LatticeKrig Vignette

Matthew Iverson

5/21/2019

Contents

1 Introduction **2**
1.1 What is Kriging? . . . 2
1.2 The LatticeKrig Model . . . 2
1.3 Glossary of Important Functions . . . 3

2 Quick Start Guide **4**

3 `LatticeKrig` Optional Arguments **8**
3.1 `Z` . . . 8
3.2 `nlevel` . . . 8
3.3 `alpha` . . . 8
3.4 `LKinfo` . . . 8

4 Kriging in Different Geometries **9**
4.1 Working with spherical coordinates . . . 9

5 Common Errors **16**
5.1 Could not find function [FunctionName] . . . 16
5.2 Need to specify NC for grid size . . . 16
5.3 Invalid 'times' argument . . . 16
5.4 Only one alpha specifed for multiple levels . . . 16
5.5 Missing value where TRUE/FALSE needed . . . 16

6 Frequently Asked Questions **17**
6.1 Does the order the parameters are listed in matter? . . . 17
6.2 The predicted values from my Kriging fit are nowhere near the data; what's wrong? . . . 17
6.3 Why aren't the settings in my LKrigSetup object aren't being used by the kriging fit? . . . 19

7 Appendix A: The Linear Algebra of Kriging **20**
7.1 Sparse Matrix Algorithms . . . 21

8 Appendix B: Comparison with kriging from fields package **23**

1 Introduction

In this vignette, we will briefly explain what kriging is, explore the functions in the LatticeKrig package, and show examples of how they can be used to solve problems.

1.1 What is Kriging?

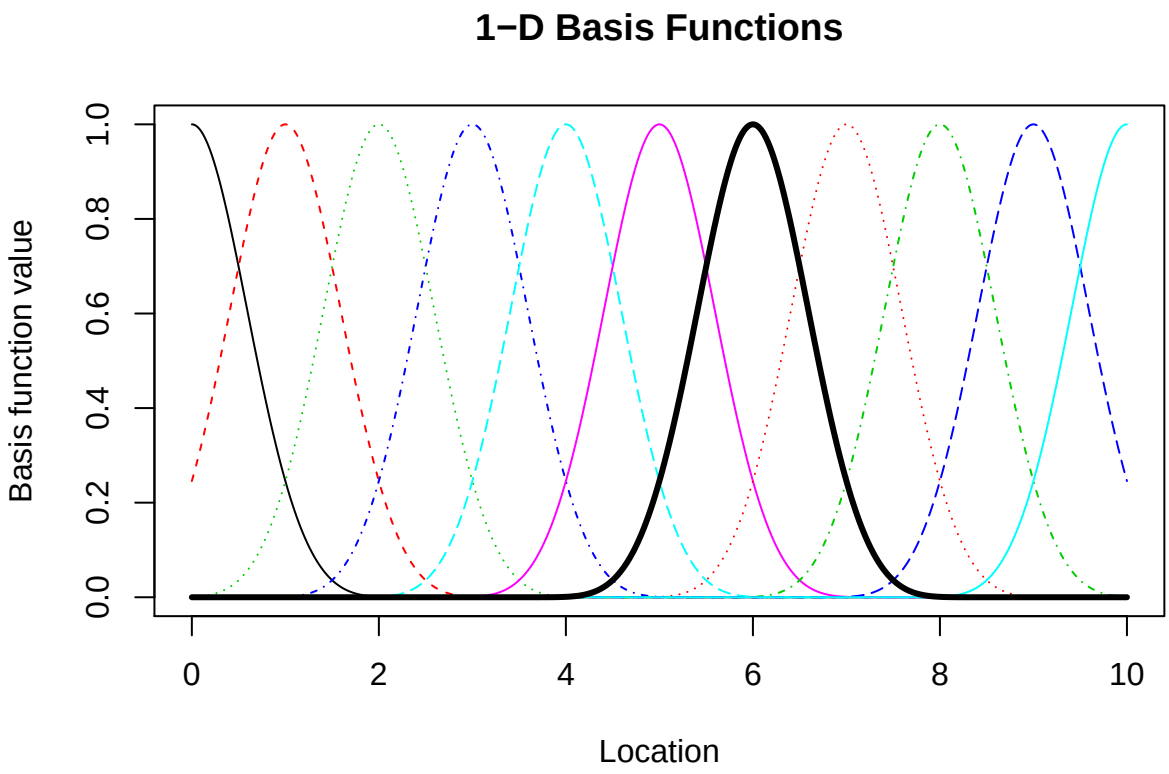
Kriging (named for South African satistician Danie Krige) is a method for making predictions from a data set. It is designed to be used on spatial data – that is, our data contains the observed variable and the location it was observed at, and pairs of observations taken close together have similar values. As such, it can be applied to a variety of physical data sets, from geological data to atmospheric data.

The goal of kriging is to create a model, based on data from some locations, that can accurately predict the observed variable at any location inside the given data.

1.2 The LatticeKrig Model

In the LatticeKrig package, we model the given data as the sum of a linear polynomial in the locations (and covariates, if provided) and a spatial process with mean 0. 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 a collection of radial 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.

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 effect sizes; 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.



1.3 Glossary of Important Functions

- `LatticeKrig`: Calls LKrig, passing in default values and estimates for the needed parameters.
- `LKrig`: Fits a kriging estimate to the given data.
- `LKrigSetup`: Creates an object to store the parameters to use for a `LatticeKrig` / `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

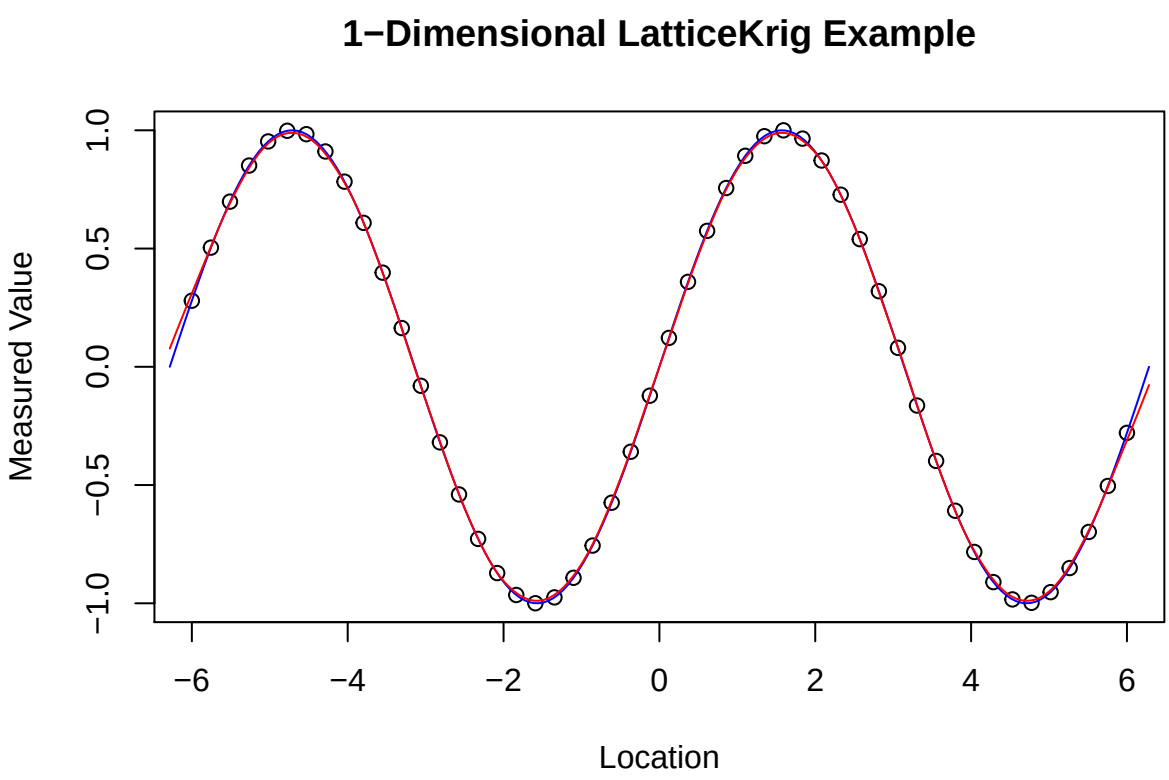
In this section, we will lay out the bare essentials of the package to make the central features as easily accessible as possible. To fit a surface and interpolate data using `LatticeKrig`, the required arguments are the measurement locations (formatted in a matrix where each row is one location) and measurement values. 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 equally spaced points on the interval $[-6, 6]$, and our variable measurements to be the values of $\sin(x)$ at these locations.

```
locations <- seq(-6,6, len=50)
observations <- sin(locations)
kFit <- LatticeKrig(locations, observations)
kFit
```

```
## Call:
## LatticeKrig(x = locations, y = observations)
##
##
##  Number of Observations:                      50
##  Number of parameters in the fixed component 2
##   Effective degrees of freedom (EDF)         12.5
##     Standard Error of EDF estimate:          0.741
##  MLE sigma                                   0.06153
##  MLE rho                                     30.62
##  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     2.0
##      2             23     1.0
##      3             35     0.5
##
## Nonzero entries in Ridge regression matrix 814
```

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)
plot(locations, observations, main="1-Dimensional LatticeKrig Example",
     xlab="Location", ylab="Measured Value")
lines(xGrid, sin(xGrid), col='blue')
lines(xGrid, predict(kFit, xGrid), col='red')
```

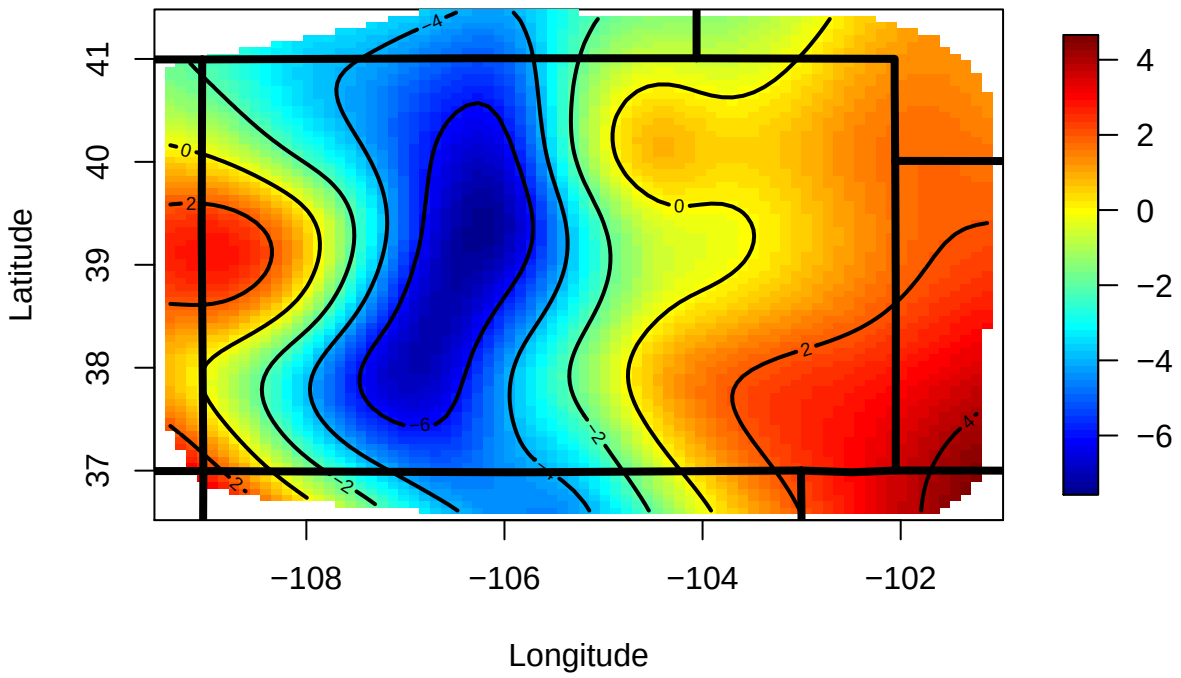


We can see that `LatticeKrig` takes in the data points (shown above in black) and produces a prediction over the whole interval (in red) that matches the true function (in blue) rather closely. For another, more practical example, we will predict the average spring temperature for locations throughout 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)
```

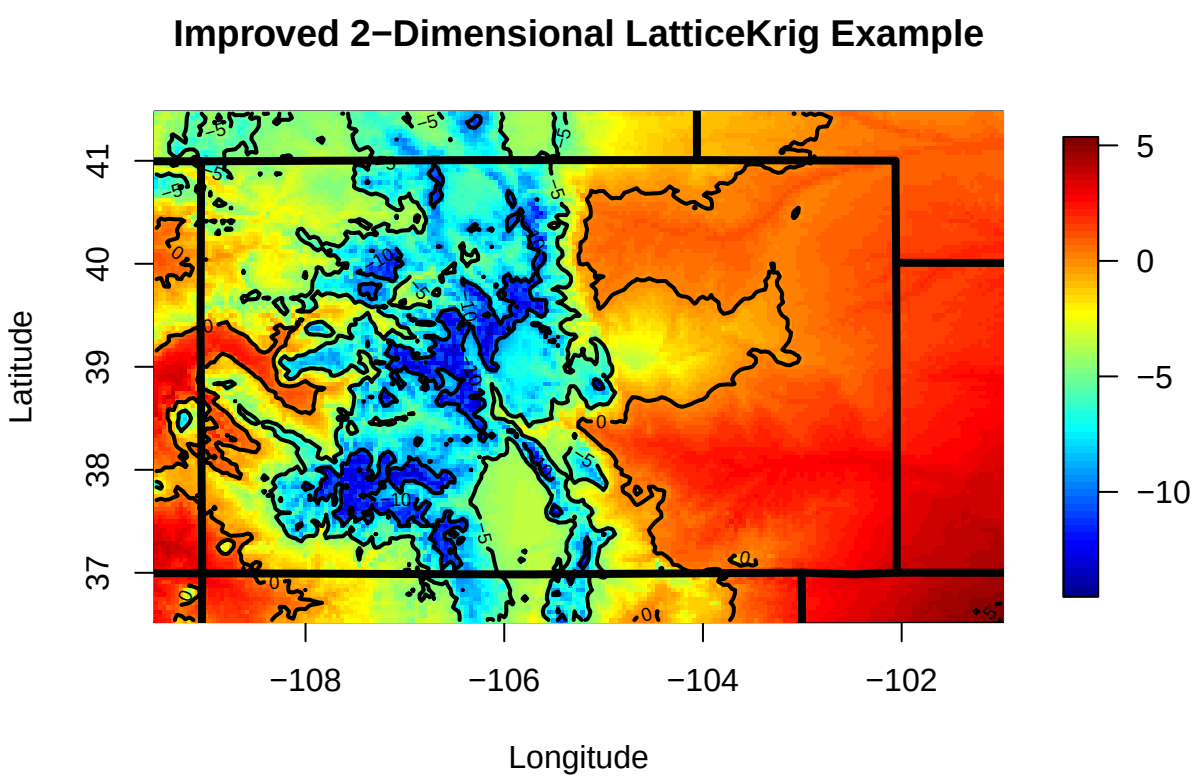


This plot is nice, but we can do better. We can see that the coldest temperatures are in the Rocky Mountains, which is unsurprising. Thus, we might expect that we will get a more accurate fit by having `LatticeKrig` account for the elevation at each location as well. Another way we can improve the plot is by increasing its resolution - the current plot is somewhat pixelated. We can tell the `surface` function to evaluate the surface at more points by using the `nx` and `ny` arguments, which will take longer to compute but produces a nicer looking, more detailed plot. Finally, we can also have `surface` extend the computation 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 `LatticeKrig` fitting method isn't designed to extrapolate and so the expected error increases dramatically when predicting outside of the given data. However, extending the plot to the corners will make it look nicer.

```
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
```

```
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")
US(add=TRUE, col='black', lwd=4)
```


Improved 2−Dimensional LatticeKrig Example
Latitude
41
40
39
38
37
−108
−106
−104
−102
Longitude
5
0
−5
−10

3 `LatticeKrig` Optional Arguments

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.

3.1 `Z`

The optional parameter Z is a matrix of covariates (variables aside from location) to include in the model. Each column of this matrix must contain the value of one of the covariates at each data location, so the number of rows in Z must match the number of rows in the required parameter x, the set of locations. For an example, see the last two plots in the Quick Start section.

3.2 `nlevel`

The optional parameter `nlevel` is an integer that determines the number of different lattice sizes to compute the basis function coefficients on. Each different lattice size is computed independently, and the resulting layers are each scaled by the values in the parameter `alpha` (provided in the LKInfo or automatically computed) the before being added together.

3.3 `alpha`

The optional parameter `alpha` is a vector of length `nlevel` 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.

3.4 `LKinfo`

The optional parameter `LKinfo` holds many other parameter values, expanded on in the next section.

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

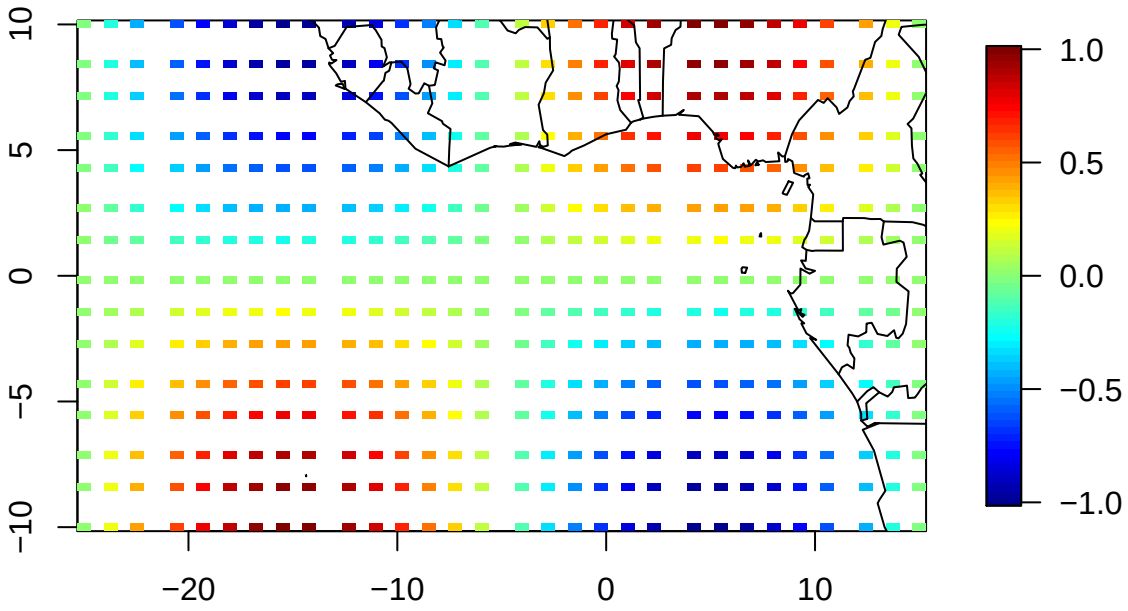
By default, `LKinfo` will use either `LKInterval`, `LKRectangle`, or `LKBox`, depending on the number of dimensions in the given location data. When using the `LKSphere` geometry, there are also different ways of measuring distance, which you can set using the `distance.type` parameter of the `LKinfo` 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.

4.1 Working with spherical coordinates

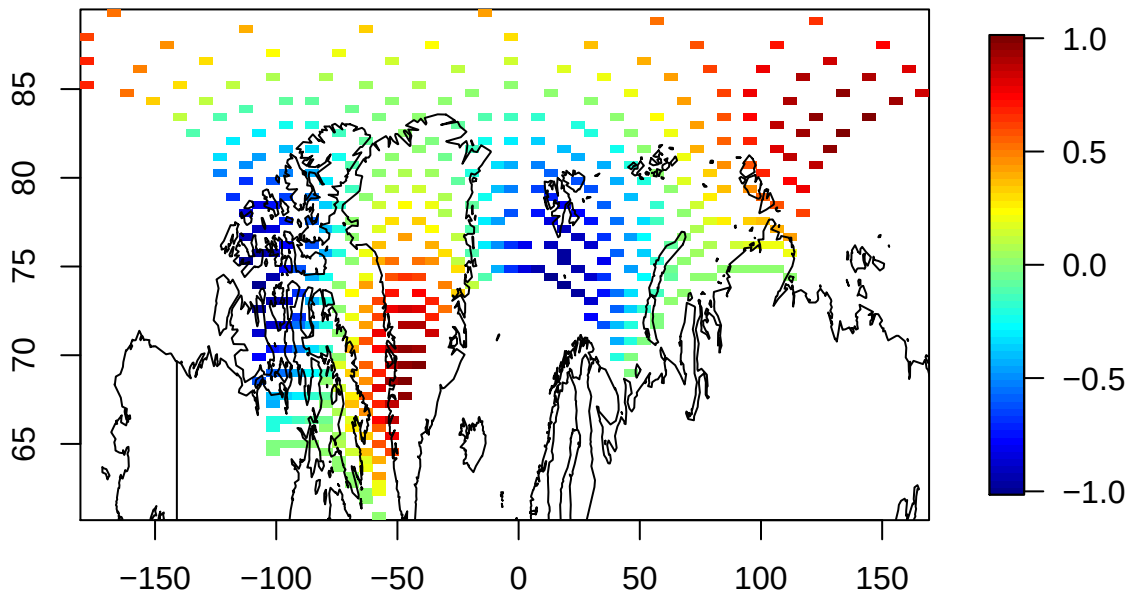
For an example of fitting data taken on the globe using spherical geometry instead of rectangular, we will create some sample data at the equator, rotate it up to near the north pole, and compare the models computed on the `LKRectangle` geometry and `LKSphere` geometry. We compute a kriging fit for the 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.

```
length = 30
gridAngle = 40
grid <- list(x = seq(-gridAngle/2, gridAngle/2, len=length)-5,
             y = seq(-gridAngle/4, gridAngle/4, len=length/2));
dataLocations <- make.surface.grid(grid)
dataValues <- sin(2*pi/gridAngle * (5+dataLocations[,1])) *sin(2*pi/gridAngle * 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)
world(add=TRUE, col="black", lwd=1)
```

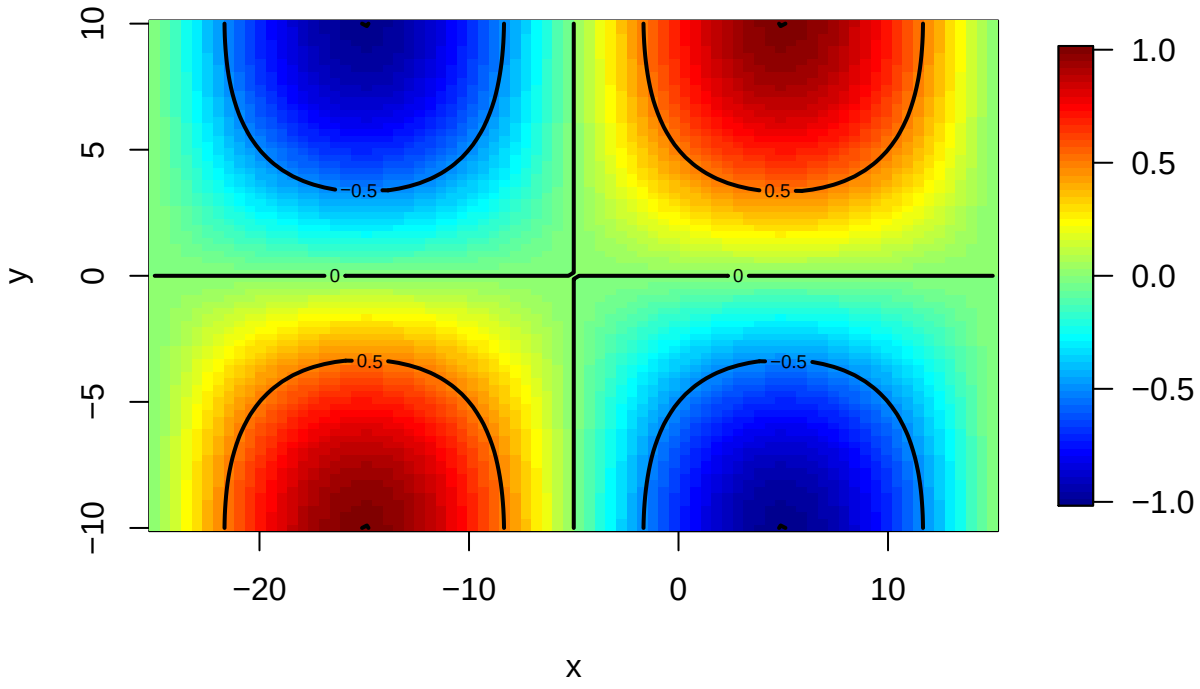



```
quilt.plot(newLocations, dataValues)
world(add=TRUE, col="black", lwd=1)
```



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

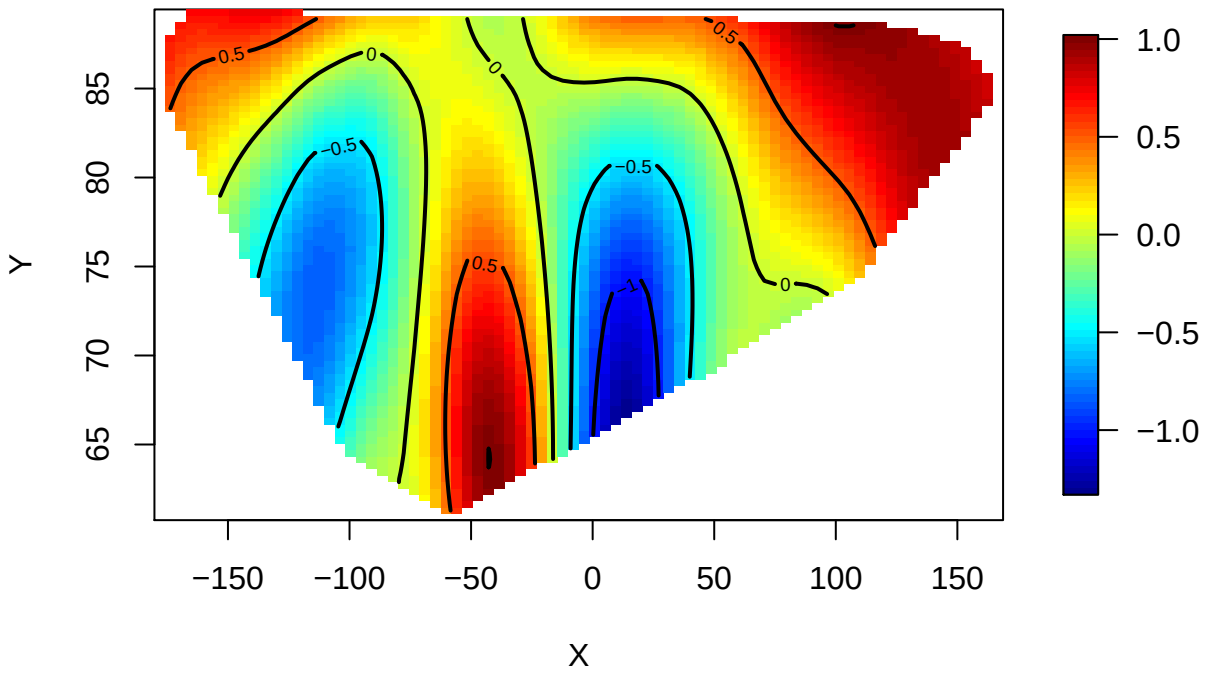
```
kFit1 <- LatticeKrig(dataLocations, dataValues)
surface(kFit1, main="Equator Surface Prediction Using Rectangular Kriging")
```




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

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

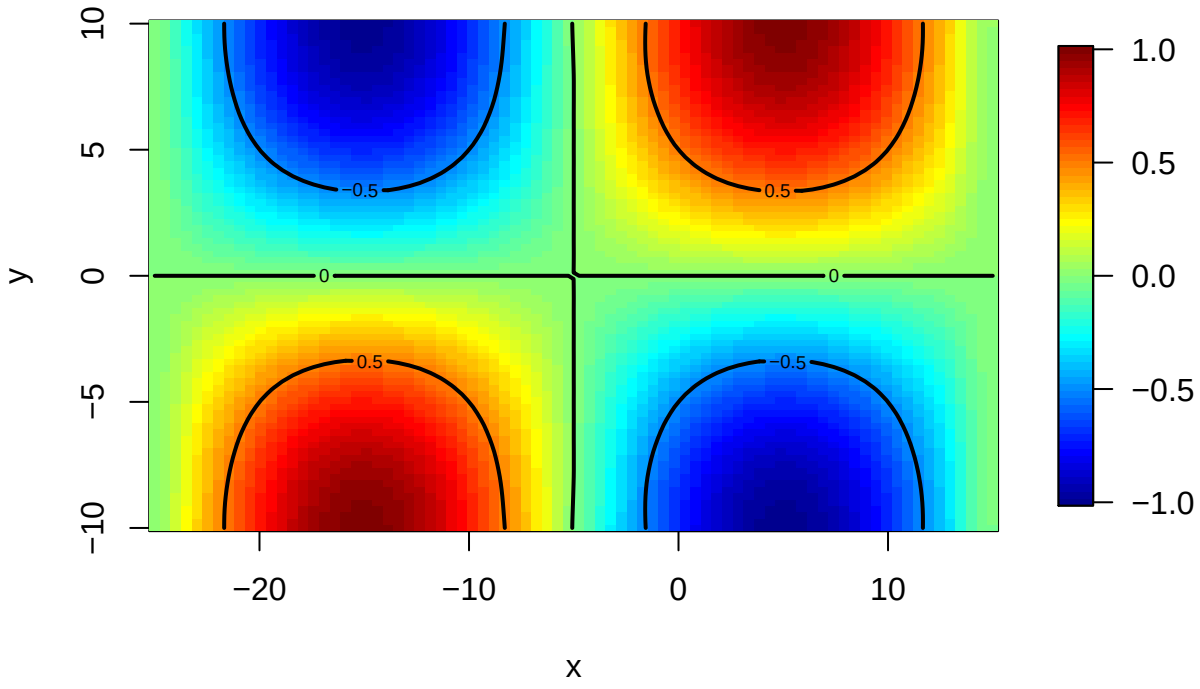
```
kFit2 <- LatticeKrig(newLocations, dataValues)
surface(kFit2, main="Polar Surface Prediction Using Rectangular Kriging")
```




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

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

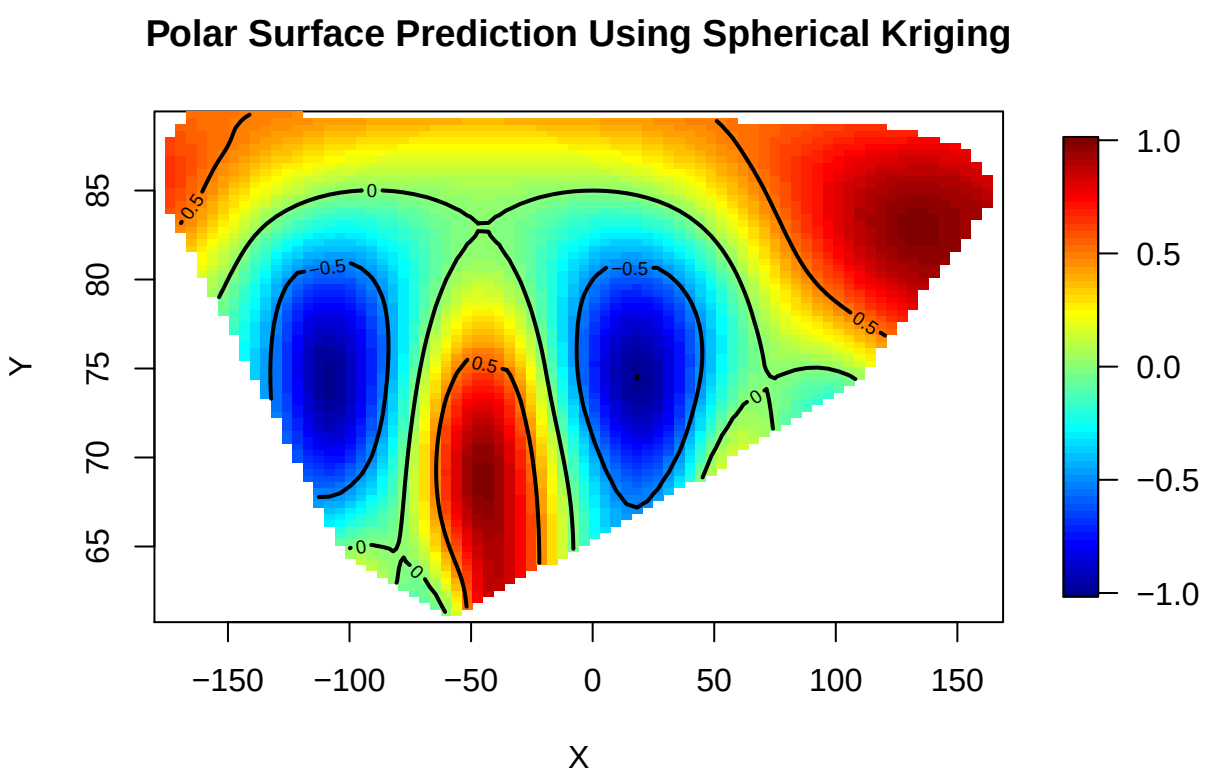
```
info1 <- LKrigSetup(dataLocations, nlevel = 2, startingLevel = 6, alpha = c(0.8, 0.2)
                    , a.wght = 1.01, LKGeometry = "LKSphere")
kFit1 <- LatticeKrig(dataLocations, dataValues, LKinfo = info1)
surface(kFit1, main="Equator Surface Prediction Using Spherical Kriging")
```




```
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")
```




```
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 Common Errors

5.1 Could not find function [FunctionName]

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

5.2 Need to specify NC for grid size

5.3 Invalid 'times' argument

5.4 Only one alpha specifed for multiple levels

5.5 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 Frequently Asked Questions

6.1 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.

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

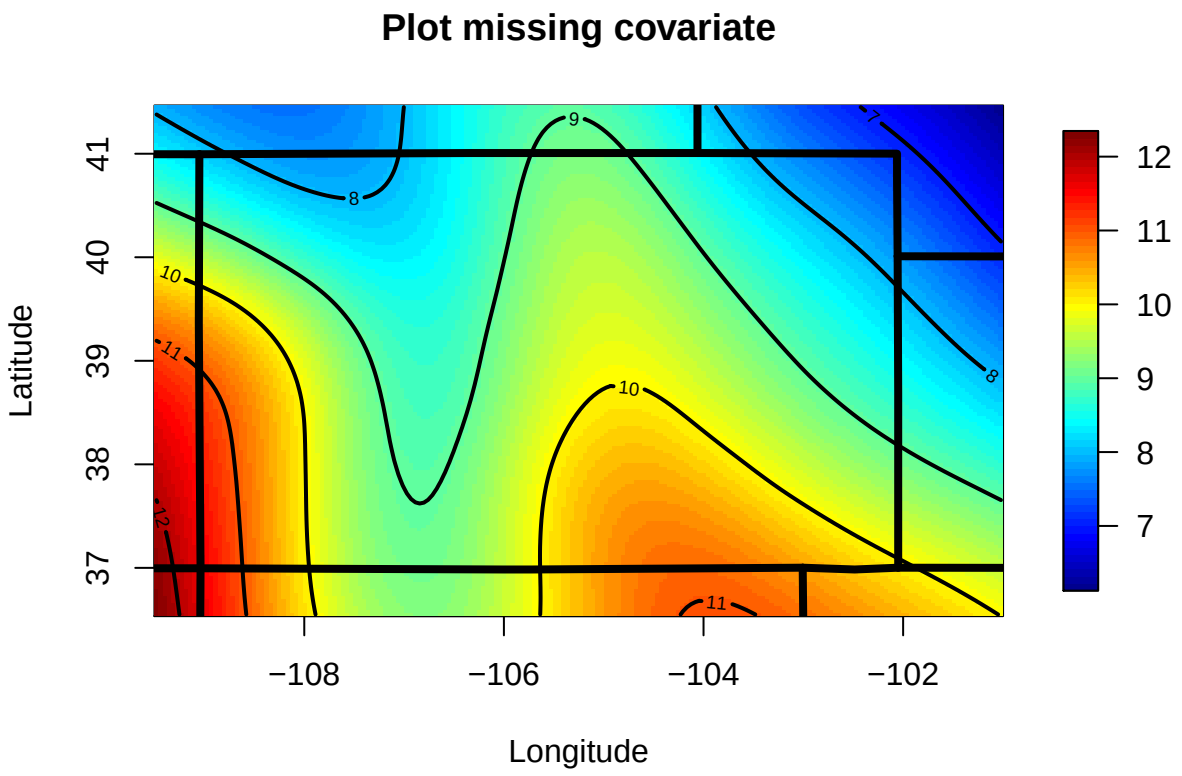
If your model includes covariates (the `Z` parameter of `LatticeKrig` and `LKrig`), your plot may not have included the effect of the covariate. The following code demonstrates this issue using the Colorado temperature data and how to fix it; first, we will set up the model.

```
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
```

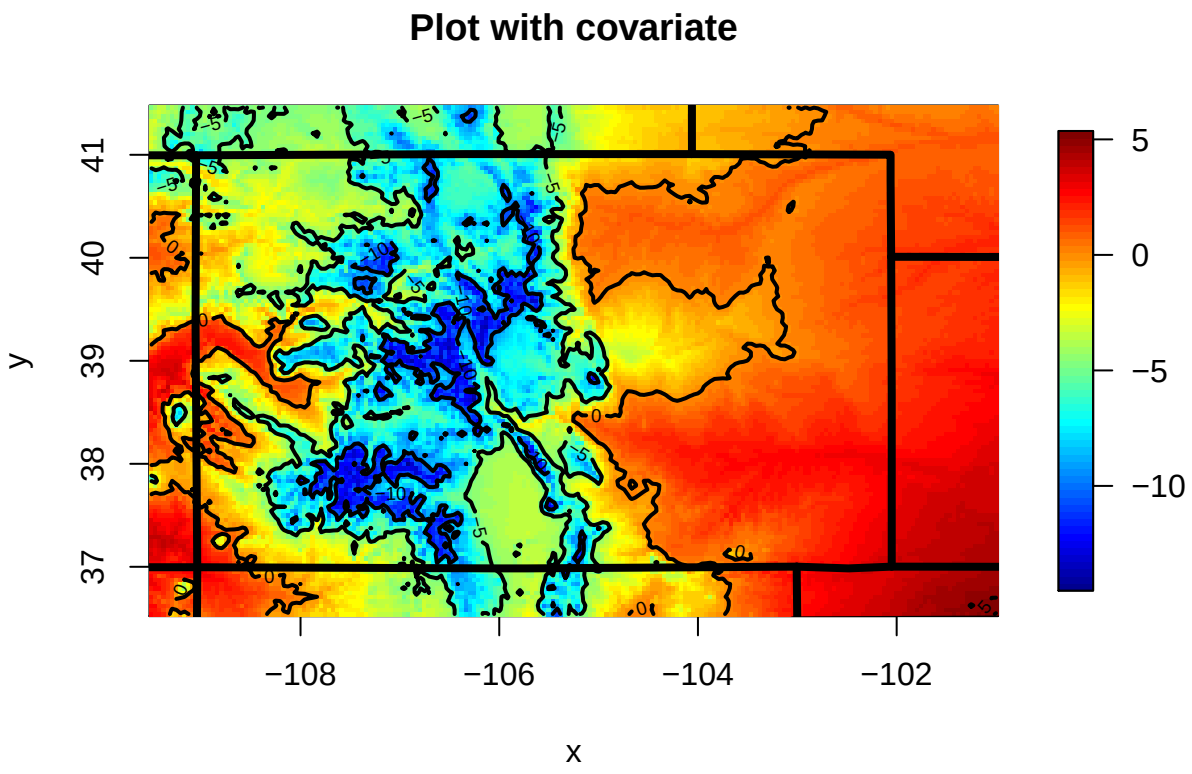
Using the `surface` function will leave out the covariate, resulting in a plot that doesn't match the original data and is smoother than we might expect.

```
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)
```



6.3 Why aren't the settings in my LKrigSetup object aren't 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.

7 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 each basis function is weighted by; 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 $Z\mathbf{d} + \Phi\mathbf{c}$ 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 $\hat{\mathbf{c}} = P\Phi^T\Sigma_{11}^{-1}(\mathbf{y} - Z\mathbf{d})$.

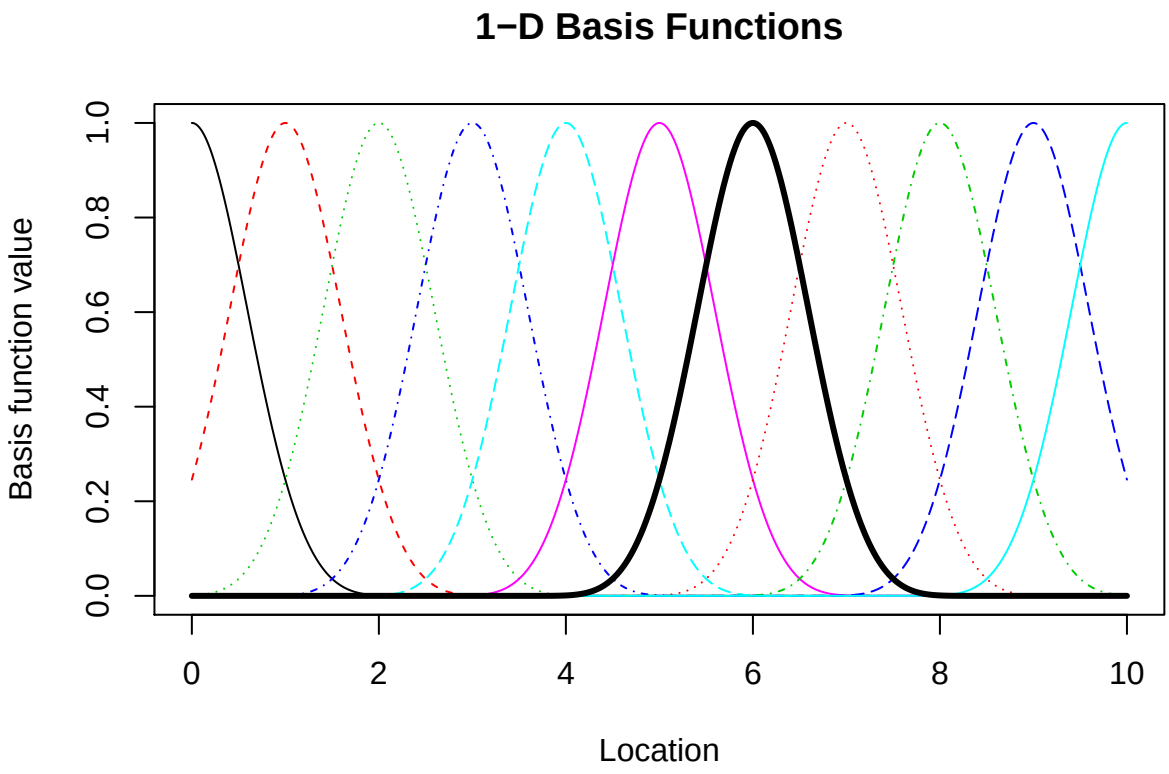
7.1 Sparse Matrix Algorithms

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:

$$\phi(d) = \frac{1}{3}(1-d)^6(35d^2 + 18d + 3)$$

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.

8 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`.