

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	5
2.1	Fitting the model in one dimension	5
2.2	Plotting the results	6
3	LKrigSetup	10
3.1	Required Parameters for LKrigSetup	10
3.2	Optional parameters	11
3.3	Effects on covariance function	12
4	Kriging in Different Geometries	14
4.1	Working with spherical coordinates	14
5	Using Sparse Matrices	17
6	The difference between LatticeKrig and LKrig	18
7	Common Errors	19
7.1	Could not find function [FunctionName]	19
7.2	Need to specify NC for grid size	19
7.3	Invalid ‘times’ argument	19
7.4	Only one alpha specified for multiple levels	19
7.5	Missing value where TRUE/FALSE needed	19
7.6	Error in object\$x %*% t(Vinv) : non-conformable arguments	19
7.7	Error in if (ncol(object\$x) != 1) { : argument is of length zero	19
8	Frequently Asked Questions	20
8.1	Does the order of the parameters matter?	20
8.2	The predicted values from my Kriging fit are nowhere near the data; what’s wrong?	20
8.3	Why aren’t the settings in my LKrigSetup object being used by the kriging fit?	21
9	Appendix A: The Linear Algebra of Kriging	22
9.1	Sparse Matrix Algorithms	23
10	Appendix B: Comparison with kriging from fields package	25
11	Appendix C: Sample LatticeKrig calculation	26
11.1	First Example: One level, no normalization	26
11.2	Second example: One level with normalization	27
11.3	Third Example: Three levels, no normalization	28
11.4	Using the kriging equations directly	30

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 statistician 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.

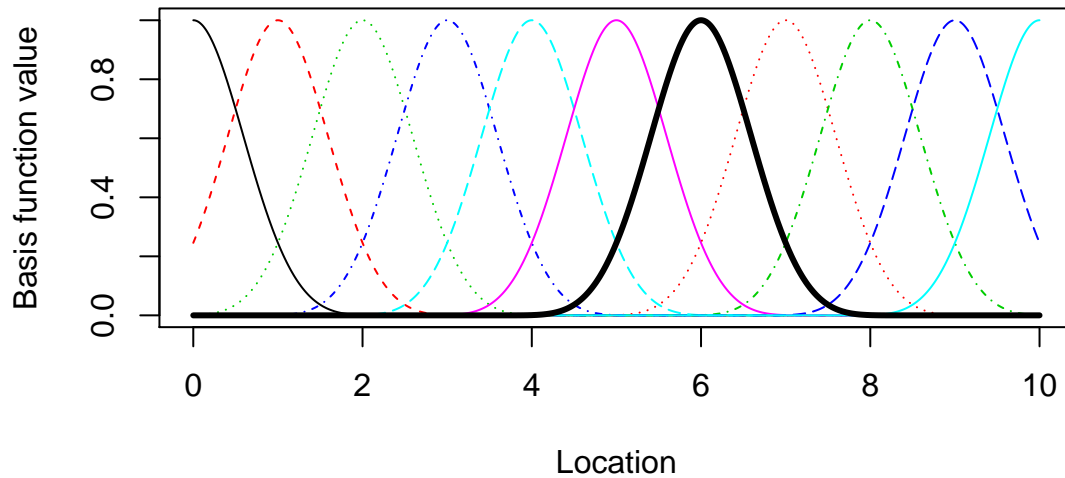
The standard spatial model for Kriging relates the observation to a sum of three components: A polynomial function of the locations (and covariates, if provided), a spatial process, and measurement error.

1.2 The `LatticeKrig` Model

The key feature of `LatticeKrig` is that we model a 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 $\mathbf{y} = \mathbf{Z}\mathbf{d} + \Phi\mathbf{c} + \mathbf{e}$, where \mathbf{y} is the vector of variable measurements, \mathbf{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.

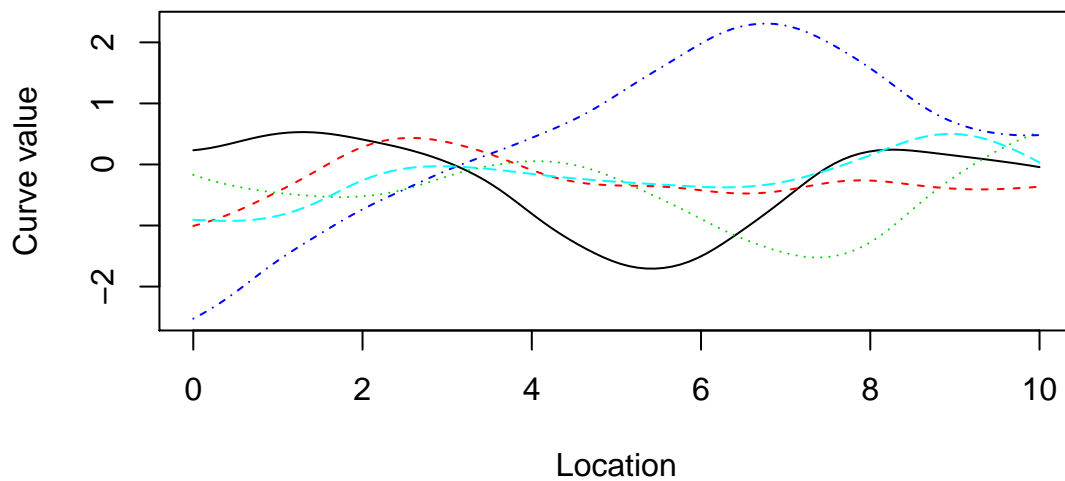
The package is named `LatticeKrig` because of the placement of the basis functions: they are equally spaced in all dimensions on a lattice. We can also consider multiple different lattice sizes simultaneously to better capture different levels of resolution; each additional level has half as much space 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-D Basis Functions



The following plot shows some examples of potential curve fits that could be produced from these basis functions.

Example 1-D Curves



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)

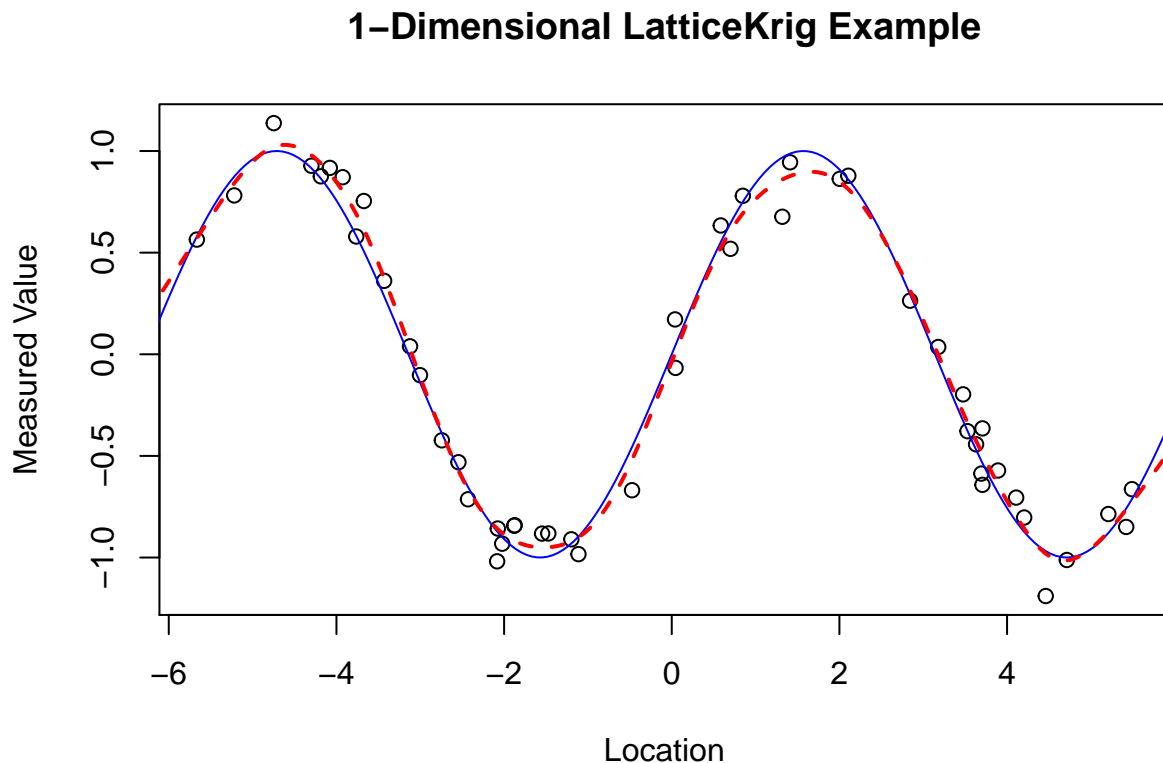
## Call:
## LatticeKrig(x = locations, y = observations)
##
##
## Number of Observations:                50
## Number of parameters in the fixed component 2
## Effective degrees of freedom (EDF)      10.34
## Standard Error of EDF estimate:        0.9605
## MLE sigma                             0.1062
## 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
```

2.2 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 adds a collection of simulations based on the fitted curve; the simulations get farther apart, meaning the standard error gets larger, where there aren't many data points and especially at the edges of the region. These simulations are easier to compute than the standard error, so they are used to estimate the standard error of a model in `LatticeKrig`.

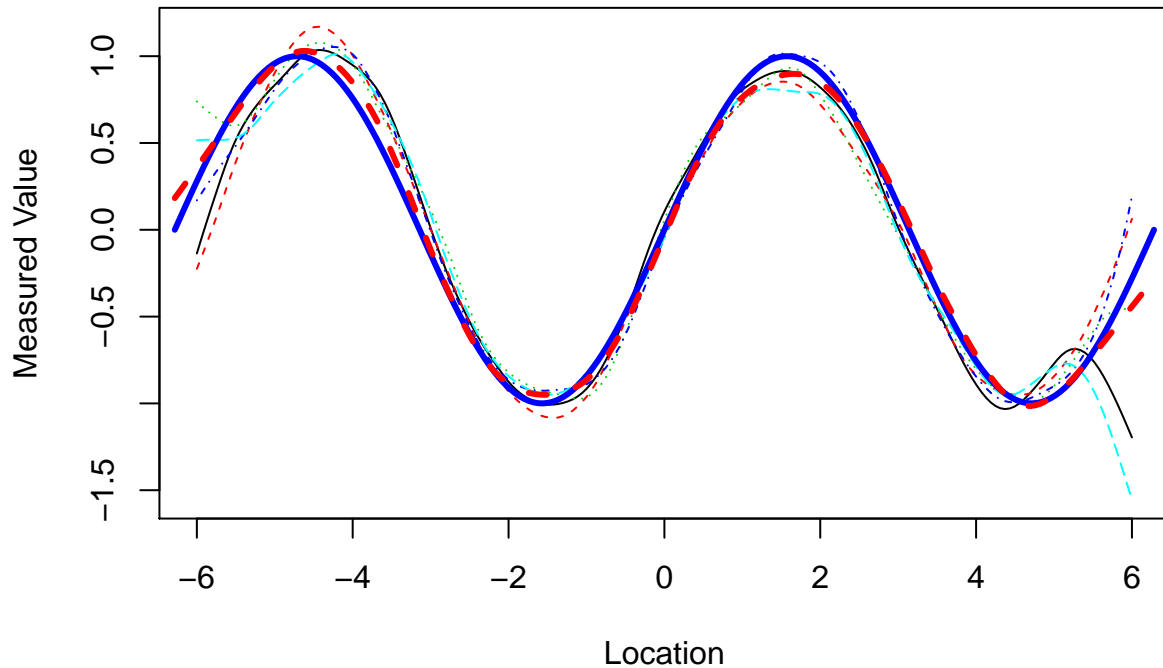
```
uncertainty <- LKrig.sim.conditional(kFit1D, x.grid = as.matrix(xGrid), M=5)
```

```
## 1 2 3 4 5
```

```
matplot(uncertainty$g.draw, type="l", x = seq(-6,6, length.out = 200),
      main = "Simulated Curves", xlab = "Location", ylab = "Measured Value")
```

```
lines(xGrid, sin(xGrid), col='blue', lwd = 3)
lines(xGrid, prediction, col='red', lty=2, lwd=3)
```

Simulated Curves



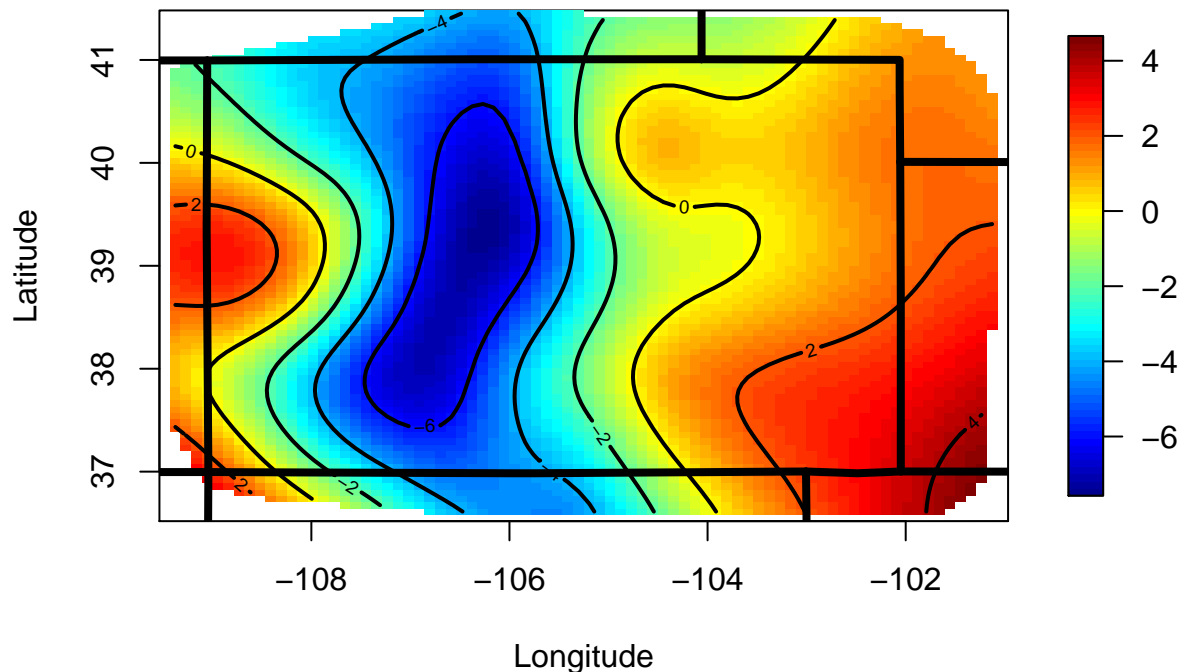
For another, more practical example, we will predict the average spring temperature for locations throughout Colorado. Using the data set `COmonthlyMet`, we can make a surface showing our predictions over a range of longitudes and latitudes, and use the `US` function to draw in the USA state borders 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
kFitWeather <- LatticeKrig(locations, observations)

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

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

2-Dimensional LatticeKrig Example



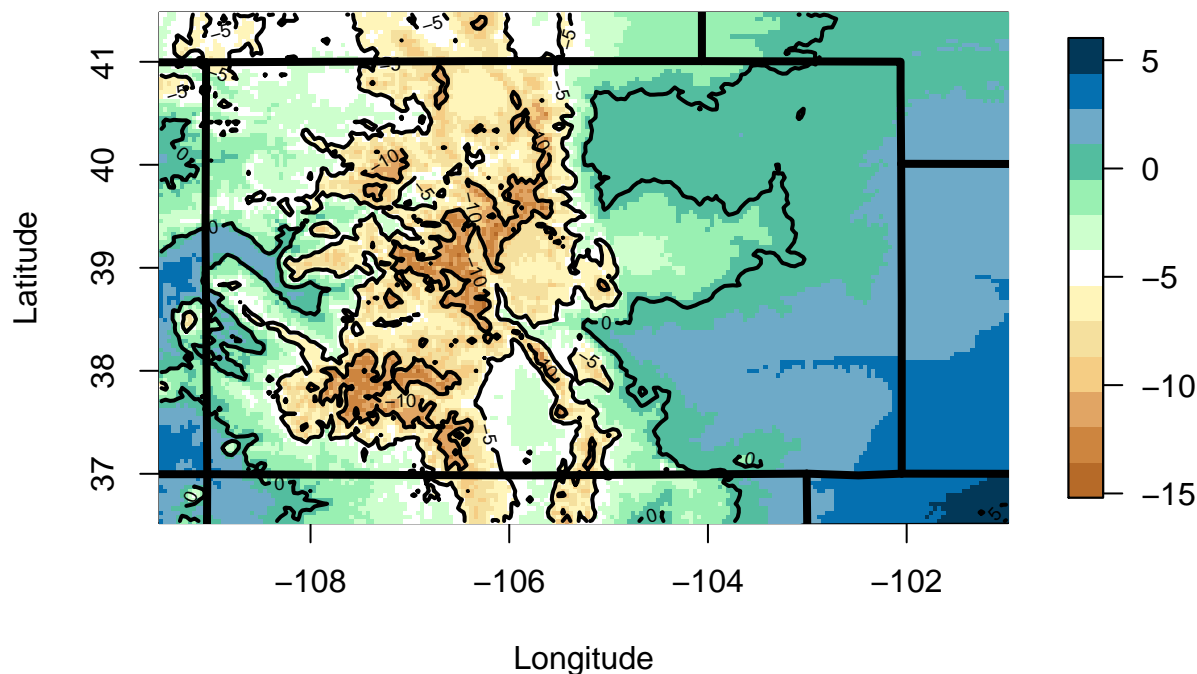
This plot is useful, but we can do better. We can see that the coldest temperatures are in the Rocky Mountains, which is not surprising. Thus, we might expect that we will get a more accurate fit by having `LatticeKrig` account for the elevation at each location as well. Another way we can improve the 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 grid of 30,000 points, which will take longer to compute but produces a nicer looking, more detailed plot. Finally, we can also have `surface` extend the evaluation 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
kFitWeather <- 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(kFitWeather, 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)
```


Improved 2-Dimensional LatticeKrig Example



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 adapt the model to be more realistic. 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`. For context, this is the printout of the `LKinfo` object created by `LatticeKrig` in the quick start guide's one-dimensional example.

```
print(kFit1D$LKinfo)

## Classes for this object are: LKinfo LKInterval
## The second class usually will indicate the geometry
##     e.g. 2-d rectangle is LKRectangle
##
## Some details on spatial autoregression flags:
## stationary:
## first order (by level):
## isotropic: TRUE
##
## Ranges of locations in raw scale:
## [1] -5.665279  5.489935
##
## Logical (collapseFixedEffect) if fixed effects will be pooled: FALSE
##
## Number of levels: 3
## delta scalings: 1.859202 0.9296012 0.4648006
## with an overlap parameter of 2.5
## alpha: 0.7619048 0.1904762 0.04761905
## based on smoothness nu = 1
##
## a.wght: 2.01 2.01 2.01
##
## Basis type: Radial using WendlandFunction and Euclidean distance.
## Basis functions will be normalized
##
## Total number of basis functions 75
## Level Basis size mLevel
##      1      17      17
##      2      23      23
##      3      35      35
##
## Lambda value: 0.0001236536
```

3.1 Required Parameters for LKrigSetup

- `x`

The parameter `x` is used to find the range of the data locations in each dimension for the lattice. As such, it is often easiest to pass in the matrix of observation locations, but you can also just pass in the range directly.

- `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`. Increasing `nlevel` will increase the potential detail of the fitted surface, and 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. Recall that each basis function is centered on a lattice point, so `NC` equivalently controls the number of lattice points to set across the region in the longest dimension. 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` or `nu`

At least one of `alpha`, `alphaObject`, and `nu` must be set. In most cases you will use `alpha` or `nu`. The parameter `alpha` should be a vector of length `nlevel` that holds the weights that scale the basis functions on each different lattice size; `nu` is a scalar that controls how quickly the values in `alpha` decay. When `nu` is set, `alpha` will be filled by setting `alpha[i] = 2^(-2 * i * nu)`, then scaling so the sum of the weights in `alpha` is 1. This scaling should always be done before passing in `alpha` 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 dimensions: 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.

3.2 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`

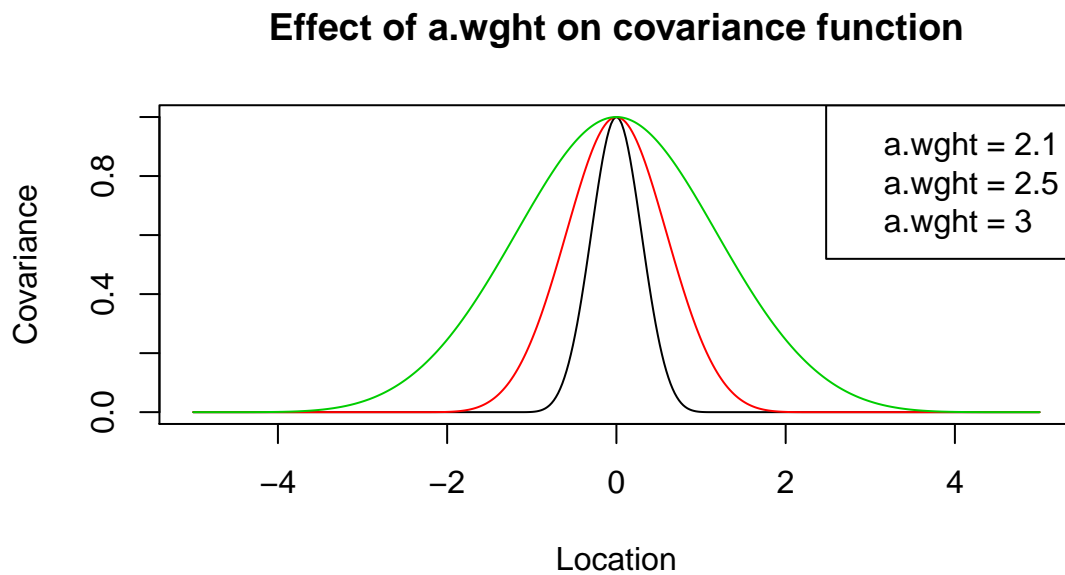
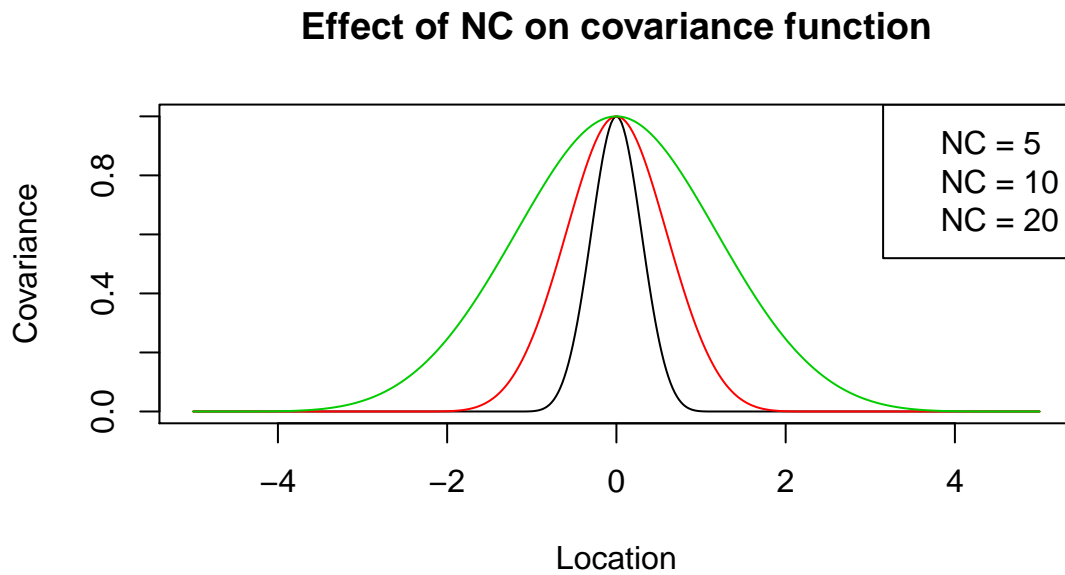
This parameter determines how many lattice points to add outside the range of the data in each direction. The effect of changing this parameter is relatively minor compared to the effect of changing `NC`, and it can only affect the prediction near the edges of the data.

- `normalize`

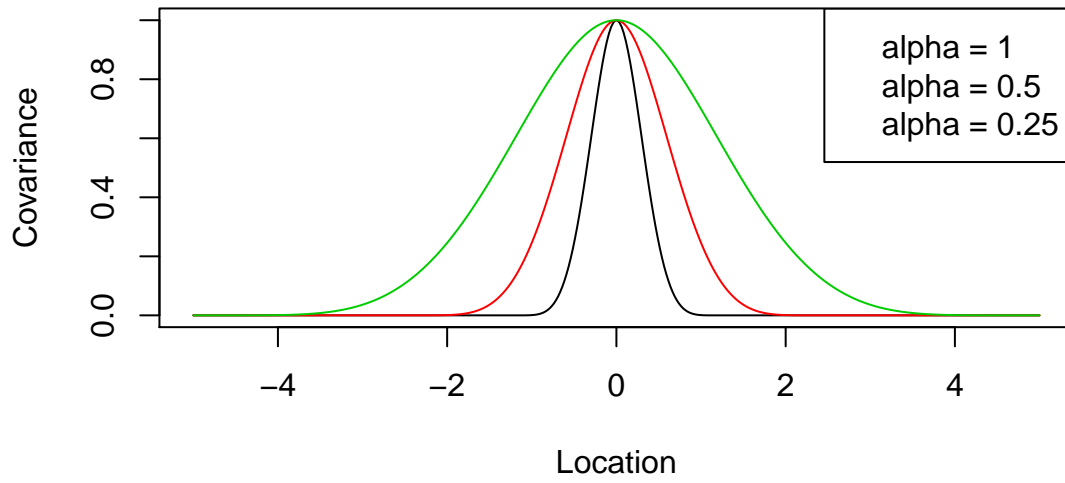
This parameter determines whether or not to normalize the basis functions after computing them, making the variance 1. This is set to **TRUE** by default, sacrificing some computing time to reduce edge artifacts created by the model that aren't present in the real data.

3.3 Effects on covariance function

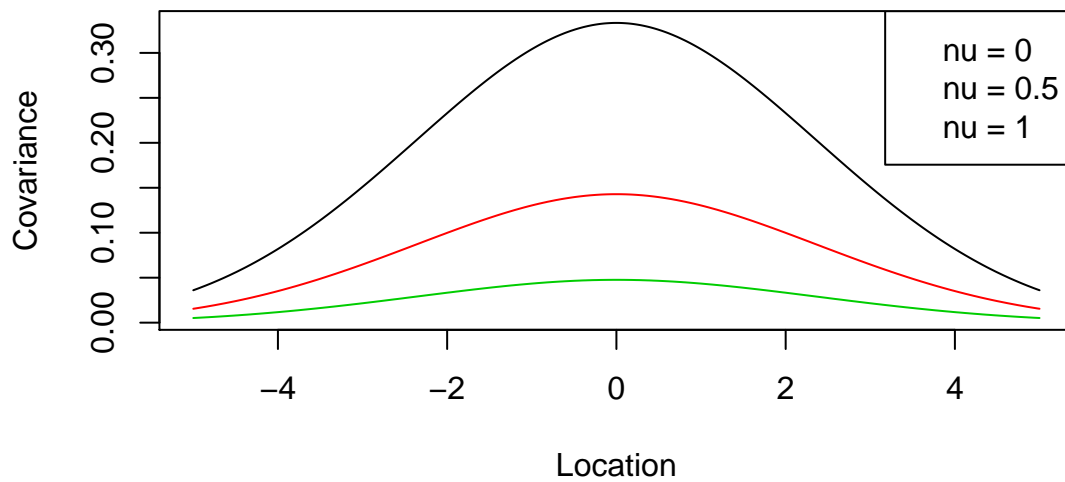
The following plots show how different values of **NC**, **a.wght**, **alpha**, and **nu** affect the covariance function. These plots are all one dimensional for ease of viewing; recall that the covariance function is radially symmetric in higher dimensions.



Effect of alpha on covariance function



Effect of nu on covariance function (3rd level)



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 for other geometries. For instance, this package can work with locations on a sphere, which is useful for locations of latitude and longitude on Earth. There are also other options for non-Euclidean geometries: a cylinder using 3 dimensional cylindrical coordinates, and a ring using 2 dimensional cylindrical coordinates (z and θ at a fixed radius). To set the geometry, set the `LKGeometry` parameter in `LKrigSetup`. These are the current choices:

- "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

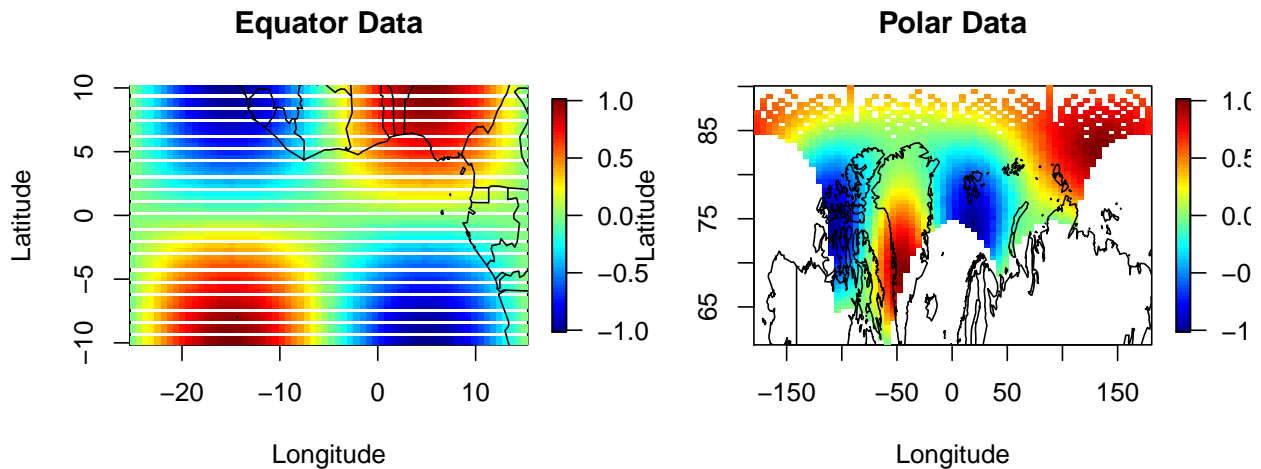
By default, `LKinfo` will use either `LKInterval`, `LKRectangle`, or `LKBox`, depending on the number of dimensions in the given location data. However, if you aren't using `LKRectangle`, it is best to set `LKGeometry` explicitly; failing to do so can cause errors. When using the `LKSphere` geometry, there are also different ways of measuring distance, which you can set using the `distance.type` argument 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 straight-line distance, treating the coordinates as 3-dimensional Euclidean locations. 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. We will expect to see very similar results for the two spherical models and noticeably different results for the two rectangular models.

```
library(LatticeKrig)
data(EquatorData)
data(PolarData)

#plot the data at the equator and at the north pole in rectangular coordinates
#note the significant distortion at the north pole
par(mfrow = c(1,2))
quilt.plot(equatorGrid, equatorGridValues, main="Equator Data", xlab="Longitude", ylab="Latitude")
world(add=TRUE)
quilt.plot(polarGrid, polarGridValues, main="Polar Data", xlab="Longitude", ylab="Latitude")
world(add=TRUE)
```



Now, we will use `LatticeKrig` to approximate the surfaces in both rectangular and spherical geometries, and print out the root mean square error of all four models.

```
par(mfrow = c(2,2))
```

```
rectangleEquatorInfo <- LKrigSetup(equatorLocations, nlevel = 2, NC = 13, NC.buffer = 2, alpha = c(0.8,
rectangleEquatorFit <- LatticeKrig(equatorLocations, equatorValues, LKinfo = rectangleEquatorInfo)
surface(rectangleEquatorFit, main="Equator Surface Prediction \nUsing Rectangular Kriging",
        xlab="Longitude", ylab="Latitude")
#world(add=TRUE)
sqrt(sum(rectangleEquatorFit$residuals^2) / nrow(equatorLocations))
```

```
## [1] 0.0007707522
```

```
rectanglePolarInfo <- LKrigSetup(polarLocations, nlevel = 2, NC = 13, NC.buffer = 2, alpha = c(0.8, 0.2)
rectanglePolarFit <- LatticeKrig(polarLocations, polarValues, LKinfo = rectanglePolarInfo)
surface(rectanglePolarFit, main="Polar Surface Prediction \nUsing Rectangular Kriging",
        xlab="Longitude", ylab="Latitude")
#world(add=TRUE)
sqrt(sum(rectanglePolarFit$residuals^2) / nrow(polarLocations))
```

```
## [1] 0.02332026
```

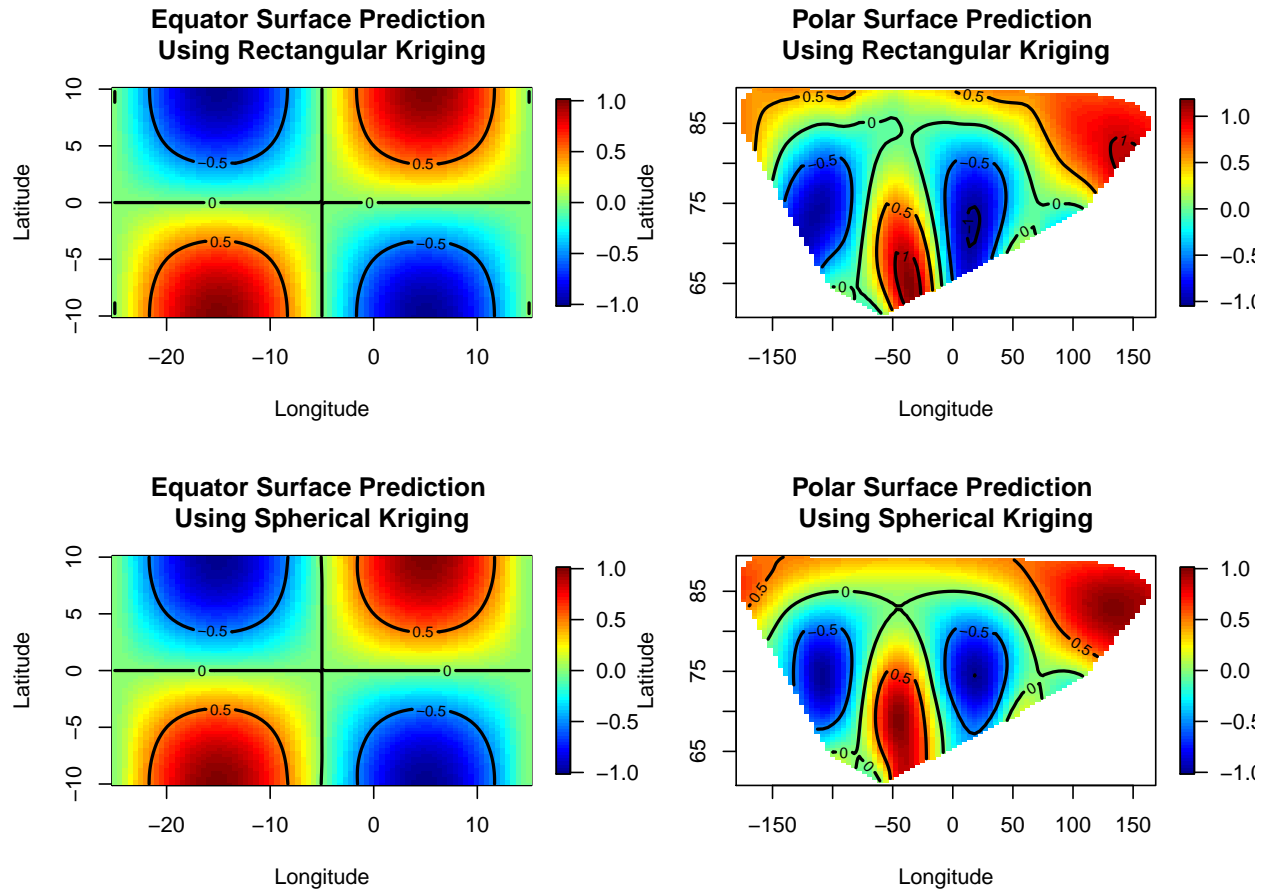
```
info1 <- LKrigSetup(equatorLocations, nlevel = 2, startingLevel = 6, alpha = c(0.8, 0.2)
                 , a.wght = 1.01, LKGeometry = "LKSphere")
sphereEquatorFit <- LatticeKrig(equatorLocations, equatorValues, LKinfo = info1)
surface(sphereEquatorFit, main="Equator Surface Prediction \nUsing Spherical Kriging", xlab="Longitude"
#world(add=TRUE)
sqrt(sum(sphereEquatorFit$residuals^2) / nrow(equatorLocations))
```

```
## [1] 0.00549062
```

```
info2 <- LKrigSetup(polarLocations, nlevel = 2, startingLevel = 6, alpha = c(0.8, 0.2)
                 , a.wght = 1.01, LKGeometry = "LKSphere")
spherePolarFit <- LatticeKrig(polarLocations, polarValues, LKinfo = info2)
sqrt(sum(spherePolarFit$residuals^2) / nrow(polarLocations))
```

```
## [1] 0.001795302
```

```
surface(spherePolarFit, main="Polar Surface Prediction \nUsing Spherical Kriging", xlab="Longitude", ylab="Latitude")
```



```
#world(add=TRUE)
```

As we can see, the rectangular fit fails badly on the data that has been rotated up to the north pole, with approximately 15 times the root mean square error of the corresponding spherical model.

5 Using Sparse Matrices

One unique feature of LatticeKrig is the assumption that the covariance function and basis function have compact support, meaning they equal 0 everywhere outside a certain region. We use radially symmetric basis functions, so the region where the functions have nonzero values is a circle (or sphere in 3D or interval in 1D). This means that many of the pairs of basis functions or data locations will be too far apart to fit in the region, so the covariance between the two points — or the basis function from one point evaluated at the other point, since the basis function and covariance function are the same — will be 0. This leads to covariance matrices and basis matrices where the majority of the entries are 0, which is called a sparse matrix. Computing with sparse matrices can be much faster than computing with normal matrices, since the computer can save space by only tracking the location and values of the nonzero entries, and algorithms can skip all of the 0 entries. This optimization makes sparse matrix computations on large data sets orders of magnitude faster than the traditional corresponding computations.

In this package, we use the `spam` package for sparse matrices. This package has built-in methods for storing, multiplying, and solving sparse matrices, as well as finding their Cholesky decomposition, all of which are used heavily in LatticeKrig. The Cholesky decomposition of a matrix A finds the lower triangular matrix L such that $LL^T = A$. This is heavily used in LatticeKrig because it is significantly easier to solve a triangular system than a normal system ($\mathcal{O}(n^2)$ v.s. $\mathcal{O}(n^3)$), which combines with the optimization of using sparse matrices to make our calculations practical on very large data sets.

6 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.

7 Common Errors

7.1 Could not find function [FunctionName]

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

7.2 Need to specify NC for grid size

7.3 Invalid ‘times’ argument

7.4 Only one alpha specifed for multiple levels

7.5 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.

7.6 Error in `object$x %*% t(Vinv) : non-conformable arguments`

This error can occur when using `LKrigSetup` (or `LatticeKrig`, by extension) on a 1-dimensional problem if you don't explicitly set `LKGeometry = "LKInterval"` for `LKrigSetup`.

7.7 Error in `if (ncol(object$x) != 1) { : argument is of length zero`

This error most commonly occurs when using `LKrigSetup` in one dimension and passing in the range of the data explicitly. For example, `LKrigSetup(c(0,1), ...)` will cause this error (assuming the other arguments are provided correctly). The issue is that the `c` function doesn't format the input as a matrix, which is the format `LKrigSetup` expects. To fix this, just call `as.matrix` on the first parameter you give to `LKrigSetup` - so we would correct the example above to `LKrigSetup(as.matrix(c(0,1)), ...)`.

8 Frequently Asked Questions

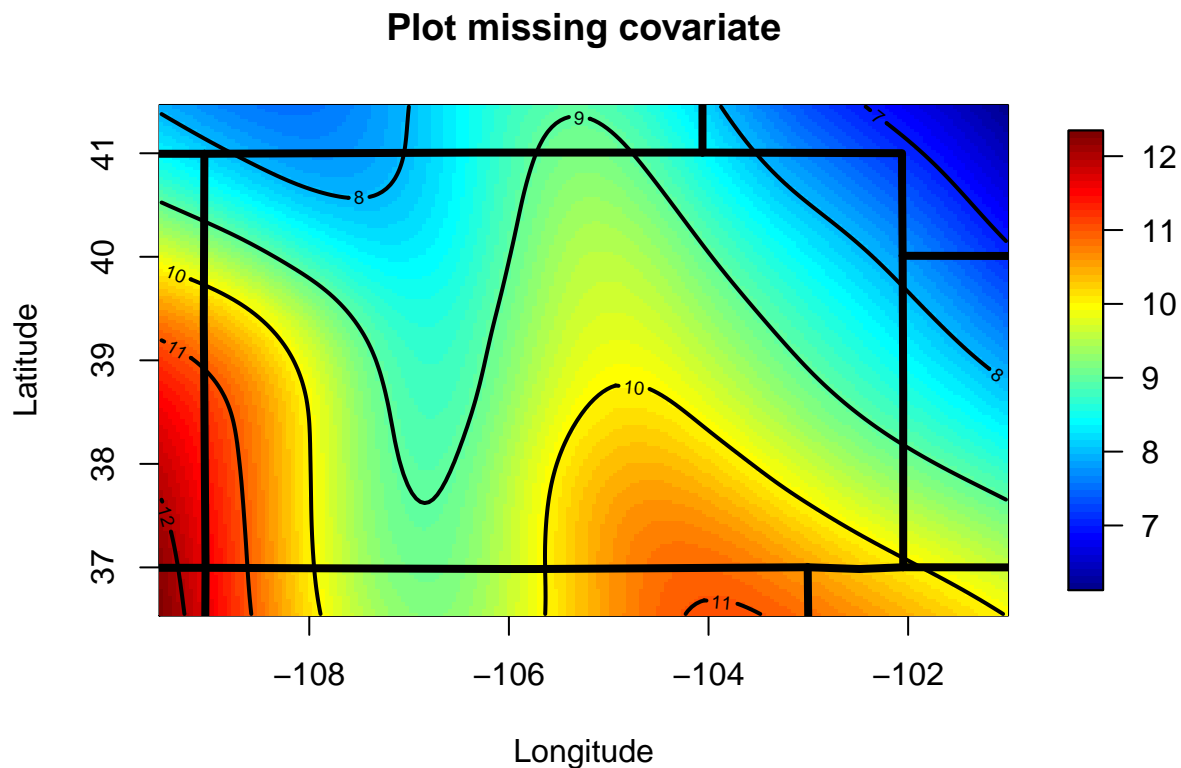
8.1 Does the order of the parameters matter?

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. For this reason, it is best practice to use the names of the parameters while passing them in, except in cases where it is obvious.

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

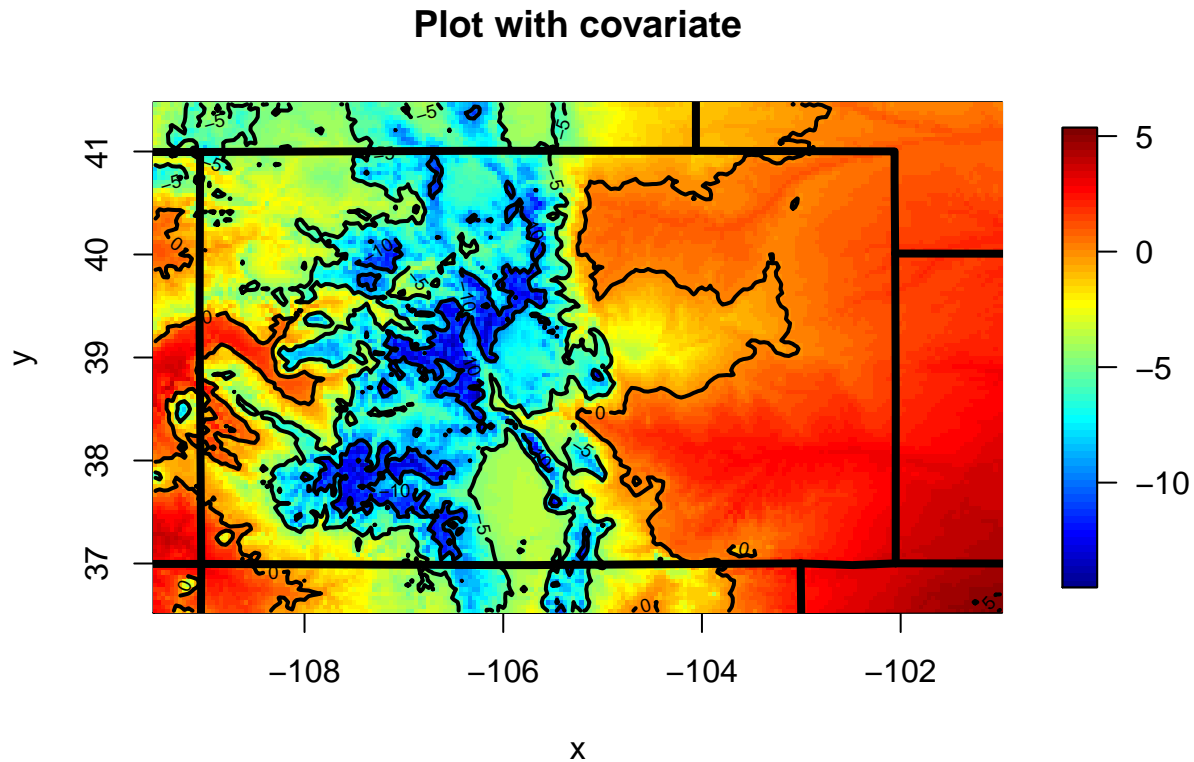
If your model includes covariates (the `Z` parameter of `LatticeKrig` and `LKrig`), your plot may not have included the effect of the covariate. The following code demonstrates this issue using the Colorado temperature data and kriging fit from the quick start guide, and how to fix the issue. Using the `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(kFitWeather, 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(kFitWeather, 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)
```



8.3 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.

9 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 vector \mathbf{Z}_i 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

$$y_i = \mathbf{z}_i^T \mathbf{d} + g(\mathbf{s}_i) + \epsilon_i$$

where $\epsilon \sim MN(\mathbf{0}, \sigma^2 I)$ and $g(\mathbf{s})$ is a Gaussian Process with mean zero and covariance function $k(\mathbf{s}, \mathbf{s}')$. 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 coefficients 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: $\mathbf{d} = (Z^T \Sigma^{-1} Z)^{-1} Z^T \Sigma^{-1} \mathbf{y}$. 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 $\hat{\mathbf{c}} = P \Phi^T \Sigma_{11}^{-1} (\mathbf{y} - Z\mathbf{d})$.

9.1 Sparse Matrix Algorithms

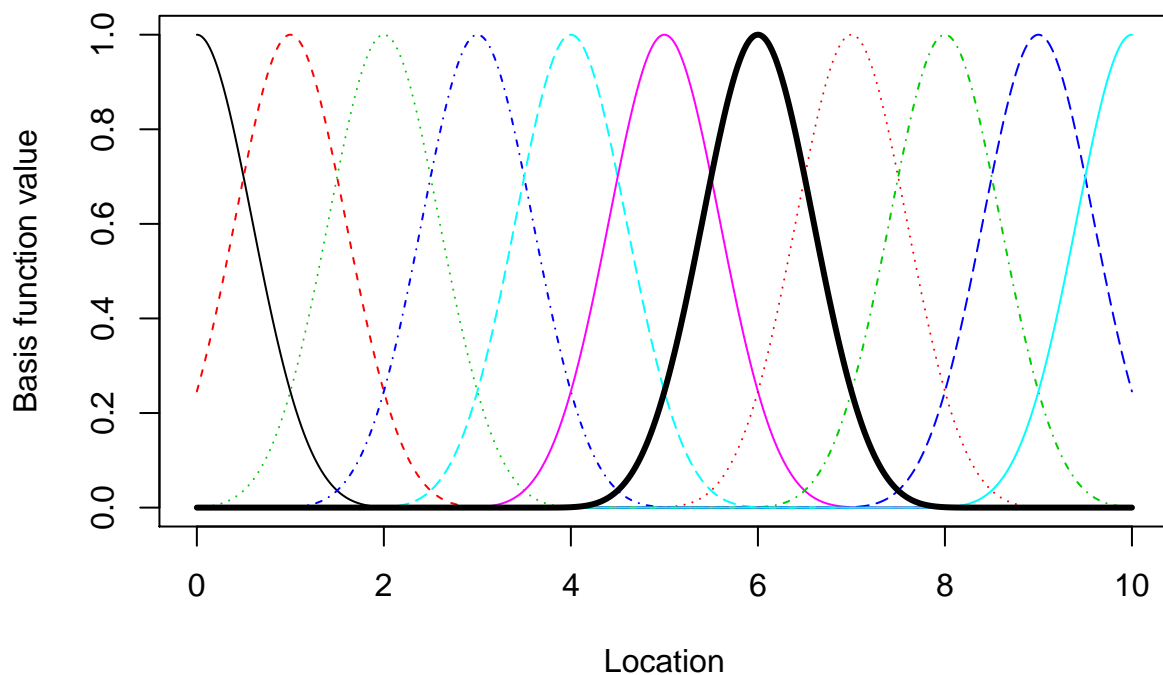
As mentioned earlier, the LatticeKrig package is able to handle large data sets because the covariance function equals 0 for large input. Recall that we make the simplifying assumption that the covariance function is the same as the radial basis function. In LatticeKrig, this function is a Wendland function:

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

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)
```

1-D Basis Functions



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.

10 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.

After loading in the data, we start by filtering out the NA values in `y`

```
data(ozone2)
x <- ozone2$lon.lat
y <- ozone2$y[16,]
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)
```

These two kriging fits produce identical predicted values and standard errors. To make `mKrig` use the same covariance function as `LKrig`, we set the parameter `cov.function="LKrig.cov"`. The `LKrig.cov` function is a top level function that computes the covariance between arbitrary sets of locations according to the model specified by the `LKinfo` object. Note that `LKrig` uses the (sparse) precision matrix instead of inverting the covariance matrix, which is one of the reasons that `LKrig` is much faster than `mKrig` for large data sets.

11 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 JCGS article.

11.1 First Example: One level, no normalization

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

```
lambda = 0.05
overlap = 2.5
basis <- function(d) {
  return(1/3 * (1-d)^6 * (35*d^2 + 18*d + 3) * (d < 1))
}

#clear x and y to make sure our data doesn't get overwritten
rm(x, y)
data(KrigingExampleData)
```

Next, we create an equally spaced lattice of 6 points in $[0,1]$ and add 5 additional points on either side; since we only have 1 level in 1 dimension, this is relatively easy.

```
nc <- 6
ncBuffer <- 5

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

#adding the buffer lattice points outside the interval
latBefore <- seq(to=0-delta, by=delta, length.out = ncBuffer)
latAfter <- seq(from=1+delta, by=delta, length.out = ncBuffer)
lattice <- c(latBefore, latInside, latAfter)
m <- 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*delta))
B <- LKDiag(c(-1, 2.01, -1), m)
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)

info <- LKrigSetup(as.matrix(c(0,1)), NC = 6, NC.buffer = 5, nlevel = 1, a.wght = 2.01, alpha = 1, lambda = 0.05)
```

```

krigFit <- LKrig(x, y, LKinfo = info)

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

#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
test.for.zero(krigPredictions, predictions)

## PASSED test at tolerance 1e-08

```

11.2 Second example: One level with normalization

In this example, 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
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

#calculating Phi matrix for prediction locations
xGrid <- seq(0,1,length = 200)
PhiPredict <- basis(rdist(xGrid, lattice) / (overlap*delta))
#normalizing PhiPredict too
DPredict <- PhiPredict %*% P %*% t(PhiPredict)
#discarding the off-diagonal elements of D
DPredictS <- diag(diag(DPredict)^(-1/2))
PhiPredict <- DPredictS %*% PhiPredict

```

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

```

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

```

```

Z <- cbind(1, x)
ZPredict <- cbind(1, xGrid)
dHat <- solve(t(Z) %*% Minverse %*% Z, t(Z) %*% Minverse %*% y)
G <- t(Phi) %*% Phi + lambda*Q
cHat <- solve(G) %*% t(Phi) %*% (y - Z %*% dHat)
predictions <- ZPredict %*% dHat + PhiPredict %*% cHat

info <- LKrigSetup(as.matrix(c(0,1)), NC = 6, NC.buffer = 5, nlevel = 1, a.wght = 2.01, alpha = 1, lambda = 0.05)
krigFit <- LKrig(x, y, LKinfo = info)
krigPredictions <- predict(krigFit, xGrid)

#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
test.for.zero(krigPredictions, predictions)

## PASSED test at tolerance 1e-08

```

11.3 Third 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))
}

```

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

L2Inside <- seq(from=0, to=1, by=delta/2)
L2Before <- seq(to=0-delta/2, by=delta/2, length.out = ncBuffer)

```

```

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

L3Inside <- seq(from=0, to=1, by=delta/4)
L3Before <- seq(to=0-delta/4, by=delta/4, length.out = ncBuffer)
L3After <- seq(from=1+delta/4, by=delta/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*delta)) * sqrt(alpha[1])
Phi2 <- basis(rdist(x, L2) / (overlap*delta/2)) * sqrt(alpha[2])
Phi3 <- basis(rdist(x, L3) / (overlap*delta/4)) * sqrt(alpha[3])
Phi <- cbind(Phi1, Phi2, Phi3)

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)

```

```

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 covariance matrix and comparing it to the LKrig one
targetBasis <- spam2full(LKrig.basis(x, info))
test.for.zero(targetBasis, Phi)

## PASSED test at tolerance 1e-08
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

```

11.4 Using the kriging equations directly

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

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

In this section, we will compute the estimates directly using these equations to show that the computations in LatticeKrig match them. For brevity, we will only consider the case with one level and no normalization; adjustments for more levels and normalization can be made in the same way as in the previous section.

```

data(KrigingExampleData)
info <- LKrigSetup(as.matrix(c(0,1)), NC = 6, NC.buffer = 5, nlevel = 1, a.wght = 3, alpha = 1, lambda = 0)
krigFit <- LKrig(x, y, LKinfo = info)
nc <- 6
ncBuffer <- 5
lambda = 0.05
delta <- 1/(nc-1)
latInside <- seq(from=0, to=1, by=delta)
latBefore <- seq(to=0-delta, by=delta, length.out = ncBuffer)
latAfter <- seq(from=1+delta, by=delta, length.out = ncBuffer)
lattice <- c(latBefore, latInside, latAfter)
overlap <- 2.5

cov <- function(d) {
  s <- d/(overlap*delta)
  return(1/3 * (1-d)^6 * (35*d^2 + 18*d + 3) * (d < 1))
}
Sigma <- cov(rdist(x, x)) + lambda*diag(1, length(x))
SigmaInv <- solve(Sigma)
Phi <- cov(rdist(x, lattice))

```

```

P <- cov(rdist(lattice, lattice))
ones <- rep(1, length(x))
Z <- cbind(ones, x)
dHat <- solve(t(Z) %*% SigmaInv %*% Z) %*% t(Z) %*% SigmaInv
cHat <- P %*% t(Phi) %*% SigmaInv %*% (y - Z %*% dHat)
test.for.zero(dHat, krigFit$d.coef)

```

```
## FAILED test value = 110.2513 at tolerance 1e-08
```

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

```
## FAILED test value = 2433.011 at tolerance 1e-08
```