Design for GPs

RSMs and Computer Experiments: Part 3

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

Goals

Design just means: choosing the x-locations where we observe the y's.

We will focus here on how one designs an experiment for GP regression,

- ultimately with an emphasis on **sequential design**
- and later **optimization** under uncertainty.

We will see that the typical GP models encourage space-filling designs

- which are in a sense "model-free".

However, in the sequential context there is a unique opportunity to learn

- about the nature of the response surface via design
- while simultaneously improving the quality of fit.

Space-filling design

Random uniform design

One option for design is random uniform in $[0, 1]^m$.

That is, to fill a design matrix with n runs we could do

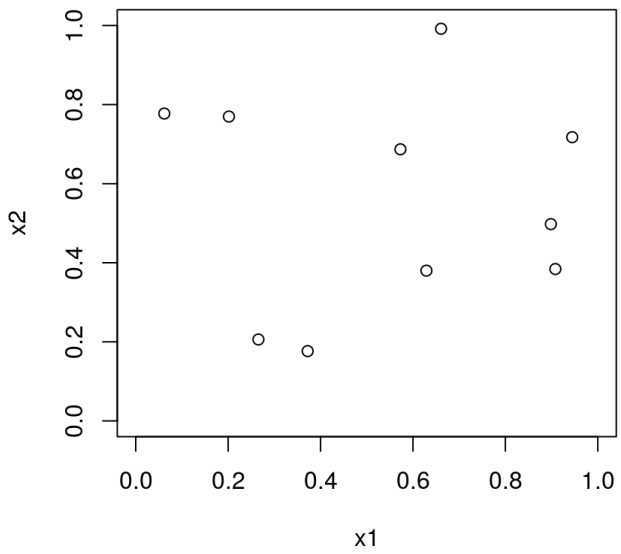
```
m <- 2
n <- 10
X <- matrix(runif(n*m), ncol=m)
colnames(X) <- paste("x", 1:m, sep="")
```

The trouble is, randomness is clumpy.

- With n of any reasonable size you're almost guaranteed to get design locations right next to one another,
- and thus gaps in other parts of the input space.

The clump

```
plot(X, xlim=c(0,1), ylim=c(0,1))
```



Latin hypercube sample

Latin Hypercube samples (LHS) were created to alleviate this problem,

- guaranteeing some spread
- while still being uniform and random.

They do that by dividing the design region evenly into cubes, and then randomly selecting *just one* side of each cube from each input direction.

- For n samples in m inputs there are n^m potential cubes on the grid, from which n are *selected*.
- In 2d the selected cubes resemble Latin squares.

Within each *selected* cube, a single point is sampled uniformly,

- yielding n samples that are reasonably well spaced out.

High-level description

More mathematically and algorithmically …

A Latin hypercube sample (LHS), or design (LHD), D in the design space $[0,1]^m$ can be generated, is an $n \times m$ matrix with $(i,j)^{\text{th}}$ entry

$$d_{ij} = \frac{l_{ij} + (n-1)/2 + u_{ij}}{n}, \quad i = 1, \ldots, n, \; j = 1, \ldots, m,$$

where

- each column l_j, with entries l_{ij}, is a random permutation of n equally spaced levels;
- the u_{ij}'s are independent random numbers in $[0,1]$.
- If instead each u_{ij} is taken to be 0.5 the result is called a **Latin sample**,
    - which will put d_{ij} right in the middle of one of the squares or cubes.

Two-dimensional LHS

The code is pretty straightforward.

- Create the levels.

```
n <- 10
l <- (-(n-1)/2):((n-1)/2)
```

- Put randomly permuted levels into a $n \times m$ matrix

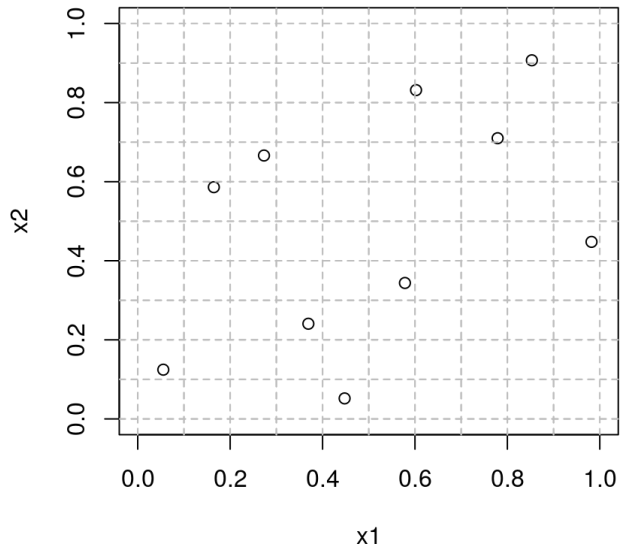
```
m <- 2
L <- matrix(NA, nrow=n, ncol=m)
for(j in 1:m) L[,j] <- sample(l, n)
```

- Create a uniform jitter matrix and combine

```
U <- matrix(runif(n*m), ncol=m)
D <- (L + (n-1)/2 + U)/n
```

Visualizing margins

```
plot(D, xlim=c(0,1), ylim=c(0,1), xlab="x1", ylab="x2")
abline(h=c((L+(n-1)/2)/n,1), col="grey", lty=2)
abline(v=c((L+(n-1)/2)/n,1), col="grey", lty=2)
```



Generic LHS for $[0,1]^m$

Before going to higher dimensions, lets write a function that does it.

```
mylhs <- function(n, m)
  {
    ## generate the Latin hypercube
    l <- (-(n-1)/2):((n-1)/2)
    L <- matrix(NA, nrow=n, ncol=m)
    for(j in 1:m) L[,j] <- sample(l, n)

    ## draw the random uniforms and turn the hypercube into a sample
    U <- matrix(runif(n*m), ncol=m)
    D <- (L + (n-1)/2 + U)/n
    colnames(D) <- paste("x", 1:m, sep="")

    ## return the design and the grid it lives on for visualization
    return(list(D=D, g=c((L+(n-1)/2)/n,1)))
  }
```

What can we expect in higher dimension?

One-dimensional uniformity

For each factor, a LHS has exactly one point in each of the n intervals

$$[0, 1/n), [1/n, 2/n), \ldots, [(n-1)/n, 1).$$

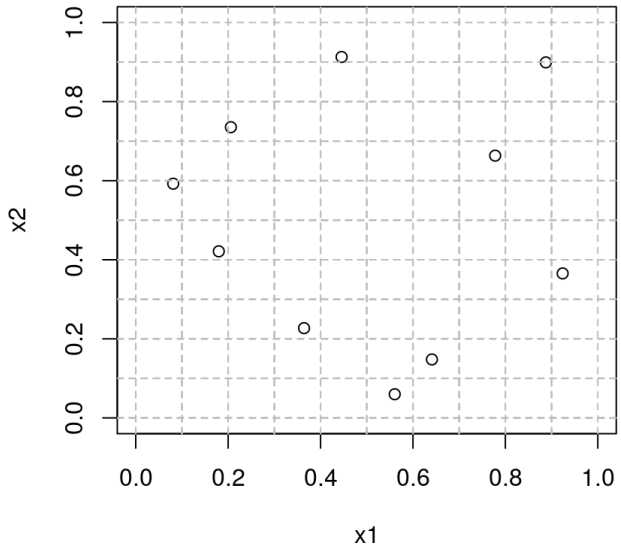
This property is referred to as **one dimensional uniformity**.

As a consequence of that, any projection into lower dimensions that can be obtained by dropping some of the coordinates will also have a uniform distribution,

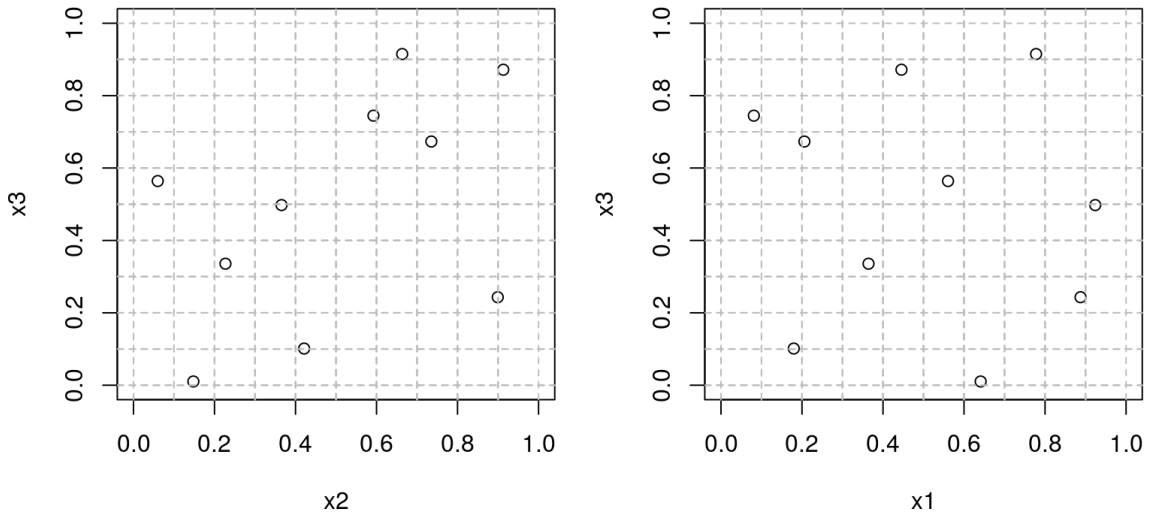
- and will *also be an LHS* in that lower dimension.

Three-dimensional LHS

```
Dlist <- mylhs(10, 3)
plot(Dlist$D[,1:2], xlim=c(0,1), ylim=c(0,1), xlab="x1", ylab="x2")
abline(h=Dlist$g, col="grey", lty=2)
abline(v=Dlist$g, col="grey", lty=2)
```

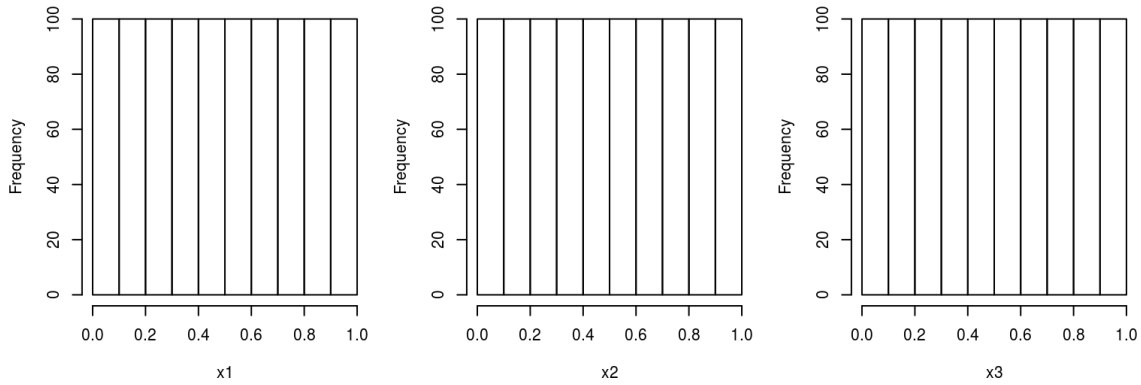



```
par(mfrow=c(1,2))
plot(Dlist$D[,2:3], xlim=c(0,1), ylim=c(0,1), xlab="x2", ylab="x3")
abline(h=Dlist$g, col="grey", lty=2)
abline(v=Dlist$g, col="grey", lty=2)
plot(Dlist$D[,c(1,3)], xlim=c(0,1), ylim=c(0,1), xlab="x1", ylab="x3")
abline(h=Dlist$g, col="grey", lty=2)
abline(v=Dlist$g, col="grey", lty=2)
```



Histogram of margins

```
X <- mylhs(1000, 3)$D
par(mfrow=c(1,3))
hist(X[,1], main="", xlab="x1")
hist(X[,2], main="", xlab="x2")
hist(X[,3], main="", xlab="x3")
```



- All uniform.

Other marginals

The LHS can be extended to marginals other than the uniform distribution.

1. Generate an ordinary LHS, and then
2. apply an inverse CDF (quantile function) of your choice on each of the marginals.

This will cause the marginals to take distributions with those CDFs,

- but at the same time ensure that there is not “more clumpiness than necessary” in the joint distribution.

The simplest application of this is to re-scale to a custom hyper-rectangle.

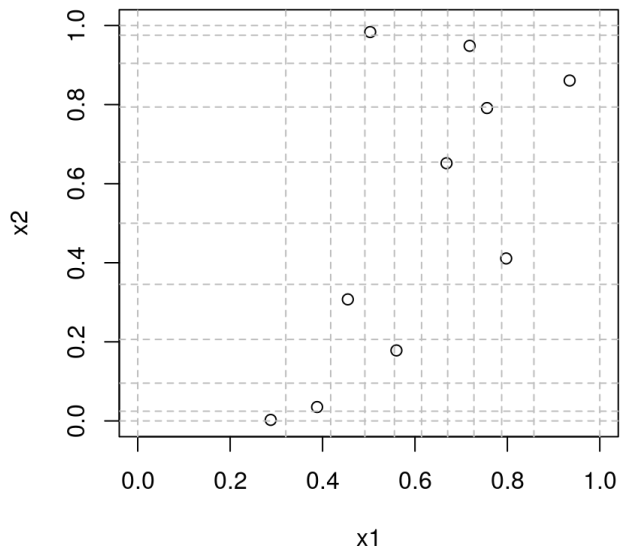
- The code on the next slide generalizes `mylhs` to utilize Beta distributed marginals, of which the uniform is a special case.

Beta marginals

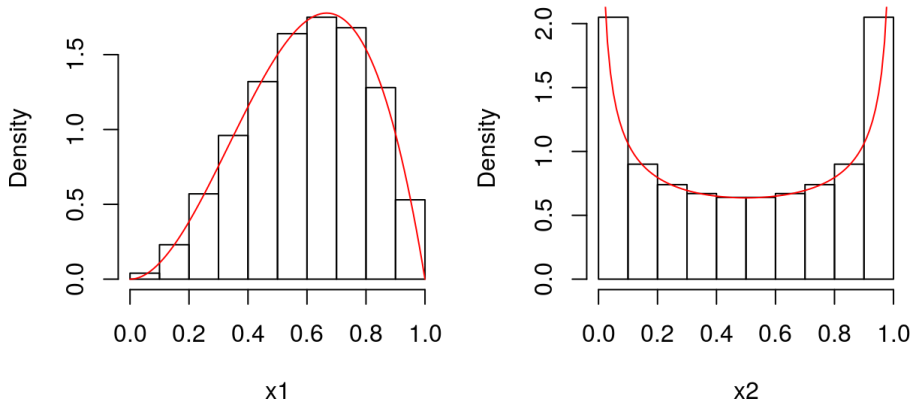
```
mylhs.beta <- function(n, m, shape1, shape2)
  {
    ## generate the Latin Hypercube and turn it into a sample
    l <- (-(n-1)/2):((n-1)/2)
    L <- matrix(NA, nrow=n, ncol=m)
    for(j in 1:m) L[,j] <- sample(l, n)
    U <- matrix(runif(n*m), ncol=m)
    D <- (L + (n-1)/2 + U)/n

    ## calculate the grid for that design
    g <- (L + (n-1)/2)/n
    g <- rbind(g, 1)

    for(j in 1:m) { ## redistrbute according to beta quantiles
        D[,j] <- qbeta(D[,j], shape1[j], shape2[j])
        g[,j] <- qbeta(g[,j], shape1[j], shape2[j])
    }
    colnames(D) <- paste("x", 1:m, sep="")

    ## return the design and the grid it lives on for visualization
    return(list(D=D, g=g))
  }
```

An illustration with Beta marginals.

```
Dlist <- mylhs.beta(10, 2, shape1=c(3,1/2), shape2=c(2,1/2))
plot(Dlist$D, xlim=c(0,1), ylim=c(0,1), xlab="x1", ylab="x2")
abline(v=Dlist$g[,1], col="grey", lty=2)
abline(h=Dlist$g[,2], col="grey", lty=2)
```




```
X <- mylhs.beta(1000, 2, shape1=c(3,1/2), shape2=c(2,1/2))$D
par(mfrow=c(1,2))
x <- seq(0,1,length=100)
hist(X[,1], main="", xlab="x1", freq=FALSE)
lines(x, dbeta(x, 3, 2), col=2)
hist(X[,2], main="", xlab="x2", freq=FALSE)
lines(x, dbeta(x, 1/2, 1/2), col=2)
```



- As prescribed.

A drawback

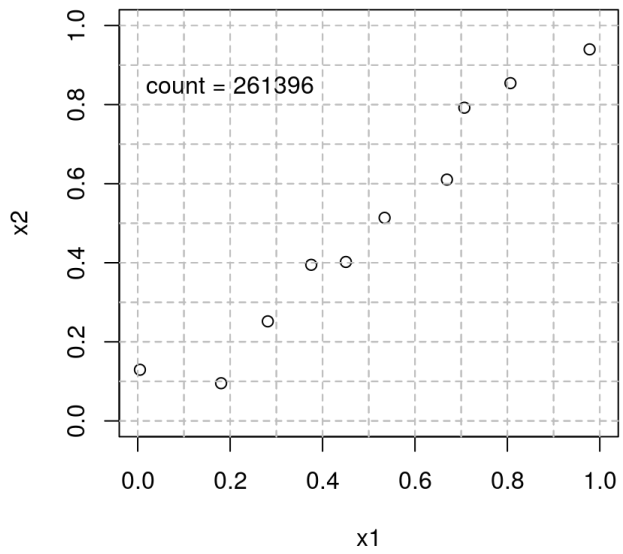
One drawback of LHS samples is that,

- although a degree of space-fillingness in guaranteed at the margins,
- there is no guarantee that the resulting design won't be somehow otherwise "aliased"
    - i.e., revealing patterns or clumpiness in other aspects of the variables' *joint* distribution.

For example, the code on the next slide repeatedly draws LHSs with $n = 10, m = 2$, stopping when one is found where

- both coordinates are roughly increasing,
- i.e., lining up along a jittered diagonal.

```
count <- 0
while(1) { count <- count+1; Dlist <- mylhs(10, 2)
    o <- order(Dlist$D[,1]); x <- Dlist$D[o,2]
    if(all(x[1:(n-1)] < x[2:n] + 1/20)) break }
plot(Dlist$D, xlim=c(0,1), ylim=c(0,1), xlab="x1", ylab="x2")
text(0.2, 0.85, paste("count =", count))
abline(v=Dlist$g, col="grey", lty=2); abline(h=Dlist$g, col="grey", lty=2)
```



Lots of patterns

It takes a while to find that pattern,

- but that's just one of many patterns,
- and in high dimensions (big m relative to n) everything looks like some kind of pattern.

One of the problems is that, despite attractive properties (besides being easy to compute)

- a condition on the marginals is a crude way to spread things out,
- with diminishing returns in increasing dimensions.

Another problem is the randomness of it,

- or is that a double-edged sword?

Maximizing minimum distance

If what we want is points to be spread out, then perhaps it makes sense to deliberately spread points out!

For that we need a notion of distance so we can measure spread,

- but lets stick with the simplest choice of Euclidean distance, for now.

$$d(x, x') = \sum_{j=1}^{m} (x_j - x'_j)^2$$

Now, how about a design of $X = \{x_1, \ldots, x_n\}$ which maximizes the minimum distance between all pairs of points, a so-called **maximin design**.

$$\max_X \min\{d(x_i, x_k) : i, k = 1, \ldots, n\}$$

Calculation

The details are all in the algorithm.

- $\max_X$ is effectively a maximization over n "dimensions",
- with a criteria that is at best $O(n^2)$ to calculate for each choice of X.

Although the criteria suggests a deterministic solution for a particular choice of n and m,

- in fact most solvers are stochastic, owing to the challenge of the optimization setting:
  - proposing a random change to one of n the x_i runs,
  - and accepting or rejecting that change by consulting the criteria.

The next slide implements a (very simplistic) version of such a search.

Stochastic search

A random starting point.

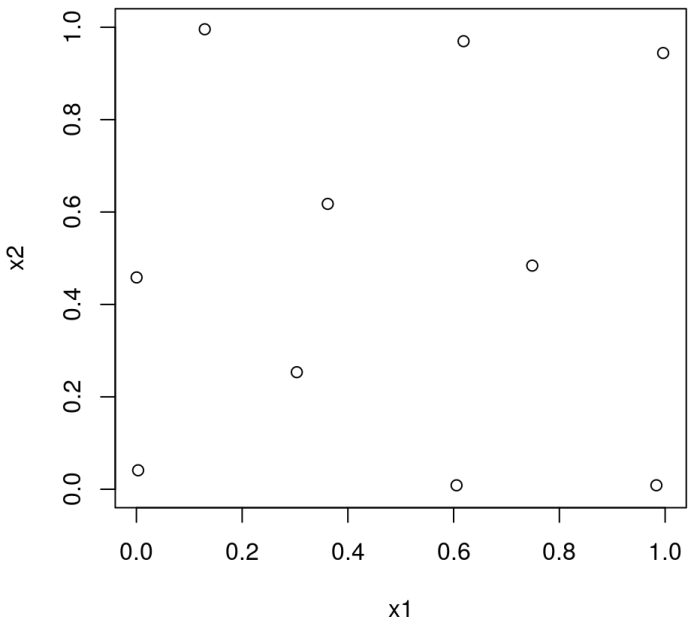
```
library(plgp)
m <- 2; n <- 10
X1 <- matrix(runif(n*m), ncol=m)
dX1 <- distance(X1)
md1 <- min(dX1)
```

Random proposals which are accepted or rejected.

```
T <- 10000
for(t in 1:T) {
    row <- sample(1:n, 1)
    xold <- X1[row,]       ## random row selection
    X1[row,] <- runif(m)   ## random new row
    dprime <- distance(X1)
    dprime <- as.numeric(dprime[upper.tri(dprime)])
    mdprime <- min(dprime)
    if(mdprime > md1) { md1 <- mdprime  ## accept
    } else { X1[row,] <- xold }         ## reject
}
```

Pretty good actually; but points are pushed to the boundaries.

```
plot(X1, xlim=c(0,1), ylim=c(0,1), xlab="x1", ylab="x2")
```



Sequential maximin design

One nice feature of maximin designs is that they naturally lend themselves to sequential application.

That is, we can

- condition on an existing design `Xorig`,
- and ask what new runs could augment that design to produce `X`
- where the new runs optimize the `maximin` criterion.

It is trivial to adapt our algorithm to take an existing design into account.

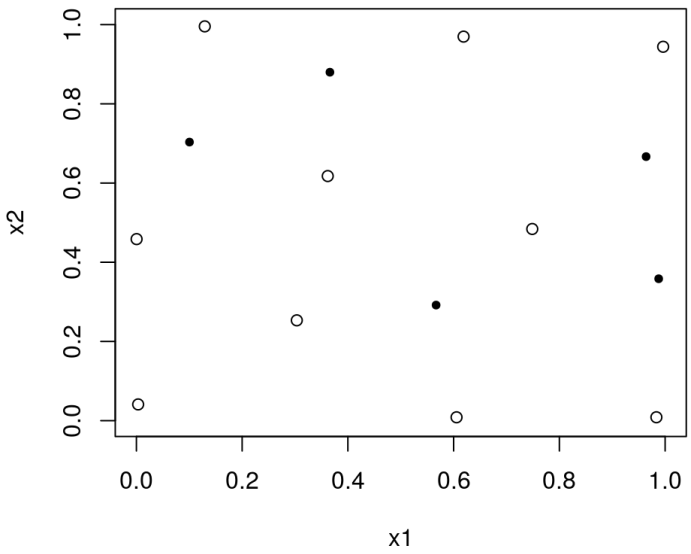
- Lets encapsulate the idea in a function called `maximin` so we can use it later …

Implementation

```
maximin <- function(n, m, T=100000, Xorig=NULL) {

    X <- matrix(runif(n*m), ncol=m)  ## initial design
    X <- rbind(X, Xorig)             ## This is the only change!
    d <- distance(X)
    d <- as.numeric(d[upper.tri(d)])
    md <- min(d)

    for(t in 1:T) {
        row <- sample(1:n, 1)
        xold <- X[row,]       ## random row selection
        X[row,] <- runif(m)   ## random new row
        dprime <- distance(X)
        dprime <- as.numeric(dprime[upper.tri(dprime)])
        mdprime <- min(dprime)
        if(mdprime > md) { md <- mdprime  ## accept
        } else { X[row,] <- xold }        ## reject
    }

    return(X)
}
```

The five new points avoind the previous ten, and avoid themselves.

```
X2 <- maximin(5, 2, Xorig=X1)
plot(X1, xlim=c(0,1), ylim=c(0,1), xlab="x1", ylab="x2")
points(X2[1:5,], pch=20)
```



Model-based design

Tell me what you want

A model-based design is one where

- the model says what x-values it wants subject to some criteria:
- in order to learn its parameters, or minimize mean-squared prediction error.

For example, with a linear model,

- you may recall that maximizing spread in the x's maximizes leverage and therefore minimizes the standard error of $\hat{\beta}$-values,
- (for better or worse).

With GPs (a nonparametric model), we might fix a hyperparamerization and

1. choose a x-values that "maximize learning" from prior to posterior, or
2. which minimize predictive variance over the input space.

Lets consider those two options in turn.

Maximum entropy designs

The **entropy** of a density $p(x)$ is defined as

$$H(X) = -\int_{\mathcal{X}} p(x) \log p(x)\ dx.$$

- Entropy is larger when the density $p(x)$ is more uniform (more surprise).
- **Information** is the negative of entropy, $I = -H$, i.e., more information is less uniformity (less surprise).

It makes sense to maximize the information change from prior to posterior, i.e., $I - I_{D_n}$.

- A **maximum entropy design** maximizes $-I_{D_n}$.

Analytic entropy for MVN

Wikipedia says (https://en.wikipedia.org/wiki/Multivariate_normal_distribution#Entropy) that the entropy of an n-variate MVN with parameters μ and Σ is

$$\frac{n}{2}\log 2\pi e + \frac{1}{2}\log |\Sigma|.$$

Therefore the entropy of Y_n observed at X_n is maximized

- when $|K_n|$ is maximized.
- And K_n depends on our design X_n.

Observe that this *also* depends on the hyperparameterization of the covariance:

- lengthscale(s) θ and nugget g, although the role of g is complicated.
- So it might make sense to do an initial experiment to choose these values.

A quick implementation

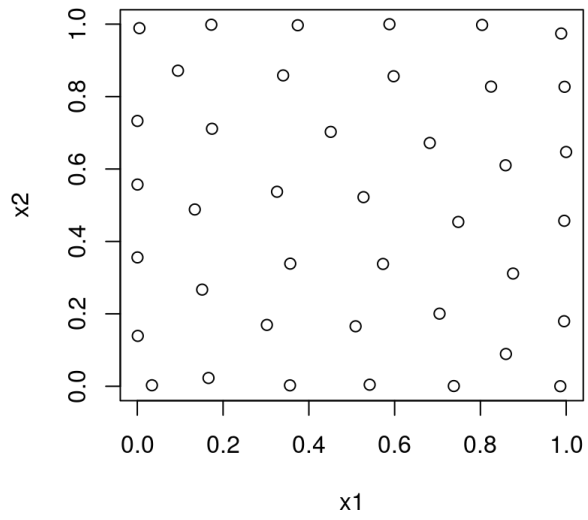
We can easily modify our `maximin` algorithm to optimize $|K_n|$.

```
maxent <- function(n, m, theta=0.1, g=0.01, T=100000)
  {
    if(length(theta) == 1) theta <- rep(theta, m)
    X <- matrix(runif(n*m), ncol=m)
    K <- covar.sep(X, d=theta, g=g)
    Kdet <- determinant(K, logarithm=TRUE)$modulus

    for(t in 1:T) {
        row <- sample(1:n, 1)
        xold <- X[row,]
        X[row,] <- runif(m)
        Kprime <- covar.sep(X, d=theta, g=g)
        Kdetprime <- determinant(Kprime, logarithm=TRUE)$modulus
        if(Kdetprime > Kdet) { Kdet <- Kdetprime
        } else { X[row,] <- xold }
    }
    return(X)
}
```

For example

In two dimensions.

```
X <- maxent(40, 2)
plot(X, xlab="x1", ylab="x2")
```



Space-filling

Actually, its not much different than a maximin design:

- all spaced out, with lots of points on the boundary,
- at least qualitatively.

That's because we used a covariance matrix that is isotropic: radially symmetric.

Perhaps if we choose some crazy kernel (so long as it is positive definite),

- I suppose we could end up a very interesting looking maximum entropy design.

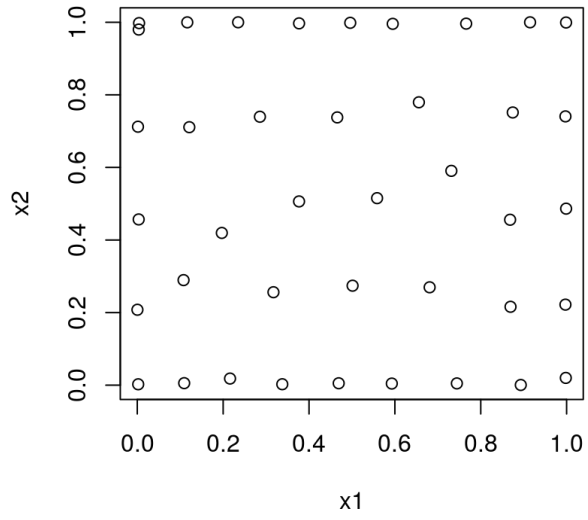
In a more common separable (Gaussian) setup,

- we will sensibly get more spread in some directions than others.
- I.e., presuming we knew that already, before we collected any data!

Separable version

Longer x_2 lengthscales than x_1.

```
X <- maxent(40, 2, theta=c(0.1, 0.5))
plot(X, xlab="x1", ylab="x2")
```



Minimizing predictive variance

As another criteria, consider the predictive variance.

At a particular location $x \in \mathcal{X}$, the predictive variance is

$$\sigma^2(x) = \tau^2[1 + g - k_n^\top(x) K_n^{-1} k_n(x)] \quad \text{where} \quad k_n(x) \equiv k(X_n, x).$$

- This is the same as what some call the **mean-squared prediction error** (MSPE):

$$\text{MSPE}[\hat{y}(x)] = \mathbb{E}\{(\hat{y} - Y(x))^2\} \equiv \sigma^2(x)$$

Now, the **Integrated MSPE (IMSPE)** criterion is defined to be the MSPE (divided by τ^2) "averaged" over $\mathcal{X}$, which we express below as a function of the design X_n.

$$J(X_n) = \int_{\mathcal{X}} \frac{\sigma^2(x)}{\tau^2} w(x)\, dx$$

where $w(\cdot)$ is a specified nonnegative weight function satisfying $\int_{\mathcal{X}} w(x)\, dx = 1$.

Comments on the IMSE criteria

This is an m-dimensional integral for m-dimensional $\mathcal{X}$, however

- there are analytic forms when $\mathcal{X}$ is rectangular, and where the covariance structure follows a familiar form (e.g., isotropic or separable Gaussian);
- and there are even closed form derivatives.

Again, the criteria depends on the θ hyperparameter: and again it is $\mathcal{O}(n^3)$.

- Similar shortcuts can reduce it to $\mathcal{O}(n^2)$ per iteration; more later.

The IMSPE criteria is a generalization of the classical A-optimality criteria.

An approximation

An effective approximation replaces the integral with a sum over a "grid" in $\mathcal{X}$,

- possibly incorporating the weights $w(\cdot)$ directly via custom "grid".
- The "grid" could be random, or itself follow a space-filling construction.

The subroutine below

- calculates the predictive variance at reference locations `Xref` for design `X` $= X_n$
- and then approximates the integral by a mean over `Xref` .

```
imspe.criteria <- function(X, Xref, theta, g)
{
    K <- covar.sep(X, d=theta, g=g)
    Ki <- solve(K)
    KXref <- covar.sep(X, Xref, d=theta, g=0)
    return(mean(1+g - diag(t(KXref) %*% Ki %*% KXref)))
}
```

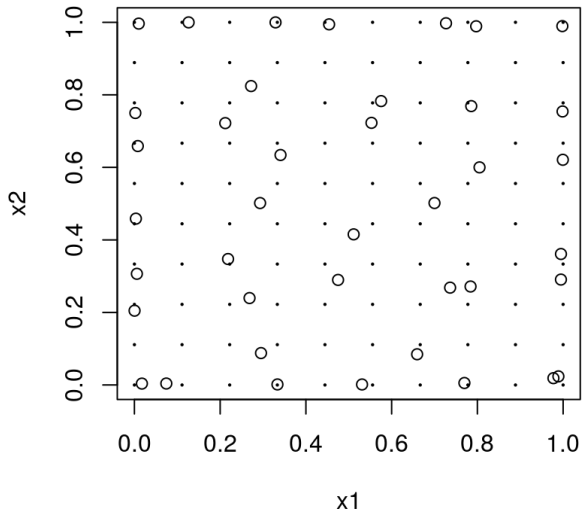
IMSE design implementation

Copying from `maxent` ; but careful to minimize!

```
imspe <- function(n, m, Xref, theta=0.5, g=0.01, T=100000)
  {
    if(length(theta) == 1) theta <- rep(theta, m)
    X <- matrix(runif(n*m), ncol=m)
    I <- imspe.criteria(X, Xref, theta, g)

    for(t in 1:T) {
        row <- sample(1:n, 1)
        xold <- X[row,]
        X[row,] <- runif(m)
        Iprime <- imspe.criteria(X, Xref, theta, g)
        if(Iprime < I) { I <- Iprime
        } else { X[row,] <- xold }
    }
    return(X)
}
```

Try IMSPE in 2d.

```
g <- expand.grid(seq(0,1,length=10), seq(0,1, length=10))
X <- imspe(40, 2, Xref=g)
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
points(g, pch=20, cex=0.25)
```



Underwhelming

Apparently another variation on a space-filling design.

However, observe that

- The design locations really want to be close to the reference set (solid dots).

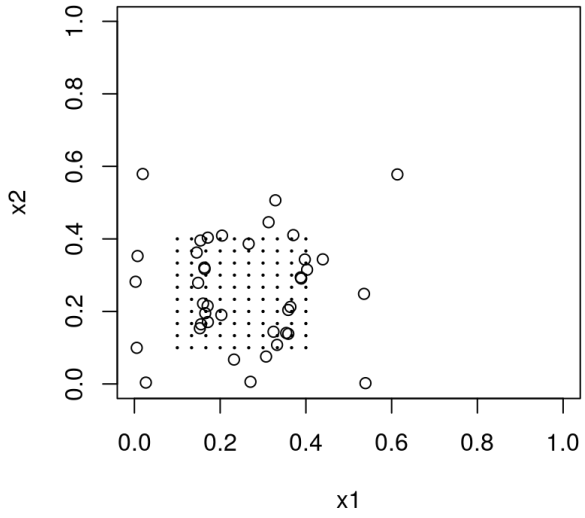
That makes the search more “discrete” than the maximum entropy one, say.

- Compared to the candidate it is replacing, proposed new design elements must *both* be
    - more spread out relative to the other design elements
    - *and* just as close to a reference location.

Our `Xref` “grid” probably needs to be finer to get a better approximation in this case.

What happens if we make the reference set occupy a smaller space; effectively tweaking $w(\cdot)$?

```
gsmall <- expand.grid(seq(0.1,0.4,length=10), seq(0.1, 0.4, length=10))
X <- imspe(40, 2, Xref=gsmall)
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
points(gsmall, pch=20, cex=0.25)
```



Curious result

What an interesting pattern!

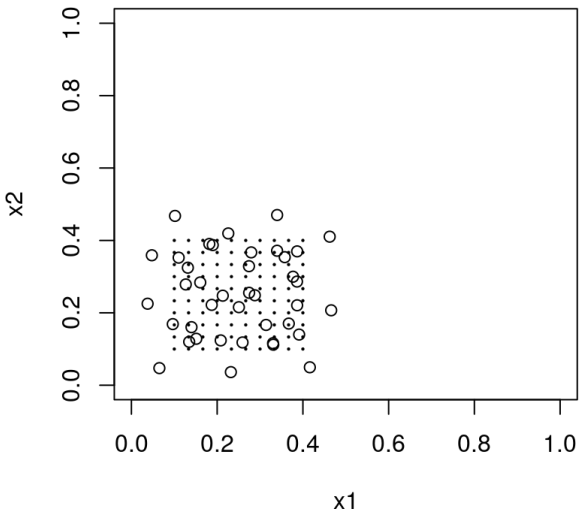
- It looks like the design is trying to “encircle” the reference set
- and at multiple scales, i.e., diameters of encirclement.

How could it be that those points way out there are useful?

- Its the lengthscale, θ!

Smaller lengthscale?

```
X <- imspe(40, 2, theta=0.1, Xref=gsmall)
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
points(gsmall, pch=20, cex=0.25)
```



Chicken or egg?

Yup, that does the trick.

- But we still have points that are outside the reference set region.

That makes sense though, when you ponder it a bit.

- We have a stationary model, meaning that spread (on multiple scales) in the design is a plays a key role,
- even if we’re only interested in predicting locally, say in a sub region of the input space.

The amount of spread that the design criteria “wants” depends on the covariance structure,

- and in particular on assumed values for the lengthscale hyperparameter.

So we shouldn’t bother with model-based design until we’ve fit the model?

Sequential design

Not just easier

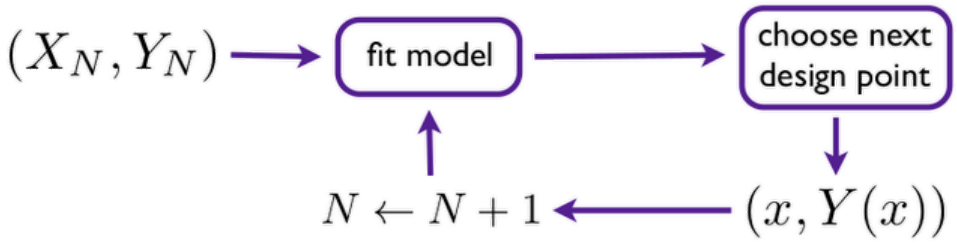
A **greedy** approach is more useful, i.e., more *practical*,

- and as a bonus there are connections to the non-sequential analog.
- These are framed as “approximations”, but to the setting where the “true” parameterization is known (which it never is).

Proceeding sequentially is also computationally more reasonable

- and some of the extra computation compared to alternatives,
    - such as decomposing matrices,
- have justifiable value in terms of the guidance that those (estimated) quantities provide in the searches that target *both* modeling and design goals.

The cartoon



1. Start with a small space-filling design.
2. Fit a model, say a GP.
3. Evaluate some criteria from the model on a candidate set of inputs,
    - or otherwise optimize the criteria.
4. Observe the response at the chosen location.
5. Repeat.

Whack-a-mole

The simplest sequential design scheme for GPs (but it works well beyond that) involves

- choosing the next point to maximize the predictive variance.

I.e., if you have data $D_n = (X_n, Y_n)$,

- infer the unknown hyperparamters τ^2, θ, g by maximizing the likelihood, say;
- choose x_{n+1} as

$$x_{n+1} = \text{argmax}_{x \in \mathcal{X}} \; \sigma^2(x)$$

- obtain $y_{n+1} = Y(x_{n+1}) = f(x_{n+1}) + \varepsilon$;
- combine to form the new data set: $D_{n+1} = ([X_n; x_{n+1}^\top], [Y_n; y_{n+1}])$;
- repeat.

A good idea?

- Mostly, yes.

MacKay (1992) (www.inference.eng.cam.ac.uk/mackay/selection.nc.ps.gz) argues that, in a certain sense, the repeated application of such a selection approximates a maximum information design.

Seo et al., (2000) (https://pdfs.semanticscholar.org/9687/e167e4175c9cdd4fce236fa2d049fb0f93ed.pdf) provided the first Gaussian Process treatment

- Calling the method **Active Learning MacKay (ALM)**,
- and that has stuck for many, including me.

The ML community has been much more “active” in sequential design than the stats community has,

- but they don’t call it design.

A little demo

Lets work again with our favorite 2d data set, and start with a small LHS.

```
ninit <- 12
library(lhs)
X <- randomLHS(ninit, 2)
f <- function(X, sd=0.01) {
    X[,1] <- (X[,1] - 0.5)*6 + 1
    X[,2] <- (X[,2] - 0.5)*6 + 1
    y <- X[,1] * exp(-X[,1]^2 - X[,2]^2) + rnorm(nrow(X), sd=sd)
}
y <- f(X)
```

And fit an initial (isotropic) GP model.

```
library(laGP)
eps <- sqrt(.Machine$double.eps)
gpi <- newGP(X, y, d=0.1, g=0.1*var(y), dK=TRUE)
g <- garg(list(mle = TRUE), y)
d <- darg(list(mle = TRUE, max=0.5), X)
mle <- jmleGP(gpi, c(d$min, d$max), c(g$min, g$max), d$ab, g$ab)
```

Prediction

Lets create some predictive quantities for later evaluation.

```
x1 <- x2 <- seq(0,1,length=100)
XX <- expand.grid(x1, x2)
ytrue <- f(XX, sd=0)
```

And save the predictive accuracy from our initial design.

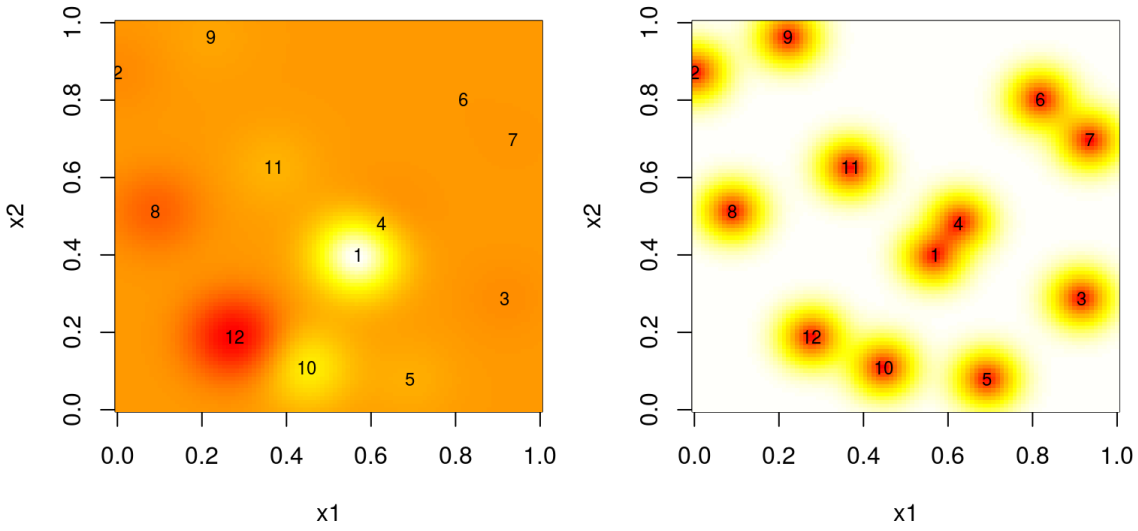
```
p <- predGP(gpi, XX, lite=TRUE)
rmse <- sqrt(mean((ytrue - p$mean)^2))
```

We'll track improvements to RMSE as a function of the design size,

- and use that to make comparisons to sequential designs derived from similar criteria.

Starting out …

```
par(mfrow=c(1,2))
image(x1, x2, matrix(p$mean, ncol=length(x1)), col=heat.colors(128))
text(X, labels=1:nrow(X), cex=0.75)
image(x1, x2, matrix(sqrt(p$s2), ncol=length(x1)), col=heat.colors(128))
text(X, labels=1:nrow(X), cex=0.75)
```



Multi-start scheme

There are lots of peaks and valleys in the variance surface;

- remember those "footballs"?
- So finding global optima could be a little challenging.

When deploying a **mult-start scheme**, it could help to "design" starting locations in parts of the input space known to have high variance

- i.e., at the widest part of the "football"—as far as possible from existing design locations.

Good rules of thumb include:

1. Place starts within the bounding box of the current design (if that is easy).
    - The search will naturally "head outside" because variance is high there.
2. Have about as many starting locations as there are current design points, n.

Objective and search

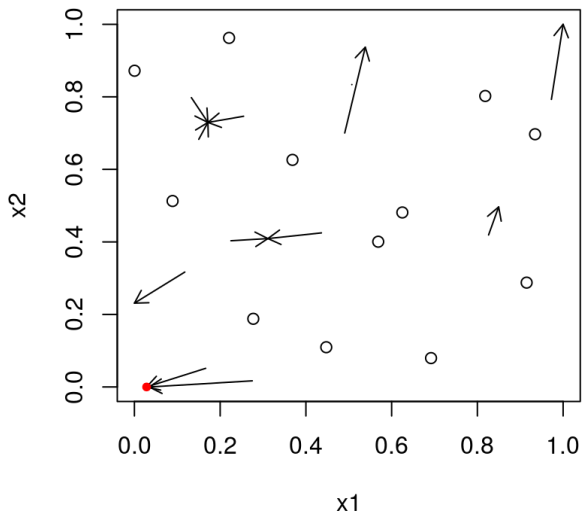
Here is the objective function in code for `optim` .

```
obj.alm <- function(x, gpi) - sqrt(predGP(gpi, matrix(x, nrow=1), lite=TRUE)$s2)
```

Here is the search routine, calling `optim` on a `maximin` set of starting values

```
alm.search <- function(X, gpi)
  {
    start <- maximin(nrow(X), 2, T=100*nrow(X), Xorig=X)[1:nrow(X),]
    xnew <- matrix(NA, nrow=nrow(start), ncol=ncol(X)+1)
    for(i in 1:nrow(start)) {
        out <- optim(start[i,], obj.alm, method="L-BFGS-B",
            lower=0, upper=1, gpi=gpi)
        xnew[i,] <- c(out$par, -out$value)
    }
    solns <- data.frame(cbind(start, xnew))
    names(solns) <- c("s1", "s2", "x1", "x2", "val")
    return(solns)
  }
```

```
solns <- alm.search(X, gpi)
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
suppressWarnings(arrows(solns$s1, solns$s2, solns$x1, solns$x2, length=0.1))
m <- which.max(solns$val)
prog <- solns$val[m]
points(solns$x1[m], solns$x2[m], col=2, pch=20)
```



Next iteration

Incorporate the new data at the chosen input location.

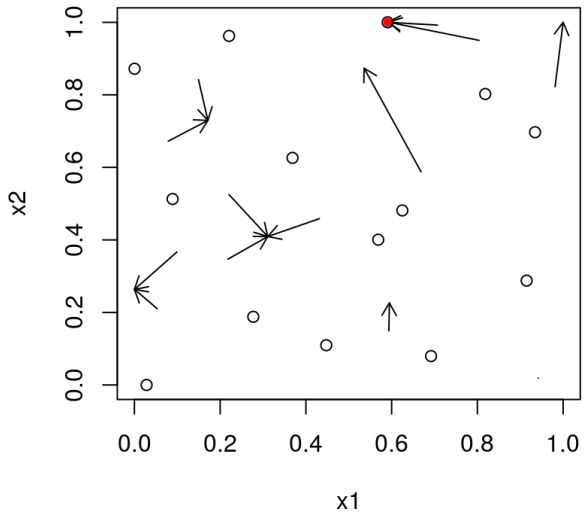
```
xnew <- as.matrix(solns[m,3:4])
X <- rbind(X, xnew)
y <- c(y, f(xnew))
updateGP(gpi, xnew, y[length(y)])
mle <- rbind(mle, jmleGP(gpi, c(d$min, d$max), c(g$min, g$max), d$ab, g$ab))
rmse <- c(rmse, sqrt(mean((ytrue - predGP(gpi, XX, lite=TRUE)$mean)^2)))
```

And do another iteration, and update.

```
solns <- alm.search(X, gpi)
m <- which.max(solns$val); prog <- c(prog, solns$val[m])
xnew <- as.matrix(solns[m,3:4])
X <- rbind(X, xnew)
y <- c(y, f(xnew))
updateGP(gpi, xnew, y[length(y)])
mle <- rbind(mle, jmleGP(gpi, c(d$min, d$max), c(g$min, g$max), d$ab, g$ab))
rmse <- c(rmse, sqrt(mean((ytrue - predGP(gpi, XX, lite=TRUE)$mean)^2)))
```

The outcome of the second iteration of search.

```
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
suppressWarnings(arrows(solns$s1, solns$s2, solns$x1, solns$x2, length=0.1))
m <- which.max(solns$val)
points(solns$x1[m], solns$x2[m], col=2, pch=20)
```



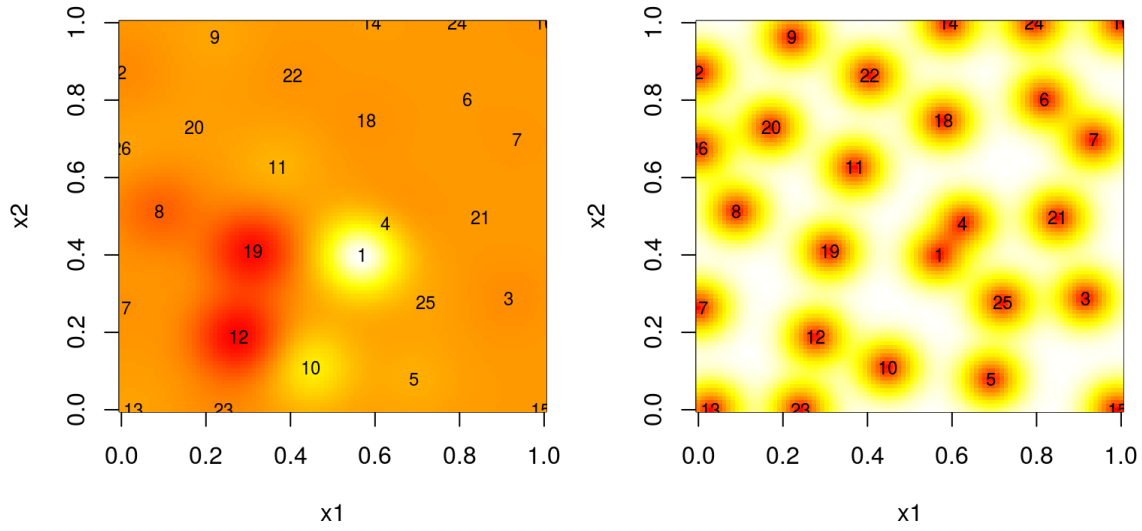
More iterations

... for a total of $n = 25$ runs.

```
for(i in nrow(X):25) {
    solns <- alm.search(X, gpi)
    m <- which.max(solns$val); prog <- c(prog, solns$val[m])
    xnew <- as.matrix(solns[m,3:4])
    X <- rbind(X, xnew); y <- c(y, f(xnew))
    updateGP(gpi, xnew, y[length(y)])
    mle <- rbind(mle, jmleGP(gpi, c(d$min, d$max), c(g$min, g$max), d$ab, g$ab))
    p <- predGP(gpi, XX, lite=TRUE)
    rmse <- c(rmse, sqrt(mean((ytrue - p$mean)^2)))
}
mle[seq(1,nrow(mle),by=4),]
```

```
##             d            g tot.its dits gits
## 1  0.01122657 0.0005610943      25   20    5
## 5  0.01005402 0.0005610682       6    4    2
## 9  0.01098673 0.0005611416       6    4    2
## 13 0.01061589 0.0005611527       5    4    1
```

```
p <- predGP(gpi, XX, lite=TRUE)
par(mfrow=c(1,2))
image(x1, x2, matrix(p$mean, ncol=length(x1)), col=heat.colors(128))
text(X, labels=1:nrow(X), cex=0.75)
image(x1, x2, matrix(sqrt(p$s2), ncol=length(x1)), col=heat.colors(128))
text(X, labels=1:nrow(X), cex=0.75)
```



Pretty good

That's pretty good,

- but again possibly not better than a simple space-filling design to start with.

One nice feature is that we get a measure of progress,

- namely the maximal variance used as the basis of each sequential design decision.

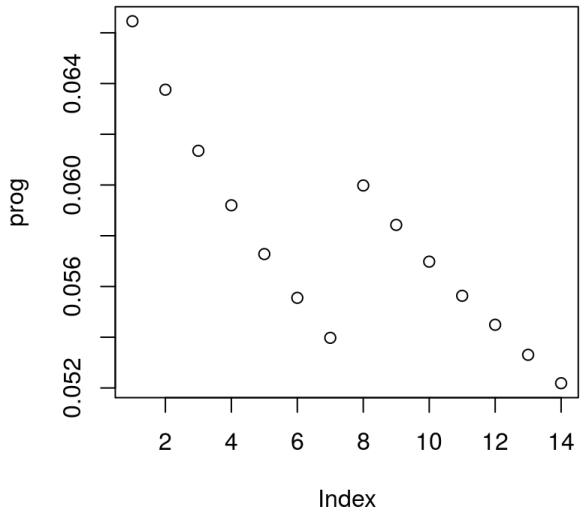
Note that when we use `predGP(...)$s2` we are taking $\hat{\tau}^2$ into account.

- This doesn't matter for the selection of each point,
- but it does mean that if the estimate of scale changes throughout the iterations, there may be "jumps" in an associated progress metric.
- At this time, the `laGP` package does not output $\hat{\tau}^2$ in order to "correct" this, if desired.

Visualizing progress

Perhaps things haven't leveled off yet, and we need a larger design?

```
plot(prog)
```



More runs …

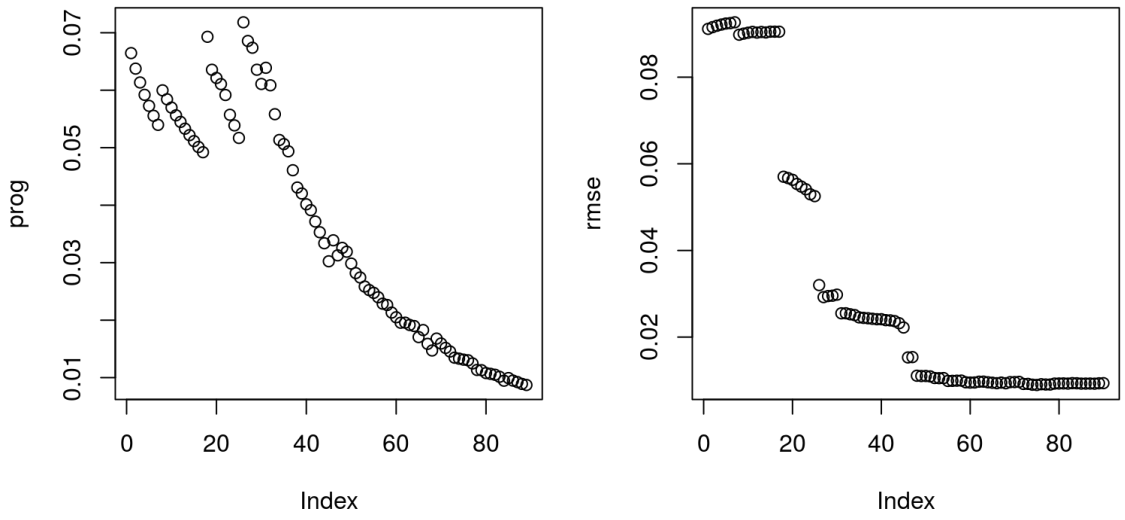
```
for(i in nrow(X):100) {
    solns <- alm.search(X, gpi)
    m <- which.max(solns$val); prog <- c(prog, solns$val[m])
    xnew <- as.matrix(solns[m,3:4])
    X <- rbind(X, xnew); y <- c(y, f(xnew))
    updateGP(gpi, xnew, y[length(y)])
    mle <- rbind(mle, jmleGP(gpi, c(d$min, d$max), c(g$min, g$max), d$ab, g$ab))
    p <- predGP(gpi, XX, lite=TRUE)
    rmse <- c(rmse, sqrt(mean((ytrue - p$mean)^2)))
}
mle[seq(1,nrow(mle), by=20),]
```

```
##             d            g tot.its dits gits
## 1  0.01122657 0.0005610943      25   20    5
## 21 0.02811070 0.0005629620       7    4    3
## 41 0.03303344 0.0005480433       7    4    3
## 61 0.03510937 0.0004897129      10    6    4
## 81 0.03465657 0.0005457987      15    7    8
```

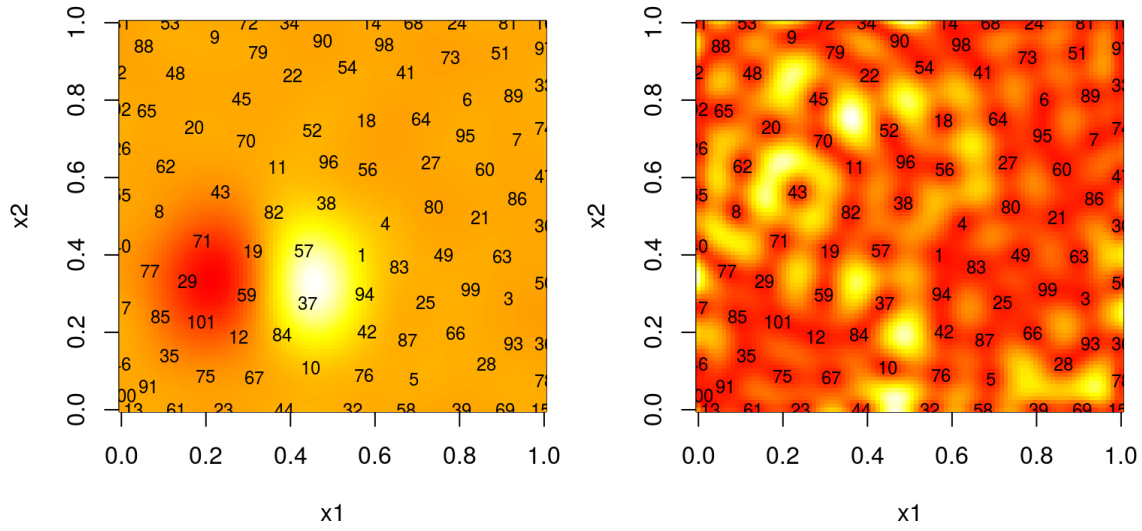
Progress update

Things are starting to level off a bit.

```
par(mfrow=c(1,2))
plot(prog); plot(rmse)
```




```
x2 <- x1 <- seq(0,1, length=100); XX <- expand.grid(x1, x2)
p <- predGP(gpi, XX, lite=TRUE)
par(mfrow=c(1,2))
image(x1, x2, matrix(p$mean, ncol=length(x1)), col=heat.colors(128))
text(X, labels=1:nrow(X), cex=0.75)
image(x1, x2, matrix(sqrt(p$s2), ncol=length(x1)), col=heat.colors(128))
text(X, labels=1:nrow(X), cex=0.75)
```



Pathalogical behavior

Sometimes ALM does misbehave

- and it has to do with the starting design.

If the starting design misses the “bumps” in the data,

- the “fit” will see all noise and no signal,
- and effectively the variance surface will be flat, with perhaps a slight increase at the edges of the space.

In that case, the sequential design will degenerate, primarily focusing on the edges of the design space, like with design for OLS.

Solutions include

- a strong prior emphasizing signal (low lengthscales) over noise (high nuggets);
- a bigger initial design, hoping for the best; or
- a more aggregate criteria.

Aggregate criteria

What else could we do?

Instead of maximizing variance,

- which ignores that sequential design choices have a more *global* impact,

you could maximize **reduction in variance** instead, averaged over the input space.

- The first person to suggest such designs in a nonparametric regression context was Cohn (1996) (http://www.cs.cmu.edu/~cohn/psyche/AIM-1491.ps.Z), for neural networks.
- Seo et al., (2000) (https://pdfs.semanticscholar.org/9687/e167e4175c9cdd4fce236fa2d049fb0f93ed.pdf) adapted it GPs and called it **active learning Cohn (ALC)**.

This is *almost* the sequential analog of IMSPE design,

- and in fact the sequential version approximates a full A-optimal design.

Deduced (reduced) variance

Recall that the predictive variance follows

$$\sigma_n^2(x) = \hat{\tau}^2[1 + \hat{g} - k_n^\top(x) K_n^{-1} k_n(x)] \quad \text{where} \quad k_n(x) \equiv k(X_n, x),$$

- written with an n subscript to emphasize dependence on Y_n values
- via $\hat{\tau}^2$, $\hat{g}$, and $\hat{\theta}$ (hidden in K_n).

Now, let $\sigma_{n+1}^2(x)$ be the *deduced* variance based on a new x_{n+1} in put location,

- but otherwise conditioning on $\hat{\tau}^2$, $\hat{g}$, and $\hat{\theta}$ estimated via Y_n.
- Since Y_n does not directly appear in the $\sigma_n^2(x)$, nor would it in $\sigma_{n+1}^2(x)$.

Therefore, quite simply

$$\sigma^2_{n+1}(x) = \hat{\tau}^2[1 + \hat{g} - k^\top_{n+1}(x)K^{-1}_{n+1}k_{n+1}(x)] \quad \text{where} \quad k_{n+1}(x) \equiv k(X_{n+1}, x),$$

- where X_{n+1} has $x^\top_{n+1}$ in its $n+1^{\text{st}}$ row.

ALC criteria

Then, the ALC criteria is

$$\begin{aligned}\Delta\sigma^2_n(x) &= \int_{x\in\mathcal{X}} \sigma^2_n(x) - \sigma^2_{n+1}(x)\, dx \\ &= c - \int_{x\in\mathcal{X}} \sigma^2_{n+1}(x)\, dx.\end{aligned}$$

So maximizing reduction in variance is the same as minimizing reduced variance,

- averaged over the input space.

Therefore the optimization that needs to be solved in each iteration is

$$x_{n+1} = \text{argmin}_{x\in\mathcal{X}} \int_{x\in\mathcal{X}} \sigma^2_{n+1}(x)\, dx.$$

- In practice the integral is often approximated by a sum over a reference set.

Implementation

The `laGP` package has functions called `alcGP` and `alcGPsep` that automate this objective, which we code in R as follows.

- It estimates $\Delta\sigma^2_n$, which we want to maximize.

```
obj.alc <- function(x, Xref, gpi) - sqrt(alcGP(gpi, matrix(x, nrow=1), Xref))
alc.search <- function(X, Xref, gpi)
 {
    start <- maximin(nrow(X), 2, T=100*nrow(X), Xorig=X)[1:nrow(X),]
    xnew <- matrix(NA, nrow=nrow(start), ncol=ncol(X)+1)
    for(i in 1:nrow(start)) {
        out <- optim(start[i,], obj.alc, method="L-BFGS-B",
            lower=0, upper=1, gpi=gpi, Xref=Xref)
        xnew[i,] <- c(out$par, -out$value)
    }
    solns <- data.frame(cbind(start, xnew))
    names(solns) <- c("s1", "s2", "x1", "x2", "val")
    return(solns)
}
```

Here we go

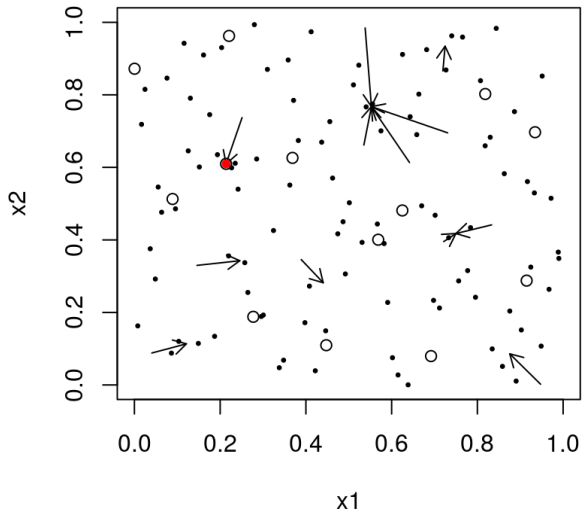
Re-initialize, and find x_{n+1} via ALC on a LHS reference set.

```
X <- X[1:ninit,]; y <- y[1:ninit]
gpi <- newGP(X, y, d=0.1, g=0.1*var(y), dK=TRUE)
mle <- jmleGP(gpi, c(d$min, d$max), c(g$min, g$max), d$ab, g$ab)
rmse.alc <- sqrt(mean((ytrue - predGP(gpi, XX, lite=TRUE)$mean)^2))
Xref <- randomLHS(100, 2)
solns <- alc.search(X, Xref, gpi)
m <- which.max(solns$val)
xnew <- as.matrix(solns[m,3:4])
prog.alc <- solns$val[m]
```

And update.

```
X <- rbind(X, xnew); y <- c(y, f(xnew))
updateGP(gpi, xnew, y[length(y)])
mle <- rbind(mle, jmleGP(gpi, c(d$min, d$max), c(g$min, g$max), d$ab, g$ab))
rmse.alc <- c(rmse.alc, sqrt(mean((ytrue - predGP(gpi, XX, lite=TRUE)$mean)^2)))
```

```
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
suppressWarnings(arrows(solns$s1, solns$s2, solns$x1, solns$x2, length=0.1))
points(solns$x1[m], solns$x2[m], col=2, pch=20)
points(Xref, cex=0.5, pch=20)
```



- The criteria prefers candidates far from X_n and close to (many) `Xref` .

More runs …

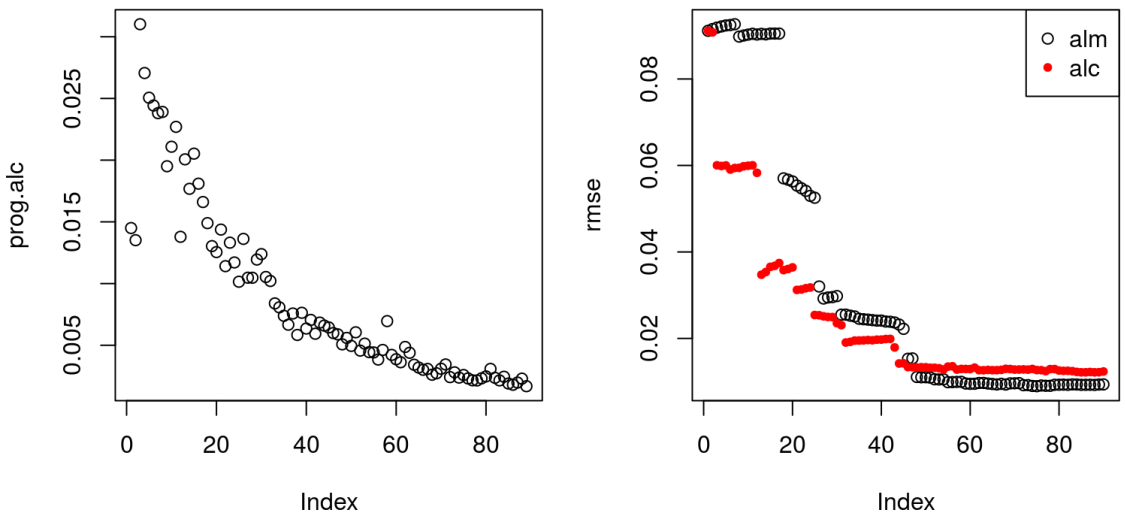
```
for(i in nrow(X):100) {
    Xref <- randomLHS(100, 2)
    solns <- alc.search(X, Xref, gpi)
    m <- which.max(solns$val); prog.alc <- c(prog.alc, solns$val[m])
    xnew <- as.matrix(solns[m,3:4])
    X <- rbind(X, xnew); y <- c(y, f(xnew))
    updateGP(gpi, xnew, y[length(y)])
    mle <- rbind(mle, jmleGP(gpi, c(d$min, d$max), c(g$min, g$max), d$ab, g$ab))
    p <- predGP(gpi, XX, lite=TRUE)
    rmse.alc <- c(rmse.alc, sqrt(mean((ytrue-p$mean)^2)))
}
mle[seq(1,nrow(mle), by=20),]
```

```
##             d            g tot.its dits gits
## 1  0.01122657 0.0005610943      25   20    5
## 21 0.03221052 0.0005564847       7    4    3
## 41 0.03656873 0.0005200306      11    6    5
## 61 0.02983366 0.0007904237      15    7    8
## 81 0.02942484 0.0013014827      42   21   21
```

Progress

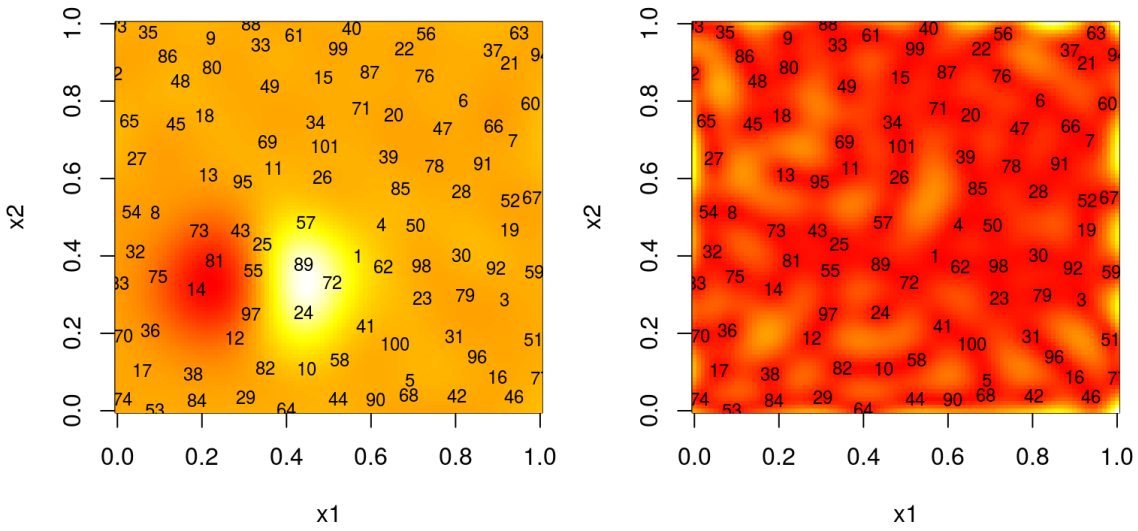
Similar behavior actually, possibly a bit better to start out with.

```
par(mfrow=c(1,2)); plot(prog.alc);
plot(rmse); points(rmse.alc, col=2, pch=20)
legend("topright", c("alm", "alc"), pch=c(21,20), col=1:2)
```



Fewer points on the boundary (owing to the integral).

```
par(mfrow=c(1,2))
image(x1, x2, matrix(p$mean, ncol=length(x1)), col=heat.colors(128))
text(X, labels=1:nrow(X), cex=0.75)
image(x1, x2, matrix(sqrt(p$s2), ncol=length(x1)), col=heat.colors(128))
text(X, labels=1:nrow(X), cex=0.75)
```



Fast GP updates

Sequential design/inference

Being able to quickly update a model fit, as data comes in, is often overlooked as an important aspect in model choice.

However in the context of sequential design, and in computer experiments

- where automation, and loops over inputs to simulations naturally create a sequential inference (and design) environment,

fast updating can crucial to the relevance of nonparametric surrogate models with otherwise large computational demands—especially when the data set gets large.

- Fortunately, GPs offer such a feature, conditional on the hyperparameterization,
- which, as we have seen, can safely be updated in a more ad hoc fashion.
- This is what `updateGP` does in `laGP` (though following with `jmleGP` somewhat defeats the purpose).

Partition inverse equations

The key to fast GP updates, as new data arrives, is fast decomposition of the covariance matrix as $K_n \rightarrow K_{n+1}$.

In the case of the inverse, the **partition inverse equations** are helpful.

- These are often expressed generically for blocks of arbitrary size, but we only need them for one new (symmetric) row/column.

$$\text{If} \qquad K_{n+1}(x_{n+1}) = \begin{bmatrix} K_n & k_n(x_{n+1}) \\ k_n^\top(x_{n+1}) & K(x_{n+1}, x_{n+1}) \end{bmatrix},$$
$$\text{then} \qquad K_{n+1}^{-1} = \begin{bmatrix} [K_n^{-1} + g_n(x_{n+1}) g_n^\top(x_{n+1}) v_n(x_{n+1})] & g_n(x_{n+1}) \\ g_n^\top(x_{n+1}) & v_n^{-1}(x_{n+1}) \end{bmatrix},$$

where $g_n(x_{n+1}) = -v_n^{-1}(x_{n+1}) K_n^{-1} k_n(x_{n+1})$ and $v_n(x_{n+1}) = K(x_{n+1}, x_{n+1}) - k_n^\top(x_{n+1}) K_n^{-1} k_n(x_{n+1})$.

- The computational cost for the update is $\mathcal{O}(n^2)$.

Updating variance

Observe that $v_n(x)$ in the formulas above is the same as our scale-free predictive variance.

- That is, $\sigma_n^2(x) = \hat{\tau}^2 v_n(x)$.

As a consequence, updating the predictive variance at locations x is also super fast:

$$\begin{aligned} v_n(x) - v_{n+1}(x) = {} & k_n^\top(x) G_n(x_{n+1}) v_n(x_{n+1}) k_n(x) \\ & + 2 k_n^\top(x) g_n(x_{n+1}) K(x_{n+1}, x) + K(x_{n+1}, x)^2 / v_n(x_{n+1}), \end{aligned}$$

where $G_n(x') \equiv g_n(x') g_n^\top(x')$, and $g_n(x') = -K_n^{-1} k_n(x') / v_n(x')$.

- The computational cost is $\mathcal{O}(n^2)$.
- This can be helpful for ALC calculation on a fixed candidate set.
- And local GP approximation (later).
- It is easy to show that there is also a fast update for $\hat{\tau}_n^2 \to \hat{\tau}_{n+1}^2$.

Determinant updates

Determinants, which are an important part of the likelihood evaluation, can also be quickly updated.

- There is nothing special about this update, as determinants are naturally specified recursively.
- However, it can be instructive to provide it in the notation of our partitioned inverse above.

$$\begin{aligned} \log |K_{n+1}| &= \log |K_n| + \log(K(x_{n+1}, x_{n+1}) + g_n^\top(x_{n+1}) k_n(x_{n+1}) v_n(x_{n+1})) \\ &= \log |K_n| + \log(v_n(x_{n+1})). \end{aligned}$$

- The computational cost is $\mathcal{O}(n)$.
- It is interesting to see that the determinant changes by the variance of the point added into the old fit.
    - This explains why ALM approximates maximum entropy designs.

Long-run cost

If you perform $\mathcal{O}(j^2)$ updates for $j = 1, \ldots, n$, say,

- the overall cost is $\mathcal{O}(n^3)$,
- just the same as if you did it all in one go.

So there is no savings, in the grand scheme of things, by proceeding sequentially.

- In fact, sequential is bound to be slower (with a bigger constant in the order notation) than one shot.

But if you have decisions to make along the way, based on intermediate designs (such as with ALM or ALC, say),

- then it is way better for those to be $\mathcal{O}(j^2)$ than $\mathcal{O}(j^3)$,
- since the latter results in $\mathcal{O}(n^4)$ as $j = 1, \ldots, n$.