Lecture 10

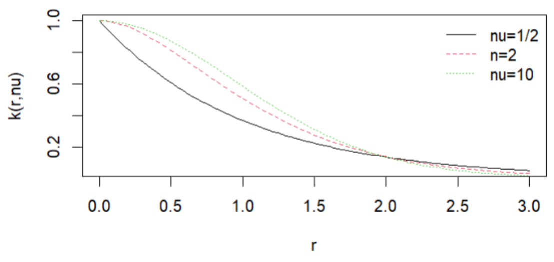
Gaussian Process Regression, Part 2

One of the elements that varies in a Gaussian process is the kernel. There are a number of common ones in use. We're going to look at a couple of alternatives more closely. The first one we'll look at is a modified Bessel function.

```
matern <- function(r, nu, theta)
 {
  rat <- r*sqrt(2*nu/theta)
  C <- (2^(1 - nu))/gamma(nu) * rat^nu * besselK(rat, nu)
  C[is.nan(C)] <- 1
  return(C)
 }
```

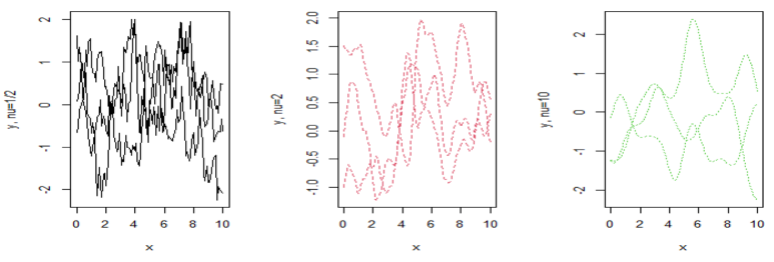
A parameter for the function we can tweak is ν (nu).

```
r <- seq(eps, 3, length=100)
plot(r, matern(r, nu=1/2, theta=1), type="l", ylab="k(r,nu)")
lines(r, matern(r, nu=2, theta=1), lty=2, col=2)
lines(r, matern(r, nu=10, theta=1), lty=3, col=3)
legend("topright", c("nu=1/2", "n=2", "nu=10"), lty=1:3, col=1:3, bty="n")
```



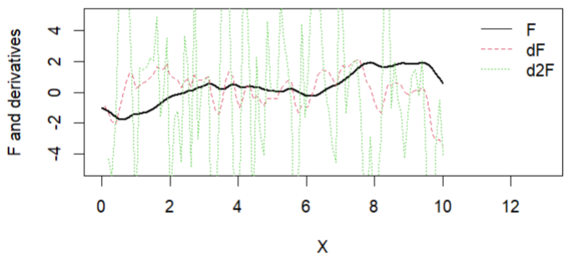
One advantage of this kernel is that for small values of ν, we don't need to add epsilon to make things work.

```
X <- seq(0, 10, length=100)
R <- sqrt(distance(X))
K0.5 <- matern(R, nu=1/2, theta=1)
K2 <- matern(R, nu=2, theta=1)
K10 <- matern(R, nu=10, theta=1) + diag(eps, 100)

par(mfrow=c(1,3))
matplot(X, t(rmvnorm(3,sigma=K0.5)), type="l", col=1, lty=1,
    xlab="x", ylab="y, nu=1/2")
matplot(X, t(rmvnorm(3,sigma=K2)), type="l", col=2, lty=2, xlab="x", ylab="y, nu=2")
matplot(X, t(rmvnorm(3,sigma=K10)), type="l", col=3, lty=3, xlab="x", ylab="y, nu=10")
par(mfrow=c(1,1))
```



Let’s consider the derivatives of these functions to examine their smoothness.

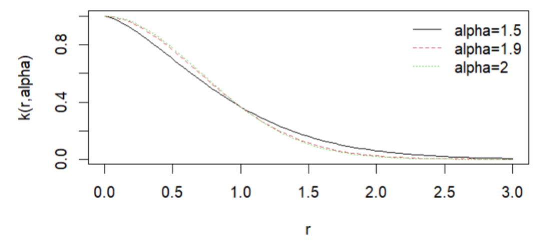
```
F <- rmvnorm(1, sigma=K2)
dF <- (F[-1] - F[-length(F)])/(X[2] - X[1])
d2F <- (dF[-1] - dF[-length(dF)])/(X[2] - X[1])

plot(X, F, type="l", lwd=2, xlim=c(0,13), ylim=c(-5,5),
  ylab="F and derivatives")
lines(X[-1], dF, col=2, lty=2)
lines(X[-(1:2)], d2F, col=3, lty=3)
legend("topright", c("F", "dF", "d2F"), lty=1:3, col=1:3, bty="n")
```



The more jagged the derivative (or second derivative), the less smooth the function is.

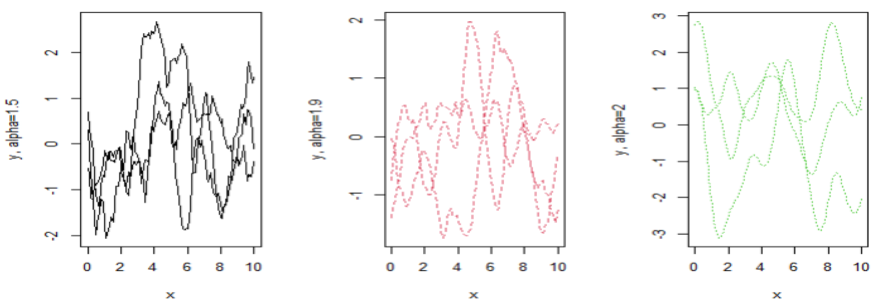
Another type of kernel is the power exponential family.

```
powerexp <- function(r, alpha, theta)
 {
  C <- exp(-(r/sqrt(theta))^alpha)
  C[is.nan(C)] <- 1
  return(C)
 }

plot(r, powerexp(r, alpha=1.5, theta=1), type="l", ylab="k(r,alpha)")
lines(r, powerexp(r, alpha=1.9, theta=1), lty=2, col=2)
lines(r, powerexp(r, alpha=2, theta=1), lty=3, col=3)
legend("topright", c("alpha=1.5", "alpha=1.9", "alpha=2"), lty=1:3, col=1:3, bty="n")
```



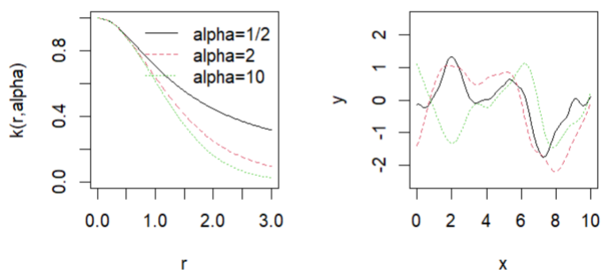
Despite the apparent similarities, the sample paths produce distinct results. Note that a power of 2 here is equivalent to the Euclidean distance.

```
Ka1.5 <- powerexp(R, alpha=1.5, theta=1)
Ka1.9 <- powerexp(R, alpha=1.9, theta=1)
Ka2 <- powerexp(R, alpha=2, theta=1) + diag(eps, nrow(R))
par(mfrow=c(1,3))
ylab <- paste0("y, alpha=", c(1.5, 1.9, 2))
matplot(X, t(rmvnorm(3, sigma=Ka1.5)), type="l", col=1, lty=1, xlab="x", ylab=ylab[1])
matplot(X, t(rmvnorm(3, sigma=Ka1.9)), type="l", col=2, lty=2, xlab="x", ylab=ylab[2])
matplot(X, t(rmvnorm(3, sigma=Ka2)), type="l", col=3, lty=3, xlab="x", ylab=ylab[3])
```



The rational quadratic kernel is a mixture of Gaussian power ($\alpha = 2$) and exponential power (with $\alpha = 1$).

```
ratquad <- function(r, alpha, theta)
 {
  C <- (1 + r^2/(2 * alpha * theta))^(-alpha)
  C[is.nan(C)] <- 1
  return(C)
 }

par(mfrow=c(1,2))
plot(r, ratquad(r, alpha=1/2, theta=1), type="l", ylab="k(r,alpha)", ylim=c(0,1))
lines(r, ratquad(r, alpha=2, theta=1), lty=2, col=2)
lines(r, ratquad(r, alpha=10, theta=1), lty=3, col=3)
legend("topright", c("alpha=1/2", "alpha=2", "alpha=10"), lty=1:3, col=1:3, bty="n")
Eps <- diag(eps, nrow(R))
plot(X, rmvnorm(1, sigma=ratquad(R, alpha=1/2, theta=1) + Eps), type="l",
  col=1, lty=1, xlab="x", ylab="y", ylim=c(-2.5,2.5))
lines(X, rmvnorm(1, sigma=ratquad(R, alpha=2, theta=1) + Eps), col=2, lty=2)
lines(X, rmvnorm(1, sigma=ratquad(R, alpha=10, theta=1) + Eps), type="l", col=3, lty=3)
```

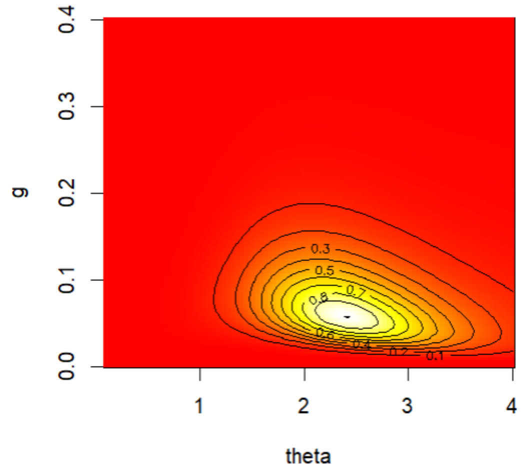


These examples are not meant to be exhaustive and it's worth considering what kernels are available in the packages you choose, and experiment with them to see which produces the best results for a given dataset.

The potential for GP customization is powerful. GP regression can either be out-of-the-box with simple covariance structures implemented by library subroutines, ready for anything, or can be tailored to the bespoke needs of a particular modeling enterprise and data type. You can get very fancy, or you can simplify, and inference for unknowns need not be too onerous if you stick to likelihood-based criteria paired with mature libraries for optimization with closed-form derivatives.

As with any statistical model, you have to be careful not to get too fancy or it may come back to bite you. The more hyperparameters purporting to offer greater flexibility or a better-tuned fit, the greater the estimation risk. By estimation risk I mean both potential to fit noise as signal, as well as its more conventional meaning (particularly popular in empirical finance) which fosters incorporation of uncertainties, inherent in high variance sampling distributions for optimized parameters, that are often overlooked. Nonparametric models and latent function spaces exacerbate the situation. Awareness of potential sources of such risk is particularly fraught, and assessing its extent even more so. At some point there might be so many knobs that its hard to argue that the "hyperparameter" moniker is apt compared to the more canonical "parameter", giving the practitioner the sense that choosing appropriate settings is key to getting good fits.

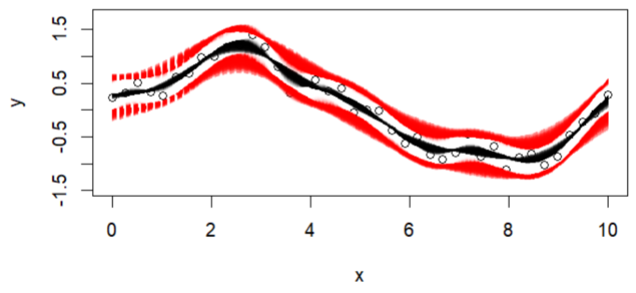
```
x <- seq(0, 10, length=40)
ytrue <- (sin(pi*x/5) + 0.2*cos(4*pi*x/5))
y <- ytrue + rnorm(length(ytrue), sd=0.2)

g <- seq(0.001, 0.4, length=100)
theta <- seq(0.1, 4, length=100)
grid <- expand.grid(theta, g)

library(laGP)
ll <- rep(NA, nrow(grid))
xx <- seq(0, 10, length=100)
pm <- matrix(NA, nrow=nrow(grid), ncol=length(xx))
psd <- matrix(NA, nrow=nrow(grid), ncol=length(xx))
for(i in 1:nrow(grid)) {
  gpi <- newGP(matrix(x, ncol=1), y, d=grid[i,1], g=grid[i,2])
  p <- predGP(gpi, matrix(xx, ncol=1), lite=TRUE)
  pm[i,] <- p$mean
  psd[i,] <- sqrt(p$s2)
  ll[i] <- llikGP(gpi)
  deleteGP(gpi)
}
l <- exp(ll - max(ll))

image(theta, g, matrix(l, ncol=length(theta)), col=cols)
contour(theta, g, matrix(l, ncol=length(g)), add=TRUE)
```



That skewness is hiding a multimodal posterior distribution over functions. The modes are “higher signal/lower noise” and “lower signal/higher noise”. Such signal–noise tension is an ordinary affair, and settling for one MLE tuple in a landscape of high values – even if you’re selecting the very highest ones – can grossly underestimate uncertainty. The best view of signal-to-noise tension is through the predictive surface, in particular what that surface would look like for a multitude of most likely hyperparameter settings.

```
q1 <- pm + qnorm(0.95, sd=psd)
q2 <- pm + qnorm(0.05, sd=psd)

plot(x,y, ylim=c(range(q1, q2)))
matlines(xx, t(pm), col=rgb(0,0,0,alpha=(l/max(l))/2), lty=1)
matlines(xx, t(q1), col=rgb(1,0,0,alpha=(l/max(l))/2), lty=2)
matlines(xx, t(q2), col=rgb(1,0,0,alpha=(l/max(l))/2), lty=2)
```

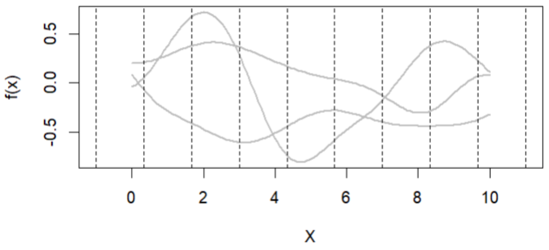
The more opaque regions mean more agreement across parameter settings. The black lines are the mean curve and the red ones the boundaries of the 95% confidence error.



Fully Bayesian GP regression, despite many UQ virtues extolled above, can all but be ruled out on computational grounds when n is even modestly large ($n > 2000$ or so), speedups coming with fancy matrix libraries notwithstanding. If it takes dozens or hundreds of likelihood evaluations to maximize a likelihood, it will take several orders of magnitude more to sample from a posterior by MCMC. Even in cases where MCMC is just doable, it's sometimes not clear that posterior inference is the right way to spend valuable computing resources.

We are going to look briefly at GPs by convolution, which is one possible help to dealing with the computational problems noted above. The basic idea is to fix certain points, which helps to break up the problem into more easily computable pieces. In some ways, the idea is similar to the idea of knots in splines.

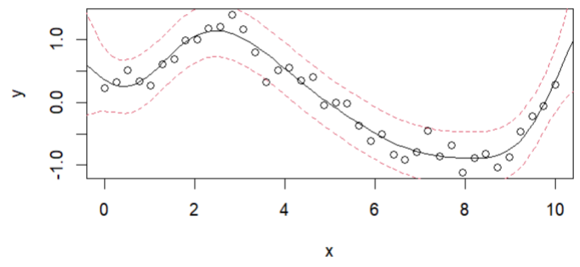
```
ell <- 10
n <- 100
X <- seq(0, 10, length=n)
omega <- seq(-1, 11, length=ell)
K <- matrix(NA, ncol=ell, nrow=n)
for(j in 1:ell) K[,j] <- dnorm(X, omega[j])

beta <- matrix(rnorm(3*ell), ncol=3)

F <- K %*% beta

matplot(X, F, type="l", lwd=2, lty=1, col="gray",
  xlim=c(-1,11), ylab="f(x)")
abline(v=omega, lty=2, lwd=0.5)
```



The vertical lines are the fixed points.

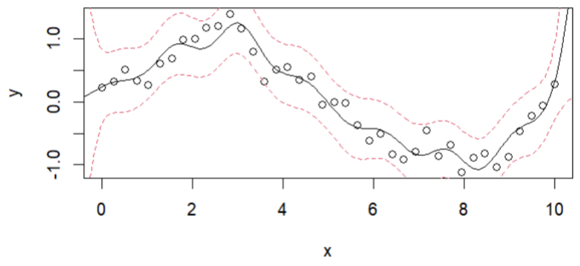
```
n <- length(x)
K <- as.data.frame(matrix(NA, ncol=ell, nrow=n))
for(j in 1:ell) K[,j] <- dnorm(x, omega[j])
names(K) <- paste0("omega", 1:ell)

fit <- lm(y ~ . -1, data=K)

xx <- seq(-1, 11, length=100)
KK <- as.data.frame(matrix(NA, ncol=ell, nrow=length(xx)))
for(j in 1:ell) KK[,j] <- dnorm(xx, omega[j])
names(KK) <- paste0("omega", 1:ell)
```

```
p <- predict(fit, newdata=KK, interval="prediction")

plot(x, y)
lines(xx, p[,1])
lines(xx, p[,2], col=2, lty=2)
lines(xx, p[,3], col=2, lty=2)
```



Here, we've created a regression model (with a zero mean GP) using the convolution approach. We can adjust the kernel and length scale for slightly different results.

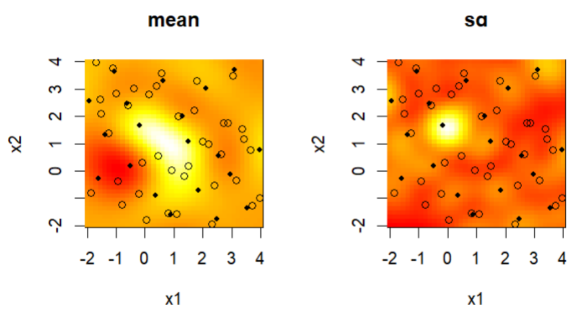
```
for(j in 1:ell) K[,j] <- dnorm(x, omega[j], sd=0.5)
fit <- lm(y ~ . -1, data=K)
for(j in 1:ell) KK[,j] <- dnorm(xx, omega[j], sd=0.5)
p <- predict(fit, newdata=KK, interval="prediction")
plot(x, y)
lines(xx, p[,1])
lines(xx, p[,2], col=2, lty=2)
lines(xx, p[,3], col=2, lty=2)
```



The kernel, the decay length and the spacing of the fixed points interact with each other.

```
ell <- 20
omega <- maximinLHS(ell, 2)
omega[,1] <- (omega[,1] - 0.5)*6 + 1
omega[,2] <- (omega[,2] - 0.5)*6 + 1

n <- nrow(X2)
K <- as.data.frame(matrix(NA, ncol=ell, nrow=n))
for(j in 1:ell)
  K[,j] <- dnorm(X2[,1], omega[j,1])*dnorm(X2[,2], omega[j,2])
names(K) <- paste0("omega", 1:ell)
```

```
fit <- lm(y2 ~ . -1, data=K)

xx <- seq(-2, 4, length=gn)
XX <- expand.grid(xx, xx)
KK <- as.data.frame(matrix(NA, ncol=ell, nrow=nrow(XX)))
for(j in 1:ell)
  KK[,j] <- dnorm(XX[,1], omega[j,1])*dnorm(XX[,2], omega[j,2])
names(KK) <- paste0("omega", 1:ell)

p <- predict(fit, newdata=KK, se.fit=TRUE)

par(mfrow=c(1,2))
image(xx, xx, matrix(p$fit, ncol=gn), col=cols, main="mean", xlab="x1", ylab="x2")
points(X2)
points(omega, pch=20)
image(xx, xx, matrix(p$se.fit, ncol=gn), col=cols, main="sd", xlab="x1", ylab="x2")
points(X2)
points(omega, pch=20)
```

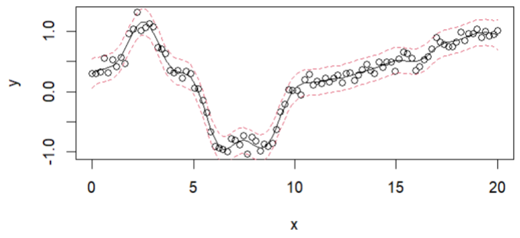


The signal isn't as clear in either plot. Several explanations suggest themselves upon reflection. One is differing implicit lengthscale, in particular the one used immediately above is not fit from data. Another has to do with locations of the ω_j, and their multitude: $\ell = 20$. Notice how both mean and sd surfaces exhibit "artifacts" near some of the ω_j.

For a spatial process to be stationary, then covariance is measured the same everywhere. This isn't always the case, but there are ways for Gaussian processes to handle this.

```
X <- seq(0,20,length=100)
y <- (sin(pi*X/5) + 0.2*cos(4*pi*X/5)) * (X <= 9.6)
lin <- X>9.6
y[lin] <- -1 + X[lin]/10
y <- y + rnorm(length(y), sd=0.1)

gpi <- newGP(matrix(X, ncol=1), y, d=0.1, g=0.1*var(y), dK=TRUE)
mle <- jmleGP(gpi)

p <- predGP(gpi, matrix(X, ncol=1), lite=TRUE)
deleteGP(gpi)
```

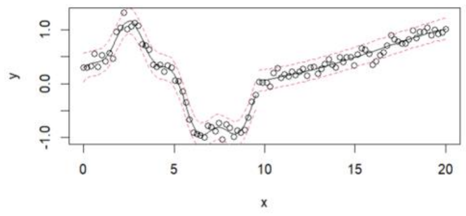
```
plot(X, y, xlab="x")
lines(X, p$mean)
lines(X, p$mean + 2*sqrt(p$s2), col=2, lty=2)
lines(X, p$mean - 2*sqrt(p$s2), col=2, lty=2)
```



Observe how the predictive equations struggle to match disparate behavior in the two regimes. Since only one lengthscale must accommodate the entire input domain, the likelihood is faced with a choice between regimes and in this case it clearly favors the left-hand one. A wiggly fit to the right-hand regime is far better than a straight fit to left. As a result, wiggliness bleeds from the left to right.

One way to address this is to separate the two domains.

```
left <- X < 9.6
gpl <- newGP(matrix(X[left], ncol=1), y[left], d=0.1, g=0.1*var(y), dK=TRUE)
mlel <- jmleGP(gpl)
gpr <- newGP(matrix(X[!left], ncol=1), y[!left], d=0.1, g=0.1*var(y), dK=TRUE)
mler <- jmleGP(gpr, drange=c(eps, 100))

pl <- predGP(gpl, matrix(X[left], ncol=1), lite=TRUE)
deleteGP(gpl)
pr <- predGP(gpr, matrix(X[!left], ncol=1), lite=TRUE)
deleteGP(gpr)

plot(X, y, xlab="x")
lines(X[left], pl$mean)
lines(X[left], pl$mean + 2*sqrt(pl$s2), col=2, lty=2)
lines(X[left], pl$mean - 2*sqrt(pl$s2), col=2, lty=2)
lines(X[!left], pr$mean)
lines(X[!left], pr$mean + 2*sqrt(pr$s2), col=2, lty=2)
lines(X[!left], pr$mean - 2*sqrt(pr$s2), col=2, lty=2)
```



The downside is that the way we've coded this, the sections don't join, but some changes could correct that, such as including the boundary point on both sides of the break.

Let's look at an overview of some packages that implement Gaussian process regression (this information will be useful for the package comparison assignment for this material).

gplm Package

- *Kernels*: Supports standard kernels such as Radial Basis Function (RBF), linear, polynomial, and custom kernels.
- *Hyperparameters*: The package allows the optimization of kernel parameters such as the length scale, variance, and noise level.
- *Spatial Data*: While it can be used for spatial data, it is generally used for non-spatial data.

```
# Install and load the gplm package
install.packages("gplm")
library(gplm)

# Fit a Gaussian Process model to predict 'mpg' using 'hp' and 'wt'
data <- mtcars[, c("mpg", "hp", "wt")]
fit <- gplm(y = data$mpg, X = data[, c("hp", "wt")], kern.type = "rbf")

# Summary of the fit
summary(fit)

# Predictions
newdata <- data.frame(hp = c(110, 150), wt = c(2.5, 3.5))
predict(fit, Xnew = newdata)
```

DiceKriging Package

- *Kernels*: Primarily focused on kriging with spatial data, it supports Matern, Gaussian (RBF), and cubic splines.
- *Hyperparameters*: The package allows the estimation and optimization of nugget effects, range parameters, and smoothness parameters.
- *Spatial Data*: This package is explicitly designed for spatial data modeling and kriging applications.

```
# Install and load the DiceKriging package
install.packages("DiceKriging")
library(DiceKriging)

# Generate a simple spatial grid
set.seed(123)
x <- seq(-5, 5, length = 20)
y <- seq(-5, 5, length = 20)
grid <- expand.grid(x = x, y = y)

# Create a response variable with a spatial pattern
z <- with(grid, sin(sqrt(x^2 + y^2)))

# Convert to a data frame
```

```
spatial_data <- data.frame(grid, z)

# Fit the Gaussian Process model using DiceKriging
gp_model <- km(
  formula = z ~ 1,                # Response variable
  design = spatial_data[, c("x", "y")],  # Predictor variables (spatial coordinates)
  response = spatial_data$z,          # Response variable
  covtype = "gauss",                  # Gaussian covariance function (kernel)
  nugget = 1e-5,                      # Small nugget effect for numerical stability
  control = list(trace = TRUE)
)

summary(gp_model)

# Generate a grid for predictions
pred_grid <- expand.grid(x = seq(-5, 5, length = 40), y = seq(-5, 5, length = 40))

# Predict using the fitted Gaussian Process model
# Type "UK" is used for Universal Kriging (most common choice)
predictions <- predict(gp_model, newdata = pred_grid, type = "UK")

# Add predictions to the grid for easy plotting
pred_grid$predicted_z <- predictions$mean
```

Additional Considerations

- *Hyperparameters*: DiceKriging automatically estimates hyperparameters (e.g., length scale, variance) using MLE. However, you can customize this process if needed.
- *Spatial Data*: While this example uses simple synthetic spatial data, you can apply similar techniques to more complex datasets. Just ensure your design matrix (predictors) and response are correctly specified.
- *Non-Spatial Data*: DiceKriging is versatile and can be used for non-spatial data as well. The core principles remain the same.

kernlab Package

- *Kernels*: Provides a wide range of kernels, including linear, polynomial, RBF, Matern, and custom-defined kernels.
- *Hyperparameters*: Supports tuning kernel parameters such as sigma for RBF, degree for polynomial kernels, and regularization parameters.
- *Spatial Data*: Typically used for non-spatial data but can be extended for spatial applications.

```
# Install and load the kernlab package
install.packages("kernlab")
library(kernlab)

# Fit a Gaussian Process model to predict 'mpg' using 'hp' and 'wt'
data <- as.matrix(mtcars[, c("hp", "wt")])
gp_fit <- gausspr(x = data, y = mtcars$mpg, kernel = "rbfdot", kpar = list(sigma = 0.1))
```

```
# Summary of the fit
print(gp_fit)

# Predictions
newdata <- as.matrix(data.frame(hp = c(110, 150), wt = c(2.5, 3.5)))
predict(gp_fit, newdata)
```

gstat Package

- *Kernels*: Implements various spatial covariance models, including spherical, exponential, and Gaussian models.
- *Hyperparameters*: Supports estimation and optimization of range, sill, and nugget parameters.
- *Spatial Data*: Primarily used for geostatistics and kriging with spatial data

spBayes Package

- *Kernels*: Implements spatial covariance functions like exponential and Gaussian.
- *Hyperparameters*: Supports Bayesian inference for hyperparameters using MCMC methods.
- *Spatial Data*: Tailored for spatial data with a focus on Bayesian spatial models.

laGP (Local Approximate Gaussian Processes)

- *Purpose*: laGP is designed to scale Gaussian Process models to large datasets by using local approximations rather than a full GP over the entire dataset.
- *Kernels*: It primarily supports squared exponential (Gaussian/RBF) kernels but can be extended with other covariance structures.
- *Hyperparameters*: Allows optimization of length scale and variance parameters, similar to other GP models.
- *Spatial Data*: It is not specifically designed for spatial data, but can be adapted for such applications.

```
# Load necessary libraries
install.packages("laGP")
library(laGP)

# Example dataset (using mtcars with mpg as the response variable and a few predictors)
X <- as.matrix(mtcars[, c("disp", "hp", "wt", "qsec")])  # Predictor variables
y <- mtcars$mpg  # Response variable

# Set hyperparameters for the GP
d <- darg(list(mle = TRUE), X)  # Automatically estimate length-scale (d)
g <- garg(list(mle = TRUE), y)  # Automatically estimate nugget (g)

# Set the number of initial points (start) for the laGP method
start <- 10  # You can choose a reasonable value based on your dataset size
end <- 20    # The total number of points to include in the final model

# Run the laGP function
result <- laGP(Xref = X, X = X, Z = y, d = d$start, g = g$start, method = "alc", start = start, end = end)

# Predictions and other results can be extracted from the result object
predictions <- result$mean
```

```
# Print predictions
print(predictions)
```

RobustGaSP (Robust Gaussian Stochastic Process)

- *Purpose*: RobustGaSP is designed to handle robust Gaussian Process regression, particularly when dealing with outliers or when the model needs to be robust against deviations from assumptions.
- *Kernels*: Supports common kernels like Gaussian, Matérn, and power exponential, with a focus on robustness.
- *Hyperparameters*: The package includes options for tuning length scale, smoothness, and nugget effect, and can handle multiple hyperparameter optimization approaches.
- *Spatial Data*: RobustGaSP is not specifically aimed at spatial data but can be applied to spatial applications with appropriate settings.

```
# Install and load the RobustGaSP package
install.packages("RobustGaSP")
library(RobustGaSP)

# Select variables from mtcars
X <- as.matrix(mtcars[, c("hp", "wt")])
y <- mtcars$mpg

# Fit a Robust Gaussian Process model
rgp_model <- rgasp(design = X, response = y, kernel_type = "matern_5_2")

# Summary of the model
summary(rgp_model)

# Predictions for new data points
new_X <- as.matrix(data.frame(hp = c(110, 150), wt = c(2.5, 3.5)))
predictions <- predict(rgp_model, testing_input = new_X)

print(predictions$mean)  # Predicted means
```

deepgp package

- *Choosing Layers*: In deepGP, you can fit deeper models (more than 2 layers) if your data and problem require it, but this increases the computational cost.
- *Complexity*: Deep GPs are more complex to train and interpret than standard GPs, but they offer greater flexibility.
- *Applicability*: The package is powerful, but it might be overkill for simple datasets. Use it for problems where the relationships between variables are complex and non-linear.

If you perform a search on CRAN, you'll find many more packages. Their documentation should provide additional examples.

Resources:

1. https://bookdown.org/rbg/surrogates/chap5.html