`fields` vignette

Ashton Wiens, Mitchell Krock, Emma Lilly and Doug Nychka

06 May, 2021

Contents

#Introduction

`fields` is an R package for curve and function fitting with an emphasis on spatial data. The major methods include:

- `spatialProcess` An easy to use method that fits a spatial process model (aka Kriging) but also estimates the key spatial parameters: nugget variance, sill variance, and range by maximum likelihood. The default covariance model is a Matern covariance function, although it is easy to provide a user supplied covariance coded in R.

- `Tps` Thin Plate spline regression including GCV and REML estimates for the smoothing parameter.
- `mKrig` (micro Krig) and `fastTps` compute fast efficient Universal Kriging and spline-like functions. They can take advantage of sparse covariance functions and thus handle very large numbers of spatial locations.
- `Krig` Spatial process estimation that is a core function of fields. Again, any covariance function implemented in R code can be used for spatial prediction. This engine uses a numerically stable eigendecomposition for computation (compared to `mKrig`).

Other noteworthy functions include:

- `vgram` find variograms for spatial data (and with temporal replications).

- `as.surface`, `as.image`, `image.plot`, `drape.plot`, `quilt.plot`, `add.image`, `designer.colors` Many convenient functions for working with image data.

- `sreg`, `splint`, `qsreg` Fast 1-D smoothing splines, interpolating cubic splines, and quantile splines.

- `image.smooth`, `smooth.2d`, `interp.surface` Some image processing functions useful for spatial data analysis.

Generic functions that support the methods:

- `plot` - diagnostic plots of fit

- `summary` - statistical summary of fit

- `print` - shorter version of summary

- `surface` - graphical display of fitted surface

- `predict` - evaluation fit at arbitrary points

- `predictSE` - prediction standard errors at arbitrary points.

- `sim.rf` - simulate a random field on a 2-d grid.

For many of these functions, we present their basic usage in this vignette. We have edited out secondary arguments in the presentation to focus on the essential options.

We have tried to present this vignette as a narrative that introduces spatial statistics, describes how to use the functions in `fields`, and also includes some relevant theory about splines and Kriging. The goal is to get the interested reader started quickly with many data based examples. Core functions in `fields` return S3 objects that are compatible with generic functions such as `plot`, `predict`, `surface`, etc. Thus, the analysis flow will seem familiar to other methods such as `lm`. If the reader is familiar with Kriging, they can skip to Chapters 6-8 to see some examples.

We begin by showing quick examples using some of `fields`' most important functions. This gives the user an idea of what kind of problems can be solved and how to implement the functions in `fields` to solve them.

In the third chapter we discuss fitting splines to univariate data. We begin with a simple time series data set, but the discussion applies more generally to data with one predictor variable x and one response variable y. The functions `splint` and `sreg` are used to fit cubic splines to this type of data. Then we introduce `Tps`, a core function in the package that can fit thin plate splines.

Chapters 4-6 build up to Kriging. We have tried give an intuitive presentation of the fundamental concepts in spatial statistics, and at the same time introduce the corresponding functions in `fields`. Initially, the Kriging linear algebra is done manually for exposition, and then the package's most important function `spatialProcess` is presented in Chapter 7.

There is a brief interlude to discuss the package's plotting functions as well as some image processing functions in Chapters 8-9. Many of them will have been used up to this point, but here we give a thorough description of the plotting options to be used with images, surfaces, and the package's models. See `quilt.plot` to get a quick plot of raw data to discern spatial patterns. There are also several functions for working with image data specifically. The functions `image.smooth` and `smooth.2d` are capable of smoothing images, and `interp.surface` performs fast bilinear interpolation (often useful when zooming in on images from a grid).

In Chapter 10, we return to covariances and Kriging in more detail, including all of the covariance options in `fields`. Then we show how the user can write their own covariance function for use with all of the functionality in `fields`. This allows use to discuss three important wrappers on `Krig` in the following chapters.

In Chapter 11, we cover the Kriging optimization workhorse `spatialProcess`. The user only needs to specify the data and a covariance model, and then `spatialProcess` will perform a grid search for the optimal values of the rest of the parameters and return the spatial model. Optimization methods include restricted maximum likelihood (REML) and generalized cross validation (GCV). The user should exercise caution, however, when using black box optimization routines. We hope this chapter will help explain the optimization procedure and assist the user with fitting the right model parameters.

In Chapter 12, we introduce the concept of using a compactly supported covariance function to generate a sparse covariance matrix. The function `mKrig` (microKrig) is set up to take advantage of this sparsity in computation by using the `spam` package. We return to the `Tps` function in Chapter 13 with spatial data. The theory of thin plate splines is presented in the framework of Kriging. Finally, we show how the `fastTps` function is an analogue of `mKrig` that can similarly take advantage of sparse linear algebra.

The Appendix gives the mathematical definition of the Kriging estimators for simple, ordinary, and universal Kriging, as well as brief explanations of restricted maximum likelihood (REML) and generalized cross validation (GCV).

#`fields` in action

This package deals with fitting curves and surfaces to data to try to represent the underlying model/process. Our overall goal is to write R functions that make data analysis fluid and simple. Much of the framework can be thought of as a generalization of what one learns in a basic statistics course - namely fitting a model by finding the optimal parameters using some measure of goodness-of-fit such as the residual sum of squares. This means that all of the model objects fit in this package will include standard statistical output such as the parameters, fitted values, and residuals. In general we strive to code the `fields` functions in a modular way with numerous comments. We also tend to avoid formula and additional object classes to keep code structure transparent.

Some readers may be interested in analyzing the source code of `fields` functions. Comments in the source code are not available when `fields` is downloaded from CRAN, but are available when downloaded from https://www.image.ucar.edu/~nychka/Fields/.

In this first section we highlight some of the high-level functions and how to use them for the impatient reader. The rest of the vignette goes into much more depth concerning the theory behind and use of the most of the external functions in `fields`.

##Visualizing raw data with `quilt.plot`

The `quilt.plot` function takes irregularly-spaced data, puts it on a grid, and plots the result. This can be useful if you have many datapoints that are nearly collocated. Consider the `NorthAmericanRainfall` dataset, which describes precipitation over North American at non-gridded longitude and latitude values. The following shows how to use `quilt.plot`.

```
library(fields)
data(NorthAmericanRainfall)
x<- cbind(NorthAmericanRainfall$longitude, NorthAmericanRainfall$latitude)
y<- NorthAmericanRainfall$precip
quilt.plot(x,y)
world(add=TRUE)
```

The `fields` commands `world(add=TRUE)` and `US(add=TRUE)` can quickly add outlines to our maps.

Of course, `ggplot2` can reproduce a `quilt.plot`, but it is often more laborious when using spatial data. Here, we include `ggplot2` code for comparison.

```
library(ggplot2)
data(NorthAmericanRainfall)
df <- data.frame(NorthAmericanRainfall$longitude,NorthAmericanRainfall$latitude,
                 NorthAmericanRainfall$precip)
names(df) <- c("Longitude","Latitude","Precip")
us = map_data("usa")

ggplot()  + geom_polygon(data = us, aes(x=long, y = lat, group = group), fill = NA, color = "black") + 
scale_colour_gradientn(colours = tim.colors()) + labs(colour="Precip")
```

Be aware that there is an important difference between these functions: `quilt.plot` by default finds the average in each grid box, while `ggplot` overlays all the individual values.

0.1 `spatialProcess` with a covariate

`fields` makes it easy to fit a spatial model from data, predict at arbitrary locations or on a grid, and plot the results. Additional covariates (e.g. elevation) can also be included in the linear trend. We can use `predictSurface` to predict on a grid and output a surface object, much like `predict` but more convenient for plotting.

```
data( COmonthlyMet)
# predicting average daily minimum temps for spring in Colorado
obj<- spatialProcess( CO.loc, CO.tmin.MAM.climate, Z= CO.elev)
out.p<-predictSurface.Krig( obj, grid.list=CO.Grid, ZGrid= CO.elevGrid, extrap=TRUE)

image.plot( out.p, col=larry.colors())
US(add=TRUE, col="grey")
contour( CO.elevGrid, add=TRUE, levels=seq(1000,3000,,5), col="black")
title("Average Spring daily min. temp in CO")
```

We can also find standard errors (for a fixed set of parameter estimates):

```
out.p<-predictSurfaceSE( obj, grid.list=CO.Grid, ZGrid= CO.elevGrid, extrap=TRUE) #ZGrid= CO.elevGrid
# error drop.Z not supported ??
image.plot( out.p, col=larry.colors())
US(add=TRUE, col="grey")
```

```r
points(CO.loc[,1], CO.loc[,2], col="magenta", pch=21, bg="white")
title("Standard errors for avg Spring daily min. temp in CO")
```

Computing standard errors can be computationally expensive. An alternative is to use `sim.spatialProcess` to conditionally simulate the process, and then use these simulations to approximate the standard errors. In practice, a large `M` (e.g. 100) should be used to produce many simulations to reduce the variability when finding sample standard deviations in the Monte Carlo sample.

```r
sim <- sim.spatialProcess(obj, xp=make.surface.grid(CO.Grid)[1:5,], Z=CO.elevGrid$z[1:5], M=30)
look <- as.surface(CO.Grid, t(sim[1,]))
look2 <- as.surface(CO.Grid, t(sim[2,]))
look3 <- as.surface(CO.Grid, t(sim[3,]))
zgrid <- matrix(CO.elevGrid, nr=NGRID, nc=NGRID)
surf <-predictSurface.Krig( fit1E, grid.list=CO.Grid, ZGrid= zgrid, extrap=TRUE)

set.panel(2,2)
surface(surf, main="Model Prediction")
surface(look, main="Simulation 1")
surface(look2, main="Simulation 2")
surface(look3, main="Simulation 3")
```

Finally, we use these conditional simulations to approximate the standard errors.

```r
surSE <- apply(sim, 2, sd)
set.panel()
image.plot(as.surface( COGridPoints, surSE))
title("Uncertainty and Observations")
points(fit1E$x, col="magenta", pch=21, bg="white")
```

0.2 Univariate Tps

A final example shows how to fit a thin plate spline for one-dimensional smoothing and interpolation. We use the `WorldBankCO2` data included in `fields`, which is a 75×5 matrix with row names identifying countries. The five columns are * `GDP.cap`: Gross Domestic Product (in dollars) per capita
* `Pop.mid`: Percentage of population within ages of 15 to 65
* `Pop.urb`: Percentage of population living in an urban environment
* `CO2.cap`: Equivalent CO2 emissions per capita
* `Pop`: Total population.

We examine the relationship between the log of `Pop.mid` and the log of`CO2.cap`. We use the `fields` function `Tps` to fit a thin plate spline and also include a plot using `lm` to fit a linear regression. (Note: In this one-dimeonsional case, the `Tps` and the classical cubic smoothing spline are the same). The confidence intervals are shown as dashed red around the fit.

```r
data("WorldBankCO2")
pairs(WorldBankCO2)
```

```r
x <- log(WorldBankCO2[,'Pop.mid'])
y <- log(WorldBankCO2[,'CO2.cap'])

set.panel(1,2)
out <- Tps(x,y)
xgrid<- seq(  min( out$x), max( out$x),,100)
fhat<- predict( out,xgrid)
plot(x,y)
title('Tps')
```

```
lines( xgrid, fhat,)
SE<- predictSE( out, xgrid)
lines( xgrid,fhat + 1.96* SE, col="red", lty=2)
lines(xgrid, fhat - 1.96*SE, col="red", lty=2)

out2 <- lm(y~x)
ci <- predict(out2, data.frame(x=xgrid), level=0.95, interval='confidence')
matplot(xgrid, ci, col=c(1,2,2), type='l', lty=c(1,2,2),xlab="x",ylab="y")
title('lm')
points(x,y)
```

1 Univariate Splines

While the `fields` package is primarily concerned with methods for fitting spatial data and higher dimensional surfaces, it also includes functions for one dimensional curve fitting as special cases with $d = 1$. These functions can be used for interpolating and smoothing data, and allow for statistical inference of uncertainty.

- Thin plate splines: `Tps`, `fastTps`
- Cubic splines: `splint`, `sreg`

- Quantile/robust regression splines: `qsreg`, `QTps`

1.1 `splint` and `sreg`

We begin with a simple example that illustrates interpolating and smoothing a 1-d time series. Here we consider the `rat.diet` data set that comes with the package. The data includes the median food intake of two groups of rats over about 100 days. The treatment group `trt` received an appetite suppressant for 65 days and then was taken off the suppressant, while the control group `con` never received the suppressant.

```
# 1d example
data("rat.diet")
x <- rat.diet$t
y <- rat.diet$trt
matplot( rat.diet$t, cbind( rat.diet$trt, rat.diet$con),
         ylab="Median Food Intake", xlab="Days", pch=16)
title("Rat diet data: treatment and control groups")
```

We might be interested in fitting a model to these data in order to predict new values or to summarize the trend over time. The functions `splint` (spline interpolation) and `sreg` (spline regression) can be used to fit interpolating or smoothing cubic splines to univariate data. Both of these functions are FORTRAN based and faster than `Tps`. `sreg` will estimate the smoothing parameter `lambda`, and `splint` is designed to be a fast interpolator and requires a fixed `lambda` value. In either case, setting `lambda` equal to zero will give an interpolation.

Both `splint` and `sreg` can work with irregularly spaced data, but `splint` will not accept repeated values in the input variable `x`. (For repeated data, use `sreg`).

Note that the function `splint` requires the `xgrid` argument for locations to predict at. Alternatively, we can `predict` at new locations with an `sreg` object to get a similar result.

Basic Usage

splint(x, y, xgrid, derivative=0, lam=0, df=NA, lambda=NULL)
sreg(x, y, lambda = NA, df = NA)
predict(sregObject , x, derivative = 0)

Value

The `splint` function returns a vector of values of the interpolating spline evaluated at `xgrid`, while `sreg` returns a list of class `sreg`. Some of the relevant components of the `sreg` list are the original data `x` and `y`, the smoothness parameters `lambda` and `df` (same as `trace`), and the `residuals` and `fitted.values`. Note that the identical parameters `lam` and `lambda` both appear for covenience.

First, we'll use interpolation to fit the control group data. We use the `x` data coordinates as nodes to interpolate, and then evaluate the model at the grid locations to plot. Interpolation of a set of data points $(x_i, y_i)_{i=1}^{n}$ means that the interpolating function passes exactly through these points.

```r
grd <- seq(range(x)[1], range(x)[2], length.out = 400)
spl <- splint(x,y,grd)
plot(y~x, pch=".", cex=5, ylab="Median Food Intake", xlab="Days")
title("Interpolating Observed Data")
lines(grd,spl)
legend( x=0 , y=30, legend=c("Interpolation using splint"),
        col=c("black"), pch=15, cex=1)
```

Often we want to smooth the data rather than interpolate, for example if we assumed possible error in data collection. Loosely, a smoothing function follows the general "trend" of the data, but it may not exactly interpolate the data points.

The parameters `df` and `lambda` both control the smoothing in these functions. Interpolating the data corresponds to $\lambda = 0$ and `df=` the number of observations. As `lambda` increases, the spline gets smoother.

In place of `lambda`, a more useful scale is in terms of the effective number of parameters (degrees of freedom) associated with the estimate. We denote this by EDF (effective degrees of freedom). There is a 1-1 relationship between λ and EDF, and both are useful measures in slightly different contexts: λ for computing with the spatial process model versus EDF for data smoothing. The user should be aware that `splint` defaults to `lambda = 0` (interpolation), while `sreg` defaults to using GCV to find `lambda`.

```r
#Note this small lambda almost interpolates, close to splint model fit above
sr <- sreg(x,y, lambda=0.001)
pr <- predict(sr, grd)

#Fit a model with fewer effective degrees of freedom
spl2 <- splint(x,y, grd, df=20)
#Think of reducing df as increasing the smoothness of the model
spl3 <- sreg(x,y,df=15)
pr3 <- predict(spl3, grd)
spl4 <- sreg(x,y)
pr4 <- predict(spl4,grd)
plot(y~x, pch=".", cex=5, ylab="Median Food Intake", xlab="Days")
title("Smoothing Observed Data")
mat <- cbind( spl, pr, spl2, pr3, pr4)
matplot(grd,mat,col=2:6,lwd=2,type="l",lty=c(rep(1,4),2),add=TRUE)
legend( x=0 , y=30, legend=c("Interpolation","lambda = 0.001", "df=20", "df=15",
                              "GCV: lambda=11.13"), col=2:6, pch=15, cex=1)
```

1.2 sreg and S3 methods

We saw in the previous example that after we that we could use the general R function `predict` on an `sreg` object. This is known as an S3 method, and there are many other S3 methods available in `fields`. Examples of classes that use these methods are `sreg`, `Tps`, `Krig`, `spatialProcess`, `qsreg`, `mKrig`, `fastTps`, etc.

`predict` provides prediction estimates at arbitrary points/new data. Other values of the smoothing parameter λ can be input and the predictions are computed efficiently.

`summary` gives a summary of the object. The components include the function call, number of observations, effective degrees of freedom, residual degrees of freedom, root mean squared error, R-squared and adjusted R-squared, log10(lambda), cost, GCV minimum and a summary of the residuals. This list automatically prints in a useful format.

```
fit <- sreg(x,y) # fit a GCV spline to test group of rats
summary( fit)
```

```
## CALL:
## sreg(x = x, y = y)
##
##  Number of Observations:             39
##  Number of unique points:            39
##  Eff. degrees of freedom for spline: 7.4
##  Residual degrees of freedom:        31.6
##  GCV est. tau                        1.387
##  lambda                              11.13
##
## RESIDUAL SUMMARY:
##     min   1st Q  median   3rd Q     max
## -2.3140 -0.9767  0.2407  0.9652  2.3520
##
## DETAILS ON SMOOTHING PARAMETER:
##  Method used:      Cost:
##    lambda       trA       GCV   GCV.one GCV.model    tauHat
##    11.127     7.446     2.378     2.378        NA     1.387
##
##  Summary of estimates for lambda
##         lambda   trA   GCV tauHat converge
## GCV      11.13 7.446 2.378  1.387        5
## GCV.one  11.13 7.446 2.378  1.387        5
```

`plot` gives a series of four diagnostic plots describing the fit. For `sreg`, plot 1 shows data vs. predicted values, and plot 2 shows predicted values vs. residuals. Plot 3 shows the criteria to select the smoothing parameter $\lambda = \sigma^2/\rho$. The x axis has transformed λ in terms of effective degrees of freedom to make it easier to interpret. Note that here the GCV function is minimized while the REML is maximized. Finally, plot 4 shows a histogram of the standard residuals.

```
set.panel(2,2)
plot(fit) # four diagnostic plots of fit
```

Finally, we will illustrate the flexibility in a evaluating the fitted function and its derivatives.

```
predict( fit) # predicted values at data points
```

```
##  [1] 18.94781 18.66388 18.11058 17.09987 16.88134 16.49334 15.86704 15.74178 15.52438 15.20371 15.14
```

```
xg <- seq(0,110,,50)
sm <-predict( fit, xg) # spline fit at 50 equally spaced points
```

```r
der.sm <- predict( fit, xg, deriv=1) # derivative of spline fit
```

```r
set.panel( 1,2)
plot( fit$x, fit$y) # the data
lines( xg, sm) # the spline
title("The Data and Spline Fit")
plot( xg,der.sm, type="l") # plot of estimated derivative
title("Predicted Derivative")
```

To get proper standard errors on our predictions, we recommend switching to the thin plate spline function.

1.3 Tps

The `Tps` function is used for fitting curves and surfaces to interpolate or smooth data by thin plate spline regression. This is a very useful technique and is often as informative as more complex methods.

The assumed model for Tps is additive. Namely,

$$Y_i = f(\mathbf{X}_i) + \epsilon_i,$$

where $f(\mathbf{X})$ is a `d` dimensional surface, and the object is to fit a thin plate spline surface to irregularly spaced data. ϵ_i are uncorrelated random errors with zero mean and variances σ^2/w_i.

This function also works for just a single dimension and is a special case of a spatial process estimate (Kriging). A "fast" version of this function uses a compactly supported Wendland covariance, computing the estimate for a fixed smoothing parameter.

Basic Usage

Tps(x, Y, m=NULL, p=NULL,)
fastTps(x, Y, m = NULL, p = NULL, aRange, lambda=0)

Value

Returns a list of class `Krig/mKrig` (see below), which includes the predicted surface of `fitted.values` and `residuals`. Also includes the results of the GCV grid search in `gcv.grid`. Note that the GCV/REML optimization is done even if `lambda` or `df` is given. Please see the documentation on `Krig` for details of the returned arguments.

`Tps` is a "wrapper function", which is why a `Krig` object is returned. Moreover, any argument that can be used in a call to `Krig` can be used in `Tps`. For example, the user can specify `lambda` or `df` just as in `sreg`. As seen in the minimization problem, `lambda` determines the weight put on the smoothness condition, giving it the same interpretation. `lambda` is the most important argument for `Tps`, which is estimated by GCV if omitted. We can also includes covariates in the linear part of the model by passing the argument `Z`.

`Tps` and `fastTps` are special cases of using the `Krig` and `mKrig` functions, respectively. The `Tps` estimator is implemented by passing the right generalized covariance function based on a radial basis function (RBF) to the more general function `Krig`. One advantage of this implementation is that once a `Tps/Krig` object is created the estimator can be found rapidly for other data and smoothing parameters provided the locations remain unchanged. This makes simulation within R efficient (see example below).

1.4 Tps theory and GCV

This section is more mathematical and offered as a supplement. The estimator of $f(\mathbf{X})$ is the minimizer of the penalized sum of squares

$$\frac{1}{n}RSS + J_m(f),$$

where RSS is a weighted residual sums of squares $\sum_i w_i(y_i - f(x_i))^2$, and J_m is a roughness penalty based on mth order derivatives and a scalar $\lambda > 0$. For `Tps`, f is seperated into a $d-1$ degree polynomial and a smooth function capturing spatial dependence

More specifically, thin plate splines can be viewed as the result of a minimization problem in a reproducting kernel Hilbert space $\mathcal{H}$. We want to minimize the residual sum of squares subject to the constraint that the function has a certain level of smoothness. The smoothness is quantified by the integral of squared mth order derivatives of the function, and this integral gives the norm of the Hilbert space. Besides controlling the order of the derivatives, the value of m also determines the base polynomial that is fit to the data.

The minimization problem written out for the case d=1 and m=2 gives the cubic spline smoothing solution:

$$\min_{f\in\mathcal{H}} \left\{ \sum_i (y_i - f(x_i))^2 + \lambda \int \left[f''(x)\right]^2 dx \right\}$$

For the case d=2 and m=2:

$$\min_{f\in\mathcal{H}} \left\{ \sum_i (y_i - f(\mathbf{x}_i))^2 + \lambda \int \left(\frac{\partial^2 f(\mathbf{x})}{\partial x_1^2}\right)^2 + 2\left(\frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2}\right)^2 + \left(\frac{\partial^2 f(\mathbf{x})}{\partial x_2^2}\right)^2 d\mathbf{x} \right\}$$

where $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. The smoothing parameter λ can be chosen from the data by GCV. That is, the estimate of the smoothing parameter can be found by minimizing the GCV function

$$V(\lambda) = \frac{1}{n}\frac{RSS(\lambda)}{\left(1 - \frac{EDF(\lambda)}{n}\right)^2}$$

It is also possible to include a cost parameter that can give more (or less) weight to the effective number of parameters beyond the base polynomial model. Note that a frequentist estimate for the residual variance σ^2 is found using the estimate for λ by

$$\hat{\sigma}^2 = \frac{RSS}{(n - EDF(\lambda))}$$

in analogy to ordinary least squares regression.

##Confidence intervals with `Tps`

First we show that we can use `Tps` to replicate the `sreg` function if we specify arguments in a certain way. `sreg` does not scale the observations when fitting, so we must instruct `Tps` not to scale either. This will make `lambda` comparable within a factor of n.

```r
# Using Tps on the rat.diet data
# Tps allows uncertainty quantification
fit.tps<-Tps( x,y, scale="unscaled")
summary( fit.tps)
```

```
## CALL:
## Tps(x = x, Y = y, scale.type = "unscaled")
##
##  Number of Observations:                39
##  Number of unique points:               39
##  Number of parameters in the null space 2
##  Parameters for fixed spatial drift     2
##  Effective degrees of freedom:          7.5
##  Residual degrees of freedom:           31.5
##  MLE tau                                1.321
##  GCV tau                                1.387
##  MLE sigma                              0.004056
##  Scale passed for covariance (sigma)    <NA>
##  Scale passed for nugget (tau^2)        <NA>
##  Smoothing parameter lambda             430
##
## Residual Summary:
##     min   1st Q  median   3rd Q     max
## -2.3150 -0.9753  0.2420  0.9654  2.3500
##
## Covariance Model: Rad.cov
##   Names of non-default covariance arguments:
##        p
##
## DETAILS ON SMOOTHING PARAMETER:
##  Method used:   GCV    Cost:  1
##    lambda       trA       GCV   GCV.one GCV.model    tauHat
##   430.041     7.460     2.378     2.378        NA     1.387
##
##  Summary of all estimates found for lambda
##            lambda   trA   GCV tauHat -lnLike Prof converge
## GCV           430 7.460 2.378  1.387        71.07        2
## GCV.model      NA    NA    NA     NA           NA       NA
## GCV.one       430 7.460 2.378  1.387           NA        2
## RMSE           NA    NA    NA     NA           NA       NA
## pure error     NA    NA    NA     NA           NA       NA
## REML         1004 6.236 2.404  1.421        70.77        6
```

Notice how similar the summary is to `sreg`'s summary. `sreg` is actually a special case of `Tps` (as shown below – note that `lambda` is not equal for the two functions!) The `m=2` default for `Tps` and leaving the data unscaled results in the cubic smoothing spline `sreg`.

```
# compare sreg and Tps results to show the adjustment to lambda.
predict( fit)-> look
predict( fit.tps, lambda=fit$lambda*fit$N)-> look2
# test.for.zero is a testing function that checks for equality within tolerance
# silence means it checks to 1e-8
test.for.zero( look, look2)
```

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

We can easily get uncertainty intervals from the `Tps` function.

```
SE <- predictSE(fit.tps)

# 95% pointwise prediction intervals
```

```r
Zvalue<- qnorm(.0975)
upper<- fit.tps$fitted.values + Zvalue* SE
lower<- fit.tps$fitted.values - Zvalue* SE

# conservative, simultaneous Bonferroni bounds
ZBvalue<- qnorm(1- .025/fit$N)
upperB<- fit.tps$fitted.values + ZBvalue* SE
lowerB<- fit.tps$fitted.values - ZBvalue* SE

plot( fit.tps$x, fit.tps$y)
lines( fit.tps$predicted, lwd=2)
matlines( fit.tps$x,
          cbind( lower, upper, lowerB, upperB), type="l",
          col=c( 2,2,4,4), lty=1)
title( "95 pct pointwise and simultaneous intervals")
```

```r
# or try the more visually honest:
plot( fit.tps$x, fit.tps$y)
lines( fit.tps$predicted, lwd=2)
segments( fit.tps$x, lowerB, fit.tps$x, upperB, col=4)
segments( fit.tps$x, lower, fit.tps$x, upper, col=2, lwd=2)
title( "95 pct pointwise and simultaneous intervals")
```

<Finally, we are able to use the uncertainty estimates from the `Tps` model to produce conditional simulations.>

Using the `WorldBankCO2` data, we show a small example using `fastTps`. We plot the data, the mean prediction line, and some approximate confidence intervals in dashed red.

```r
x <- WorldBankCO2[,'Pop.urb']
y <- log10(WorldBankCO2[,'CO2.cap'])
out.fast <- fastTps(x,y,lambda=2, aRange=20)
plot(x,y)
xgrid<- seq(  min(x), max(x),,300)
fhat.fast <- predict( out.fast,xgrid)
#se.fast <- predictSE( out.fast)
lines( xgrid, fhat.fast)
#lines( xgrid, fhat.fast+1.96*se.fast, col=2, lty=2)
#lines( xgrid, fhat.fast-1.96*se.fast, col=2, lty=2)
title('fastTps fit')
```

#Introduction to Spatial Statistics

Here we'll cover the fundamentals of spatial statistics. In particular, we will examine the basic definitions and concepts of the field before giving a geostatistics mini-course. As this is a vignette, we can only scratch the theory behind spatial statistics. A few useful resources are:

- *Statistics for Spatial Data* by Noel Cressie
- *Handbook of Spatial Statistics* by Alan E. Gelfand, Peter J. Diggle, Montserrat Fuentes, and Peter Guttorp
- *Statistics for Spatio-Temporal Data* by Noel Cressie and Christopher K. Wikle

1.5 The spatial problem

Let's return to a previous `quilt.plot`. Suppose we want to predict the maximum March/April/May temperature at a new location $(-103, 39.5)$ in Colorado, which is shown below as the `x`.

An initial thought may be to use OLS regression with covariates of `longitude` and `latitude`.

Recall Tobler's First Law of Geography: "Everything is related to everything else, but near things are more related than distant things." We see this reflected in the plot above, as temperatures in one area are similar to the temperature in nearby areas. This spatial autocorrelation violates many typical statistical assumptions – our observations are no longer independent! Fortunately, in the field of spatial statistics, we can relax our assumption of independence and work with observations that are spatially dependent.

To make a "spatial prediction", we need some way to quantify the change in our observation variable (temperature) as a function of distance.

1.6 Covariance

A covariance function $\text{Cov}(Y(\mathbf{x}), Y(\mathbf{x}'))$ describes the joint variability between a stochastic process $Y(\cdot)$ at two locations $\mathbf{x}$ and $\mathbf{x}'$. This covariance function is vital in spatial prediction. The `fields` package includes common parametric covariance families (e.g. exponential and Matern) as well as nonparametric models (e.g. radial and tensor basis functions).

When modeling $\text{Cov}(Y(\mathbf{x}), Y(\mathbf{x}'))$, we are often forced make simplifying assumptions.

- Stationarity assumes we can represent the covariance function as

$$\text{Cov}(Y(\mathbf{x}+\mathbf{h}), Y(\mathbf{x})) = C(\mathbf{h})$$

  for some function $C : \mathbb{R}^d \to \mathbb{R}$ where $\dim(\mathbf{x}) = d$.

- Isotropy assumes we can represent the covariance function as

$$\text{Cov}(Y(\mathbf{x}+\mathbf{h}), Y(\mathbf{x})) = C(\|\mathbf{h}\|)$$

  for some function $C : \mathbb{R} \to \mathbb{R}$, where $\|\cdot\|$ is a vector norm.

The most common covariance functions are the exponential and Matern. These parameterized covariances are functions of distance, and return a covariance corresponding to that distance based on the response. Each is isotropic, meaning that $\text{Cov}(Y(\mathbf{x}), Y(\mathbf{x}'))$ *only* depends on the distance $r := \|\mathbf{x} - \mathbf{x}'\|$. The exact formulas are provided below.

- Exponential. $\text{Cov}(Y(\mathbf{x}), Y(\mathbf{x}')) = C(r) = \rho e^{-r/a} + \sigma^2 \mathbf{1}_{\mathbf{x}=\mathbf{x}'}$.
- Matern. $\text{Cov}(Y(\mathbf{x}), Y(\mathbf{x}')) = C(r) = \rho\left(\frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{r}{a}\right)^{\nu} K_{\nu}\left(\frac{r}{a}\right)\right) + \sigma^2 \mathbf{1}_{\mathbf{x}=\mathbf{x}'}$, where K_ν is a modified Bessel function of the second kind, of order ν.

Note that the Matern covariance function depends on parameters (ρ, a, ν, σ^2), and the Exponential covariance depends on parameters (ρ, a, σ^2). The parameters ρ, a, σ^2 and ν respectively denote the marginal variance (or sill), range, nugget effect, and smoothness of our process.

The range a of the process is the distance at which observations become uncorrelated. The sill ρ is the marginal variance of the spatial process. The nugget effect σ^2 corresponds to small-scale variation such as measurement error. The smoothness ν corresponds to how "smooth" our spatial process appears. An illustration in the `vgram` section shows the visual interpretation of these parameters.

For such isotropic covariances, the `fields` packages uses relies on the function `rdist`, which is used to calculate the distance between two sets of locations.

1.6.1 `rdist`

`rdist` is a function that computes pairwise distance given one or two vectors of locations, say $\mathbf{x}_1$ and $\mathbf{x}_2$.

<The function `rdist.vec` calculates the pairwise distances between vectors. >

For applications, `rdist.earth` computes the geographic distance (Great Circle distance) between two input matrices, and the syntax is similar to that of `rdist`.

If `x2=NULL`, then `rdist(x1,x1)` is performed.

Basic Usage

rdist(x1, x2 = NULL)
< > rdist.vec(x1, x2) >

Value

The result of `rdist` is a matrix whose i, j-th entry is the Euclidean distance between the i-th element of $\mathbf{x}_1$ and the j-th element of $\mathbf{x}_2$. < The result of `rdist.vec` is a vector whose i-th entry is the Euclidean distance between the i-th element of $\mathbf{x}_1$ and the i-th element of $\mathbf{x}_2$. >

Here are two examples of `rdist` (in 1D and 2D).

```
#compute pairwise distance vector
x1 = 1:3    #x1 = (1,2,3)
x2 = 3:1    #x2 = (3,2,1)
rdist(x1,x2)
```

```
##      [,1] [,2] [,3]
## [1,]    2    1    0
## [2,]    1    0    1
## [3,]    0    1    2
```

```
data(COmonthlyMet)
y.CO <- CO.tmax.MAM.climate
z.CO <- CO.elev
grd.CO <- as.matrix(CO.loc)
keep <- !is.na(y.CO)
y.CO <- y.CO[keep]
z.CO <- z.CO[keep]
grd.CO <- grd.CO[keep,]

quilt.plot(grd.CO,y.CO)
US(add=TRUE)
```

It is easy two find the distance between two points on the `quilt.plot`. Here, we use `rdist.earth` since our vectors consist of longitude/latitude pairs.

```
three.locations <- grd.CO[1:3,]
#location one is (-109.10, 36.90)
#location two is (-103.17, 40.12)
#location three is (-105.85, 37.43)
rdist.earth(three.locations)  #i,j entry is the distance between location i and location j
```

```
##            347      1012     1014
## 347    0.0000 390.5456 182.8560
## 1012 390.5456   0.0000 235.5842
## 1014 182.8560 235.5842   0.0000
```

1.6.2 Exponential and Matern covariances

The `Exponential` and `Matern` functions produce these isotropic covariances in R. The inputs are a matrix of distances `d`, either the range parameter `range` (this is a in our notation) or `alpha = 1/range`, and finally the marginal variance `phi` (which is ρ in our notation), and the smoothness `nu` for the Matern covariance. Note that the exponential covariance is identical to the Matern covariance with smoothnes `nu = 0.5`.

There are other (generalized) covariances in `fields`, such as `RadialBasis` and `Wendland`, that will be discussed later. To set the nugget variance σ^2, we use the `sigma2` argument in the functions `Krig` or `mKrig` that we also discuss later.

Below are plots that illustrate the shape of these covariance functions. The Exponential, Matern, and Wendland correlations (marginal variances $\rho = 1$) behave exactly how one would expect. That is, the covariance between two nearby objects is close to 1, indicating a strong positive correlation among the two observations. The appearance of the Radial-Basis correlation may be shocking – we will return to it later.

Basic Usage

Exponential(d, range=1, alpha=1/range, phi=1.0)
Matern(d , range=1, alpha=1/range, smoothness = 0.5, nu=smoothness, phi=1.0)

Value

The result is a matrix of covariances with the same dimension as the input distance matrix `d`.

```
d <- seq(0,10,,200) #Will visualize a 1D set of images
e <- Exponential(d, range = 1) #alpha = 1/range, phi=1.0
m <- Matern(d , range = 1.5, smoothness = 1.5) #alpha=1/range, nu=smoothness, phi=1.0)
rbf <- RadialBasis(d,M=2,dimension=2, derivative = 0)
w <- Wendland(d, aRange=1, dimension=1, k=2)
w2 <- Wendland(d, aRange=5, dimension=1, k=5)
dat <- cbind(e,m,rbf,w,w2)
matplot(d, dat, type = c("l"), lwd=2, ylim=c(0,1), xlab="Distance",
        ylab="Correlation",col = c("blue","red", "green", "orange", "purple"))
title("Correlation (covariance) functions")
legend(x=6, y=0.9, legend=c("Exp", "Matern", "RBF", "Wendland (aRange=1, k=2)", "Wendland (aRange=5, k=
       col=c("blue", "red", "green", "orange", "purple"), pch=15,cex=0.5)
```

1.6.3 Matern.cor.to.range

The range parameter means something different for different values of smoothness in the Matern family. To compare the spatial dependence across different covariance functions, it is helpful to quantify them in a similar way.

For a given smoothness `nu`, `Matern.cor.to.range` returns the range at which the Matern correlation function drops below numeric `cor.target`.

Basic Usage

Matern.cor.to.range(d, nu, cor.target=.5)

Value

The result is the range value `aRange` at which `Matern( d, range=aRange, nu=nu) == cor.target`.

The example following shows how much the range parameter by itself can vary for different smoothness parameters.

```
r1<- Matern.cor.to.range( 10, nu=.5, cor.target=.1)
r2<- Matern.cor.to.range( 10, nu=1.0, cor.target=.1)
r3<- Matern.cor.to.range( 10, nu=2.0, cor.target=.1)
# note that these equivalent ranges with respect to this correlation length are
# quite different due the different smoothness parameters.
d<- seq( 0, 15,,200)
y<- cbind( Matern( d, range=r1, nu=.5), Matern( d, range=r2, nu=1.0), Matern( d, range=r3, nu=2.0))
matplot( d, y, type="l", lty=1, lwd=2)
xline( 10)
yline( .1)
legend( x=6, y=0.9, legend=c("Range = r1", "Range = r2", "Range = r3"),col=c("black", "red", "green"),
```

1.6.4 `Exp.cov` and `stationary.cov`

The functions `Exp.cov` and `stationary.cov` are similar to the previous covariance functions mentioned, but they take two sets of locations as inputs rather than a precomputed distance matrix. This form is what is required for the spatial methods in `fields`.

Notation is largely the same as with `Matern` and `Exponential`; of course, `aRange` controls the range of the covariance model, while `smoothness` can be used only with the Matern. The `Exp.cov` function includes an additional argument `p`, where `p=1` corresponds to the typical exponential covariance and `p=2` corresponds to the Gaussian covariance. Both `Exp.cov` and `stationary.cov` have the input `C`. If `C` is specified, then the resulting covariance matrix is multiplied by `C`. This feature offers the potential for more efficient memory because the entire covariance matrix need not be created at once to accomplish the matrix multiplication.

`stationary.cov` is essentially a wrapper function. One can input `Matern` or `Exponential` in the `Covariance=` argument, as well an argument `Distance`, which takes character strings `rdist` or `rdist.earth`.

These functions can take advantage of sparsity in the covariance matrix and are also capable of taking a precomputed distance matrix as an argument to save repeated computation. This feature, while largely hidden from the user, often provides added speed to the statistical computations in `fields`.

Basic Usage

Exp.cov(x1, x2=NULL, aRange = 1, p=1)
stationary.cov(x1, x2=NULL, Covariance = “Exponential”, Distance = “rdist”, aRange = 1)

Value

If the argument `C` is `NULL`, then the (cross-)covariance matrix is returned. If `C` is a vector of length n, then returned value is the multiplication of the (cross-)covariance matrix with this vector. The `C` functionality is used internally by all **fields** Kriging-related functions.

In the following example we demonstrate that we can specify a particular covariance in several ways, and they all end up being equivalent.

```
x1 <- seq(1,10 ,,100)
x2 <- seq(1,20,,100)
MyaRange <- 5

cov1 <- Exp.cov(x1,x2, aRange=MyaRange)
cov2 <- stationary.cov( x1,x2, aRange=MyaRange, Distance= "rdist", Covariance="Exponential")
cov3 <- stationary.cov( x1,x2, aRange=MyaRange, Distance= "rdist", Covariance="Matern",smoothness=.5)

{test.for.zero(cov1,cov2)
test.for.zero(cov2,cov3)}
```

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

1.6.5 Simulating Random Fields (`sim.rf` and `image.cov`)

To gain an intuition of how the smoothness `nu` affects a stochastic process, we need to create random fields and visualize them.

Suppose $Y(\cdot)$ is a mean zero Gaussian process with covariance $\mathrm{Cov}(\cdot,\cdot)$. To simulate $Y(\cdot)$ at locations $\mathbf{x}_1,\ldots,\mathbf{x}_n$, perform the following steps:

- Form the covariance matrix $\Sigma = (\mathrm{Cov}(\mathbf{x}_i,\mathbf{x}_j))_{i,j=1}^{n}$.
- Take the Cholesky Decomposition $\Sigma = LL^T$.
- Multiply $L\boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim N(\mathbf{0}, I)$.

The resulting random vector $L\boldsymbol{\epsilon}$ is an exact simulation of a Gaussian process. The only problem with this approach is the computations and storage grow rapidly for larger grids. For example, a 128×128 image would mean that the dimension of Σ is huge $(16,000 \times 16,000)$ and effectively prohibit the use of the Cholesky decomposition. For rectangular grids, one can sometimes simulate Gaussian random fields very efficiently using an alternative algorithm called "circulant embedding" that is based on the Fast Fourier Transform. The restrictions are that the covariance function needs to be stationary and the correlation range needs to the small relative to the size of the domain (see Chan and Wood, 1994).

The `sim.rf` function uses circulant embedding to simulate a stationary Gaussian random field (GRF) on a regular grid with unit marginal variance (i.e. $\rho = 1$). Note that the marginal variance is readily changed by scaling the resulting GRF.

A limitation when using `sim.rf` presents itself with `Error in sim.rf(obj) : FFT of covariance has negative values`. This comes from the circulant embedding method used to create the GRF; in short, it occurs when the correlation range is too large. One fix is to increase the domain size so this correlation is then small relative to the size of the domain.

The input of `sim.rf` is a list that includes information about the covariance function, its FFT, and the grid for evaluation. Usually this is created by a setup call `image.cov` (i.e. `Exp.image.cov`, `matern.image.cov`, `stationary.image.cov`).

Basic Usage

sim.rf(obj)
matern.image.cov(ind1, ind2, Y, cov.obj = NULL, setup = FALSE, grid ,aRange= 1.0, smoothness=.5)
Exp.image.cov(ind1, ind2, Y, cov.obj = NULL, setup = FALSE, grid, ...)

Value

A matrix with the random field values.

```
grid <- list( x= seq( 0,20,,100), y= seq(0,20,,100))
obj1 <- matern.image.cov(grid=grid, aRange = 0.5 , smoothness = 0.5, setup=TRUE)
obj2 <- matern.image.cov(grid=grid, aRange = 0.5, smoothness = 1, setup=TRUE)
obj3 <- matern.image.cov(grid=grid, aRange = 0.5, smoothness = 2, setup=TRUE)
obj4 <- matern.image.cov(grid=grid, aRange = 0.5, smoothness = 2.5, setup=TRUE)

set.seed(2008)

look1 <- sim.rf(obj1)
look2 <- sim.rf(obj2)
look3 <- sim.rf(obj3)
look4 <- sim.rf(obj4)

set.panel(2,2)
image.plot(grid$x, grid$y, look1, main='Smoothness = 0.5',zlim=c(-4.25,4.25))
image.plot(grid$x, grid$y, look2, main='Smoothness = 1',zlim=c(-4.25,4.25))
image.plot(grid$x, grid$y, look3, main='Smoothness = 2',zlim=c(-4.25,4.25))
image.plot(grid$x, grid$y, look4, main='Smoothness = 2.5',zlim=c(-4.25,4.25))
```

1.7 Variograms

Given a set of spatial data, we need some tools to investigate the covariance of the underlying spatial process. The variogram describes correlation over distance, which can be useful when looking at isotropic covariances.

We define the theoretical variogram of a stationary process $Y(\cdot)$ as $\gamma(\mathbf{h}) = \frac{1}{2}\,\mathrm{Var}[Y(\mathbf{s}+\mathbf{h}) - Y(\mathbf{s})]$, where $\mathbf{h} \in \mathbb{R}^d$. Assuming $Y(\cdot)$ has covariance $C(\cdot)$, it is easy to verify that $\gamma(\mathbf{h}) = C(\mathbf{0}) - C(\mathbf{h})$. Therefore, $\gamma(\mathbf{h})$ is simply a shift and reflection of the covariance function $C(\mathbf{h})$.

In the context of variograms, the spatial parameters (ρ, a, ν, σ^2) have a straightforward visual interpretation.

The definition is readily translated to an *empirical* variogram. Let $N(h)$ be the set of pairs of observations y_i, y_j such that $\|\mathbf{x}_i - \mathbf{x}_j\| = h$. The empirical variogram (`vgram`) is defined as

$$\widehat{\gamma}(h) = \frac{1}{2 \cdot |N(h)|} \sum_{(i,j) \in N(h)} (y_i - y_j)^2$$

The default "cloud" variogram is done for each distance h, but it is noisy and hard to interpret. Often, we specify a number of bins `N` in which values for nearby values are averaged. In this way, the choices for a variogram may seem similar to finding a histogram. The `breaks` argument allows the user to explicitly provide distances at which bins are created.

`crossCoVGram` is the same as `vgram` but differences are taken across different variables rather than the same variable. `boxplotVGram` uses the base R boxplot function to display the variogram neatly.

Basic Usage

vgram(loc, y, d = NULL, lon.lat = FALSE, N = NULL, breaks = NULL)
boxplotVGram(x, N=10, breaks = pretty(x$d, N, eps.correct = 1))
vgram.matrix(dat, R)

Value

`vgram` and `crossCoVGram` return a `vgram` object containing the following values:

- `vgram`: Variogram or covariogram values
- `d`: Pairwise distances
- `call`: Calling string
- `stats`: Matrix of statistics for values in each bin. Rows are the summaries returned by the stats function or describe. If not either breaks or N arguments are not supplied then this component is not computed.
- `centers`: Bin centers.

`vgram.matrix` returns an object of class vgram.matrix with the following components: `d`, a vector of distances for the differences, and `vgram`, the variogram values. This is the traditional variogram ignoring direction.

Note that `vgram.matrix` also has components:

- `d.full`: a vector of distances for all possible shifts up distance `R`
- `ind`: a two column matrix giving the x and y increment used to compute the shifts
- `vgram.full`: the variogram at each of these separations
- `vgram.robust`: Cressie's version of a robust variogram statistic.

```
v <- vgram(loc=grd.CO,y=y.CO,N=30,lon.lat=TRUE) #use 30 bins
plot(v$d, v$vgram, ylab="sqrt(Variance)", xlab="distance")
lines(v$stats["mean",]~v$centers,main='Binned Variogram',col='red',ylab="sqrt(Variance)", xlab="distanc
```

```
plot(v$stats["mean",]~v$centers,main='Binned Variogram',ylab="sqrt(Variance)", xlab="distance")     #red
```

```
boxplotVGram(v,y.CO,N=30,breaks=v$breaks,lon.lat=TRUE, ylab="sqrt(Variance)", xlab="distance")
```

Finally, we show how to use `vgram.matrix` with a synthetic field `look1` generated in the previous section. We look for neighbors within a radius of `R=5`.

```
vm <- vgram.matrix(look1,R=5)
plot(vm$d, vm$vgram, ylab="sqrt(Variance)", xlab="distance")
```

1.8 Simple Kriging

The additive spatial model is

$$Y(\mathbf{x}) = \mathbf{Z}(\mathbf{x})\mathbf{d} + g(\mathbf{x}) + \epsilon(\mathbf{x}),$$

where $Y(\cdot)$ is an observation, $\mathbf{Zd}$ is a deterministic (nonrandom) product of covariates $\mathbf{Z}$ with a weight vector $\mathbf{d}$ that acts as a mean function, $g(\cdot) \sim N(\mathbf{0}, \rho\mathbf{K})$ is spatial process, and $\epsilon(\cdot) \sim N(\mathbf{0}, \sigma^2\mathbf{I})$ is error (i.e. white noise).

Thus, we assume that our observations $Y(\cdot)$ are Gaussian; in particular, $\mathbf{Y} \sim N(\mathbf{Zd}, \rho\mathbf{K} + \sigma^2\mathbf{I})$

For now, we'll assume $\mathbf{Zd} \equiv \mathbf{0}$ and focus on the spatial aspect $g(\mathbf{x})$. This is known as *simple* kriging, where the mean of the stochastic process is known. Denote our locations of observed points as $\mathbf{x}_1, \dots, \mathbf{x}_n$.

Suppose we want to predict $Y(\cdot)$ at a new location $\mathbf{x}_0$. The kriging estimate $\hat{Y}(\mathbf{x}_0)$ of our mean zero Gaussian process $Y(\cdot)$ is given by:

$$\hat{Y}(\mathbf{x}_0) = \Sigma_0 \Sigma^{-1} \mathbf{Y}$$

- $\mathbf{Y} = \begin{pmatrix} Y(\mathbf{x}_1) \\ \vdots \\ Y(\mathbf{x}_n) \end{pmatrix}$ is of dimension $n \times 1$,
- $\Sigma_0 = (\mathrm{Cov}[Y(\mathbf{x}_0), Y(\mathbf{x}_1)], \mathrm{Cov}[Y(\mathbf{x}_0), Y(\mathbf{x}_2)], \dots, \mathrm{Cov}[Y(\mathbf{x}_0), Y(\mathbf{x}_n)])$ is of dimension $1 \times n$,
- $\Sigma = (\mathrm{Cov}[Y(\mathbf{x}_i), Y(\mathbf{x}_j)])_{i,j=1}^n$ is the $n \times n$ covariance matrix of $Z(\cdot)$ with ij-entry equal to $\mathrm{Cov}[Y(\mathbf{x}_i), Y(\mathbf{x}_j)]$.

Briefly, when the covariance parameters are known, the Kriging estimate represents the best prediction at $\mathbf{x}_0$ based on a linear combination of the observations. By best we mean in terms of mean squared error.

In the next section, we see an example of how to perform simple kriging. Other types of kriging (i.e. ordinary kriging and universal kriging) are explained in the appendix.

2 Traditional Geostatistics

In this chapter, we begin with an example of a simple Kriging prediction with covariance parameters estimated from a variogram. Next, we show why one should be cautious using a variogram to estimate parameters, and finally we recommend alternatives to variogram fitting.

##Fitting a Variogram

Consider the following spatial data: we have measurements of the daily maximum temperature in Colorado during March, April, and May (MAM) at several hundred locations, along with the elevation at each longitude/latitude pair. We want to predict the temperature value at $\mathbf{x}_0 = (-103, 39.5)$.

```r
data(COmonthlyMet)
y.CO <- CO.tmax.MAM.climate
z.CO <- CO.elev
grd.CO <- as.matrix(CO.loc)
keep <- !is.na(y.CO)       #removing all NA's from the data
y.CO <- y.CO[keep]
z.CO <- z.CO[keep]
grd.CO <- grd.CO[keep,]
quilt.plot(grd.CO,y.CO)
US(add=TRUE)
points(-103,39.5, pch=4,lwd=3,cex=1.25, col='black')
```

For now, we'll assume a zero mean ($\mathbf{Zd} \equiv \mathbf{0}$) and focus on the spatial aspect, which is clearly an incorrect assumption. One can remove the mean trend using least squares, but then this requires universal Kriging instead of simple Kriging. We prefer to start with simple Kriging in this exposition. Details for universal Kriging can be done in the appendix. Denote our locations of observed points as $\mathbf{x}_1, \dots, \mathbf{x}_n$. In this example, $n = 213$ since we have 213 temperature observations.

Suppose we want to predict the temperature $Y(\cdot)$ at a new location $\mathbf{x}_0$. Recall that the simple Kriging estimate $\hat{Y}(\mathbf{x}_0)$ of our mean zero Gaussian process $Y(\cdot)$ is given by:

$$\hat{Y}(\mathbf{x}_0) = \Sigma_0 \Sigma^{-1} \mathbf{Y}$$

Now, the question is how to determine the covariance function of $Y(\cdot)$. For this example, we'll consider the (isotropic) exponential covariance: $\text{Cov}[Y(\mathbf{x}), Y(\mathbf{x}')] = \rho e^{-\|\mathbf{x}-\mathbf{x}'\|/a}$. Here, \$a \$ is a range parameter that corresponds to the length of spatial autocorrelation of our process, and ρ is the overall variance of the process.

The covariance matrix Σ thus has i, j-th entry

$$\Sigma_{i,j} = e^{-\|\mathbf{x}_i-\mathbf{x}_j\|/a} + \sigma^2 \mathbf{1}_{\{\mathbf{x}_i=\mathbf{x}_j\}},$$

where $\mathbf{1}_{\{\mathbf{x}_i=\mathbf{x}_j\}} = \begin{cases} 1 & \mathbf{x}_i = \mathbf{x}_j \\ 0 & \text{else} \end{cases}$

Earlier, we defined the variogram, which measures how our data $Y(\mathbf{x})$ is related over distance. To find paramters (a, ρ, σ^2) for the covariance function, a common geostatistical approach is to fit a curve to the (binned) semivariogram. Since we are working with longitudinal and latitudinal data, we use the option `lon.lat=TRUE`.

```
my.vgram <- vgram(loc=grd.CO,y=y.CO,N=30,lon.lat=TRUE) #use 30 bins
plot(my.vgram$stats["mean",]~my.vgram$centers,main='Binned Semivariogram')
```

Often, only the first part of a variogram is an accurate representation of an isotropic covariance function (see the next section for an illustration). Accordingly, we only fit a polynomial to the first half of the curve.

```
plot(my.vgram$stats["mean",1:15]~my.vgram$centers[1:15],main='Binned Semivariogram')
```

Next, we fit parameters to the above semivariogram using least squares and `L-BFGS` optimization, an alternative to fitting by eye. The exponential variogram formula is as follows:

$$\gamma(r) = \begin{cases} \rho\left(1 - e^{-r/a}\right) + \sigma^2 & r > 0 \\ 0 & r = 0 \end{cases}$$

The Matern semivariogram (with smoothing parameter ν) is:

$$\gamma(r) = \begin{cases} \rho\left(1 - \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{r}{a}\right)^{\nu} K_\nu\left(\frac{r}{a}\right)\right) + \sigma^2 & r > 0 \\ 0 & r = 0 \end{cases}$$

In each case, we minimize least squares to find estimates of the covariance parameters.

```
#Exponential plot
ls.exponential <- function(par){ # par = (rho,aRange,sigma^2)
    theoretical.vgram <- par[1] * (1 - exp(-my.vgram$centers[1:15] / par[2])) + par[3]
    sum( (theoretical.vgram - my.vgram$stats["mean",1:15])^2 ) }

##providing initial guesses and bounds
out.exponential <- optim(par=c(2,50,1),fn=ls.exponential,method="L-BFGS", lower = c(0,0,0), upper = c(1

rho.exponential <- out.exponential$par[1]
aRange.exponential <- out.exponential$par[2]
sigma2.exponential <- out.exponential$par[3]

plot(my.vgram$stats["mean",1:15]~my.vgram$centers[1:15], main="Exponential vs. Matern")
lines(c(sigma2.exponential + rho.exponential*(1-exp(-my.vgram$centers/aRange.exponential)))~my.vgram$ce

#Matern plot
ls.matern <- function(par){ # par = (rho,aRange,nu,sigmasq)
    theoretical.vgram <- par[1] * (1 - Matern(my.vgram$centers[1:15], range= par[2], nu = par[3])) + pa
    sum((theoretical.vgram - my.vgram$stats["mean",1:15])^2 )  }
```

```
##providing initial guesses and bounds
out.matern <- optim(par=c(2,50,2.5,1),fn=ls.matern,method="L-BFGS", lower = c(0,0,1,0), upper = c(25,10

rho.matern <- out.matern$par[1]
aRange.matern <- out.matern$par[2]
nu.matern <- out.matern$par[3]
sigma2.matern <- out.matern$par[4]

lines(c(sigma2.matern + rho.matern*(1-Matern(my.vgram$centers[1:15],range = aRange.matern, nu = nu.mate

legend(x=2, y=19 ,legend=c("Exponential","Matern"),col=c(1,2),pch=15, cex=1)
```

It is clear that the Matern correlation provides a better fit to the binned sample variogram. Therefore, we use the estimated Matern covariance function for prediction.

##Kriging predictor

To compute our prediction at $\mathbf{x}_0$, we use this set of parameters along with the kriging predictor $\Sigma_0\Sigma^{-1}\mathbf{Y}$. Since we have assumed an isotropic Matern covariance, the Σ_0 and Σ^{-1} matrices are easily computed with the `rdist.earth` function.

```
x0 <- as.matrix(cbind(-103,39.5))   #rdist only takes matrices
dist0.mat <- rdist.earth(grd.CO,x0)
dist.mat <- rdist.earth(grd.CO,grd.CO)

Sigma0 <- rho.matern * Matern(dist0.mat, range=aRange.matern, nu=nu.matern)
Sigma <- rho.matern * Matern(dist.mat, range=aRange.matern, nu=nu.matern)
diag(Sigma) <- diag(Sigma) + sigma2.matern
Sigma.inverse <- solve(Sigma)

weights <- t(Sigma0) %*% Sigma.inverse
prediction <- weights %*% y.CO
```

The predicted value at $\mathbf{x}_0$ is calculated above, and after plotting, we see that it is reasonable given the neighboring values.

```
set.panel(1,2)
quilt.plot(grd.CO,y.CO, main="Observations")
US(add=TRUE)
points(-103,39.5, pch=4,lwd=3,cex=1.25, col='black')

grd.x0 <- rbind(grd.CO,x0)
y.x0 <- c(y.CO,prediction)
quilt.plot(grd.x0,y.x0,main='Prediction')
US(add=TRUE)
```

Here are the spatial weights for each spatial location when predicting the value at $\mathbf{x}_0$.

```
quilt.plot(grd.CO,weights)
title(expression('Weights for predicting at x'[0]))
points(-103,39.5, pch=4,lwd=3,cex=1.25, col='black')
US(add=TRUE)
```

We could easily extend $\mathbf{x}_0$ to be a vector of locations at which we want to predict.

```
lonvals <- seq(min(CO.loc[,1]),max(CO.loc[,1]),length.out=100)
latvals <- seq(min(CO.loc[,2]),max(CO.loc[,2]),length.out=100)
```

```r
pred.grd <- as.matrix(expand.grid(lonvals,latvals))
```

The kriging estimate at each location follows after modifying Σ_0.

```r
dist0.mat <- rdist.earth(grd.CO,pred.grd)
Sigma0 <- rho.matern * Matern(dist0.mat, range=aRange.matern, nu=nu.matern)
grid.predictions <-  t(Sigma0) %*% Sigma.inverse %*% y.CO

set.panel(1,2)
quilt.plot(grd.CO,y.CO)
US(add=TRUE)
quilt.plot(pred.grd,grid.predictions,main='Prediction')
US(add=TRUE)
```

Recall that we assumed that the process has mean zero, which is clearly inaccurate. The `fields` package creates a mean function consisting of a low-order polynomial in the spatial coordinates. When Kriging, we often wish to use an additional covariate to help estimate the mean. For example, consider the following plot of elevation in Colorado:

```r
quilt.plot(make.surface.grid(CO.Grid),CO.elevGrid$z)
title("Elevation in CO")
US(add=TRUE,col='grey')
```

The elevation plot is a near inverse of our prediction grid – this agrees with our intuition that there are lower temperatures at higher elevations. If we include a covariate `Z` corresponding to elevation, then we can produce even more accurate predictions (a sneak-peek is shown below).

##The Problem with Fitting a Variogram

Variogram fitting is often a poor representation of the underlying structure of an isotropic process. Below, we generate several random fields, and then compare the corresponding variograms to the true correlation functions.

```r
n <- 100
x <- matrix(seq(0,1,,n), nc=1)
d <- rdist(x)
cov <- Exponential(d, range=0.3)
cov.c <- chol(cov)

x0 <- 0  #getting the true variogram gamma(h) = C(0) - C(h)
d0 <- rdist(x0,x)
c0 <- Exponential(d0, range=0.3)
e0 <- (1-c0[1,])

vg <- list()
brk<- seq( 0, 1,, (25 + 1) )
set.panel(3,3)
for (i in 1:9){
  sim <- t(cov.c)%*%rnorm(n)
  vg[[i]] <- vgram(x, sim, N=25)
  plot(vg[[i]], breaks=brk, col=4, ylim=c(0,1.5))  #plot simulated variogram
  lines(x,e0)
}
```

Note that the fits of the sample variogram are poor estimates of the true correlation function, and they are particularly flawed at larger distances. For this reason, curve-fitting to the binned sample variogram is an unreliable method of estimating the underlying covariance parameters. Instead, parameters must be found through Maximum Likelihood Estimation (MLE) or Generalized Cross Validation. The top level `fields` functions `spatialProcess` and `mKrig` both automate these estimation methods for the covariance parameters.

#`spatialProcess`

Spatial statistics refers to the class of models and methods for data collected over a spatial region, and informally we will refer to spatial estimates based on a covariance model as Kriging. Examples of such regions might be a mineral field, a quadrant in a forest, or a geographic region. Also, these methods are not limited to two or three dimensions of geographical coordinates. A typical problem is to predict values of a measurement at locations where it is not observed, or, if the measurements are observed with error, to estimate a smooth spatial process from the data. The Kriging functions in `fields` have the advantage that it can use arbitrary covariance functions. For this reason, `spatialProcess`, `mKrig`, etc., are not limited to two dimensional problems or standard models.

Suppose we have observations at locations $\mathbf{x}_1, \dots, \mathbf{x}_n$. The `fields` package assumes a spatial model of the form

$$Y(\mathbf{x}_i) = P(\mathbf{x}_i) + \mathbf{Z}(\mathbf{x}_i)\mathbf{d}_i + g(\mathbf{x}_i) + \epsilon_i$$

where $P(\cdot)$ is a low order polynomial (default is a linear function `m=2`), $\mathbf{Z}$ is a matrix of covariates weighted by a vector $\mathbf{d}$, $g(\cdot)$ is a mean zero Gaussian stochastic process with a covariance $\rho \cdot k(\mathbf{x}, \mathbf{x}')$ that is known up to a scale constant ρ, and $\boldsymbol{\epsilon} \sim N(\mathbf{0}, \sigma^2 W^{-1})$ is measurement error. Typically, the weights matrix W is taken as the identity I, so $\epsilon_i \sim N(0, \sigma^2)$. Consistent with the spline estimate, we take $\lambda = \rho/\sigma^2$. The covariance function $k(\cdot, \cdot)$ may also depend on other parameters: we explain below the parameter estimation process in `spatialProcess` and the remaining parameters that should be specified. Throughout this discussion the reader should keep in mind that a low order polynomial $\mathbf{P}$ fixed effect is also part of the estimate. By default this is a linear function in the spatial coordinates, but the degree can be changed.

Of course, we could consider the spatial coordinates as covariates in the $\mathbf{Z}$ matrix and simply write the model as

$$Y(\mathbf{x}_i) = \mathbf{Z}(\mathbf{x}_i)\mathbf{d}_i + g(\mathbf{x}_i) + \epsilon_i.$$

However, we want to make it clear that the spatial polynomial $P(\cdot)$ is (by default) included in the kriging estimate, while other covariates $\mathbf{Z}$ must be specified.

`spatialProcess` represents one-stop shopping for fitting a spatial model in `fields`. It was designed so that users are able to quickly fit models and visualize them. We also add one caution: it hides several default model choices from the user. After a covariance model is chosen, a grid search is performed to find the optimal values of the parameters `aRange`, `rho`, and `sigma2`, and then the spatial model is computed with these estimated parameters. Any other covariance parameters (e.g. the smoothness) need to be specified. The default call uses the covariance function `stationary.cov`, specifically with the `Matern` and smoothness `nu=1`. The optimization is done by `MLESpatialProcess`.

A good way to think about the spatial model functions in `fields` is in the context of engines and wrappers. The two engines in `fields` which perform the Kriging computations are `Krig` and `mKrig`. `Krig` is a more numerically stable algorithm: if a covariance matrix is close to singular, `mKrig` will fails whereas `Krig` may still be able to perform the calculations. On the other hand, `mKrig` is set up to handle large data sets with sparse covariance matrices by using the `SPAM` package.

A wrapper is an easy to use function that is based on an underlying function or engine. Often default parameters values are set to make the function accessible to the user. The function `Tps` is a wrapper on `Krig`. `Tps` performs thin plate spline regression, which is a special case of Kriging, so it makes sense to use the `Krig` engine. On the other hand, the functions `fastTps` and `spatialProcess` are wrappers for `mKrig`. See the chapters and examples on these functions for specifics.

There is always a hazard in providing a simple to use method that makes many default choices for the spatial model. As in any analysis be aware of these choices and try alternative models and parameter values to assess the robustness of your conclusions. Also examine the residuals to check the adequacy of the fit!

Basic Usage

spatialProcess(x, y, Z = NULL, cov.function = “stationary.cov”,
cov.args = list(Covariance = “Matern”, smoothness = 1), aRange = NULL)
Krig(x, Y, aRange, Covariance=“Matern”, smoothness, Distance=“rdist”)

Value

`spatialProcess` returns object of classes `mKrig` and `spatialProcess`. The main difference from `mKrig` is an extra component `MLEInfo` that has the results of the grid evaluation over `aRange` (maximizing `lambda`), joint maximization over `aRange` and `lambda`, and a grid evaluation over `lambda` with `aRange` fixed at its MLE. The `Krig` function produces an object of class `Krig`.

There are a number of S3 functions associated with `spatialProcess` objects (also `mKrig`, `Krig`, `fastTps`, etc). The `coef` method gives the coefficients `d` from the fixed part of the model (null space or spatial drift). `surface` returns an image of the model fit with `Z` covariates dropped by default. `predict` allows the user to predict the fit at new locations, and `predictSE` gives the predicted standard errors at new locations. `plot` returns a series of diagostic plots, and `print(fit)` or `summary(fit)` give details about the estimated parameters and model.

2.1 Analysis of soil pH

This example is meant to quickly highlight the basic functionality of `spatialProcess` for fitting a model using pH soil samples. The data can be found at http://homepage.divms.uiowa.edu/~dzimmer/spatialstats/soilphmatrix.dt.

We visualize the raw data, fit a `spatialProcess` model, and plot the resulting predictions for comparison.

```
file1 <- read.table("soilphmatrix.txt", sep=" ")
dat <- as.matrix(file1, nr=11, nc=11)
image.plot(dat) #from sp package
title("pH in Soil")
```

```
grid.list <- list(x=seq(1,11), y=seq(1,11))
full.grid <- make.surface.grid(grid.list)
fit <- spatialProcess(x=full.grid, y=c(dat))
surface(fit)
title("Kriging Predictions")
```

Next we look at the summary of the model parameters. The optimization search is automated in `fields`, but all of the information is easily accessible for the user. Recall the range is \$a \$, the nugget is σ^2, the sill is ρ, and we set $\lambda = \sigma^2/\rho$.

```
print(c(fit$cov.function.name,fit$args))
```

```
## [[1]]
## [1] "stationary.cov"
##
## $Covariance
## [1] "Matern"
##
## $smoothness
## [1] 1
##
## $aRange
## [1] 1.124259
##
## $onlyUpper
## [1] FALSE
##
## $distMat
## [1] NA
```

```
fit$summary # same as fit$MLEInfo$summary
```

```
## lnProfileLike.FULL lnProfileREML.FULL             lambda                tau             sigma2
##        31.59561718        25.35246419         0.93216944         0.14404301         0.02225817
```

The summary shows the value of the profile log likelihood function, the estimates for the parameters, and the number of evaluations needed in the optimization.

We can use `plot` to view diagnostic plots of the fit. The first three plots are the same as `sreg` and `mKrig` objects. Plot 1 shows data vs. predicted values, and plot 2 shows predicted values vs. residuals. Plot 3 shows the criteria to select the smoothing parameter $\lambda = \sigma^2/\rho$. The x axis has transformed λ in terms of effective degrees of freedom to make it easier to interpret. Note that here the GCV function is minimized while the REML is maximized.

One of the main features of `spatialProcess` is the ability to optimize over `aRange` as well as the usual `lambda`/`rho`/`sigma` combination. For this reason, plot 4 shows the profile likelihood versus values of `aRange` instead of a histogram of the standard residuals displayed in the plots of other class objects

```
set.panel(2,2)
plot(fit)
```

2.2 Analysis of Coal Ash

`coalash` is a classic geostatistics data set used to illustrate statistical methods. This example shows in more depth the features that are available for use with a `spatialProcess` model.

```
library(gstat)
data(coalash)
x <- cbind(coalash$x, coalash$y)
y <- coalash$coalash
quilt.plot(x, y,col=topo.colors(100))
```

```
fit <- spatialProcess(x, y)
surface(fit,col=topo.colors(100))
```

Note that `surface` displays the full model, except `Z` covariates if they are included; i.e. by default `Z` is dropped. To additionally include the `Z` covariates, one can use `predict` or `predictSurface`. In these functions, the default is to retain `Z` in the computation.

We can use our spatial model to predict on a grid using `predict`, or `predictSurface` which is more convenient for plotting.

```
grid.list <- list(x=seq(0,17), y=seq(0,24) )
pred.grid <- predict(fit, grid=grid.list)
look <- as.surface(grid.list, pred.grid)
surface(look,col=topo.colors(100))  #Try image.plot(look)
```

We use `predictSurface` to quickly generate the full model, and then supply `just.fixed=TRUE` to see the fixed polynomial. Thus, the difference of the two will just be the spatial part of the model. Refer to the last example in this section to see such a decomposition of a spatial model with a `Z` covariate included.

(Note: `smallplot` is used to properly format this document, and the reader can ignore it.)

```
out.full <- predictSurface(fit,extrap=TRUE)
out.poly <- predictSurface(fit,just.fixed=TRUE,extrap=TRUE)

out.spatial <- out.full
out.spatial$z <- out.full$z - out.poly$z

out.check <- out.full
out.check$z <- out.poly$z +  out.spatial$z

set.panel(1,3)
surface(out.full,smallplot= c(.88,.9,0.2,.8),col=topo.colors(100) ); title("Full model")
surface(out.poly,smallplot= c(.88,.9,0.2,.8),col=topo.colors(100)); title("Low-order spatial polynomial
surface(out.spatial,smallplot= c(.88,.9,0.2,.8),col=topo.colors(100)); title("Spatial part g(x)")
```

We can also use `sim.spatialProcess` to produce conditional simulations by supplying the model, locations at which to simulate in `xp`, and the number of Monte Carlo simulations with `M`.

```
full.grid <- make.surface.grid(grid.list)
sim <- sim.spatialProcess(fit, xp=full.grid, M=2)
sim.surf <- as.surface(full.grid, sim[2,])

set.panel(1,2)
surface(sim.surf,col=topo.colors(100)); title("Conditional Simulation")

drape.plot(sim.surf, border=NA, aRange = 30, phi= 25,col=topo.colors(100)) -> dp
```

```
## Warning in persp.default(x, y, z, theta = theta, phi = phi, col = drape.info$color.index, : "aRange"
```

```
title("3D Perspective with observations in black")
trans3d( x[,1], x[,2],y,dp)-> uv
points( uv, col="black", pch="+", cex=0.5)
```

Finally we take a look at the uncertainty in our model predictions with `predictSE`. The white points show the locations of the observations.

```
set.panel(1,2)
SEobs <- predictSE.mKrig(fit)
SEout <- predictSE(fit, xnew=full.grid)
lookSE <- as.surface(full.grid, SEout)
{surface(lookSE,col=topo.colors(100))
title("Kriging Uncertainty")
points(x[,1],x[,2],col="magenta",bg="white",pch=21)}
{drape.plot(lookSE, border=NA, aRange=160, phi=55,col=topo.colors(100)) -> dp2
title("Drape Plot")
pushpin(x[,1],x[,2],SEobs,dp2, cex=0.4, col="white")}
```

```
## Warning in persp.default(x, y, z, theta = theta, phi = phi, col = drape.info$color.index, : "aRange"
```

##ozone2

Now we consider the `ozone2` dataset in `fields`, which consists of CO2 observations over 89 days at a set of 153 locations. We'll restrict ourselves to day 16 (June 18, 1987) and remove all `NA` values. We fit several covariance models: Matern, exponential, and Wendland. We first visually inspect the different surfaces, then we compare the parameters of the three models using `summary`.

```
data( ozone2)
x<- ozone2$lon.lat
y<- c(ozone2$y[16,])
set.panel(1,2)
US( xlim= c(-94,-83),  ylim=c(37, 45) )
points( ozone2$lon.lat, pch="o")
title("Locations")

keep <- !is.na(y)
y <- y[keep]
x <- ozone2$lon.lat[keep,]

quilt.plot(x,y,add.legend=TRUE,main="Values for Day 16", col=heat.colors(100))
US(add=TRUE)
```

```
# exponential vs Wendland covariance function
obj <- spatialProcess( x, y, Distance = "rdist.earth")
obj2<- spatialProcess( x, y, Distance = "rdist.earth",
            cov.args = list(Covariance ="Exponential") )
obj3<- spatialProcess( x, y, Distance = "rdist.earth",
           cov.args = list(Covariance = "Wendland",
           dimension = 2, k = 2) )
rbind(obj$summary,obj2$summary,obj3$summary)
```

Since the exponential is a Matern covariance with $\nu = 0.5$, the first two fits can be compared in terms of their likelihoods. The REML value is slightly higher for `obj` verses `obj2` ($598.4 > 596.7$). These are the negative log likelihoods so this suggests a preference for the exponential model. But... does it really matter in terms of spatial prediction?

```
library(sp) #for colors
set.panel(1,3)
surface(obj, col=heat.colors(100),smallplot= c(.88,.9,0.2,.8))
US( add=TRUE)
title("Matern sm=1.0")
surface(obj2, col=heat.colors(100),smallplot= c(.88,.9,0.2,.8))
US( add=TRUE)
title("Matern sm=.5")
surface(obj3, col=heat.colors(100),smallplot= c(.88,.9,0.2,.8))
US( add=TRUE)
title("Wendland k=2")
```

It is always a good idea to take a look at the prediction standard errors. Of course, the standard errors are lower in areas with many observations, and they are higher in regions with few observations. These computations take a while because prediction errors are based directly on the kriging weight matrix. See the chapter on `mKrig` for an approximate alternative using conditional simulations.

```
set.panel(1,2)
quilt.plot(x,y,add.legend=TRUE,main="Values for Day 16", col=heat.colors(100))
US(add=TRUE)
xg <- fields.x.to.grid(x)
std.err <- predictSE(obj,x=make.surface.grid(xg),Distance="rdist.earth")   #takes a while
out.p <- as.surface(xg,std.err)
#out.p<- predictSurfaceSE(obj, nx=40,ny=40)
```

```
surface(out.p, col=heat.colors(100))
US( add=TRUE)
title("Matern sm= 1.0 SE")
points( x, col="white")
```

Finally, we use this data set to test whether the MLE computations yield the same results using `spatialProcess` in `fields` versus `likfit` in `geoR`.

```
#library(geoR)
obj<- spatialProcess( x, y, mKrig.args= list(m=1), smoothness = .5 )
ml.n <- likfit(coords= x, data=y, ini = c(570, 3), nug = 50)
```

```
# Comparing the two
stuffFields<- obj$MLEInfo$MLEJoint$summary[c(1,3,4,5)]
stuffGeoR<- c( ml.n$loglik, ml.n$phi,
               sqrt(ml.n$nugget),ml.n$sigmasq)

print(rbind(stuffFields,stuffGeoR))
```

```
##                  [,1]     [,2]     [,3]     [,4]
## stuffGeoR -611.0195 1.908864 7.136655 778.2998
```

```
test.for.zero( stuffFields, stuffGeoR, tol=.005)
```

```
## PASSED test at tolerance  0.005
```

2.3 COmonthlyMet

We'll return to the dataset `COmonthlyMet`. When using other `fields` functions for Kriging such as `Krig` and `mKrig`, the user has to pass in a range `aRange`, which can be initally guessed for a quick fit by looking at a variogram.

Note that the estimated `aRange` without including the `Z` covariate `CO.elev` will probably change if do we include `Z`.

Ultimately, using a variogram to determine spatial parameters for our covariance is not a reliable method. A more rigorous way to determine parameters is via Maximum Likelihood, which is performed within `spatialProcess`. To be precise, the parameters `aRange`, `rho`, and `sigma2` are estimated using Maximum Likelihood, and the user only needs to input `x`, `y`, and a type of covariance. (Note: If a Matern covariance is used, then smoothness must be supplied. See below how one might select between different smoothness values.)

Of course, `spatialProcess` runs slower than the functions would with a user-supplied `aRange`. One must take caution when using `spatialProcess` with large datasets. Below, we run through this climate example using `spatialProcess` to construct our model.

```
data(COmonthlyMet)
y.CO <- CO.tmax.MAM.climate
z.CO <- CO.elev
grd.CO <- as.matrix(CO.loc)
keep <- !is.na(y.CO)  #removing all NA's from the temperatures
y.CO <- y.CO[keep]
z.CO <- z.CO[keep]
grd.CO <- grd.CO[keep,]

out0.5 <- spatialProcess(grd.CO,y.CO,Z=z.CO, Distance="rdist.earth",
                          cov.args=list(Covariance="Matern"), smoothness=0.5)
out1 <- spatialProcess(grd.CO,y.CO,Z=z.CO, Distance="rdist.earth",
                           cov.args=list(Covariance="Matern"), smoothness=1)
out2.5 <- spatialProcess(grd.CO,y.CO,Z=z.CO, Distance="rdist.earth",
                           cov.args=list(Covariance="Matern"), smoothness=2.5)
out5 <- spatialProcess(grd.CO,y.CO,Z=z.CO, Distance="rdist.earth",
                           cov.args=list(Covariance="Matern"), smoothness=5)

rbind(out0.5$summary,out1$summary,out2.5$summary,out5$summary)
```

```
##      lnProfileLike.FULL lnProfileREML.FULL    lambda       tau   sigma2    aRange   eff.df       GCV
## [1,]          -241.9493          -254.8387 0.2899935 0.5850161 1.180178 241.11158 67.07374 0.5111728
## [2,]          -241.6651          -254.5076 0.3249437 0.6573736 1.329892 167.36154 31.38195 0.5143462
## [3,]          -240.9861          -253.9855 0.3001930 0.6844790 1.560701  95.12465 17.18247 0.5158045
## [4,]          -240.4959          -253.5272 0.3252950 0.6884790 1.457149  59.47066 14.77198 0.5150254
```

Observe the largely different values for parameters, but similar values for log-likelihood. Note that there is an interplay between smoothness and range, so directly comparing `aRange` is not appropriate. Will the underlying spatial process be much different if we vary the `smoothness`? We'll use the `drop.Z` command to make the differences in the spatial process clear.

<Note that `predictSurface.Krig` and `predictSurface.mKrig` are the same. With a `spatialProcess` object, just using `predictSurface` throws an error, you must use `predictSurface.Krig`. (Take this out soon.)>

```
pred0.5 <- predictSurface.Krig( out0.5,grid.list=CO.Grid,ZGrid=CO.elevGrid,drop.Z=TRUE)
pred1 <- predictSurface.Krig( out1,grid.list=CO.Grid,ZGrid=CO.elevGrid,drop.Z=TRUE)
pred2.5 <- predictSurface.Krig( out2.5,grid.list=CO.Grid,ZGrid=CO.elevGrid,drop.Z=TRUE)
pred5 <- predictSurface.Krig( out5,grid.list=CO.Grid,ZGrid=CO.elevGrid,drop.Z=TRUE)

set.panel(2,2)

mycol <- rev(bpy.colors())

surface(pred0.5,type="C",col = mycol)
title("Smoothness 0.5")
US(add=TRUE)

surface(pred1,type="C",col = mycol)
title("Smoothness 1")
US(add=TRUE)

surface(pred2.5,type="C",col = mycol)
title("Smoothness 2.5")
US(add=TRUE)

surface(pred5,type="C",col = mycol)
title("Smoothness 5")
US(add=TRUE)
```

Notice that smoothness values 2.5 and 5 look nearly identical. In practice, one usually selects smoothness values from the set $\{0.5, 1, 1.5, 2, 2.5\}$.

<We see that `MLEspatialProcess` estimates `aRange` decently well but `sigmaMLE` is far from the truth. Try with more data points! Now we switch to a Matern covariance, then compute the joint MLE of range, smoothness, and lambda. >

2.4 Decomposition of a Spatial Model

Finally, we show how to access the low-order polynomial $P(\cdot)$, the mean trend **Zd**, and the stochastic process $g(\cdot)$ that make up a spatial process.

```
data(COmonthlyMet)
# predicting average daily minimum temps for spring in Colorado
obj<- spatialProcess( CO.loc, CO.tmin.MAM.climate, Z= CO.elev)
out.p<-predictSurface( obj, grid.list=CO.Grid, ZGrid= CO.elevGrid, extrap=TRUE)

image.plot(out.p, col =  mycol)
US(add=TRUE, col="grey")
contour( CO.elevGrid, add=TRUE, levels=seq(1000,3000,,5), col="black")
title("Average Spring daily min. temp in CO")
```

Now that we have our `spatialProcess`, we will extract these different components of the model.

```
out.dropZ <- predictSurface( obj, grid.list=CO.Grid, ZGrid=CO.elevGrid,
                             drop.Z=TRUE, extrap=TRUE)
out.fixed <- predictSurface( obj, grid.list=CO.Grid, ZGrid=CO.elevGrid,
                             just.fixed=TRUE, extrap=TRUE)
out.poly <- predictSurface( obj, grid.list=CO.Grid, ZGrid=CO.elevGrid,
                            just.fixed=TRUE, drop.Z=TRUE, extrap=TRUE)

Zd <- out.fixed$z - out.poly$z
out.Z <- as.surface(CO.Grid, Zd)

spatial.part <- out.dropZ$z - out.poly$z
out.sp <- as.surface(CO.Grid, spatial.part)

set.panel(3,2)
surface( out.p,  main="Full model",col =  mycol)
surface(out.poly, main="Spatial polynomial P(x)",col = mycol)
surface(out.fixed, main="Mean Part: P(x) + Zd ",col = mycol)
surface( out.dropZ, main="Spatial Part + Spatial Polynomial",col = mycol)
surface(out.Z, main="Covariate trend Zd",col = mycol)
surface(out.sp, main="Only Spatial Part",col = mycol)
```

```
temp <- out.poly$z + out.Z$z + out.sp$z
out.total <- as.surface(CO.Grid, temp)
surface(out.total, main="Full model, checked by adding parts",col = rev(bpy.colors()))
```

3 Images, Surfaces, and Plotting

In the course of spatial analysis of 2-d fields, one needs simple functions to create, plot, and smooth rectangular images. We present many of the `fields` functions for working with image data and simulatneously highlight the plotting capabilites of the package. By a surface or image object, we mean a list with components `x`, `y` and `z`, where `x` and `y` are equally spaced grids and `z` is a matrix of values. This is the same format as used by the R functions `contour`, `image`, and `persp`. The `x` and `y` components index each pixel and also index the matrix, whereas the `z` component gives the pixel values.

3.1 `as.image`

`as.image` creates an image object from irregular `x`, `y`, and `z` coordinates by discretizing the 2-d locations to a grid and then producing an image object with the `z` values in the correct cells. The basic idea is that each observation is identified with a grid box, and then the mean value for all observations is returned for that box. If no observations fall into a particular box, an `NA` is returned for that location. Missing values are handled deftly by the contouring and image functions, such as `image.plot`.

The required argument for `as.image` is `Z`, the values for the image and the most common optional argument is `x`, a 2 column matrix with the spatial locations of `Z`.

In the following example we grid the `RMprecip` data and plot it. The default is a 64 by 64 grid using the ranges of the data. One can also specify alternative grids: the third panel in the next plot demonstrates a 25 by 40 point discretization.

```
set.panel(1,3)
out.p <- as.image( RMprecip$y,  x= RMprecip$x)
image( out.p)
#compare to quilt.plot
quilt.plot(RMprecip$x, RMprecip$y)
grid<- list( x=seq( -111, -99,,25), y=seq( 35, 45,,40))
as.image( RMprecip$y, RMprecip$x, grid=grid)-> out.p
image(out.p)
```

Several other useful options are available. One can specify weights to use in finding a weighted mean for each box, or just give the number of grid points and use the ranges of the data for gridding. Also note that the list returned by `as.image` has components to indicate the counts in each bin and membership.

Basic Usage

as.image(Z, ind=NULL, grid=NULL, x=NULL, nrow=64, ncol=64,weights=NULL)

Value

`as.image` returns a list: in addition to x,y, and z components, the list has the counts `N`, the indices of the nonmissing boxes `ind` and sum of the weights for each nonmissing box `weights`.

3.2 Plotting with `image`, `image.plot`

The function `image.plot` has the same functionality as `image` but adds a legend strip. It can be used to add a legend strip to an existing plot or create a new image and legend. Its chief benefit is that this function resizes the plot region automatically to make room for the legend strip. The argument `horizontal`, if true, will put the legend strip under the plot. One effective graphical plot is a panel of images on a similar scale and plot region size but with only one legend strip. `image.plot` makes this fairly easy to do.

The function `add.image` is useful for adding an image to an existing plot. The user gives the matrix of intensities of the image as `z`, as well as simple arguments that control the location and size. `xpos` and `ypos` give the position of the image, and `adj.x` and `adj.y` determine whether the image is centered on that location (0.5,0.5), to the bottom right (1,1), Finally, we can change the width and height of the image as a fraction of the plotting region with `image.width` and `image.height`. If `NULL`, height is scaled to make image pixels square.

To lead by example, we plot the `RMprecip` data at several discrtetizations. We see that indeed only `image.plot` includes a colorbar, and the color palettes are quite different. Then we use `add.image` to put John on the Four Corners for fun!

```
set.panel( 1,3)
par( pty="s")   # square plotting regions
image.plot( as.image( RMprecip$y, x = RMprecip$x), col=heat.colors(12)); US( add=TRUE, col="black", lwd=
image(  as.image( RMprecip$y, x = RMprecip$x, ncol=32,nrow =32)); US( add=TRUE, col="black", lwd=2)
image(as.image( RMprecip$y, x = RMprecip$x, ncol=16, nrow=16)); US( add=TRUE, col="black", lwd=2)
data(lennon)
add.image( -109,37, lennon , col=grey( (0:256)/256))
```

Basic Usage

> image.plot(image, col = tim.colors(nlevel))
> add.image(xpos, ypos, z, adj.x = 0.5, adj.y = 0.5, image.width = 0.15, image.height = NULL)

<Here is an example using the full functionality of `image.plot`. >

3.3 `grid.list`, `make.surface.grid`, and `as.surface`

`grid.list` refers to an x, y list format for a grid. `make.surface.grid` (a wrapper around the R base function `expand.grid`) is a useful function that expands the grid from the `grid.list` into a full set of locations.

`as.surface` is a similar function to `as.image`. This function was written to simply to go back and forth between a matrix of gridded values and the stacked vector obtained by stacking columns. The input `obj` can be a `grid.list` of x and y coordinates, or a matrix of the grid points produced by `make.surface.grid`. `z` denotes the values of the function at the grid points, and finally `order.variables` (either `"xy"` or `"yx"`) specifies how the x and y variables used to evaluate the function are matched with the x and y grids in the surface object. (Note: to convert irregular 2-d data to a surface object where there are missing cells, see the function `as.image`.)

Basic Usage

> make.surface.grid(grid.list)
> as.surface(obj, z, order.variables="xy")

Value

The result will be a list with x, y, and z components suitable for plotting with functions such as `persp`, `image`, `surface`, `contour`, `drape.plot`.

`make.surface.grid` returns a full set of grid locations as a matrix, and `as.surface` returns a list of class `surface`.

```
grid.list <- list( x= -111:-99, y=35:45)
xg <- make.surface.grid(grid.list)
str(grid.list)
```

```
## List of 2
##  $ x: int [1:13] -111 -110 -109 -108 -107 -106 -105 -104 -103 -102 ...
##  $ y: int [1:11] 35 36 37 38 39 40 41 42 43 44 ...
```

```
str(xg)
```

```
##  int [1:143, 1:2] -111 -110 -109 -108 -107 -106 -105 -104 -103 -102 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "x" "y"
##  - attr(*, "grid.list")=List of 2
##   ..$ x: int [1:13] -111 -110 -109 -108 -107 -106 -105 -104 -103 -102 ...
##   ..$ y: int [1:11] 35 36 37 38 39 40 41 42 43 44 ...
```

A typical operation is to go from a `grid.list` to the set of grid locations, evaluate a function at these locations, and then reformat this as an image/surface for plotting. Here is how to do this cleanly:

```
fit <- spatialProcess(RMprecip$x, RMprecip$y, aRange=20)
grid.list <- list( x= -111:-99, y=35:45)
xg<- make.surface.grid(grid.list)
look <- predict(fit, xg)
look.surface <- as.surface(xg, look)
```

The next example gives the user an idea of the comparison of the plotting functions in `fields` or the standard libraries that are relevant to images, surfaces, grids, and object in classes defined in `fields`.

```
image(look.surface)
```

Using `image.plot` in `fields` adds a colorbar and a different default color scale.

```
image.plot(look.surface)
```

Here is the standard R function `contour`.

```
contour(look.surface)
```

Using `surface` combines `image.plot` with `contour` as its default.

```
surface(look.surface)
```

There is a graphics function `persp` which can produce 3-d plots. `drape.plot` is a wrapper function in `fields` that automatically adds color based on how it's done in `image.plot`/`surface`.

```
set.panel(1,2)
persp(look.surface)
drape.plot(look.surface)
```

In summary, we fit a `spatialProcess` object from our data, created a “grid.list”, constructed a full set of gridded locations using the “grid.list,” found Kriging estimates at the gridded locations, converted back to a “grid.list”, and then plotted with `surface`. Compare this with the convenient `surface.mKrig`.

```
fit <- spatialProcess(RMprecip$x, RMprecip$y, aRange=20)
surface(fit)
```

3.4 `surface`

We saw in the last example that we can use `surface` to visualize both `surface` objects and also `spatialProcess/mKrig/Krig` objects. It is a great tool to be able to fit a class object and quickly plot predictions in a few short lines of code.

This surface function is essentially a combination of predictSurface and plot.surface. It may not always give a great rendition but is easy to use for checking the fitted surface. The default of extrap=F is designed to discourage looking at the estimated surface outside the range of the observations. Note that any Z covariates in the the model will be dropped and only the spatial part of the model will be evaluated.

Basic Usage

surface(object, grid.list = NULL, extrap = FALSE, type="C")

The type argument has the following options: "p" gives persp, "c" gives contour with legend strip (image.plot), and "C" gives image plot with contours overlaid.

```
quilt.plot(ChicagoO3$x, ChicagoO3$y,col=terrain.colors(50))
```

```
fit<- Krig(ChicagoO3$x,ChicagoO3$y, aRange=30)
set.panel(1,2)
surface(fit, type="C", nx=128, ny=128,col=terrain.colors(50))
surface(fit, type="p", nx=56, ny=56, extrap = TRUE, border=NA, aRange=-8, phi=20,col=terrain.colors(50)
```

```
## Warning in persp.default(x, y, z, theta = theta, phi = phi, col = drape.info$color.index, : "aRange"
```

4 Smoothing and Bilinear Interpolation of Images

4.1 Smoothing with image.smooth and smooth.2d

image.smooth and smooth.2d are two functions in fields that will smooth 2-d data using a kernel. image.smooth requires the data in the form of an image matrix, and it can handle missing values for cells. smooth.2d takes irregular data as input but then discretizes it to a regular grid (possibly with missing cells) and applies the same smoothing operations as image.smooth. In both cases the default kernel function is an exponential, although different kernel forms can be specified. For the exponential, the bandwidth parameter is passed as the argument aRange.

Using smooth.2d is essentially the same as applying as.image to coerce an irregular grid of data to become equispaced and then image.smooth. Both of these functions use the base R fft function for efficiency.

Basic Usage

smooth.2d(Y, cov.function = gauss.cov)
image.smooth(x, kernel.function = double.exp, aRange = 1)
setup.image.smooth(nrow = 64, ncol = 64, dx = 1, dy = 1, kernel.function = double.exp, aRange = 1)

Value

smooth.2d return either a matrix of smoothed values or a surface object. The surface object also has a component ind that gives the subscripts of the image matrix where the data is present. image.smooth returns a list with components x, y and z.

Here is an example smoothing the `RMprecip` data and plotting the smoothed image.

```
set.panel(1,2)
out<- smooth.2d( RMprecip$y, x=RMprecip$x, aRange=1.0)
quilt.plot( RMprecip$y, x=RMprecip$x,zlim=c(0,250),main="Image")
image( out, zlim=c(0,250),col=tim.colors(),main="Smoothed Image (same z-scale)")
```

If we think of an image as a matrix of pixel values, these smoothers work by multiplying every matrix entry by a smaller smoothing matrix, such as

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

We can control the behavior of the smoother by specifying `cov.function/kernel.function`. The kernel/covariance function is discretized into the form of a smoothing matrix, then the matrices are convolved. Below are examples where we define new kernels based on the gaussian and gamma distributions. Parameters such as `sigma` in the gaussian kernel control the smoothing weights. Especially in this case, since the Fourier transform of a gaussian is gaussian, we can be certain of which high frequencies we attenuate in the signal. To see an example of how to define a new covariance function, see the `Covariance` section of this vignette. A kernel function that is only being passed to `image.smooth` does not need to satisfy the covariance requirements in the package (see the `C` parameter for covariances). `setup.image.smooth` creates a weighting object outside of a call to `image.smooth`. This is useful when one wants to smooth different data sets but on the same grid with the same kernel function. The argument `aRange` is the bandwidth parameter in `setup.image.smooth`.

```
gaussianKern <- function(x, sigma=2){
  1/sqrt(2*pi*sigma^2) * exp(-0.5*(x)^2 / sigma^2)
}
wght <-  setup.image.smooth(nrow = 256, ncol = 256, dx = 1, dy = 1,
                    kernel.function = gaussianKern, aRange = 0.5)
```

We use this setup to smooth the `RMprecip` for the final time, showing how we can use either `smooth.2d` or `image.smooth`. We use kernels as well as different levels of resolution in this example.

```
# Use our kernel: first coerce data as.image, then image.smooth with our wght setup
look<- image.smooth( as.image( RMprecip$y, x = RMprecip$x), wght)
# Normal kernel smooth of the precip data with bandwidth of .5 (degree), now using smooth.2d
look2 <- smooth.2d( RMprecip$y, x=RMprecip$x, aRange=.5)
# finer resolution used in computing the smooth
look3<-smooth.2d( RMprecip$y, x=RMprecip$x, aRange=.25, nrow=256,
                  ncol=256,Nwidth=32,Mwidth=32)
look4 <- image.smooth( as.image(RMprecip$y, x=RMprecip$x),
                    kernel.function=double.exp,aRange=0.5)
set.panel(2,2)
image.plot( look, zlim=c(0,250)); title("gaussianKern, aRange=0.5"); US( add=TRUE, col="grey", lwd=2)
image( look4, zlim=c(0,250),col=tim.colors()); title("double.exp, aRange=0.5"); US( add=TRUE, col="grey
image.plot( look2, zlim =c(0,250)); title("smooth.2d, aRange=0.5"); US( add=TRUE, col="grey", lwd=2)
image( look3, zlim =c(0,250),col=tim.colors()); title("smooth.2d, finer res, aRange=.25"); US( add=TRUE
```

##`image.smooth` and `lennon`

Here is another quick example showing how to work with an actual image such as a JPEG. Pick any image you want from the web, smooth it, and look at the information lost. We first define a new kernel based on the gamma distribution.

```
t <- seq(0,1,,1000)
plot(dgamma(t, shape=1.1)~t, type="l", col="mediumaquamarine", main
     ="Gamma Distributions", xlab="", ylab="")
legend(x=0.75,y=0.25,legend=c("Shape=1.1", "Shape=1.4", "Shape=1.8"), pch=21, cex=1,
       col=c("mediumaquamarine", "red4", "blue2"))
lines(dgamma(t,shape=1.4)~t, col="red4")
lines(dgamma(t,shape=1.8)~t, col="blue2")
```

```
gammaKern <- function(x, shape=1.1){
  dgamma(x, shape=shape)
}
set <-  setup.image.smooth(nrow = 256, ncol = 256, dx = 1, dy = 1,
                 kernel.function = double.exp,aRange = 1)
set2 <-  setup.image.smooth(nrow = 256, ncol = 256, dx = 1, dy = 1,
                kernel.function = gammaKern,aRange = 1)
```

Now we apply `image.smooth` to the data.

```
data(lennon)
x1 <- image.smooth(lennon, set)
x2 <- image.smooth(lennon, set2)
set.panel(1,3)
image.plot(lennon, col=grey.colors(100), main="Original image",zlim=c(0,200))
image(x1$z, col=grey.colors(100), main="Smoothed with double.exp",zlim=c(0,200))
image(x2$z, col=grey.colors(100), main="Smoothed with gammaKern",zlim=c(0,200))
```

We can look at the difference between the original image and the smoothed version, and also the difference between the two smooths.

```
set.panel(1,3)
image.plot(lennon-x1$z, col=grey.colors(10), zlim=c(-5, 60), main="double.exp smoothing difference")
image(lennon-x2$z, col=grey.colors(10), zlim=c(-5, 60), main="gammaKern smoothing difference")
image(x2$z-x1$z, col=grey.colors(10), zlim=c(-5,60), main="Difference between smoothing kernels")
```

Here is one final example of smoothing gridded Colorado elevation data many times. We also give details on the nice plotting function `drape.plot` and some of the color palettes included in `fields`. Since we display four plots with the same scale, we withhold the legend and use `image.plot` to manually add one.

```
data(COmonthlyMet)
x <- CO.elevGrid$x
y <- CO.elevGrid$y
z <- CO.elevGrid$z

smoothNtimes <- function(im, N=1){
  for (i in 1:N){
    temp <- im
    im <- image.smooth(temp)
  }
  im
}
z2 <- smoothNtimes(z,1)
z3 <- smoothNtimes(z2,4)
z4 <- smoothNtimes(z3,15)
{par(oma=c( 0,0,0,4))
set.panel(2,2)
drape.plot( x,y,z, col=larry.colors(),border=NA,main="Elevation (CO)",
           xlab="New Mexico",ylab="Kansas", add.legend = FALSE) -> dp
drape.plot( x,y,z2$z, col=larry.colors(),border=NA,
           xlab="New Mexico",ylab="Kansas",main="1 Smooth", add.legend = FALSE)
drape.plot( x,y,z3$z, col=larry.colors(),border=NA,
           xlab="New Mexico",ylab="Kansas",main="5 Smooths", add.legend = FALSE)
drape.plot( x,y,z4$z, col=larry.colors(),border=NA, add.legend = FALSE,
           xlab="New Mexico",ylab="Kansas",main="20 Smooths")
pushpin(-105.2705,40.0150,1655,dp, text="Boulder", height=0.15, cex=1)
par(oma=c( 0,0,0,1))
image.plot(legend.only=TRUE, horizontal=TRUE, zlim=c(900,3300), col=larry.colors())}
```

4.2 drape.plot and pushpin

`drape.plot` produces the usual wireframe perspective plot with the facets being filled with different colors. `drape.plot` is a higher level function that works by calling `drape.color` followed by `persp`, and finally the legend strip is added with `image.plot`. Note that `surface` with `type="p"` gives the same results.

By default, the colors are assigned from a color bar based on the `z` values. `drape.color` can be used to create a color matrix different from the `z` matrix used for the wireframe. When using `drape.color`, just put the results into the `col` argument of `persp`. The color scales essentially default to the ranges of the `z` values. However, by specifying `zlim` and/or `zlim2` one has more control of how the perspective plot is scaled and the limits of the color scale used to fill the facets. The color assignments are done by dividing up the `zlim2` interval into equally spaced bins and adding a very small inflation to these limits. The mean `z2` values, comprising an $M-1$ by $N-1$ matrix, for each facet are discretized to the bins. The bin numbers then become the indices used for the color scale. If `zlim2` is not specified it is the range of the `z2` matrix that is used to generate the ranges of the color bar. Note that this may be different than the range of the mean facets. If `z2` is not passed then `z` is used in its place and in this case the `zlim2` or `zlim` argument can used to define the color scale.

The legend strip may obscure part of the plot. If so, add this as another step using `image.plot`. This kind of plot is also supported through the wireframe function in the `lattice` package. The advantage of the `fields` version is that it uses the standard R graphics functions and is written in R code. Note that the drape plot is also drawn by the `surface` function with `type="P"`.

`pushpin` is a small function that adds to an existing 3-d perspective plot a push pin to locate a specific point. Arguments needed are the coordinates and either projection information returns by `persp` in `p.out` or a `persp`/`drape.plot` assigned to a variable. One can adjust the size, length, and color of the pin and also add text.

Basic Usage

drape.plot(x, y, z, z2=NULL, col = tim.colors(64), zlim = range(z, na.rm=TRUE), zlim2 = NULL)
drape.color(z, col = tim.colors(64), zlim = NULL,breaks,transparent.color = “white”)
pushpin(x,y,z,p.out, height=.05,col=“black”,text=NULL,adj=-.1,cex=1.0,. . .)

Value

If an assignment is made, the projection matrix from `persp` is returned. This information can be used to add additional 3-d features to the plot, such as `pushpin`. See the `persp` help file for an example how to add additional points and lines using the `trans3d` function and also the example below.

`drape.color` returns a list with `color.index` and the numerical `breaks`.

Using the built in `volcano` data set in R, it is easy to see this wrapper on `persp` is very useful, especially using the `zlim2` argument.

```
# an obvious choice:
# Dr. R's favorite New Zealand Volcano!
data( volcano)
M<- nrow( volcano)
N<- ncol( volcano)
x<- seq( 0,1,,M)
y<- seq( 0,1,,N)
drape.plot( x,y,volcano, col=terrain.colors(128))-> pm
```

```
# use different range for color scale and persp plot
# setting of border omits the mesh lines
drape.plot( x,y,volcano, col=terrain.colors(128),zlim=c(0,300),
            zlim2=c( 120,200), border=NA) -> pm2
# note tranparent color for facets outside the zlim2 range
max( volcano)-> zsummit
ix<- row( volcano)[volcano==zsummit]
iy<- col( volcano)[volcano==zsummit]
trans3d( x[ix], y[iy],zsummit,pm2)-> uv
points( uv, col="magenta", pch="+", cex=2)
```

```
# overlay volcano wireframe with gradient in x direction.
dz<- (
  volcano[1:(M-1), 1:(N-1)] - volcano[2:(M), 1:(N-1)] +
    volcano[1:(M-1), 2:(N)] - volcano[2:(M), 2:(N)]
)/2
# convert dz to a color scale:
zlim<- range( c( dz), na.rm=TRUE)
zcol<-drape.color( dz, zlim =zlim)$color.index
# wireframe with these colors
persp( volcano, col=zcol, aRange=30, phi=20)
```

```
## Warning in persp.default(volcano, col = zcol, aRange = 30, phi = 20): "aRange" is not a graphical pa
```

```
# add legend using image.plot function
image.plot( zlim=zlim, legend.only =TRUE, horizontal =TRUE, col=tim.colors(64))
```

4.3 Color palettes

Colors in R can be represented as three vectors in RGB coordinates and these coordinates are interpolated separately using a cubic spline to give color values that are intermediate to the specified colors.

`tim.colors` gives a very nice spectrum that is close to the `jet` color scale in MATLAB® and is the default for `image.plot` and the like.

`larry.colors` is particularly useful for visualizing fields of climate variables.

`two.colors` is really about three different colors. For other colors try `fields.color.picker` to view possible choices. `start="darkgreen"`, `end="azure4"` are the options used to get a nice color scale for rendering aerial photos of ski trails. (http://www.image.ucar.edu/Data/MJProject)

`designer.colors` is the master function for `two.colors` and `tim.colors`. It can be useful if one wants to customize the color table to match quantiles of a distribution. e.g. if the median of the data is at .3 with respect to the range then set `x` equal to c(0,.3,1) and specify three colors to provide a transtion that matches the median value. In `fields` language, this function interpolates between a set of colors at locations `x`. While you can be creative about choosing these colors, just using another color scale as the basis is easy. For example, `designer.color( 256, rainbow(4), x= c( 0,.2,.8,1.0))` leaves the choice of the colors to Dr. R after a thunderstorm.

`color.scale` assigns colors to a numerical vector in the same way as the `image` function. This is useful to keep the assigment of colors consistent across several vectors by specifiying a common `zlim` range. Finally, `plotColorScale` is a simple function to plot a vector of colors to examine their values.

Also, don't forget the built in `heat.colors`, `topo.colors`, `terrain.colors`, `grey.colors`, etc.

Basic Usage

> tim.colors(n = 64, alpha=1.0)
> larry.colors()
> two.colors(n=256, start="darkgreen", end="red", middle="white",alpha=1.0)
> designer.colors(n=256, col= c("darkgreen", "white", "darkred"), x=seq(0,1,, length(col)) ,alpha=1.0)

Value

A vector giving the colors in a hexadecimal format, two extra hex digits are added for the alpha channel.

First we take a look at some of the color options.

```
set.panel( 2,2)
z<- outer( 1:20,1:20, "+")
obj<- list( x=1:20,y=1:20,z=z )
image( obj, col=tim.colors( 200), main="Tim") # 200 levels
image( obj, col=two.colors(), main="Default two.colors")
image( obj, col=larry.colors(), main="Larry")
image( obj, col=two.colors(start="red", end="blue", middle="green"), main="RGB two.colors")
```

Here is an example using designer colors.

```
coltab<- designer.colors(col=c("blue", "grey", "green"), x= c( 0,.3,1) )
image( obj, col= coltab )
```

Note that using tranparency without alpha in the image plot would cover points.

```
set.panel(1,2)
plot( 1:20,1:20)
image(obj, col=two.colors(alpha=.5), add=TRUE)
plot(1:20,1:20)
image(obj, col=two.colors(), add=TRUE)
```

##`interp.surface`

Finally, we discuss `interp.surface`, which performs bilinear interpolation on a regular grid. Bilinear interpolation is common in image processing: it is an easy way to to fill in an image and avoid sharp pixel boundaries. Note that bilinear interpolation can produce some artifacts related to the grid and not reproduce higher behavior in the surface. For, example the extrema of the interpolated surface will always be at the parent grid locations. There is nothing special about bilinear interpolation to another grid, this function just includes a for loop over one dimension and a call to the function for irregular locations. It was included in fields for convenience since the grid format is so common.

We recommend this method for fast visualization and presentation of images. However, if the actual interpolated values will be used for analysis, use a statistical method such as `Tps` or `fastTps` to get an interpolation that is based on first principles.

Here is a brief explanation of the interpolation: Suppose that the location (`locx`, `locy`) lies in between the first two grid points in both x and y. That is, `locx` is between x1 and x2 and `locy` is between y1 and y2. Let $e_x = (l_1 - x_1)/(x_2 - x_1)$ and $e_y = (l_2 - y_1)/(y_2 - y_1)$. The interpolant is

$$(1 - e_x)(1 - e_y)z_{11} + (1 - e_x)(e_y)z_{12} + (e_x)(1 - e_y)z_{21} + (e_x)(e_y)z_{22}$$

where the z's are the corresponding elements of the `Z` matrix. See also the `akima` package for fast interpolation from irregular locations.

Basic Usage

interp.surface(obj, loc)
interp.surface.grid(obj, grid.list)

Value

`interp.surface` returns a vector of interpolated values. `NA` are returned for regions of the `obj$z` that are `NA` and also for locations outside of the range of the parent grid.

In this example, we take a subset of the `lennon` image and predict at several random points and also on a grid. We show how predicting on a fine grid can give us seemingly higher resolution when plotting.

```
# evaluate an image at a finer grid
# take a subset of the lennon image
data( lennon)
# create an example in the right list format like image or contour
obj<- list( x= 1:20, y=1:20, z= lennon[ 201:220, 201:220])
set.seed( 123)
# lots of random points
N<- 500
loc<- cbind( runif(N)*20, runif(N)*20)
z.new<- interp.surface( obj, loc)
# compare the image with bilinear interpolation at scattered points
set.panel(1,2)
image.plot( obj, main="Original Image",zlim=c(20,110))
quilt.plot( loc, z.new, main="Bilinear interp at random points",add.legend=FALSE,zlim=c(20,110))
```

```
# sample at 100X100 equally spaced points on a grid
grid.list<- list( x= seq( 1,20,,100), y= seq( 1,20,,100))
interp.surface.grid( obj, grid.list)-> look
# take a look
set.panel(1,2)
image.plot( obj, main="Original image",zlim=c(20,110))
image( look, main="Bilinear interpolation on grid",col=tim.colors(),zlim=c(20,110))
```

#Covariance Models Revisited

We have seen in the sections on spatial statistics that the crux of modeling spatial data lies in choosing a covariance model and specifying its parameters. In this section we discuss the remaining covariance functions available in `fields`. (Note: the package documentation lists all of the options under `Covariance functions`).

Recall that these functions take one or two sets of locations and compute the (cross) covariance matrix. Most commonly fit are the exponential, Gaussian, and covariance Matern models we have discussed. There are a wealth of other options. We first discuss radial basis functions (RBFs).

##`Rad.cov`, `Rad.cov.simple`, `cubic.cov`, and `wendland.cov`

Radial basis functions are the natural covariance that comes out of the thin plate spline minimization problem (see the section `Tps` Revisited). The functional form is the following:

$$\text{Cov}(Y(\mathbf{x}), Y(\mathbf{x}')) = C(r) = c_0(m,d) * \begin{cases} r^{2m-d}, & d \text{ odd} \\ r^{2m-d}\ln(r), & d \text{ even} \end{cases}$$

where $r = \|\mathbf{x} - \mathbf{x}'\|$ and $c_0(m,d)$ is a multiplicative constant depending on m and d.

The argument `d` is inferred from the dimension of the input data. If `m` isn't specified by the user, by default it is set to the smallest integer such that $p = 2m - d$ is positive. Alternatively, the user can specify `p`.

Note that these RBFs are generalized covariance functions, which means they are only valid covariances when restricted to a subspace of linear combinations of the field. The function `Rad.simple.cov` is a straightforward implementation in R code (computes fewer checks on the data). It can be used like `Exp.cov` to write a new covariance function.

`cubic.cov` is a specific case of `Rad.cov` where we set `d=1` and `m=1`.

All of these covariance functions can be equipped to take advantage of sparsity in the covariance matrix by using the `SPAM` package. The functions `Wendland` and `wendland.cov` allow easy use of the Wendland compactly supported polynomials as covariance functions. This will output a sparse covariance matrix, which helps shorten the computation time when working with large datasets. The main argument (besides location) to in the `Wendland` function is `aRange`, which by default is set to 1. This is the taper range, or the radius of support of the function. The Wendland function is identically 0 for distances greater than `aRange`. If the user would like to scale each coordinate independently, they can provide a matrix or the diagonal of a matrix for the argument `V`, which is the inverse linear transformation of the coordinates before distances are found. Another parameter that can be chosen is `k`, the order of the Wendland polynomial, which determines the smoothness.

```
## Warning in persp.default(gl$x, gl$y, RBF, aRange = 30, phi = 20, shade = 0.3, : "aRange" is not a gra
```

```
## Warning in title(main = main, sub = sub, ...): "aRange" is not a graphical parameter
```

4.4 `stationary.taper.cov`

Just like `stationary.cov`, `stationary.taper.cov` is a general covariance function for mixing and matching different options. The user can choose the covariance model, its parameters, and the distance function used just like in `stationary.cov`. `stationary.taper.cov` additionally takes the argument `Taper` with default `Wendland`. This function uses `rdist.near` with the `spam` package to efficiently compute a sparse covariance matrix. The resulting cross covariance is the direct (or Schur) product of the tapering function and covariance function. In R code, given location matrices x1 and x2 and using Euclidean distance we have `Cov(rdist(x1,x2)/aRange) * Taper(rdist(x1,x2)/Taper.args$aRange)`.

Basic Usage

Rad.cov(x1, x2, p = 1, m=NA)
Rad.simple.cov(x1, x2, p=1)
cubic.cov(x1, x2, aRange = 1)
stationary.cov(x1, x2=NULL, Covariance = "Exponential", Distance = "rdist",aRange = 1, V = NULL)
stationary.taper.cov(x1, x2, Covariance="Exponential",Taper="Wendland",aRange=1.0,V=NULL)
wendland.cov(x1, x2, aRange = 1, V=NULL, k = 2)

Value

These functions all return the cross-covariance matrix. In general if `nrow(x1)=m` and `nrow(x2)=n` then the returned matrix will be m by n.

4.5 Writing your own covariance function

Construction of covariance functions in `fields` is straightforward. Once built, we can then pass the new covariances into `spatialProcess`, `mKrig`, or `Krig` to make covariance matrices.

If the user creates a function (e.g. `spherical(d,aRange)`) that takes distances as an argument, they can use this in `stationary.cov`.

Suppose we want to create the isotropic spherical covariance, which is compactly supported and only depends on the parameter distance $r = \|\mathbf{x} - \mathbf{x}'\|$ and the range $a $:

$$C(r) = \left(1 - \frac{3r}{2a} + \frac{1}{2}\left(\frac{r}{a}\right)^3\right) \cdot \mathbf{1}_{\{0 \le r < a\}}$$

Here, we show two alternatives to constructing and using a covariance function in `fields`.

```
  spherical <-  function (x1, x2=NULL, aRange=1, C=NA, Distance="rdist" ) {
      if (is.null(x2)) {x2 <- x1}
      if (Distance == "rdist.earth"){
        r <- rdist.earth(x1,x2)
      }else {r <- rdist(x1,x2) }
      if (is.na(C[1]) ) {
          return( ifelse(r<aRange , 1 - 3*r / (2*aRange) + (1/2) * (r/aRange)^3,0))
      }
      if (!is.na(C[1])) {
          return( ifelse(r<aRange , 1 - 3*r / (2*aRange) + (1/2) * (r/aRange)^3,0) %*% C)
      }
  }


spherical.d <- function(d, aRange=1)
{
  return( ifelse(d<aRange , 1 - 3*d / (2*aRange) + (1/2) * (d/aRange)^3,0))
}
```

Now we can use `Krig` or `mKrig` with our newly constructed covariances.

```
data(COmonthlyMet)
y <- CO.tmax.MAM.climate
x <- CO.elev
grd <- as.matrix(CO.loc)
```

```
keep <- !is.na(y)      #removing all NA's from the temperatures
y <- y[keep]
x <- x[keep]
grd <- grd[keep,]
rm(keep)

fit <- Krig(grd,y, cov.function = "spherical",aRange=60, Distance="rdist.earth")
fit2 <- Krig(grd,y,Covariance = "spherical.d", aRange=60, Distance="rdist.earth")
fit3 <- Krig(grd,y,Covariance="Matern",aRange=60, Distance="rdist.earth")

rainbow <- rainbow_hcl(n=50)

par(mfrow=c(1,3))
surface(fit, main="Spherical Covariance",col=rainbow)
surface(fit2, main="Spherical Covariance (alternative)",col=rainbow)
surface(fit3, main="Matern Covariance",col=rainbow)
```

Notice that the first two plots are identical and the third plot is different, especially in the bottom right.

4.6 `stationary.taper.cov` details

Using `stationary.taper.cov` boils down to multiplying a chosen covariance function by a Wendland. (See a direct comparison below). This makes the overall covariance model compactly supported, which in turn speeds up computation time.

```
data( ozone2)
x<- ozone2$lon.lat
y<- ozone2$y[16,]
good<- !is.na( y)
x<- x[good,]
y<- y[good]
# Note that default covariance is exponential and default taper is Wendland (k=2).
temp <- stationary.taper.cov( x[1:3,],x[1:10,], aRange=1.5, Taper.args= list(k=2,aRange=2.0,dimension=2
temp2<- Exp.cov( x[1:3,],x[1:10,], aRange=1.5) * Wendland(rdist(x[1:3,],x[1:10,]),aRange= 2.0, k=2, dim
test.for.zero( as.matrix(temp), temp2)
```

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

Now we visualize the difference in predictions when using a taper.

```
spatialProcess(x,y, cov.function = "stationary.taper.cov", aRange=1.5,cov.args =
       list(Taper.args= list(k=2, dimension=2,aRange=2.0) )) -> out2
spatialProcess(x,y, aRange= 1.5)-> out
set.panel(1,2)
surface(out, main="Exponential",smallplot= c(.89,.9,0.2,.8) ,zlim=c(15,170),col=rev(heat_hcl(50)))
surface(out2, main="Exponential * Wendland",smallplot= c(.89,.9,0.2,.8),zlim=c(15,170),col=rev(heat_hcl
```

5 mKrig

The `mKrig` function is both a Kriging/linear algebra engine and also user-level function for fitting spatial models. The `mKrig` engine is a simple and fast version of `Krig`. The "m" stands for micro, being a succinct (and we hope readable) function. The function focuses on the same computations as in `Krig.engine.fixed` done for a fixed `lambda` parameter, for unique spatial locations and for data without missing values. `mKrig` is optimized for large data sets, sparse linear algebra, and a clear exposition of the computations. The underlying code in `mKrig` is more readable and straightforward, but it does fewer checks on the data than `Krig`. One savings in computation is that Monte Carlo simulations are used to compute the trace of the smoothing matrix $\mathrm{tr}(A(\lambda))$, which can be used for the effective degrees of freedom of the model.

Just as in other functions, we can use `cov.function` and `cov.args` to specify which covariance we want to use in the model. `mKrig` is often used with a compactly supported covariance to take advantage of sparsity in computations. See `fastTps` as an example of how it uses `mKrig`.

There are other arguments that can be specified in `chol.args` for efficiency when dealing with a large data set. For example, `nnzR` is a guess at how many nonzero entries there are in the Cholesky decomposition of the linear system used to solve for the basis coefficients. Specifying this to increase size will help to avoid warnings from the `spam` package.

`mKrig.trace` is an internal function called by `mKrig` to estimate the effective degrees of freedom. The Kriging surface estimate at the data locations is a linear function of the data and can be represented as $A(\lambda)y$.

The trace of A is one useful measure of the effective degrees of freedom used in the surface representation. In particular this figures into the GCV estimate of the smoothing parameter. It is computationally intensive to find the trace explicitly, but there is a simple Monte Carlo estimate that is often very useful. If $\mathbf{E}$ is a vector of iid $N(0,1)$ random variables then the trace of A is the expected value of E^TAE. Note that AE is simply predicting a surface at the data location using the synthetic observation vector E. This is done for $N \cdot \mathrm{tr}(A)$ independent $N(0,1)$ vectors and the mean and standard deviation are reported in the `mKrig` summary. Typically as the number of observations is increased, this estimate becomse more accurate. If $N \cdot \mathrm{tr}(\mathrm{A})$ is as large as the number of observations *np*, then the algorithm switches to finding the trace exactly based on applying A to *np* unit vectors.

`mKrig` has been written to handle multiple data sets. Specifically, if the spatial model is the same except for different observed values (such as multiple days of observations), one can pass `y` as a matrix and the computations are done efficiently for each set. Note that this is not a multivariate spatial model just an efficient computation over several data vectors without explicit looping. See the example for the defaults.

<In the examples below we frequently use the S3 methods available to `mKrig` objects. `print/summary` and `surface` are used for model diagnostics and quick visualizaiton. `predict` and `predictSurface` can predict the model at the observations or an arbitrary region (derivatives supported with Wendland covariance). `predictSE` gives model uncertainty, and>

Finally, `sim.mKrig.approx` has the capability to quantify the uncertainty in the estimated function using conditional simulation.

Basic Usage

mKrig(x, y, Z = NULL, cov.function = "stationary.cov", lambda = 0, m = 2)

Value

Returns an object of class `mKrig`.

##`ozone2`

First we show that we can reproduce the `Krig` computations using `mKrig`. The algorithm/parameter estimation is done differently for the two functions, so this is a useful check. We fit an exponential covariance with range equal to 2 with both functions.

```
# Midwest ozone data 'day 16' stripped of missings
data( ozone2)
y<- ozone2$y[16,]
good<- !is.na( y)
y<-y[good]
x<- ozone2$lon.lat[good,]
out<- mKrig( x,y, aRange = 2.0, lambda=.01)
out2 <- Krig( x,y, aRange=2.0, lambda=.01)
```

```
## HERE
```

```
grid.list <- fields.x.to.grid(x)
grid <- make.surface.grid(grid.list)
out.p <- matrix(predict(out,grid),nrow=80,ncol=80)
out.p2 <- matrix(predict(out2,grid),nrow=80,ncol=80)
test.for.zero(out.p, out.p2)
```

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

##COmonthlyMet

We use this data set in almost every chapter for completeness. Here we quickly show how to use `mKrig` with a `Z` covariate.

```
data(COmonthlyMet)
yCO<- CO.tmin.MAM.climate
good<- !is.na(yCO)
yCO<-yCO[good]
xCO<- CO.loc[good,]
Z<- CO.elev[good]
out<- mKrig(xCO,yCO, Z=Z, cov.function="stationary.cov", Covariance="Matern", aRange=4.0, smoothness=1.
pred0 <- predict(out, grid.list=CO.Grid, Z= as.vector(CO.elevGrid$z))
pred <- as.surface(make.surface.grid(CO.Grid) , pred0)
set.panel(1,3)
quilt.plot(xCO, predict(out), main="Data",smallplot= c(.88,.9,0.2,.8),col=blue2yellow(35)) #colorRamps
surface(out, main="Spatial Part",smallplot= c(.88,.9,0.2,.8),col=blue2yellow(35))
surface(pred, main="Full Model",smallplot= c(.88,.9,0.2,.8),col=blue2yellow(35))
```

##flame

The `flame` data consists of a 2 column matrix `x` with different fuel and oxygen flow rates for the burner, and vector `y` with the intensity of light at a particular wavelength indicative of Lithium ions. The characteristics of an ionizing flame are varied with the intent of maximizing the intensity of emitted light for lithuim in solution. Areas outside of the measurements are where the mixture may explode! Note that the optimum is close to the boundary.

This example shows the versatility of `fields`, fitting `Krig`, `spatialProcess`, and `mKrig` models to the data. Note how much faster `mKrig` is, and how even with a Wendland it is able to reproduce the other fits' behaviours fairly well.

```
x <- flame$x
y <- flame$y
look <- as.image(Z=y, x=x)
fit <- Krig(x=x, Y=y, cov.function="stationary.cov", aRange=5)
fit1 <- spatialProcess(x,y)
```

```
fit2 <- mKrig(x, y, cov.function = "wendland.cov", k=2, aRange=4, lambda=0.75)
set.panel(2,2)
image.plot(look, main="Data",smallplot= c(.88,.9,0.2,.8),zlim=c(0.005,0.05),col=brewer.pal(9,'Reds'))
surface(fit, main="Exponential Krig",legend.shrink=0.6,smallplot= c(.88,.9,0.2,.8),zlim=c(0.005,0.05),co
surface(fit1, main="Matern spatialProcess",smallplot= c(.88,.9,0.2,.8),zlim=c(0.005,0.05),col=brewer.pal
surface(fit2, main="Wendland mKrig",smallplot= c(.88,.9,0.2,.8),zlim=c(0.005,0.05),col=brewer.pal(9,'Re
```

##Krig.replicates with NWSC

We now consider the NWSC data set. The data we consider are the temperature and relative humidity as our spatial indexing variables, and our response is power usage. The observations aren't at unique locations, so to use mKrig or spatialProcess, we will have to collapse the observations to unique locations with the function Krig.replicates.

```
library(sp) #for colors
load("NWSC.rda")
x <- cbind(NWSC$RH, NWSC$Otemp)
y <- NWSC$Mpower
quilt.plot(x,y, main="Data with replicates",col=rev(bpy.colors()))
```

```
new.dat <- Krig.replicates(x=x, y=y)
xM <- new.dat$xM
yM <- new.dat$yM
quilt.plot(xM, yM, main="Unique data",col=rev(bpy.colors()))
```

```
###fit <- spatialProcess(xM, yM) takes too long
fit <- mKrig(xM, yM, cov.function="wendland.cov", aRange=2,lambda=2)
#try different aRange and lambda
surface(fit, main="Wendland, aRange=2, lambda=2",col=rev(bpy.colors()))
```

```
pred <- predictSurface(fit)
```

```
set.panel()
library(sp) #for colors
drape.plot(pred, border=NA, aRange=155, phi=25, horizontal=FALSE, xlab="Humidity", ylab="Temp", zlab="P
title("Wendland          ")
```

##mKrig and a GCV grid with ozone2

We return to ozone2 for a final time. We use mKrig to interpolate using a tapered version of the exponential. The taper scale is set to 1.5, and the default taper covariance is the Wendland. Tapering will done at a scale of 1.5 relative to the scaling.

```
data( ozone2)
y<- ozone2$y[16,]
good<- !is.na( y)
y<-y[good]
x<- ozone2$lon.lat[good,]
out2 <- mKrig( x,y,cov.function="stationary.taper.cov",aRange = 2.0, lambda=.01,
       Taper="Wendland", Taper.args=list(aRange = 1.5, k=2, dimension=2))
```

Now, we will compare many lambda's on a grid using GCV for this small data set. One should really just use Krig or Tps, but this is an example of approximate GCV that will work for much larger data sets using sparse covariances and the Monte Carlo trace estimate.

```r
lgrid<- 10**seq(-1,1,,15)   # a grid of lambdas
GCV<- matrix( NA, 15,20)
trA<- matrix( NA, 15,20)
GCV.est<- rep( NA, 15)
eff.df<- rep( NA, 15)
logPL<- rep( NA, 15)
for( k in 1:15){    # loop over lambda's
  out<- mKrig( x,y,cov.function="stationary.taper.cov", aRange = 2.0,
              lambda=lgrid[k], Taper="Wendland", Taper.args=list(aRange = 1.5, k=2, dimension=2))
  GCV[k,]<- out$GCV.info
  trA[k,]<- out$trA.info
  eff.df[k]<- out$eff.df
  GCV.est[k]<- out$GCV
  logPL[k]<- out$summary["lnProfileLike.FULL"]
}
```

For each lambda, we have fitted an `mKrig` object and then extracted `trA` and `GCV`. Remember `trA` is the effective degrees of freedom `df` which is really `lambda` on another scale. The dashed black lines show the effective degrees of freedom versus the approximate Monte Carlo GCV in `GCV.info`. The average of these MC simulations is the red line. Note how close it is to the solid black line, which the Monte Carlo approximation automatically computed by `mKrig`. We fit a `Krig` object to the same data, which computes the exact GCV, which is shown in dark green. Finally, the negative log profile likelihood is shown in blue.

```r
par(mar=c(5,4,4,6))
matplot( trA, GCV, type="l", col=1, lty=2,
         xlab="effective degrees of freedom", ylab="GCV")
lines( eff.df, GCV.est, lwd=2, col=2)
lines( eff.df, rowMeans(GCV), lwd=2)
# add exact GCV computed by Krig
out0<- Krig( x,y,cov.function="stationary.taper.cov", aRange = 2.0, Taper="Wendland",
            Taper.args=list(aRange = 1.5, k=2, dimension=2), spam.format=FALSE)
lines( out0$gcv.grid[,2:3], lwd=4, col="darkgreen")
# add profile likelihood
utemp<- par()$usr
utemp[3:4] <- range( -logPL)
par( usr=utemp)
lines( eff.df, -logPL, lwd=2, col="blue", lty=2)
axis( 4)
mtext( side=4,line=3, "-ln profile likelihood", col="blue")
title( "GCV ( green = exact) and -ln profile likelihood", cex=2)
```

##Collapse fixed effects with `ozone2`

```r
# fits for first 10 days from ozone data
data( ozone2)
NDays<- 10
O3MLE<-NULL
for( day in 1: NDays){
  ind<- !is.na(ozone2$y[day,] )
  x<- ozone2$lon.lat[ind,]
  y<- ozone2$y[day,ind]
  tmpFit<- spatialProcess( x,y,Distance="rdist.earth")
  O3MLE<- rbind( O3MLE, tmpFit$MLESummary)
}
# last two columns are repeated
O3MLE<- O3MLE[, -c(11,12)]
```

```
plot( O3MLE[,"lambda"], O3MLE[,"aRange"], log = 'xy', xlab = "lambda", ylab = "aRange")
title("aRange vs lambda MLEs for 10 days")
```

```
plot( O3MLE[,"eff.df"], O3MLE[,"aRange"], log = 'xy', xlab = "eff.df", ylab = "aRange")
title("aRange vs effective degrees of freedom  for 10 days")
```

An alternative to fitting a different mean part of the model for each day is to pool all of the days and compute the fixed effects by averaging. The function `mKrig.coef` takes care of this for us. One just needs to pass an `mKrig` object, and a vector `y` of new observations. If `y` is a matrix, the columns are treated as multiple realizations of the same field.

Basic Usage

mKrig(x, y, collapseFixedEffect = TRUE)
predict(object, collapseFixedEffect = object$collapseFixedEffect)
mKrig.coef(object, y, collapseFixedEffect=TRUE)

Value

`mKrig.coef` finds the `d` and `c` coefficients representing the solution using the previous cholesky decomposition for a new data vector. This is used in computing the prediction standard error in `predictSE.mKrig` and can also be used to evalute the estimate efficiently at new vectors of observations provided the locations and covariance remain fixed.

Since this data set has `NA`'s, we remove any columns (observation locations) with an `NA`. Beware of blindly removing data like this - we are just illustrating the `mKrig.coef` functionality.

```
goody <- na.omit(t(ozone2$y))
omitted <- attr(goody, "na.action")
x <- ozone2$lon.lat[-omitted,]
Y <- goody
out <- mKrig(x, Y[,1], Distance="rdist.earth")
out2 <- mKrig(x, Y, Distance="rdist.earth",collapseFixedEffect=TRUE)
comp <- rbind(c(out$beta),out2$beta[1:3,1])
rownames(comp) <- c("Day 1", "All Days")
comp
```

```
##               [,1]     [,2]       [,3]
## Day 1    156.0267 1.678379  0.7792557
## All Days 177.2253 1.018249 -0.9177961
```

After collapsing the fixed effects, we can predict! We compare the original day 1 fit versus the collapsed fit based on all the days.

```
xnew <- fields.x.to.grid(x)
pred1 <- predictSurface(out2, xnew, collapseFixedEffect = TRUE)
pred2 <- predictSurface(out, xnew)
set.panel(1,2)
```

```
surface(pred2, main="Day 1 Fit",col=brewer.pal(9,'Spectral'))
surface(pred1, main="Collapsed Fit",col=brewer.pal(9,'Spectral'))
```

5.1 Approximate standard errors in mKrig with ozone2

```
data( ozone2)
y<- ozone2$y[16,]
good<- !is.na( y)
y<-y[good]
x<- ozone2$lon.lat[good,]
xMissing<- ozone2$lon.lat[!good,]

O3.fit<- mKrig( x,y, Covariance="Matern", aRange=.5,smoothness=1.0, lambda= .01 )
```

<surface(O3.fit,col=brewer.pal(9,‘Spectral’))>

Now, we’ll look at some approximate conditional simulations of our random field and the missing data.

```
set.seed(122)

O3.sim<- sim.mKrig.approx( O3.fit, nx=100, ny=100, gridRefinement=3, M=2 )
O3.sim.missing <- sim.mKrig.approx( O3.fit, xMissing, nx=80, ny=80, gridRefinement=3, M=2 )

set.panel(2,2)

  for (k in 1:2){
    image.plot( as.surface( O3.sim$predictionPoints, O3.sim$Ensemble[,k]),zlim=c(-70,230),col=brewer.pa
    title("Simulation")
  }
for(k in 1:2){
    image.plot( as.image(O3.sim.missing$Ensemble[,k],  O3.sim.missing$predictionPoints,nx=24,ny=24) ,zl
    title("Simulation for Missing Data")
}
```

5.2 Finding taper range

Here is a series of examples with bigger datasets that are randomly generated. We use a compactly supported covariance directly.

```
set.seed(334)
N<- 1500
x<- matrix(2*(runif(2*N)-.5),ncol=2)
values <- sin( 1.8*pi*x[,1])*sin( 2.5*pi*x[,2])

y<- values + rnorm(N)*.1
look2 <- mKrig( x,y, cov.function="wendland.cov",k=2, aRange=.2,lambda=.1)
predictSurface(look2) -> out.p
surface( out.p,col=brewer.pal(9,'Blues'))
title("Surface")
```

Even though there are a large number of points, this works because the number of nonzero elements within distance `aRange` are less than the default maximum allocated size of the sparse covariance matrix. See `options()` for the default values. The following will give a warning for `aRange=.9` because allocation for the covariance matirx storage is too small. Here `aRange` controls the support of the covariance and so indirectly the number of nonzero elements in the sparse matrix.

```
look2<- mKrig(x,y, cov.function="wendland.cov",k=2, aRange=.9, lambda=.1)
```

```
## Warning in nearest.dist(structure(c(0.963543639518321, 0.0860745231620967, : You ask for a 'dense' sp
## To avoid the iteration, increase 'nearestdistnnz' option to something like
## 'options(spam.nearestdistnnz=c(750000,400))'
## (constructed 1213 lines out of 1500).

## Warning in nearest.dist(structure(c(0.963543639518321, 0.0860745231620967, : You ask for a 'dense' sp
## To avoid the iteration, increase 'nearestdistnnz' option to something like
## 'options(spam.nearestdistnnz=c(750000,400))'
## (constructed 1213 lines out of 1500).
```

The warning resets the memory allocation for the covariance matrix according the to values `options(spam.nearestdistnnz=c(416052,400))`. This is inefficient because the preliminary pass failed. The following call completes the computation in “one pass” without a warning and without having to reallocate more memory.

```
options(spam.nearestdistnnz=c(4.2E5,400))
look2<- mKrig( x,y, cov.function="wendland.cov",k=2, aRange=.9, lambda=1e-2)
```

```
## Warning in nearest.dist(structure(c(0.963543639518321, 0.0860745231620967, : You ask for a 'dense' sp
## To avoid the iteration, increase 'nearestdistnnz' option to something like
## 'options(spam.nearestdistnnz=c(600000,400))'
## (constructed 966 lines out of 1500).

## Warning in nearest.dist(structure(c(0.963543639518321, 0.0860745231620967, : You ask for a 'dense' sp
## To avoid the iteration, increase 'nearestdistnnz' option to something like
## 'options(spam.nearestdistnnz=c(600000,400))'
## (constructed 966 lines out of 1500).
```

As a check notice that `print(look2)` reports the number of nonzero elements consistent with the specifc allocation increase in `spam.options`.

Now we generate a new data set of 1500 locations.

```
set.seed( 234)
N <- 1500
x <- matrix( 2*(runif(2*N)-.5),ncol=2)
y <- values + rnorm(N)*.01
```

The following is an example of where the allocation (for `nnzR`) for the cholesky factor is too small. A warning is issued and the allocation is increased by 25. To avoid the warning, use the second call to `mKrig`.

```
look2<- mKrig( x,y,cov.function="wendland.cov",k=2, aRange=.1, lambda=1e2 )

look2<-mKrig( x,y,cov.function="wendland.cov", k=2, aRange=.1,lambda=1e2,
              chol.args=list(pivot=TRUE, memory=list(nnzR = 450000)))
```

Next, we show how to fit multiple data sets observed at the same locations.

```
y1 <- values + rnorm(N)*.01
y2 <- values + rnorm(N)*.01
Y <- cbind(y1, y2)
look3<- mKrig( x,Y,cov.function="wendland.cov",k=2,aRange=.1, lambda=1e2 )
print( look3)
```

```
## [1]  0.013450047 -0.008442535  0.009224268
## Call:
## mKrig(x = x, y = Y, cov.function = "wendland.cov", lambda = 100,
##     k = 2, aRange = 0.1)
##
##  Number of Locations:                          1500
##  Number of data sets fit:                      2
##  Degree of polynomial null space ( base model): 1
##  Total number of parameters in base model      3
##   Estimate Eff. degrees of freedom             18.2
##      Standard Error of Eff. Df                 0.769
##  Smoothing parameter                           100
##  Nonzero entries in covariance                 18614
##
##
## Summary of fixed effects
##     estimate       SE pValue
## d1  0.013450 0.009545 0.1588
## d2 -0.008443 0.016670 0.6125
## d3  0.009224 0.016320 0.5718
##
## Covariance Model: wendland.cov
##    Non-default covariance arguments and their values
##    Argument: k  has the value(s):
## [1] 2
##    Argument: aRange  has the value(s):
## [1] 0.1
```

Note the slight difference in the summary output because two data sets have been fit.

<We continue by showing a method for finding a good choice for `aRange` as a taper. Suppose the target `aRange` is a spatial prediction using roughly 50 nearest neighbors (tapering covariances is effective for roughly 20 or more in the situation of interpolation) see Furrer, Genton and Nychka (2006). We begin with a random sample of 100 points to get an idea of scale and save computation time.>

<We declare `min(hold2)` is close enough, so now the following will use no less than 55 nearest neighbors due to the tapering.>

5.3 Using `V`, `mKrig`,`Tps` with `RMprecip`

In this example we return to the Rocky Mountain precipitation data. We make a grid and append our data observations to it, then interpolate using a `Tps` model. We compare this to an `mKrig` model using a Wendland covariance. We also use the `V` argument in `mKrig`, which is a matrix that describes the inverse linear transformation of the coordinates before distances are found. In R code this transformation is `x1 %*% t(solve(V))`. The default is `NULL`, i.e. $V = a \cdot \mathbf{I}_{n\times n}$. If one has a vector of `aRange` values that are the scalings for each coordinate then just express this as `V = diag(aRange)` in the call to this function.

Again, `smallplot` is used strictly for formatting this document.

```
library(gstat)
data(coalash)
x <- cbind(coalash$x, coalash$y)
y <- coalash$coalash
```

```r
pg <- brewer.pal(9,'PRGn')
quilt.plot(x, y,col=pg)
```

```r
#equispaced grid with length 1 in both lon and lat
dlon <- 1
dlat <- 1
V <- diag( c(2.5*dlon, 2.5*dlat) )

# this is an interplotation of the values using a compact support but Tps-like covariance.
out2 <- mKrig( x, y, cov.function="wendland.cov",k=4, V=V, lambda=1)
look <- predictSurface( out2, nx=400, ny=400)

surface(look, main="mKrig Wendland",smallplot= c(.88,.9,0.2,.8),col=pg)
US(add=TRUE, col="white")
```

6 Tps and fastTps

6.1 Tps and Krig

We return to Tps, but this time consider spatial data. The Tps estimate can be thought of as a special case of a spatial process estimate using a particular generalized covariance function. Because of this correspondence the Tps function is a wrapper that determines the polynomial order for the null space and then calls Krig with the radial basis generalized covariance rad.cov. The RBF is the reproducing kernel that generates the reproducing kernel Hilbert space in which the minimization problem is set. See Wahba's *Spline Models for Observational Data*.

In this way Tps requires little extra coding beyond the Krig function itself, and in fact the returned list from Tps is a Krig object and so it is adapted to all the special functions developed for this object format. One downside is some confusion in the summary of the thin plate spline as a spatial process type estimator. For example, the summary refers to a covariance model and maximum likelihood estimates. This is not something usually reported with a spline fit! Similarly, fastTps is a convenient wrapper function that calls mKrig with the Wendland covariance function.

Basic Usage

Tps(x, Y, m = NULL, p = NULL, lon.lat = FALSE)
fastTps(x, Y, m = NULL, p = NULL, aRange, lon.lat=FALSE, lambda=0)

Value

Both functions returns a list of class Krig. m determines the degree (m-1) of the polynomial function capturing the spatial drift/trend component of the model. The default is the value such that 2m-d is greater than zero where d is the dimension of x. p is the polynomial power for Wendland radial basis functions. The default is 2m-d. aRange is the tapering range that is passed to the Wendland compactly supported covariance. The covariance (i.e. the radial basis function) is zero beyond range aRange. The larger aRange is, the closer this model will approximate the standard thin plate spline.

Some care needs to exercised in specifying the taper range aRange and power p which describes the polynomial behavior of the Wendland at the origin. Note that unlike Tps the locations are not scaled to unit range and this can cause havoc in smoothing problems with variables in very different units. So rescaling the locations x <- scale(x) is a good idea for putting the variables on a common scale for smoothing. This function does have the potential to approximate estimates of Tps for very large spatial data sets. Also, the function predictSurface.fastTps has been made more efficient for the case of k = 2 and m = 2.

6.2 ChicagoO3

We return to the simple Chicago ozone measurement data, fitting models with different inputs for `df`. Remember `lambda` and `df` both represent the smoothness of the model on different scales. One of the great things about `Tps` is that the `predict` function is quite flexible: one can pass in a different `df` or `lambda` value to predict with. For example, the first call to `predict` in the example below evaluates the estimate at `lambda = 2` instead of the GCV estimate. It does this very efficiently from the `Krig` fit object.

```
fit <- Tps(ChicagoO3$x, ChicagoO3$y)
surface(fit)
```

```
look <- predict( fit, lambda=2.0)
look <- predict( fit, df=7.5)
set.panel(2,2)
for (i in seq(0,7.5,,4)){
  look<- predictSurface(fit, df=i)
  image.plot(look,smallplot= c(.88,.9,0.2,.8),zlim=c(36,43))
}
```

Even if `lambda` or `df` is specified in the `Tps` call, the GCV grid search is still performed and the information stored in the object. In the third panel of the diagnostic plot, the GCV function has a vertical line at the value of `df` given.

```
fit1<- Tps(ChicagoO3$x, ChicagoO3$y,df=7.5)
set.panel(2,2)
plot(fit1)
```

The basic matrix decompositions are the same for both `fit` and `fit1` objects. `predict(fit1)` is the same as `predict(fit, df=7.5)`, and `predict(fit1, lambda=fit$lambda)` is equivalent to `predict(fit)`.

##COmonthlyMet

To be thorough, we include an example of using `Tps` with a `Z` covariate.

```
data( COmonthlyMet)
obj <- Tps( CO.loc, CO.tmin.MAM.climate, Z= CO.elev)
out.p <-predictSurface(obj, grid.list=CO.Grid, ZGrid= CO.elevGrid,extrap=TRUE)
library(sp)
image.plot( out.p, col=brewer.pal(9,"RdGy"))
US(add=TRUE, col="grey")
contour( CO.elevGrid, add=TRUE, levels=c(1000,1500, 2000, 2500, 3500), col="black")
```

<Again, we use elevation data as a covariate, but this time with precipitation data instead of pollutant data. We fit `Tps` models with and without the covariate.>

<We cannot set `drop.Z` to `TRUE` unless we have a set of elevations at the prediction locations available. We fit several models for comparison. `fit.full` is the full model, `fit.fixed` is the linear fixed part of the second spatial model including intercept, lon, lat, and elevation. `fit.smooth` is the smooth spatial part but also with the linear lon, lat, and intercept terms.>

##BD

The models in `fields` can accept higher dimensional data sets than we have been using so far. The default when using `surface` with this type of data is to hold 3rd and 4th dimensions fixed at the median values.

```r
str(BD)
```

```
## 'data.frame':    89 obs. of  5 variables:
##  $ KCl  : num  30 30 20 50 25 10 40 15 50 10 ...
##  $ MgCl2: num  5 5 4 7 7 4 4 6 6 7 ...
##  $ KPO4 : num  25 25 20 20 30 25 20 45 35 25 ...
##  $ dNTP : num  625 625 1500 250 1500 1250 1250 1250 1500 1250 ...
##  $ lnya : num  12.9 12.67 9.17 9.87 9.88 ...
```

```r
fit <- Tps(BD[,1:4],BD$lnya,scale.type="range")
surface(fit,col=rev(cm.colors(50)))
```

##fastTps with several lambda values

The `fastTps` function is actually a wrapper for the `mKrig` function with the compactly supported Wendland covariance function. The taper range `aRange` and the the polynomial power for the Wendland rbf `p` must be chosen. The defaults are `aRange = 1` and `p = 2m-d` where `m` is chosen such that `p` is positive.

Both `Tps` and `fastTps` return a list of class `Krig` which includes fitted values, the predicted surface at the observations, and the residuals. The list also includes the values of the GCV grid search.

We start by showing how different `lambda` values affect the `fastTps` fit using the Rocky Mountain precipitation data.

```r
hc <- rev(heat_hcl(12,c=c(80,30),l=c(30,90),power=c(1/5,1.5)))

quilt.plot(RMprecip$x,RMprecip$y,col=hc)
```

```r
fastTps( RMprecip$x,RMprecip$y,m=2,lambda= 1e-2, aRange=3.0) -> out1
fastTps( RMprecip$x,RMprecip$y,m=2,lambda= 1e-1, aRange=3.0) -> out2
fastTps( RMprecip$x,RMprecip$y,m=2,lambda= 1, aRange=3.0) -> out3
```

```r
set.panel(1,3)
surface(out1, main="lambda=1e-2",col=hc)
US( add=TRUE, col="grey")
surface(out2, main="lambda=1e-1",col=hc)
US( add=TRUE, col="grey")
surface(out3, main="lambda=1",col=hc)
US( add=TRUE, col="grey")
```

Note that setting `aRange` to 3 degrees is a very generous taper range. Use some trial `aRange` values with `rdist.nearest` to determine a a useful taper (see the examples in `mKrig`). Some empirical studies suggest that in the interpolation case in 2d, the taper should be large enough so there are about 20 non-zero nearest neighbors for every location.

Now we use great circle distance for this smooth. Note that when `lon.lat=TRUE`, `aRange` for the taper support is automatically converted to degrees.

```r
fastTps( RMprecip$x,RMprecip$y,m=2,lambda= 1e-2,lon.lat=TRUE, aRange=300) -> out4
out3$eff.df
out4$eff.df
```

Note the difference in the effective degrees of freedom between the two models.

```r
set.panel(1,2)
surface(out1, main="lon.lat=FALSE",smallplot= c(.88,.9,0.2,.8),col=rev(heat_hcl(12,c=c(80,30),l=c(30,90
US( add=TRUE, col="grey")
```

```
surface(out4, main="Great circle distance",smallplot= c(.88,.9,0.2,.8),col=rev(heat_hcl(12,c=c(80,30),l=
US( add=TRUE, col="grey")
```

##`sim.fastTps.approx` and `ozone2`

In this final example, we show how to use `sim.fastTps.approx`. We produce a local conditional simulation using the covariance from `fastTps`. The `sim.approx` methods evaluate the conditional surface on a grid and simulate the values of $h(x)$ off the grid using bilinear interpolation of the four nearest grid points. Because of this approximation it is important to choose the grid to be fine relative to the spacing of the observations. The advantage of this approximation is that one can consider conditional simulation for large grids beyond the size possible with exact methods. Here, the method for simulation is circulant embedding and so is restricted to stationary fields. The circulant embedding method is known to fail if the domain is small relative to the correlation range. The argument `gridExpansion` can be used to increase the size of the domain to make the algorithm work. `sim.fastTps.approx` is optimized for the approximate thin plate spline estimator in two dimensions and `k=2`. For efficiency, the ensemble prediction locations must be on a grid.

```
data(ozone2)
y<- ozone2$y[16,]
good<- !is.na( y)
y<-y[good]
x<- ozone2$lon.lat[good,]
O3Obj<- fastTps( x,y, aRange=1.5 )
grid.list<- fields.x.to.grid( O3Obj$x, nx=100, ny=100)
O3Sim<- sim.mKrig.approx( O3Obj,predictionPointsList=grid.list,M=5)
# controlling the grids
xR<- range( x[,1], na.rm=TRUE)
yR<- range( x[,2], na.rm=TRUE)
simulationGridList<- list( x= seq(xR[1],xR[2],,400), y= seq( yR[1],yR[2], ,400))
# very fine localized prediction grid
O3GridList<- list( x= seq( -90.5,-88.5,,200), y= seq( 38,40,,200))
O3Sim<- sim.mKrig.approx( O3Obj, M=5, predictionPointsList=O3GridList, simulationGridList = simulationGr
set.panel(1,2)
plot( O3Obj$x, main="Simulation 1",xlab="",ylab="")
US( add=TRUE)
image.plot( as.surface( O3GridList,O3Sim$Ensemble[,2] ), add=TRUE,smallplot= c(.94,.96,0.2,.8),zlim=c(15
points( O3Obj$x, pch=16, col="magenta")
plot( O3Obj$x, main="Simulation 2 (same z-scale)",xlab="",ylab="")
US( add=TRUE)
image( as.surface( O3GridList,O3Sim$Ensemble[,1] ), add=TRUE,zlim=c(15,140),col=tim.colors())
points( O3Obj$x, pch=16, col="magenta")
```

#Appendix

This appendix will formulate the kriging estimates described in the vignette, as well as the theory behind the parameter estimation in `fields`. Here, we change the kriging notation, where we now absorb the low-order spatial polynomial $P(\cdot)$ into the covariate matrix $\mathbf{Z}$.

##Kriging theory

There are three main types of kriging.

1. Simple Kriging. Assume the mean function $\mathbf{Zd}$ is spatially constant and known; denote it as μ. Thus our model is $Y(\mathbf{s}) = \mu + g(\mathbf{s}) + \epsilon(\mathbf{s})$. It follows that $Y(\mathbf{s}) - \mu$ has mean zero, and hence we krige on the residuals $R(\mathbf{s}) := Y(\mathbf{s}) - \mu$ and add μ for our prediction.

$$\hat{Y}(\mathbf{x}_0) = \mu + \Sigma_0 \Sigma^{-1}(\mathbf{Y} - \boldsymbol{\mu})$$

2. Ordinary Kriging. Again, we assume that the mean function $\mathbf{Zd} = \mu$ is spatially constant, but here we suppose μ is unknown.

$$\hat{Y}(\mathbf{x}_0) = \Sigma_0\Sigma^{-1}\mathbf{Y} + \frac{1 - \Sigma_0\Sigma^{-1}\mathbf{1}}{\mathbf{1}^T\Sigma^{-1}\mathbf{1}}\mathbf{1}^T\Sigma^{-1}\mathbf{Y} = \text{ simple kriging + correction for estimating } \mu$$

3. Universal Kriging. Here, the mean function $\mathbf{Zd}$ is a general linear function. Let $\mathbf{Z}_0$ be the matrix of covariates at the location $\mathbf{x}_0$.

$$\hat{Y}(\mathbf{x}_0) = \left(\Sigma_0 + (\mathbf{Z}_0 - \Sigma_0\Sigma^{-1}\mathbf{Z})(\mathbf{Z}^T\Sigma^{-1}\mathbf{Z})^{-1}\mathbf{Z}^T\right)\Sigma^{-1}\mathbf{Y}$$

In general, the kriging predictor of our `fields` model $Y(\mathbf{x}) = \mathbf{Z}(\mathbf{x})\mathbf{d} + g(\mathbf{x}) + \epsilon(\mathbf{x})$ is given by

$$\hat{Y}(\mathbf{x}_0) = \mathbf{Z}_0\hat{\mathbf{d}} + \Sigma_0(\Sigma + \lambda I)^{-1}(\mathbf{Y} - \mathbf{Z}_0\hat{\mathbf{d}})$$

where $\hat{\mathbf{d}}$ is the Generalized Least Squares estimate of $\mathbf{d}$ and $\lambda = \frac{\sigma^2}{\rho}$ is the nugget-to-sill ratio.

##REML

Variogram fitting is often not a reliable way of finding the parameters of our spatial model. Statisticians usually take two routes for parameter estimation: generalized cross validation (GCV), or maximum likelihood estimation (MLE).

Under the `fields` spatial model $Y(\mathbf{x}) = \mathbf{Z}(\mathbf{x})\mathbf{d} + g(\mathbf{x}) + \epsilon(\mathbf{x})$, we have Gaussian observations $\mathbf{Y} \sim N(\mathbf{Zd}, \rho\mathbf{K} + \sigma^2\mathbf{I})$. Thus, calculating the log-likelihood $\ell(\mathbf{d}, \boldsymbol{\xi})$ is easy. Here, we denote the variance of $\mathbf{Y}$ as $\Sigma := \rho\mathbf{K} + \sigma^2\mathbf{I}$ and use $\boldsymbol{\xi}$ as a vector of spatial parameters (e.g. $\boldsymbol{\xi} = (\rho, \sigma^2, a, \nu)$).

$$\ell(\mathbf{d}, \boldsymbol{\xi}) \propto -\frac{1}{2}\left(\log(\det(\Sigma)) + (\mathbf{Y} - \mathbf{Zd})^T\Sigma^{-1}(\mathbf{Y} - \mathbf{Zd})\right)$$

For any fixed $\boldsymbol{\xi}$, the value of $\mathbf{d}$ that maximizes ℓ is the Generalized Least Squares (GLS) estimate $\hat{\mathbf{d}} = (\mathbf{Z}^T\Sigma^{-1}\mathbf{Z})^{-1}\mathbf{Z}^T\Sigma^{-1}\mathbf{Y}$.

To maximize ℓ, one often "profiles" and eliminates $\mathbf{d}$ in the equation for ℓ by substituting $\mathbf{d} = \hat{\mathbf{d}}$. Then $\ell(\mathbf{d}, \boldsymbol{\xi}) = \ell(\boldsymbol{\xi})$ only depends on $\boldsymbol{\xi}$. Numerical optimization (nonlinear methods like BFGS) is used to find the remaining parameters, and then the GLS estimate $\hat{\mathbf{d}}$ is updated with the optimal parameter values. (See Handbook of Spatial Statistics, p. 46).

However, this process of Maximum Likelihood Estimation (MLE) tends to give biased results, as

$$(\mathbf{Y} - \mathbf{Z}\hat{\mathbf{d}})^T\Sigma^{-1}(\mathbf{Y} - \mathbf{Z}\hat{\mathbf{d}}) \leq (\mathbf{Y} - \mathbf{Zd})\Sigma^{-1}(\mathbf{Y} - \mathbf{Zd}), \quad \forall\mathbf{d}.$$

Intuitively, this means that $\mathbf{Z}\hat{\mathbf{d}}$ is "closer" to $\mathbf{Y}$ than any other estimate $\mathbf{Zd}$, even if $\mathbf{d}$ is the true parameter! As a result, the MLE for $\boldsymbol{\xi}$ is biased due to this simultaneous estimation of $\mathbf{d}$. In particular, the MLE for $\boldsymbol{\xi}$ tends to underestimate the process variance.

This leads to Restricted Maximum Likelihood (REML), which attempts to reduce this bias when using maximum likelihood methods. We use a linear transformation $\mathbf{Y}^* = \mathbf{AY}$ of the data so that the distribution of $\mathbf{Y}^*$ does not depend on $\mathbf{d}$. In particular, we use the matrix $\mathbf{A} = \mathbf{I} - \mathbf{Z}(\mathbf{Z}^T\mathbf{Z})^{-1}\mathbf{Z}^T$ that projects $\mathbf{Y}$ to ordinary least squares residuals. The resulting matrix product $\mathbf{AY}$ is independent of $\mathbf{d}$.

$$\ell_R(\boldsymbol{\xi}) = -\frac{1}{2}\left(\log(\det(\Sigma)) + \frac{1}{2}\log(\det(\mathbf{Z}^T\Sigma^{-1}\mathbf{Z})) + \mathbf{Y}^T(\Sigma^{-1} - \Sigma^{-1}\mathbf{Z}(\mathbf{Z}^T\Sigma^{-1}\mathbf{Z})^{-1}\mathbf{Z}^T\Sigma^{-1})\mathbf{Y}\right)$$

This ends up being the profile log-likelihood with an additional term involving $\log(\det(\mathbf{Z}^T\Sigma^{-1}\mathbf{Z}))$. Again, this expression must be maximized numerically with respect to $\boldsymbol{\xi}$. The procedure concludes by estimating $\mathbf{d}$ with GLS and the estimate $\hat{\boldsymbol{\xi}}$ plugged in.

##GCV

The GCV function is

$$V(\lambda) = \frac{\frac{1}{n}\|(\mathbf{I} - \mathbf{A}(\lambda))\mathbf{Y}\|^2}{\left(\frac{1}{n} \cdot \operatorname{tr}(\mathbf{I} - \mathbf{A}(\lambda)\right)^2}$$

where $\mathbf{A}(\lambda) = \Sigma_0(\Sigma + \lambda I)^{-1}$ (with $\lambda = \frac{\sigma^2}{\rho}$) and the GCV estimate for λ is $\hat{\lambda} = \operatorname{argmin} V(\lambda)$. GCV is a predictive MSE criteria.