pri = args.operandValues()[1]
        forwarded_args = [x]
        exp = ["quadmemmedSP"] + [["quote",val] for val in forwarded_args]
        env = VentureEnvironment(None,["quadmemmedSP"],[self.f_node])
        return Request([ESR(id,exp,emptyAddress,env)])

# TODO Perhaps this could subclass ESRRefOutputPSP to correctly handle
# back-propagation?
class QuadMComputerOutputPSP(DeterministicPSP):
    def simulate(self,args):
        assert len(args.esrNodes()) ==  1
        return args.esrValues()[0]

    def incorporate(self, value, args):
        x = args.operandValues()[0].getNumber()
        priority = args.operandValues()[1].getNumber()
        y = value.getNumber()
```

```
        samples = args.spaux().samples
        samples.insert((priority, x, y))

    def unincorporate(self, value, args):
        x = args.operandValues()[0].getNumber()
        priority = args.operandValues()[1].getNumber()
        y = value.getNumber()
        sample = (priority, x, y)
        samples = args.spaux().samples
        samples.remove(sample)
```