laGP: Large-Scale Spatial Modeling via Local Approximate Gaussian Processes in R

Robert B. Gramacy
The University of Chicago
Booth School of Business

Abstract

Gaussian process (GP) regression models make for powerful predictors in out of sample exercises, but cubic runtimes for dense matrix decompositions severely limits the size of data—training and testing—on which they can be deployed. That means that in computer experiment, spatial/geo-physical, and machine learning contexts, GPs no longer enjoy privledged status as modern data sets continue balloon in size. We discuss an implementation of local approximate Gaussian process models, in the **laGP** package for R, which offers a particular sparse-matrix remedy uniquely positioned to leverage modern parallel computing architectures. The **laGP** approach can be seen as an update on the spatial statistical method of local *kriging* neighborhoods. We briefly review the method, and provide extensive illustrations of the features in the package through worked-code examples. The appendix covers custom building options, for symmetric multi-processor and graphical processing unit support which are not enabled by default, and the built-in wrapper routines that automate distribution over a simple network of workstations.

Keywords:˜sequential design, active learning, surrogate/emulator, calibration, local kriging, symmetric multi-processor, graphical processing unit, cluster computing, big data.

1. Gaussian process regression and sparse approximation

Gaussian process (GP) regression models, sometimes called a Gaussian spatial process (GaSP), have been popular for decades in spatial data contexts like geostatistics (e.g., Cressie 1993) where they are known as *kriging* (Matheron 1963), and in computer experiments where they are deployed as *surrogate models* or *emulators* (Sacks, Welch, Mitchell, and Wynn 1989; Santner, Williams, and Notz 2003). More recently, they have become a popular prediction engine in the machine learning literature (Rasmussen and Williams 2006). The reasons are many, but the most important are probably that the Gaussian structure affords a large degree of analytic capability not enjoyed by other general-purpose approaches to nonpatrametric nonlinear modeling, and because they perform well in out-of-sample tests. They are not, however, without their drawbacks, and two important ones are computational tractability and nonstationary flexibility, which we shall return to shortly.

A GP is technically a prior over functions (Stein 1999), with finite dimensional distributions defined by a mean $\mu(x)$ and positive definite covariance $\Sigma(x, x')$, for p-dimensional input(s) x and x'. For N input x-values this defines a μ_N N-vector and Σ_N positive definite $N \times N$ matrix whereby the output is a random N-vector $Y_N \sim \mathcal{N}_N(\mu_N, \Sigma_N)$. However, for regression

applications a likelihood perspective provides a more direct view of the relevant quantities for inference and prediction. In that setup, N data (training) pairs $D_N = (X_N, Y_N)$, comprised of $N \times p$-dimensional design X_N define a multivariate normal (MVN) likelihood for an N-vector of scalar responses Y_N through a small number of parameters θ that describe how X_N is related to μ_N and Σ_N. Linear regression is a special case where $\theta = (\beta, \tau^2)$ and $\mu_N = X_N\beta$ and $\Sigma_N = \tau^2 I_N$.

Whereas the linear case puts most of the "modeling" structure in the mean, GP regression focuses more squarely on the covariance structure. In many computer experiments contexts the mean is taken to be zero (e.g., Santner *et˜al.* 2003), which is the simplifying assumption we shall make throughout, although it is easy to generalize to a mean described by a polynomial basis. Let $K_\theta(x, x')$ be a correlation function so that $Y_N \sim \mathcal{N}_N(0, \tau^2 K_N)$ where K_N is a $N \times N$ positive definite matrix comprised of entries $K_\theta(x_i, x_j)$ from the rows of X_N. Here we are changing the notation slightly so that θ is reserved explicity for K_θ, isolating τ^2 as a separate scale parameter. Choices of $K_\theta(\cdot, \cdot)$ determine stationarity, smoothness, differentiability, etc., but most importantly they determine the decay of spatial correlation.

A common first choice is the so-called *isotropic Gaussian*: $K_\theta(x, x') = \exp\{-\sum_{k=1}^{p}(x_k - x'_k)^2/\theta\}$, where correlation decays exponentially fast at rate θ. Since $K_\theta(x, x) = 1$ the resulting regression function is an interpolator, which is appropriate for many deterministic computer experiments. For smoothing noisy data, or for a more robust to modeling computer experiments (Gramacy and Lee 2011), a *nugget* can be added to $K_{\theta,\eta}(x, x') = K_\theta(x, x') + \eta \mathbb{I}_{\{x=x'\}}$. Much of the technical work described below, and particularly in Section 2, is generic to the particular choice of $K(\cdot, \cdot)$, excepting that it be differentiable in all parameters. Although the **laGP** package favors the isotropic Gaussian case, we argue that many of the drawbacks of that overly simplistic choice, leading theorists and practitioners alike to prefer other choices like the Matérn (Stein 1999), are less of a concern in our particular approach to sparse approximation. The package also provides a limited set of routines which can accommodate a separable Gaussian correlation function; more details are provided in Section 3.2. Our empirical work will contain examples where correlation parameters (θ, η) are *both* estimated from data, however, we emphasize cases where η is fixed at a small value which is typical for numerically robust near-interpolation of computer experiments.

1.1. Inference and prediction

GP regression is popular because inference (for all parameters but particularly for θ) is easy, and (out-of-sample) prediction is highly accurate and conditionally (on θ and η) analytic. It the spatial and computer experiments literatures it has become convention to deploy a reference $\pi(\tau^2) \propto 1/\tau^2$ prior (Berger, De˜Oliveira, and Sanso 2001) and obtain a marginal likelihood for the remaining unknowns.

$$p(Y_N | K_\theta(\cdot, \cdot)) = \frac{\Gamma[N/2]}{(2\pi)^{N/2}|K_N|^{1/2}} \times \left(\frac{\psi_N}{2}\right)^{-\frac{N}{2}} \quad \text{where} \quad \psi_N = Y_N^\top K_N^{-1} Y_N \tag{1}$$

Derivatives are available analytically, leading to fast Newton-like schemes for maximizing. Some complications can arise when the likelihood is multi-modal for θ, however, where fully Bayesian inference may be preferred (e.g., Rasmussen and Williams 2006, Chapter 5).[1]

[1]Eq.˜(1) emphasizes $K_\theta(\cdot, \cdot)$, dropping η to streamline the notation in the following discussion. Everything applies to $K_{\theta,\eta}(\cdot, \cdot)$ as well.

The predictive distribution $p(y(x)|D_N, K_\theta(\cdot,\cdot))$, is Student-$t$ with degrees of freedom N,

$$\text{mean} \qquad \mu(x|D_N, K_\theta(\cdot,\cdot)) = k_N^\top(x) K_N^{-1} Y_N, \tag{2}$$

$$\text{and scale} \qquad \sigma^2(x|D_N, K(\cdot,\cdot)) = \frac{\psi_N[K_\theta(x,x) - k_N^\top(x) K_N^{-1} k_N(x)]}{N}, \tag{3}$$

where $k_N^\top(x)$ is the N-vector whose i^{th} component is $K_\theta(x, x_i)$. Using properties of the Student-t, the variance of $Y(x)$ is $V_N(x) \equiv \mathbb{V}\text{ar}[Y(x)|D_N, K_\theta(\cdot,\cdot)] = \sigma^2(x|D_N, K_\theta(\cdot,\cdot)) \times N/(N-2)$.

As an example illustrating both inference and prediction, consider a simple sinusoidal "data set" treated as a deterministic computer simulation, i.e., modeled without noise.

```
R> X <- matrix(seq(0,2*pi,length=6), ncol=1)
R> Z <- sin(X)
```

The code below uses some low-level routines in the package to initialze a GP representation with $\theta = 2$ and $\eta = 10^{-6}$. Then, a derivative-based MLE sub-routine is used to find $\hat{\theta}_{N=7}$, maximizing (1).

```
R> gp <- newGP(X, Z, 2, 1e-6, dK=TRUE)
R> mleGP(gp, tmax=20)

$d
[1] 4.386202

$its
[1] 6
```

The output printed to the screen shows the inferred $\hat{\theta}_N$ value, called `d` in the package, and the number of Newton iterations required. The `mleGP` command alters the stored GP object (`gp`) to contain the new representation of the GP using $\hat{\theta}_{N=7}$. Now, the code below obtains the parameters of the predictive equations on a grid of new x-values `XX`, following Eqs.˜(2–3).

```
R> XX <- matrix(seq(-1,2*pi+1, length=499), ncol=ncol(X))
R> p <- predGP(gp, XX)
R> deleteGP(gp)
```

The last line, above, frees the internal representation of the GP object, as we no longer need it to complete this example. The moments stored in `p` can be used to plot mean predictions and generate sample predictive paths, as follows.

```
R> library(mvtnorm)
R> N <- 100
R> ZZ <- rmvt(N, p$Sigma, p$df)
R> ZZ <- ZZ + t(matrix(rep(p$mean, N), ncol=N))
```

Finally, Figure 1 provides a visualization of those sample paths on a scatter plot of the data. Each gray line, plotted by `matplot`, is a single random realization of $Y(x)|D_N, \hat{\theta}_N$. Obseve

```
R> matplot(XX, t(ZZ), col="gray", lwd=0.5, lty=1, type="l",
+         bty="n", main="simple sinusoidal example", xlab="x",
+         ylab="Y(x) | theta-hat")
R> points(X, Z, pch=19)
```

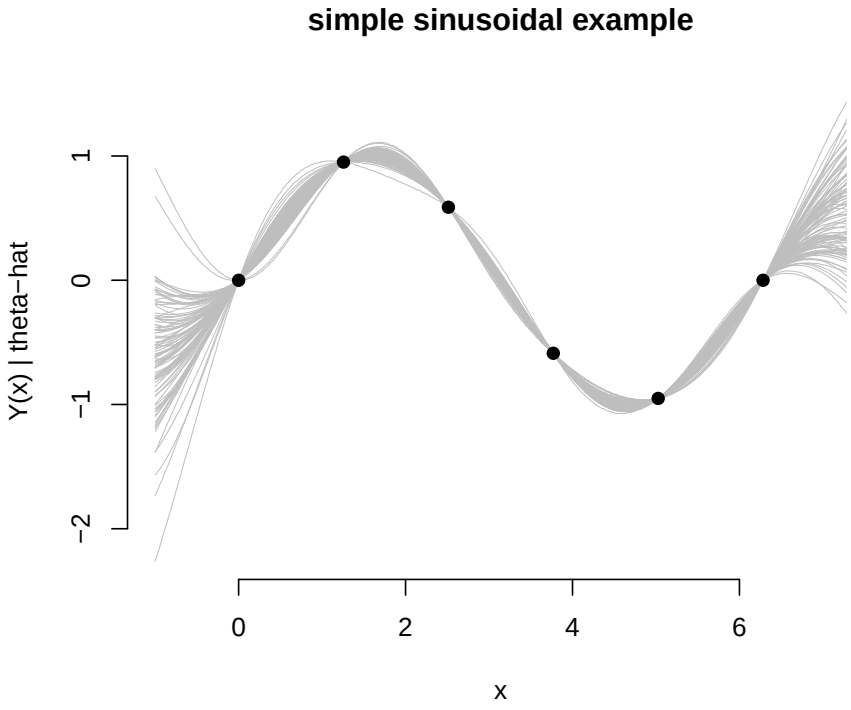

Figure 1: Predictions from fitted GP regression model on simple sinusoidal data.

how the predictive variance narrows for x nearby elements of X_N and expands out in a "football shape" away from them. This feature has attractive uses in design: high variance inputs represent sensible choices for new simulations (Gramacy and Lee 2009).

1.2. Supercomputing and sparse approximation for big data

Despite its many attractive features, GP regression implementations are buckling under the weight of the growing size of data sets in many modern applications. For example, supercomputers make submitting one job as easy as thousands, leading to ever larger computer simulation data. The problem is $O(N^3)$ matrix decompositions required to calculate K_N^{-1} and $|K_N|$ in Eqs.˜(1–3). In practice that limits N to the low thousands for point inference, and high hundreds for sampling-based inference like the bootstrap or Bayesian MCMC. This has pushed some practitioners towards wholly new modeling apparatuses, say via trees (Pratola, Chipman, Gattiker, Higdon, McCulloch, and Rust 2013; Gramacy, Taddy, and Wild 2013; Chipman, Ranjan, and Wang 2012). Although trees offer an appealing divide-and-conquer approach, their obvious drawback is that they struggle to capture the smoothness known, in many cases, to exist in the physical and mathematical quantities being modeled.

One approach to salvaging GP inference for use in larger contexts has been to allocate supercomputer resources. Franey, Ranjan, and Chipman (2012) were the first to use graphical processing unit (GPU) linear algebra subroutines, extending the N which could be accommodated by an order of magnitude. Paciorek, Lipshitz, Zhuo, Prabhat, Kaufman, and Thomas (2013) developed a package for R called **bigGP** that combined symmetric-multiprocessor, cluster, and GPU facilities to gain yet another order of magnitude. Paciorek *et˜al.* were able to handle $N = 67275$. To go too far down that road, however, may miss the point in certain contexts. Computer model emulation is meant to *avoid* expensive computer simulation, not be a primary consumer of it.

An orthogonal approach is to perform approximate GP regression, and a common theme in that literature is sparsity, leading to fast matrix decompositions (e.g., Kaufman, Bingham, Habib, Heitmann, and Frieman 2012; Sang and Huang 2012). Again, the expansion of capability is one- to two-orders of magnitude, albeit without tapping supercomputer resources which is more practical for most applications. For example, Kaufman *et˜al.* reported on an experiment with $N = 20000$. Some approaches in a similar vein include fixed rank kriging (Cressie and Johannesson 2008) and using ‘‘pseudo-inputs” (Snelson and Ghahramani 2006).

Hybrid approximate GP regression and big-computer resources have been combined to push the boundary even farther. Eidsvik, Shaby, Reich, Wheeler, and Niemi (2014) suggest composite likelihood approach, rather than directly leveraging spare matrix library, and when combined with a GPU implementation their method is able to cope with $N = 173405$. This represents a substantial inroad into retaining many of the attractive features of GP regression in larger data applications. However, a larger (and thriftier) capability would certainly be welcome. Pratola *et˜al.* (2013) found it necessary to modify a tree-based approach for distribution over the nodes of a supercomputer to handle an $N = 7$M sized design.

The remainder of the paper is outlined as follows. In Section 2 we discuss the local approximate Gaussian process method for large scale inference and prediction. Several variations are discussed, including parallelized and GPU versions for combining with supercomputing resources in order to order to handle large-N problems in reasonable computation times (e.g., under an hour). Live-code examples, demonstrating the features of the **laGP** package for R, are peppered throughout paper, however Sections 3 and 4 devoted to larger scale and more exhaustive demonstration of the features in action, first demonstrating local emulation and regression/smoothing and then adapting them to the important problem of large scale computer model calibration. Section 5 discusses extra features of the package which are useful in other contexts, like the optimization of blackbox functions, and the potential for customization and extension of existing subroutines in the package. The appendix describes how the package can be compiled to enable SMP and GPU support, and discusses a variation a the key wrapper function `aGP`, which enables distribution of predictions across the nodes of a cluster.

2. Local approximate Gaussian process models

The methods in the **laGP** package take a two-pronged approach to large data GP regression. They utilize a sparse representation, but in fact only work with small dense matrices. And the many-independent nature of the calculations required are amenable to massive parallelization. The result is an approximate GP regression capability that can accommodate orders of magnitude larger training and testing sets than ever before. The method can be seen as a

modernization of *local kriging* from the spatial statistics literature (Cressie 1993, pp,131–134). It involves focusing on approximate predictive equations at a particular generic location, x, via a subset of the data $D_n(x) \subseteq D_N$, where the sub-design $X_n(x)$ is (primarily) comprised of X_N close to x. The thinking is that, with the typical choices of $K_\theta(x, x')$, where correlation between elements $x' \in X_N$ decays quickly for x' far from x, remote x's have vanishingly small influence on prediction anyways. Ignoring them in order to work with much smaller, n-sized, matrices will bring a big computational savings with little impact on accuracy.

This is a sensible idea: It can be shown to induce a valid stochastic process; when $n \ll 1000$ the method is fast and accurate, and as n grows the predictions increasingly resemble their full N-data counterparts; and for smaller n $V_n(x)$ is organically inflated relative to $V_N(x)$, acknowledging greater uncertainty in approximation. The simplest version of such a scheme would be via nearest neighbors (NN): $X_n(x)$ comprised of closest elements of X_N to x. Emory (2009) showed that this works well for many common choices of K_θ. However, NN designs are known to be sub-optimal (Vecchia 1988; Stein, Chi, and Welty 2004) as it pays to have some spread in $X_n(x)$ in order to obtain good estimates of correlation hyperparameters like θ. Still, searching for the optimal sub-design, which involves choosing n from N things, is a combinatorially huge undertaking.

Gramacy and Apley (2014) showed how a greedy search could provide designs $X_n(x)$ where predictors based on $D_n(x)$ out-performed the NN alternative out-of-sample, yet required no more computational effort than NN, i.e., they worked in $O(n^3)$ time. The idea is to search iteratively, starting with a small NN set $D_{n_0}(x)$, and choosing x_{j+1} to augment $X_j(x)$ to form $D_{j+1}(x)$ according to one of several simple objective criteria. Importantly, they showed that the criteria they chose, on which we elaborate below, along with the the other relevant GP quantities for inference and prediction (1–3) can be updated as $j \to j+1$ in $O(j^2)$ time as long as the parameters, θ, remain constant across iterations. Therefore over the entirety of those iterations, $j = n_0, \ldots, n$, the scheme is in $O(n^3)$. The idea of sequential updating for GP inference, whether for design or for speed, is not new (Gramacy and Polson 2011; Haaland and Qian 2011), however the focus of previous approaches has been global. Working local to particular x brings both computational and modeling/accuracy advantages.

2.1. Criterion for local design

Gramacy and Apley considered two criteria in addition to NN, one being a special case of the other. The first is to minimize the empirical Bayes mean-square prediction error (MSPE)

$$J(x_{j+1}, x) = \mathbb{E}\{[Y(x) - \mu_{j+1}(x|D_{j+1}, \hat{\theta}_{j+1})]^2 | D_j(x)\}$$

where $\hat{\theta}_{j+1}$ is the estimate for θ based on D_{j+1}. The predictive mean $\mu_{j+1}(x|D_{j+1}, \hat{\theta}_{j+1})$ follows equation (2), except that the $j+1$ subscript has been added in order to indicate dependence on x_{j+1} and the future, unknown y_{j+1}. They then derive the approximation

$$J(x_{j+1}, x) \approx V_j(x|x_{j+1}; \hat{\theta}_j) + \left(\frac{\partial \mu_j(x;\theta)}{\partial \theta}\Big|_{\theta=\hat{\theta}_j} \right)^2 \Big/ \mathcal{G}_{j+1}(\hat{\theta}_j). \tag{4}$$

The first term in (4) estimates predictive variance at x after x_{j+1} is added into the design,

$$V_j(x|x_{j+1};\theta) = \frac{(j+1)\psi_j}{j(j-1)} v_{j+1}(x;\theta),$$
$$\text{where} \quad v_{j+1}(x;\theta) = \left[K_{j+1}(x,x) - k_{j+1}^\top(x) K_{j+1}^{-1} k_{j+1}(x)\right]. \tag{5}$$

Minimizing predictive variance at x is a sensible goal. The second term in (4) estimates the rate of change of the predictive mean at x, weighted by the expected *future* inverse information, $\mathcal{G}_{j+1}(\hat{\theta}_j)$, after x_{j+1} and the corresponding y_{j+1} are added into the design. The weight, which is constant in x comments on the value of x_{j+1} for estimating the parameter of the correlation function, θ, by controlling the influence of the rate of change (derivative) of the predictive mean at x on the overall criteria.

The influence of that extra term beyond the reduced variance is small. The full MSPE criteria tends to yield qualitatively similar local designs $X_n(x)$ as ones obtained using just $V_j(x|x_{j+1};\hat{\theta}_j)$, but at a fraction of the computational cost (since no derivative calculations are necessary). This simplified criteria is equivalent to choosing x_{j+1} to maximize *reduction* in variance:

$$\begin{aligned} &v_j(x;\theta) - v_{j+1}(x;\theta) \\ &\quad = k_j^\top(x) G_j(x_{j+1}) m_j^{-1}(x_{j+1}) k_j(x) + 2k_j^\top(x) g_j(x_{j+1}) K(x_{j+1},x) + K(x_{j+1},x)^2 m_j(x_{j+1}), \end{aligned} \tag{6}$$

where $G_j(x') \equiv g_j(x') g_j^\top(x')$,

$$g_j(x') = -m_j(x') K_j^{-1} k_j(x') \quad \text{and} \quad m_j^{-1}(x') = K_j(x',x') - k_j^\top(x') K_j^{-1} k_j(x'). \tag{7}$$

Observe that only $O(j^2)$ caluclations are required above. Although known for some time in other contexts, Gramacy and Apley chose the acronym ALC to denote the use of that decomposition in local design in order to recognize a similar approach to *global* design via a method called *active learning Cohn* (1996).

To illustrate local designs derived under greedy application of both criteria, consider the following gridded design in $[-2,2]^2$.

```
R> x <- seq(-2, 2, by=0.02)
R> X <- as.matrix(expand.grid(x, x))
R> N <- nrow(X)
```

Here we have $N = 40401$, a very large design by traditional GP standards. You cannot invert an $N \times N$ matrix for N that big on even the best modern workstation. As a point of reference, it takes about seven seconds to perform a single decomposition of an 4000×4000 matrix using hyperthreaded libraries on a 2010 iMac.

The `laGP` function requires a vector of responses to perform local design, even though the design itself doesn't directly depend on the responses—a point which we will discuss at greater length shortly. The synthetic response Gramacy and Apley used for illustrations is coded below, and we shall elucidate that nature of input/output relationships in due course.

```
R> f2d <- function(x)
+   {
```

```
+     g <- function(z)
+       return(exp(-(z-1)^2) + exp(-0.8*(z+1)^2) - 0.05*sin(8*(z+0.1)))
+     -g(x[,1])*g(x[,2])
+   }
R> Y <- f2d(X)
```

Now, consider a prediction location x, denoted by `Xref` in the code below, and local designs for prediction at that x based on MSPE and ALC criteria.

```
R> Xref <- matrix(c(-1.725, 1.725), nrow=TRUE)
R> p.mspe <- laGP(Xref, 6, 50, X, Y, d=0.1, method="mspe")
R> p.alc <- laGP(Xref, 6, 50, X, Y, d=0.1, method="alc")
```

Both designs use $n_0 = 6$ nearest neigbors to start, make greedy selections until $n = 50$ locations are chosen, and use $\theta = 0.1$. The output object from `laGP` contains indices into

```
R> plot(X[p.mspe$Xi,], xlab="x1", ylab="x2", type="n",
+   main="comparing local designs")
R> text(X[p.mspe$Xi,], labels=1:length(p.mspe$Xi), cex=0.7)
R> text(X[p.alc$Xi,], labels=1:length(p.alc$Xi), cex=0.7, col=2)
R> points(Xref[1], Xref[2], pch=19, col=3)
R> legend("topright", c("mspe", "alc"), text.col=c(1,2), bty="n")
```

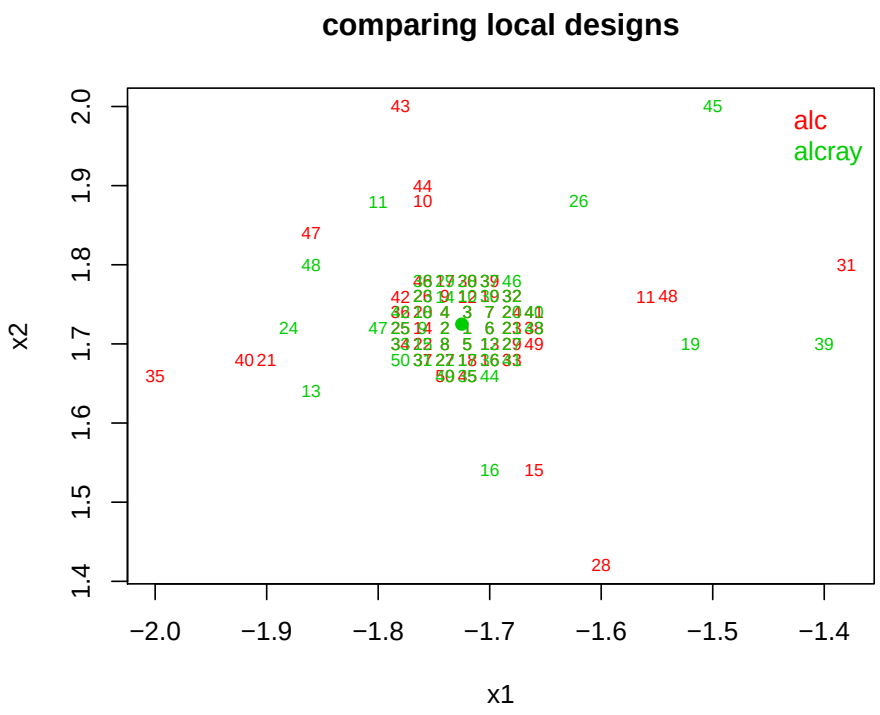

Figure 2: Local designs at x (green dot), derived under MSPE and ALC criteria.

the original design. Those locations, and the order in which they were chosen, are plotted

in Figure 2. They are not identical under the two criteria, but any qualitative differences are subtle. Both contain a clump of nearby points with satellite points emanating along rays from x, the green dot. The satellite points are still relatively close to x considering the full scope of locations in X_N—all locations chosen are in the upper-left quadrant of the space.

It is perhaps intriguing that the greedy local designs differ from NN ones. An exponentially decaying $K_\theta(\cdot,\cdot)$, like our isotropic Gaussian choice, should substantially devalue locations far from x. Gramacy and Haaland (2014) offer an explanation, which surprisingly has little to do with the particular choice of K_θ. Consider two potential locations, an x_{j+1} close to x and x'_{j+1} farther away. One naturally wonders: how could the latter choice, x'_{j+1} with y'_{j+1}-value exponentially less correlated with $y(x)$ than y_{j+1} via x_{j+1}, be preferred over the closer x_{j+1} choice? The answer lies the form of (7). Although quadratic in $K_\theta(x_{j+1}, x)$, the "distance" between the x and the potential new local design location x_{j+1}, it is also quadratic in $g_j(x_{j+1})$, a vector measuring "inverse distance", via K_j^{-1}, between x_{j+1} and the current local design $X_j(x)$. So the criteria makes a tradeoff: minimize "distance" to x while maximizing "distance" (or minimizing "inverse distance") to the existing design. Or in other words, the potential value of new design element (x_{j+1}, y_{j+1}) depends not just on its proximity to x, but also on how potentially different that information is to where we already have (lots of) it, at $X_j(x)$.

Returning to the code example, we see below that the predictive equations are also very similar under both local designs.

```
R> p <- rbind(c(p.mspe$mean, p.mspe$s2, p.mspe$df),
+   c(p.alc$mean, p.alc$s2, p.alc$df))
R> colnames(p) <- c("mean", "s2", "df")
R> rownames(p) <- c("mspe", "alc")
R> p
```

```
           mean           s2 df
mspe -0.3723017 1.674662e-05 50
alc  -0.3722813 1.503320e-05 50
```

Although the designs are built using a fixed $\theta = 0.1$, the predictive equations output at the end are derived based on a local MLE calculation given the data $D_n(x)$.

```
R> p.mspe$mle
```

```
          d dits
1 0.2509476    6
```

```
R> p.alc$mle
```

```
          d dits
1 0.2530758    6
```

MLE calculations can be turned off by adjusting the `laGP` call to include `d=list(start=0.1, mle=FALSE)` as an argument. More about local inference for θ is deferred until Section 2.2. For now we note that the implementation is same as the one behind the `mleGP` routine described earlier in Section 1.1.

Finally, both local design methods are fast,

```
R> c(p.mspe$time, p.alc$time)

elapsed elapsed
  0.230   0.112
```

though ALC is about 2.1 times faster since it doesn't require evaluation of derivatives, etc. Although a more thorough out of sample comparison on both time and accuracy fronts is left to Section 3, the factor of (at least) two speedup in execution time, together with the simpler implementation, led Gramacy and Apley to prefer ALC in most cases.

2.2. Global inference, prediction and parallelization

The simplest way to extend the analysis to cover a dense design of predictive locations $x \in \mathcal{X}$ is to serialize: loop over each x collecting approximate predictive equations, each in $O(n^3)$ time. For $T = |\mathcal{X}|$ the total computational time is in $O(Tn^3)$. Obtaining each of the full GP sets of predictive equations, by contrast, would require computational time in $O(TN^2 + N^3)$, where the latter N^3 is attributable to obtaining K^{-1}.[2] One of the nice features of standard GP emulation is that once K^{-1} has been obtained the computations are fast $O(N^2)$ operations for each location x. However, as long as $n \ll N$ our approximate method is even faster despite having to rebuild and re-decompose $K_j(x)$'s for each x.

The approximation at x is built up sequentially, but completely independently of other predictive locations. Since a high degree of local spatial correlation is a key modeling assumption this may seem like an inefficient use of computational resources, and indeed it would be in serial computation for each x. However, independence allows trivial parallelization requiring token programmer effort. When compiled correctly [see Appendix A.1] the **laGP** package can exploit symmetric multiprocessor (SMP) via `OpenMP` pragmas in its underlying `C` implementation. The simplest way this is accomplished is via a "parallel-for" pragma.

```
#ifdef _OPENMP
  #pragma omp parallel for private(i)
#endif
  for(i=0; i<npred; i++) { ...
```

That is actual code from an early implementation, where `npred` $= |\mathcal{X}|$, leading to a nearly linear speedup: runtimes for P processors scale roughly as $1/P$. Later versions of the package use the "`parallel`" pragma which has slightly less overhead.

To illustrate, consider the following predictive grid in $[-2, 2]^2$ spaced to avoid the original $N = 40$K design.

```
R> xx <- seq(-1.97, 1.95, by=0.04)
R> XX <- as.matrix(expand.grid(xx, xx))
R> YY <- f2d(XX)
```

The `aGP` function iterates over the elements of $\tilde{X} \equiv$ `XX`. The package used in this illustration is compiled for `OpenMP` support, and the `omp.threads` argument controls the number of threads

[2] If only the predictive mean is needed, and not the variance, then the time reduces to $O(TN + N^3)$.

used by aGP, divvying up XX. You can specify any positive integer for omp.threads, however a good rule-of-thumb is to match the number of cores. Here we set the default to two, since nearly all machines these days have at least one hyperthreaded core (meaning it behaves like two cores). However, this can be overwritten by the OMP_NUM_THREADS environment variable.

```
R> nth <- as.numeric(Sys.getenv("OMP_NUM_THREADS"))
R> if(is.na(nth)) nth <- 2
R> print(nth)

[1] 8
```

If your machine has fewer cores, is not compiled with OpenMP or caps the number of OpenMP threads to a lower value (see Appendix A.1), then it will take longer to run the examples here.

```
R> P.alc <- aGP(X, Y, XX, omp.threads=nth, verb=0)
```

Note that the default method is ALC. The results obtained with method = "mspe" are similar, but require more computation time. Further comparison is delayed until Section 3. The verb=0 argument suppresses a progress meter which is printed to the screen. Figure 3 shows

```
R> persp(xx, xx, -matrix(P.alc$mean, ncol=length(xx)), phi=45, theta=45,
+       main="", xlab="x1", ylab="x2", zlab="y-hat(x)")
```

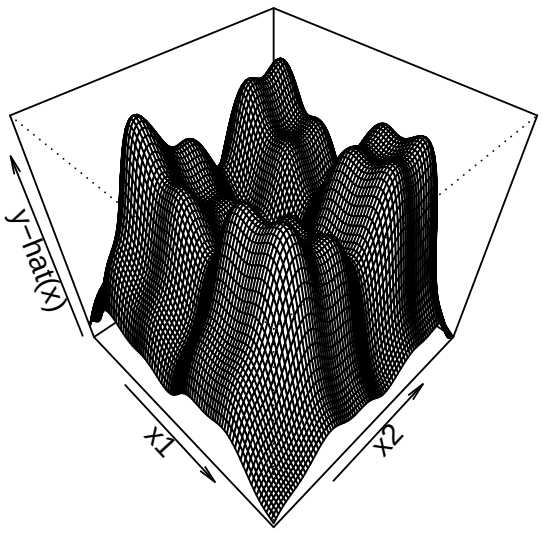

Figure 3: Emulated surface based on $N = 40$K and $|\mathcal{X}| = 10$K gridded predictive locations.

the resulting (predictive mean) emulation surface.[3] Although the input dimension is low, the input-output relationship is nuanced and merits a dense design in the input space to fully map.

For a closer look, Figure 4 shows a slice through that predictive surface at $x_w = 0.51$ along with the true responses (completely covered by the prediction) and error-bars. Observe that

[3]The negative is shown for better visibility.

```
R> med <- 0.51
R> zs <- XX[,2] == med
R> sv <- sqrt(P.alc$var[zs])
R> r <- range(c(-P.alc$mean[zs] + 2*sv, -P.alc$mean[zs] -2*sv))
R> plot(XX[zs,1], -P.alc$mean[zs], type="l", lwd=2, , ylim=r, xlab="x1",
+      ylab="predicted & true response", bty="n",
+      main="slice through surface")
R> lines(XX[zs,1], -P.alc$mean[zs] + 2*sv, col=2, lty=2, lwd=2)
R> lines(XX[zs,1], -P.alc$mean[zs] -2*sv, col=2, lty=2, lwd=2)
R> lines(XX[zs,1], YY[zs], col=3, lwd=2, lty=3)
```

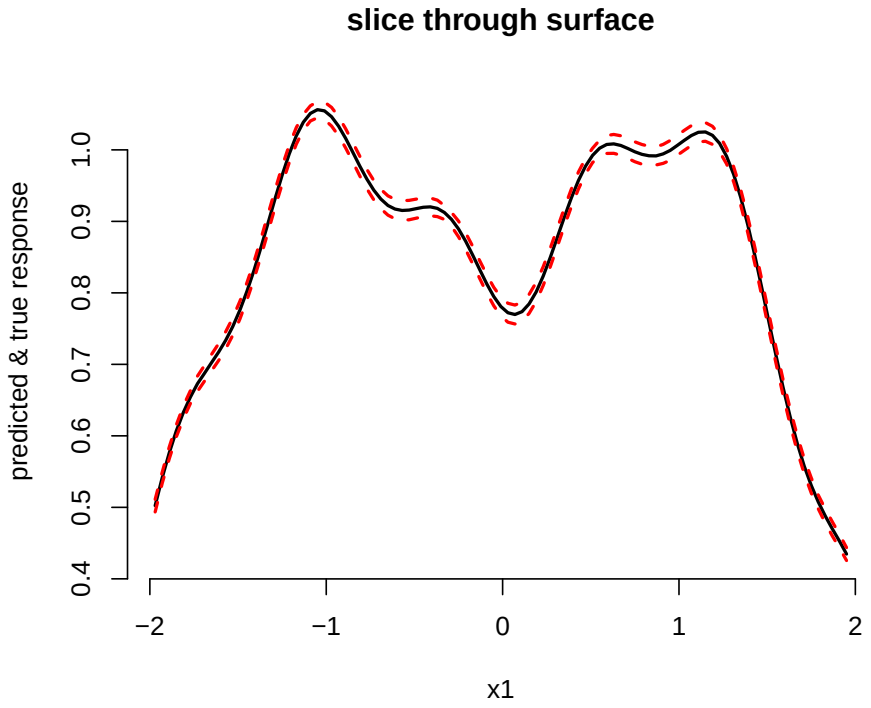

Figure 4: Slice of the predictive surface shown in Figure 3 including the true surface [covered by the mean] and predictive interval.

the error bars are very tight on the scale of the response, and that although no continuity is enforced—calculations at nearby locations are independent and occur potentially in parallel—the resulting surface looks smooth to the eye. This is not always the case, as we illustrate in Section 3.3.

Accuracy, however, is not uniform. Figure 5 shows that predictive bias oscillates across the same slice of the input space shown in Figure 4. Crucially, however, notice that the magnitude of the bias is small: one-hundredth of a tick on the scale of the response. Still, given the density of the input design one could easily guess that the model may not be flexible enough to characterize the fast-moving changes in the input-output relationships.

```
R> diff <- P.alc$mean - YY
R> plot(XX[zs,1], diff[zs], type="l", lwd=2,
+      main="systematic bias in prediction",
+      xlab="x1", ylab="y(x) - y-hat(x)", bty="n")
```

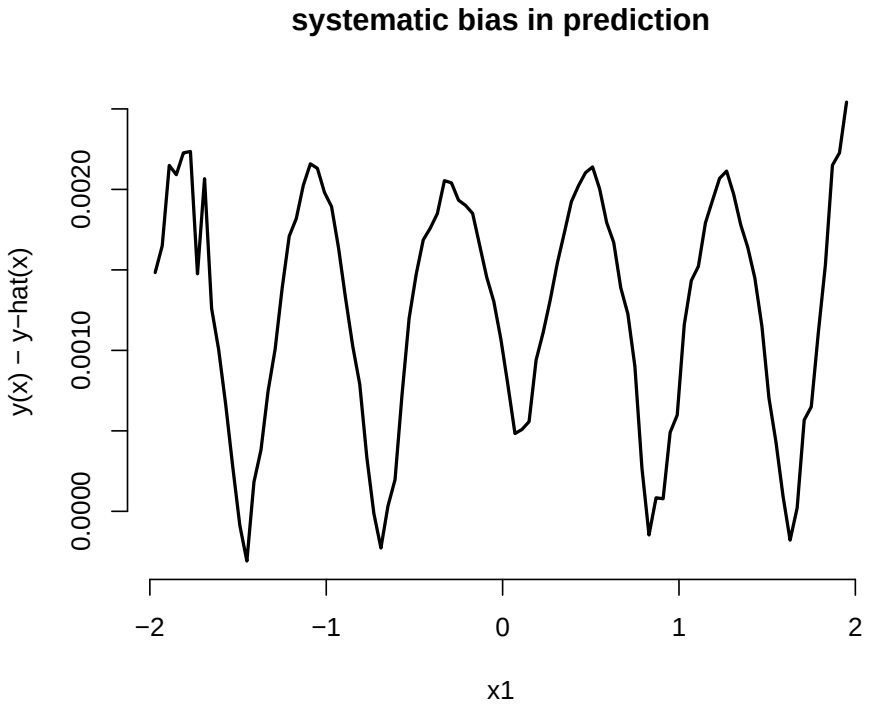

Figure 5: Bias in the predictive mean surface shown in Figure 4.

Although an approximation, the local nature of modeling means that, from a global perspective, the predictor is *more* flexible that the full-N stationary Gaussian process predictor. Here, stationarity loosely means that the covariance structure is modeled uniformly across the input space. Most choices of $K_\theta(\cdot,\cdot)$, like the isotropic Gaussian we use, induce stationarity in the spatial field. Inferring separate independent predictors across the elements of a vast predictive grid lends `aGP` a degree of non-stationarity. In fact, by default, `aGP` goes beyond that by learning separate $\hat{\theta}_n(x)$ local to each $x \in \mathcal{X}$ by maximizing the local likelihood (or posterior probabilities). Figure 6 shows indeed that the estimated lengthscales vary spatially. However, apparently even this extra degree of flexibility is not enough to mitigate the small amount of bias shown in Figure 5.

Gramacy and Apley recommend a two-stage scheme wherein the above process is repeated wherein new $X_n(x)$ are chosen conditional upon $\hat{\theta}_n(x)$ values from the first stage. i.e., so that the second iteration's local designs use locally estimated parameters. This leads to a globally non-stationary model and generally more accurate predictions than the single-stage scheme. The full scheme is outlined algorithmically in Figure 7. Step 2(b) of the algorithm implements the ALC reduction in variance scheme, via Eq.˜(6), although MSPE (4) or any other criteria could be deployed there, at each greedy stage of local design. Of course, more

```
R> plot(XX[zs,1], P.alc$mle$d[zs], type="l", lwd=2,
+      main="spatially varying lengthscale",
+      xlab="x1", ylab="theta-hat(x)", bty="n")
```

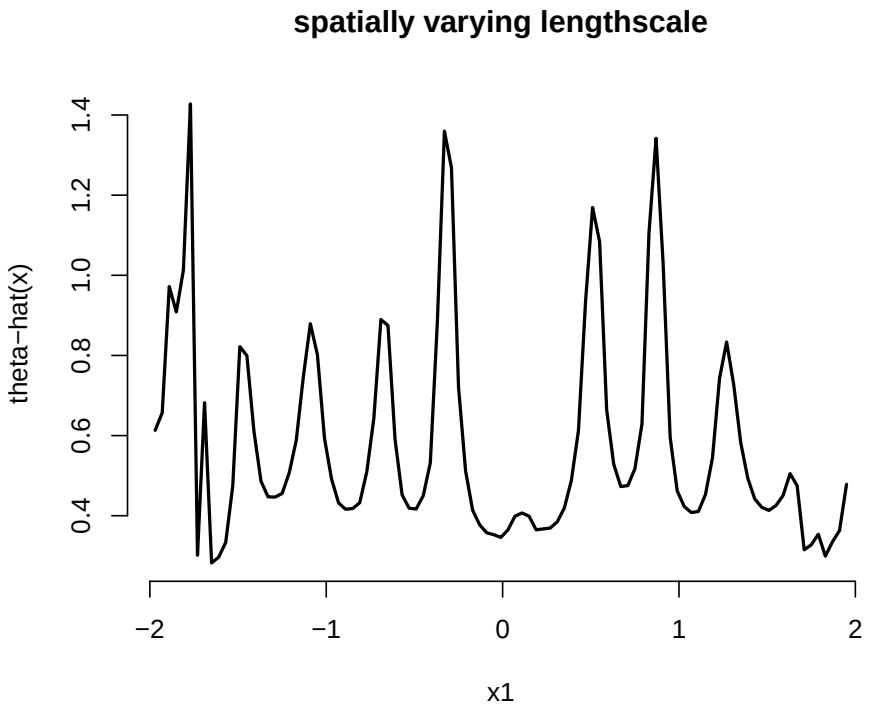

Figure 6: Spatially varying lengthscale estimated along the slice shown in Figure 4.

than two repetitions of the global search scheme can be performed, but in many examples two has been sufficient to achieve rough convergence of the overall iterative scheme. Optionally, the $\hat{\theta}_n(x)$-values can be smoothed (e.g., by **loess**) before they are fed back into the local design schemes. Considering other popular approaches to adapting a stationary model to achieve nonstationary surfaces, which usually involve orders of magnitude more computation (e.g. Schmidt and O'Hagan 2003, and references therein), this small adaptation is a thrifty alternative that does not change the overall computational order of the scheme.

Consider the following illustration continuing on from our example above.

```
R> df <- data.frame(y=log(P.alc$mle$d), XX)
R> lo <- loess(y~., data=df, span=0.01)
R> P.alc2 <- aGP(X, Y, XX, d=exp(lo$fitted), omp.threads=nth, verb=0)
```

This causes the design, for each element of `XX`, to initialize search based on the smoothed `d`-values output from the previous `aGP` run, stored in the `P.alc` object. Compariing the predictions from the first iteration to those from the second, we can see that the latter has lower RMSE.

```
R> rmse <- data.frame(alc=sqrt(mean((P.alc$mean - YY)^2)),
```

1. Choose a sensible starting global $\theta_x = \theta_0$ for all x.
2. Calculate local designs $X_n(x, \theta_x)$ based on ALC, independently for each x:
   (a) Choose a NN design $X_{n_0}(x)$ of size n_0.
   (b) For $j = n_0, \ldots, n-1$, set
   $$x_{j+1} = \arg \max_{x_{j+1} \in X_N \setminus X_j(x)} v_j(x; \theta_x) - v_{j+1}(x; \theta_x),$$
   and then update $D_{j+1}(x, \theta_x) = D_j(x, \theta_x) \cup (x_{j+1}, y(x_{j+1}))$.
3. Also independently, calculate the MLE $\hat{\theta}_n(x)|D_n(x, \theta_x)$ thereby explicitly obtaining a globally nonstationary predictive surface. Set $\theta_x = \hat{\theta}_n(x)$.
4. Repeat steps 2–3 as desired.
5. Output predictions $Y(x)|D_n(x, \theta_x)$ for each x.

Figure 7: Multi-stage approximate local GP modeling algorithm.

```
+   alc2=sqrt(mean((P.alc2$mean - YY)^2)))
R> rmse

          alc         alc2
1 0.00102464 0.0008455226
```

This result is not impressive, but it is statistically significant across a wide range of examples. For example Gramacy and Apley (2014) provided an experiment based on the borehole data [more in Section 3] showing that the second iteration consistently improves upon predictions from the first. Although explicitly facilitating a limited degree of non-stationarity, second stage local designs do not solve the bias problem completely. The method is still locally stationary, and indeed locally isotropic in its **laGP** implementation. Finally, the subsequent stages of design tend to be slightly faster than earlier stages since the number of Newton iterations required to converge on $\hat{\theta}_n(x)$ is reduced given refined starting values for search.

2.3. Computational techniques for speeding up local search

The most expensive step in Algorithm 7 is the inner-loop of Step 2(b), iterating over all $N - j$ remaining candidates in $X_N \setminus X_j(x)$ in search of X_{j+1}. Assuming the criteria involves predictive variance (3) in some way, every candidate entertained involves an $O(j^2)$ calculation. Viewed pessimistically, one could argue the scheme actually requires computation in $O(Nn^3)$ not $O(n^3)$. However, there are several reasons to remain cheery about computational aspects. One is that $O(Nn^3)$ is not $O(N^3)$. The others require more explanation, and potentially slight adjustments in implementation.

Not all $N-j$ need be entertained for the method to work well. For the same reason prediction is localized to x in the first place, that correlation decays quickly away from x, we can probably

afford to limit search to $N' \ll N - j$ candidates near to x. By default, `laGP` and `aGP` limit search to the nearest $N' = 1000$ locations, though this can be adjusted with the `close` argument. One can check [not shown here] that increasing `close` by an order of magnitude, to 2000 or 10,000 uses more compute cycles but yields identical results in the applications described in this document.

But it is risky to reduce `close` too much, as doing so will negate the benefits of search, eventually yielding the NN GP predictor. Another option, allowing N' to be greatly increased if desired, is to deploy further parallelization. Gramacy, Niemi, and Weiss (2014b) showed that ALC-based greedy search is perfect for graphical processing unit (GPU) parallelization. Each potential candidate, up to 65K candidates, can be entertained on a seprate GPU block, and threads within that block can be used to perform many of the required dense linear algebra operations in Eq.˜(6) in parallel. In practice they illustrate that this can result in speedups of between twenty and seventy times, with greater efficiencies for large n and N'. Enabling GPU subroutines requires custom compilation of CUDA source code via the NVIDIA compiler `nvcc` and re-compilation of the C code in the **laGP** package. For more details see Appendix A.2. For best results, enabling `OpenMP` support [Appendix A.1] is also recommended.

Finally, Gramacy and Haaland (2014) suggested that the discrete and exhaustive nature of search could be bypassed all together. They studied the topology of the reduction in variance landscape—the spatial surface searched in Step 2(b) via Eq.˜(6))—and observed many that regularities persist over choices of $K_\theta(\cdot, \cdot)$ and its parameterization. As long as the X_N is reasonably space-filling, local designs predictably exhibit the features observed in Figure 2: a substantial proportion of NNs accompanied by farther out satellite points positioned roughly along rays emanating from the reference predictive location, x. To mimic that behavior without exhaustive search they proposed a continuous one-dimensional line search along rays emanating from x. Optimizing along the ray is fast and can be implemented with library routines, like `Brent_fmin` (Brent 1973), the workhorse behind R's `optimize` function.

The code below calculates an such an ALC-ray based design, augmenting our example from Section 2.

```
R> p.alcray <- laGP(Xref, 6, 50, X, Y, d=0.1, method="alcray")
```

Although a similar idea could be deployed for finding MSPE-based designs based on rays, this is not implemented in the **laGP** package at the present time. Figure 8 compares local designs based on ray and exhaustive search. The exhaustive search design is identical to the ALC one shown in Figure 2, and just like in that example the ray-based version is not idential to the others but clearly exhibits similar qualitative features. The time required to derive the ALC-ray local design is:

```
R> p.alcray$time
```

```
elapsed
  0.031
```

and this is 3.6 times better than the exhaustive alternative. The predictive equations are nearly identical.

```
R> plot(X[p.alc$Xi,], xlab="x1", ylab="x2", type="n",
+   main="comparing local designs")
R> text(X[p.alc$Xi,], labels=1:length(p.alc$Xi), cex=0.7, col=2)
R> text(X[p.alcray$Xi,], labels=1:length(p.mspe$Xi), cex=0.7, col=3)
R> points(Xref[1], Xref[2], pch=19, col=3)
R> legend("topright", c("alc", "alcray"), text.col=c(2,3), bty="n")
```

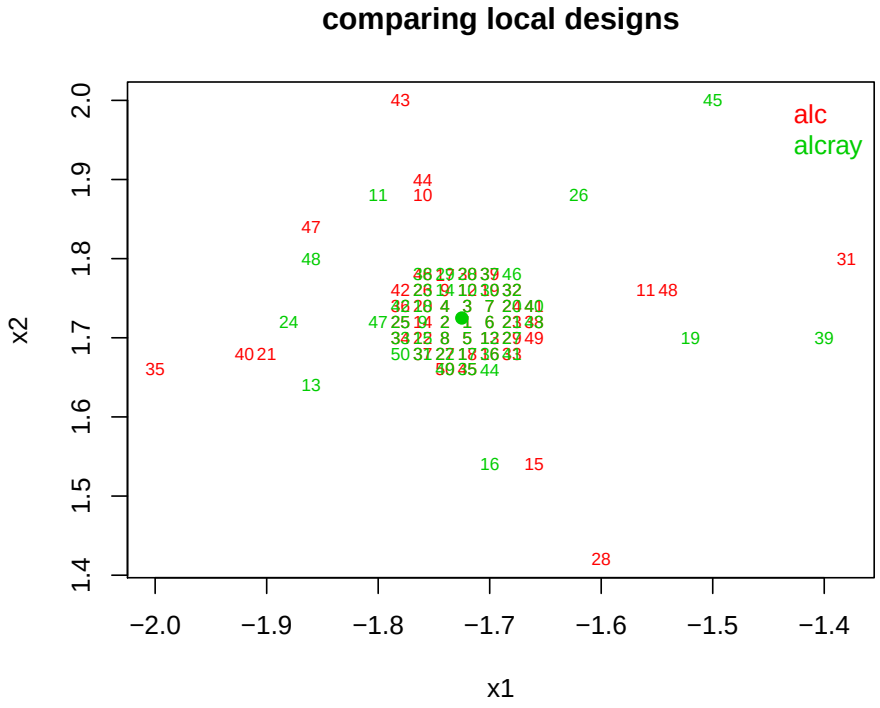

Figure 8: Local designs at x (green dot), derived under ALC and ALC-ray search criteria.

```
R> p <- rbind(p, c(p.alcray$mean, p.alcray$s2, p.alcray$df))
R> rownames(p)[3] <- c("alcray")
R> p
```

```
             mean           s2 df
mspe   -0.3723017 1.674662e-05 50
alc    -0.3722813 1.503320e-05 50
alcray -0.3722140 1.542859e-05 50
```

Gramacy and Haaland recommend using p rays per greedy search iteration, where p is the dimension of the input space. However this can be adjusted with the `numrays` argument, fine-tuning the exhaustiveness of search relative to the computational expense.

To complete the picture, the code below performs two stage global/local design based on ALC-ray searches.

```
R> P.alcray <- aGP(X, Y, XX, method="alcray", omp.threads=nth, verb=0)
R> dfray <- data.frame(y=log(P.alcray$mle$d), XX)
R> loray <- loess(y~., data=dfray, span=0.01)
R> P.alcray2 <- aGP(X, Y, XX, method="alcray", d=exp(loray$fitted),
+   omp.threads=nth, verb=0)
```

The result is a global predictor which is 9.2 times faster, echoing the single-x results form `laGP` above

```
R> c(P.alcray$time, P.alcray2$time)
```

```
elapsed elapsed
 28.184  26.578
```

and provides nearly identical out-of-sample accuracy via RMSE:

```
R> rmse <- cbind(rmse,
+   data.frame(alcray=sqrt(mean((P.alcray$mean - YY)^2)),
+     alcray2=sqrt(mean((P.alcray2$mean - YY)^2))))
R> rmse
```

```
         alc         alc2       alcray      alcray2
1 0.00102464 0.0008455226 0.0009678679 0.0009248219
```

Time in 55 seconds on a 2010 desktop to accurately emulate at 10K locations from an input design of $N = 40$K is an unmatched capability in the recent computer experiment literature.

3. Examples

The 2-d example above, while illustrative, was somewhat simplistic. Below we present three further examples which offer a more convincing demonstration of the merits of local GP prediction and expand its feature set to accommodate a wider range of application. After exploring its performance on the "borehole" data, a classic computer experiment benchmark, we illustrate how noisy data can be accommodated by estimating local nuggets. The following section provides a further example of how it can be deployed for computer model calibration.

3.1. Borehole data

The borehole experiment (Worley 1987; Morris, Mitchell, and Ylvisaker 1993) involves an 8-dimensional input space, and our use of it here follows the setup of Kaufman *et~al.* (2012); more details can be found therein. The response y is given by

$$y = \frac{2\pi T_u[H_u - H_l]}{\log\left(\frac{r}{r_w}\right)\left[1 + \frac{2LT_u}{\log(r/r_w)r_w^2 K_w} + \frac{T_u}{T_l}\right]}. \tag{8}$$

The eight inputs are constrained to lie in a rectangular domain:

$$r_w \in [0.05, 0.15] \quad r \in [100, 5000] \quad T_u \in [63070, 115600] \quad T_l \in [63.1, 116]$$
$$H_u \in [990, 1110] \quad H_l \in [700, 820] \quad L \in [1120, 1680] \quad K_w \in [9855, 12045].$$

We use the following implementation in R which accepts inputs in the unit 8-cube.

```
R> borehole <- function(x){
+   rw <- x[1] * (0.15 - 0.05) + 0.05
+   r <-  x[2] * (50000 - 100) + 100
+   Tu <- x[3] * (115600 - 63070) + 63070
+   Hu <- x[4] * (1110 - 990) + 990
+   Tl <- x[5] * (116 - 63.1) + 63.1
+   Hl <- x[6] * (820 - 700) + 700
+   L <-  x[7] * (1680 - 1120) + 1120
+   Kw <- x[8] * (12045 - 9855) + 9855
+   m1 <- 2 * pi * Tu * (Hu - Hl)
+   m2 <- log(r / rw)
+   m3 <- 1 + 2*L*Tu/(m2*rw^2*Kw) + Tu/Tl
+   return(m1/m2/m3)
+ }
```

We consider a modestly big training set ($N = 100000$), to illustrate how large emulations can proceed with relatively little computational effort. However, we keep the testing set somewhat smaller so that we can so that we can duplicate part of a Monte Carlo experiment (i.e., multiple repeats of random training and testing sets) from Gramacy and Apley (2014) without requiring too many compute cycles.

```
R> N <- 100000
R> Npred <- 1000
R> dim <- 8
R> library(lhs)
```

The experiment involves ten repetitions of inputs from a random Latin hypercube sample (LHS; McKay, Conover, and Beckman 1979) which generate the training data and testing sets, with responses from `borehole`, and automates calculations of a sequence of (local GP) estimators fit to the training sets followed by out-of-sample RMSE calculations on the testing sets. Storage for those RMSEs, along with timing info, is allocated as follows

```
R> T <- 10
R> nas <- rep(NA, T)
R> times <- rmse <- data.frame(mspe=nas, mspe2=nas,
+   alc.nomle=nas, alc=nas, alc2=nas,
+   nn.nomle=nas, nn=nas, big.nn.nomle=nas, big.nn=nas,
+   big.alcray=nas, big.alcray2=nas)
```

The names of the columns of the data frame are indicative of the corresponding estimator. For example, `big.nn.nomle` indicates a nearest neighbor (NN) estimator fit to with a larger local neighborhood ($n = 200$) using a sensible, but not likelihood maximizing, global value of θ. The other estimators describe variations either via a smaller local neighborhood ($n = 50$), greedy search, and local calculation of $\hat{\theta}_n(x)$.

The `for` loop below iterates over each Monte Carlo repetition.

```
R> for(t in 1:T) {
+
```

```
+   ## generate random training and testing data
+   x <- randomLHS(N+Npred, dim)
+   y <- apply(x, 1, borehole)
+   ypred.0 <- y[-(1:N)]; y <- y[1:N]
+   xpred <- x[-(1:N),]; x <- x[1:N,]
+
+   ## arguments common to all comparators
+   formals(aGP)[c("omp.threads", "verb")] <- c(8, 0)
+   formals(aGP)[c("X", "Z", "XX")] <- list(x, y, xpred)
+
+   out1<- aGP(d=list(mle=FALSE, start=0.7))
+   rmse$alc.nomle[t] <- sqrt(mean((out1$mean - ypred.0)^2))
+   times$alc.nomle[t] <- out1$time
+
+   out2 <- aGP(d=list(max=20))
+   rmse$alc[t] <- sqrt(mean((out2$mean - ypred.0)^2))
+   times$alc[t] <- out2$time
+
+   out3 <- aGP(d=list(start=out2$mle$d, max=20))
+   rmse$alc2[t] <- sqrt(mean((out3$mean - ypred.0)^2))
+   times$alc2[t] <- out3$time
+
+   out4 <- aGP(d=list(max=20), method="alcray")
+   rmse$alcray[t] <- sqrt(mean((out4$mean - ypred.0)^2))
+   times$alcray[t] <- out4$time
+
+   out5 <- aGP(d=list(start=out4$mle$d, max=20), method="alcray")
+   rmse$alcray2[t] <- sqrt(mean((out5$mean - ypred.0)^2))
+   times$alcray2[t] <- out5$time
+
+   out6<- aGP(d=list(max=20), method="mspe")
+   rmse$mspe[t] <- sqrt(mean((out6$mean - ypred.0)^2))
+   times$mspe[t] <- out6$time
+
+   out7 <- aGP(d=list(start=out6$mle$d, max=20), method="mspe")
+   rmse$mspe2[t] <- sqrt(mean((out7$mean - ypred.0)^2))
+   times$mspe2[t] <- out7$time
+
+   out8 <- aGP(d=list(mle=FALSE, start=0.7), method="nn")
+   rmse$nn.nomle[t] <- sqrt(mean((out8$mean - ypred.0)^2))
+   times$nn.nomle[t] <- out8$time
+
+   out9 <- aGP(end=200, d=list(mle=FALSE), method="nn")
+   rmse$big.nn.nomle[t] <- sqrt(mean((out9$mean - ypred.0)^2))
+   times$big.nn.nomle[t] <- out9$time
+
+   out10 <- aGP(d=list(max=20), method="nn")
```

```
+   rmse$nn[t] <- sqrt(mean((out10$mean - ypred.0)^2))
+   times$nn[t] <- out10$time
+
+   out11 <- aGP(end=200, d=list(max=20), method="nn")
+   rmse$big.nn[t] <- sqrt(mean((out11$mean - ypred.0)^2))
+   times$big.nn[t] <- out11$time
+
+   out12 <- aGP(end=200, d=list(max=20), method="alcray")
+   rmse$big.alcray[t] <- sqrt(mean((out12$mean - ypred.0)^2))
+   times$big.alcray[t] <- out12$time
+
+   out13 <- aGP(end=200, d=list(start=out12$mle$d, max=20), method="alcray")
+   rmse$big.alcray2[t] <- sqrt(mean((out13$mean - ypred.0)^2))
+   times$big.alcray2[t] <- out13 $time
+ }
```

The code below collects summary information into a table, whose rows are ordered by average RMSE value. The final column of the table shows the p-value of a one-sided t-test for differences between adjacent rows in the table—indicating if the RMSE in the row is statistically distinguishable from the one below it.

```
R> timev <- apply(times, 2, mean, na.rm=TRUE)
R> rmsev <- apply(rmse, 2, mean)
R> tab <- cbind(timev, rmsev)
R> o <- order(rmsev, decreasing=FALSE)
R> tt <- rep(NA, length(rmsev))
R> for(i in 1:(length(o)-1)) {
+   tto <- t.test(rmse[,o[i]], rmse[,o[i+1]], alternative="less",
+     paired=TRUE)
+   tt[o[i]] <- tto$p.value
+ }
R> tab <- cbind(tab, data.frame(tt))
R> tab[o,]
```

	timev	rmsev	tt
big.alcray2	334.6791	0.3890996	8.912393e-03
big.alcray	327.8781	0.4039805	1.001619e-13
big.nn	43.5437	0.7280420	8.718216e-02
mspe2	63.3292	0.7366060	9.904689e-04
alc2	30.3777	0.7443531	7.150094e-04
alcray2	11.1302	0.7656684	1.768074e-06
alcray	10.7334	0.8219211	4.636272e-05
big.nn.nomle	4.0813	0.9115157	4.022528e-01
mspe	64.0382	0.9142446	6.327467e-02
alc	30.8694	0.9225462	1.342060e-06
alc.nomle	29.9871	1.0347613	1.140479e-13
nn	1.6623	2.1713632	3.170771e-14
nn.nomle	0.8079	3.6184245	NA

The two biggest takeaways from the table are that (1) everything is fast on a data set of this size by comparison to the state of the art in GP emulation, approximately or otherwise; (2) local inference of the lengthscale parameter, $\hat{\theta}_n(x)$ leads to substantial improvements in accuracy. Gramacy and Apley's similar experiments included variations on the method of compactly supported covariances (CSC) (Kaufman *et˜al.* 2012) which provided estimators with similar accuracies, but required at least an order magnitude more compute time. In fact, they commented that $N = 10000$ was the limit that CSC could accommodate on their machine due to memory swapping issues.

The best methods, based on a larger local neighborhood and ray-based search, point to an impressive emulation capability. In a time that is comparable to a plain NN-based emulation strategy (with local inference for $\hat{\theta}_n(x)$; i.e., `nn` in the table), a greedy design is three times more accurate out-of-sample. Gramacy and Haaland (2014) show that the trend continues as N is increased, indicating the potential for extremely accurate emulation on testing and training sets of size $N > 1$M in a few hours. Pairing with cluster-style distribution, across 96 16-CPU nodes, that can be reduced to 188 seconds, or extended to $N > 8M$ in just over an hour. They show that smaller (yet still large) designs $N < 100000$, searching exhaustively rather than by rays leads to more accurate predictors. In those cases, massive parallelization over a cluster and/or with GPUs (Gramacy *et˜al.* 2014b) can provide accurate predictions on a commensurately sized testing set (N) in about a minute.

3.2. Challenging global/local isotropy

Our choice of isotropic correlation function was primarily one of convenience. It is a common first choice for computer experiments, and since it has just one parameter, θ, inferential schemes like maximum likelihood via Newton-like methods are vastly simplified. When deployed for local inference over thousands of elements of a vast predictive grid, that simplicity is a near necessity from an engineering perspective. However, the local GP methodology is not limited to this choice. Indeed Gramacy and Apley (2014) developed all of the relevant equations for a generic choice of separable correlation function. Here, separable means the joint correlation over all input directions factors as a product of a simpler one in each direction, independently. The simplest example is a separable Gaussian form, $K_\theta(x, x') = \exp\{-\sum_{k=1}^{p}(x_k - x'_k)^2/\theta_k\}$. It is easy to imagine, as in our eight-dimensional borehole example above, that the spatial model could benefit for allowing differential rate of decay θ_k in each input direction.

The **laGP** package contains limited support for a separable correlation function in the context of *global*, that is canonical, GP inference. On a data set of size $N = 100$K like the one we entertain above, this is not a reasonable undertaking. But we have found it useful on subsets of the data for the purpose of obtaining a rough re-scaling of the inputs so that a (local) isotropic analysis is less objectionable. For example, the code below considers ten random subsets of size $n = 1$K from the full $N = 100$K design, and collects $\hat{\theta}$ vectors under the separable Gaussian formulation.

```
R> ## allocate space
R> thats <- matrix(NA, nrow=T, ncol=dim)
R> its <- rep(NA, T)
R> n <- 1000
R> ## get reasonable starting values and ranges
```

```
R> g2 <- garg(list(mle=TRUE), y)
R> d2 <- darg(list(mle=TRUE, max=100), x)
R> ## do reps
R> for(t in 1:T) {
+
+   ## get random subset
+   subs <- sample(1:N, n, replace=FALSE)
+
+   ## estimate with fixed small nugget
+   gpsepi <- newGPsep(x[subs,], y[subs], rep(d2$start, dim), g=1/1000)
+   that <- mleGPsep(gpsepi, param="d", tmin=d2$min, tmax=d2$max,
+     ab=d2$ab, maxit=200)
+   thats[t,] <- that$d
+   its[t] <- that$its
+
+   ## free space
+   deleteGPsep(gpsepi)
+ }
```

The `mleGPsep` function uses `optim` with `method="L-BFGS-B"` together with analytic derivatives of the log likelihood; the function `mleGP` offers a similar feature for the isotropic Gaussian correlation, except that it uses a Newton-like method with analytic first and second derivatives. The package also offers `jmleGPsep`, an analog of `jmleGP`, which automates a profile-like approach to iterating over $\theta|\eta$ and $\eta|\theta$ where the latter is performed with a Newton-like scheme leveraging first and second derivatives. We do not demonstrate `jmleGPsep` on this example since the large data subset ($n = 1000$) combined with very smooth deterministic outputs from moderately size (8-dim) inputs, from the borehole experiment, leads to estimating near-zero nuggets and ill-conditioning in the matrix decompositions, arising from our choice of Gaussian correlation function.

For estimating nuggets in this setup, where the response is both deterministic and extremely smooth (and stationary), we recommend the **GPfit** (MacDoanld, Chipman, and Ranjan 2014) based on the methods of Ranjan, Haynes, and Karsten (2011). However, we caution that our experience is **GPfit** can be slow on data sets as large as $N = 1000$.

Figure 9 shows the distribution of estimated lengthscales obtained by randomizing over subsets of size $n = 1000$. We see that some lengthscales are orders of magnitude smaller than others, suggesting that some inputs may be more important than others. Input one (r_w) has a distribution that is highly concentrated near small values suggesting that it may be the most important. Perhaps treating all inputs equally when performing a global/local approximation, as in Section 3.1, is leaving some predictability on the table. The **laGP** package does not support using a separable correlation function for local analysis, however we can pre-scale the data globally to explore whether there is any benefit from treating some inputs differently than others.

```
R> scales <- sqrt(apply(thats, 2, median))
R> xs <- x; xpreds <- xpred
R> for(j in 1:ncol(xs)) {
+   xs[,j] <- xs[,j]/scales[j]
```

```
R> boxplot(thats, main="distribution of thetas", xlab="input", ylab="theta")
```

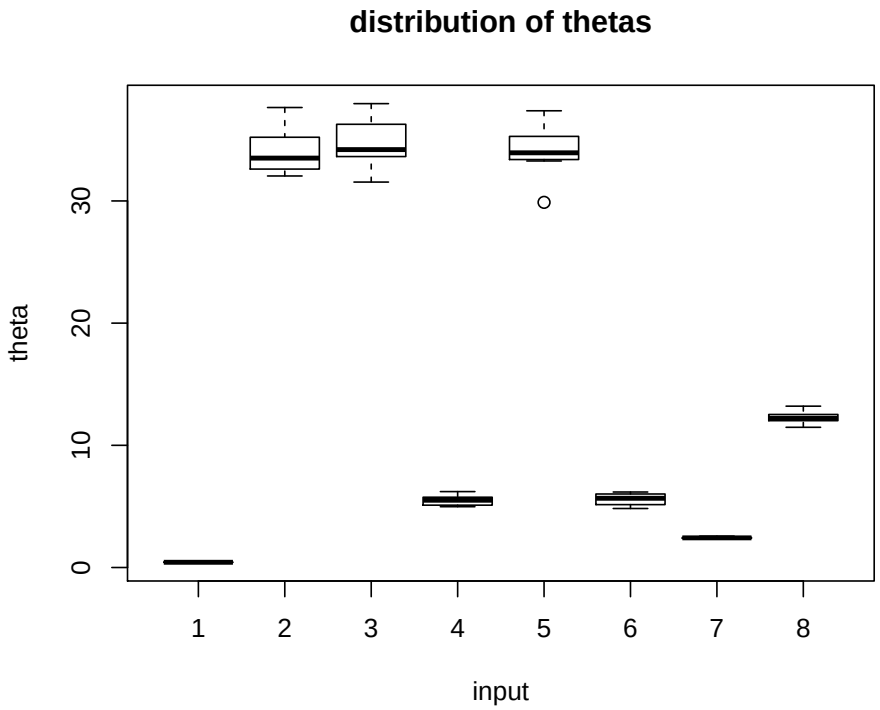

Figure 9: Distribution of maximum *a' posteriori* lengthscales over random subsets of the borehole data.

```
+   xpreds[,j] <- xpreds[,j]/scales[j]
+ }
```

Using the new inputs, consider the following global approximation for the final iteration in the Monte Carlo experiment from Section 3.1.

```
R> out14 <- aGP(xs, y, xpreds, d=list(max=20), method="alcray")
```

The RMSE obtained,

```
R> sqrt(mean((out14$mean - ypred.0)^2))
```

```
[1] 0.3885364
```

is competitive with the best methods in the study above—which are based on $n = 200$ whereas only the default $n = 50$ was used here. Also observe that the RMSE we just obtained is better than half of the one we reported "`alcray`" in the Monte Carlo experiment.

Determining if this reduction is statistically significant would require incorporating it into the Monte Carlo. We encourage the reader to test that off-line, if so inclined, and permit us

and simply conclude here that it can be beneficial to perform a cursory global analysis with a separable correlation function to determine if the inputs should be scaled before performing a local (isotropic) analysis on the full data set.

3.3. Motorcycle data

For a simple illustration of heteroskedastic local GP modeling, consider the motorcycle accident data (Silverman 1985), simulating the acceleration of the head of a motorcycle rider as a function of time in the first moments after an impact. It can be found in the **MASS** package for R. For comparison, we first fit a simple GP model to the full data set ($N = 133$), estimating both lengthscale θ and nugget η.

```
R> library(MASS)
R> d <- darg(NULL, mcycle[,1,drop=FALSE])
R> g <- garg(list(mle=TRUE), mcycle[,2])
R> motogp <- newGP(mcycle[,1,drop=FALSE], mcycle[,2], d=d$start,
+   g=g$start, dK=TRUE)
R> jmleGP(motogp, drange=c(d$min, d$max), grange=c(d$min, d$max),
+   dab=d$ab, gab=g$ab)

         d         g tot.its dits gits
1 54.28291 0.2771448      82   26   56
```

Now consider the predictive equations derived from that full-data, alongisde a local approximate alternative (via ALC) with a local neighborhood size of $n = 30$.

```
R> XX <- matrix(seq(min(mcycle[,1]), max(mcycle[,1]), length=100), ncol=1)
R> motogp.p <- predGP(motogp, XX=XX, lite=TRUE)
R> motoagp <- aGP(mcycle[,1,drop=FALSE], mcycle[,2], XX, end=30,
+   d=d, g=g, verb=0)
```

Figure 10 shows the predictive surfaces obtained for the two predictors in terms of means and 90% credible intervals. The (full) GP mean surface, shown as solid-black, is smooth and tracks the center of the data nicely from left to right over the range of x-values. However, it is poor at capturing the heteroskedastic nature of the noise (dashed-black). The local GP mean is similar, except near $x = 35$ where it is not smooth. This is due to the small design. With only $N = 132$ there isn't much opportunity for smooth transition as the local predictor tracks across the input space, leaving little wiggle room to make a trade-off between smoothness ($n = 132$, reproducing the full GP results exactly) and adaptivitity ($n \ll 132$). Although the mean of the local GP may disappoint, the variance offers an improvement over the full GP. It is conservative where the response is wiggly, being similar to the full GP but slightly wider, and narrower where the response is flat.

It is interesting to explore how the local GP approximation would fare on a larger version of the same problem, where otherwise a local approach is not only essential for computational reasons, but also potentially more appropriate from a nonstationary modeling perspective on this data. For a crude simulation of a larger data setup we replicated the data ten times with a little bit of noise on the inputs.

```
R> plot(mcycle, cex=0.5, main="motorcycle data")
R> lines(XX, motogp.p$mean, lwd=2)
R> q1 <- qnorm(0.05, mean=motogp.p$mean, sd=sqrt(motogp.p$s2))
R> q2 <- qnorm(0.95, mean=motogp.p$mean, sd=sqrt(motogp.p$s2))
R> lines(XX, q1, lty=2, lwd=2)
R> lines(XX, q2, lty=2, lwd=2)
R> lines(XX, motoagp$mean, col=2, lwd=2)
R> q1 <- qnorm(0.05, mean=motoagp$mean, sd=sqrt(motoagp$var))
R> q2 <- qnorm(0.95, mean=motoagp$mean, sd=sqrt(motoagp$var))
R> lines(XX, q1, lty=2, col=2, lwd=2)
R> lines(XX, q2, lty=2, col=2, lwd=2)
```

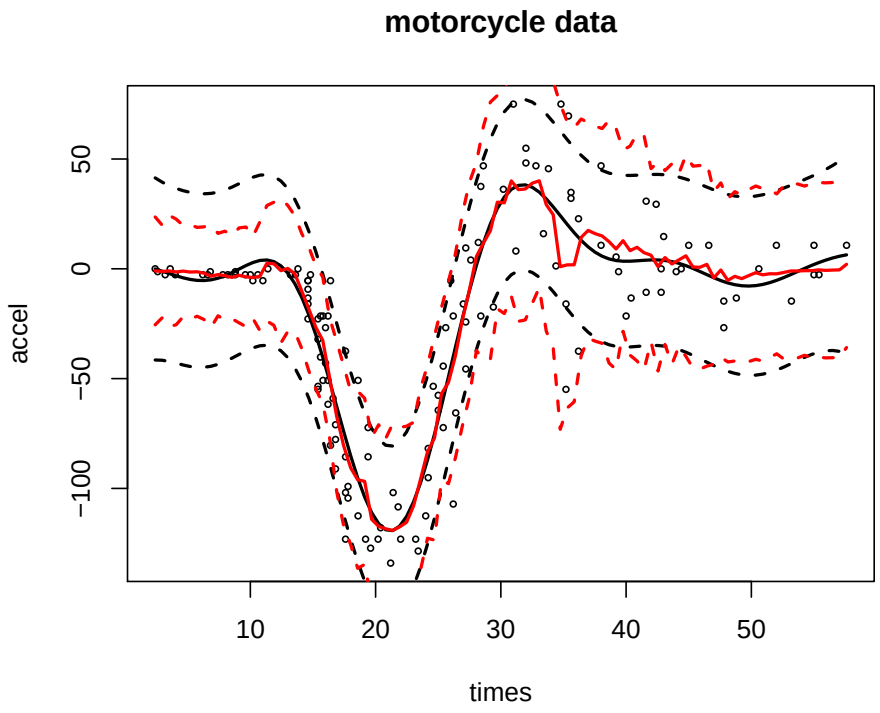

Figure 10: Comparison of a global GP predictive surface (black) with a local one (red). Predictive means (solid) and 90% interval (dashed) shown.

```
R> X <- matrix(rep(mcycle[,1], 10), ncol=1)
R> X <- X + rnorm(nrow(X), sd=1)
R> Z <- rep(mcycle[,2], 10)
R> motoagp2 <- aGP(X, Z, XX, end=30, d=d, g=g, verb=0)
```

Figure 11 shows the resulting predictive surface. Notice how it does a much better job of tracing predictive uncertainty across the input space. The predictive mean is still overly wiggly, but it does appear to be reasonably smooth also reveals structure in the data that may not have been evident from the scatter-plot alone, and likewise is disguised (or overly

```
R> plot(X, Z, main="simulating a larger data setup", xlab="times", ylab="accel")
R> lines(XX, motoagp2$mean, col=2, lwd=2)
R> q1 <- qnorm(0.05, mean=motoagp2$mean, sd=sqrt(motoagp2$var))
R> q2 <- qnorm(0.95, mean=motoagp2$mean, sd=sqrt(motoagp2$var))
R> lines(XX, q1, col=2, lty=2, lwd=2)
R> lines(XX, q2, col=2, lty=2, lwd=2)
```

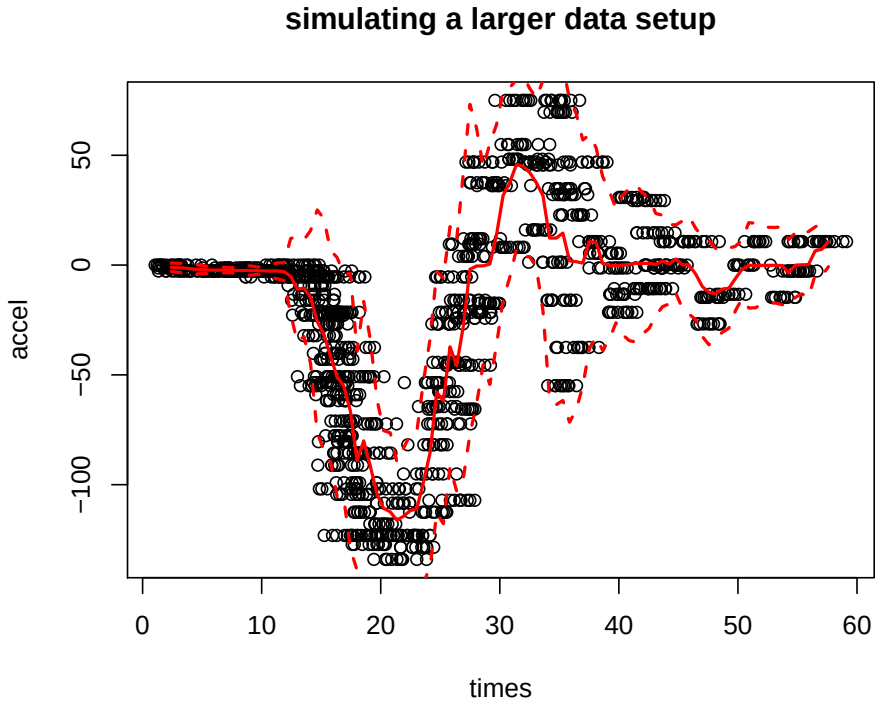

Figure 11: Predictive surface obtained after combining ten replications of the data with jittered x-values.

smoothed) by the full GP fit. The local GP is picking up oscillations for larger input values which make sense considering the output is measuring a whiplash effect. However, that may simply be wishful thinking; the replicated response values paired with the jittered predictors may not be representative of what would have been observed in a larger simulation.

4. Calibration

Computer model *calibration* is the enterprise of matching a simulation engine with real, or field, data to ultimately build an accurate predictor for the real process at novel inputs. In the case of large computer simulations, calibration represents a capstone application uniquely blending (and allowing review of) features, for both large and small-scale spatial modeling via GPs, provided by the **laGP** package.

Kennedy and O'Hagan (2001) were the first to propose a statistical framework for combining

simulation output and noisy field observations for model calibration. Their hierarchical model links field measurements to potentially biased computer simulations. It is paired with a Bayesian inferential framework for jointly estimating, using data from both processes, the bias, noise level, and any parameters required to run the computer simulation—so-called *calibration parameter(s)*—but which cannot be controlled or observed in the field. The setup, which we review below, has many attractive features, however it scales poorly when simulations get large. We explain how Gramacy, Bingham, Holloway, Grosskopf, Kuranz, Rutter, Trantham, and Drake (2014a) modified that setup using **laGP** and provide a live demonstration via an example extracted from that paper.

4.1. A hierarchical model for Bayesian inference

Consider data comprised of runs of a computer model M at a large space-filling design, and a much smaller number observations from a physical or field experiment F which follows a design that respects limitations of the experimental apparatus. It is typical to assume that the runs of M are deterministic, and that its input space fully contains that of F. Use x to denote *design variables*, that can be adjusted, or at leased measured, in the physical system; and let u to denote *calibration* or *tuning parameters*, whose values are required to simulate the system, but are unknown in the field. The primary goal is to predict the result of new field data experiments, via M, which in turn means finding a good u.

Toward that goal, Kennedy and O'Hagan (2001, hereafter KOH) proposed the following coupling of M and F. Let $y_j^F(x)$ denote the j^{th} replication of the field "run" at x, and $y^M(x,u)$ denote the (deterministic) output of a computer model run. KOH represent the *real* mean process R as the computer model output at the best setting of the tuning parameters, u^*, plus a bias term acknowledging that there can be systematic discrepancies between the computer model and the underlying mean of the physical process. In symbols, the mean of the physical process is $y^R(x) = y^M(x,u^*) + b(x)$. The field observations connect reality with data:

$$\begin{aligned} & y_j^F(x) = y^R(x) + \varepsilon_{xj}, && \varepsilon_{xj} \overset{\text{iid}}{\sim} \mathcal{N}(0,\sigma_\varepsilon^2), \quad j = 1,\ldots,n_x. \\ \text{giving} \quad & y_j^F(x) = y^M(x,u^*) + b(x) + \varepsilon_{xj}, && \end{aligned} \tag{9}$$

The unknown parameters are u^*, σ_ε^2, and the discrepency or bias $b(\cdot)$.

If evaluating the computer model is fast, then inference can proceed via residuals $y_j^F(x) - y^M(x,u)$, which can be computed at will for any (x,u) (Higdon, Kennedy, Cavendish, Cafeo, and Ryne 2004). However, y^M simulations are usually time consuming, in which case it helps to build an emulator $\hat{y}^M(\cdot,\cdot)$ fit to code outputs obtained on a computer experiment design of N_M locations (x,u). KOH recommend a GP prior for y^M, however rather learn $\hat{y}^M$ in isolation, using just the N_M runs, as we have been doing throughout this document, they recommend inference joint with $b(\cdot)$, u, and σ_ε^2 using both field observations and runs of the computer model. From a Bayesian perspective this is the coherent thing to do: infer all unknowns jointly given all data.

This is a practical approach when the computer model is *very* slow, giving small N_M. In that setup, the field data can be informative for emulation of $y^M(\cdot,\cdot)$, especially when the bias $b(\cdot)$ is very small or easy to estimate. Generally however, the computation required for inference in this setup is fraught with challenges, especially in the fully Bayesian formulation

recommended by KOH. The coupled $b(\cdot)$ and $y^M(\cdot,\cdot)$ lead to parameter identification and MCMC mixing issues. And GP regression, taking a substantial computational toll when deployed in isolation, faces a compounded burden when coupled with other processes.

4.2. Calibration as optimization

Gramacy *et˜al.* (2014a) proposed a thriftier approach pairing local approximate GP models for emulation with a modularized calibration framework (Liu, Bayarri, and Berger 2009) and derivative free optimization (Conn, Scheinberg, and Vicente. 2009). *Modularized* calibration sounds fancy, but its really just back-to-basics: fitting the emulator $\hat{y}^M(\cdot,\cdot)$ separately or independently from the bias, using only the outputs of runs at a design of N_M inputs (x,u). Liu *et˜al.*'s justification for modularization stemmed from a "contamination" concern echoed by other researchers (e.g., Joseph 2006; Santner *et˜al.* 2003) where, in the fully Bayesian scheme, joint inference allows "pristine" field observations to be contaminated by an imperfect computer model.

Gramacy *et˜al.* motivate modularization from a more practical perspective, that of decoupling inference for computational tractability in large N_M settings. They argue that there is little harm in doing so for most modern calibration applications, in terms of the quality of estimates obtained irrespective of computational considerations. Due to the relative costs, the number of computer model runs involved increasingly dwarfs the data available from the field, i.e., $N_M \gg N_F$, making it unlikely that field data would substantively enhance the quality of the emulator, leaving only risk that joint inference with the bias will obfuscate traditional computer model diagnostics, and possibly stunt their subsequent re-development or refinement.

Combining modularization with local approximate GPs for emulation, and full GP regressions (with nugget η) for estimating bias-plus-noise from a relatively small number of field data observations, N_F, Gramacy *et˜al.* recommend viewing calibration as an optimization which acts as the glue that "sticks it all together". Algorithm 1 provides pseudocode comprised

Require: Calibration parameter u, fidelity parameter n_M, computer data $D^M_{N_M}$, and field data $D^F_{N_F}$.

1: **for** $j = 1, \ldots, N_F$ **do**
2: $I \leftarrow \texttt{laGP}(x_j^F, u \mid n_M, D^M_{N_M})$ {get indicies of local design}
3: $\hat{\theta} \leftarrow \texttt{mleGP}(D^M_{N_M}[I])$ {local MLE of correlation parameter(s)}
4: $y_j^{M|u} \leftarrow \texttt{muGP}(x_j^F \mid D^M_{N_M}[I], \hat{\theta})$ {predictive mean emulation following Eq.˜(3)}
5: **end for**
6: $\hat{b}_{N_F} \leftarrow Y^F_{N_F} - Y^{M|u}$ {vectorized bias calculation}
7: $D^{\hat{b}}_{N_F} \leftarrow (\hat{b}_{N_F}, X^F_{N_F})$ {create data for estimating $\hat{b}(\cdot)|\mu$}
8: $\hat{\theta} \leftarrow \texttt{mleGP}(D^{\hat{b}}_{N_F})$ {full GP estimate of $\hat{b}|u$}
9: **return** $\texttt{predGP}(Y^F_{N_M}|X_{N_M}, \hat{\theta})$ {multivariate Student-t density generalizing (3)}

Algorithm 1: Objective function evaluation for modularized local GP calibration.

library functions describing the objective function. In **laGP**, this objective is implemented as `fcalib`, comprising of first (steps 1–5) a call to `aGP.seq` to emulate on a schedule of sequential stages of local refinements [Figure 7]; and then (6–8) a call to `discrep.est` which estimates the

GP discrepancy or bias term. The notation used in the psuedo-code, and further explanation, is provided below.

Let the field data be denoted as $D^F_{N_F} = (X^F_{N_F}, Y^F_{N_F})$ where $X^F_{N_F}$ is the design matrix of N_F field data inputs, paired with a N_F vector of y^F observations $Y^F_{N_F}$. Similarly, let $D^M_{N_M} = ([X^M_{N_M}, U_{N_M}], Y^M_{N_M})$ be the N_M computer model input-output combinations with column-combined x- and u-design(s) and y^M-outputs. Then, with an emulator $\hat{y}^M(\cdot, u)$ trained on $D^M_{N_M}$, let $\hat{Y}^{M|u}_{N_F} = \hat{y}^M(X^F_{N_F}, u)$ denote a vector of N_F emulated output y-values at the X_F locations obtained under a setting, u, of the calibration parameter. With local approximate GP modeling, each $\hat{y}^{M|u}_j$-value therein, for $j = 1, \ldots, N_F$, can be obtained independently (and in parallel) with the others via local sub-design $X_{n_M}(x^F_j, u) \subset [X^M_{N_M}, U_{N_M}]$ and local inference for the correlation structure. A key advantage of this approach, which makes **laGP** methods well-suited to the task, is that emulation is performed only where it is needed, at a small number N_F of locations $X^F_{N_F}$, regardless of the size N_M of the computer model data. The size of the local sub-design, n_M, is a fidelity parameter, meaning that larger values provide more accurate emulation at greater computational expense. Finally, denote the N_F-vector of fitted discrepancies as $\hat{b}_{N_F} = Y^F_{N_F} - \hat{Y}^{M|u}_{N_F}$. Given these quantities, the objective function for calibration of u, coded in Algorithm 1, is the (log) joint probability density of observing $Y^F_{N_F}$ at inputs $X^F_{N_F}$. Since N_F is small, this can be obtained from a best-fitting GP regression model trained on data $D^{\hat{b}}_{N_F} = (\hat{b}_{N_F}, X^F_{N_F})$, representing the bias estimate $\hat{b}(\cdot)$.

Objective function in hand, we turn to optimizing. The discrete nature of independent local design searches for $\hat{y}^M(x^F_j, u)$ ensures that the objective is not continuous in u. It can look 'noisy', although it is in fact deterministic. This means that optimization with derivatives—even numerically approximated ones—is fraught with challenges. Gramacy *et˜al.* suggest a derivative-free approach via the mesh adaptive direct search (MADS) algorithm (Audet and Dennis, Jr. 2006) known as **NOMAD** (Le˜Digabel 2011). The authors of the **crs** package (Racine and Nie 2012) provide `snomadr`, an R wrapper to the underlying C++ **NOMAD** implementation. MADS/**NOMAD** proceeds by successive pairs of *search* and *poll* steps, trying inputs to the objective function on a sequence of meshes which are refined in such a way as to guarantee convergence to a local optima under very weak regularity conditions; for more details see Audet and Dennis, Jr. (2006).

As MADS is a local solver, **NOMAD** requires initialization. Gramacy *et˜al.* recommend choosing starting u-values from the best value(s) of the objective found on a small random space-filling design. We note here that although **laGP** provides functions like `fcalib`, `aGP.seq` and `discrep.est` to facilitate calibration via optimization, there is no single subroutine automating the combination of all elements: selection of initial search point, executing search, and finally utilizing the solution to make novel predictions in the field. The illustrative example below in Section 4.3 is intended to double as a skeleton for novel application. It involves a `snomadr` call with objective `fcalib`, after pre-processing to find an initial u-value via simple iterative search over `fcalib` calls. Then, after optimization returns an optimal u^* value, the example demonstrates how estimates of $\hat{b}(x)$ and $\hat{y}^M(x, u^*)$ can be obtained by retracing steps in Algorithm 1 to extract a local design and correlation parameter (via `aGP.seq`), parallelized for many x. Finally, using saved $D^{\hat{b}}_{N_F}$ and $\hat{\theta}$ from the optimization, or quickly re-computing them via `discrep.est`, it builds a predictor for the field at new x locations. Emulations and biases are thus combined into distribution for the $y^F(x)|u^*$, a sum of Student-t random vari-

ables for $\hat{y}^M(x,u)$ and $\hat{b}(x)$ comprising $y^F(x)|u^*$. However, if $N_F, n_M \geq 30$ summing normals suffices.

4.3. An illustrative example

Consider the following computer model test function used by Goh, Bingham, Holloway, Grosskopf, Kuranz, and Rutter (2013), which is an elaboration of one first described by Bastos and O'Hagan (2009).

```
R> M <- function(x,u)
+   {
+     x <- as.matrix(x)
+     u <- as.matrix(u)
+     out <- (1-exp(-1/(2*x[,2])))
+     out <- out * (1000*u[,1]*x[,1]^3+1900*x[,1]^2+2092*x[,1]+60)
+     out <- out / (100*u[,2]*x[,1]^3+500*x[,1]^2+4*x[,1]+20)
+     return(out)
+   }
```

Goh *et˜al.* paired this with the following discrepancy function to simulate real data under a process like (9).

```
R> bias <- function(x)
+   {
+     x<-as.matrix(x)
+     out<- 2*(10*x[,1]^2+4*x[,2]^2) / (50*x[,1]*x[,2]+10)
+     return(out)
+   }
```

Data coming from the "real" process is simulated under a true (but unknown) u-value, and then augmented with bias and noise processes as follows.

```
R> library(tgp)  ## for lhs sampling in non-unit rectangle
R> rect <- matrix(rep(0:1, 4), ncol=2, byrow=2)
R> ny <- 50
R> X <- lhs(ny, rect[1:2,])
R> u <- c(0.2, 0.1)
R> Zu <- M(X, matrix(u, nrow=1))
R> sd <- 0.5
R> ## Y <- computer output + bias + noise
R> reps <- 2
R> Y <- rep(Zu,reps) + rep(bias(X),reps) + rnorm(reps*length(Zu), sd=sd)
```

The code uses `Y` denote field data observations $Y^F_{N_F}$ with $N_M =$˜`ny`˜$=$˜50, which stores two replicates at each $X^F_{N_F} =$ `X` location. Gramacy *et˜al.* (2014a) illustrated this example with ten replicates. We keep it smaller here for faster execution in live demonstration.

The computer model runs are generated as follows

```
R> nz <- 10000
R> XU <- lhs(nz, rect)
R> XU2 <- matrix(NA, nrow=10*ny, ncol=4)
R> for(i in 1:10) {
+   I <- ((i-1)*ny+1):(ny*i)
+   XU2[I,1:2] <- X
+ }
R> XU2[,3:4] <- lhs(10*ny, rect[3:4,])
R> XU <- rbind(XU, XU2)
R> Z <- M(XU[,1:2], XU[,3:4])
```

Observe that the design $X^M_{N_M} =$ `~XU` is a large LHS in four dimensions, i.e., over design and calibration parameters jointly, augmented with ten-fold replicated field design inputs paired with LHS u-values. This recognizes that it is sensible to run the computer model at inputs where field runs have been observed. `Z` is used to denote $Y^M_{N_M}$.

The following block sets priors and specifies details of the model(s) to be estimated.

```
R> bias.est <- TRUE
R> methods <- rep("alc", 2)
R> da <- d <- darg(NULL, XU)
R> g <- garg(list(mle=TRUE), Y)
```

Changing `bias.est = FALSE` will cause estimation of bias $\hat{b}(\cdot)$ to be skipped, and instead only the level of noise between computer model and field data is estimated. The `methods` vector specifies the nature of search and number of passes through the data for local design and inference. Finally `da`, `d` and `g` contain default priors for the lengthscale of the computer model emulator, the and the bias parameters respectively. The prior is completed with a (log) prior density on the calibration parameter, u, which we choose to be independent Beta with a mode in the middle of the space.

```
R> beta.prior <- function(u, a=2, b=2, log=TRUE)
+ {
+   if(length(a) == 1) a <- rep(a, length(u))
+   else if(length(a) != length(u)) stop("length(a) must be 1 or length(u)")
+   if(length(b) == 1) b <- rep(b, length(u))
+   else if(length(b) != length(u)) stop("length(b) must be 1 or length(u)")
+   if(log) return(sum(dbeta(u, a, b, log=TRUE)))
+   else return(prod(dbeta(u, a, b, log=FALSE)))
+ }
```

Now we are ready to evaluate the objective function on a "grid" to search for a good initialization for a more penetrating search by **NOMAD**. The following code builds the "grid" via a space-filling design on a slightly smaller domain than the input space allows. Experience suggests that initializing too close to the boundary of the input space leads to poor performance in **NOMAD** searches.

```
R> initsize <- 10*ncol(X)
R> imesh <- 0.1
```

```
R> irect <- rect[1:2,]
R> irect[,1] <- irect[,1] + imesh/2
R> irect[,2] <- irect[,2] - imesh/2
R> uinit.cand <- lhs(10*initsize, irect)
R> uinit <- dopt.gp(initsize, Xcand=lhs(10*initsize, irect))$XX
R> llinit <- rep(NA, nrow(uinit))
R> for(i in 1:nrow(uinit)) {
+   llinit[i] <- fcalib(uinit[i,], XU, Z, X, Y, da, d, g, beta.prior,
+                       methods, M, bias.est, nth, verb=0)
+ }
```

By default, `fcalib` echoes the input and calculated objective value (log likelihood or posterior probability) to the screen. This can be useful for tracking progress for an optimization, say via **NOMAD**, however we suppress this prevent clutter. The `fcalib` function has an argument called `save.global` which (when not `FALSE`) causes the information which would be printed to the screen to be saved in a global variable called `fcalib.save` in the environment indicated (e.g., `save.global = .GlobalEnv`), which can be useful for visualization once the optimization has completed. That flag isn't engaged above, since the required quantities, `uinit` and `llinit` respectively, are already in hand. We will, however, utilize this feature below as `snomadr` does not provide an alternative mechanism for saving progress information for later inspection.

The next code chunk loads the **crs** library which contains `snomadr`, the R interface to **NOMAD**, and then creates a list of options that are passed to **NOMAD** via `snomadr`.

```
R> library(crs)
R> opts <- list("MAX_BB_EVAL"=1000, "INITIAL_MESH_SIZE"=imesh,
+   "MIN_POLL_SIZE"="r0.001", "DISPLAY_DEGREE"=0)
```

We have found that these options work well when the input space is scaled to the unit cube. They are derived from defaults recommended in the **NOMAD** documentation.

Now we are ready to invoke `snomadr` on the best input(s) found on grid established above. The code below orders those inputs by their objective value, and then loops over them until a minimum number of **NOMAD** iterations has been reached. Usually, this threshold results in just one pass through the `while` loop, however it offers some robustness in the face of occasional pre-mature convergence. In practice it may be sensible to perform a more exhaustive search if computational resources are abundant.

```
R> its <- 0
R> o <- order(llinit)
R> i <- 1
R> out <- NULL
R> while(its < 10) {
+   outi <- snomadr(fcalib, 2, c(0,0), 0, x0=uinit[o[i],],
+             lb=c(0,0), ub=c(1,1), opts=opts, XU=XU,
+             Z=Z, X=X, Y=Y, da=da, d=d, g=g, methods=methods, M=M,
+             bias=bias.est, omp.threads=nth, uprior=beta.prior,
+             save.global=.GlobalEnv, verb=0)
```

```
+   its <- its + outi$iterations
+   if(is.null(out) || outi$objective < out$objective) out <- outi
+   i <- i + 1;
+ }

iterations: 11
time:       106
```

From the two major chunks of code above, we collect evaluations of `fcalib`, combining a space-filling set of `u`-values and onces placed along stencils in search of the `u`-value which maximizes the likelihood (or posterior probability). In this 2-d problem, that's enough to get good resolution on the log likelihood/posterior surface in `u`. The code below discards any input pairs that are not finite. Infinite values result when **NOMAD** tries input settings that lie exactly on the bounding box.

```
R> Xp <- rbind(uinit, as.matrix(fcalib.save[,1:2]))
R> Zp <- c(-llinit, fcalib.save[,3])
R> wi <- which(!is.finite(Zp))
R> if(length(wi) > 0) { Xp <- Xp[-wi,]; Zp <- Zp[-wi]}
R> surf <- interp(Xp[,1], Xp[,2], Zp, duplicate="mean")
```

Figure 12 shows an image plot of the surface, with lighter- colored values indicating a larger value of likelihood/posterior probability. The initialization points (open circles), evaluations along the **NOMAD** search (black dots), and the ultimate value found in optimization (green dot) are also shown.

Observe, by comparing to the true `u`-value (cross-hairs), that the `u.hat` value we found is far from the value that generated the data. In fact, while the surface is fairly peaked around the best `u.hat`-value that we found, it gives very little support to the true value. Since there are were far fewer evaluations made near the true value, it is worth checking if the solver missed an area of high likelihood/probability.

```
R> Xu <- cbind(X, matrix(rep(u, ny), ncol=2, byrow=TRUE))
R> Mhat.u <- aGP.seq(XU, Z, Xu, da, methods, ncalib=2, omp.threads=nth, verb=0)
R> cmle.u <- discrep.est(X, Y, Mhat.u$mean, d, g, bias.est, FALSE)
R> cmle.u$ll <- cmle.u$ll + beta.prior(u)
```

Comparing log likelihood/posterior probabilities yields:

```
R> data.frame(u.hat=-outi$objective, u=cmle.u$ll)

      u.hat         u
1 -130.6817 -134.7703
```

Well that's reassuring in some ways—the optimization part is performing well—but not in others. Perhaps modeling apparatus introduces some identification issues that prevent recovering the data-generating `u`-value by maximizing likelihood/posterior probability.

Before searching for an explanation, lets check predictive accuracy in the field on a holdout set, again pitting the true `u`-value against our `u.hat`. We first create a random testing design and set aside the true predicted values on those inputs for later comparison.

```
R> image(surf, xlab="u1", ylab="u2", main="posterior surface",
+   col=heat.colors(128), xlim=c(0,1), ylim=c(0,1))
R> points(uinit)
R> points(fcalib.save[,1:2], col=3, pch=18)
R> u.hat <- outi$solution
R> points(u.hat[1], u.hat[2], col=4, pch=18)
R> abline(v=u[2], lty=2)
R> abline(h=u[1], lty=2)
```

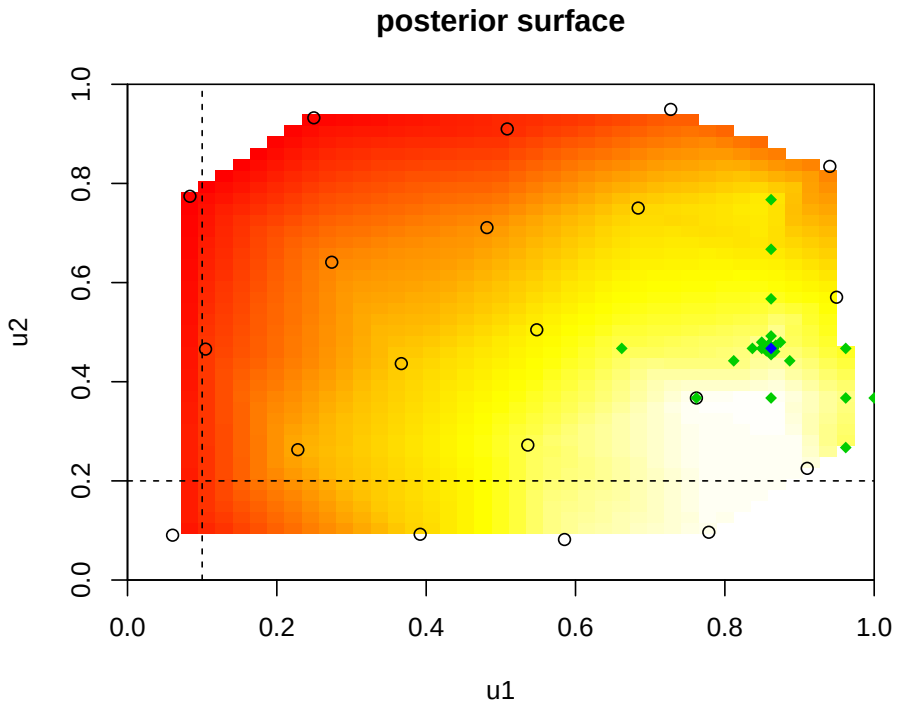

Figure 12: A view of the log likelihood/posterior surface as a function of the calibration inputs, with the optimal `u.hat` value (green dot), the initial grid (open circles) and points of evaluation along the **NOMAD** search (black dots), and the true `u`-value (cross-hairs) shown.

```
R> nny <- 1000
R> XX <- lhs(nny, rect[1:2,],)
R> ZZu <- M(XX, matrix(u, nrow=1))
R> YYtrue <- ZZu + bias(XX)
```

Now we can calculate an out-of-sample RMSE value, first based on the true `u`-value.

```
R> XXu <- cbind(XX, matrix(rep(u, nny), ncol=2, byrow=TRUE))
R> Mhat.oos.u <- aGP.seq(XU, Z, XXu, da, methods, ncalib=2,
+   omp.threads=nth, verb=0)
R> YYm.pred.u <- predGP(cmle.u$gp, XX)
R> YY.pred.u <- YYm.pred.u$mean + Mhat.oos.u$mean
```

```
R> rmse.u <- sqrt(mean((YY.pred.u - YYtrue)^2))
R> deleteGP(cmle.u$gp)
```

Turning to an RMSE calculation using the estimated `u.hat` value, we must re-build some key objects under that value as those objects are not returned to us via either `fcalib` or `snomadr`.

```
R> Xu <- cbind(X, matrix(rep(u.hat, ny), ncol=2, byrow=TRUE))
R> Mhat <- aGP.seq(XU, Z, Xu, da, methods, ncalib=2, omp.threads=nth, verb=0)
R> cmle <- discrep.est(X, Y, Mhat$mean, d, g, bias.est, FALSE)
R> cmle$ll <- cmle$ll + beta.prior(u.hat)
```

As a sanity check, it is nice to see that the value of the log likelihood/posterior probability matches with the one we obtained from `snomadr`:

```
R> print(c(cmle$ll, -outi$objective))
```

```
[1] -130.6817 -130.6817
```

Now we can repeat what we did with the true `u`-value with our estimated one `u.hat`.

```
R> XXu <- cbind(XX, matrix(rep(u.hat, nny), ncol=2, byrow=TRUE))
R> Mhat.oos <- aGP.seq(XU, Z, XXu, da, methods, ncalib=2, omp.threads=nth, verb=0)
R> YYm.pred <- predGP(cmle$gp, XX)
R> YY.pred <- YYm.pred$mean + Mhat.oos$mean
R> rmse <- sqrt(mean((YY.pred - YYtrue)^2))
```

Wrapping up the comparison, we obtain the following:

```
R> data.frame(u.hat=rmse, u=rmse.u)
```

```
      u.hat         u
1 0.1390981 0.1549145
```

Indeed, our estimated `u.hat`-value leads to better predictions of the field data out-of-sample. Gramacy *et˜al.* (2014a) offer an explanation. The KOH model is, with GPs for emulation and bias, overly flexible and consequently challenges identification of the unknown parameters. Authors have commented on this before, including KOH to a limited extent. Interlocking GP predictors (Bah and Joseph 2012) and the introduction of auxiliary inputs (Bornn, Shaddick, and Zidek 2012), of which the u-values are an example, have recently been proposed as deliberate mechanisms for handling non-stationary features in response surface models, particularly for computer experiments. The KOH framework combines both, and predates those works by more than a decade, so in some sense the model being fit is leveraging tools designed for flexibility in response surface modeling, possibly at the expense of being faithful to the underlying meanings of parameters like u and bias processes $b(\cdot)$. In any event, we draw comfort from evidence that the method yields accurate predictions, which in most calibration applications is the primary aim.

5. Ongoing development and extensions

The **laGP** package is under active development, and the corpus of code was developed with ease of extension in mind. The calibration application from Section 4 is a perfect example: simple functions tap into local GP emulators and full GP discrepancies alike, and are paired with existing direct optimizing subroutines from other packages for a powerful solution to large scale calibration problems that are becoming commonplace in the recent literature.

The library comprises of roughly fifty R functions, although barely a fraction of those are elevated to the user's namespace for use in a typical R session. Many of the inaccesible/undocumented functions have a purpose which, at this time, seem less directly useful outside their calling environment, but may eventually be promoted. Many higher level functions, like `laGP` and `aGP` which access C subroutines, have a development-analog (`laGP.R` and `aGP.R`) which implement similar (usually with identical output, our a superset of output) subroutines entirely in R. These were used as stepping stones in the development of the C versions, however they remain as a window into the inner-workings of the package and as a skeleton for curious users development of new extensions. The local approximate GP methodology is, in a nutshell, just a judicious combination of established subroutines from the recent spatial statistics and computer experiments literature. We hope that exposing those combinations in well-organized code will spur others to take a similar tack in developing their own solutions in novel contexts.

As one example we provide a final illustration, here, of how some of the basic functionality in the package—only utilizing full (non local) GP subroutines—was useful in solving hard blackbox optimization problems under constraints. Gramacy, Gray, Le˜Digabel, Lee, Ranjan, Wells, and Wild (2014) showed how the augmented Lagrangian, an apparatus popular for solving similar constrained optimization problems in the recent literature (see, e.g., Kannan and Wild 2012), could be combined with the method of expected improvement (EI; Jones, Schonlau, and Welch 1998) to solve a particular type of optimization where the objective was known (and in particular was linear), but where the constraints required (potentially expensive) simulation. Searching for an optimal valid setting of the inputs to the blackbox function could be substantially complicated by a difficult-to-map constraint satisfaction boundary.

Consider the following objective and constraint function meeting that description.

```
R> blackbox <- function(x)
+ {
+   f <- sum(x)
+   c1 <- 1.5-x[1]-2*x[2]-0.5*sin(2*pi*(x[1]^2-2*x[2]))
+   c2 <- x[1]^2+x[2]^2-1.5
+   return(list(obj=f, c=c(c1,c2)))
+ }
```

The input space is two-dimensional, and the goal is to find a solution `xstar` in the bounding box, `B`

```
R> B <- matrix(c(rep(0,2),rep(1,2)),ncol=2)
R> B

     [,1] [,2]
```

```
[1,]    0    1
[2,]    0    1
```

for which `all(blackbox(xstar)$c <= 0`, meeting the constraint, yet `blackbox(xstar)$obj` minimized. There are three local minima which satisfy the constraints; for more details see Gramacy *et˜al.* (2014).

The **laGP** package implements a hybrid EI–augmented Lagrangian method, which is described in detail by the above reference, in the function `optim.auglag`.

```
R> out.aug <- optim.auglag(blackbox, B, ab=c(3/2,8), end=100, verb=0)
```

The output object contains information about the inputs that were sent to the blackbox for evaluation, the corresponding outputs obtained in terms of objective and constraints, and also tracks the best value of the objective with a valid input (i.e., the progress).

```
R> out.aug$prog[100]

[1] 0.6023861

R> valid <- apply(out.aug$C, 1, function(x) { all(x <= 0) })
R> m <- which.min(out.aug$obj[valid])
R> (out.aug$obj[valid])[m]

[1] 0.6023861

R> (out.aug$X[valid,])[m,]

[1] 0.2001098 0.4022763
```

The algorithm is initialized stochastically, and in 99% of restarts it finds the correct global (valid) minimum (0.6) by the end of the budget of 100 blackbox evaluations. Inspecting the code in `optim.auglag` reveals several useful GP-related functions provided in the package, such as `updateGP`, `rbetter`, `alGP`.

By way of comparison, consider deploying the method of simulated annealing (SA; Kirkpatrick, Gelatt, and Vecci 1983) as implemented by `method="SANN"` in the `optim` function for R. The `optim` function only supports box constraints, so we cannot provide the `blackbox` function directly. To convert a mixed objective and constraint function into a workable hybrid objective, we deploy the additive penalty method (APM).

```
R> blackbox.apm <- function(x, B=matrix(c(rep(0,2),rep(1,2)),ncol=2))
+ {
+   ## check bounding box
+   for(i in 1:length(x)) {
+     if(x[i] < B[i,1] || x[i] > B[i,2]) return(Inf)
+   }
+
```

```
+   ## evaluate objective and constraints
+   f <- sum(x)
+   c1 <- 1.5-x[1]-2*x[2]-0.5*sin(2*pi*(x[1]^2-2*x[2]))
+   c2 <- x[1]^2+x[2]^2-1.5
+
+   ## return APM composite
+   return(f + abs(c1) + abs(c2))
+ }
```

Now we are ready to optimize with SA.

```
R> out.sann <- optim(runif(2), blackbox.apm, method="SANN")
```

The solution returned is:

```
R> out.sann

$par
[1] 0.9973219 0.2453480

$value
[1] 1.712229

$counts
function gradient
   10000       NA

$convergence
[1] 0

$message
NULL
```

Note that the final value typically does not satisfy the constraints as the search typically approaches local optima from the invalid side of the boundary. If the final solution isn't valid, it is usually very close to the boundary.

```
R> blackbox(out.sann$par)

$obj
[1] 1.24267

$c
[1]  0.02440597 -0.44515341
```

The objective value found is near 0.6 only about 60% of the time in repeated random restarts despite performing hundreds of blackbox evaluations.

A. Custom compilation

Here we provide hints for enabling the parallelization hooks, via **OpenMP** for multi-core machines and CUDA for graphics cards. The package also includes some wrapper functions, like `aGP.parallel`, which allow a large predictive set to be divvied up amongst multiple nodes in a cluster established via the **parallel** or **snow** packages.

A.1. With OpenMP for SMP parallelization

Several routines in the **laGP** package include support for parallelization on multi-core machines. The most important one is `aGP` which allows large prediction problems to be divvied up and distributed across multiple threads to be run in parallel. The speedups are roughly linear as long as the numbers of threads is less than or equal to the number of cores. This is controlled through the `omp.threads` argument.

If R is compiled with **OpenMP** support enabled—which at the time of writing standard in most builds—-then no special action is needed in order to extend that functionality to **laGP**. It will just work. One way to check if this is the case on your machine is to provide an `omp.threads` argument, say to `aGP`, which is bigger than one. If **OpenMP** support is not enabled then you will get a warning. If you are working within a well-managed supercomputing facility, with a custom R compilation, it is likely that R has been properly compiled with **OpenMP** support. If not, perhaps it is worth requesting that it be re-compiled as there are many benefits to doing so, beyond those which extend to the **laGP** package. For example, many linear algebra intensive packages, of which **laGP** is one, benefit from liking to MKL libraries from Intel, in addition to **OpenMP**.

In the case where you are using a standard R binary, it is still possible to compile **laGP** from source with **OpenMP** features assuming your compiler (e.g., GCC) supports them. This is a worthwhile step if you are working on a multi-core machine, which is rapidly becoming the standard setup for desktops and laptops alike. For those with experience compiling R packages from source, the procedure is quite straightforward and does not require compiling or installing a bespoke version of R. Obtain the package source (e.g., from CRAN) and, before compiling, open up the package and make two small edits to laGP/src/Makevars. These instructions assume a GCC compiler. For other compilers, please consult documentation for appropriate flags.

1. Replace `$(SHLIB_OPENMP_CFLAGS)` in the `PKG_CFLAGS` line with `-fopenmp`.
2. Replace `$(SHLIB_OPENMP_CFLAGS)` in the `PKG_LIBS` line with `-lgomp`

The laGP/src/Makevars file contains commented out lines which implement these changes. Once made, simply install the package as usual, either doing "R CMD INSTALL" on the modified directory our, or after re-tarring it up. Note that for Apple machines as of Xcode v5, with OSX Mavericks, the `Clang` compiler provided by Apple does not include OpenMP support. We suggest downloading GCC v9 or later, for example from `http://hpc.sourceforge.net`, and following the instructions therein.

If hyperthreading is enabled, then a good default for `omp.threads` is two-times the number of cores. Choosing a `omp.threads` value which is greater than the max allowed by the **OpenMP**

configuration on your machine leads to a notice being printed indicating that the max-value will be used instead.

A.2. With NVIDIA CUDA GPU support

The package supports graphics card acceleration of a key subroutine: searching for the next local design sight x_{j+1} over a potentially vast number of candidates $X_N \setminus X_n(x)$—Step 2(b) in Figure 7. Custom complication is required to enable this feature, the details of which are described here, and also requires a properly configured Nvidia Graphics card, drivers, and compilation programs (e.g., the Nvidia CUDA compiler `nvcc`). Compiling and linking to CUDA libraries can be highly archictecture and operating system specific, therefore the very basic instructures here may not work widely. They have been tested on a varity of Unix-alikes including Intel-based Ubuntu Linux and OSX systems.

First compule the `alc_gpu.cu` file into an object using the Nvidia CUDA compler. E.g., after untarring the packge change into `laGP/src` and do

```
% nvcc -arch=sm_20 -c -Xcompiler -fPIC alc_gpu.cu -o alc_gpu.o
```

Alternatively, you can use/edit the "`alc_gpu.o:`" definition in the `Makefile` provided.

Then, make the following changes to `laGP/src/Makevars`, possibly augmenting changes made above to accommodate **OpenMP** support, as described above. **OpenMP** (i.e., using multiple CPU threads) brings out the best in our GPU implementation.

1. Add `-D_GPU` to the `PKG_FLAGS`
2. Add `alc_gpu.o -L /software/cuda-5.0-el6-x86_64/lib64 -lcudart` to the `PKG_LIBS`. Please replace "`/software/cuda-5.0-el6-x86_64/lib64`" with the path to the CUDA libs on your machine. CUDA 4.x has also been tested.

The `laGP/src/Makvars` file contains commented out lines which implement these changes. Once made, simply install the package as usual. Alternatively, use `make allgpu` to via the definitions in the `Makefile` to compile a standalone shared object.

The four functions in the package with GPU support are `alcGP`, `laGP`, `aGP`, and `aGP.parallel`. The first two have a simple switch which allows a single search (Step 2(b)) to be off-loaded to a single GPU. Both also support off-loading the same calculations to multiple cores in a CPU, via `OpenMP` if enabled. The latter `aGP` variations control the GPU interface via two arguments: `num.gpus` and `gpu.threads`. The former specifies how many GPUs you wish to use, and indicating more you actually have will trip an error. The latter, which defaults to `gpu.threads = num.gpus`, specifies how many CPU threads should be used to queue GPU jobs. Having `gpu.threads < num.gpus` is an inefficient use of resources, whereas `gpu.threads > num.gpus`, up to `2*num.gpus` will give modest speedups. Having multiple threads queue onto the same GPU reduces the amount of time the GPU is idle. `OpenMP` support must be included in the package to have more than one GPU thread.

By default, `omp.threads` is set to zero when `num.gpus > 1` since divvying the work amongst GPU and CPU threads can present load balancing challenges. However, if you get the load balancing right you can observe substantial speedups. Gramacy *et˜al.* (2014b) observe up to 50% speedups, and recommend a scheme for allocating `omp.threads=10` with a setting of

`nn.gpu` that allocates about 90% of the work to GPUs (`nn.gpu = floor(0.9*nrow(XX))`) and 10% to the ten **OpenMP** threads. As with `omp.threads`, `gpu.threads` maxes out at the maximum number of threads indicated by your **OpenMP** configuration. Moreover, `omp.threads + gpu.threads` must not exceed that value. When that happens both are first thresholded independently, then `omp.threads` may be further reduced to stay within the limit.

References

Audet C, Dennis, Jr J (2006). "Mesh Adaptive Direct Search Algorithms for Constrained Optimization." *SIAM Journal on Optimization*, **17**(1), 188–217. doi:doi:10.1137/040603371. URL http://dx.doi.org/doi:10.1137/040603371.

Bah S, Joseph V (2012). "Composite Gaussian process models for emulating expensive functions." *Annals of Applied Statistics*, **6**(4), 1838–1860.

Bastos L, O'Hagan A (2009). "Diagnostics for Gaussian Process Emulators." *Technometrics*, **51**(4), 425–438.

Berger J, De˜Oliveira V, Sanso B (2001). "Objective Bayesian Analysis of Spatially Correlated Data." *Journal of the American Statistical Association*, **96**, 1361–1374.

Bornn L, Shaddick G, Zidek J (2012). "Modelling Nonstationary Processes Through Dimension Expansion." *Journal of the American Statistical Association*, **107**(497), 281–289.

Brent R (1973). *Algorithm for Minimization without Derivatives*. Prentice–Hall, Englewood Cliffs, N.J.

Chipman H, Ranjan P, Wang W (2012). "Sequential Design for Computer Experiments with a Flexible Bayesian Additive Model." *Technical report*, Acadia University. ArXiv:1203.1078.

Cohn DA (1996). "Neural Network Exploration using Optimal Experimental Design." In *Advances in Neural Information Processing Systems*, volume 6(9), pp. 679–686. Morgan Kaufmann Publishers.

Conn A, Scheinberg K, Vicente L (2009). *Introduction to Derivative-Free Optimization*. SIAM, Philadelphia.

Cressie N (1993). *Statistics for Spatial Data,* revised edition. John Wiley and Sons, Inc.

Cressie N, Johannesson G (2008). "Fixed Rank Kriging for Very Large Data Sets." *Journal of the Royal Statistical Soceity, Series B*, **70**(1), 209–226.

Eidsvik J, Shaby BA, Reich BJ, Wheeler M, Niemi J (2014). "Estimation and prediction in spatial models with block composite likelihoods." *Journal of Computational and Graphical Statistics*, **23**(2), 295–315. doi:10.1080/10618600.2012.760460.

Emory X (2009). "The kriging update equations and their application to the selection of neighboring data." *Computational Geosciences*, **13**(3), 269–280.

Franey M, Ranjan P, Chipman H (2012). "A Short Note on Gaussian Process Modeling for Large Datasets using Graphics Processing Units." *Technical report*, Acadia University.

Goh J, Bingham D, Holloway JP, Grosskopf MJ, Kuranz CC, Rutter E (2013). "Prediction and Computer Model Calibration Using Outputs From Multi-fidelity Simulators." *Technometrics. to appear*.

Gramacy R, Bingham D, Holloway JP, Grosskopf MJ, Kuranz CC, Rutter E, Trantham M, Drake PR (2014a). "Calibrating a large computer experiment simulating radiative shock hydrodynamics." *Technical report*, The University of Chicago. ArXiv:1410.3293.

Gramacy R, Haaland B (2014). "Speeding up neighborhood search in local Gaussian process prediction." *Technical report*, The University of Chicago. ArXiv:1409.0074.

Gramacy R, Lee H (2011). "Cases for the nugget in modeling computer experiments." *Statistics and Computing*, **22**(3).

Gramacy R, Niemi J, Weiss R (2014b). "Massively Parallel Approximate Gaussian Process Regression." *Journal of Uncertainty Quantification*, ***to appear***. See arXiv:1310.5182.

Gramacy R, Polson N (2011). "Particle Learning of Gaussian Process Models for Sequential Design and Optimization." *Journal of Computational and Graphical Statistics*, **20**(1), 102–118. doi:10.1198/jcgs.2010.09171.

Gramacy R, Taddy M, Wild S (2013). "Variable Selection and Sensitivity Analysis via Dynamic Trees with an Application to Computer Code Performance Tuning." *Annals of Applied Statistics*, **7**, 51–80. doi:10.1214/12-AOAS590.

Gramacy RB, Apley DW (2014). "Local Gaussian process approximation for large computer experiments." *Journal of Computational and Graphical Statistics. to appear*; see arXiv:1303.0383.

Gramacy RB, Gray G, Le˜Digabel S, Lee H, Ranjan P, Wells G, Wild S (2014). "Modeling an Augmented Lagrangian for Improved Blackbox Constrained Optimization." *Technical report*, The University of Chicago. ArXiv:1403.4890.

Gramacy RB, Lee HKH (2009). "Adaptive Design and Analysis of Supercomputer Experiments." *Technometrics*, **51**(2), 130–145.

Haaland B, Qian P (2011). "Accurate Emulators for Large-Scale Computer Experiments." *Annals of Statistics*, **39**(6), 2974–3002. doi:10.1214/11-AOS929.

Higdon D, Kennedy M, Cavendish JC, Cafeo JA, Ryne RD (2004). "Combining field data and computer simulations for calibration and prediction." *SIAM Journal on Scientific Computing*, **26**(2), 448–466.

Jones DR, Schonlau M, Welch WJ (1998). "Efficient Global Optimization of Expensive Black Box Functions." *J. ˜of Global Optimization*, **13**, 455–492.

Joseph V (2006). "Limit Kriging." *Technometrics*, **48**(4), 548–466.

Kannan A, Wild SM (2012). "Benefits of Deeper Analysis in Simulation-based Groundwater Optimization Problems." In *Proceedings of the XIX International Conference on Computational Methods in Water Resources (CMWR 2012)*.

Kaufman C, Bingham D, Habib S, Heitmann K, Frieman J (2012). "Efficient Emulators of Computer Experiments Using Compactly Supported Correlation Functions, With An Application to Cosmology." *Annals of Applied Statistics*, **5**(4), 2470–2492.

Kennedy M, O'Hagan A (2001). "Bayesian Calibration of Computer Models (with discussion)." *Journal of the Royal Statistical Society, Series B*, **63**, 425–464.

Kirkpatrick S, Gelatt CD, Vecci MP (1983). "Optimization by simulated annealing." *Science*, **220**(671–680).

Le˜Digabel S (2011). "Algorithm 909: NOMAD: Nonlinear Optimization with the MADS algorithm." *ACM Transactions on Mathematical Software*, **37**(4), 44:1–44:15. doi:10.1145/1916461.1916468. URL http://dx.doi.org/10.1145/1916461.1916468.

Liu F, Bayarri M, Berger J (2009). "Modularization in Bayesian analysis, with emphasis on analysis of computer models." *Bayesian Analysis*, **4**(1), 119–150.

MacDoanld B, Chipman H, Ranjan P (2014). *GPfit: Gaussian Processes Modeling*. R package version 0.2-0, URL http://CRAN.R-project.org/package=GPfit.

Matheron G (1963). "Principles of Geostatistics." *Economic Geology*, **58**, 1246–1266.

McKay M, Conover W, Beckman R (1979). "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code." *Technometrics*, **21**(2), 239–245. URL http://www.jstor.org/stable/1268522.

Morris D, Mitchell T, Ylvisaker D (1993). "Bayesian Design and Analysis of Computer Experiments: Use of Derivatives in Surface Prediction." *Technometrics*, **35**, 243–255.

Paciorek C, Lipshitz B, Zhuo W, Prabhat, Kaufman C, Thomas R (2013). "Parallelizing Gaussian Process Calculations in R." *Technical report*, University of California, Berkeley. ArXiv:1303.0383.

Pratola MT, Chipman H, Gattiker J, Higdon D, McCulloch R, Rust W (2013). "Parallel Bayesian Additive Regression Trees." *Journal of Computational and Graphical Statistics. To appear*.

Racine JS, Nie Z (2012). `crs`*: Categorical regression splines*. R package version 0.15-18, URL https://github.com/JeffreyRacine/R-Package-crs/.

Ranjan P, Haynes R, Karsten R (2011). "A Computationally Stable Approach to Gaussian Process Interpolation of Deterministic Computer Simulation Data." *Technometrics*, **53**(4), 363–378.

Rasmussen CE, Williams CKI (2006). *Gaussian Processes for Machine Learning*. The MIT Press.

Sacks J, Welch WJ, Mitchell TJ, Wynn HP (1989). "Design and Analysis of Computer Experiments." *Statistical Science*, **4**, 409–435.

Sang H, Huang JZ (2012). "A Full Scale Approximation of Covariance Functions for Large Spatial Data Sets." *Journal of the Royal Statistical Society: Series B*, **74**(1), 111–132. ISSN 1467-9868. doi:10.1111/j.1467-9868.2011.01007.x.

Santner TJ, Williams BJ, Notz WI (2003). *The Design and Analysis of Computer Experiments*. Springer-Verlag, New York, NY.

Schmidt AM, O'Hagan A (2003). "Bayesian Inference for Nonstationary Spatial Covariance Structure via Spatial Deformations." *Journal of the Royal Statistical Society, Series B*, **65**, 745–758.

Silverman BW (1985). "Some Aspects of the Spline Smoothing Approach to Non-Parametric Curve Fitting." *Journal of the Royal Statistical Society Series B*, **47**, 1–52.

Snelson E, Ghahramani Z (2006). "Sparse Gaussian Processes using Pseudo-inputs." In *Advances in Neural Information Processing Systems*, pp. 1257–1264. MIT press.

Stein ML (1999). *Interpolation of Spatial Data*. Springer, New York, NY.

Stein ML, Chi Z, Welty LJ (2004). "Approximating Likelihoods for Large Spatial Data Sets." *Journal of the Royal Statistical Society, Series B*, **66**(2), 275–296.

Vecchia A (1988). "Estimation and model identification for continuous spatial processes." *Journal of the Royal Statistical Soceity, Series B*, **50**, 297–312.

Worley B (1987). "Deterministic Uncertainty Analysis." *Technical Report ORN-0628*, National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA.

Affiliation:

Robert B. Gramacy
Booth School of Business
The University of Chicago
5807 S.˜Woodlawn Ave
Chicago, IL 60605 USA
E-mail: rbgramacy@chicagobooth.edu
URL: http://faculty.chicagobooth.edu/robert.gramacy/