Gaussian Process Regression

RSMs and Computer Experiments: Part 2

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

Goals

To understand the Gaussian Process

- as a prior over random functions,
- a posterior over functions given observed data,
- as a tool for spatial data modeling and computer experiments,
- and simply as a flexible "nonparametric" regression tool.

We'll see that, almost in spite of a technical (over) analysis of its properties, and sometimes strange vocabulary used to describe its features,

- it is a simple extension to the linear (regression) model.
- Knowing that is all it takes to make use of it as a nearly unbeatable regression tool when input–output relationships are relatively smooth.
- And even sometimes when they are not.

Gaussian process prior

Multivariate normal modeling

Gaussian process (GP) is a very generic term.

All it means is that any finite collection of realizations (or observations) have a multivariate normal (MVN) distribution.

That, in turn, means that the characteristics of those realizations are completely described by their

- mean vector, μ, or **mean function** $\mu(x)$
- and covariance matrix Σ, or **covariance function** $\Sigma(x, x')$.

You'll hear people talk about function spaces, and reproducing kernel Hilbert spaces, and so on,

- and sometimes that is important, depending on the context.

Covariance structure

But mostly that makes things seem fancier than they really are.

- It is all in the covariance.

Consider a covariance function defined by Euclidean distance:

$$\mathbb{C}\mathrm{ov}(Y(x), Y(x')) = \Sigma(x, x') = \exp\{-||x - x'||^2\}$$

- i.e., covariance between $Y(x)$ and $Y(x')$ decays exponentially fast as x and x' become farther apart in the input, or x-space.
- In this specification, observe that $\Sigma(x, x) = 1$ and $\Sigma(x, x') < 1$ for $x' \neq x$.

Finally, note that if we define a covariance matrix Σ_n, based evaluating $\Sigma(x_i, x_j)$ on pairs of n x-values $x_1, \dots, x_n$, it must be **positive definite**

$$x^\top \Sigma_n x > 0 \quad \text{for all } x > 0,$$

- to be a valid covariance matrix for a MVN.

Data generating mechanism

To see how GPs can be used to perform regression, lets first see how they can be used to *generate* random data following a smooth functional relationship.

Suppose we

- take a bunch of x-values: $x_1, \dots, x_n$;
- define Σ_n via $\Sigma_n^{ij} = \Sigma(x_i, x_j)$, for $i, j = 1, \dots, n$.
- draw an n-variate realization $Y \sim \mathcal{N}_n(0, \Sigma_n)$,
- and plot the result in the x-y plane.

Note that

- The mean of this MVN is zero; it need not be but it is quite surprising how well things work even in this special case.
- We'll talk about generalizing this later.

Generating data in R

Here is a version of that with x-values on a 1d grid.

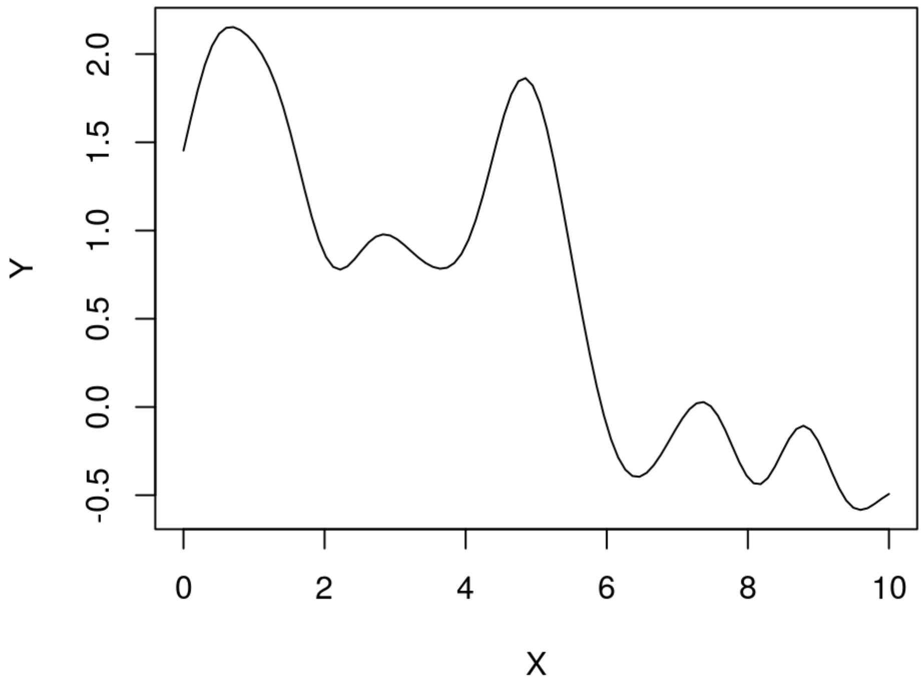
```
n <- 100
X <- matrix(seq(0, 10, length=n), ncol=1)
library(plgp, quietly=TRUE)
D <- distance(X)
eps <- sqrt(.Machine$double.eps) ## defining a small number
Sigma <- exp(-D + diag(eps, n))  ## for numerical stability
library(mvtnorm, quietly=TRUE)
Y <- rmvnorm(1, sigma=Sigma)
```

That's it!

- We've generated a finite realization of a random function under a GP prior
- with a particular covariance structure.
- Now all that's left is to visualize it on the x-y plane.

Visualizing

```
plot(X, Y, type="l")
```



- Nice looking random function!

Properties?

What are the properties of this function?

Several are super easy to deduce from the form of the covariance structure.

- We'll get a range of about $[-2, 2]$ with 95% probability, because the scale of the covariance is 1.
- We'll get lots of bumps in the x-range of $[0, 10]$ because short distances are highly correlated and long distances are essentially uncorrelated:

```
c(exp(-1^2), exp(-4^2))
```

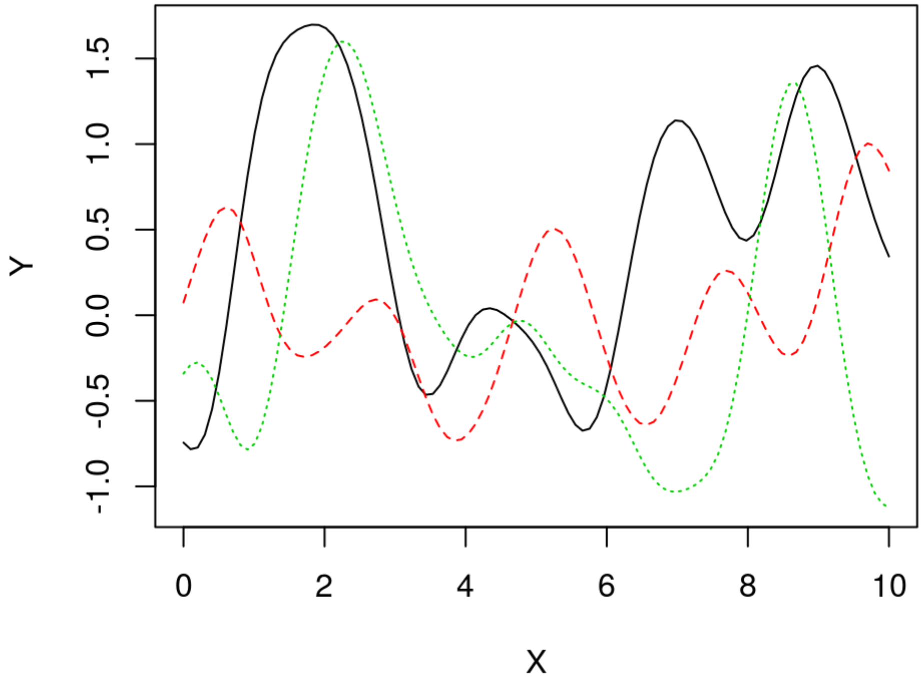
```
## [1] 3.678794e-01 1.125352e-07
```

- The surface is going to be extremely smooth because the covariance function is infinitely differentiable,
  - a discussion for another time.

Multiple draws

But we can't anticipate much else about the nature of a particular realization.

```
Y <- rmvnorm(3, sigma=Sigma)
matplot(X, t(Y), type="l", ylab="Y")
```



Posterior

Of course, we're not in the business of generating random functions.

- I'm not sure what that would be useful for.
- Who cares what $Y(x)$'s you can produce from GP prior: $Y(x) \sim \mathcal{GP}$?

Instead, we want to ask

- Given examples of a function via pairs $(x_1, y_1), \dots, (x_n, y_n)$, comprising of data $D_n = (X_n, Y_n)$,
- what random function realizations could explain those values?

I.e., we want to know about the conditional distribution of $Y(x) \mid D_n$.

- If we call $Y(x) \sim \mathcal{GP}$ the prior, then $Y(x) \mid D_n$ must be the posterior.

Nonparametric regression

But we don't need to get all Bayesian about it,

- as much as I love all things Bayesian.

That conditional distribution, the predictive distribution, is the cornerstone of **regression** analysis.

Forget, for the moment, that

- when regressing one is often interested in other aspects (relevance of predictors, etc.), via estimates of parameters,
- and that so far our random functions look like they have no noise!

The curious, and most noteworthy, thing is that so far **there are no parameters** in the current setup!

Lets shelve interpretation (Bayesian updating or a twist on simple regression) for a moment and just focus on conditional distributions.

MVN partition

Deriving that predictive distribution is a simple application of deducing conditional distributions from a (joint) MVN.

From Wikipedia, if an N dimensional random vector x is partitioned as

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \text{with sizes} \quad \begin{pmatrix} q \times 1 \\ (N-q) \times 1 \end{pmatrix},$$

and accordingly μ and Σ are partitioned as,

$$\mu = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} \quad \text{with sizes} \quad \begin{pmatrix} q \times 1 \\ (N-q) \times 1 \end{pmatrix}$$

and

$$\Sigma = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix} \text{ with sizes } \begin{pmatrix} q \times q & q \times (N-q) \\ (N-q) \times q & (N-q) \times (N-q) \end{pmatrix},$$

MVN conditional

… then, the distribution of x_1 conditional on x_2 is MVN $x_1 \mid x_2 \sim \mathcal{N}_q(\bar{\mu}, \bar{\Sigma})$, where

$$\begin{aligned} \bar{\mu} &= \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2) \\ \text{and} \qquad \bar{\Sigma} &= \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. \end{aligned}$$

An interesting feature of this result is that conditioning upon x_2 alters (decreases) the variance of x_1 compared to its marginal variance Σ_{11},

- but the amount by which it is decreased *does not* depend on the value of x_2.
- Whereas the mean is also altered, by an amount that *does* depend upon x_2.
- Also note that $\Sigma_{12} = \Sigma_{21}^\top$.

Joint GP modeling

How do we deploy that to derive $Y(x) \mid D_n$?

Consider an $n+1^{\text{st}}$ observation $Y(x)$. Now, let $Y(x)$ and Y_n have a joint MVN distribution with **mean zero** and covariance function $\Sigma(x, x')$.

That is, stack

$$\begin{pmatrix} Y(x) \\ Y_n \end{pmatrix} \quad \text{with sizes} \quad \begin{pmatrix} 1 \times 1 \\ n \times 1 \end{pmatrix},$$

and if $\Sigma(X_n, x)$ is the $n \times 1$ matrix comprised of $\Sigma(x_1, x), \dots, \Sigma(x_n, x)$, its covariance structure can be partitioned as follows.

$$\begin{pmatrix} \Sigma(x, x) & \Sigma(x, X_n) \\ \Sigma(X_n, x) & \Sigma_n \end{pmatrix} \text{ with sizes } \begin{pmatrix} 1 \times 1 & 1 \times n \\ n \times 1 & n \times n \end{pmatrix}$$

- Recall that $\Sigma(x, x) = 1$ with our choice of covariance structure.
- Note that $\Sigma(x, X_n) = \Sigma(X_n, x)^\top$.

GP prediction

Our conditional results for the MVN give us the following predictive distribution

$$Y(x) \mid D_n \sim \mathcal{N}(\mu(x), \sigma^2(x))$$

with

$$\begin{aligned} \text{mean} \quad \mu(x) &= \Sigma(x, X_n)\Sigma_n^{-1} Y_n \\ \text{and variance} \quad \sigma^2(x) &= 1 - \Sigma(x, X_n)\Sigma_n^{-1} \Sigma(x, X_n)^\top. \end{aligned}$$

$\mu(x)$ is linear in the observations Y_n.

- We have a linear predictor! In fact it is the best linear unbiased predictor (BLUP).

$\sigma^2(x)$ is lower than the marginal variance $\Sigma(x, x) = 1$.

- So we learn something from the data Y_n; in fact the amount it goes down is a function of the distance between x and X_n.
- But it doesn't depend on the Y_n values.

Multiple prediction

Those were "pointwise" predictive calculations.

- We can apply them, separately, for many x locations,
- but that would ignore the obvious correlation they would experience in a big MVN analysis.

However, that's easily fixed by considering a bunch of x locations, in a predictive design $\mathcal{X}$ of n' rows, say, all at once:

$$Y(\mathcal{X}) \mid D_n \sim \mathcal{N}_{n'}(\mu(\mathcal{X}), \Sigma(\mathcal{X}))$$

with

$$\begin{aligned} \text{mean} \quad \mu(\mathcal{X}) &= \Sigma(\mathcal{X}, X_n)\Sigma_n^{-1} Y_n \\ \text{and variance} \quad \Sigma(\mathcal{X}) &= \Sigma(\mathcal{X}, \mathcal{X}) - \Sigma(\mathcal{X}, X_n)\Sigma_n^{-1} \Sigma(\mathcal{X}, X_n)^\top. \end{aligned}$$

- where $\Sigma(\mathcal{X}, X_n)$ is an $n' \times n$ matrix.

For example …

Consider some super simple data in 1d.

Here are the relevant data quantities.

```r
n <- 8
X <- matrix(seq(0,2*pi,length=n), ncol=1)
y <- sin(X)
D <- distance(X)
Sigma <- exp(-D)
```

Here are the relevant predictive quantities.

```r
XX <- matrix(seq(-0.5, 2*pi+0.5, length=100), ncol=1)
DXX <- distance(XX)
SXX <- exp(-DXX) + diag(eps, ncol(DXX))
DX <- distance(XX, X)
SX <- exp(-DX)
```

Predictive equations in R

Now we just follow the formulas.

```
Si <- solve(Sigma)
mup <- SX %*% Si %*% y
Sigmap <- SXX - SX %*% Si %*% t(SX)
```

Joint draws are obtained from the predictive distribution as follows.

```
YY <- rmvnorm(100, mup, Sigmap)
```

We can plot those $Y(\mathcal{X}) =$ YY samples as a function of the input $\mathcal{X} =$ XX locations.

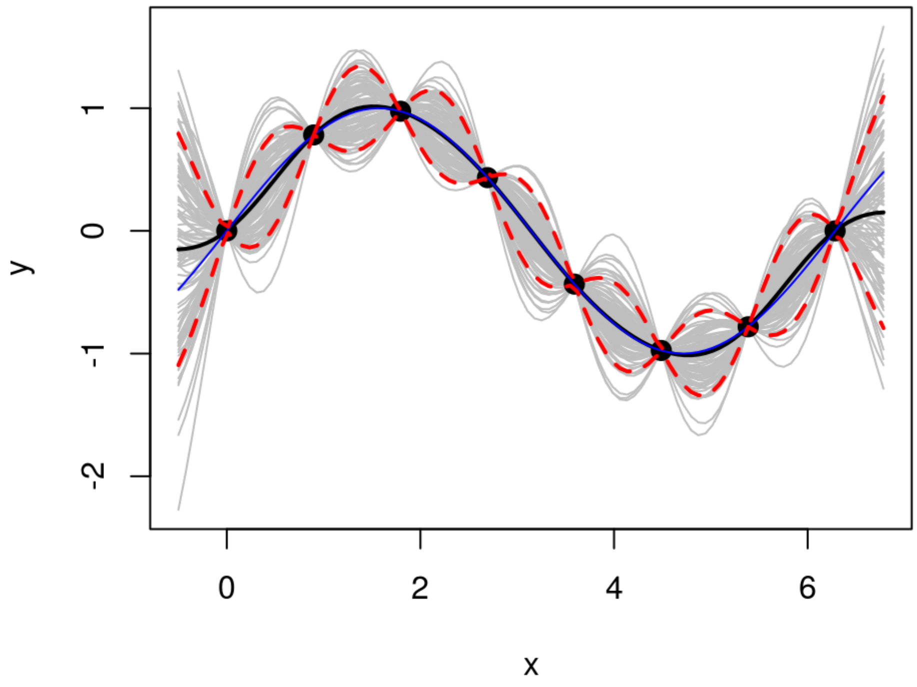
- Maybe along with some pointwise quantile-based error bars.

```
q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap)))
```

- You ready?

Visualizing

```
matplot(XX, t(YY), type="l", col="gray", lty=1, xlab="x", ylab="y")
points(X, y, pch=20, cex=2)
lines(XX, mup, lwd=2); lines(XX, sin(XX), col="blue")
lines(XX, q1, lwd=2, lty=2, col=2); lines(XX, q2, lwd=2, lty=2, col=2)
```



Commentary

What do we observe?

- Notice how the predictive surface interpolates the data.
- That's because $\Sigma(x, x) = 1$ and $\Sigma(x, x') \rightarrow 1^-$ as $x' \rightarrow x$.
- The error-bars have a "football" shape, being widest at locations that are farthest from the x_i values in the data.
- The error-bars get really big outside the range of the data,
- but look at how the predictive mean is mean-reverting (to zero).
- The predictive variance, we know, will level off to 1.
- That means we can't trust the extrapolations too far outside the data range, but at least they're behavior is not unpredictable.

These features, but especially the football shape, is what makes GPs popular for computer experiments.

- Oh, and they're really accurate out-of sample!

Higher dimension?

Nothing special here, except visualization is lots simpler in 1d or 2d.

Consider a random function in 2d sampled from the GP prior.

- First lets create a grid

```
nx <- 20
x <- seq(0,2,length=nx)
X <- expand.grid(x, x)
```

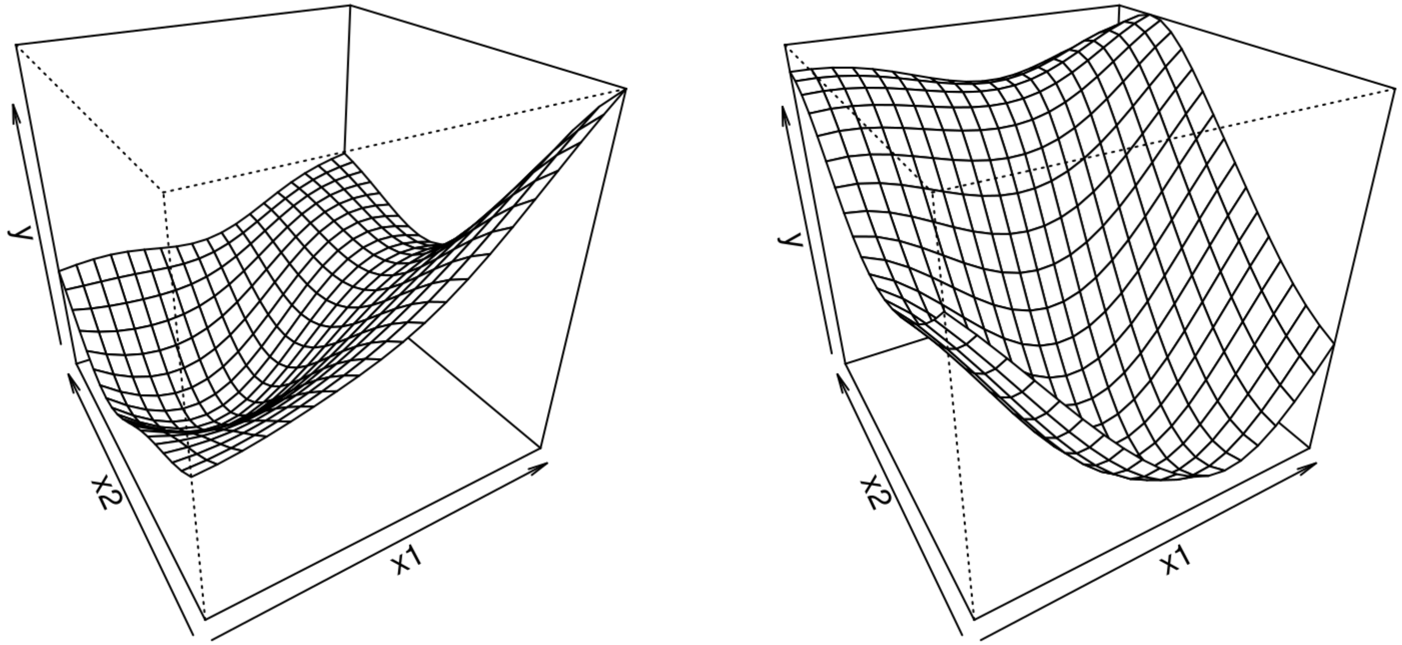
- Then calculate distances and evaluate the covariance on that grid.

```
D <- distance(X)
Sigma <- exp(-D) + diag(eps, nrow(X))
```

- Now we can make random draws and stretch them over a mesh.

Two realizations:

```
Y <- rmvnorm(2, sigma=Sigma)
par(mfrow=c(1,2), mar=c(1,0.5,0.5,0.5))
persp(x,x, matrix(Y[1,], ncol=nx), theta=-30,phi=30,xlab="x1",ylab="x2",zlab="y")
persp(x,x, matrix(Y[2,], ncol=nx), theta=-30,phi=30,xlab="x1",ylab="x2",zlab="y")
```



Prediction in 2d?

Consider some 2d synthetic data and a 2d predictive grid.

```
library(lhs)
X <- randomLHS(40, 2)
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)
xx <- seq(-2,4, length=40); XX <- expand.grid(xx, xx)
```

Here are the relevant data quantities, essentially cut-and-paste from above.

```
D <- distance(X)
Sigma <- exp(-D)
```

Here are the relevant predictive quantities.

```
DXX <- distance(XX)
SXX <- exp(-DXX) + diag(eps, ncol(DXX))
DX <- distance(XX, X)
SX <- exp(-DX)
```

MVN Conditionals

Now we just follow the formulas: these are identical.

```
Si <- solve(Sigma)
mup <- SX %*% Si %*% y
Sigmap <- SXX - SX %*% Si %*% t(SX)
```

It is hard to visualize a multitude of sample paths in 2d,

- but we can obtain them with the same `rmvnorm` command if we'd like.

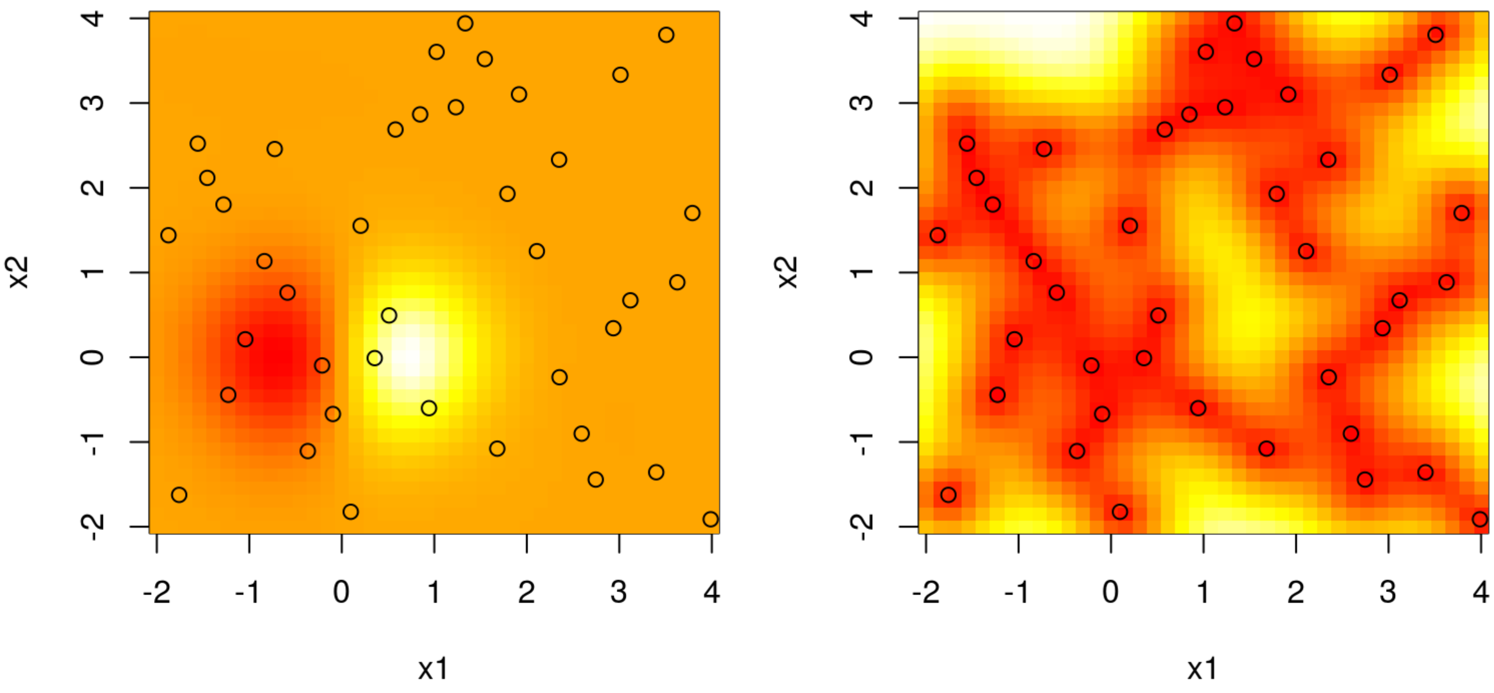
Instead, we'll focus on plotting pointwise summaries, namely

- the predictive mean, `mup` above, and
- the predictive standard deviation:

```
sdp <- sqrt(diag(Sigmap))
```

Beautiful:

```
par(mfrow=c(1,2)); cols <- heat.colors(128)
image(xx,xx, matrix(mup, ncol=length(xx)), xlab="x1",ylab="x2", col=cols)
points(X[,1], X[,2])
image(xx,xx, matrix(sdp, ncol=length(xx)), xlab="x1",ylab="x2", col=cols)
points(X[,1], X[,2])
```

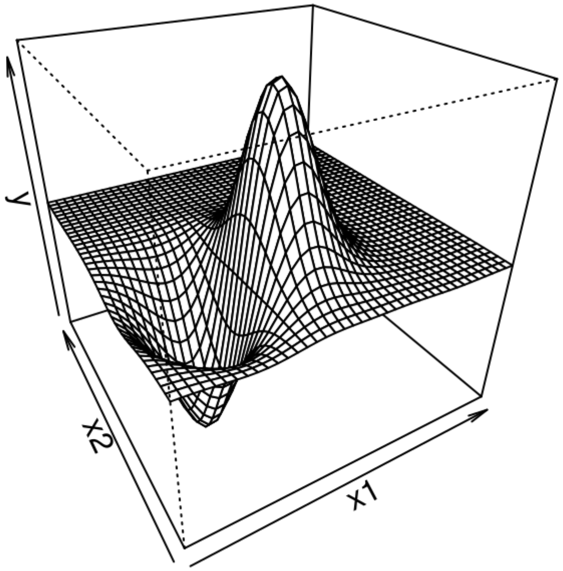


What do we observe? Pretty much the same thing as in the 1d case.

- We can't see it, but the predictive surface interpolates.
- The predictive standard deviation is highest away from the x_i values in the data.

Here is another look at what we predicted.

```
par(mar=c(1,0.5,0,0.5))
persp(xx,xx, matrix(mup, ncol=40), theta=-30,phi=30,xlab="x1",ylab="x2",zlab="y")
```



Where do we go from here?

Hopefully you're starting to be convinced that GPs represent a powerful nonparametric regression tool.

Its kinda-cool that they do so well without really having to *learn* anything.

- It is just a simple application of MVN conditionals
- paired with a distance-based notion of covariance.

But when you think about it a little bit, there are lots of (hidden) assumptions which are going to be violated by most real-data contexts.

- Data can be noisy.
- The amplitude of the function is not 2 (i.e., we don't know it in advance).
- Correlation doesn't decay uniformly in all directions (i.e., radially).
- Even the most ideally smooth physical relationships are rarely *infinitely smooth*.

GP hyperparameters

Scale

Lets suppose you wanted your prior to generate random functions which had an amplitude larger than one.

- You could introduce a scale parameter, lets call it τ^2,
- and then take $\Sigma_n = \tau^2 C_n$.

Here, C is basically the same as our Σ before: a correlation function for which $C(x,x) = 1$ and $C(x,x') < 1$ for $x \neq x'$, and positive definite. E.g.,

$$C(x,x') = \exp\{||x - x'||^2\}$$

- But now we need a more nuanced notion of covariance, to accommodate scale, so we're re-parameterizing a bit.

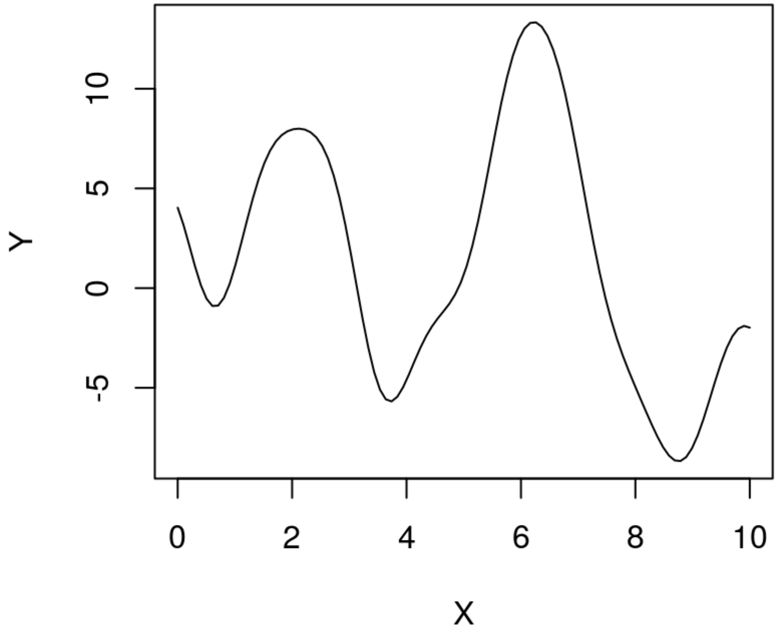
Now our MVN generator looks like

$$Y \sim \mathcal{N}_n(0, \tau^2 C_n).$$

Checking

That ought to do the trick. E.g., for an amplitude of 10, choose $\tau^2 = 5^2 = 25$.

```
n <- 100; X <- matrix(seq(0, 10, length=n), ncol=1)
D <- distance(X); C <- exp(-D + diag(eps, n))
tau2 <- 25; Y <- rmvnorm(1, sigma=tau2 * C)
plot(X, Y, type="l")
```



Inference

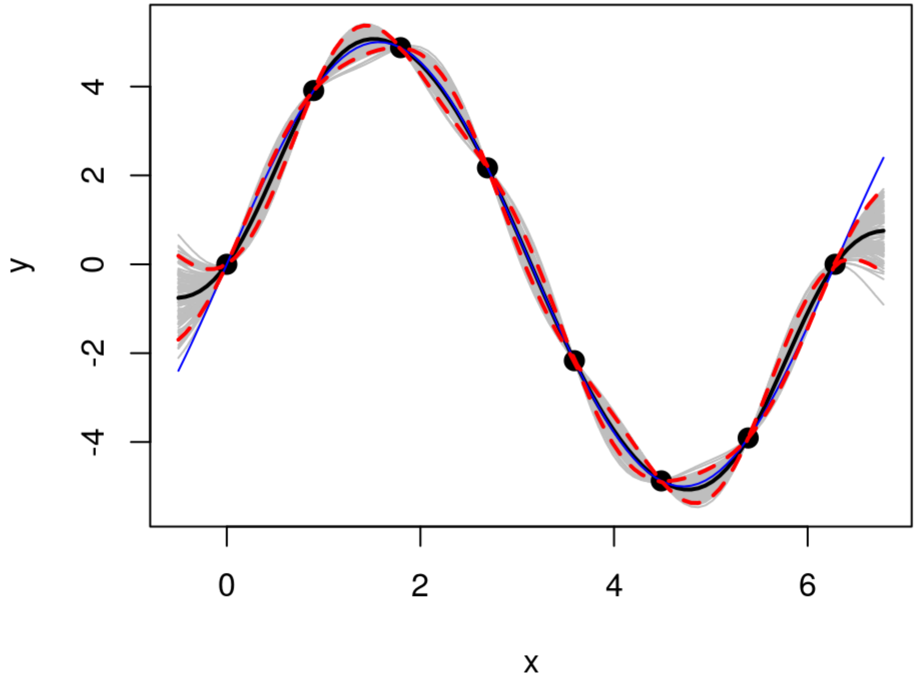
Again, who cares about generating data?

- We want to be able to learn about a function on any scale.

What would happen if we had some data with an amplitude of 5, but we used a GP with a built-in scale of 1 [amplitude of 2].

```r
n <- 8
X <- matrix(seq(0,2*pi,length=n), ncol=1)
y <- 5*sin(X)  ## this is the only difference
D <- distance(X)
Sigma <- exp(-D)
XX <- matrix(seq(-0.5, 2*pi+0.5, length=100), ncol=1)
DXX <- distance(XX)
SXX <- exp(-DXX) + diag(eps, ncol(DXX))
DX <- distance(XX, X)
SX <- exp(-DX)
Si <- solve(Sigma);
mup <- SX %*% Si %*% y
Sigmap <- SXX - SX %*% Si %*% t(SX)
```

```
YY <- rmvnorm(100, mup, Sigmap)
q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap)))
matplot(XX, t(YY), type="l", col="gray", lty=1, xlab="x", ylab="y")
points(X, y, pch=20, cex=2)
lines(XX, mup, lwd=2); lines(XX, 5*sin(XX), col="blue")
lines(XX, q1, lwd=2, lty=2, col=2); lines(XX, q2, lwd=2, lty=2, col=2)
```



What happened?

Actually, the "scale 1" GP was pretty robust.

- It gets the predictive mean almost perfectly, despite using the "wrong prior" relative to the actual data generating mechanism.

But it is over confident.

- Besides a change of scale, the data exhibit no change in relative error,
- nor any other changes for that matter, compared to the example we did above where the scale was actually 1.

So we are under-estimating the predictive uncertainty,

- which is obvious by visually comparing the error-bars.

And if we look closely, we can see that the true function goes well outside our predictive interval at the edges of the input space.

- That didn't happen before.

Estimating the scale

How do we estimate the scale?

- … which in this context is called a **hyperparameter**, because its impact on the overall estimation procedure is really more of a "fine tuning";
- The real flexibility in the predictive surface often materializes, as we have seen, in spite of such parameter settings.

As with any "parameter", we have many choices when it comes to estimation.

- method of moments
- likelihood (maximum likelihood, Bayesian inference)
- cross validation
- the "eyeball" method

I have a strong preference for likelihood-based methods (MLE/Bayes) because they are relatively hands-off, and nicely generalize to higher dimensional parameter spaces.

Likelihood

What is the likelihood?

- It's kinda bizarre that we've been talking about priors and posteriors without ever talking about likelihood.
- We'll get into finer detail (and possibly debate!) later.

For now, recognize that our data-generating process is $Y \sim \mathcal{N}_n(0, \tau^2 C_n)$,

- so it must be that our likelihood comes from the MVN pdf:

$$L \equiv L(\tau^2, C) = (2\pi\tau^2)^{-\frac{n}{2}} |C_n|^{-\frac{1}{2}} \exp\left\{ -\frac{1}{2\tau^2} Y_n^\top C_n^{-1} Y_n \right\}.$$

- And the log of that is

$$\ell = \log L = -\frac{n}{2} \log 2\pi - \frac{n}{2} \log \tau^2 - \frac{1}{2} \log |C_n| - \frac{1}{2\tau^2} Y_n^\top C_n^{-1} Y_n.$$

MLE for τ^2

To maximize that likelihood with respect to τ^2, just differentiate and solve.

$$\begin{aligned} 0 \overset{\text{set}}{=} \ell' &= -\frac{n}{2\tau^2} + \frac{1}{2(\tau^2)^2} Y_n^\top C_n^{-1} Y_n. \\ \text{so } \hat{\tau}^2 &= \frac{Y_n^\top C_n^{-1} Y_n}{n} \end{aligned}$$

- i.e., the mean residual sum of squares (under the quadratic form).

How would this analysis change if we were to take a Bayesian approach?

- Say under …

$$\tau^2 \sim \mathrm{IG}(a/2, b/2)$$

- What happens when $a = b = 0$, which is equivalent to $p(\tau^2) \propto 1/\tau^2$?
    - A so-called reference prior in this context.

Predictive equations

Now when we plug $\hat{\tau}^2$ into the predictive equations, we're technically turning a (multivariate) normal into a (multivariate) Student-t with n degrees of freedom.

- but lets presume, for now, that n is large enough so that doesn't matter.
- We'll see that, as we generalize to more hyperparameters, it could indeed matter.

We have

$$\begin{aligned} Y(\mathcal{X}) \mid D_n &\sim \mathcal{N}_{n'}(\mu(\mathcal{X}), \Sigma(\mathcal{X})) \\ \text{with mean} \quad \mu(\mathcal{X}) &= C(\mathcal{X}, X_n) C_n^{-1} Y_n \\ \text{and variance} \quad \Sigma(\mathcal{X}) &= \hat{\tau}^2 [C(\mathcal{X}, \mathcal{X}) - C(\mathcal{X}, X_n) C_n^{-1} C(\mathcal{X}, X_n)^\top]. \end{aligned}$$

- Notice how $\hat{\tau}^2$ does not factor into the predictive mean
- but it does figure into the predictive variance
  - and that means the Y_n-values are finally involved!

Back to our example

First estimate τ^2.

```
CX <- SX; Ci <- Si; CXX <- SXX
tau2hat <- drop(t(y) %*% Ci %*% y / length(y))
2*sqrt(tau2hat)
```

```
## [1] 5.486648
```

- Close to what we know is the truth.

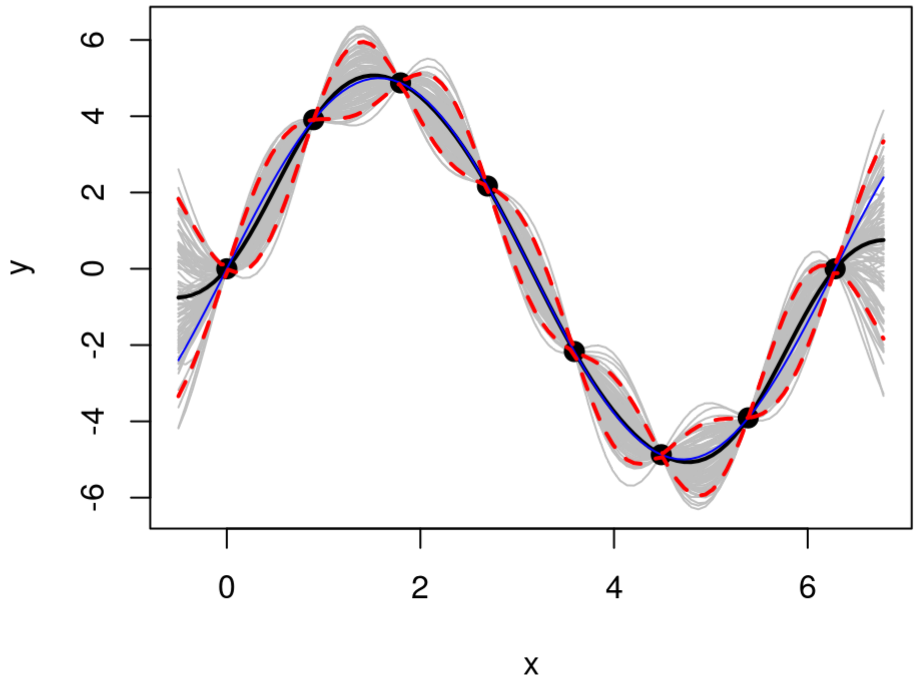
Now estimate the predictive mean vector and covariance matrix …

```
mup2 <- CX %*% Ci %*% y
Sigmap2 <- tau2hat * (CXX - CX %*% Ci %*% t(CX))
YY <- rmvnorm(100, mup2, Sigmap2)
q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap2)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap2)))
```

… and visualize …

Much better.

```
matplot(XX, t(YY), type="l", col="gray", lty=1, xlab="x", ylab="y")
points(X, y, pch=20, cex=2)
lines(XX, mup, lwd=2); lines(XX, 5*sin(XX), col="blue")
lines(XX, q1, lwd=2, lty=2, col=2); lines(XX, q2, lwd=2, lty=2, col=2)
```



Noise and nuggets

To "break" interpolation, we need to "break" the continuity of the correlation structure on the diagonal.

- Right now we have $C(x, x') \to 1^-$ as $x \to x'$
- which says that the closer x is to x' the higher the correlation, until the correlation is perfect,
  - which is what "connects the dots".

There are a lot of ways to skin this cat (and a lot of storytelling that goes with it), but the simplest way to "break it" is to say

$$K(x, x') = C(x, x') + g\delta_{x,x'}$$

- where $g > 0$ is a hyperparameter, sometimes called the **nugget**, which determines the size of the discontinuity as $x' \to x$.
- δ is the Kroneker delta function, although the way it is written above makes it look like the Dirac delta.

Covariance

What does that mean?

We only add in g when *indices* of x are the same, not simply for identical values.

- $K(x_i, x_j) = C(x_i, x_j)$ when $i \neq j$, even if $x_i = x_j$;
- only $K(x_i, x_i) = C(x_i, x_i) + g$.

So the previous slide abused notation a little.

This leads to the following representation of the data-generating mechanism.

$$Y \sim \mathcal{N}_n(0, \tau^2 K_n)$$

I.e., the covariance matrix Σ_n is comprised of entries

$$\Sigma_n^{ij} = \tau^2 (C(x_i, x_j) + g\delta_{ij})$$

- or in other words $\Sigma_n = \tau^2 K_n = \tau^2 (C_n + g\mathbb{I}_n)$.

Inference

How, then, do we estimate the hyperparameter g?

- Again, we have all the usual suspects, but I like likelihood-based methods the best.

The MLE $\hat{\tau}^2$, given g (the only other hyperparameter) is

$$\hat{\tau}^2 = \frac{Y_n^\top K_n^{-1} Y_n}{n} = \frac{Y_n (C_n + \mathbb{I}_g)^{-1} Y_n}{n}.$$

So lets plug that back into our log likelihood, to get a *concentrated* log likelihood (or profile likelihood) involving just the remaining parameter g.

$$\begin{aligned} \ell(g) &= -\frac{n}{2}\log 2\pi - \frac{n}{2}\log \hat{\tau}^2 - \frac{1}{2\hat{\tau}^2} Y_n^\top K_n^{-1} Y_n \\ &= c - \frac{n}{2}\log Y_n K_n^{-1} Y_n - \frac{1}{2}\log |K_n| \end{aligned}$$

Numerical methods

Unfortunately, maximizing $\ell(g)$ requires numerical methods.

The simplest thing to do is to throw it into `optimize`.

- So lets code up $-\ell(g)$ in R.

```
counter <- 0
nlg <- function(g, D, Y)
  {
    n <- length(Y)
    K <- exp(-D) + diag(g, n)
    Ki <- solve(K)
    ldetK <- determinant(K, logarithm=TRUE)$modulus
    ll <- - (n/2) * log(t(Y) %*% Ki %*% Y) - (1/2) * ldetK
    counter <<- counter + 1
    return(-ll)
  }
```

- Now that's our objective function.
- A sensible range for g to narrow our search might be between `eps` and `var(y)`.

For example

Defining some noisy data quantities.

```
X <- rbind(X, X); n <- nrow(X); D <- distance(X)
y <- 5*sin(X) + rnorm(n, sd=1)
```

Optimizing to estimate the nugget.

```
print(g <- optimize(nlg, interval=c(eps, var(y)), D=D, Y=y)$minimum)
```

```
## [1] 0.1256754
```

And that gives …

```
K <- exp(-D) + diag(g, n)
Ki <- solve(K)
print(tau2hat <- drop(t(y) %*% Ki %*% y / n))
```

```
## [1] 6.310993
```

Now prediction using the estimated hyperparameters …

```
DX <- distance(XX, X); KX <- exp(-DX)
KXX <- exp(-DXX) + diag(g, nrow(DXX))
```

Derive the predictive mean vector and covariance matrix.

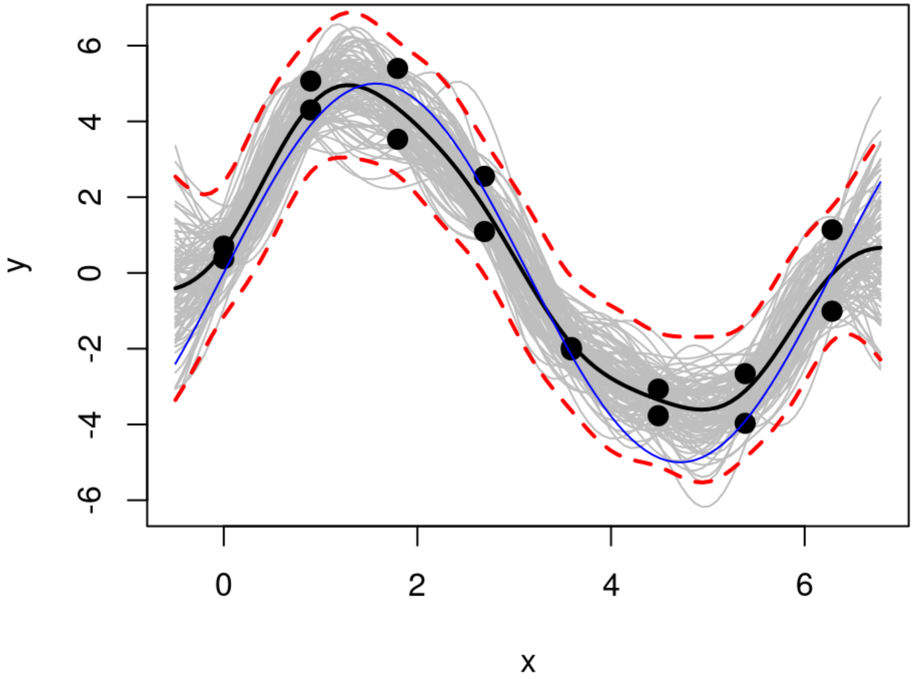
```
mup <- KX %*% Ki %*% y
Sigmap <- tau2hat * (KXX - KX %*% Ki %*% t(KX))
q1 <- mup + qnorm(0.05, 0, sqrt(diag(Sigmap)))
q2 <- mup + qnorm(0.95, 0, sqrt(diag(Sigmap)))
```

If we want to show sample predictive realizations, and want them to look pretty, we should "subtract" out the extrinsic noise,

- i.e., the part due to the nugget g.
- Otherwise our sample paths will be all "jaggety".

```
Sigma.int <- tau2hat * (exp(-DXX) + diag(eps, nrow(DXX)) - KX %*% Ki %*% t(KX))
YY <- rmvnorm(100, mup, Sigma.int)
```

```
matplot(XX, t(YY), type="l", lty=1, col="gray", xlab="x", ylab="y")
points(X, y, pch=20, cex=2)
lines(XX, mup, lwd=2); lines(XX, 5*sin(XX), col="blue")
lines(XX, q1, lwd=2, lty=2, col=2); lines(XX, q2, lwd=2, lty=2, col=2)
```



- The error-bars, containing extrinsic noise, are mostly outside the gray lines.

Derivative-based optimization

It can be unsatisfying to "brute-force" an optimization for a hyperparameter like g,

- even though 1-d solving via `optimize` is often superior to cleverer methods. Can we improve on the number of evaluations?

```
print(nlg.count <- counter)
```

```
## [1] 17
```

How about using derivative information? For that, the following is useful.

$$\frac{\partial K_n^{-1}}{\partial \phi} = -K_n^{-1} \frac{\partial K_n}{\partial \phi} K_n^{-1} \quad \text{and} \quad \frac{\partial \log |K_n|}{\partial \phi} = \text{tr} \left\{ K_n^{-1} \frac{\partial K_n}{\partial \phi} \right\}$$

- These are framed in terms of a generic parameter ϕ involved in building K_n.

Derivative of the log likelihood

By the chain rule,

$$\begin{aligned}\ell'(g) &= -\frac{n}{2}\frac{Y_n\frac{\partial K_n^{-1}}{\partial g}Y_n}{Y_nK_n^{-1}Y_n} - \frac{1}{2}\frac{\partial \log|K_n|}{\partial g} \\ &= \frac{n}{2}\frac{Y_nK_n^{-1}\frac{\partial K_n}{\partial g}K_n^{-1}Y_n}{Y_nK_n^{-1}Y_n} - \frac{1}{2}\mathrm{tr}\left\{K_n^{-1}\frac{\partial K_n}{\partial g}\right\}\end{aligned}$$

- The off diagonal elements of K_n have no g in them,
- and the diagonal is simply $1+g$,
- so $\frac{\partial K_n^{-1}}{\partial g}$ is simply an n-dimensional identity matrix.

Therefore, we have

$$\ell'(g) = \frac{n}{2}\frac{Y_n(K_n^{-1})^2Y_n}{Y_nK_n^{-1}Y_n} - \frac{1}{2}\mathrm{tr}\left\{K_n^{-1}\right\}.$$

Coded in R

Here is an implementation of the gradient of our (negative) log likelihood in R.

```
gnlg <- function(g, D, Y)
 {
    n <- length(Y)
    K <- exp(-D) + diag(g, n)
    Ki <- solve(K)
    KiY <- Ki %*% Y
    dll <- (n/2) * t(KiY) %*% KiY / (t(Y) %*% KiY) - (1/2) * sum(diag(Ki))
    return(-dll)
 }
```

Lets throw that into `optim` and see what happens.

- I like `method="L-BFGS-B"` because it supports derivatives, and bounds.

Solution

```
out <- optim(0.1*var(y), nlg, gnlg, method="L-BFGS-B",
  lower=eps, upper=var(y), D=D, Y=y)
c(g, out$par)
```

```
## [1] 0.1256754 0.1256663
```

- Very similar to the `optimize` output.

How many iterations?

```
out$counts
```

```
## function gradient
##       14       14
```

- 14 iterations to optimize something is pretty excellent!
- But possibly not noteworthy compared to `optimize`'s 17.

What else?

How about the rate of decay of correlation in terms of distance.

- Surely unadulterated Euclidean distance is not equally suited to all data.

Consider the following generalization of the covariance function.

$$C_\theta(x, x') = \exp\left\{ - \frac{||x - x'||^2}{\theta} \right\}.$$

This (hyper-) parameterized family of correlation functions,

- indexed by the scalar hyperparameter θ, called the characteristic **lengthscale**,

is called the **isotropic Gaussian** family.

- Isotropic because correlation decays radially;
- Gaussian because squared distance is used.

Inference for θ?

This is no different than our inference for g,

- except now we have two unknown parameters.

Lets first go the brute-force route for maximizing the likelihood.

```
counter <- 0
nl <- function(par, D, Y)
  {
    theta <- par[1]  ## change 1
    g <- par[2]
    n <- length(Y)
    K <- exp(-D/theta) + diag(g, n)  ## change 2
    Ki <- solve(K)
    ldetK <- determinant(K, logarithm=TRUE)$modulus
    ll <- - (n/2) * log(t(Y) %*% Ki %*% Y) - (1/2) * ldetK
    counter <<- counter + 1
    return(-ll)
  }
```

- And shove it into `optim`.

Back to the 2d example

For fun, lets switch back to our 2d example.

```
library(lhs)
X <- randomLHS(40, 2)
X <- rbind(X,X)
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=0.01)
```

Estimating a lengthscale and the nugget is an attempt at resolving a tension between signal and noise.

- (That's one reason why I have re-generated the data here to contain replications.)
- (And in the 1d sinusoidal example too.)

Joint optimization

It helps to think a little about starting values and search ranges.

```
D <- distance(X)
out <- optim(c(0.1, 0.1*var(y)), nl, method="L-BFGS-B",
  lower=eps, upper=c(10, var(y)), D=D, Y=y)
out$par
```

```
## [1] 1.132886675 0.009561202
```

- Actually, pretty close to our initial version with implied $\theta = 1$.

Since `"L-BFGS-B"` is calculating a gradient numerically, the reported count of evaluations in the output doesn't match the number of actual evaluations:

```
print(brute <- c(out$counts, actual=counter))
```

```
## function gradient   actual
##       27       27      135
```

Predictive surface

Re-building the data quantities

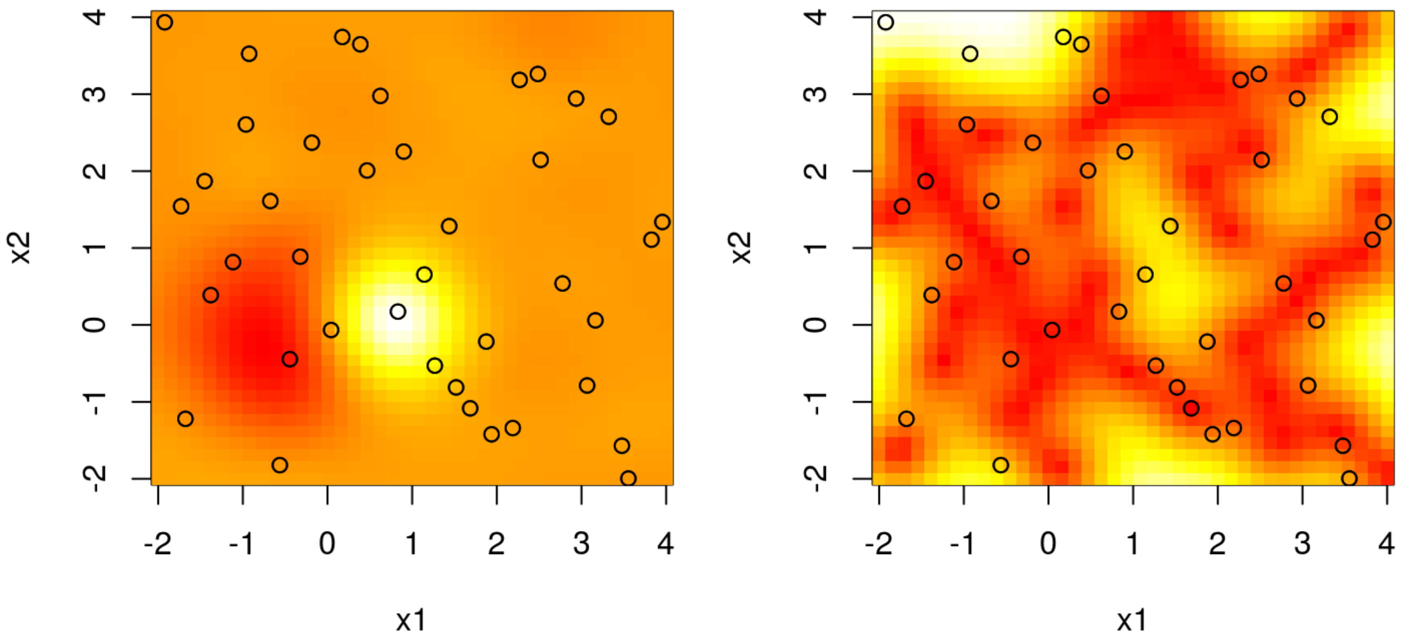
```
K <- exp(- D/out$par[1]) + diag(out$par[2], nrow(X))
Ki <- solve(K)
tau2hat <- drop(t(y) %*% Ki %*% y / nrow(X))
```

And then the predictive quantities.

```
xx <- seq(-2,4, length=40)
XX <- expand.grid(xx, xx)
DXX <- distance(XX)
KXX <- exp(-DXX/out$par[1]) + diag(out$par[2], ncol(DXX))
DX <- distance(XX, X)
KX <- exp(-DX/out$par[1])
```

Pretty much the same as before.

```
mup <- KX %*% Ki %*% y; Sigmap <- tau2hat * (KXX - KX %*% Ki %*% t(KX))
par(mfrow=c(1,2))
image(xx,xx, matrix(mup, ncol=length(xx)), xlab="x1",ylab="x2", col=cols)
points(X[,1], X[,2])
image(xx,xx, matrix(sdp, ncol=length(xx)), xlab="x1",ylab="x2", col=cols)
points(X[,1], X[,2])
```



Can we do better?

What if we take derivatives with respect to θ, and combine with those for g and form a gradient?

We'll need $\dot{K}_n \equiv \frac{\partial K_n}{\partial \theta}$.

- The diagonal is zero.
- The off diagonal entries of $\dot{K}_n$ are calculated as follows.

$$\begin{aligned} \text{Since} \qquad & K_\theta(x_i, x_j) = \exp\left\{ - \frac{||x - x'||^2}{\theta} \right\} \\ \text{we have} \qquad & \frac{\partial K_\theta(x_i, x_j)}{\partial \theta} = K_\theta(x_i, x_j) \frac{||x - x'||^2}{\theta^2}. \end{aligned}$$

So actually we have $\dot{K}_n = K_n \cdot \mathrm{Dist}_n / \theta^2$ where the product is component-wise, and Dist_n contains a matrix of Euclidean distances.

The full derivative

So we have

$$\ell'(\theta) = \frac{n}{2} \frac{Y_n K_n^{-1} \dot{K}_n K_n^{-1} Y_n}{Y_n K_n^{-1} Y_n} - \frac{1}{2} \mathrm{tr} \left\{ K_n^{-1} \dot{K}_n \right\}$$

```
gradnl <- function(par, D, Y)
 {
    theta <- par[1]; g <- par[2]
    n <- length(Y)
    K <- exp(-D/theta) + diag(g, n)
    Ki <- solve(K)
    dotK <- K * D / theta^2
    KiY <- Ki %*% Y

    ## for theta then g
    dlltheta <- (n/2) * t(KiY) %*% dotK %*% KiY / (t(Y) %*% KiY) -
      (1/2) * sum(diag(Ki %*% dotK))
    dllg <- (n/2) * t(KiY) %*% KiY / (t(Y) %*% KiY) - (1/2) * sum(diag(Ki))

    return(-c(dlltheta, dllg))
 }
```

How does it work?

```
counter <- 0
outg <- optim(c(0.1, 0.1*var(y)), nl, gradnl, method="L-BFGS-B",
  lower=eps, upper=c(10, var(y)), D=D, Y=y)
rbind(grad=outg$par, brute=out$par)
```

```
##           [,1]        [,2]
## grad  1.133165 0.009526717
## brute 1.132887 0.009561202
```

- Nearly identical result.

What about number of evaluations?

```
rbind(grad=c(outg$counts, actual=counter), brute)
```

```
##       function gradient actual
## grad        12       12     12
## brute       27       27    135
```

- Woah, way better!

Alright, what else?

Lets expand the dimension a bit, and get ambitious. Visualization will be hard, but we have other (relative) progress metrics.

Consider the so-called Friedman function from the MARS paper.

```
fried <- function (n=50, p=6)
{
  if(p < 5) stop("must have at least 5 cols")
  X <- matrix(runif(n * p), nrow = n)
  Ytrue <- 10*sin(pi*X[,1]*X[,2]) + 20*(X[,3]-0.5)^2 + 10*X[,4] + 5*X[,5]
  Y <- Ytrue + rnorm(n, 0, 1)
  return(data.frame(X, Y, Ytrue))
}
```

Lets generate training and testing sets.

```
n <- 200; p <- 7
train <- fried(n, p); test <- fried(1000, p)
X <- as.matrix(train[,1:p]); XX <- as.matrix(test[,1:p])
y <- drop(train$Y); yy <- drop(test$Y); yytrue <- drop(test$Ytrue)
```

Fitting the GP hyperparameters

Lets learn the isotropic Gaussian lengthscale θ, and the nugget g.

```
D <- distance(X)
print(out <- optim(c(0.1, 0.1*var(y)), nl, gradnl, method="L-BFGS-B",
  lower=eps, upper=c(10, var(y)), D=D, Y=y))
```

```
## $par
## [1] 2.261433371 0.006635327
##
## $value
## [1] 667.0765
##
## $counts
## function gradient
##       33       33
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

Completing the fit

Re-building the data quantities.

```
K <- exp(- D/out$par[1]) + diag(out$par[2], nrow(D))
Ki <- solve(K)
tau2hat <- drop(t(y) %*% Ki %*% y / nrow(D))
```

And then the predictive quantities.

```
DXX <- distance(XX)
KXX <- exp(-DXX/out$par[1]) + diag(out$par[2], ncol(DXX))
DX <- distance(XX, X)
KX <- exp(-DX/out$par[1])
```

Predicting

```
mup <- KX %*% Ki %*% y
Sigmap <- tau2hat * (KXX - KX %*% Ki %*% t(KX))
```

- Before looking at raw results, what are we going to compare them to?

A comparator

How about MARS?

```
library(mda)
fit.mars <- mars(X, y)
p.mars <- predict(fit.mars, XX)
```

Which wins on RMSE to the truth?

```
rmse <- c(gp=sqrt(mean((yytrue-mup)^2)),mars=sqrt(mean((yytrue-p.mars)^2)))
rmse
```

```
##       gp     mars
## 1.259379 1.509923
```

- Usually the GP; and MARS doesn't come with a notion of predictive variance.

Even better?

How can we improve upon our GP results?

Is it reasonable for correlation to decay uniformly in each input direction?

- I.e., to decay radially, modulated by a scalar θ?

How about the following generalization?

$$C_\theta(x, x') = \exp\left\{ -\sum_{k=1}^{m} \frac{(x_k - x'_k)^2}{\theta_k} \right\}$$

- Here we are using a vectorized lengthscale parameter $\theta = (\theta_1, \dots, \theta_m)$.

This correlation function is called the **separable** or **anisotropic Gaussian**.

- Separable because the sum is a product when taken outside the exponent, implying independence in each coordinate direction.

Inference?

How would we do inference for such a vectorized parameter?

Simple; just expand the log likelihood and derivative functions to work with a vectorized θ.

- In particular, a `for` in the gradient function can iterate over coordinates.
- For that we'll need to plug

  $$\frac{\partial K_n^{ij}}{\partial \theta_k} = K_n^{ij} \frac{(x_{ik} - x_{jk})^2}{\theta_k^2}.$$

  into our formula for each $\ell'(\theta_k)$, which is otherwise unchanged.

Each coordinate has a different θ_k, so pre-computing a distance matrix isn't helpful.

- Instead we'll use the `covar.sep` function from the `plgp` package which takes `d` $= \theta$ and `g` arguments.

Separable log likelihood

Before going derivative crazy, lets focus on the likelihood.

```
nlsep <- function(par, X, Y)
  {
    theta <- par[1:ncol(X)]
    g <- par[ncol(X)+1]
    n <- length(Y)
    K <- covar.sep(X, d=theta, g=g)
    Ki <- solve(K)
    ldetK <- determinant(K, logarithm=TRUE)$modulus
    ll <- - (n/2) * log(t(Y) %*% Ki %*% Y) - (1/2) * ldetK
    counter <<- counter + 1
    return(-ll)
  }
```

- As simple as that.

Optimizing separable lengthscale

Here we go.

```
tic <- proc.time()[3]; counter <- 0
out <- optim(c(rep(0.1, ncol(X)), 0.1*var(y)), nlsep, method="L-BFGS-B", X=X, Y=y,
  lower=eps, upper=c(rep(10, ncol(X)), var(y)))
out$par
```

```
## [1]  1.21206177  0.97862525  1.38052827  7.36016544 10.00000000  8.55949130
## [7]  9.81061953  0.00927567
```

And how about the number of evaluations?

```
brute <- c(out$counts, actual=counter, time=proc.time()[3]-tic)
brute
```

```
##     function     gradient       actual time.elapsed
##       93.000       93.000     1581.000       12.952
```

- Lots!

Gradient calculation

```
gradnlsep <- function(par, X, Y)
 {
    theta <- par[1:ncol(X)]
    g <- par[ncol(X)+1]
    n <- length(Y)
    K <- covar.sep(X, d=theta, g=g)
    Ki <- solve(K)
    KiY <- Ki %*% Y

    ## loop over theta components
    dlltheta <- rep(NA, length(theta))
    for(k in 1:length(dlltheta)) {
      dotK <- K * distance(X[,k])/(theta[k]^2)
      dlltheta[k] <- (n/2) * t(KiY) %*% dotK %*% KiY / (t(Y) %*% KiY) -
        (1/2) * sum(diag(Ki %*% dotK))
    }

    ## for g
    dllg <- (n/2) * t(KiY) %*% KiY / (t(Y) %*% KiY) - (1/2) * sum(diag(Ki))

    return(-c(dlltheta, dllg))
 }
```

Now with closed form gradients.

```
tic <- proc.time()[3]; counter <- 0
outg <- optim(c(rep(0.1, ncol(X)), 0.1*var(y)), nlsep, gradnlsep,
  method="L-BFGS-B", lower=eps, upper=c(rep(10, ncol(X)), var(y)), X=X, Y=y)
round(rbind(grad=outg$par, brute=out$par), 5)
```

```
##          [,1]    [,2]    [,3]    [,4] [,5]     [,6]     [,7]    [,8]
## grad  1.26196 1.02280 1.50416 9.68139   10 10.00000 10.00000 0.00841
## brute 1.21206 0.97863 1.38053 7.36017   10  8.55949  9.81062 0.00928
```

- Nearly identical result.

What about number of evaluations?

```
rbind(grad=c(outg$counts, actual=counter, time=proc.time()[3]-tic), brute)
```

```
##       function gradient actual time.elapsed
## grad       133      133    133        6.034
## brute       93       93   1581       12.952
```

- Far fewer!

Predictive accuracy

How does the separable GP compare against the isotropic one and MARS?

```
K <- covar.sep(X, d=out$par[1:ncol(X)], g=out$par[ncol(X)+1]); Ki <- solve(K)
tau2hat <- drop(t(y) %*% Ki %*% y / nrow(X))
KXX <- covar.sep(XX, d=out$par[1:ncol(X)], g=out$par[ncol(X)+1])
KX <- covar.sep(XX, X, d=out$par[1:ncol(X)], g=0)
mup <- KX %*% Ki %*% y
Sigmap <- tau2hat * (KXX - KX %*% Ki %*% t(KX))
```

Evaluating by RMSE.

```
rmse <- c(rmse, gpsep=sqrt(mean((yytrue - mup)^2)))
rmse
```

```
##        gp      mars     gpsep
## 1.2593787 1.5099234 0.8645556
```

- Woot! But all this cutting and pasting …

… isn't there a library for that?

```
library(laGP)
tic <- proc.time()[3]
gpi <- newGPsep(X, y, d=0.1, g=0.1*var(y), dK=TRUE)
## the MLE calculation is (Bayes) integrated rather than concentrated
mle <- mleGPsep(gpi, param="both", tmin=c(eps, eps), tmax=c(10, var(y)))
elapsed <- as.numeric(proc.time()[3] - tic)
p <- predGPsep(gpi, XX)
deleteGPsep(gpi)
rmse <- c(rmse, bobby=sqrt(mean((yytrue - p$mean)^2)))
rmse
```

```
##        gp      mars     gpsep     bobby 
## 1.2593787 1.5099234 0.8645556 0.8444533
```

- There are several libraries, but `laGP`'s optimized `C` backend is the fastest.

```
elapsed
```

```
## [1] 0.709
```