How this package works

This package is designed to be modular and separate the steps of computation, prediction and simulation. It also uses R's function overloading with S3 and S4 objects to simplify the coding. This strategy is especially helpful for different geometries and distance functions. Finally, the use of sparse matrix methods is also implemented as S4 methods through the spam package and so much of linear algebra uses the standard R matrix multiplication operator `%*%` even through sparse computations are being done behind the scenes.

Despite this complexity it is important to keep in mind the computation and statistical results are just the usual ones associated with Kriging and maximum likelihood associated with a Gaussian spatial process model. The difference is that the spatial process is specified in a way that is suited to generating sparse matrices for the computations. Also overloading function calls makes the code handling different geometries easier to read and write, as demonstrated below.

An example of classes and methods

To fix some concepts we give a simple illustration of overloading a function using S3 methods. The (trival) operation is to find an equally spaced grid based on the ranges and spacing delta and we would like this for 1D and 2D. First we define the method `makeGrid` for the two classes: `1D` and `2D`.

```
makeGrid <- function(gridInfo, ...) {
    UseMethod("makeGrid")
}

makeGrid.1DGrid <- function(gridInfo,...) {
  out<- list(x= seq(gridInfo$min, gridInfo$max, gridInfo$delta))
  return(out)
}

makeGrid.2DGrid <- function(gridInfo,...) {
  out<- list(x = seq(gridInfo$minX, gridInfo$maxX, gridInfo$delta),
             y = seq(gridInfo$minY, gridInfo$maxY, gridInfo$delta))
  return(out)
}
```

The first function is a template that is used for disptaching and the two following functions actually handle the two cases. Now to use these we create some objects and just call `makeGrid`.

```
test1 <- list(min=0.0, max=1.0, delta=.2)
class(test1) <- "1DGrid"
out1 <- makeGrid(test1)
print(out1)
```

```
## $x
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

```
test2 <- list(minX=0.0, maxX=1.0,
              minY=0.0, maxY=2.0, delta=.2)
class(test2) <- "2DGrid"
out2 <- makeGrid(test2)
print(out2)
```

```
## $x
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
##
## $y
```

```
##  [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

The analogy in the LattticeKrig coding is that there are several generic steps in defining the spatial model that benefit from using a method without haing to add many different cases in the top level functions based on geometry. In this example one could just have a line such as `out <- makeGrid( gridInfo)` and the class of `gridInfo` would direct which function is called.

LKinfo object

The model computation is controlled by the `LKinfo` object. This is a list of class `LKinfo` that has all the information needed to specify the spatial model. One could build this list directly including all the necessary information although is it is easier to use the function `LKrigSetup` to make sure all the details are filled in correctly. This function will also makes some checks on the passed arguments and will fill in some with defaults if not specified. LatticeKrig provides a print method for this object class and that creates the summary of the model when this object is printed. Printing this as a raw list would usually be a big mess of output and not very helpful! See the source code for `print.LKinfo` to see how this is done. The `LKinfo` object also has a second class that is the geometry. This controls how other components in this object are filled in. In particular what is in the component `LatticeInfo` will change reflecting different lattices based on the geometries. See `help(LKGeometry)` for more details on what needs to done for a given geometry.

As a specific example here is how these steps fill in the lattice information for the LKInterval geometry, the 1-D model.

In `LKrigSetup` is the code

```r
LKinfo$latticeInfo <- do.call("LKrigSetupLattice", c(list(object = LKinfo,
        verbose = verbose), setupArgs))
```

The LKinfo object in this case has already been given the class "LKInterval" and so the call to `LKrigSetupLattice` branches to the actual function `LKrigSetupLattice.LKInterval`

Although all the details of this function are not important for this illustration, there are several key features. First, all the information for constructing the lattice comes from components of LKinfo. Here `NC`, the number of initial lattce points in the spatial interval, is used to determine the grid spacing and a `for` loop is used to fill in the lattice points for the sequential levels, if there is more than one level specified. Finally, all the resulting parts describing the lattice are combined as a list: this object becomes the component `latticeInfo` in the ttt{LKinfo} object. These components are consistent across the different geometries and so it makes it possible to have a single summary/print method for the `LKinfo` object.

LKrig function

The basic computation where the basis coefficients are estimated based on locations and data is done in `LKrig`. These steps track the explicit linear algebra in Section 11.1 and as coded, hide the details from different models. 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. Essentially the `LKinfo` object is the reference for what needs to be done. And the coding at this level does not have explicit branches to different geometries.

For example the line

```r
Q <- LKrig.precision(LKinfo)
```

Creates the correct precision matrix for the basis function coefficients by using the information from `LKinfo`.

This line in LKrig

```
G <- t(wX) %*% wX + LKinfo$lambda * (Q)
```

assembles the G matrix from the precision matrix, the value of lambda and the weighted basis function matrix. Although these matrices are in sparse format, note that the %% operator is used because the spam package has provided methods for addition and multiplication using the typical operators. Creating the different components of the model in LatticeKrig is also an example of overloading where the class is the value of LKgeometry.

One advantage of this structure is that new features can be added to the LatticeKrig models without having to change the basic LKrig function and its computational steps. It also results in many key steps only happening one place. For example, the Cholesky decomposition of the G matrix created above is done only in one place in this package. Moreover, the subsequent steps of finding the basis coefficients (LKrig.coef), computing the predicted values, evaluating the likelihood (LKrig.lnPlike), and finding the approximate model degrees of freedom (LKrig.traceA) are the same for any model. That is, they are unique functions that work for any geometry or configuration of the lattice and SAR weights. Of course the advantage here is that the code base needs to changed in only one place if any of these basic steps are modified or corrected.

Estimating covariance parameters.

The function LKrig is designed to compute the model fit assuming the parameters a.wght and lambda are known. With these parameters fixed the likelihood can be maximized in closed form for the remaining parameters, d, sigma and rho. By default these values are used in fitting the model in this function. The component lnProfileLike returned in the LKrig object is the log likelihood having maximized over d, sigma and rho.

The parameters lambda and a.wght are estimated using maxmimum liklihood with built in optimizers in R and the wrapper functions LKrigFindLambda for a fixed a.wght (using optim) and LKrigFindLambdaAwght for finding both parameters (using optimize). In eiher case, the likelihood is evalatued by calls from the optimizer to LKrig. The search over lambda is done in the log scale and over a.wght in a scale (called omega) that is propotional to the log of the correlation range parameter. For a 2-D rectangle this is log(a.wght - 4)/2. (See Awght2Omega)

One useful feature of the optimzation code is that each evalution of the likelihood is saved and these are returned as part of the optimization object. This helps to get an idea of the likelihood surface and determine the reliabilty of the result as a global maximum. See the component MLE$lnLike.eval in the LatticeKrig output object. One could call the LKrig function over a grid of parameters to explore the likelihood surface, however, often the points of opportunity evaluated by the optimizer are enough to interrogate the surface.

LatticeKrig

The top level function LatticeKrig is an easy way to estimate these models parameters from a minimal specification of the model and then to evaluate using these estimates. It is also convenient to setup the LKinfo object. In particular LatticeKrig also makes use of the LatticeKrigEasyDefaults method depending on LKGeometry and provides flexibility of filling in standard model default choices without continually retyping them.

Prediction

Model predictions at the data locations are returned in the fitted.values component of the LatticeKrig and LKrig objects. Predictions of the fitted curve or surface at arbitrary locations are found by multiplying the new covariate vectors with the d.coef linear model parameters and multiplying the basis functions with the c.coef coefficients. It is probably no surprise that creating the basis functions depends on the components of the LKinfo object. But with this structure we have a simple form for making predictions, in keeping with other methods in R. In general if `MyLKinfo` is the model specification and `x1` are the locations for evaluating the process, we predict the model at the locations with the line

```
gHat <- predict(MyLKinfo,x1)
```

Here the returned vector has the predictions of the smooth function and the low order polynomial terms (if present) at the locations `x1`.

One complication in this process is the need to evaluate the predictions on a grid of covariates. In the Colorado climate example one may want to predict just the smooth function of climate based on latitude and longitude or add to it the linear model adjustment due to elevation. Moreover, one might want to evaluate these on a grid to make it easier to plot the results on a map. In 2 or more dimensions keeping track of the grids adds a step to this process; refer to the help files and the Quick Start example for more details.

Simulation

A feature of the LatticeKrig model is that it is efficient to simulate realizations of the process. This operation, called *unconditional simulation*, can be then used to generate conditional simulations (conditioned on the data points) to determine approximate prediction standard errors and quantify the estimate's uncertainty.

The LKinfo object contains all the model attributes needed to simulate a realization of the process. In general if `MyLKinfo` is the model specification and `x1` are the locations for evaluating the process,

```
gSim <- LKrig.sim(x1, LKinfo)
```

simulates the process at the locations and returns the values as a vector in `gSim`. Note that this function is designed to only simulate the random process part of the model. The fixed linear part involving the `Z` covariates is not included.

Although it is beyond the scope of this Vignette to go into the details of conditional simulation it is useful to explain how the package is designed to do this computation – and what it is good for! Suppose you have fit a model to data, with the results in `MyFit` as a LKrig or LatticeKrig object and suppose `Z1` are the covariates at the locations `x1`.
The following code generates 10 draws from the distribution of the unknown process *given* (i.e. conditional on) the observations. This random sample is often known as an ensemble. As a frequentist-based package, the conditioning in LatticeKrig also assumes that sigma, rho, alpha and a.wght covariance parameters are known. (A Bayesian approach would also sample these from their posterior distribution.)

```
 aDraw<- LKrig.sim.conditional( MyFit, M=10, x.grid= x1, Z.grid=Z1)
```

The interpretation of `aDraw` is that each column of aDraw is an equally likely representation of the process and linear model at locations `x1` given the observed data. As M becomes large the sample mean of the ensemble will converge to the estimate from LKrig. The sample covariances of the ensemble will converge to the correct covariance matrix expressing the unceratinty in the estimate. Refer to the Quick Start for an example of the difference between unconditional and conditional simulation.

We note that like the unconditional simulation this code is depends on the LKinfo object in MyFit, applies the estimate computation from LKrig, and the predict function for an LKrig object. In this way the basic computational alogrithms are reused from the code base and appear only in only one place.