

# GP Fidelity & Scale

## *RSMs and Computer Experiments*

Robert B. Gramacy ([rbg@vt.edu](mailto:rbg@vt.edu) (<mailto:rbg@vt.edu>) : <http://bobby.gramacy.com> (<http://bobby.gramacy.com>))  
Department of Statistics, Virginia Tech

## Goals

GPs are fantastic, but they are not without their drawbacks.

- Computational complexity is one:  $\mathcal{O}(n^3)$  matrix decompositions and  $\mathcal{O}(n^2)$  storage can severely limit data sizes.

Flexibility is another.

- Stationarity is a nice simplifying assumption,
- but it is clearly not appropriate for all data generating mechanisms;
- e.g., the LGBB rocket booster data.

In this segment we'll try to address those issues, ideally *simultaneously*.

- The literature on GP approximation is booming: approximation and sparsity are common themes.
- The literature on nonstationary modeling is more niche.

## Sparse Covariance

## Big data remedies

Inroads into faster GP modeling are being made from a number of angles, usually by approximation.

- "Pseudo inputs" (Snelson & Ghahramani, 2006) ([http://www.gatsby.ucl.ac.uk/~snelson/SPGP\\_up.pdf](http://www.gatsby.ucl.ac.uk/~snelson/SPGP_up.pdf))
- Iterating over batches (Haaland & Qian, 2012) (<https://projecteuclid.org/euclid.aos/1327413775>)
- Fixed rank kriging (Cressie & Johannesson, 2008) (<http://onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2007.00633.x/abstract>)
- Compactly supported covariances and fast sparse linear algebra (Kauffman, et al, 2011; (<http://projecteuclid.org/euclid.aos/1324399603>) Sang & Huang, 2011) (<http://onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2011.01007.x/abstract>)
- Treed Gaussian processes (Gramacy & Lee, 2008) (<http://www.tandfonline.com/doi/abs/10.1198/016214508000000689>)
- Composite likelihood (Eidsvik, et al., 2013) (<http://www.tandfonline.com/doi/abs/10.1080/10618600.2012.76046>)
- Local Gaussian Processes (Gramacy & Apley, 2015) (<http://www.tandfonline.com/doi/abs/10.1080/10618600.2014.914442?journalCode=uugs20>)

The trouble is, not many of them come with software.

## Underlying themes

But there are a couple of underlying themes, and representative softwares therein.

- sparse covariances
- partitioning
- local/composite approximation

And in fact all three can be seen as mechanisms for inducing sparsity in the covariance structure.

- But they differ in how they leverage that sparsity to speed up calculations,

- and in how they offer scope for enhanced fidelity.

It is worth noting that you can't just truncate small entries of  $K_{ij}$ ,

- because the result will almost certainly not be positive definite.

## Compactly supported kernels

A kernel  $k_{r_{\max}}(r)$ , where  $r = |x - x'|$ , is said to have **compact support** if  $k_{r_{\max}}(r) = 0$  when  $r > r_{\max}$ .

- We may proceed component-wise with  $r_j = |x_j - x'_j|$  and  $r_{j,\max}$  for a separable (compactly supported) kernel.
- Nuggets are allowed, but we'll drop them here for convenience.
- We'll talk about lengthscales shortly.

A **compactly supported kernel (CSK)** introduces zeros into the the covariance matrix,

- so sparse matrix methods can speed up computations.

## Polynomial basis

Now more sparsity means faster computation,

- but at the expense of “long distance” correlation.

A key idea in Kaufman et al. (2011) (<http://projecteuclid.org/euclid.aoas/1324399603>) is to use a *rich* nonlinear mean function to “mop up” the long range non-linearity,

- leaving the residual to be modeled by shorter-range (sparse) correlations.
- Kauffman, et al., recommend degree 5 “tensor product” Legendre polynomials.
- Inference, integrating out over the “partition” between long and short spatial effects, is carried out by Bayesian MCMC.

And they provide software.

**Library** (SparseEm)

## A big-ish example

Consider the borehole data (<https://www.sfu.ca/~ssurjano/borehole.html>), which is a classic computer experiments example (Morris, et al., 1993) (<http://amstat.tandfonline.com/doi/abs/10.1080/00401706.1993.10485320#.WCKSswsrKfY>).

It is a function of eight inputs, modeling water flow through a borehole.

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

The input ranges are

$$\begin{array}{lll} r_w \in [0.05, 0.15] & r \in [100, 5000] & T_u \in [63070, 115600] \\ T_l \in [63.1, 116] & H_u \in [990, 1110] & H_l \in [700, 820] \\ L \in [1120, 1680] & K_w \in [9855, 12045]. & \end{array}$$

## Borehole simulation

Here is an implementation in coded inputs.

```
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)
}
```

## LHS train/test partition

Here is a LHS training and testing partition.

```
n <- 4000; npred <- 500; dim <- 8
library(lhs)
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,]
```

The `SparseEm` package includes a subroutine to “find” the desired level of sparsity, `mc`.

```
degree <- 2; maxint <- 2
sparsity <- 0.99
mc <- find.tau(den = 1 - sparsity, dim = ncol(x)) * ncol(x)
```

## Sampling from the posterior

Now, lets gather 2000 samples from the posterior,

- saving the compute time for a later comparison.

```
B <- 2000
suppressWarnings({
  time1 <- system.time(
    tau <- mcmc.sparse(y, x, mc=mc, degree=degree, maxint=maxint, B=B, verbose=FALSE))
})
```

And then make predictions on the testing set,

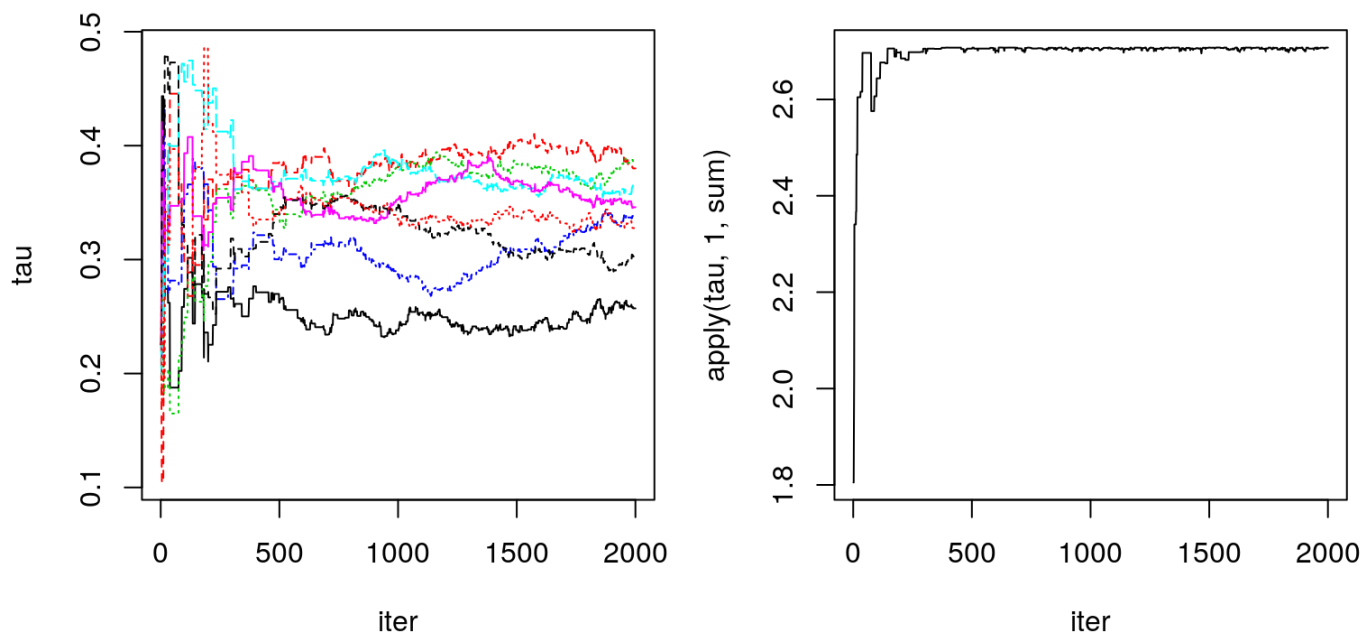
- removing burn-in iterations, and sub-sampling every 10th iteration.

```
burnin <- 500
index <- seq(burnin+1, B, by = 10)
suppressWarnings({
  time2 <- system.time(ypred.sparse <-
    pred.sparse(tau[index,], x, y, xpred, degree=degree, maxint=maxint, verbose=FALSE))
})
```

## Trace plots (sparse)

Pretty good on the convergence front (recall  $\tau$  threshold/lengthscale parameter).

```
par(mfrow=c(1,2)); matplot(tau, type = "l", xlab="iter")
plot(apply(tau, 1, sum), type = "l", xlab="iter")
```



## A fair comparison

Kauffman, et al. provide a non-sparse version,

- that otherwise works identically,
- for timing and accuracy comparisons.

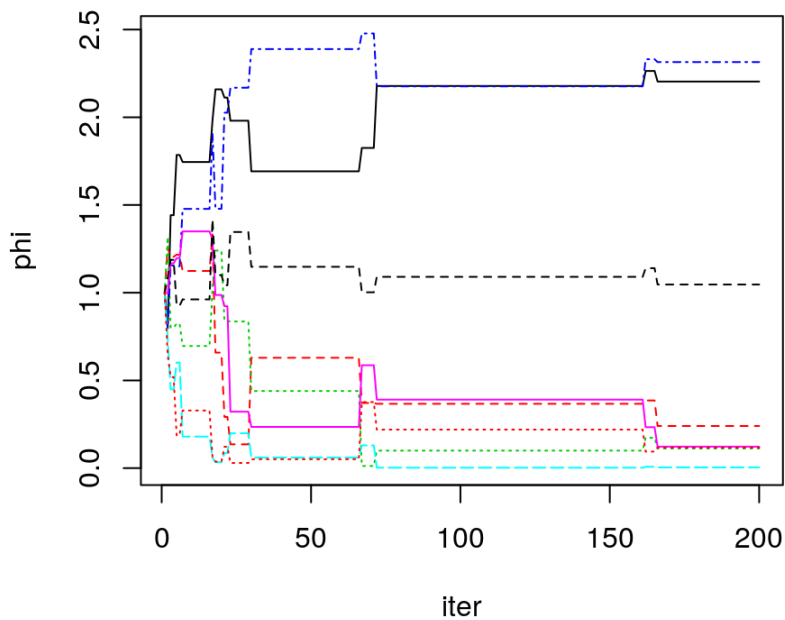
This is going to be really slow, so we'll do an order of magnitude fewer MCMC iterations.

```
B <- 200
suppressWarnings({
time3 <- system.time(phi <- mcmc.nonsparse(y, x, B=B, verbose=FALSE))
burnin <- 50
index <- seq(burnin+1, B, by = 10)
time4 <- system.time(ypred.nonsparse <-
  pred.nonsparse(phi[index,], x, y, xpred, 2, verbose=FALSE))
})
```

## Trace plots (non-sparse)

Not converged, but that's about all we can do in a reasonable amount of time.

```
matplot(phi, type = "l", xlab="iter")
```



## How do they compare?

In terms of time, about a 3× speedup.

- Not super impressive, but good potential.

```
print(times <- c(sparse=as.numeric(time1[3]+time2[3]),
  dense=10*as.numeric(time3[3]+time4[3])))
```

```
##   sparse   dense
## 3457.123 9861.910
```

In terms of accuracy ... (unfair to the dense version since it hasn't burned in)

```
s2s <- ypred.sparse$var; s2n <- ypred.nonsparse$var
print(scores <- c(sparse=mean(- (ypred.sparse$mean - ypred.0)^2/s2s - log(s2s)),
  dense=mean(- (ypred.nonsparse$mean - ypred.0)^2/s2n - log(s2n))))
```

```
##   sparse   dense
## -1.508444 -2.131856
```

## 99.9% sparse?

Lets see how much faster (and how much less accurate) a 99.9% sparse version is.

```

sparsity <- 0.999
mc <- find.tau(den = 1 - sparsity, dim = ncol(x)) * ncol(x)
B <- 2000
suppressWarnings({
time5 <- system.time(
  tau <- mcmc.sparse(y, x, mc=mc, degree=degree, maxint=maxint, B=B, verbose=FALSE))
})
burnin <- 500
index <- seq(burnin+1, B, by = 10)
suppressWarnings({
time6 <- system.time(ypred.sparse <-
  pred.sparse(tau[index,], x, y, xpred, degree=degree, maxint=maxint, verbose=FALSE))
})

```

- Essentially cutting and pasting from above.

## Expanded comparison

Much faster.

```

times <- c(times, s999=as.numeric(time5[3]+time6[3]))
times

```

```

##   sparse   dense   s999
## 3457.123 9861.910 420.356

```

A little less accurate.

```

s2 <- ypred.sparse$var
scores <- c(scores, s999=mean(- (ypred.sparse$mean - ypred.0)^2/s2 - log(s2)))
scores

```

```

##   sparse   dense   s999
## -1.508444 -2.131856 -1.680680

```

## Partition models

### Divvy it up

Another way to induce sparsity in the covariance structure is to

- partition the input space into independent regions.
- Then the covariance is essentially block-diagonal.

The trouble is, its hard to know just how to split things up.

- Ideally, we'd let the data decide.

Once we've figured that out, it makes sense to fit hyperparameters independently in each partition,

- thereby obtaining a cheap non-stationary model.
- But all bets for continuity are off, unless we can average over all (likely) partitions.

### Easy to say ...

... hard to do, especially with Gaussian processes.

I know of only two (successful) attempts.

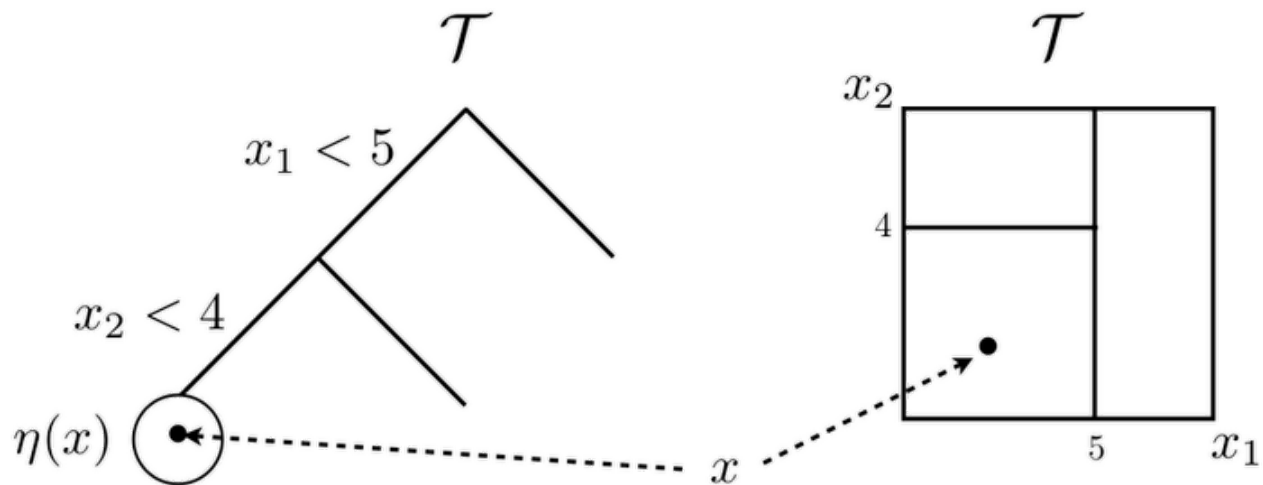
1. via Voronoi tessellations (Kim, et al. ,2012) (<http://amstat.tandfonline.com/doi/abs/10.1198/016214504000002014?journalCode=uasa20>)
2. via trees (Gramacy & Lee, 2008) (<http://www.tandfonline.com/doi/abs/10.1198/016214508000000689>)

Tessellations are easy to characterize mathematically, but a nightmare computationally.

Trees are easy mathematically too,

- and much easier computationally.

## Divide-and-conquer regression



- Recursive, axis-aligned, splits divvy up the input space into independent predictive models for the response ( $y$ ).
- Predictions  $\hat{y} = f_n(x)$  are dictated by **tree structure** and **leaf model**.

The plan is to use GPs at the leaves, but lets take a step back first ...

## Turn back the clock

Use of trees in regression dates back to **AID (Automatic Interaction Detection)** by Morgan and Sonquist (1963) ([https://www.cs.nyu.edu/~roweis/csc2515-2006/readings/morgan\\_sonquist63.pdf](https://www.cs.nyu.edu/~roweis/csc2515-2006/readings/morgan_sonquist63.pdf))

**Classification and Regression Trees (CART)** (Breiman, et al., 1984) (<https://www.amazon.com/Classification-Regression-Wadsworth-Statistics-Probability/dp/0412048418>),

- a suite of methods obtaining fitted partition trees,
- popularized the idea.

The selling point was that

- trees facilitate parsimonious divide-and-conquer, leading to flexible yet *interpretable* modeling.

Fitting a partition structure (depth, splits, etc.) requires

- a leaf prediction rule and goodness-of-fit criteria,
- and a “search” algorithm.

## Likelihood-based modeling

I prefer the likelihood, whenever possible.

Given a particular tree,  $\mathcal{T}$ , the (marginal) likelihood factorizes into a product form.

$$p(y^n | \mathcal{T}, x^n) \equiv p(y_1, \dots, y_n | \mathcal{T}, x_1, \dots, x_n) = \prod_{\eta \in \mathcal{L}_{\mathcal{T}}} p(y^\eta | x^\eta)$$

The simplest leaf model for regression is the “constant model”:

- a “local” Gaussian with unknown mean and variance ( $\theta_\eta = (\mu_\eta, \sigma_\eta^2)$ ).
- Gaussian modeling means a convenient closed form for each  $p(y^\eta | x^\eta)$ .

## Tree prior

Some kind of **regularization** is needed for inference,

- otherwise the product-form likelihood is maximized when there is a leaf for each observation.

Chipman, George & McCulloch (1998) ([https://www.jstor.org/stable/2669832?seq=1#page\\_scan\\_tab\\_contents](https://www.jstor.org/stable/2669832?seq=1#page_scan_tab_contents))

describe how trees may stochastically be grown from a leaf node  $\eta$  with a probability that depends on the depth  $D_\eta$  of that node in the tree,  $\mathcal{T}$ .

$$p_{\text{split}}(\eta, \mathcal{T}) = \alpha(1 + D_\eta)^{-\beta}$$

This induces a prior for the full tree  $\mathcal{T}$  via the probability that internal nodes  $\mathcal{I}_{\mathcal{T}}$  split and leaves  $\mathcal{L}_{\mathcal{T}}$  do not:

$$\pi(\mathcal{T}) \propto \prod_{\eta \in \mathcal{I}_{\mathcal{T}}} p_{\text{split}}(\eta, \mathcal{T}) \prod_{\eta \in \mathcal{L}_{\mathcal{T}}} [1 - p_{\text{split}}(\eta, \mathcal{T})].$$

- Everything else is uniform.

## Inference by MCMC

Inference then proceeds by MCMC.

- Note that there are no parameters except the tree,  $\mathcal{T}$ ,
- when the leaf-node parameters  $\theta_\eta$  are integrated out.

Here is how the MCMC would go.

- Randomly choose part of the tree to alter, and how to alter it (prune, grow, change, swap, rotate, ...).
- Accept the move with Metropolis–Hastings probability:

$$\frac{p(\mathcal{T}' | y^n, x^n)}{p(\mathcal{T} | y^n, x^n)} = \frac{p(y^n | \mathcal{T}', x^n)}{p(y^n | \mathcal{T}, x^n)} \times \frac{\pi(\mathcal{T}')}{\pi(\mathcal{T})}.$$

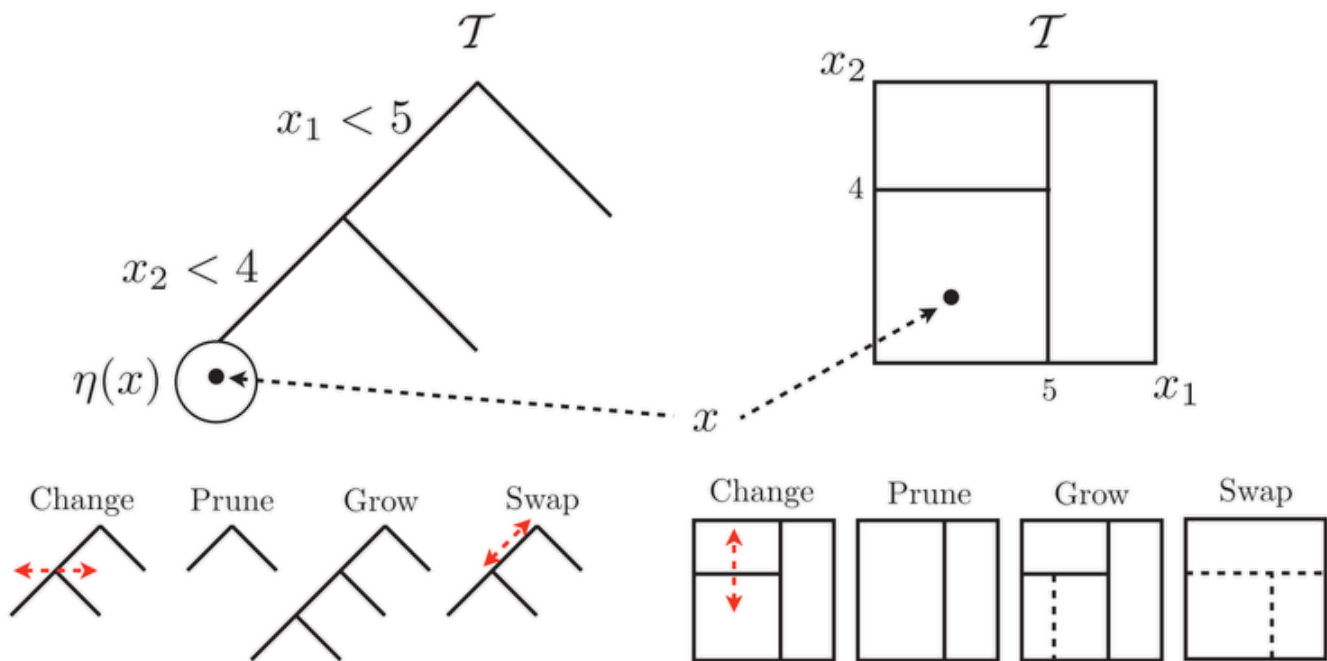
Potential savings comes with local moves in tree space,

- since terms above cancel for the un-altered leaves in the product form of the marginal likelihood.

## Tree proposals

What do tree proposals look like? Here is an example of the four most popular “tree moves”.





## Motorcycle data

The `tgpc` package will sample from the Bayesian treed constant model ("Bayesian CART") posterior.

```
library(tgpc)
```

As an illustration, consider the motorcycle accident data in the `MASS` library for R.

```
library(MASS)
```

Samples from the posterior predictive distribution are gathered at locations `XX`:

```
XX <- seq(0, max(mcycle[,1]), length=1000)
out.bcart <- bcart(X=mcycle[,1], Z=mcycle[,2], XX=XX, R=100, verb=0)
```

- For *maximum a posteriori (MAP)* predictor, you can run the following afterwards.

```
outp.bcart <- predict(out.bcart, XX=XX)
```

## Moto-visualization macro

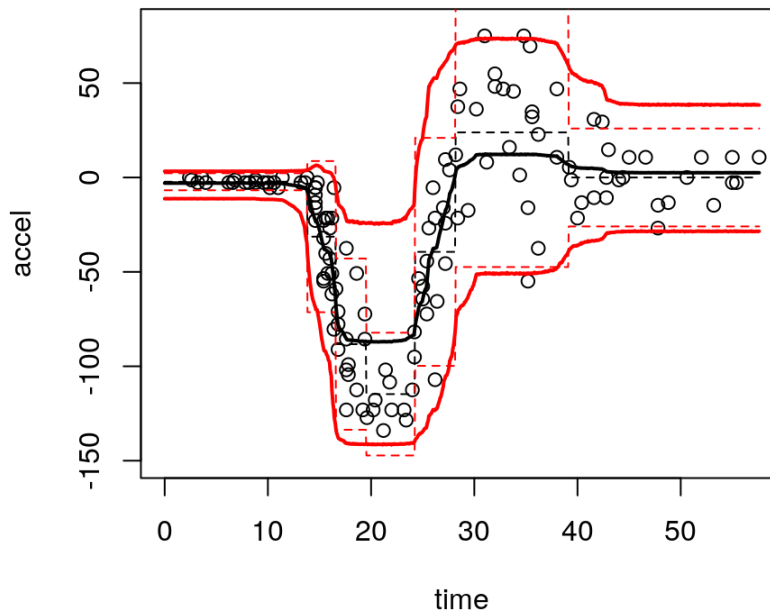
To visualize the predictive surface(s), I'm going to write a little macro that we can re-use in several variations later.

```
plot.moto <- function(out, outp)
{
  plot(outp$XX[,1], outp$ZZ.km, ylab="accel", xlab="time",
       ylim=c(-150, 80), lty=2, col=1, type="l")
  points(mcycle)
  lines(outp$XX[,1], outp$ZZ.km + 1.96*sqrt(outp$ZZ.ks2), col=2, lty=2)
  lines(outp$XX[,1], outp$ZZ.km - 1.96*sqrt(outp$ZZ.ks2), col=2, lty=2)
  lines(outp$XX[,1], outp$ZZ.mean, col=1, lwd=2)
  lines(outp$XX[,1], outp$ZZ.q1, col=2, lwd=2)
  lines(outp$XX[,1], outp$ZZ.q2, col=2, lwd=2)
}
```

# Bayesian CART surface(s)

The MCMC is good at smoothing out rough transitions.

```
plot.moto(out.bcart, outp.bcart)
```



## BCART surface features

What do we get?

- Organic non-stationarity and heteroskedasticity.
- But both are obviously limited. E.g., the variance is too high at the start and the end.

The MAP estimate has many of those features,

- and can be taken as an analogue of the “old CART way”.
- The hard breaks are unappealing.
- The smoothed predictions from the MCMC are a bit better.

Try `tgpl.trees(out.bcart)` to visualize the best trees.

## Bayesian treed linear model

Any data type/leaf model may be used *without extra computational effort* as long as  $p(y^i | x^i)$  is analytic.

- Linear models at the leaves are one example, leading to the so-called **Bayesian treed linear model (BTLM)** (Chipman, et al., 2002) (<http://link.springer.com/article/10.1023/A:1013916107446>) uses

Here is the fit in `tgpl`,

```
out.bt1m <- bt1m(X=mcycle[,1], Z=mcycle[,2], XX=XX, R=100, verb=0)
```

- The MAP predictor can be extracted as follows.

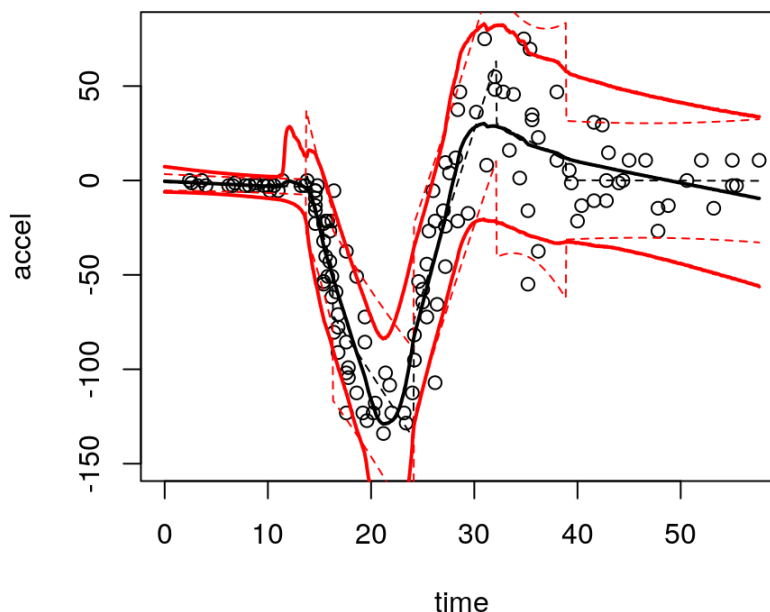
```
outp.bt1m <- predict(out.bt1m, XX=XX)
```

We'll use the plotting macro we made earlier for easy visualization.

## BTLM surface(s)

Fewer partitions, but is it better?

```
plot.moto(out.btlm, outp.btlm)
```



## What else?

If the response is **categorical**,

- then a multinomial leaf model and Dirichlet prior pair leads to an analytic marginal likelihood (Chipman, et al., 1998) ([https://www.jstor.org/stable/2669832?seq=1#page\\_scan\\_tab\\_contents](https://www.jstor.org/stable/2669832?seq=1#page_scan_tab_contents)).

Other members of the exponential family proceed similarly:

- Poisson, exponential, negative binomial, ...
- and others too (potentially).

However, to my knowledge none of these choices have been actually implemented as leaf models in a Bayesian treed regression setting.

## Non-analytic marginal likelihood

Technically, any leaf model can be deployed

- by extending the Monte Carlo to integrate over leaf parameters  $\theta_\eta$  too.
- But deep trees/many leaves could result in a prohibitively large parameter space.

An important exception is GPs.

- GPs offer a parsimonious take on nonlinear nonparametric regression,
- mopping up much of the variability left to the tree with simpler leaf models.

GP leaves encourage shallow trees with fewer leaf nodes.

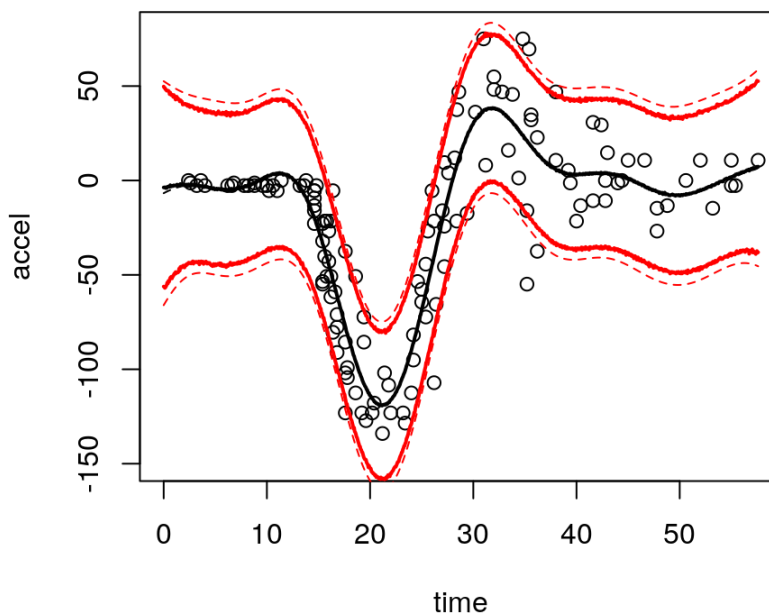
- At the same time, treed partitioning enables (axis aligned) regime changes in stationarity and skedasticity.

First, lets look at a stationary GP fit to the motorcycle data.

## Stationary “BGP” fit

Smooth, but equally as bad as BCART and BTLM in other (opposite) respects?

```
out.bgp <- bgp(X=mcycle[,1], Z=mcycle[,2], XX=XX, R=10, verb=0)
outp.bgp <- predict(out.bgp, XX=XX)
plot.moto(out.bgp, outp.bgp)
```



## Treed Gaussian process

Bayesian **treed Gaussian process (TGP)** models (Gramacy & Lee, 2008)

(<http://www.tandfonline.com/doi/abs/10.1198/016214508000000689>) can offer the best of both worlds, marrying

- the smooth global perspective of GPs,
- with the thrifty local adaptivity of trees.

Their divide-and-conquer nature mean

- faster computation: smaller matrix inversions
- and disparate spatial dependencies.

```
out.btgp <- btgp(X=mcycle[,1], Z=mcycle[,2], XX=XX, R=30, bprior="b0", verb=0)
```

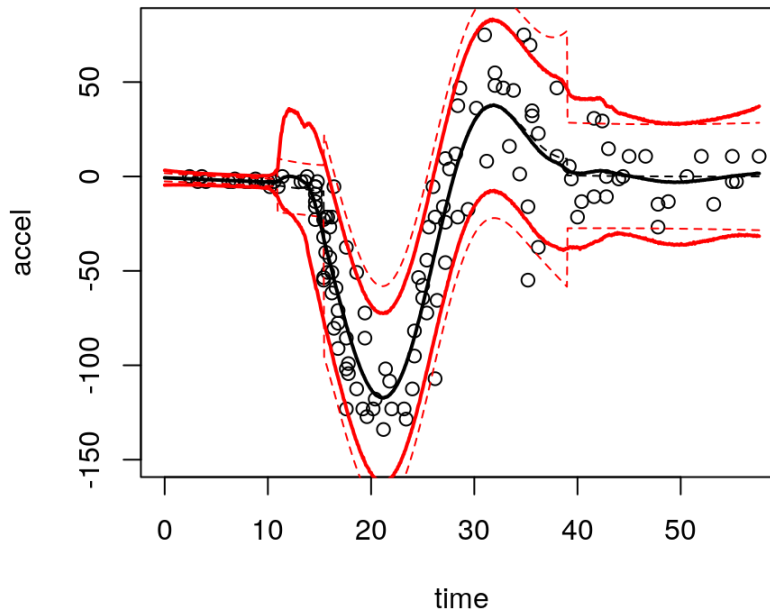
As before, we can extract the MAP predictor, mostly for comparison.

```
outp.btgp <- predict(out.btgp, XX=XX)
```

## BTGP surface(s)

Pretty darn good, if you ask me.

```
plot.moto(out.btgp, outp.btgp)
```



## 2-d surface

Lets revisit the 2-d exponential data,

- which was actually created to showcase subtle nonstationarity with `tgpr`.
- It even comes in the package.

```
exp2d.data <- exp2d.rand(n1=25, n2=75)
X <- exp2d.data$X; Z <- exp2d.data$Z
XX <- exp2d.data$XX
```

First lets fit an ordinary (Bayesian) GP.

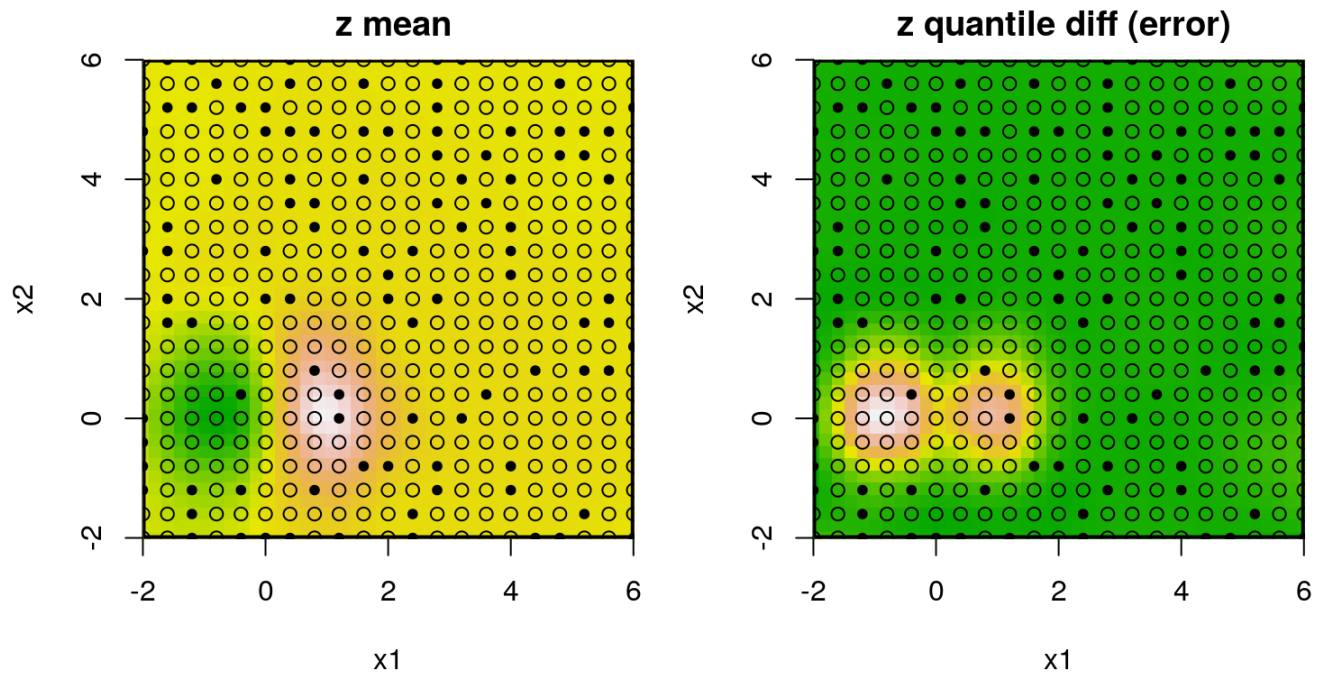
- Since there is radial symmetry we can use an isotropic correlation with `corr="exp"`.

```
out.bgp <- bgp(X=X, Z=Z, XX=XX, corr="exp", verb=0)
```

## 2-d Stationary GP surface

A stationary process means uniform uncertainty (in distance).

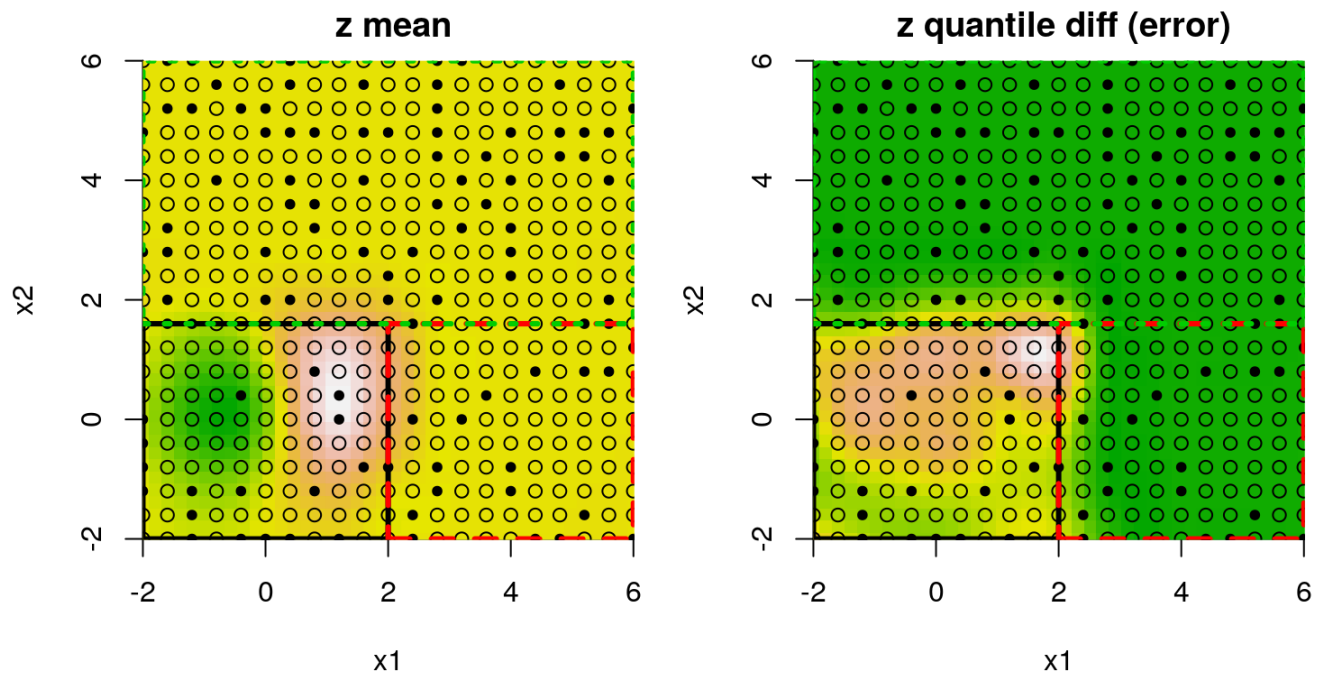
```
plot(out.bgp, pc="c")
```



## 2-d TGP surface

A nonstationary process means we can learn that it is hard in the SW corner.

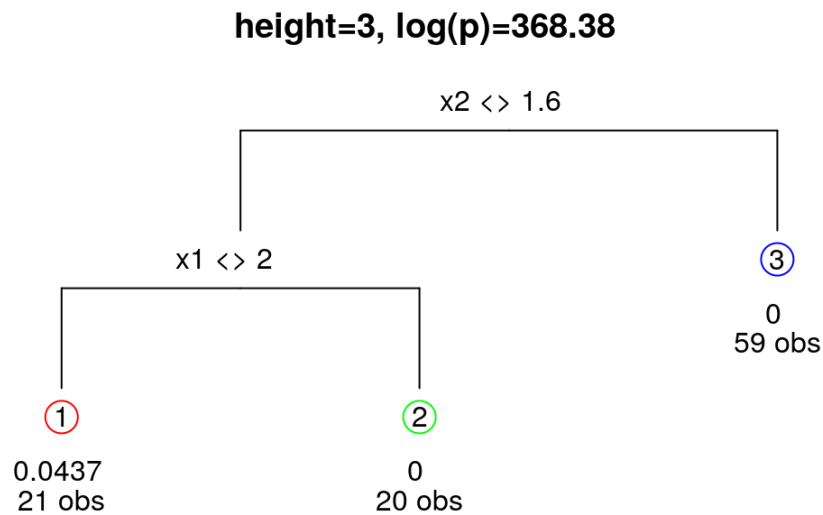
```
out.btgp <- btgp(X=X, Z=Z, XX=XX, corr="exp", R=10, verb=0)
plot(out.btgp, pc="c")
```



## Maximum a' posteriori tree

A clean treed partition of the input space.

```
tgp.trees(out.btgp, heights="map")
```



## LGBB data

Treed GPs were invented for the rocket booster (LGBB) data.

- NASA scientists knew they needed to partition the model (and the design) to separate subsonic and supersonic speeds,
- but they didn't know right where the partition should be.
- They wanted the data to tell them;
- and to tell them if more splits were helpful.

```
lgbb.as <- read.table("lgbb/lgbb_as.txt", header=TRUE)
lgbb.rest <- read.table("lgbb/lgbb_as_rest.txt", header=TRUE)
```

Those files contain

- inputs/outputs on a sequential (ALC) design selected from a dense candidate grid (Gramacy & Lee, 2009) (<http://amstat.tandfonline.com/doi/abs/10.1198/TECH.2009.0015#.Vgw-37SbOTA>),
- and the un-selected elements from that grid.

## Final prediction

Here we develop the “final predictive surface” after the sequential design effort, on the `lift` response.

```
X <- lgbb.as[,2:4]
Y <- lgbb.as$lift
XX <- lgbb.rest[2:4]
c(X=nrow(X), XX=nrow(XX))
```

```
##      X      XX
##    780 37128
```

The fit is computationally intensive, so we won't do any restarts (with default `R=1` ).

```
t1 <- system.time(fit <- btgppllm(X=X, Z=Y, XX=XX, bprior="b0", verb=0))
t1[3]
```

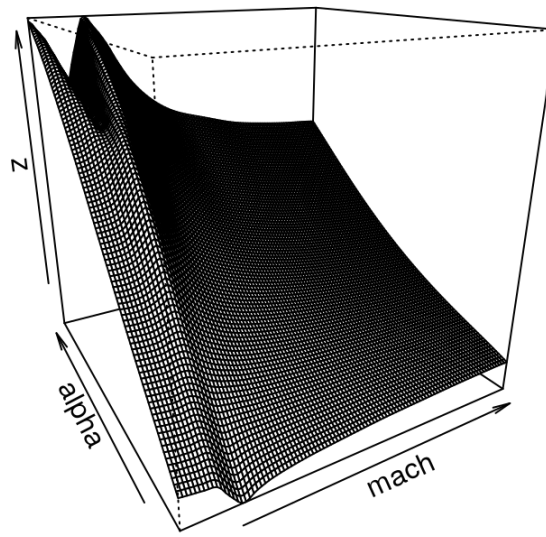
```
## elapsed  
## 7414.643
```

## Visualizing the (lift) predictive mean

We can inspect a 2-d slice of the posterior predictive surface, say for a side-slip-angle of zero.

```
plot(fit, slice=list(x=3, z=0), gridlen=c(100, 100), layout="surf", span=0.01)
```

**z mean, with (beta) fixed to (0)**



## Lift MAP tree

A clean two-element partition.

```
tgp.trees(fit, heights="map")
```



height=2, log(p)=3294.57

mach <> 1.45

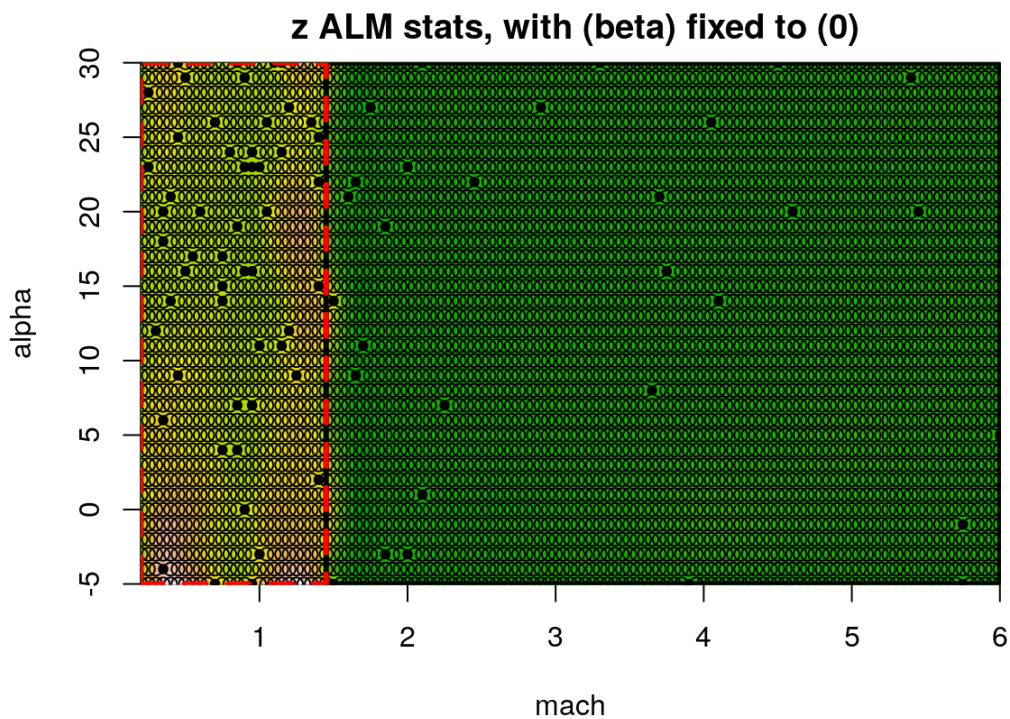
①  
0.0026  
537 obs

②  
0.0217  
243 obs

## Visualizing ALM

Future samples will be in the sub-sonic regime.

```
plot(fit, slice=list(x=3,z=0), gridlen=c(100,100), layout="as",as="alm",span=0.01)
```



## Speed

But yeah, that was pretty slow.

tgp contains a number of “switches” and “knobs” to help speed things up,

- at the expense of faithful modeling.

One way is to change the prior.

- Change  $p_{\text{split}}$  arguments  $\alpha$  (bigger) and  $\beta$  (smaller) to encourage deeper trees.

Another way is via the MCMC.

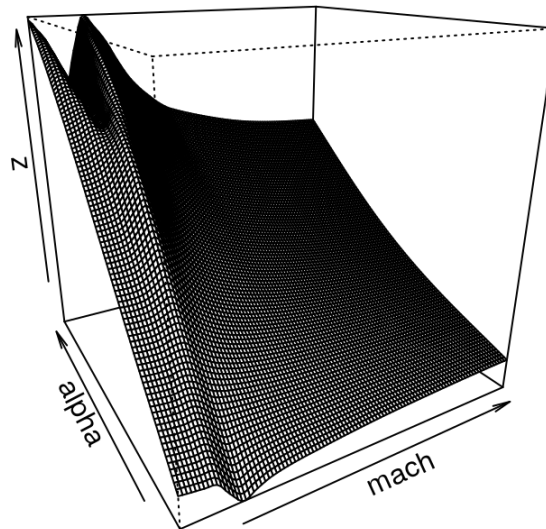
- The argument `linburn=TRUE` will “burn-in” the BTGP MCMC with a BTLM, and then “switch-on” GPs at the leaves toward the end.
- That ensures smaller partitions, but initializes the chain in a local mode of the tree space.
- The MCMC will usually “prune” back out of the modes, but almost never entirely.

## linburn predictive mean

Not much impact on the surface.

```
t2 <- system.time(fit2 <-  
  btgpllm(X=X, Z=Y, XX=XX, bprior="b0", linburn=TRUE, verb=0))  
plot(fit2, slice=list(x=3, z=0), gridlen=c(100, 100), layout="surf", span=0.01)
```

**z mean, with (beta) fixed to (0)**



## Pretty good

An order of magnitude faster.

```
c(full=t1[3], linburn=t2[3])
```

```
##      full.elapsed  linburn.elapsed  
##      7414.643      1253.932
```

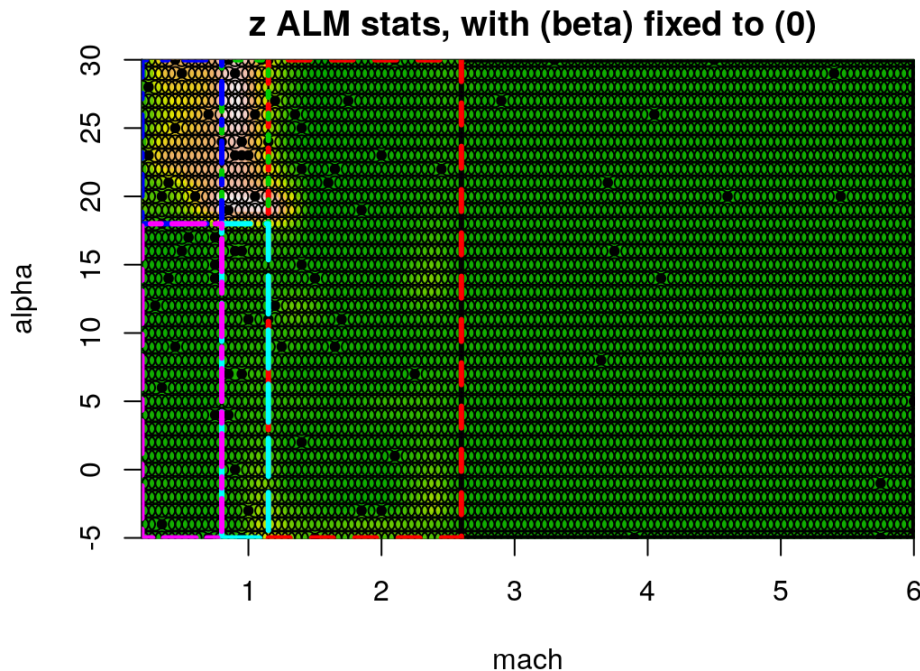
Indeed, much of the space is plausibly piece-wise linear anyway,

- but it helps to smooth out rough edges with the GP.

# Sequential design

The predictive variance make great *active learning* heuristics.

```
plot(fit2, slice=list(x=3,z=0), gridlen=c(100,100), layout="as",as="alm",span=0.01)
```



## Other heuristics

Integrated (over the input space) predictive variance is available with `Ds2X=TRUE`.

- An analog of Integrate Mean-Squared Prediction Error (IMSPE) sequential design.

Giving `improv=TRUE` returns samples of **posterior mean of improvement**

- So you get a fully Bayesian EI, averaged over tree and leaf model uncertainty,
- for a *truly* Bayesian Optimization.

For more details/tutorials, see

- Gramacy (2007) (<https://www.jstatsoft.org/article/view/v019i09>): a beginners primer;
- Gramacy & Taddy (2010) (<https://www.jstatsoft.org/article/view/v033i06>): advanced topics like EI, categorical inputs, sensitivity analysis, and tempering.

## Local approximate GPs

### Local kriging neighborhoods

The local approximate GP ( *laGP* ) idea has aspects in common with partition based schemes,

- in the sense that it creates sparsity in the covariance structure in a “geographically” local way.

It is reminiscent of what Cressie (1991, pp. 131-134) ([https://www.amazon.com/Statistics-Spatial-Data-Wiley-Probability-ebook/dp/B00VOY1LGS/ref=sr\\_1\\_1?ie=UTF8&qid=1479478617&sr=8-1&keywords=cressie+noel](https://www.amazon.com/Statistics-Spatial-Data-Wiley-Probability-ebook/dp/B00VOY1LGS/ref=sr_1_1?ie=UTF8&qid=1479478617&sr=8-1&keywords=cressie+noel)) called an “ad hoc” method of **local kriging neighborhoods**,

- which is not a very nice thing to say about a decent idea.

I think Cressie didn't anticipate

- the scale of modern data,
- applications to computer models and machine learning data (with inputs other than longitude and latitude),
- and the architecture of modern computers (mult-core/cluster computing begs for divide-and-conquer).

## Transductive learning

All together, the idea is more modern than could have been anticipated in 1991 (and earlier).

It draws, in part, on recent findings

- for approximate likelihoods in spatial data (e.g., Stein et al., 2004) (<http://onlinelibrary.wiley.com/doi/10.1046/j.1369-7412.2003.05512.x/pdf>),
- and active learning techniques for sequential design (e.g., Cohn, 1996) (<http://www.cs.cmu.edu/~cohn/psyche/AIM-1491.ps.Z>).

But a big divergence from previous approaches, particularly those from the spatial stats literature, lies in an emphasis on prediction

- which is the primary goal in computer experiments and machine learning applications.

laGP is an example of **transductive learning** (Vapnik, 1995) ([https://www.amazon.com/Statistical-Learning-Information-Science-Statistics/dp/0387987800/ref=sr\\_1\\_2?ie=UTF8&qid=1479479476&sr=8-2&keywords=The+nature+of+statistical+learning+theory](https://www.amazon.com/Statistical-Learning-Information-Science-Statistics/dp/0387987800/ref=sr_1_2?ie=UTF8&qid=1479479476&sr=8-2&keywords=The+nature+of+statistical+learning+theory)), as opposed to inductive learning, in that

- the behavior of the method/fit depends crucially on the *testing* locations.

## Local sub-design

For the next little bit, focus on prediction at a single predictive location  $x$ .

- $x$  is arbitrary; it is only important that it be a single location (for now).

Lets think about the properties of GP predictive equations (an emulator in the computer experiments context, say) at  $x$ .

- Data far from  $x$  vanishingly small influence on GP predictions.
- This is what motivates a CSK approach to inducing sparsity,
- but the difference here is that we're thinking about a particular  $x$ , not the entire spatial field.

So how about we *search* for the most useful data points (a **sub-design** relative to  $x$ )

- for prediction at  $x$ , without considering/handling large matrices.

## Nearest neighbor GP prediction

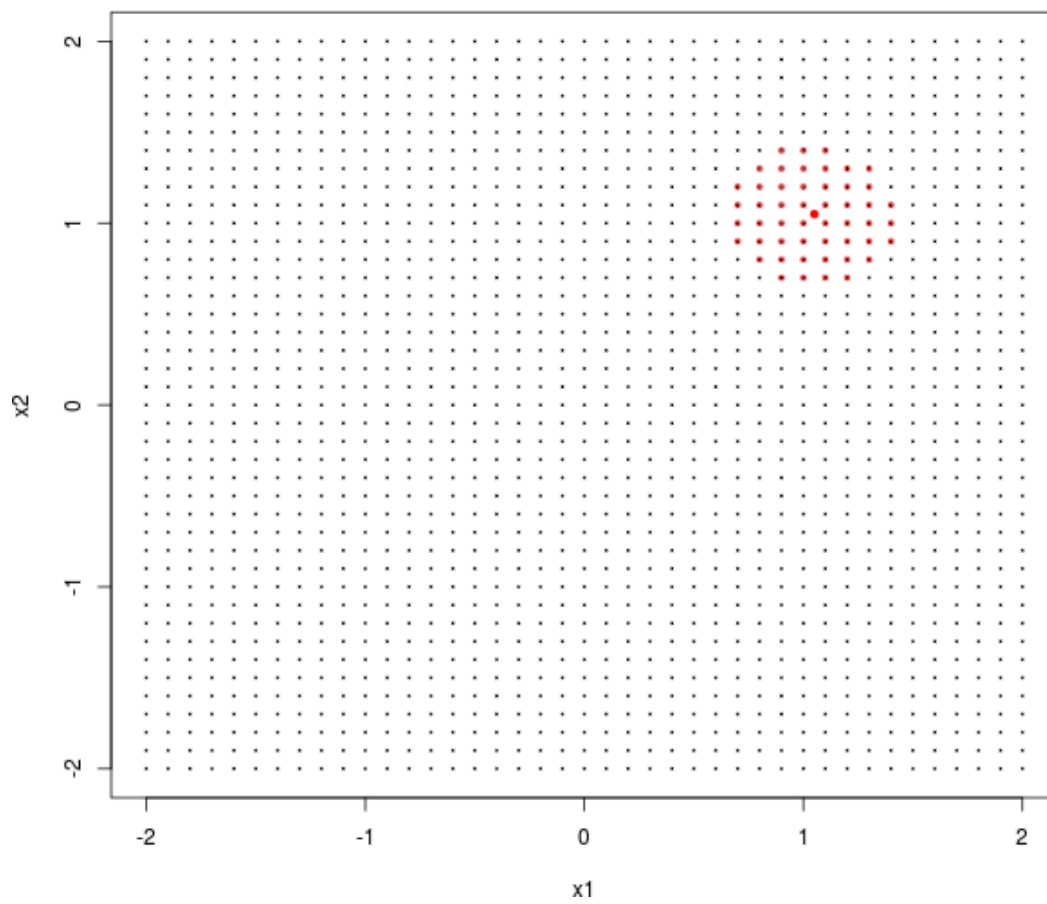
One option is a **nearest neighbor (NN)** subset:

1. Fill  $X_n(x) \subset X_N$  with local- $n \ll$  full- $N$  closest locations to  $x$ .
2. Emulate with  $Y(x) \mid D_n(x)$  where  $D_n(x) = (X_n, Y_n)$ .

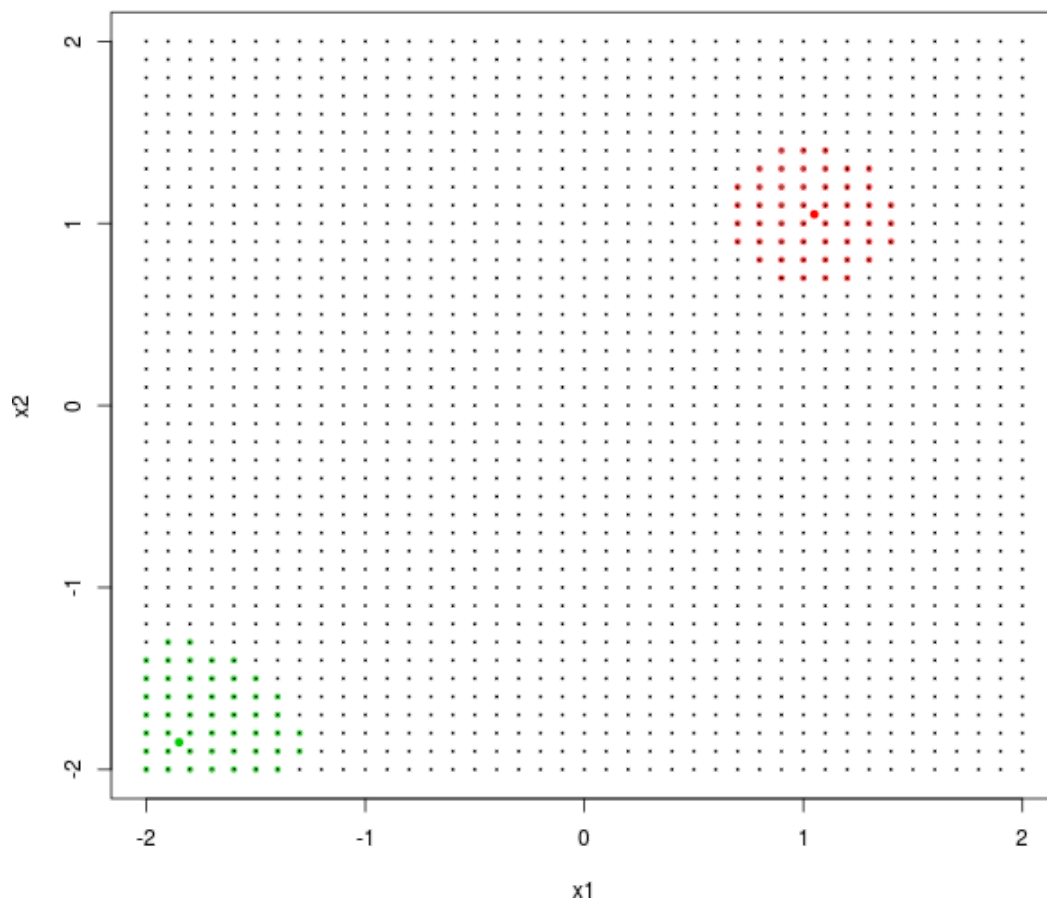
The best modern reference for this idea is Emery (2009) (<http://link.springer.com/article/10.1007/s10596-008-9116-8>).

- This is a very simple prediction rule, and potentially very fast:  $O(n^3)$  not  $O(N^3)$
- Choose  $n$  as large as computational constraints allow.

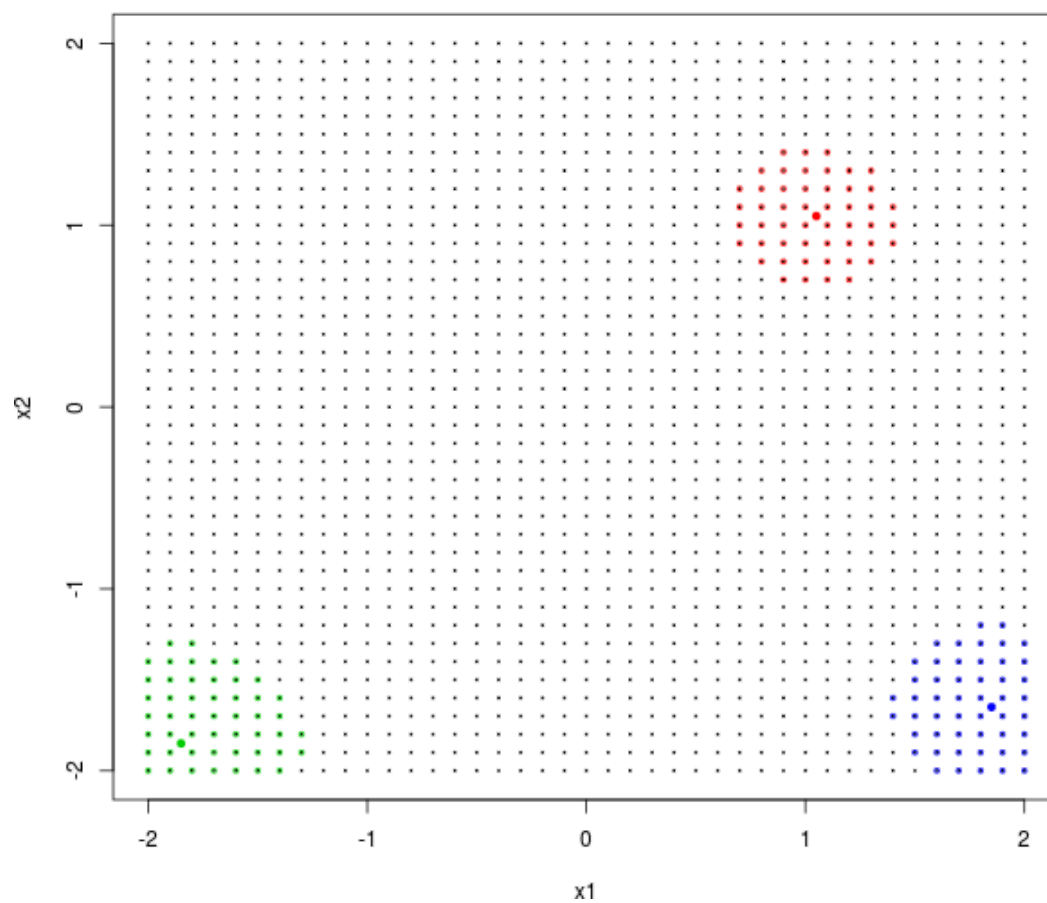
An animation ... (1)



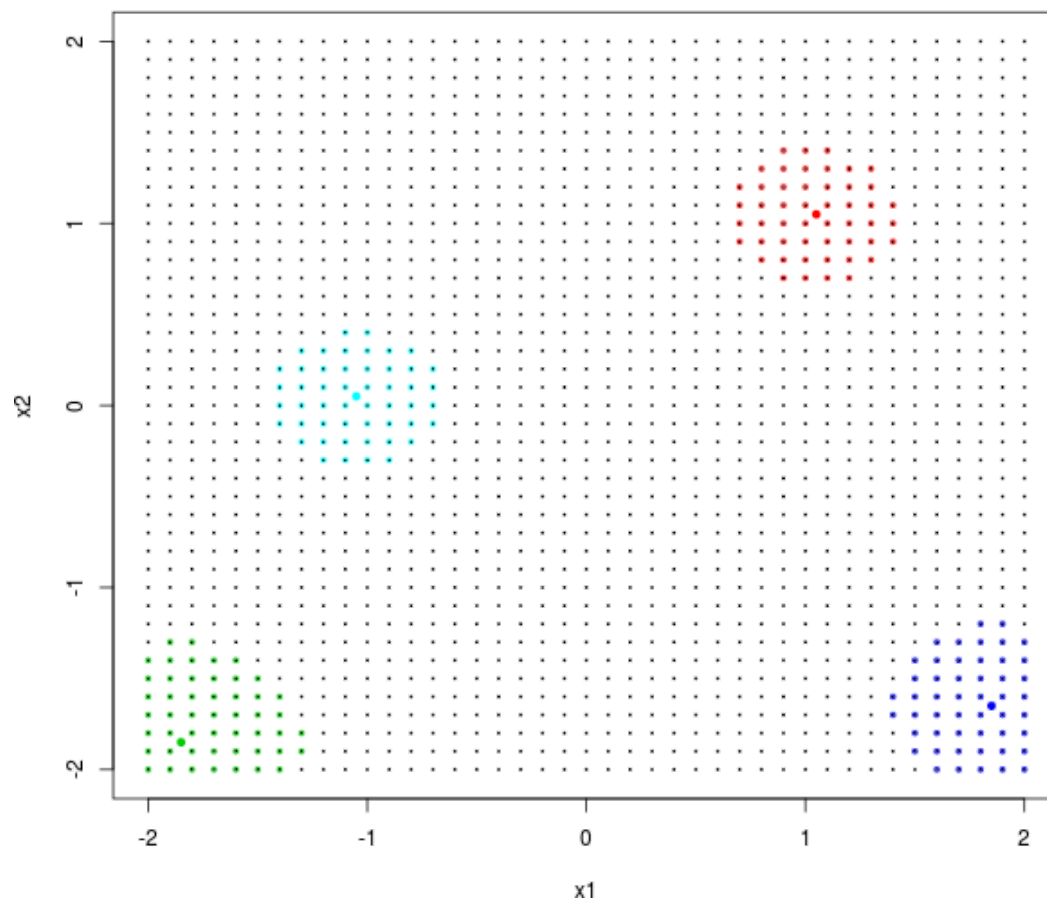
An animation ... (2)

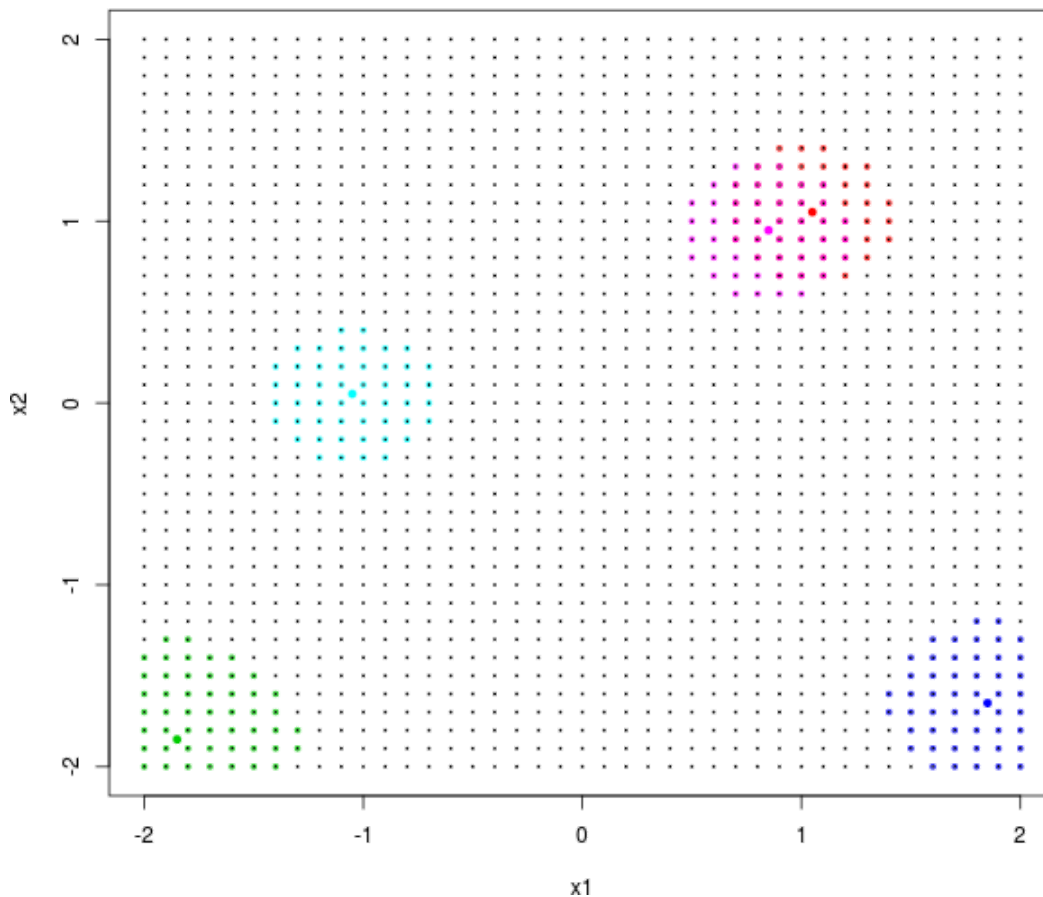


An animation ... (3)



An animation ... (4)





## What about that?

Is it sensible?

- As  $n \rightarrow N$ , predictions  $Y(x) \mid D_n \rightarrow Y(x) \mid D_N$ .
- $V(x) \mid D_n \gg V(x) \mid D_N$ , reflecting uncertainties inflated by the smaller design, where  $\sigma^2(x) = \hat{\tau}^2 V(x)$ .

Is it good?

- It's *not optimal* given computational limits,  $n$  (Vecchia, 1998 ([https://www.jstor.org/stable/2345768?seq=1#page\\_scan\\_tab\\_contents](https://www.jstor.org/stable/2345768?seq=1#page_scan_tab_contents)); Stein, et al., 2004) (<http://onlinelibrary.wiley.com/doi/10.1046/j.1369-7412.2003.05512.x/pdf>).
- But, finding the optimal solution(s) is a combinatorially huge undertaking.

Our questions:

- Can we do better than NN (in terms of prediction accuracy) without much extra effort (in terms of computational cost)?
- I.e., with computation in  $O(n^3)$ ?

## Greedy scheme

Yes!: with a **greedy**/forward stepwise scheme.

For a particular  $x$ , solve a sequence of *easy* decision problems.

For  $j = n_0, \dots, n$

1. given  $D_j(x)$ , choose  $x_{j+1}$  according to some criterion;

2. augment the design  $D_{j+1}(x) = D_j(x) \cup (x_{j+1}, y(x_{j+1}))$  and update the GP approximation.

Optimizing the criterion (1), and updating the GP (2), must not exceed  $O(j^2)$  so the total scheme remains in  $O(n^3)$ .

- Initialize with a small  $D_{n_0}(x)$  comprised of NNs.

## A criterion ...

... for sequential (sub-) design:

Given  $D_j(x)$  for particular  $x$ , we search for  $x_{j+1}$  by minimizing empirical Bayes **mean-squared prediction error**:

$$\begin{aligned} J(x_{j+1}, x) &= \mathbb{E}\{[Y(x) - \mu_{j+1}(x; \hat{\theta}_{j+1})]^2 \mid D_j(x)\} \\ &\approx V_j(x \mid x_{j+1}; \hat{\theta}_j) + \text{something small} \end{aligned}$$

That is:

- The reduced variance at  $x$  after  $x_{j+1}$  is added into the design;
- plus something that is relatively small.

## For example

Consider a full- $N$  sized design on a 2-d grid.

```
xg <- seq(-2, 2, by = 0.02)
X <- as.matrix(expand.grid(xg, xg))
print(N <- nrow(X))
```

```
## [1] 40401
```

And evaluate that to get responses (we optimized this with the AL earlier).

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

## Local sub-design(s)

Consider prediction location  $x$ , denoted by `Xref` in the code below,

- with local designs built up to  $n = 50$  with  $n_0 = 6$  initial NNs.

```
library(laGP)
Xref <- matrix(c(-1.725, 1.725), nrow=1)
p <- laGP(Xref, 6, 50, X, Y, d=0.1)
```

Pretty fast:

```
p$time
```

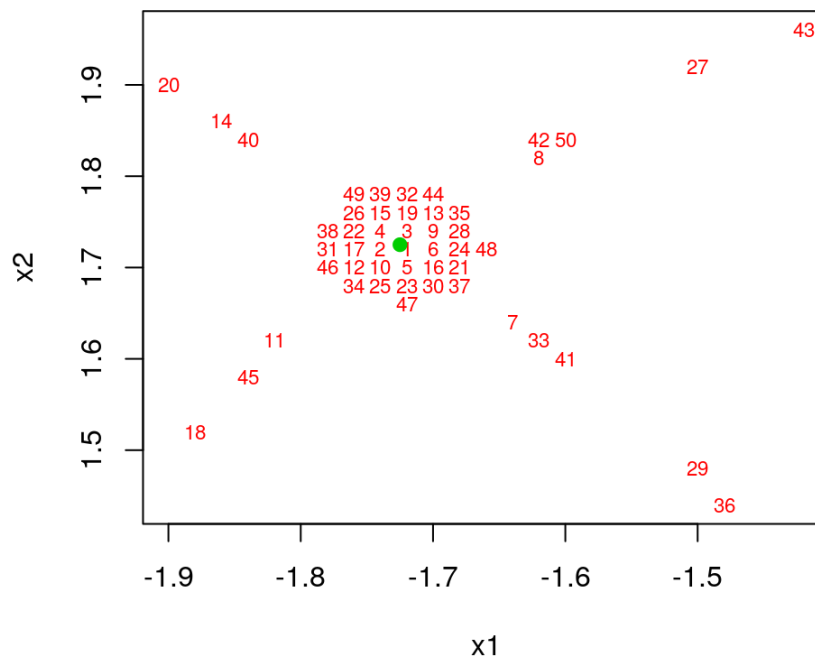


```
## elapsed
## 0.051
```

For a point of reference, inverting a  $4000 \times 4000$  matrix takes about five seconds on the same machine, using a multi-threaded BLAS/Lapack.

- Never mind a  $40,000 \times 40,000$  one – impossible on my machine.

```
plot(X[p$Xi,], xlab="x1", ylab="x2", type="n",
     xlim=range(X[p$Xi,1]), ylim=range(X[p$Xi,2]))
text(X[p$Xi,], labels=1:length(p$Xi), cex=0.7, col=2)
points(Xref[1], Xref[2], pch =19, col=3)
```



## A surprising result?

Why does the criteria not prefer the closest possible points, i.e., the NNs?

- An exponentially decaying correlation should substantially devalue locations far from  $x$ .

Gramacy & Haaland (2016) (<http://www.tandfonline.com/doi/abs/10.1080/00401706.2015.1027067>) explain that the form of the correlation has very little to do with it.

The *reduction* in variance

$$v_j(x; \theta) - v_{j+1}(x; \theta) = k_j^\top(x) G_j(x_{j+1}) v_j(x_{j+1}) k_j(x) + \dots + K(x_{j+1}, x)^2 / v_j(x_{j+1})$$

- is quadratic in inverse distance:  $K(x_{j+1}, x)^2$ ,
- but also quadratic in *inverse* inverse distance via

$$G_j(x') \equiv g_j(x') g_j^\top(x') \quad \text{where} \quad g_j(x') = -K_j^{-1} k_j(x') / v_j(x').$$

## A trade-off

So the criteria makes a trade-off:

- minimize “distance” to  $x$

- while maximizing “distance” (or minimizing inverse distance) to the existing design  $X_j(x)$ .

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  $X_j(x)$ .

## Global emulation

How do we extend this to predict on a big testing/predictive set?

One option is to serialize:

- for loop over each  $x \in \mathcal{X}$ .

But why serialize when you can **parallelize**?

- Each  $D_n(x)$  is obtained independently of the other  $x$ 's,
- so they can be constructed simultaneously.

In `laGP`'s C implementation, that's as simple as a “parallel-for” OpenMP pragma.

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

## Big prediction

To illustrate, consider the following  $\sim 10\text{K}$ -element predictive grid in  $[-2, 2]^2$ ,

- spaced to avoid the original  $N = 40\text{K}$  design.

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

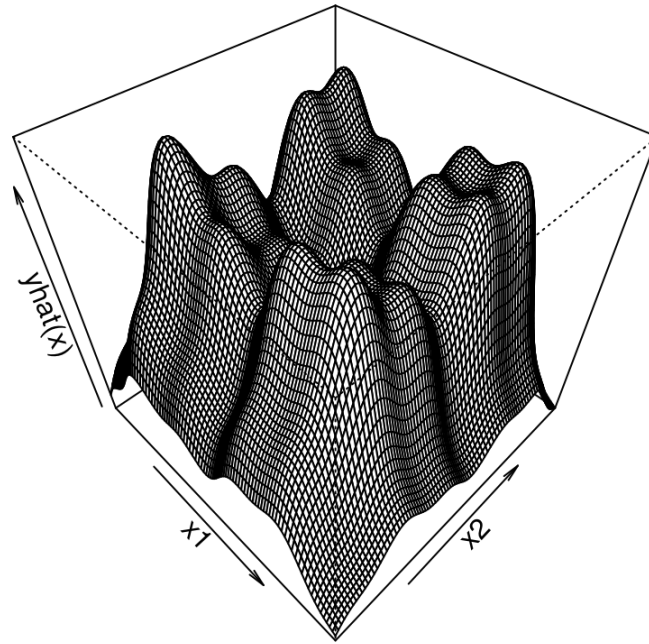
The `aGP` function iterates over the elements of  $\mathcal{X} = \text{XX}$ ,

- and its `omp.threads` argument controls the number of OpenMP threads.
- Here we'll use 8 threads, even though my desktop is hyperthreaded (can do 16).

```
nth <- 8
P.alc <- aGP(X, Y, XX, omp.threads=nth, verb=0)
```

## Visualizing the surface

```
persp(xx, xx, -matrix(P.alc$mean, ncol = length(xx)), phi = 45,
      theta = 45, xlab = "x1", ylab = "x2", zlab = "yhat(x)")
```



## A challenging surface

Although the input dimension is low,

- the input–output relationship is nuanced
- and merits a dense design in the input space to fully map.

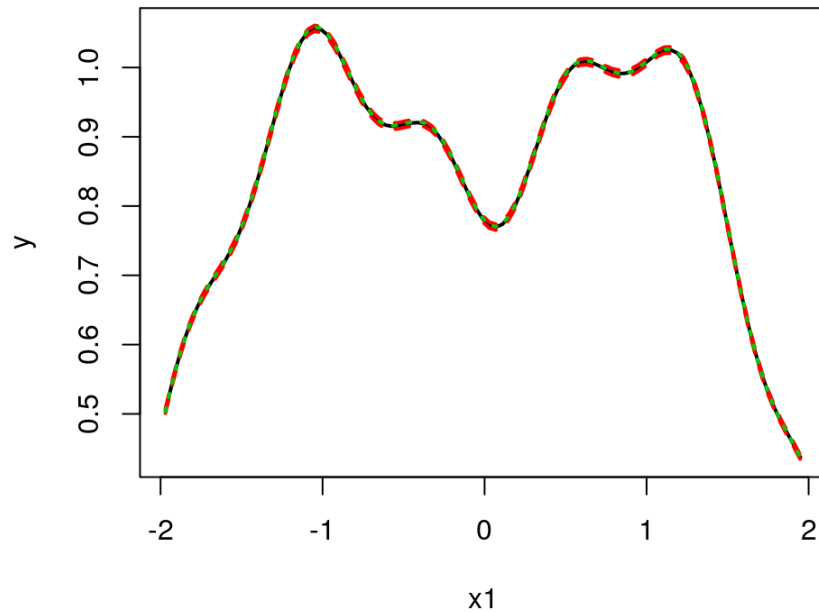
For a closer look, consider a slice through the predictive surface at  $x_2 = 0.51$ .

The code below sets up the slice and its plot.

```
med <- 0.51
zs <- XX[, 2] == med
sv <- sqrt(P.alc$var[zs])
r <- range(c(-P.alc$mean[zs] + 2*sv, -P.alc$mean[zs] - 2*sv))
```

## Visualizing the slice

```
plot(XX[zs,1], -P.alc$mean[zs], type="l", lwd=2, ylim=r, xlab="x1", ylab="y")
lines(XX[zs,1], -P.alc$mean[zs] + 2*sv, col=2, lty=2, lwd=2)
lines(XX[zs,1], -P.alc$mean[zs] - 2*sv, col=2, lty=2, lwd=2)
lines(XX[zs,1], -YY[zs], col=3, lwd=2, lty=3)
```



## What do we see?

The error bars are very tight on the scale of the response,

- and although no continuity is enforced
  - (calculations at nearby locations are independent and potentially occur in parallel)
- the resulting surface looks smooth to the eye.

### What don't we see?

Accuracy, however, is not uniform.

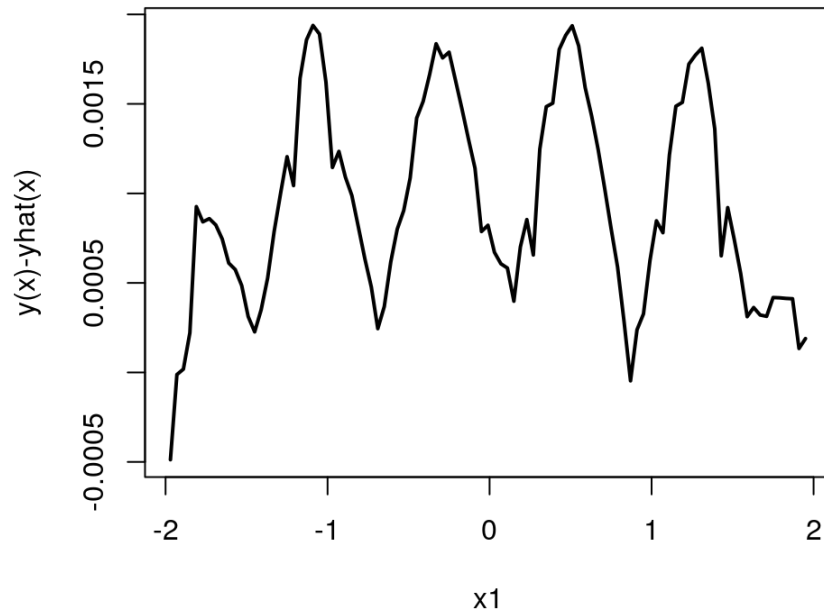
- Consider the discrepancy with the truth.

```
diff <- P.alc$mean - YY
```

## Visualizing discrepancy

Systematic bias in prediction, although extremely small.

```
plot(XX[zs,1], diff[zs], type="l", lwd=2, xlab = "x1", ylab = "y(x)-yhat(x)")
```



## Lacking fully dynamic ability

Considering 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 relationship.

Although an approximation, the local nature of modeling means that, from a global perspective,

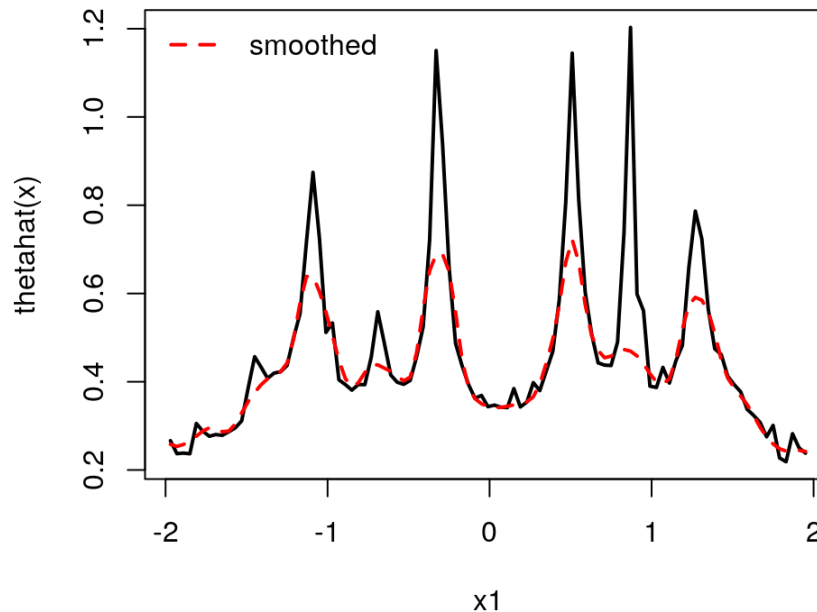
- the predictor is *more* flexible than the full- $N$  stationary Gaussian process predictor.
- Inferring separate independent predictors across the elements of a vast predictive grid lends aGP a degree of nonstationarity.

aGP goes beyond that by learning separate  $\hat{\theta}_n(x)$  local to each  $x \in \mathcal{X}$

- by maximizing the local likelihoods.

In fact, the lengthscales vary spatially, and relatively smoothly.

```
plot(XX[zs,1], P.alc$mle$d[zs], type="l", lwd=2, xlab="x1", ylab="thetahat(x)")
df <- data.frame(y = log(P.alc$mle$d), XX)
lo <- loess(y ~ ., data=df, span=0.01)
lines(XX[zs,1], exp(lo$fitted)[zs], col=2, lty=2, lwd=2)
legend("topleft", "smoothed", col=2, lty=2, lwd=2, bty="n")
```



## Local isotropy, global nonstationarity

So even though the spatial field may be *locally* isotropic,

- and therefore assumes stationarity to a certain extent,
- *globally* the characteristics are less constrained.

Nevertheless, even the extra degree of flexibility afforded by spatially varying  $\hat{\theta}_n(x)$  is not enough to entirely mitigate the small amount of bias we saw.

Several enhancements offer potential for improved performance.

- Anisotropic/separable correlation structure (hold that thought a sec).
- A two-stage scheme, re-designing  $X_n(x) \mid \hat{\theta}_n(x)$ , ...

## Two-stage local/global approximation

Here, sub-design search is based on the smoothed first-stage lengthscales.

```
P.alc2 <- aGP(X, Y, XX, d=exp(lo$fitted), omp.threads=nth, verb=0)
```

Now consider comparing the predictions from the first iteration to those from the second in terms of RMSE.

```
rmse <- data.frame(alc = sqrt(mean((P.alc$mean - YY)^2)),
  alc2 = sqrt(mean((P.alc2$mean - YY)^2)))
rmse
```

```
##           alc           alc2
## 1 0.0006434536 0.0003222866
```

- Possibly not impressive (50%), but consistent across a large range of examples.
- It doesn't completely solve the bias problem, but our sleeves have more tricks.

First, a harder example ...

## Back to the borehole

Check this out.

```
out1 <- aGP(x,y, xpred, d=list(max=20), omp.threads=nth, verb=0)
out2 <- aGP(x,y, xpred, d=list(start=out1$mle$d,max=20), omp.threads=nth, verb=0)
```

Much faster; much more accurate.

```
print(times <- c(times, aGP=as.numeric(out1$time), aGP2=as.numeric(out2$time)))
```

```
##   sparse   dense   s999   aGP   aGP2
## 3457.123 9861.910 420.356 4.653 4.474
```

```
s21 <- out1$var; s22 <- out2$var
print(scores <- c(scores, aGP=mean(-(out1$mean - ypred.0)^2/s21 - log(s21)),
  aGP2=mean(-(out1$mean - ypred.0)^2/s22 - log(s22))))
```

```
##   sparse   dense   s999   aGP   aGP2
## -1.5084444 -2.1318560 -1.6806796 -0.5698724 -0.5508215
```

## Can we do even better?

And we did all that with an isotropic correlation function,

- compared to Kauffman's separable CSK. Lets be fair ...

```
outs <- aGPsep(x, y, xpred, d=list(max=20), omp.threads=nth, verb=0)
```

Similar compute times; quite a bit more accurate.

```
print(times <- c(times, aGPs=as.numeric(outs$time)))
```

```
##   sparse   dense   s999   aGP   aGP2   aGPs
## 3457.123 9861.910 420.356 4.653 4.474 4.898
```

```
s2 <- outs$var
print(scores <- c(scores, aGPs=mean(-(outs$mean - ypred.0)^2/s2 - log(s2))))
```

```
##   sparse   dense   s999   aGP   aGP2   aGPs
## -1.5084444 -2.1318560 -1.6806796 -0.5698724 -0.5508215 0.2059281
```

## Global/local modeling

Surely something is lost on this local approach to GP approximation.

Kaufman et al. astutely observed that, especially when inducing sparsity in the covariance structure,

- it is important to “put something global back in”.
- They partition modeling between basis-expanded (non-stationary) linear trend and locally (stationary) spatial residual.

That's not easily mapped to the `laGP` setup,

- where the local part is where the nonstationary effect comes in.

But the idea has merit, and we ought to be able to find an appropriate analog in the `laGP` world.

# Global lengthscale

Instead, consider not a partition between trend and residual,

- but rather between lengthscales.

Liu (2014) (<https://rucore.libraries.rutgers.edu/rutgers-lib/44163/PDF/1/>) showed that a **consistent** estimator of the *global* (separable) lengthscale can be estimated via (more manageably sized) random data subsets.

```
n <- 1000
d2 <- darg(list(mle=TRUE, max=100), x)
subs <- sample(1:nrow(x), n, replace=FALSE)
gpsi <- newGPsep(x[subs, ], y[subs], rep(d2$start, dim), g=1/1000, dK=TRUE)
that <- mleGPsep(gpsi, param="d", tmin=d2$min, tmax=d2$max, ab=d2$ab, maxit=200)
psub <- predGPsep(gpsi, xpred, lite=TRUE)
deleteGPsep(gpsi)
```

## Predicting with the subset

Observe that local subset GP prediction is pretty good on its own,

- because the response is super smooth and pretty stationary.
- And 1000 is a sizable portion of 4000.

```
s2 <- psub$s2
print(scores <- c(scores, sub=mean(-(psub$mean - ypred.0)^2/s2 - log(s2))))
```

##	sparse	dense	s999	aGP	aGP2	aGPs	sub
##	-1.5084444	-2.1318560	-1.6806796	-0.5698724	-0.5508215	0.2059281	0.8140304

- (Makes you wonder what was going on with CSK.)

The estimated lengthscales, stored in `that`, are super handy.

- They can be used to “pre-scale” the inputs, so that afterwards the global lengthscale will be 1.

## Multi-resolution global/local GP

Don’t forget to scale both training and testing inputs.

```
scale <- sqrt(that$d)
xs <- x; xpreds <- xpred
for(j in 1:ncol(xs)) {
  xs[,j] <- xs[,j] / scale[j]
  xpreds[,j] <- xpreds[,j] / scale[j]
}
```

Now fit a local GP on the the scaled inputs, achieving a **multiresolution effect**; note that  $\theta$  is initialized to 1.

```
out3 <- aGP(xs, y, xpreds, d=list(start=1, max=20), omp.threads=nth, verb=0)
s2 <- out3$var
print(scores <- c(scores, aGPsm=mean(-(out3$mean - ypred.0)^2/s2 - log(s2))))
```



```
##      sparse      dense      s999      aGP      aGP2      aGPs      sub
## -1.5084444 -2.1318560 -1.6806796 -0.5698724 -0.5508215  0.2059281  0.8140304
##      aGPsm
## 1.2003101
```

## One more thing

The default nugget value in `laGP` and `aGP` is too large for most *deterministic* computer experiments applications.

- It is conservative so users don't get errors.
- But we can dial it way down for this borehole example.

```
g <- 1/100000000
out4 <- aGP(xs, y, xpreds, d=list(start=1, max=20), g=g, omp.threads=nth, verb=0)
s2 <- out4$var
print(scores <- c(scores, aGPsm2=mean(-(out4$mean - ypred.0)^2/s2 - log(s2))))
```

```
##      sparse      dense      s999      aGP      aGP2      aGPs      sub
## -1.5084444 -2.1318560 -1.6806796 -0.5698724 -0.5508215  0.2059281  0.8140304
##      aGPsm      aGPsm2
## 1.2003101  5.5685688
```

Holy smokes!

## Satellite drag

## Hubble space telescope

Lets revisit the HST drag data we introduced a while back ([lect1.html#57](#)).

Recall that the goal was to be able to predict the drag coefficient (response), globally,

- to within 1% root-mean-squared percentage error (RMSPE).

There are eight inputs, including HST's panel angle, and files with runs obtained on LHS designs, separately for each chemical species (O, O<sub>2</sub>, N, N<sub>2</sub>, He, H).

- We'll focus on 1 million runs in Helium (He) here;
  - one of the harder species.

## Coding the data

Read in the data ...

```
hstHe <- read.table("lanl/HST/hstHe.dat", header=TRUE)
nrow(hstHe)
```

```
## [1] 1000000
```

... and (as usual) work with coded the inputs.

```

m <- ncol(hstHe)-1
X <- hstHe[,1:m]
Y <- hstHe[,m+1]
maxX <- apply(X, 2, max)
minX <- apply(X, 2, min)
for(j in 1:ncol(X)) {
  X[,j] <- X[,j] - minX[j]
  X[,j] <- X[,j]/(maxX[j]-minX[j])
}

```

## Cross-validation

Consider a 10-fold CV setup ...

```

cv.folds <- function (n, folds = 10)
  split(sample(1:n), rep(1:folds, length = n))
f <- cv.folds(nrow(X), 10)

```

... but only “loop” through one fold here.

- (The other folds proceed very similarly.)

```

o <- f[[1]]
Xtest <- X[o,]; Xtrain <- X[-o,]
Ytest <- Y[o]; Ytrain <- Y[-o]
c(test=length(Ytest), train=length(Ytrain))

```

```

## test train
## 100000 900000

```

## Subset GP first

We'll need it for our multiresolution approach later anyway, so lets start with a subset GP first.

- Recall that this data is not deterministic.
- It involves a Monte Carlo, and so has a small amount of noise.

```

da.orig <- darg(list(mle=TRUE), Xtrain, samp.size=10000)
sub <- sample(1:nrow(Xtrain), 1000, replace=FALSE)
gpsi <- newGPsep(Xtrain[sub,], Ytrain[sub], d=0.1, g=1/1000, dK=TRUE)
mle <- mleGPsep(gpsi, tmin=da.orig$min, tmax=10*da.orig$max, ab=da.orig$ab)
psub <- predGPsep(gpsi, Xtest, lite=TRUE)
deleteGPsep(gpsi)
rmspe <- c(sub=sqrt(mean((100*(psub$mean - Ytest)/Ytest)^2)))
rmspe

```

```

## sub
## 10.35999

```

- Not even close to our 1% target.

## Local GP

How about a separable local GP?

```
alcsep <- aGPsep(Xtrain, Ytrain, Xtest, d=da.orig, omp.threads=nth, verb=0)
print(rmspe <- c(rmspe, alc=sqrt(mean((100*(alcsep$mean - Ytest)/Ytest)^2))))
```

```
##      sub      alc
## 10.35999  5.86535
```

- Much better, but not quite to our 1% goal.

Notice that a (10×) smaller nugget doesn't help here.

```
g <- 1/100000
alcsep2 <- aGPsep(Xtrain, Ytrain, Xtest, d=da.orig, g=g, omp.threads=nth, verb=0)
print(rmspe <- c(rmspe, alc2=sqrt(mean((100*(alcsep2$mean - Ytest)/Ytest)^2))))
```

```
##      sub      alc      alc2
## 10.359987  5.865350  5.948451
```

## Multiresolution global/local GP

First pre-scale the inputs with the `mle` calculated on the subset above.

```
for(j in 1:ncol(Xtrain)) {
  Xtrain[,j] <- Xtrain[,j] / sqrt(mle$d[j])
  Xtest[,j] <- Xtest[,j] / sqrt(mle$d[j])
}
```

Construct a default prior appropriate for the scaled inputs.

```
da.s <- darg(list(mle=TRUE), Xtrain, samp.size=10000)
da.s$start <- 1; if(da.s$max < 2) da.s$max <- 2
```

Now the local fit on the scaled inputs. Woot!

```
alcsep.s <- aGPsep(Xtrain, Ytrain, Xtest, d=da.s, omp.threads=nth, verb=0)
print(rmspe <- c(rmspe, alcs=sqrt(mean((100*(alcsep.s$mean - Ytest)/Ytest)^2))))
```

```
##      sub      alc      alc2      alcs
## 10.3599865  5.8653502  5.9484513  0.7824052
```

## Calibration

### Local is ideal

Recall the modularized KOH calibration (lect5.html) apparatus,

- which relied on computer model emulations at a small number  $X^F$  field data sites,
- paired with *promising* values of the calibration parameter  $u$ .

That is, we only need GP predictions at a relatively small set of locations,

- determined “on-line” as optimization over  $u$  proceeds in search of  $\hat{u}$ ,
- regardless of the (potentially massive) size of the computer experiment.

Local GPs couldn't be more ideal for this setup.

# A drawback

One drawback, however, is that the discrete nature of independent local design searches for  $\hat{y}^M(x_j^F, u)$ ,

- for each index  $j = 1, \dots, N_F$  into  $X^F$ ,

is going to ensure that our likelihood-based calibration objective is not a continuous in  $u$

- which will thwart most local optimization methods.

Gramacy, et al. (2015) (<http://projecteuclid.org/euclid.aoas/1446488734>) suggest a derivative-free approach:

- the MADS algorithm (Audet & Dennis, Jr., 2006) (<http://www.caam.rice.edu/caam/trs/2004/TR04-02.pdf>)
  - as implemented in NOMAD (<https://www.gerad.ca/nomad/Project/Home.html>).

## Initialization & implementation

As MADS is a local solver, NOMAD requires initialization.

Gramacy et al. suggest choosing starting  $u$ -values from the best value(s) of the objective found on a small space-filling design.

The `laGP` package contains several functions that automate that objective, e.g.,

- `fcalib` is like the `calib` function we implemented for the full GP case;
- `discrep.est` is like our `bhat` function;
- special cases for unbiased calibration are also implemented.

## A synthetic example

I'd love to show you the CRASH calibration ([lect1.html#48](#)),

- but there are too many nuances for a tutorial setting.
- Instead, we'll look at a synthetic analog from the same paper (<http://projecteuclid.org/euclid.aoas/1446488734>).

Consider a computer model formulated below, and its implementation in R.

$$y^M(x, u) = \left(1 - e^{-\frac{1}{2x_2}}\right) \frac{1000u_1x_1^3 + 1900x_1^2 + 2092x_1 + 60}{100u_2x_1^3 + 500x_1^2 + 4x_1 + 20}.$$

```
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)
}
```

## Bias and field data

The field data is generated as

$$y^F(x) = y^M(x, u^*) + b(x) + \varepsilon, \quad \text{where } b(x) = \frac{10x_1^2 + 4x_2^2}{50x_1x_2 + 10}$$

and  $\varepsilon \stackrel{\text{iid}}{\sim} \mathcal{N}(0, 0.5^2)$ ,

using  $u^* = (0.2, 0.1)$ .

In 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)
}
```

## Field data

Consider  $N_F = 100$  field data runs comprised of two replicates of a 50-sized 2d LHS of  $x$ -values.

- $Z_u$  is the intermediate computer model evaluation at  $u^*$ .

```
ny <- 50
X <- randomLHS(ny, 2)
u <- c(0.2, 0.1)
Zu <- M(X, matrix(u, nrow=1))
sd <- 0.5
reps <- 2
Y <- rep(Zu, reps) + rep(bias(X), reps) + rnorm(reps*length(Zu), sd=sd)
length(Y)
```

```
## [1] 100
```

## Computer model runs

Augment the field data with  $N_M = 10500$  computer model runs comprised of

- a 4d LHS of size 10000 of  $(x, u)$ -values,
- and runs at the 50 field data locations, paired with 500 2d-LHSs of  $u$ -values.

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

```
## [1] 10500
```

- The variable  $Z$  contains our  $y^M$  values.

## The setup

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

```

bias.est <- TRUE                ## change to FALSE for unbiased version
methods <- rep("alc", 2)       ## two passes of laGP design/MLE
da <- d <- darg(NULL, XU)
g <- garg(list(mle = TRUE), Y)

```

The prior is completed with a (log) prior density on the calibration parameter,  $u$ , chosen to discourage settings on the “edges” of the space.

```

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)))
}

```

## Initialization search

Now we are ready to evaluate the objective on a “grid” to search for a starting value for `NOMAD`.

Here is the “grid”, via maximin LHS away from the edges.

```

initsize <- 10*ncol(X)
uinit <- maximinLHS(initsize, 2)
uinit <- 0.9*uinit + 0.05

```

Here are the objective evaluations on that “grid”.

```

llinit <- rep(NA, nrow(uinit))
for(i in 1:nrow(uinit)) {
  llinit[i] <- fcalib(uinit[i,], XU, Z, X, Y, da, d, g, beta.prior,
    methods, NULL, bias.est, nth, verb=0)
}

```

## Now NOMAD

An R interface to `NOMAD` is provided by `snomadr` in the `crs` package,

- which allows the passing of a number of `NOMAD` options.
- The options below have been found to work well in a number of `laGP`-based calibration examples.

```

library(crs)
imesh <- 0.1
opts <- list("MAX_BB_EVAL"=1000, "INITIAL_MESH_SIZE"=imesh,
  "MIN_POLL_SIZE"="r0.001", "DISPLAY_DEGREE"=0)

```

The code on the following slide invokes `snomadr` on the best input(s) found on the “grid”,

- looping over them until a minimum number of `NOMAD` iterations has been reached.

Usually one pass is sufficient to meet the iteration threshold.

```

its <- 0; i <- 1; out <- NULL
o <- order(llinit)
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=NULL, verb=0,
    bias=bias.est, omp.threads=nth, uprior=beta.prior, save.global=.GlobalEnv)
  its <- its + outi$iterations
  if(is.null(out) || outi$objective < out$objective) out <- outi
  i <- i + 1
}

```

Then, extract information for visualizing/interpolating a posterior surface over  $u$ .

```

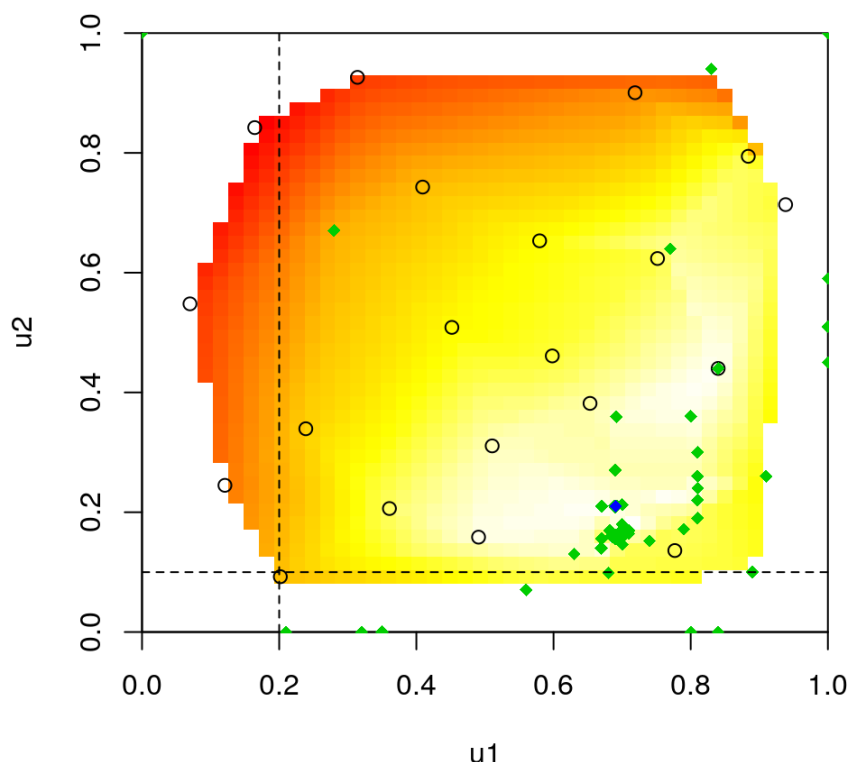
Xp <- rbind(uinit, as.matrix(fcalib.save[,1:2]))
Zp <- c(-llinit, fcalib.save[,3])
wi <- which(!is.finite(Zp))
if(length(wi) > 0) { Xp <- Xp[-wi, ]
  Zp <- Zp[-wi]}
library(akima)
surf <- interp(Xp[,1], Xp[,2], Zp, duplicate = "mean")
u.hat <- out$solution

```

```

image(surf, xlab="u1", ylab="u2", col=heat.colors(128), xlim=c(0,1), ylim=c(0,1))
points(uinit); points(fcalib.save[,1:2], col = 3, pch = 18)
points(u.hat[1], u.hat[2], col = 4, pch = 18)
abline(v=u[1], lty = 2); abline(h=u[2], lty = 2)

```



## Did we do good?

Observe that the true  $u^*$  value is far from the  $\hat{u}$  that we found.

- Indeed, the surface is fairly peaked around around  $\hat{u}$ ,

- giving very little support to the true value.

Since there are far fewer evaluations made near  $u^*$ ,

- it is worth checking if the solver missed an area of high likelihood.

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

```
##      u.hat      u
## -130.5904 -133.6634
```

- Nope, that's not it.

## Out-of-sample exercise

Lets see which ( $\hat{u}$  or  $u^*$ ) leads to better prediction out-of-sample.

```
nny <- 1000
XX <- randomLHS(nny, 2)
ZZu <- M(XX, matrix(u, nrow=1))
YYtrue <- ZZu + bias(XX)
```

First, prediction with the true  $u^*$ .

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

## Estimated version

For the estimated  $\hat{u}$  we need to backtrack through what we did earlier,

- and save the intermediate steps to re-build the composite for prediction.

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

Here is a sanity check that this gives the same objective evaluation as what came out of `snomadr`.

```
print(c(cmle$ll, -out$objective))
```

```
## [1] -130.5904 -130.5904
```

## Predicting with $\hat{u}$

Now we can repeat what we did with the true  $u^*$  value with our estimated one  $\hat{u}$ .



```

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

```

How do our RMSEs compare?

```
c(u.hat=rmse, u=rmse.u)
```

```

##      u.hat      u
## 0.1451248 0.1821815

```

- Indeed, our estimated  $\hat{u}$  version leads to better predictions.
- Clearly there is an identifiability issue in this supremely flexible calibration apparatus; but it does a good job of predicting.