LatticeKrig Vignette

Matthew Iverson

5/21/2019

Contents

1 Introduction

In this vignette, we will explore the functions in the LatticeKrig package and show examples of how they can be used to solve problems. The LatticeKrig model is an example of the spatial statistics method known as kriging, adapted to large data sets.

1.1 What is Kriging?

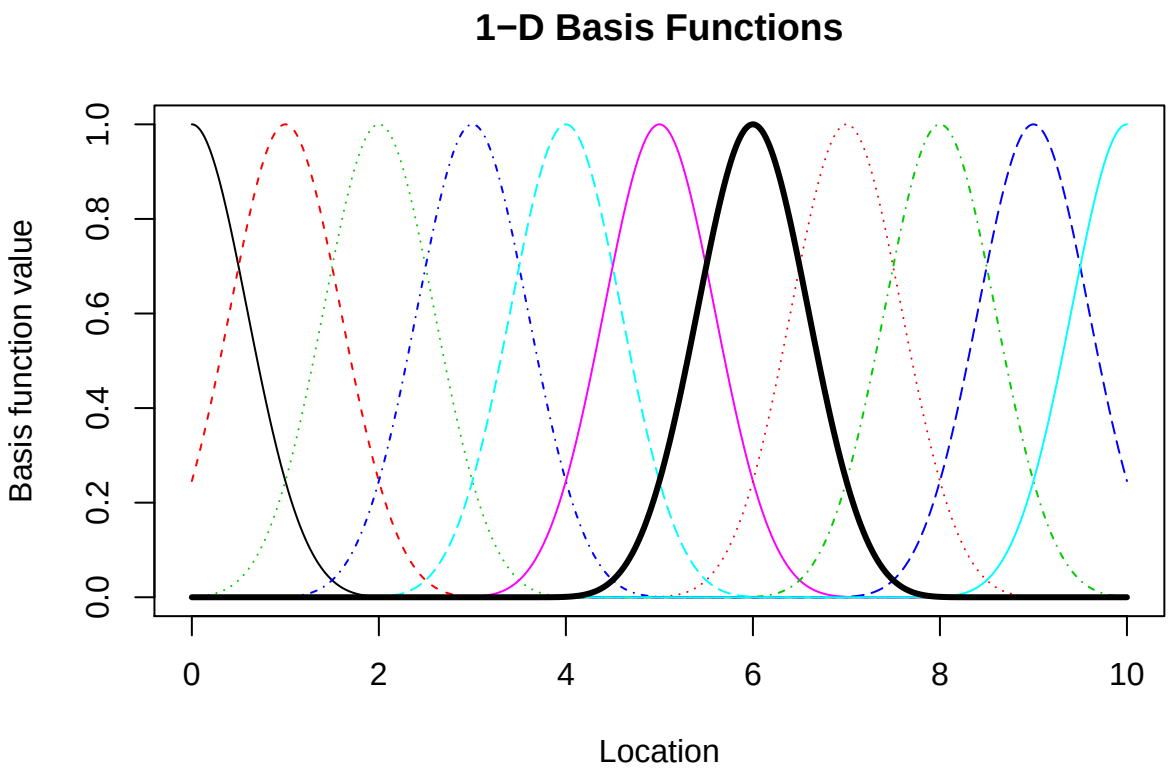
Kriging (named for South African satistician Danie Krige) is a method for making predictions from a spatial data set. A spatial data set means the data contains the observed variable and its location, and pairs of observations taken close together have similar values. For example, the current temperature in cities would be spatial data. As such, kriging can be applied to a variety of important data sets, from geological data to atmospheric data.

The standard spatial model for Kriging relates the observation to a sum of three components: A polynomial function of the locations (and covariates, if provided), a spatial process, and measurement error.

1.2 The LatticeKrig Model

In the LatticeKrig package, we model spatial process as the sum of basis functions scaled by coefficients, which we assume are correlated. The smooth basis functions and correlated coefficients create a smooth function representation for the spatial process. The structure of the basis functions and covariance has some flexibility, so you can change the structure to make a more reasonable model for a certain problem. We fit the linear polynomial using generalized least squares, so the linear polynomial will be as close as possible to all of the data. To approximate the spatial process, we then fit the basis functions to the residuals from the linear model. In terms of linear algebra, the model is $\mathbf{y} = Z\mathbf{d} + \Phi\mathbf{c} + \mathbf{e}$, where $\mathbf{y}$ is the vector of variable measurements, Z is the matrix of locations and covariates, $\mathbf{d}$ is the vector of coefficients for the linear model, Φ is the matrix of basis functions evaluated at the data points, $\mathbf{c}$ is the vector of coefficients for each basis function, and $\mathbf{e}$ is the measurement error. We show the derivations of the equations for $\mathbf{c}$ and $\mathbf{d}$ in Appendix A, and show how all of these calculations are done in Appendix C.

The package is named LatticeKrig because of the placement of the basis functions: they are equally spaced in all dimensions on a lattice. We can also consider multiple different lattice sizes simultaneously to better capture different levels of resolution; each additional level has half as much space between the basis functions in each dimension. The following plot shows an example of 11 basis functions in the region from 0 to 10, with the one centered around 6 highlighted for emphasis.



1.3 Glossary of Important Functions

- `LatticeKrig`: Top level function that sets up the default spatial model, estimates some key spatial parameters, and uses the `LKrig` function for the kriging computation. `LatticeKrig` can use a minimal set of inputs and is a quick way to fit a kriging model to data.
- `LKrig`: Performs the Kriging computation for a fixed LatticeKrig model. This is the main computational step in the package.
- `LKrigSetup`: Creates an `LKinfo` object, which is a list to store the parameters to use for a `LatticeKrig` or `LKrig` call; especially useful for examining the effect of changing one parameter on the fit.
- `surface`: Plots a fitted surface in 2D space as a color plot and adds contour lines.
- `image.plot`: Plots a dataset or fitted surface in 2D space as a color plot without contour lines.
- `predictSurface`: Computes the values from a Kriging fit and makes a surface, but doesn't plot it.

2 Quick Start Guide

In this section, we will lay out the bare essentials of the package as a quick overview. To fit a surface and interpolate data using `LatticeKrig`, the only required arguments are, naturally, the measurement locations (formatted in a matrix where each row is one location) and measurement values. However, we highly recommend using some of the optional parameters to customize the model to your specific data problem - several ways to do this are illustrated in this vignette. Calling the `LatticeKrig` function and passing in the locations and values will produce an `LKrig` object that contains all the information needed to predict the variable at any location.

For a simple, 1-dimensional example, we will take our locations to be 50 randomly spaced points on the interval $[-6, 6]$, and our variable measurements to be the values of $\sin(x)$ at these locations. The goal of our kriging fit is to estimate this smooth curve from the observations.

2.1 Fitting the model in one dimension

```
set.seed(223)
locations <- runif(50, min=-6, max=6)
locations <- as.matrix(locations)
observations <- sin(locations) + rnorm(50, sd = 1e-1)
kFit1D <- LatticeKrig(locations, observations)
```

Now we will print out the `LKrig` object: this list features the data's estimated covariance scale `rho` and estimated standard measurement error `sigma`, and many basis function settings: the type of basis function, how distance is measured, and the number and spacing of basis functions. In this example, all of this information is determined by `LatticeKrig` from the defaults, but can be changed with optional parameters.

```
print(kFit1D)
```

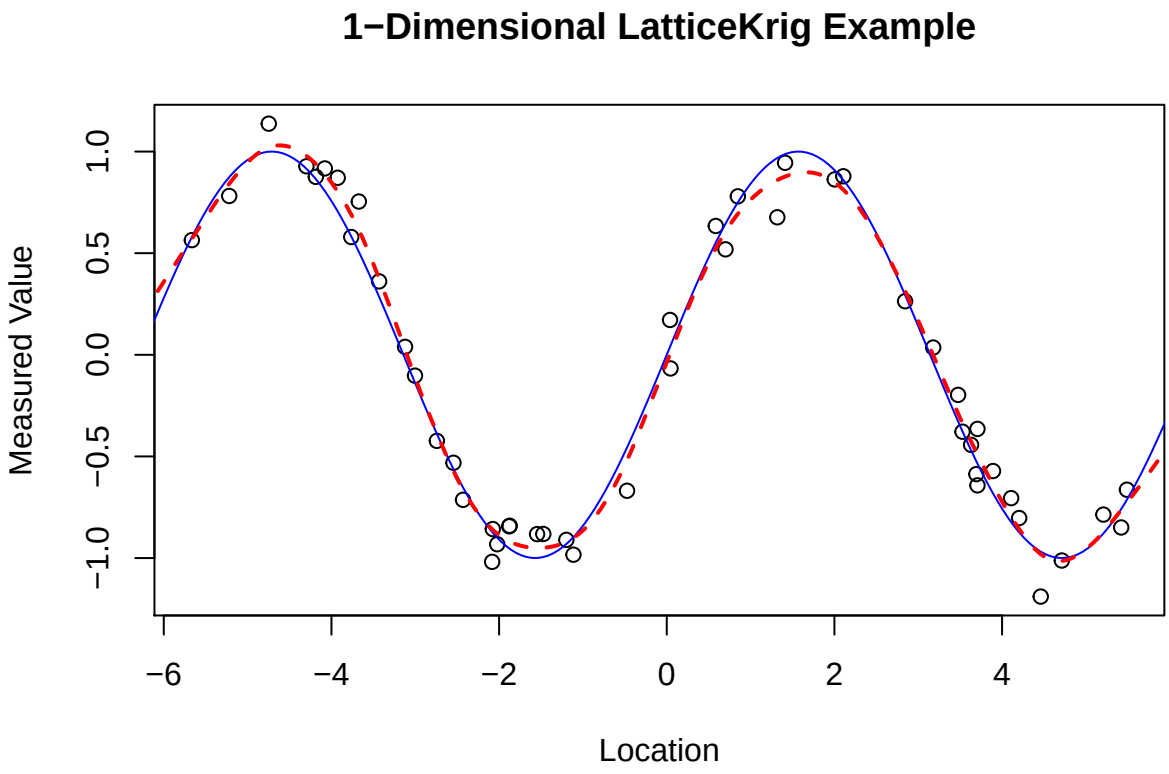
```
## Call:
## LatticeKrig(x = locations, y = observations)
##
##
##  Number of Observations:                      50
##  Number of parameters in the fixed component 2
##   Effective degrees of freedom (EDF)         10.34
##     Standard Error of EDF estimate:          0.9605
##  MLE sigma                                   0.1062
##  MLE rho                                     91.19
##  MLE lambda = sigma^2/rho                    0.0001237
##
## Fixed part of model is a polynomial of degree 1 (m-1)
## Basis function :  Radial
## Basis function used:  WendlandFunction
## Distance metric:  Euclidean
##
## Lattice summary:
## 3  Level(s) 75 basis functions with overlap of  2.5 (lattice units)
##
##  Level Lattice points   Spacing
##      1             17 1.8592024
##      2             23 0.9296012
##      3             35 0.4648006
##
## Nonzero entries in Ridge regression matrix 806
```

```
## NULL
```

##Plotting the results

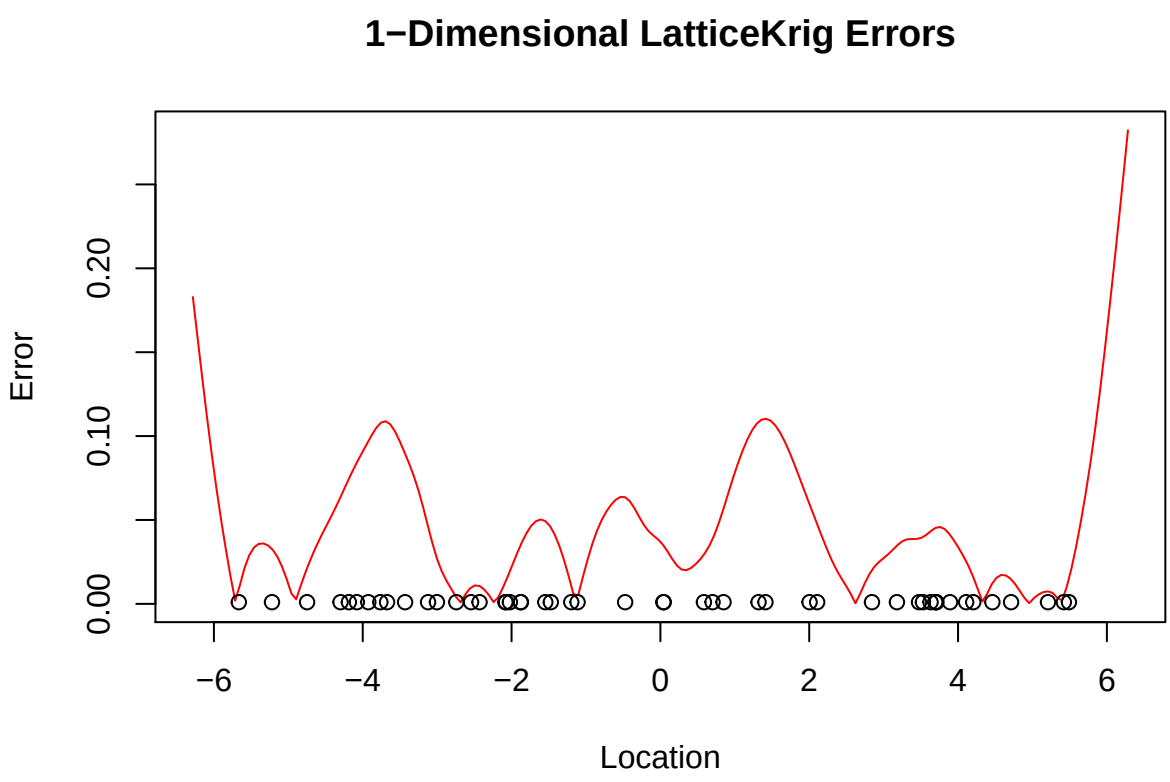
Now, we'll make a plot of the original 50 data points and the true function ($\sin(x)$) and the `LatticeKrig` fit at 200 equally spaced points to compare them.

```
xGrid <- seq(-2*pi, 2*pi, len=200)
prediction <- predict(kFit1D, xGrid)
plot(locations, observations, main="1-Dimensional LatticeKrig Example",
     xlab="Location", ylab="Measured Value")
lines(xGrid, sin(xGrid), col='blue')
lines(xGrid, prediction, col='red', lty=2, lwd=2)
```



For this example, the fitted curve (in red) matches the true function (in blue) rather closely. This next plot shows the error in the fitted curve; note that the error gets larger where there aren't many data points, and especially at the edges of the region.

```
#uncertainty <- LKrig.sim.conditional(kFit1D)
plot(xGrid, abs(sin(xGrid) - predict(kFit1D, xGrid)), col='red', type="l",
     main="1-Dimensional LatticeKrig Errors", xlab="Location", ylab="Error")
#plot(uncertainty)
points(locations, rep(1e-3, 50))
```

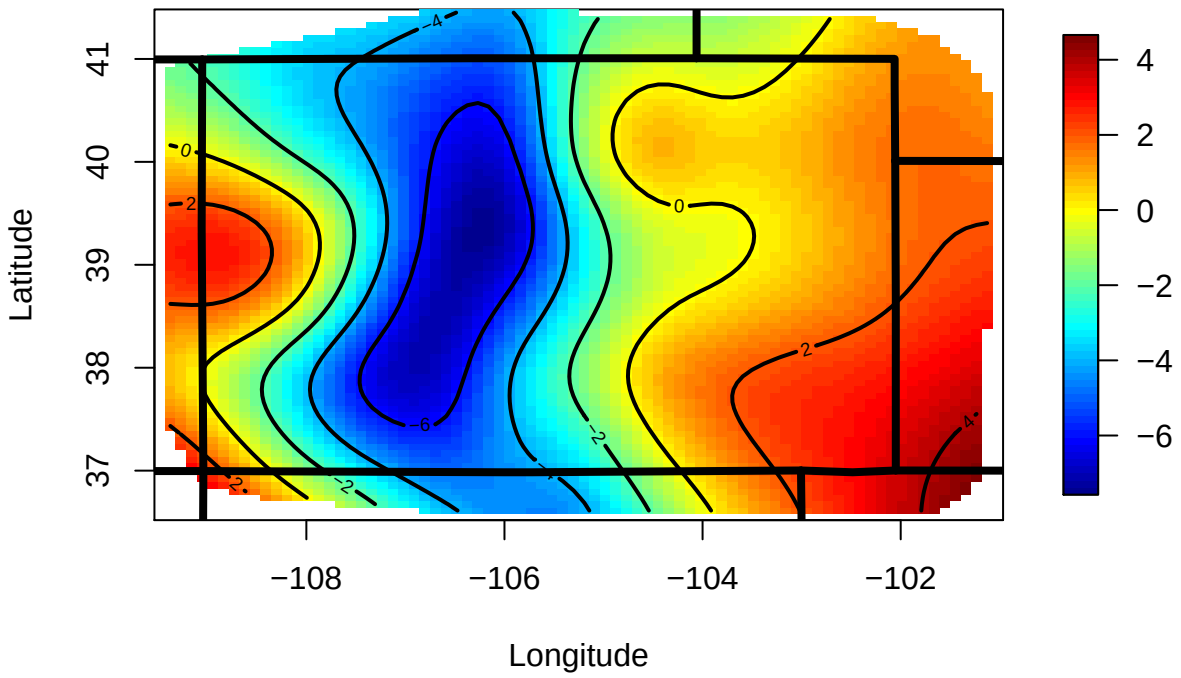


For another, more practical example, we will predict the average spring temperature for locations throughout Colorado. Using the data set `COmonthlyMet`, we can make a surface showing our predictions over a range of longitudes and latitudes, and use the `US` function to draw in the USA state borders to show where Colorado is. Notice that `LatticeKrig` will automatically discard any data points with missing values (NAs) if needed.

```
data(COmonthlyMet)
locations <- CO.loc
observations <- CO.tmean.MAM.climate
kFit <- LatticeKrig(locations, observations)
```

```
## Warning in LatticeKrig(locations, observations): NAs removed
```

```
surface(kFit, main = "2-Dimensional LatticeKrig Example",
        xlab="Longitude", ylab="Latitude")
US(add=TRUE, col='black', lwd=4)
```

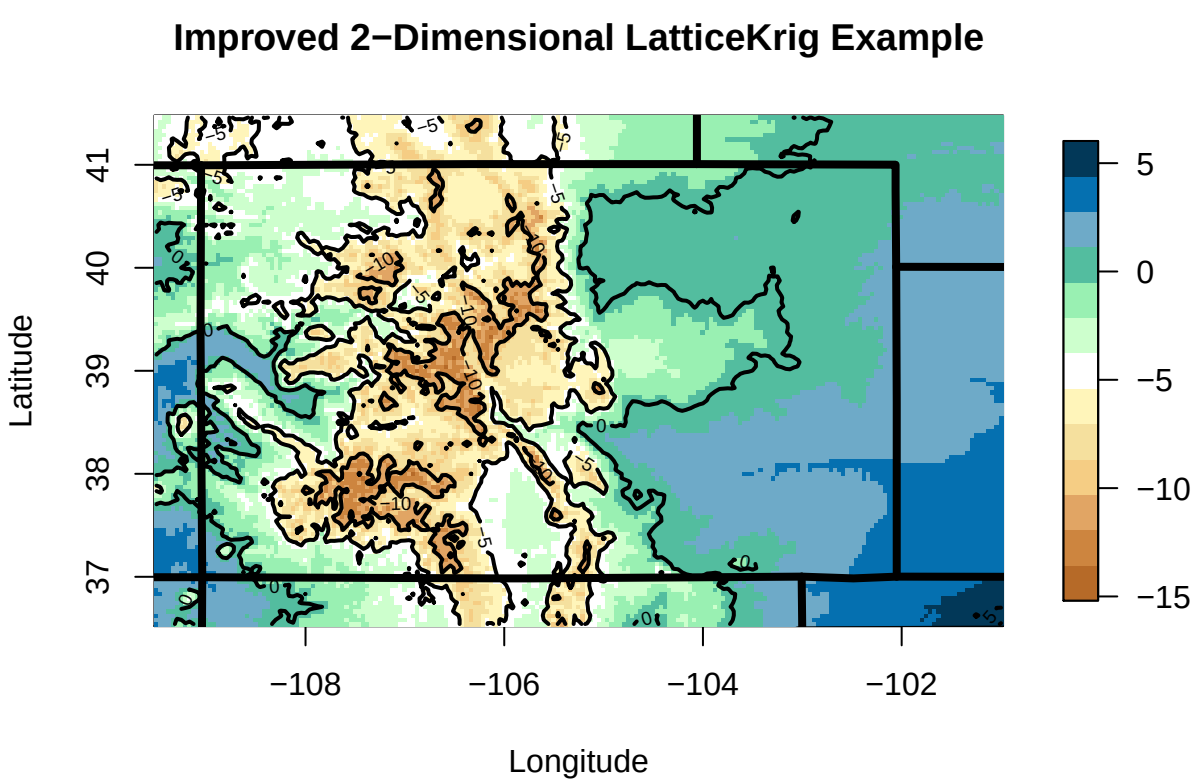


This plot is useful, but we can do better. We can see that the coldest temperatures are in the Rocky Mountains, which is not surprising. Thus, we might expect that we will get a more accurate fit by having `LatticeKrig` account for the elevation at each location as well. Another way we can improve the plot is by increasing its resolution - the current image is somewhat pixelated. The `surface` function will evaluate the surface at more points if we increase the `nx` and `ny` arguments: setting `nx=200, ny=150` will produce a , which will take longer to compute but produces a nicer looking, more detailed plot. Finally, we can also have `surface` extend the calculation all the way to the corners of the window by using the `extrap` argument; by default it doesn't extrapolate outside of the existing data, since the error often increases dramatically when predicting outside of the given data. However, extending the plot to the corners will make it look nicer. For the sake of example, we will also change the color scale in the image by setting the `col` parameter.

```
data(COmonthlyMet)
locations <- CO.loc
observations <- CO.tmean.MAM.climate
elevations <- CO.elev
kFit <- LatticeKrig(locations, observations, Z=cbind(elevations))
```

```
## Warning in LatticeKrig(locations, observations, Z = cbind(elevations)): NAs
## removed
```

```
# look at the help file in fields for information on the grid.list format
prediction <- predictSurface(kFit, grid.list = CO.Grid, ZGrid = CO.elevGrid,
                             nx = 200, ny = 150, extrap = TRUE)
surface(prediction, main = "Improved 2-Dimensional LatticeKrig Example",
        xlab="Longitude", ylab="Latitude", col=larry.colors())
US(add=TRUE, col='black', lwd=4)
```



This surface is so rough because it accounts for elevation; we can see that the plot is fairly smooth in the eastern half of the state, and extremely rough in the mountains.

Finally, it is important to note some potential issues that `LatticeKrig` calculations won't catch. Because `LatticeKrig` estimates some parameters of the data, the model could be a poor fit if the estimates aren't reasonable. The `LatticeKrig` model also approximates a thin plate spline by default, which may not be a good fit for a given problem. Finally, as with other curve fitting techniques, you should examine the residuals of the model for any patterns or features that may indicate a poor fit.

3 LKrigSetup

The only required arguments for the LatticeKrig function are the set of locations and variable observations. However, LatticeKrig also allows for a variety of optional arguments to tweak the model that LatticeKrig uses. In this section we will list some of the most important optional parameters that can be passed into LatticeKrig; for a complete list, check the LatticeKrig help page. The LKrigSetup function is a convenient (and, in some cases, the only) way to pass in a group of parameters to LatticeKrig or LKrig. We will cover the required parameters and some of the more important optional parameters here; for full descriptions, check the help pages for LatticeKrig and LKrigSetup.

##Required Parameters for LKrigSetup

- x

The parameter x is the matrix of observation locations, which is needed for making the covariance matrix, basis function matrix, and basis function lattice.

- nlevel

The parameter nlevel is an integer that determines the number of different lattice sizes the computation should run on. This is set to 3 by default in LatticeKrig, so it is recommended to set nlevel = 3 here as well unless there is a reason to change it. However, note that increasing nlevel will increase the computation time significantly. The coefficients at each different lattice size is computed independently, and the resulting coefficients are scaled by the weights in alpha.

- NC

The parameter NC is an integer that determines the number of basis functions to put along the largest dimension at the first level. Note that the actual number of basis functions will be different because there are 5 additional basis functions (this can be changed with the optional NC.buffer parameter) added outside the domain on the each end of longest side. For example, if the domain for the data locations is a rectangle whose length is double its width and NC = 6, the first level of basis functions will contain 16x13 basis functions (6x3 inside the domain with 5 extended from each edge).

- alpha or alphaObject

At least one of alpha and alphaObject must be set. In most cases you will use alpha, which should be a vector of length nlevel that holds the weights that scale the basis functions on each different lattice size. Since each level is calculated independently, the sum of the weights in alpha should be 1 to make sure the model fits correctly.

- a.wght or a.wghtObject

At least one of a.wght and a.wghtObject must be set. In most cases you will use a.wght, which can be either a scalar or a vector of length nlevel. The minimum value for this parameter varies depending on the geometry and the number of dimenions: in default geometry, the minimum value is two times the number of dimensions, and it is recommended to add a small fraction. For example, in 2 dimensions, you might set a.wght = 4.01. When using the LKSphere geometry, the minimum value for a.wght is 1, and again a small decimal should be added on.

3.1 Optional parameters

- lambda

Lambda is an estimate of the measurement error in the data. If not listed, LatticeKrig and LKrig will estimate it using generalized cross-validation.

- LKGeometry

Changing `LKGeometry` allows you to change the geometry used for kriging. For example, if the dataset covers the whole earth, it would be more appropriate to base the kriging on a sphere than a rectangle. This is covered in more depth in the next section.

- `distance.type`

When using a different `LKGeometry` than default, you may also need to change the `distance.type`. This is also covered in more depth in the next section.

- `NC.buffer`

This parameter determines how many lattice points to add outside the range of the data in each direction. The effect of changing this parameter is relatively minor compared to the effect of changing `NC`, and it can only affect the prediction near the edges of the data.

4 Kriging in Different Geometries

By default, `LatticeKrig` will interpret the location data it receives as points in n-dimensional Euclidean space, and calculate the distance accordingly. However, this package also supports distance measurements on a sphere, interpreting the given locations as latitude and longitude on Earth. There are also other options for non-Euclidean geometries: a cylinder, which uses 3 dimensional cylindrical coordinates, and a ring, which takes 2 dimensional cylindrical coordinates (z and θ, with fixed radius). You can change the geometry used by passing it into the `LKGeometry` parameter of an `LKinfo` object. These are the names that `LKinfo` recognizes:

- `LKInterval`: 1 dimensional Euclidean space
- `LKRectangle`: 2 dimensional Euclidean space
- `LKBox`: 3 dimensional Euclidean space
- `LKSphere`: 2 dimensional spherical coordinates
- `LKCylinder`: 3 dimensional cylindrical coordinates
- `LKRing`: 2 dimensional cylindrical coordinates

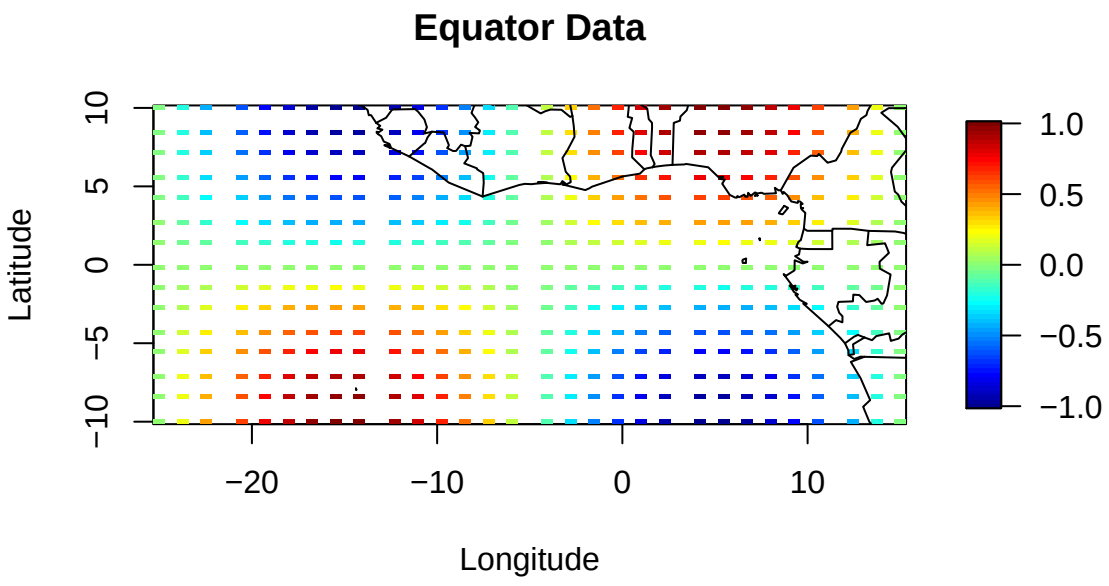
By default, `LKinfo` will use either `LKInterval`, `LKRectangle`, or `LKBox`, depending on the number of dimensions in the given location data. When using the `LKSphere` geometry, there are also different ways of measuring distance, which you can set using the `distance.type` parameter of the `LKinfo` object - the default is `GreatCircle`, which measures the shortest distance over the surface of the sphere, or you can use `Chordal` to measure the shortest straight-line distance, going under the surface of the sphere. Finally, when using the spherical geometry, you need to set `startingLevel`, which serves a similar role to `NC` from the Euclidean space. The `startingLevel` parameter controls how fine of a grid to use at the lowest level of the fit in spherical coordinates; for more information, check the `LKSphere` help page.

4.1 Working with spherical coordinates

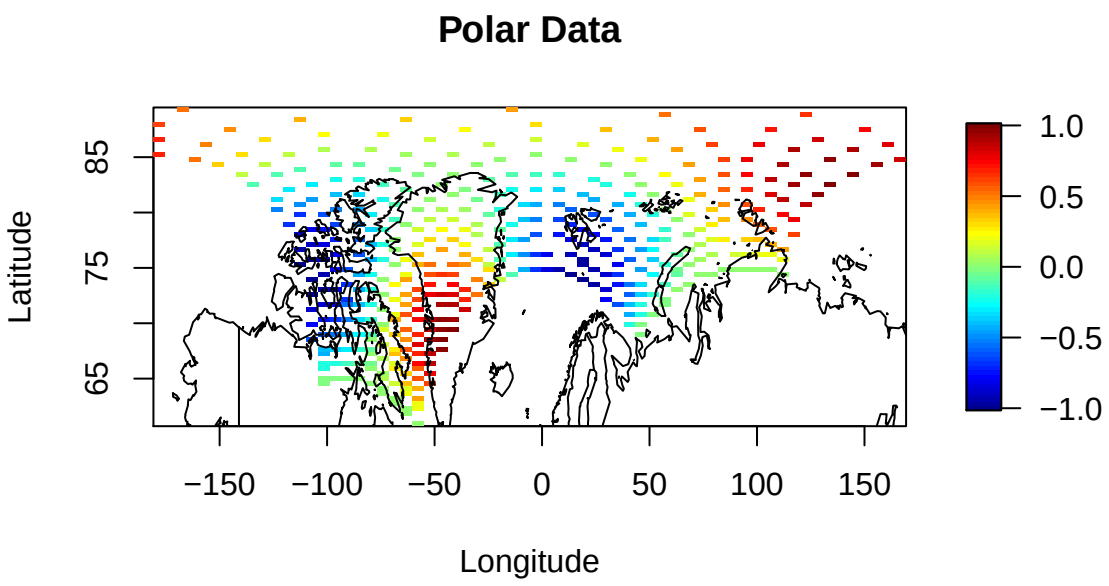
For an example of fitting data taken on the globe using spherical geometry instead of rectangular, we will create some sample data at the equator, rotate it up to near the north pole, and compare the models computed on the `LKRectangle` geometry and `LKSphere` geometry. We compute a kriging fit for the original data and the rotated data using the rectangular geometry and the spherical geometry, and print out the sum of squared errors as a measurement of how accurately the different fits match the data.

```
library(LatticeKrig)
#making a rectangle of points from (-20, -10) to (20, 10)
grid <- list(x = seq(-20, 20, len=30)-5, y = seq(-10, 10, len=15));
dataLocations <- make.surface.grid(grid)
dataValues <- sin(pi/20 * (5+dataLocations[,1])) *sin(pi/20 * dataLocations[,2])

#rotating the locations up 85 degrees to the north pole
theta = 85 * (pi/180)
cosineGrid <- directionCosines(dataLocations)
rotationMatrix <- cbind(c(cos(theta), 0, sin(theta)), c(0,1,0), c(-sin(theta), 0, cos(theta)))
newCosineGrid <- t(rotationMatrix %*% t(cosineGrid))
newLocations <- toSphere(newCosineGrid)
#plot the data at the equator and at the north pole in rectangular coordinates
#note the significant distortion at the north pole
quilt.plot(dataLocations, dataValues, main="Equator Data", xlab="Longitude", ylab="Latitude")
world(add=TRUE)
```

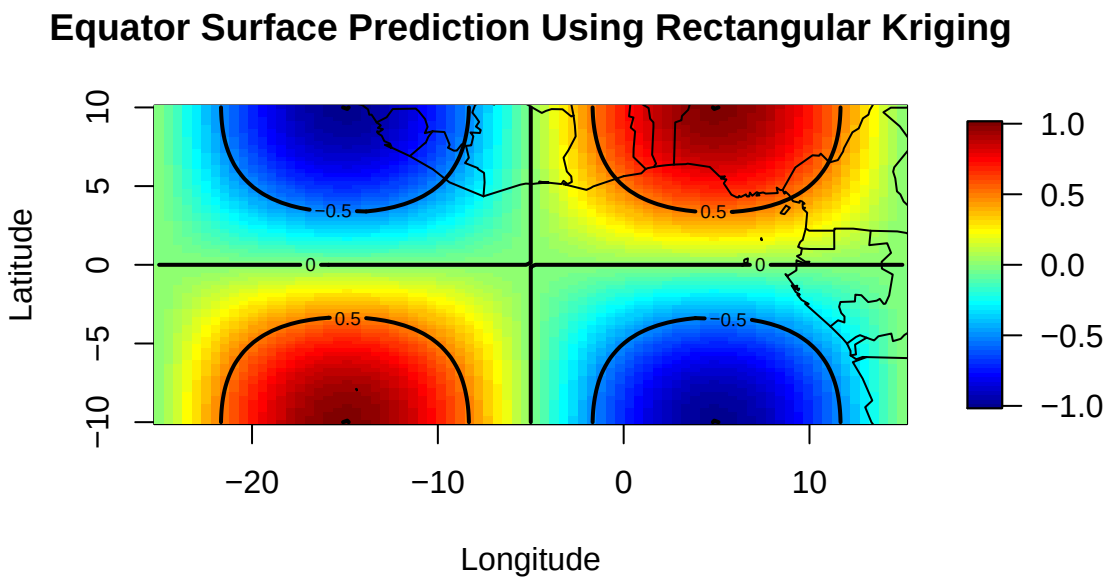



```
quilt.plot(newLocations, dataValues, main="Polar Data", xlab="Longitude", ylab="Latitude")
world(add=TRUE)
```



Now, we will use `LatticeKrig` to approximate the surface in both rectangular and spherical geometries.

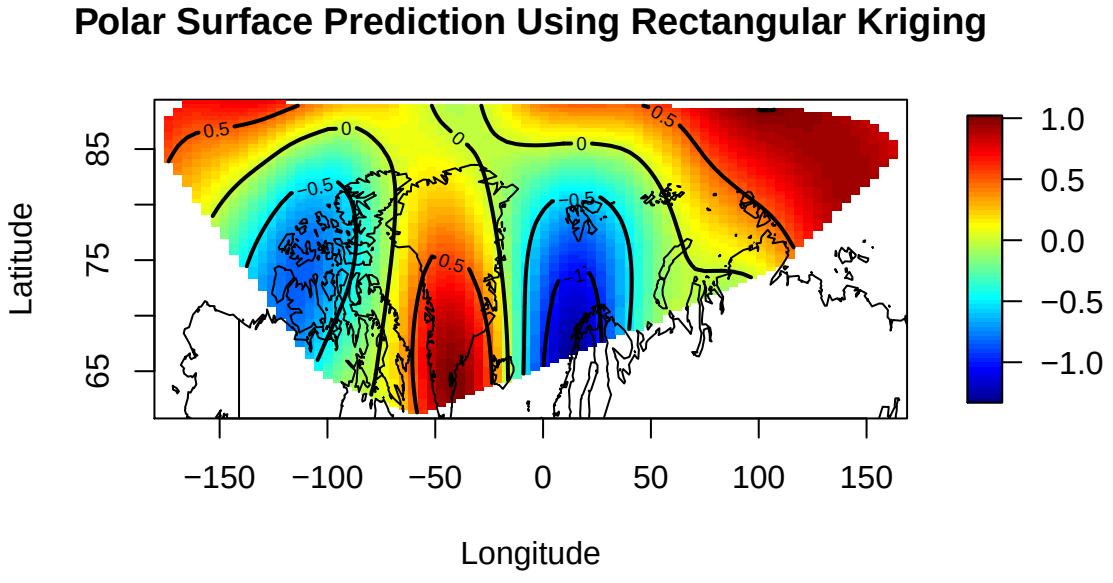
```
kFit1 <- LatticeKrig(dataLocations, dataValues)
surface(kFit1, main="Equator Surface Prediction Using Rectangular Kriging", xlab="Longitude", ylab="Lat
world(add=TRUE)
```




```
sum(kFit1$residuals^2)
```

```
## [1] 0.0008239103
```

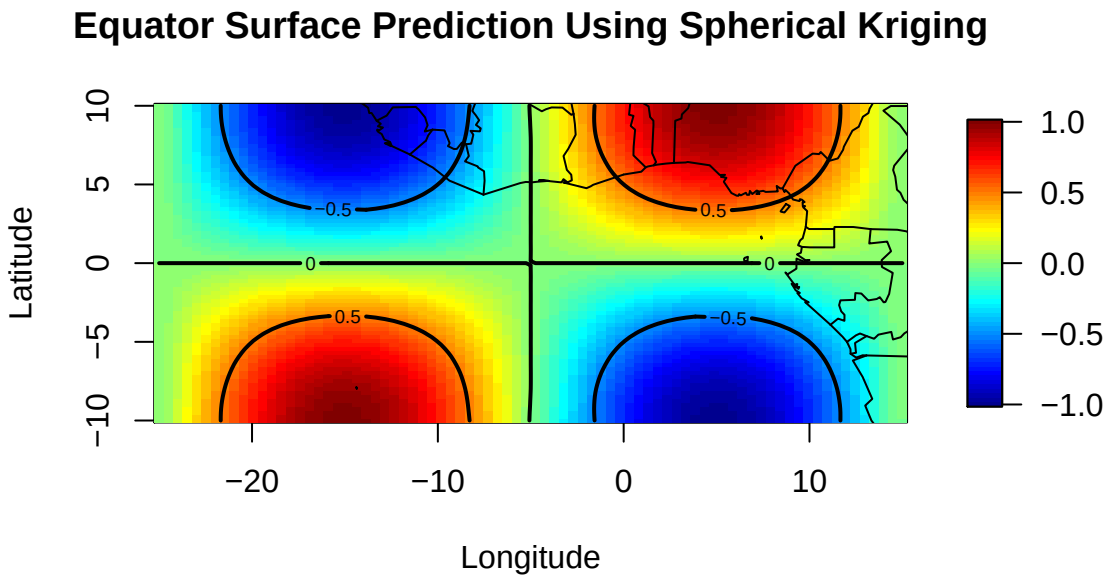
```
kFit2 <- LatticeKrig(newLocations, dataValues)
surface(kFit2, main="Polar Surface Prediction Using Rectangular Kriging", xlab="Longitude", ylab="Latitu
world(add=TRUE)
```




```
sum(kFit2$residuals^2)
```

```
## [1] 5.359673
```

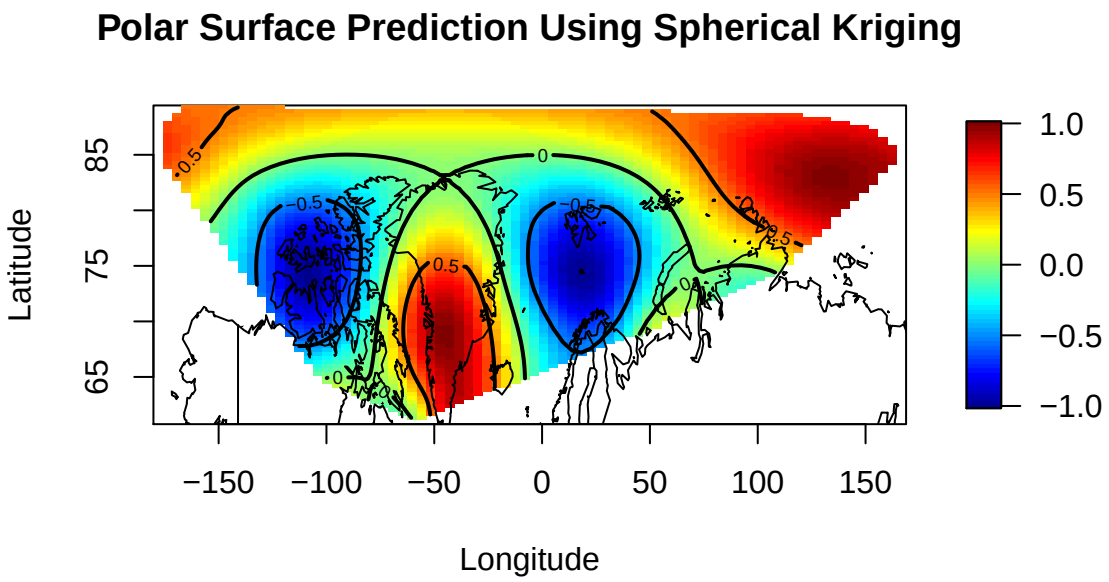
```
info1 <- LKrigSetup(dataLocations, nlevel = 2, startingLevel = 6, alpha = c(0.8, 0.2)
                    , a.wght = 1.01, LKGeometry = "LKSphere")
kFit1 <- LatticeKrig(dataLocations, dataValues, LKinfo = info1)
surface(kFit1, main="Equator Surface Prediction Using Spherical Kriging", xlab="Longitude", ylab="Latitu
world(add=TRUE)
```




```
sum(kFit1$residuals^2)
```

```
## [1] 0.01356611
```

```
info2 <- LKrigSetup(newLocations, nlevel = 2, startingLevel = 6, alpha = c(0.8, 0.2)
                    , a.wght = 1.01, LKGeometry = "LKSphere")
kFit2 <- LatticeKrig(newLocations, dataValues, LKinfo = info2)
surface(kFit2, main="Polar Surface Prediction Using Spherical Kriging", xlab="Longitude", ylab="Latitud
world(add=TRUE)
```




```
sum(kFit2$residuals^2)
```

```
## [1] 0.0014504
```

As we can see, the rectangular fit fails badly on the data that has been rotated up to the north pole, while the corresponding spherical model matches the data very well.

5 The difference between `LatticeKrig` and `LKrig`

As mentioned earlier, `LKrig` is the function that runs all the logic to compute the kriging fit: it calls the functions to create the covariate matrix, basis function matrix, and precision matrix; runs a series of intermediate calculations; calls another function that calculates the estimates for **c** and **d**; uses those coefficients to calculate the fitted values and residuals; calls a function that estimates the likelihood of the calculated fit; and finally estimates the effective degrees of freedom in the fitted surface. While this is a very powerful function, it lacks some ease-of-use features which are provided by `LatticeKrig`. `LKrig` has several parameters that need to be set before it can be called; `LatticeKrig` only needs the data and the locations. `LatticeKrig` can also automatically filter out any missing data values, but `LKrig` will fail if provided with missing values. `LatticeKrig` also makes use of the `LatticeKrigEasyDefaults` function, which (as the name implies) provides default values for `NC`, `nu`, and `a.wght`. It then builds an `LKinfo` object, which it uses in a function that estimates `lambda`, before finally calling `LKrig` to calculate the kriging fit as described above.

6 Common Errors

6.1 Could not find function [FunctionName]

Make sure that the library is installed (`install.packages("LatticeKrig")`) and activated (`library(LatticeKrig)`).

6.2 Need to specify NC for grid size

6.3 Invalid 'times' argument

6.4 Only one alpha specifed for multiple levels

6.5 Missing value where TRUE/FALSE needed

All of these errors can be caused by using `LKrig` instead of `LatticeKrig`. The `LatticeKrig` function has ways to either supply defaults or estimate all of the optional parameters that `LKrig` doesn't, so `LKrig` will produce errors like the ones above while `LatticeKrig` will work correctly.

6.6 Error in mLevel[l] <- nrow(grid.all.levels[[l]]) : replacement has length zero

This error can occur when using the `LKSphere` kriging geometry if your data only covers part of the sphere. This error occurs when not enough of the grid points on the sphere fall inside the range of the given data locations. To fix this problem, you can increase the value of `startingLevel` in the `LKinfo` object, or (if possible) take data over a larger section of the sphere. If neither of these options fixes the problem, then your dataset is likely small enough that it can be well approximated by the default `LKRectangle` geometry, unless it is very close to one of the poles. If it is, consider rotating it to near the equator (similar to in the spherical kriging example in this vignette) and then using `LKRectangle` geometry.

7 Frequently Asked Questions

7.1 Does the order the parameters are listed in matter?

The order of the parameters only matters when you pass them in without specifying their names: for example `LatticeKrig(locations, values)` works, but `LatticeKrig(values, locations)` doesn't. However, if the names are specified, either order will work correctly: both `LatticeKrig(x = locations, y = values)` and `LatticeKrig(y = values, x = locations)` work as intended. The optional parameters can also be listed without their names, but then they would need to be in the correct order and every single one would need to be specified, so it is highly recommended to include the names alongside each optional parameter.

7.2 The predicted values from my Kriging fit are nowhere near the data; what's wrong?

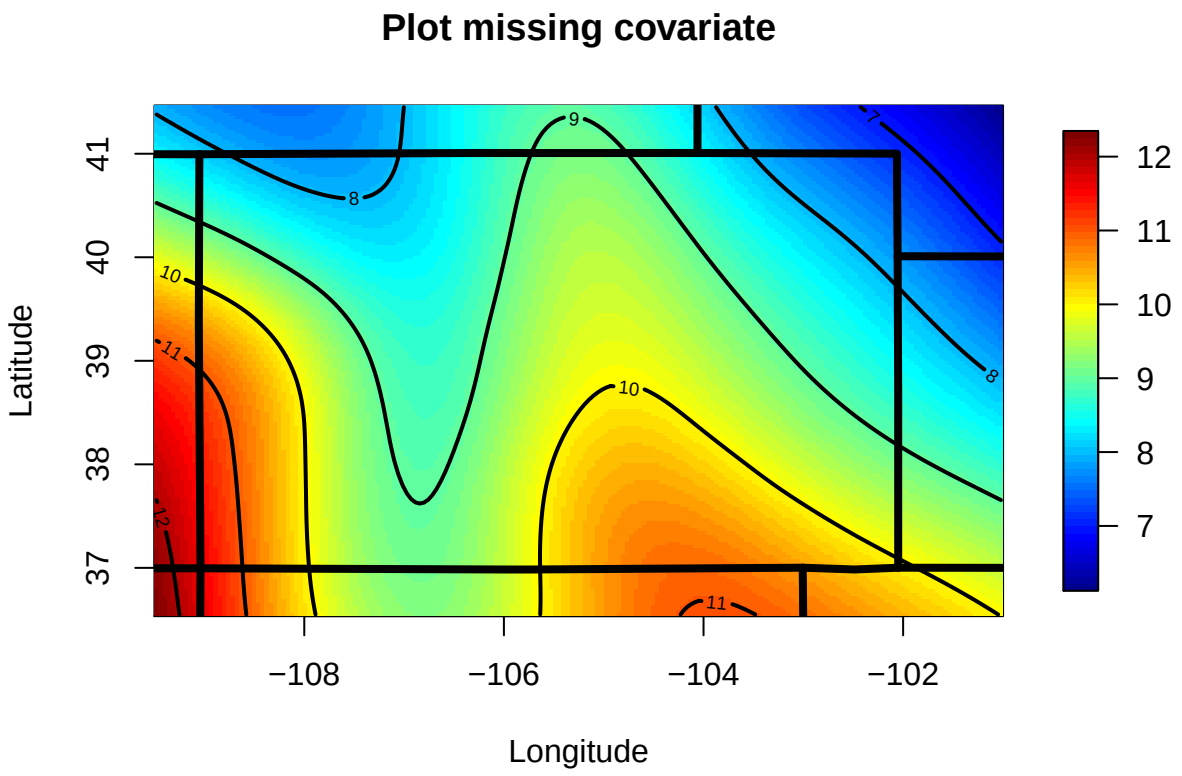
If your model includes covariates (the `Z` parameter of `LatticeKrig` and `LKrig`), your plot may not have included the effect of the covariate. The following code demonstrates this issue using the Colorado temperature data and how to fix it; first, we will set up the model.

```
data(COmonthlyMet)
locations <- CO.loc
observations <- CO.tmean.MAM.climate
elevations <- CO.elev
kFit <- LatticeKrig(locations, observations, Z=cbind(elevations))
```

```
## Warning in LatticeKrig(locations, observations, Z = cbind(elevations)): NAs
## removed
```

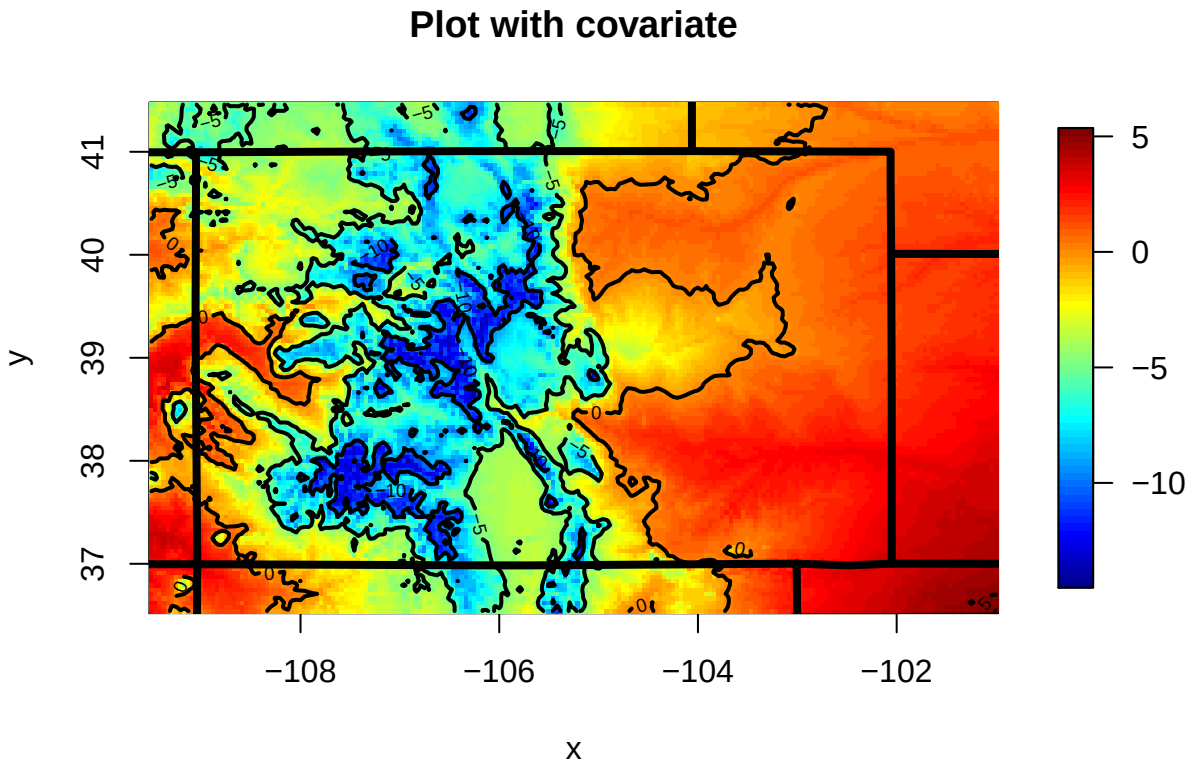
Using the `surface` function will leave out the covariate, resulting in a plot that doesn't match the original data and is smoother than we might expect.

```
surface(kFit, nx = 200, ny = 150, extrap = TRUE, main="Plot missing covariate",
        xlab = "Longitude", ylab="Latitude")
US(add=TRUE, col='black', lwd=4)
```



To fix this, call `surface` on a `predictSurface` object instead of on an `LKrig` object, and make sure to pass in the `grid.list` and `ZGrid` parameters to the `predictSurface` call.

```
prediction <- predictSurface(kFit, grid.list = CO.Grid,
                  ZGrid = CO.elevGrid, nx = 200, ny = 150, extrap = TRUE)
surface(prediction, main="Plot with covariate")
US(add=TRUE, col='black', lwd=4)
```



7.3 Why aren't the settings in my LKrigSetup object being used by the kriging fit?

First, make sure everything is spelled correctly; R variables are case sensitive. For example, `LatticeKrig(x, y, LKInfo = info)` will not work, because the 'i' in "LKinfo" must be lowercase. Next, make sure that every parameter is being set correctly: in particular, don't confuse `x` with `X` or `alpha` with `a.wghts`. Also make sure that parameters that need to be passed as strings are in quotes, e.g. `LKGeometry = "LKSphere", distance.type="GreatCircle"`. If everything is set correctly and spelled correctly, make sure that the list from LKrigSetup is being passed in to your `LatticeKrig` or `LKrig` call.

8 Appendix A: The Linear Algebra of Kriging

Suppose we have a vector $\mathbf{y}$ of observations, where each observation y_i is taken at location $\mathbf{s}_i$, and a covariate matrix Z containing the coordinates of the locations and possibly other related information. Assuming that the observations are a linear combination of the covariates with a Gaussian process of mean 0, we have

$$\mathbf{y} = Z\mathbf{d} + \epsilon$$

where $\epsilon \sim MN(\mathbf{0}, \Sigma)$ for some covariance matrix Σ. We can then make assumptions to determine the form of Σ: Assuming the process is stationary, σ_{ij} will only depend on the vector $\mathbf{s}_i - \mathbf{s}_j$; assuming the process is isotropic, σ_{ij} will only depend on the scalar $||\mathbf{s}_i - \mathbf{s}_j||$, which also means that Σ will be symmetric. This then allows us to establish a covariance function, c, such that $\sigma_{ij} = c(||\mathbf{s}_i - \mathbf{s}_j||)$. The covariance function describes how strongly correlated observations at varying distances are; as such, we would expect that c has a global maximum at 0. We can make further assumptions about the covariance function to make computations easier. In LatticeKrig, we assume the covariance function is a Wendland function, which has compact support on $[0, 1]$. This compact support will lead to a sparse Σ, which makes computing with Σ significantly faster and allows us to compute kriging estimates on very large data sets in a reasonable amount of time. Alternatively, in fixed-rank kriging, it is assumed that $\Sigma = S^T K S$, where K is a matrix of fixed size, independent of the number of observations. This form of Σ also makes computations easier, making it another technique for kriging on large data sets.

In LatticeKrig, we assume that $\epsilon = \Phi\mathbf{c} + \mathbf{e}$, where Φ is a matrix of radial basis functions (so ϕ_{ij} is the j^{th} basis function evaluated at the i^{th} point), and each radial basis function is the same except for a shift in location; $\mathbf{c}$ is the vector of coefficents that scale each basis function; and $\mathbf{e}$ is the vector of measurement errors, distributed $N(0, \sigma^2 I)$. Thus, our total model is $\mathbf{y} = Z\mathbf{d} + \Phi\mathbf{c} + \mathbf{e}$. We can't predict measurement error, so instead we focus on predicting $\mathbf{f} = Z\mathbf{d} + \Phi\mathbf{c}$ at new locations. The matrix of covariates Z and the matrix of basis functions Φ are both determined from the points we choose to predict at: the unknowns we need to estimate are $\mathbf{c}$ and $\mathbf{d}$. We estimate $\mathbf{d}$ by using the generalized least squares estimate: $\mathbf{d} = (Z^T\Sigma^{-1}Z)^{-1}Z^T\Sigma^{-1}$. Estimating $\mathbf{c}$ is more involved. First, we partition Z and $\mathbf{y}$ into two parts: the parts corresponding to the known data, Z_1 and $\mathbf{y}_1$, and the parts corresponding to the data we want to predict, Z_2 and $\mathbf{y}_2$. Since we assume that y follows a Gaussian process, we can write

$$\begin{pmatrix}\mathbf{y}_1\\ \mathbf{y}_2\end{pmatrix} \sim N\left(\begin{pmatrix}\mu_1\\ \mu_2\end{pmatrix}, \begin{bmatrix}\Sigma_{11} & \Sigma_{12}\\ \Sigma_{21} & \Sigma_{22}\end{bmatrix}\right).$$

It is known from multivariate probability theory that

$$E[\mathbf{y}_2|\mathbf{y}_1] = \mu_2 + \Sigma_{21}\Sigma_{11}^{-1}(\mathbf{y}_1 - \mu_1).$$

Where μ_1 and μ_2 are the means of $\mathbf{y}_1$ and $\mathbf{y}_2$, respectively. Since $\epsilon = \Phi\mathbf{c} + \mathbf{e}$ has mean 0, the mean must come from the $Z\mathbf{d}$ term: that is, $\mu_1 = Z_1\mathbf{d}$ and $\mu_2 = Z_2\mathbf{d}$. Since $E[\mathbf{y}_2|\mathbf{y}_1]$ is the best estimator of the values of $\mathbf{y}_2$, we want to find a value of $\mathbf{c}$ that makes our model reproduce this estimator, so we set $E[\mathbf{y}_2|\mathbf{y}_1] = Z_2\mathbf{d} + \Phi_2\mathbf{c}$, where Φ_2 is the matrix of all basis functions evaluated at the points where we're trying to predict y. This gives us the equation

$$Z_2\mathbf{d} + \Phi_2\mathbf{c} = Z_2\mathbf{d} + \Sigma_{21}\Sigma_{11}^{-1}(\mathbf{y}_1 - \mu_1).$$

Now, consider what happens if we make the covariance function and basis function match. Each entry in Σ_{21} is the covariance function of the distance between the j^{th} data point and the i^{th} prediction point, which would be equal to the basis function of the distance between the j^{th} data point and the i^{th} prediction point, which is each entry in Φ_2. This means we can substitute $\Phi_2 = \Sigma_{21}$ into our equation, giving us:

$$\begin{aligned}
Z_2\mathbf{d} + \Phi_2\mathbf{c} &= Z_2\mathbf{d} + \Sigma_{21}\Sigma_{11}^{-1}(\mathbf{y}_1 - \mu_1)\\
\Phi_2\mathbf{c} &= \Sigma_{21}\Sigma_{11}^{-1}(\mathbf{y}_1 - \mu_1)\\
\Phi_2\mathbf{c} &= \Phi_2\Sigma_{11}^{-1}(\mathbf{y}_1 - \mu_1)\\
\mathbf{c} &= \Sigma_{11}^{-1}(\mathbf{y}_1 - \mu_1)
\end{aligned}$$

This gives the best coefficient vector if each basis function is centered at a data point. Since our basis functions are instead centered on a lattice, we need $\hat{\mathbf{c}} = P\Phi^T\mathbf{c}$, where P is the covariance matrix for the centers of the basis functions and Φ is the basis function matrix. Thus, our final estimate for $\mathbf{c}$ is $\hat{\mathbf{c}} = P\Phi^T\Sigma_{11}^{-1}(\mathbf{y} - Z\mathbf{d})$.

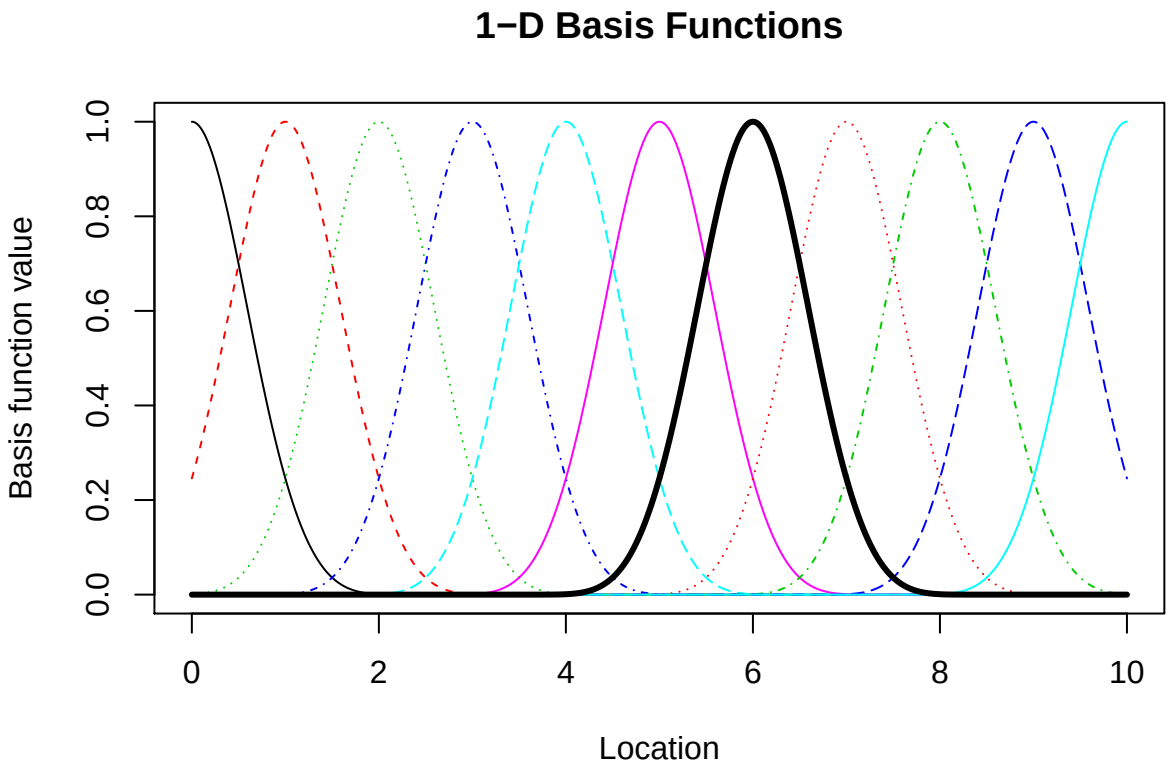
8.1 Sparse Matrix Algorithms

As mentioned earlier, the LatticeKrig package is able to handle large data sets because the covariance function equals 0 for large input. Recall that we make the simplifying assumption that the covariance function is the same as the radial basis function. In LatticeKrig, this function is a Wendland function:

$$\phi(d) = \begin{cases} \frac{1}{3}(1-d)^6(35d^2+18d+3) & 0 \leq d \leq 1 \\ 0 & 1 < d \end{cases}$$

More specifically, a given radial basis function will be 0 at a distance of at least the gap in the lattice multiplied by the parameter `overlap` (which is 2.5 by default). This description is rather opaque, so here is a visualization for the 1-dimensional case.

```
phi <- function(d) {
  return(1/3 * (1-d)^6 * (35*d^2 + 18*d + 3) * (d < 1))
}
overlap <- 2.5
basisCenters <- 0:10
gridPoints <- seq(0, 10, length=1000)
distances <- rdist(gridPoints, basisCenters)
values <- phi(distances / overlap)
matplot(x = gridPoints, values, type="l", xlab="Location",
        ylab = "Basis function value", main="1-D Basis Functions")
lines(values[,7], x=gridPoints, type="l", col="black", lwd=3)
```



We can see that the basis functions all overlap significantly, which is necessary to get a smooth fit. We can see from the highlighted basis function, centered around 6, that the radius of each basis function is 2.5, so the highlighted function is 0 outside of the interval (3.5, 8.5). The graphs appear to reach 0 at a radius of 2

because they go to 0 smoothly, so they don't get far enough from 0 to see the difference near the borders. The basis functions behave similarly in higher dimensions; they are all radially symmetric about their centers.

Since the basis functions and covariance functions are nonzero only on a compact interval, the covariance between many pairs of points will be 0, and equivalently the basis functions will be 0 at many of the points they are evaluated at. This means that the matrices P, Φ, and Σ_{11}^{-1} will all be sparse, which makes the computations much faster. For a further improvement, we can use the Cholesky decomposition of these matrices, which is both triangular and sparse, to speed up calculations even more.

9 Appendix B: Comparison with kriging from fields package

In this section we will compare the kriging done in LatticeKrig with ordinary kriging, such as the kriging done in fields. The chief difference is that LatticeKrig assumes a particular covariance function that leads to a sparse precision matrix (the precision matrix is the inverse of the covariance matrix). However, when we do ordinary kriging with this particular covariance function, we will see that the results come out the same for both algorithms, though the ordinary kriging uses dense matrix operations so it takes much longer with large data. To investigate this, we will use `LKrig` (the function that does the computation in `LatticeKrig`) and `mKrig` to compute models for the data. To make sure the parameters match up, we use an `LKinfo` object to store the parameters for the kriging.

```
data(ozone2)
x <- ozone2$lon.lat
y <- ozone2$y[16,]
# Find location that are not 'NA'.
# (LKrig is not set up to handle missing observations.)
good <- !is.na(y)
x<- x[good,]
y<- y[good]
lambda <-  1.5
# The covariance "parameters" are all in the list LKinfo
# to create this special list outside of a call to LKrig use
testInfo <- LKrigSetup(x, NC=16, nlevel=1, alpha=1.0, a.wght=5)
obj1 <- LKrig(x, y, lambda=lambda, iseed=122, LKinfo = testInfo)

# this call to mKrig should be identical to the LKrig results
# because it uses the LKrig.cov covariance with all the right parameters.
obj2 <- mKrig(x, y, lambda=lambda, m=2, cov.function="LKrig.cov",
              cov.args=list( LKinfo=testInfo), iseed=122)
# compare the two results this is also an
# example of how tests are automated in fields
# set flag to have tests print results
test.for.zero.flag<- TRUE
test.for.zero(obj1$fitted.values, obj2$fitted.values,
               tag="comparing predicted values LKrig and mKrig")
```

```
## Testing:  comparing predicted values LKrig and mKrig
## PASSED test at tolerance  1e-08
```

```
# compare standard errors.
se1 <- predictSE.LKrig(obj1)
se2 <- predictSE.mKrig(obj2)
test.for.zero(se1, se2, tag="comparing standard errors for LKrig and mKrig")
```

```
## Testing:  comparing standard errors for LKrig and mKrig
## PASSED test at tolerance  1e-08
```

As we can see, these two kriging fits produce identical predicted values and standard errors, as we would expect. To make `mKrig` use the same covariance function as `LKrig`, we set the parameter `cov.function="LKrig.cov"`. The `LKrig.cov` function computes the covariance between the lattice points and the data points in such a way that the precision matrix will be sparse and have a certain form. We can then produce the precision matrix directly instead of inverting the covariance matrix, which is one of the reasons that `LKrig` is much faster than `mKrig`.

10 Appendix C: Sample `LatticeKrig` calculation

The computations inside of `LatticeKrig` and `LKrig` can be hard to understand, so here we will work through several examples, showing all of the linear algebra used. Some of the variable names will be changed from the code in the previous section so that they match the names in the linear algebra appendix and in the paper.

10.1 First Example: One level, no normalization

First, we create the data, create the basis/covariance function `basis`, and call `LKrig` to fit the data.

```
library(LatticeKrig)
#clear x and y to make sure our data doesn't get overwritten
rm(x, y)
lambda = 0.05
overlap = 2.5
basis <- function(d) {
  return(1/3 * (1-d)^6 * (35*d^2 + 18*d + 3) * (d < 1))
}
data(KrigingExampleData)
info <- LKrigSetup(as.matrix(c(0,1)), NC = 6, NC.buffer = 5, nlevel = 1, a.wght = 2.01, alpha = 1, lambd
krigFit <- LKrig(x, y, LKinfo = info)
```

Next, we create the lattice; since we only have 1 level in 1 dimension, this is relatively easy.

```
nc <- 6
ncBuffer <- 5

#finding the spacing for the lattice
LGap <- 1/(nc-1)
latInside <- seq(from=0, to=1, by=LGap)

#adding the buffer lattice points outside the interval
latBefore <- seq(to=0-LGap, by=LGap, length.out = ncBuffer)
latAfter <- seq(from=1+LGap, by=LGap, length.out = ncBuffer)
lattice <- c(latBefore, latInside, latAfter)
s <- length(lattice)
```

Now we create the covariance matrix for $\mathbf{y}$, which is M_λ, and the covariance matrix for the basis functions, which is P.

```
Phi <- basis(rdist(x, lattice) / (overlap*LGap))
B <- LKDiag(c(-1, 2.01, -1), s)
Q <- t(B) %*% B
P <- solve(Q)
M <- Phi %*% P %*% t(Phi) + lambda*diag(1, length(x))
Minverse <- solve(M)
```

Finally, we can calculate our estimates for $\mathbf{c}$ and $\mathbf{d}$: `cHat` and `dHat`, respectively.

```
ones <- rep(1, length(x))
Z <- cbind(ones, x)
dHat <- solve(t(Z) %*% Minverse %*% Z, t(Z) %*% Minverse %*% y)
G <- t(Phi)  %*% Phi + lambda*Q
cHat <- solve(G) %*% t(Phi) %*% (y - Z %*% dHat)

#making covariance matrix and comparing it to the LKrig one
testCov <- Phi %*% P %*% t(Phi)
```

```
targetCov <- LKrig.cov(x, x, info)
test.for.zero(testCov, targetCov)
```

```
## PASSED test at tolerance  1e-08
```

```
test.for.zero(dHat, krigFit$d.coef)
```

```
## PASSED test at tolerance  1e-08
```

```
test.for.zero(cHat, krigFit$c.coef)
```

```
## PASSED test at tolerance  1e-08
```

```
#compare kriging prediction with calculated prediction
xGrid <- seq(0,1,length = 200)
krigPredictions <- predict(krigFit, xGrid)
PhiPredict <- basis(rdist(xGrid, lattice) / (overlap*LGap))
ZPredict <- cbind(rep(1, length(x)), xGrid)
predictions <- ZPredict %*% dHat + PhiPredict %*% cHat
test.for.zero(krigPredictions, predictions)
```

```
## PASSED test at tolerance  1e-08
```

10.2 Second Example: Three levels, no normalization

The setup in this example is almost the same as in the first one; the only differences are the different random seed and the different values of `nlevel` and `alpha` in the `LKinfo` object. The value of `alpha` is chosen so that each level has half as much weight as the previous and the sum of all the weights is 1.

```
lambda <- 0.05
overlap <- 2.5
basis <- function(d) {
  return(1/3 * (1-d)^6 * (35*d^2 + 18*d + 3) * (d < 1))
}
data(KrigingExampleData)
info <- LKrigSetup(as.matrix(c(0,1)), NC = 6, nlevel = 3, a.wght = 2.01, alpha = c(4,2,1)/7, lambda = 0
krigFit <- LKrig(x, y, LKinfo = info)
```

Making the lattice is now more complicated, since we need to create three different levels. However, note that the first level is the same as before, and the new levels just have lattice points 2x and 4x closer together.

```
nc <- 6
ncBuffer <- 5
LGap <- 1 / (nc-1)
L1Inside <- seq(from=0, to=1, by=LGap)
L1Before <- seq(to=0-LGap, by=LGap, length.out = ncBuffer)
L1After <- seq(from=1+LGap, by=LGap, length.out = ncBuffer)
L1 <- c(L1Before, L1Inside, L1After)

L2Inside <- seq(from=0, to=1, by=LGap/2)
L2Before <- seq(to=0-LGap/2, by=LGap/2, length.out = ncBuffer)
L2After <- seq(from=1+LGap/2, by=LGap/2, length.out = ncBuffer)
L2 <- c(L2Before, L2Inside, L2After)

L3Inside <- seq(from=0, to=1, by=LGap/4)
L3Before <- seq(to=0-LGap/4, by=LGap/4, length.out = ncBuffer)
L3After <- seq(from=1+LGap/4, by=LGap/4, length.out = ncBuffer)
```

```
L3 <- c(L3Before, L3Inside, L3After)

s1 <- length(L1)
s2 <- length(L2)
s3 <- length(L3)
c(s1, s2, s3)
```

```
## [1] 16 21 31
```

Note that the values of `s1, s2, s3` don't follow a strict 1:2 ratio as we might expect; this is because of the lattice points outside the region, and because of the boundaries. Specifically, `s1 = 16` because there are `nc = 6` lattice points covering the interval, with 5 gaps between them, and an additional 5 lattice points on each side of the interval. At the second level, the gaps are half as wide, so the 5 gaps become 10; there are now 11 lattice points in the interval and 5 on each side, giving the total `s2 = 21`. Similarly, at the third level the 10 gaps become 20, making 21 lattice points in the interval and 5 on either side, so we have `s3 = 31`.

Now we create the covariance matrix for $\mathbf{y}$, which is M_λ, and the covariance matrix for the basis functions, which is P. Now that we have 3 different lattice sizes, making $Q = P^{-1}$ becomes more difficult, since it's a block-diagonal matrix with a block entry for each different lattice size.

```
alpha <- c(4, 2, 1)/7
Phi1 <- basis(rdist(x, L1) / (overlap*LGap)) * sqrt(alpha[1])
Phi2 <- basis(rdist(x, L2) / (overlap*LGap/2)) * sqrt(alpha[2])
Phi3 <- basis(rdist(x, L3) / (overlap*LGap/4)) * sqrt(alpha[3])
Phi <- cbind(Phi1, Phi2, Phi3)
targetBasis <- spam2full(LKrig.basis(x, info))
test.for.zero(targetBasis, Phi)
```

```
## PASSED test at tolerance  1e-08
```

```
B1 <- LKDiag(c(-1, 2.01, -1), s1)
B2 <- LKDiag(c(-1, 2.01, -1), s2)
B3 <- LKDiag(c(-1, 2.01, -1), s3)
Q1 <- t(B1) %*% B1
Q2 <- t(B2) %*% B2
Q3 <- t(B3) %*% B3
Q <- matrix(0, nrow = s1+s2+s3, ncol = s1+s2+s3)

#putting Q1, Q2, Q3 into block-diagonal matrix Q
Q[1:s1, 1:s1] <- Q1
Q[(s1+1):(s1+s2), (s1+1):(s1+s2)] <- Q2
Q[(s1+s2+1):(s1+s2+s3), (s1+s2+1):(s1+s2+s3)] <- Q3
P <- solve(Q)
M <- Phi %*% P %*% t(Phi) + lambda*diag(1, length(x))
Minverse <- solve(M)
```

Finding coefficients

```
ones <- rep(1, length(x))
Z <- cbind(ones, x)
dHat <- solve(t(Z) %*% Minverse %*% Z, t(Z) %*% Minverse %*% y)
G <- t(Phi)  %*% Phi + lambda*Q
cHat <- solve(G) %*% t(Phi) %*% (y - Z %*% dHat)

#making covariance matrix and comparing it to the LKrig one
testCov <- Phi %*% P %*% t(Phi)
targetCov <- LKrig.cov(x, x, info)
```

```
test.for.zero(testCov, targetCov)
```

```
## PASSED test at tolerance  1e-08
```

```
test.for.zero(dHat, krigFit$d.coef)
```

```
## PASSED test at tolerance  1e-08
```

```
test.for.zero(cHat, krigFit$c.coef)
```

```
## PASSED test at tolerance  1e-08
```

10.3 Third example: One level with normalization

The bulk of this example is identical to the first one: the only differences appear after we have the basis function matrix, `Phi`. All the code in the following block is identical to the code from the first example (except that the `LKinfo` object no longer sets `normalize=FALSE`).

```
rm(x, y)
```

```
## Warning in rm(x, y): object 'x' not found
```

```
## Warning in rm(x, y): object 'y' not found
```

```
lambda = 0.05
overlap = 2.5
basis <- function(d) {
  return(1/3 * (1-d)^6 * (35*d^2 + 18*d + 3) * (d < 1))
}
data(KrigingExampleData)
info <- LKrigSetup(as.matrix(c(0,1)), NC = 6, NC.buffer = 5, nlevel = 1, a.wght = 2.01, alpha = 1, lambd
krigFit <- LKrig(x, y, LKinfo = info)

nc <- 6
ncBuffer <- 5

#finding the spacing for the lattice
LGap <- 1/(nc-1)
latInside <- seq(from=0, to=1, by=LGap)

#adding the buffer lattice points outside the interval
latBefore <- seq(to=0-LGap, by=LGap, length.out = ncBuffer)
latAfter <- seq(from=1+LGap, by=LGap, length.out = ncBuffer)
lattice <- c(latBefore, latInside, latAfter)
s <- length(lattice)
Phi <- basis(rdist(x, lattice) / (overlap*LGap))
B <- LKDiag(c(-1, 2.01, -1), s)
Q <- t(B) %*% B
P <- solve(Q)
Phi[5:10, 5:10]
```

```
##              [,1]      [,2]      [,3]       [,4]         [,5] [,6]
## [1,] 0.0826412739 0.8627041 0.5185841 0.01268147 0.000000e+00    0
## [2,] 0.0528101862 0.7741131 0.6209304 0.02364147 0.000000e+00    0
## [3,] 0.0472068618 0.7518035 0.6448217 0.02699678 0.000000e+00    0
## [4,] 0.0044214072 0.3827491 0.9566797 0.14276509 7.523868e-05    0
## [5,] 0.0017342916 0.2957705 0.9931238 0.20103119 3.711216e-04    0
```

```
## [6,] 0.0009251796 0.2511137 0.9999117 0.24039814 7.780977e-04    0
```

Now we normalize `Phi` so that the basis functions have covariance 1 at each data point. We also print out the diagonal of the covariance matrix, $\Phi P \Phi^T$ - note that it is all ones.

```
D <- Phi %*% P %*% t(Phi)
#discarding the off-diagonal elements of D
DS <- diag(diag(D)^(-1/2))
Phi <- DS %*% Phi
diag(Phi %*% P %*% t(Phi))
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

The rest of the calculations proceed in the same way as without normalization.

```
M <- Phi %*% P %*% t(Phi) + lambda*diag(1, length(x))
Minverse <- solve(M)

ones <- rep(1, length(x))
Z <- cbind(ones, x)
dHat <- solve(t(Z) %*% Minverse %*% Z, t(Z) %*% Minverse %*% y)
G <- t(Phi)  %*% Phi + lambda*Q
cHat <- solve(G) %*% t(Phi) %*% (y - Z %*% dHat)

#making covariance matrix and comparing it to the LKrig one
testCov <- Phi %*% P %*% t(Phi)
targetCov <- LKrig.cov(x, x, info)
test.for.zero(testCov, targetCov)
```

```
## PASSED test at tolerance  1e-08
```

```
test.for.zero(dHat, krigFit$d.coef)
```

```
## PASSED test at tolerance  1e-08
```

```
test.for.zero(cHat, krigFit$c.coef)
```

```
## PASSED test at tolerance  1e-08
```

```
#compare kriging prediction with calculated prediction
xGrid <- seq(0,1,length = 200)
krigPredictions <- predict(krigFit, xGrid)
PhiPredict <- basis(rdist(xGrid, lattice) / (overlap*LGap))
#normalizing PhiPredict too
DPredict <- PhiPredict %*% P %*% t(PhiPredict)
#discarding the off-diagonal elements of D
DPredictS <- diag(diag(DPredict)^(-1/2))
PhiPredict <- DPredictS %*% PhiPredict

ZPredict <- cbind(rep(1, length(x)), xGrid)
predictions <- ZPredict %*% dHat + PhiPredict %*% cHat
test.for.zero(krigPredictions, predictions)
```

```
## PASSED test at tolerance  1e-08
```

10.4 Using the kriging equations directly

Recall the equations for $\hat{\mathbf{c}}$ and $\hat{bldd}$ in Appendix A:

$$\hat{\mathbf{d}} = (Z^T \Sigma^{-1} Z)^{-1} Z^T \Sigma^{-1}$$
$$\hat{\mathbf{c}} = P \Phi^T \Sigma_{11}^{-1} (\mathbf{y} - Z\mathbf{d})$$

In this appendix, we will compute the estimates directly using these equations to show that the computations in LatticeKrig match them.