Final Project: Gaussian Processes

STP 540

Antonio Campbell, Yumeng Cao, Andrew Herren

April 30, 2021

Created using R Markdown.

Before proceeding to solutions, we first load the necessary R packages and set a random seed.

```r
pkg.list <- c("MASS", "knitr", "randomForest", "lhs", "GPfit")
lapply(pkg.list, require, character.only = TRUE)
knitr::opts_chunk$set(echo = TRUE, fig.align = 'center')
set.seed(1234)
```

1. GP Log-Likelihood

Any Gaussian process modeling endeavor begins in earnest by defining the covariance structure of observations. In this project, we use a squared exponential kernel.

```r
# Define a squared distance matrix (note, there is likely a more efficient
# way to do this, consider refactoring)
sq_dist_mat <- function(x, x_new = NULL){
    if (is.null(x_new)){
        x_new <- x
    }
    X.mat <- t(matrix(rep(x, length(x_new)), ncol = length(x), byrow = T))
    X.new.mat <- t(matrix(rep(x_new, each = length(x)), ncol = length(x), byrow = T))
    return(((X.mat - X.new.mat)^2))
}
# Define the noiseless component of the kernel, using a squared
# exponential similarity function
se_kernel <- function(x, x_new = NULL, sigma.f = 1, ell = 1){
    dist_mat <- sq_dist_mat(x, x_new)
    return((sigma.f^2)*exp(-(dist_mat)/(2*(ell^2))))
}
# Define the "noisy" part of the GP kernel, which adds uncertainty
# to observed samples
gp_kernel_noise <- function(x, sigma.y){
    return((sigma.y^2)*diag(length(x)))
}
```

To estimate the log-likelihood for a Gaussian process, we implement lines 1, 2, and 6 of Algorithm 15.1 from

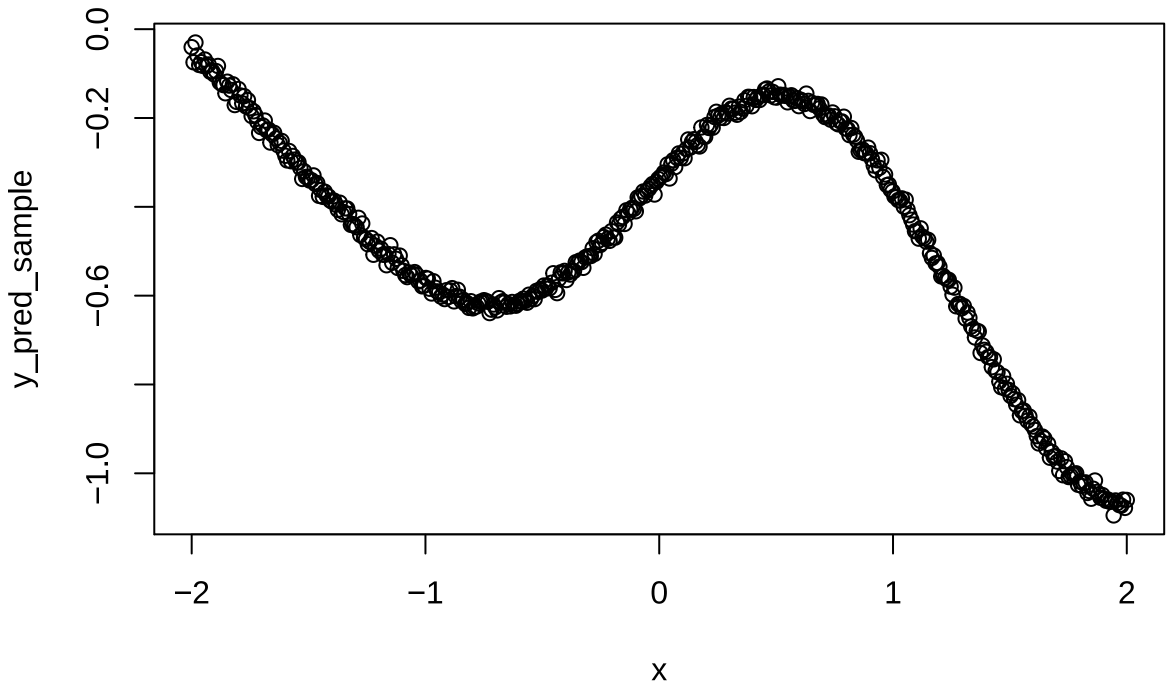
Murphy [2012]:

$$\begin{aligned}
L &= \text{cholesky}\left(K_y\right) \\
\alpha &= L' \setminus \left(L \setminus y\right) \\
\mathbb{E}(f^*) &= k_*'\alpha \\
v &= L \setminus k_* \\
\mathbb{V}(f^*) &= k_{*,*} - v'v \\
\log p(y \mid X) &= -\frac{1}{2}y'\alpha - \sum_i logL_{ii} - \frac{N}{2}log(2\pi)
\end{aligned}$$

```r
gploglik <- function(x, y, sigma.f, ell, sigma.y){
    n <- length(x)
    K <- se_kernel(x, x_new = NULL, sigma.f = sigma.f, ell = ell)
    L <- t(chol(K + gp_kernel_noise(x, sigma.y)))
    alpha <- qr.solve(t(L), qr.solve(L, y))
    loglik <- -0.5*as.numeric(t(y)%*%alpha) - sum(diag(log(L))) - (n/2)*log(2*pi)
    return(loglik)
}
```

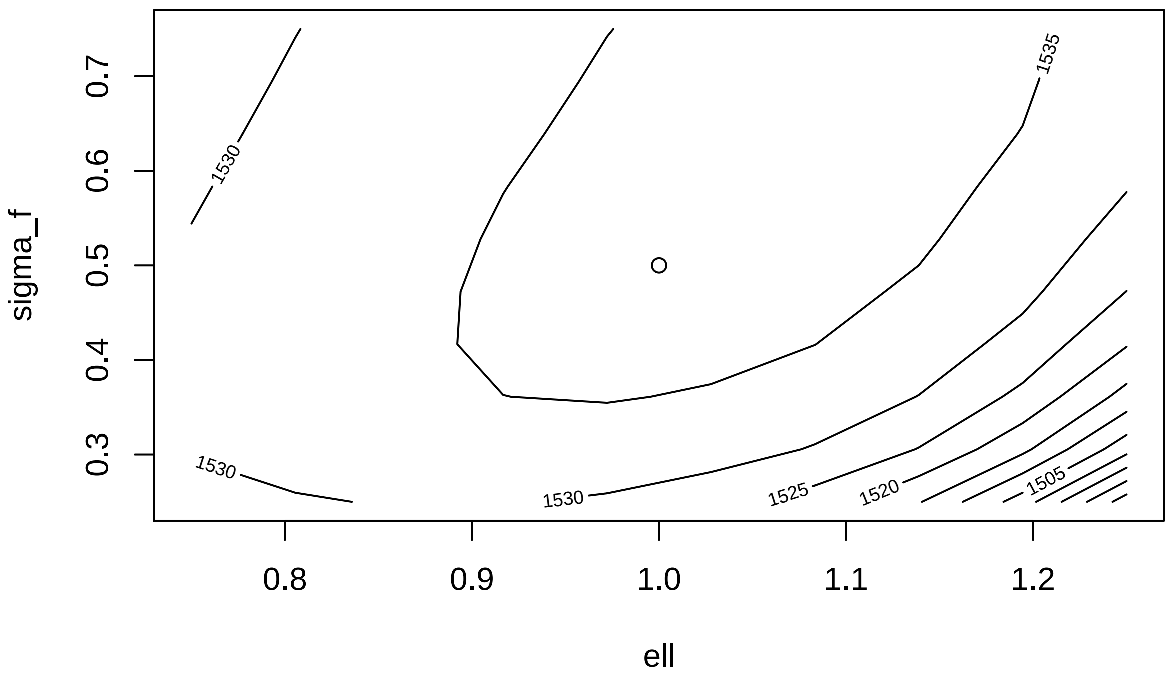
We can verify this likelihood as follows. Since the marginal distribution of $Y \mid X$ is $\mathcal{N}\left(\mathbf{0}, K + \sigma_y^2 I\right)$ with $K = \kappa\left(X, X, \sigma_f^2, \ell, \sigma_y^2\right)$, we fix values of $\sigma_f^2 = 0.25$, $\ell = 1$, and $\sigma_y^2 = 0.0001$ and draw samples of $Y \mid X$ for a fixed grid of X values.

```r
# Generate some noisy data
n <- 500
x <- seq(-2, 2, length.out = n)

# Generate y | x ~ N(0, K + sigma^2_y I)
sigma.f.true <- 0.5
ell.true <- 1
sigma.y.true <- 0.01
mu_gp <- rep(0, length(x))
sigma_gp <- (
    se_kernel(x, x_new = NULL, sigma.f = sigma.f.true,
              ell = ell.true) +
    gp_kernel_noise(x, sigma.y = sigma.y.true)
)
y_pred_sample <- mvrnorm(n = 1, mu = mu_gp, Sigma = sigma_gp)

# Plot the data
plot(x, y_pred_sample)
```



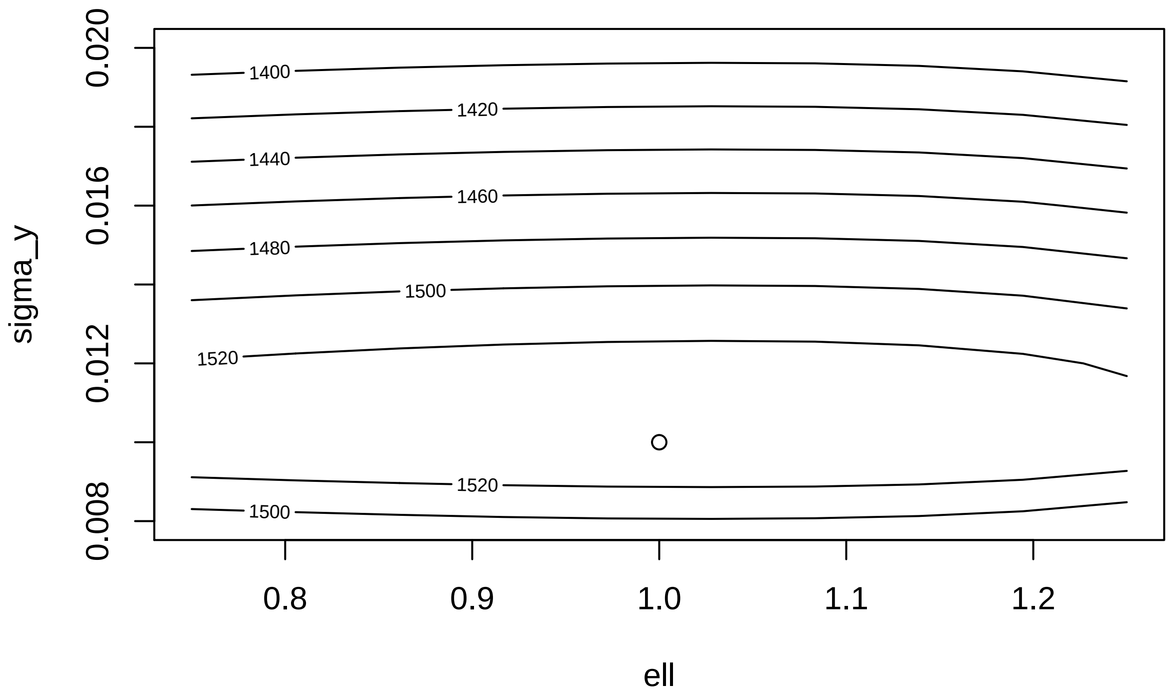
We can verify our likelihood implementation several ways. First, we fix $\sigma_y = 0.01$ and construct a contour plot on a grid of σ_f and ℓ values to see if the contours appear roughly maximized at $\sigma_f = 0.5$ and $\ell = 1$.

```
n.grid <- 10
sigma.f.seq <- seq(0.25, 0.75, length.out = n.grid)
ell.seq <- seq(0.75, 1.25, length.out = n.grid)
param.grid <- expand.grid(sigma.f.seq, ell.seq)
logliks <- apply(param.grid, 1, function(z)
    gploglik(x, y_pred_sample, sigma.f = z[1],
             ell = z[2], sigma.y = sigma.y.true)
)
contour(
    ell.seq, sigma.f.seq, xlab = "ell", ylab = "sigma_f",
    matrix(logliks, nrow = n.grid, byrow = T)
)
points(ell.true, sigma.f.true)
```



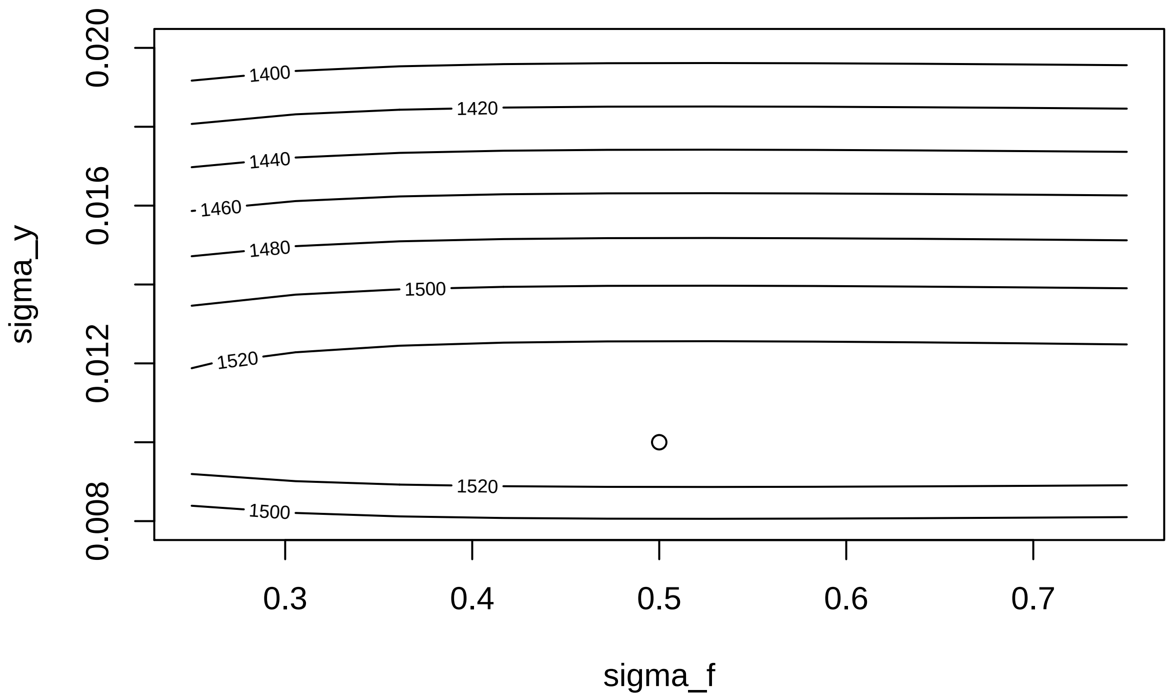
Now, we fix $\sigma_f = 0.5$ and run a contour plot of ℓ and σ_y.

```
sigma.y.seq <- seq(0.008, 0.02, length.out = n.grid)
ell.seq <- seq(0.75, 1.25, length.out = n.grid)
param.grid <- expand.grid(sigma.y.seq, ell.seq)
logliks <- apply(param.grid, 1, function(z)
    gploglik(x, y_pred_sample, sigma.f = sigma.f.true,
             ell = z[2], sigma.y = z[1])
)
contour(
    ell.seq, sigma.y.seq, xlab = "ell", ylab = "sigma_y",
    matrix(logliks, nrow = n.grid, byrow = T)
)
points(ell.true, sigma.y.true)
```



Finally, we fix $\ell = 1$ and run a contour plot of σ_f and σ_y.

```
sigma.y.seq <- seq(0.008, 0.02, length.out = n.grid)
sigma.f.seq <- seq(0.25, 0.75, length.out = n.grid)
param.grid <- expand.grid(sigma.y.seq, sigma.f.seq)
logliks <- apply(param.grid, 1, function(z)
    gploglik(x, y_pred_sample, sigma.f = z[2],
             ell = ell.true, sigma.y = z[1])
)
contour(
    sigma.f.seq, sigma.y.seq, xlab = "sigma_f", ylab = "sigma_y",
    matrix(logliks, nrow = n.grid, byrow = T)
)
points(sigma.f.true, sigma.y.true)
```



Of course, with finite data and three free parameters, the results will not be perfect, but we see that the likelihood contours highlight regions of the parameter space close to the true hyperparameters.

Now, for sampling purposes, we also implement lines 1 - 5 of Algorithm 15.1 which enables us to stably compute the conditional mean and variance parameters for a Gaussian process with a given set of hyperparameters.

```r
gp_pred_mean <- function(x_train, x_pred, y_train,
                         sigma.f, ell, sigma.y){
    K_x <- se_kernel(x_train, x_new = NULL, sigma.f, ell)
    K_x_xprime <- se_kernel(x_train, x_pred, sigma.f, ell)
    K_xprime_xprime <- se_kernel(x_pred, x_new = NULL, sigma.f, ell)
    L <- t(chol(K_x + gp_kernel_noise(x_train, sigma.y)))
    alpha <- qr.solve(t(L), qr.solve(L, y_train))
    E_f_pred <- as.numeric(t(K_x_xprime) %*% alpha)
    return(E_f_pred)
}
gp_pred_var <- function(x_train, x_pred, y_train,
                        sigma.f, ell, sigma.y){
    K_x <- se_kernel(x_train, x_new = NULL, sigma.f, ell)
    K_x_xprime <- se_kernel(x_train, x_pred, sigma.f, ell)
    K_xprime_xprime <- se_kernel(x_pred, x_new = NULL, sigma.f, ell)
    L <- t(chol(K_x + gp_kernel_noise(x_train, sigma.y)))
    alpha <- qr.solve(t(L), qr.solve(L, y_train))
    v <- qr.solve(L, K_x_xprime)
    var_f_pred <- K_xprime_xprime - t(v) %*% v
    return(var_f_pred)
}
```

Now, we see that we can use these three routines to estimate GP hyperparameters and then draw samples for new data. The next section will be devoted to methods for tuning (i.e. optimizing or otherwise carefully selecting) hyperparamters using the GP log-likelihood.

2. Tuning GP hyperparameters

2.a. Stochastic Gradient Descent

There are many ways to optimize this likelihood (or posterior) now that it has been implemented. One such possibility is to use gradient descent, in which we compute the gradient of the objective (either the likelihood or the full posterior) with respect to the Gaussian process hyperparameters and update the parameters iteratively until the gradients are at or close to 0. Below, we implement stochastic gradient descent which performs each parameter update using a gradient calculated on a subset of the data.

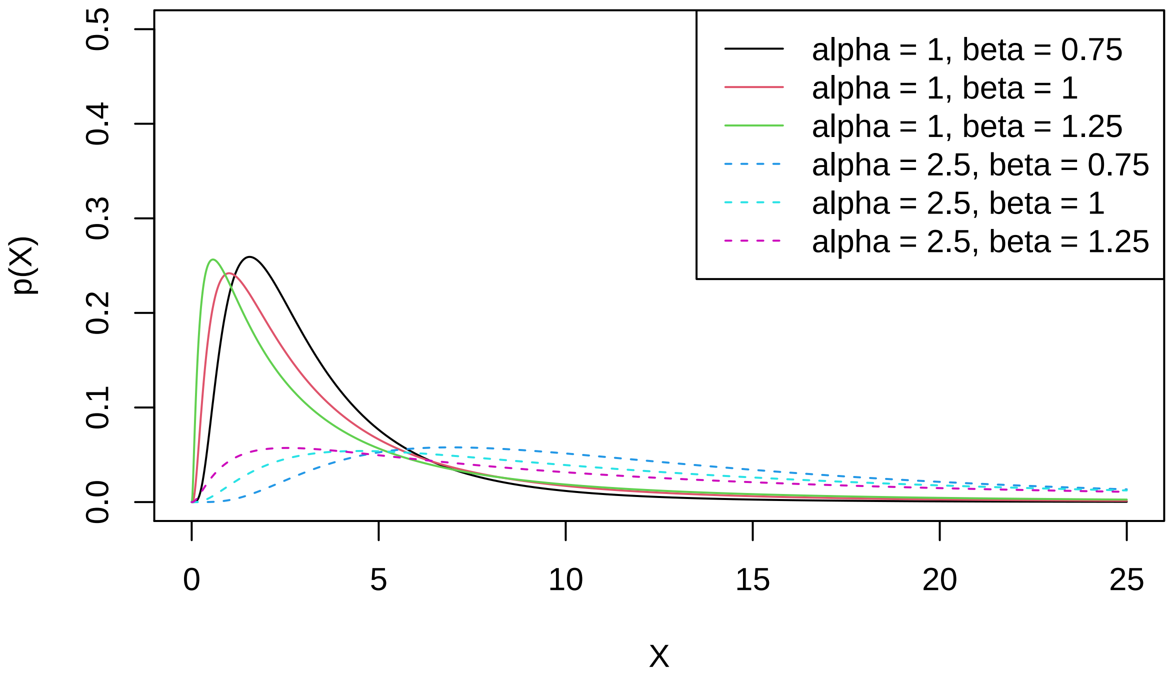
For the posterior, we establish priors for each of σ_f, σ_y and ℓ.

$$\sigma_f, \sigma_y, \ell \sim \text{lognormal}\,(\alpha, \beta)$$

where α is a mean parameter and β is a scale parameter.

We use a brief visual analysis to select α and β parameters that will ensure the optimization process includes a reasonably wide range of parameter values.

```
alpha_vals <- c(1, 1, 1, 2.5, 2.5, 2.5)
beta_vals <- c(0.75, 1, 1.25, 0.75, 1, 1.25)
x_grid <- seq(0.000001, 25, length.out = 2500)
plot(x_grid, dlnorm(x_grid, meanlog = alpha_vals[1], sdlog = beta_vals[1]),
     type = "l", xlab = "X", ylab = "p(X)",
     col = 1, lty = 1, xlim = c(0, 25), ylim = c(0, 0.5))
for (i in 2:length(alpha_vals)){
  if (i > 3){
      lines(x_grid, dlnorm(
        x_grid, meanlog = alpha_vals[i], sdlog = beta_vals[i]
      ), col = i, lty = 2)
  } else{
    lines(x_grid, dlnorm(
      x_grid, meanlog = alpha_vals[i], sdlog = beta_vals[i]
    ), col = i, lty = 1)
  }
}
legend_labels <- c("alpha = 1, beta = 0.75",
                   "alpha = 1, beta = 1",
                   "alpha = 1, beta = 1.25",
                   "alpha = 2.5, beta = 0.75",
                   "alpha = 2.5, beta = 1",
                   "alpha = 2.5, beta = 1.25")
legend("topright", legend_labels, col = 1:6, lty = c(1,1,1,2,2,2))
```



It looks like lognormal$(1, 1.25)$ provides a good balance of small parameter values with a long tail as well.

Now for our implementation of SGD to optimize our hyperparameters. The first helper functions we write compute the gradients of the kernel with respect to the hyperparameters.

We note first and foremost that our searches on hyperparameters are performed in log-space, so that the first operation performed in most of the functions defined below is to exponentiate the parameters so that they are > 0. This is an optimization trick suggested in 15.2.4 of Murphy [2012].

```r
############################################################################
# Split the partial derivative of the kernel function into
# two parts: a function that operates on all elements of the distance
# matrix and a function that operates on the diagonal elements only
# The reason we need to split it up this way is the "indicator" function
# I(X == a) is not a primitive that can be symbolically differentiated by R

# This is the function that works on all elements of the distance matrix
# Note, to make the results more numerically stable, we reparameterize
# as theta_new = exp(theta_old) and we plug that into the function to search
# in log space
# Since all of sigma.f, sigma.y and ell are truncated below at
# zero, and in particular, in the case of sigma.y for this simulation,
# the "true" value is close to zero, this makes the gradient
# more stable by transforming a small range of values (0.0001 vs 0.001)
# into a wider range of (positive and negative) values
kernderiv.all <- deriv(
    ~(((exp(sigma.f))^2)*exp(-(x_dist)/(2*((exp(ell))^2)))), name = c("sigma.f", "ell", "sigma.y"),
    function.arg = c("sigma.f", "ell", "sigma.y", "x_dist")
)
# This the function that only operates on diagonal elements of the matrix
kernderiv.diag.only <- deriv(
    ~(((exp(sigma.y))^2)), name = c("sigma.f", "ell", "sigma.y"),
    function.arg = c("sigma.f", "ell", "sigma.y", "x_dist")
)
```

```
# This combines the two functions, ensuring that
# it is only run when dist < numeric zero (which could technically include
# very close points, but this is unlikely to occur in our simulations)
kernderiv <- function(sigma.f, ell, sigma.y, x_dist, param_name = NULL){
    if (is.null(param_name)){
        entire.matrix.output <- unname(as.numeric(
          attr(kernderiv.all(sigma.f, ell, sigma.y, x_dist), "gradient")
        ))
        diag.entries.only <- unname(as.numeric(
          attr(kernderiv.diag.only(sigma.f, ell, sigma.y, x_dist), "gradient")
        ))
    } else{
        entire.matrix.output <- unname(
          attr(kernderiv.all(sigma.f, ell, sigma.y, x_dist), "gradient")[,param_name]
        )
        diag.entries.only <- unname(
          attr(kernderiv.diag.only(sigma.f, ell, sigma.y, x_dist), "gradient")[,param_name]
        )
    }
    outval <- entire.matrix.output
    if ((abs(x_dist) < .Machine$double.eps)){
        outval <- outval + diag.entries.only
    }
    return(outval)
}
```

Using the above functions, we can compute the gradient of the log likelihood as specified in (15.24) of Murphy [2012].

$$\frac{\partial}{\partial \theta_j} \log p(y \mid X) = \frac{1}{2} \text{tr} \left(\left(\alpha\alpha' - K_y^{-1} \right) \frac{\partial K_y}{\partial \theta_j} \right)$$

where K_y is a matrix whose elements are defined as in (15.19) of Murphy [2012]:

$$K_{y,i,j} = \sigma_f^2 \exp \left(-\frac{(x_i - x_j)^2}{2\ell^2} \right) + \sigma_y^2 \delta_{ij}$$

and α is defined as in algorithm 15.1 of Murphy [2012] (details in section 1).

```
loglik.deriv <- function(sigma.f, ell, sigma.y, x, y){
    # This computes the (alpha alpha^T - K_y^{-1})
    # matrix which is common to the gradient for each of the three parameters
    dist_mat <- sq_dist_mat(x)
    K <- ((exp(sigma.f))^2)*exp(-(dist_mat)/(2*((exp(ell))^2)))
    K_y <- K + ((exp(sigma.y))^2)*diag(nrow(dist_mat))
    K_y_inv <- solve(K_y)
    # L <- t(chol(K_y))
    # alpha <- qr.solve(t(L), qr.solve(L, y))
    # Note: here we simply invert K_y, although we could also
    # use the backslash approach in Murphy (2012)
    alpha <- K_y_inv %*% y
    shared_matrix <- (alpha %*% t(alpha) - K_y_inv)
    # Compute the partial derivatives of K_y with respect to each of the parameters
    dK.d.sigma.f <- apply(dist_mat, c(1,2),
                          function(z) kernderiv(sigma.f,ell,sigma.y,z,"sigma.f"))
```

```r
    dK.d.ell <- apply(dist_mat, c(1,2),
                      function(z) kernderiv(sigma.f,ell,sigma.y,z,"ell"))
    dK.d.sigma.y <- apply(dist_mat, c(1,2),
                          function(z) kernderiv(sigma.f,ell,sigma.y,z,"sigma.y"))
    # Run each of the matrix multiplications
    pretrace.sigma.f <- shared_matrix %*% dK.d.sigma.f
    pretrace.ell <- shared_matrix %*% dK.d.ell
    pretrace.sigma.y <- shared_matrix %*% dK.d.sigma.y
    # Return the partial derivative with respect to each of the three parameters
    dloglik.d.theta <- c(
        0.5*sum(diag(pretrace.sigma.f)),
        0.5*sum(diag(pretrace.ell)),
        0.5*sum(diag(pretrace.sigma.y))
    )
    names(dloglik.d.theta) <- c("sigma.f", "ell", "sigma.y")
    return(dloglik.d.theta)
}
```

For each prior we consider using in posterior computation, we will need to derive its gradient with respect to the hyperparameter (which is treated as "the data" in the prior). Below, we implement a gradient of the log prior for a lognormal distribution.

```r
# This differentiates the provided expression symbolically and
# returns a function that evaluates the gradient of the log prior
log.prior.deriv <- deriv(
    ~(
        log((1/(b*exp(sigma.f)*sqrt(2*pi)))*exp(-(((log(exp(sigma.f))-a)^2)/(2*(b^2))))) +
        log((1/(b*exp(ell)*sqrt(2*pi)))*exp(-(((log(exp(ell))-a)^2)/(2*(b^2))))) +
        log((1/(b*exp(sigma.y)*sqrt(2*pi)))*exp(-(((log(exp(sigma.y))-a)^2)/(2*(b^2)))))
    ),
    name = c("sigma.f", "ell", "sigma.y"),
    function.arg = c("sigma.f", "ell", "sigma.y", "a", "b")
)
# This function simply evaluates the log prior
log.prior <- function(sigma.f, ell, sigma.y, a, b){
  out_val <- (log((1/(b*exp(sigma.f)*sqrt(2*pi)))*exp(-(((log(exp(sigma.f))-a)^2)/(2*(b^2))))) +
   log((1/(b*exp(ell)*sqrt(2*pi)))*exp(-(((log(exp(ell))-a)^2)/(2*(b^2))))) +
   log((1/(b*exp(sigma.y)*sqrt(2*pi)))*exp(-(((log(exp(sigma.y))-a)^2)/(2*(b^2))))))
  return(out_val)
}
```

Now we can combine these two gradients additively to approximate the gradient of the log posterior with respect to the hyperparameters (up to a constant). We also define a function that directly evaluates the proportional log posterior, which we can use for derivative free optimization methods.

```r
# As above, create a function that evaluates the
# gradient of the log posterior
log.posterior.deriv <- function(sigma.f, ell, sigma.y, x, y, a, b){
    # Compute the log-likelihood gradient
    loglik.grad <- loglik.deriv(sigma.f, ell, sigma.y, x, y)
    # Compute the log-prior gradient
    log.prior.grad <- as.numeric(attr(log.prior.deriv(sigma.f, ell, sigma.y, a, b), "gradient"))
    names(log.prior.grad) <- c("sigma.f", "ell", "sigma.y")
    return(loglik.grad + log.prior.grad)
}
```

```r
# Evaluate the log posterior
log.posterior <- function(sigma.f, ell, sigma.y, x, y, a, b){
    # Compute the log-likelihood gradient
    loglik <- gploglik(x, y, exp(sigma.f), exp(ell), exp(sigma.y))
    # Compute the log-prior gradient
    log.prior <- log.prior(sigma.f, ell, sigma.y, a, b)
    return(loglik + log.prior)
}
```

Now, we write a function to compute a given number of stochastic gradient descent steps. This function takes a number of inputs:

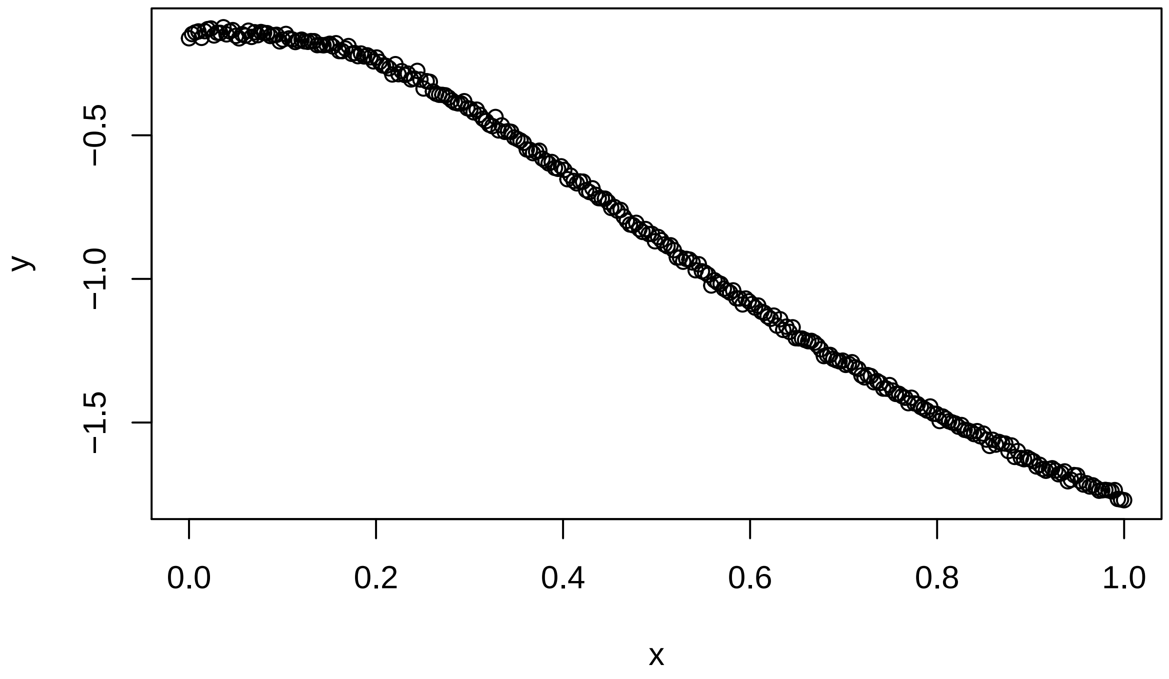
- `x`: the independent values of the regression problem
- `y`: dependent values of the regression problem
- `batch_size`: the number of samples in each batch of data used for a gradient step
- `n_iter`: the total number of iterations to run through the entire dataset
- `stop_iter`: early stopping parameter, specifies a number of iterations after which the algorithm should be stopped if the objective has not shifted
- `convergence_limit`: a threshold value below which successive iterations of hyperparameter values will not be considered as "different" (assessed in terms of mean squared differences)
- `n_restarts`: the number of times the algorithm should be randomly restarted at different initial values of the hyperparameters
- `step_size`: value by which the objective gradient is scaled before taking a gradient step
- `momentum_gamma`: algorithm parameter that preserves information about previous gradient steps
- `opt_type`: likelihood or posterior (currently, using a lognormal prior)
- `prior_parameter_a`: the first parameter of the lognormal prior (used for initialization even if opt_type == "likelihood")
- `prior_parameter_b`: the second parameter of the lognormal prior (used for initialization even if opt_type == "likelihood")

```r
sgd.optimize <- function(x, y, batch_size, n_iter, stop_iter, convergence_limit,
                         n_restarts, step_size, momentum_gamma, opt_type,
                         prior_parameter_a, prior_parameter_b){
  # Data dimensions
  n <- length(y)
  # Batch construction
  num_batches <- n %/% batch_size
  remainder_batch_size <- n %% batch_size
  if (remainder_batch_size > 0){
      total_batches <- num_batches + 1
  } else {
      total_batches <- num_batches
  }
  batch_labels_for_resampling <- rep(1:num_batches, each = batch_size)
  if (total_batches > num_batches){
      batch_labels_for_resampling <- c(
          batch_labels_for_resampling,
          rep(total_batches, n - length(batch_labels_for_resampling))
      )
```

```r
}
# Matrix to store results
result_mat <- matrix(NA, nrow = n_restarts, ncol = 3)
iter_mat <- matrix(NA, nrow = (n_iter*n_restarts), ncol = 3)
for (k in 1:n_restarts){
    # Randomly initialize the parameters
    # (log-transforming for stability)
    sigma.f.star <- log((rlnorm(1, prior_parameter_a, prior_parameter_b)))
    ell.star <- log((rlnorm(1, prior_parameter_a, prior_parameter_b)))
    sigma.y.star <- log((rlnorm(1, prior_parameter_a, prior_parameter_b)))

    # Split the data into batches
    batch_labels <- sample(batch_labels_for_resampling, size = n, replace = F)
    # Initialize the momentum vector
    v1 <- v2 <- v3 <- 0

    # Run the algorithm from each random starting point for n_iter epochs
    for (i in 1:n_iter){
        # Run one gradient descent step on each mini-batch of data
        for (j in 1:total_batches){
            # Take subset of the data
            x_subset <- x[batch_labels == j]
            y_subset <- y[batch_labels == j]
            n_j <- length(y_subset)
            # Compute the approximate gradient on that sample
            if (opt_type == "posterior"){
              param_grad <- log.posterior.deriv(sigma.f.star, ell.star, sigma.y.star,
                                                x_subset, y_subset, prior_parameter_a,
                                                prior_parameter_b)
            } else if (opt_type == "likelihood"){
              param_grad <- loglik.deriv(sigma.f.star, ell.star, sigma.y.star,
                                         x_subset, y_subset)
            }

            # Conduct one update of all of the parameters
            v1 <- momentum_gamma*v1 + step_size*param_grad["sigma.f"]
            v2 <- momentum_gamma*v2 + step_size*param_grad["ell"]
            v3 <- momentum_gamma*v3 + step_size*param_grad["sigma.y"]
            sigma.f.star <- sigma.f.star + v1
            ell.star <- ell.star + v2
            sigma.y.star <- sigma.y.star + v3
        }
        iter_mat[(k-1)*n_iter + i,1] <- sigma.f.star
        iter_mat[(k-1)*n_iter + i,2] <- ell.star
        iter_mat[(k-1)*n_iter + i,3] <- sigma.y.star

        # Early stopping
        if (i > stop_iter){
            max_params <- apply(
                iter_mat[(((k-1)*n_iter + i - stop_iter + 1):((k-1)*n_iter + i)),],
                2, max)
            min_params <- apply(
                iter_mat[(((k-1)*n_iter + i - stop_iter + 1):((k-1)*n_iter + i)),],
                  2, min)
              if (sum(max_params - min_params)^2 < convergence_limit){
                  break
              }
          }
      }
      result_mat[k,1] <- sigma.f.star
      result_mat[k,2] <- ell.star
      result_mat[k,3] <- sigma.y.star
  }
  # Remove NAs left after early stopping
  iter_mat <- iter_mat[apply(is.na(iter_mat), 1, sum) == 0, ]
  return(list(result_mat = result_mat, iter_mat = iter_mat))
}
```

With this implemented, we can test that it converges to the "correct" hyperparameters for a simulated dataset.

```
# Draw the data
n <- 300
sigma.f <- 1; ell <- 0.5; sigma.y <- 0.01
x <- seq(0, 1, length.out = n)
cov_y_given_x <- (se_kernel(x, sigma.f = sigma.f, ell = ell) +
                    gp_kernel_noise(x, sigma.y = sigma.y))
y <- mvrnorm(n = 1, mu = rep(0, n), Sigma = cov_y_given_x)

# Plot the data
plot(x, y)
```

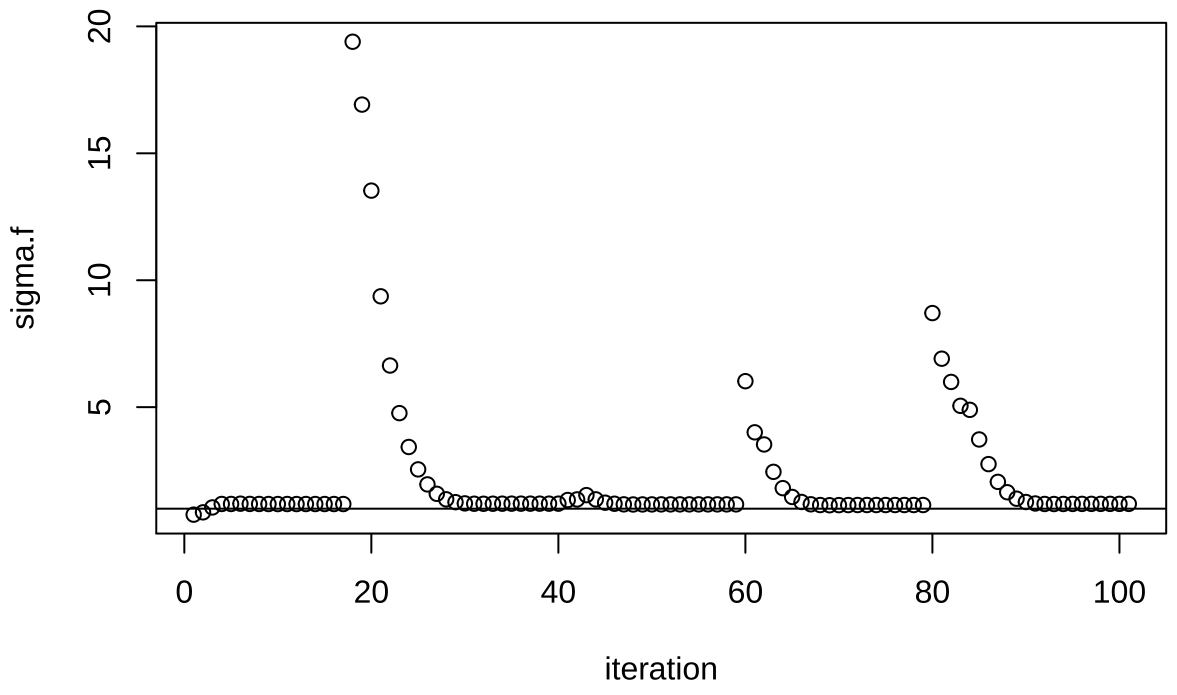


We optimize the likelihood with respect to the parameters

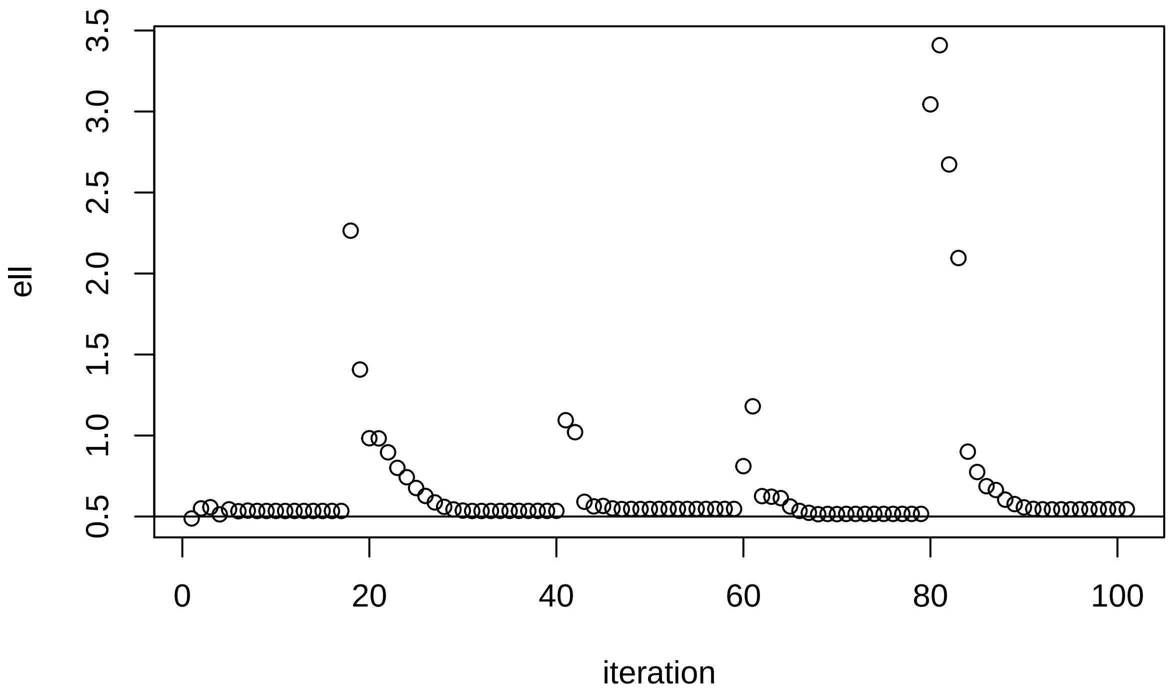
```
opt_results_lik <- sgd.optimize(
  x, y, batch_size = 20, n_iter = 1000, stop_iter = 10,
  convergence_limit = 0.0001, n_restarts = 5, step_size = 0.001,
  momentum_gamma = 0.9, opt_type = "likelihood",
  prior_parameter_a = 1, prior_parameter_b = 1.25
)
```

We can visualize the results and see that the model generally descends to the correct values.

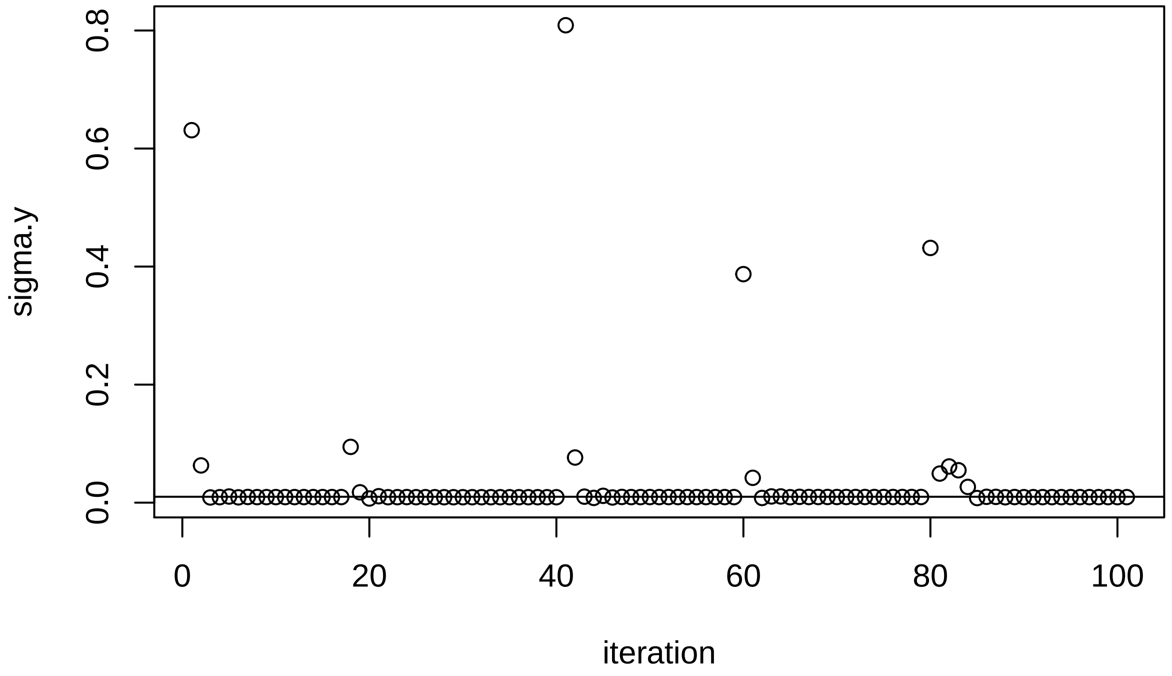
```
result_mat <- opt_results_lik[[1]]
iter_mat <- opt_results_lik[[2]]
num_plot_points <- nrow(iter_mat)
n_restarts <- nrow(result_mat)
# Look at a graph of the parameters by iteration of the algorithm
plot(1:num_plot_points, exp(iter_mat[,1]), xlab = "iteration", ylab = "sigma.f")
abline(h = sigma.f)
```



```
plot(1:num_plot_points, exp(iter_mat[,2]), xlab = "iteration", ylab = "ell")
abline(h = ell)
```



```
plot(1:num_plot_points, exp(iter_mat[,3]), xlab = "iteration", ylab = "sigma.y")
abline(h = sigma.y)
```



```
# # Average the final parameter estimates across each restart
mean(exp(result_mat[,1]))
```

```
## [1] 1.177825
```

```
mean(exp(result_mat[,2]))
```

```
## [1] 0.5356012
```

```r
mean(exp(result_mat[,3]))
```

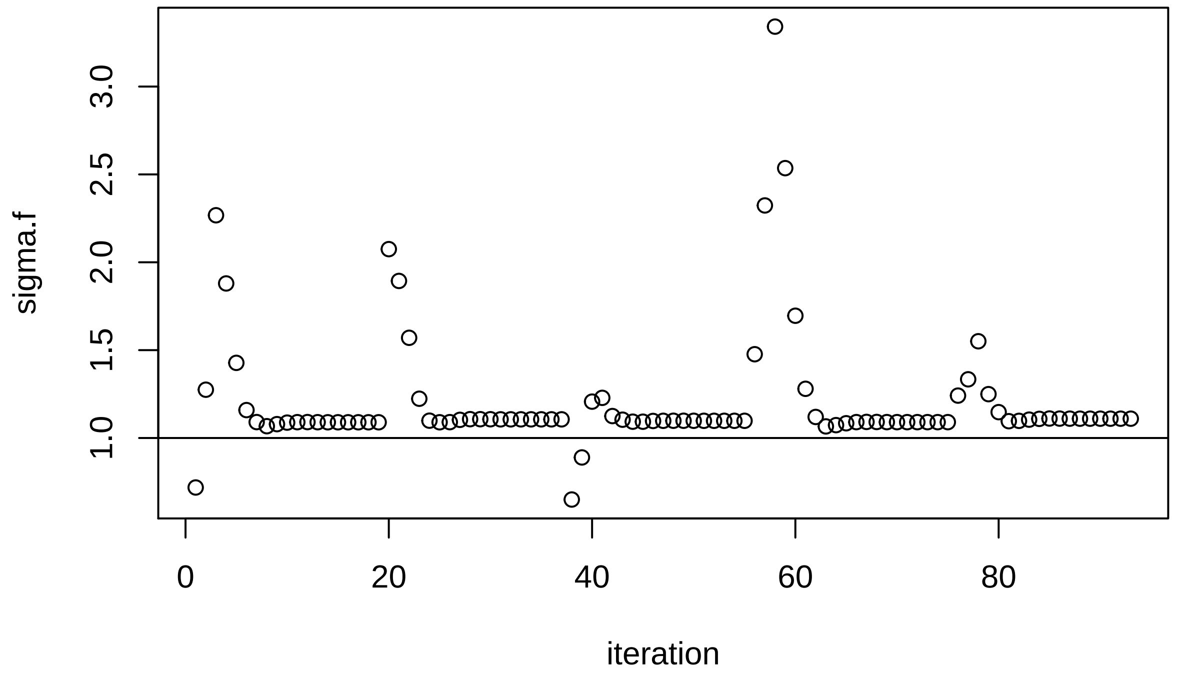
```
## [1] 0.009425324
```

And we repeat the same process for the posterior

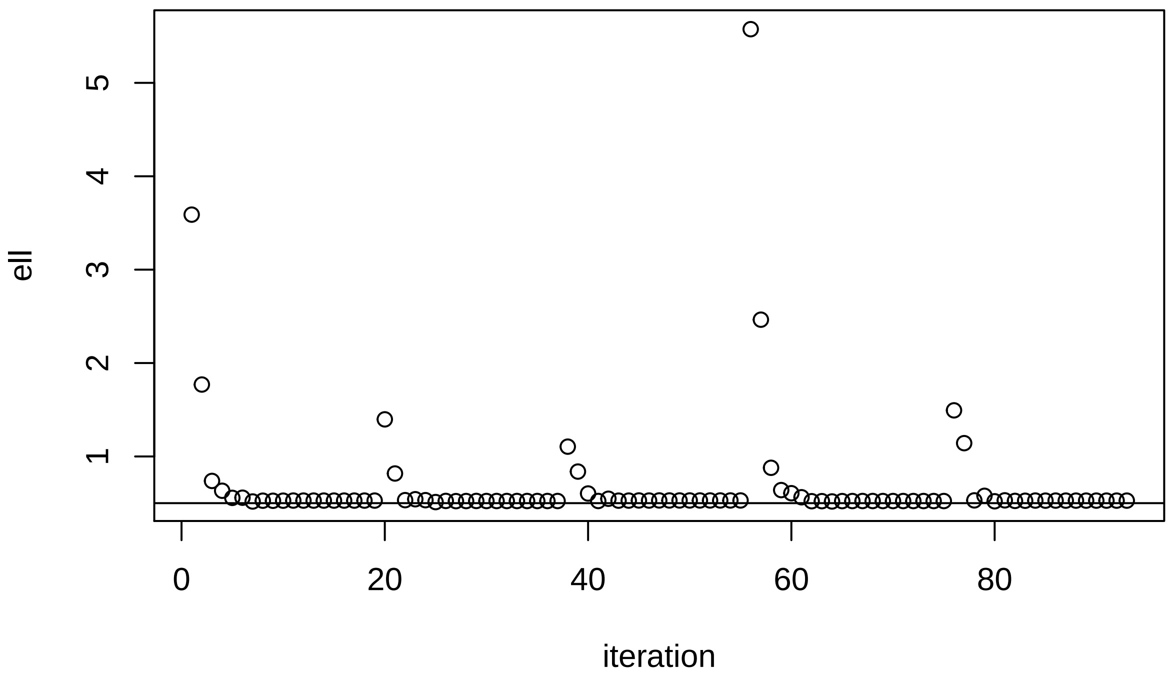
```r
opt_results_post <- sgd.optimize(
  x, y, batch_size = 20, n_iter = 1000, stop_iter = 10,
  convergence_limit = 0.0001, n_restarts = 5, step_size = 0.001,
  momentum_gamma = 0.9, opt_type = "posterior",
  prior_parameter_a = 1, prior_parameter_b = 1.25
)
```

And again we visualize the results.

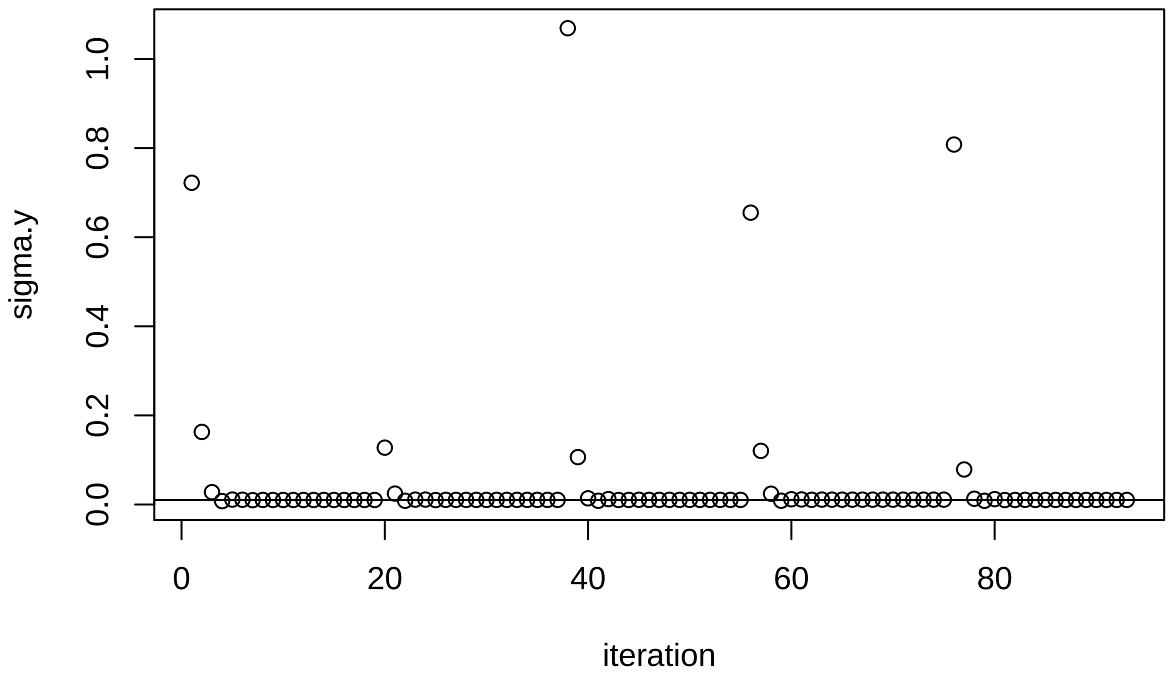
```r
result_mat <- opt_results_post[[1]]
iter_mat <- opt_results_post[[2]]
num_plot_points <- nrow(iter_mat)
n_restarts <- nrow(result_mat)
# Look at a graph of the parameters by iteration of the algorithm
plot(1:num_plot_points, exp(iter_mat[,1]), xlab = "iteration", ylab = "sigma.f")
abline(h = sigma.f)
```



```r
plot(1:num_plot_points, exp(iter_mat[,2]), xlab = "iteration", ylab = "ell")
abline(h = ell)
```



```
plot(1:num_plot_points, exp(iter_mat[,3]), xlab = "iteration", ylab = "sigma.y")
abline(h = sigma.y)
```



```
# # Average the final parameter estimates across each restart
mean(exp(result_mat[,1]))
```

```
## [1] 1.098771
```

```
mean(exp(result_mat[,2]))
```

```
## [1] 0.5258038
```

```
mean(exp(result_mat[,3]))
```

```
## [1] 0.01042312
```

We close this section by writing a helper function that runs SGD and choose the very last set of hyperparameter estimates from the last restart of the algorithm as the "optimal" hyperparameters.

```
sgd.optimize.hyperparams <- function(
        x, y, batch_size, n_iter, stop_iter, convergence_limit, n_restarts, step_size,
        momentum_gamma, opt_type, prior_parameter_a, prior_parameter_b
    ){
    # Return a point estimate from the SGD algorithm
    sgd_results <- sgd.optimize(
        x, y, batch_size, n_iter, stop_iter, convergence_limit, n_restarts, step_size,
        momentum_gamma, opt_type, prior_parameter_a, prior_parameter_b
    )
    if (is.null(dim(sgd_results[[1]]))){
        return(exp((sgd_results[[1]])[length(sgd_results[[1]])]))
    }
    else {
        return(exp((sgd_results[[1]])[nrow(sgd_results[[1]]),]))
    }
}
```

2.b. Posterior Sampling (Metropolis-within-gibbs)

We now implement a Metropolis-within-Gibbs Sampler to draw samples from the posterior distribution of hyperparameters. We specify the same priors on the hyperparameters as before, that is

$$\sigma_f, \sigma_y, \ell \sim \text{lognormal}\left(\alpha, \beta\right),$$

where α and β are lognormal mean and scale parameters. Using the logic of a Gibbs sampler we iterate through the procedure by taking draws of the posterior by keeping all but one of the hyperparameters constant. We implement posterior sampling by using the Metropolis Hastings algorithm for each hyperparameter draw. We do this for each hyperparameter iteratively holding the other constant in the presence of a hyperparameter draw. We complete this procedure for a desired number of iterations.

The procedure is expressed as follows:

We have $\boldsymbol{\theta} = (\sigma_f, \ell, \theta_y)$ with a joint distribution denoted by the product of three independent lognormal random variables. We can derive expressions for the posterior distribution of one particular hyparameter θ_i $i \in \{1, 2, 3\}$ in the presence of the others. At i we hold all other hyperparameters constant. Call the fixed remaining hyperparameters θ_{-i}. Then we can evaluate:

$$\log(p(\theta_i|\theta_{-i}, y)) \propto \log(p(y|x, \boldsymbol{\theta})) + \log(p(\boldsymbol{\theta}))$$

.

For each Monte Carlo draw, $\boldsymbol{\theta}^t$, we obtain a proposed value for hyperparameter i, ${\theta_i}^t$, from a lognormal distribution with mean at the previous value, $q(\theta_i^t|\theta_i^{t-1})$. Define the acceptance probability α_i as,

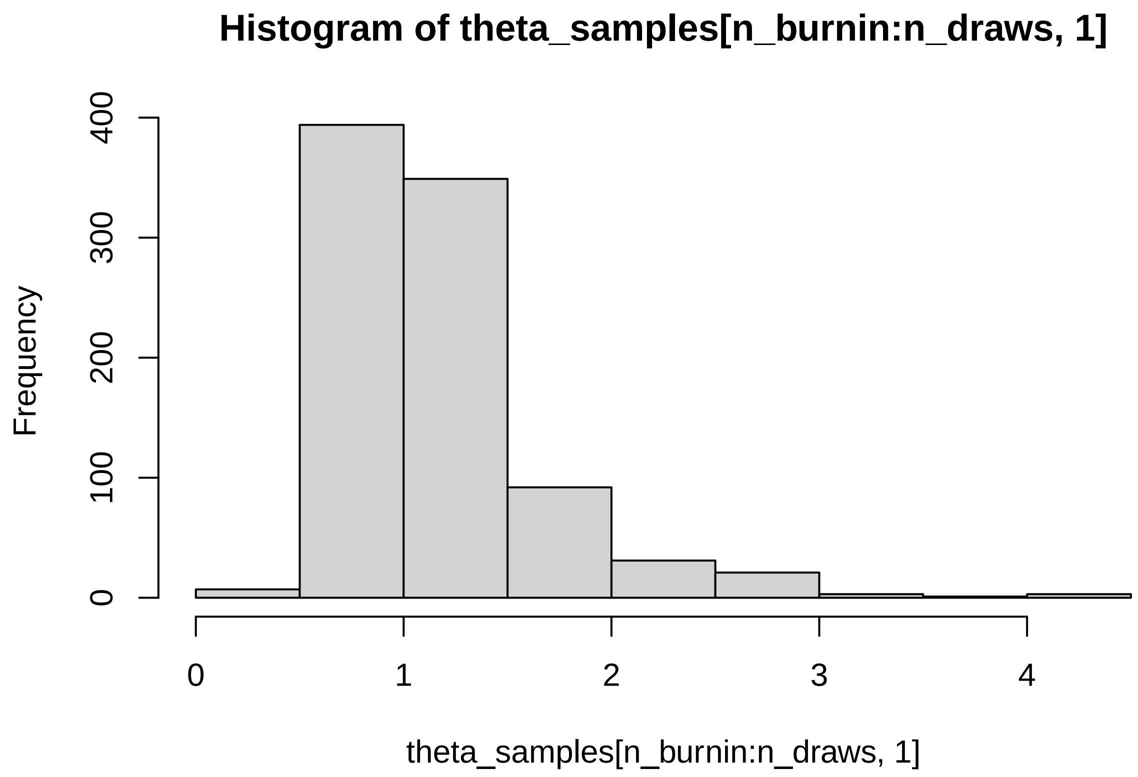
$$\alpha_i = \min\left\{1, \frac{p({\theta^t}_i|{\theta^t}_{-i}, y)}{p({\theta^{t-1}}_i|{\theta^{t-1}}_{-i}, y)} \times \frac{q(\theta_i^{t-1}|\theta_i^{t-1})}{q(\theta_i^t|\theta_i^{t-1})}\right\}$$

And $q(\cdot)$ can be omitted if the proposal distribution is symmetric. We draw from a binomial distribution with this as the acceptance probability and if there is a success we keep the proposed value or else we keep the previous value for our Monte Carlo draw of the ith hyparameter. We do this for all hyperparameters for a full Monte Carlo draw of $\boldsymbol{\theta}$

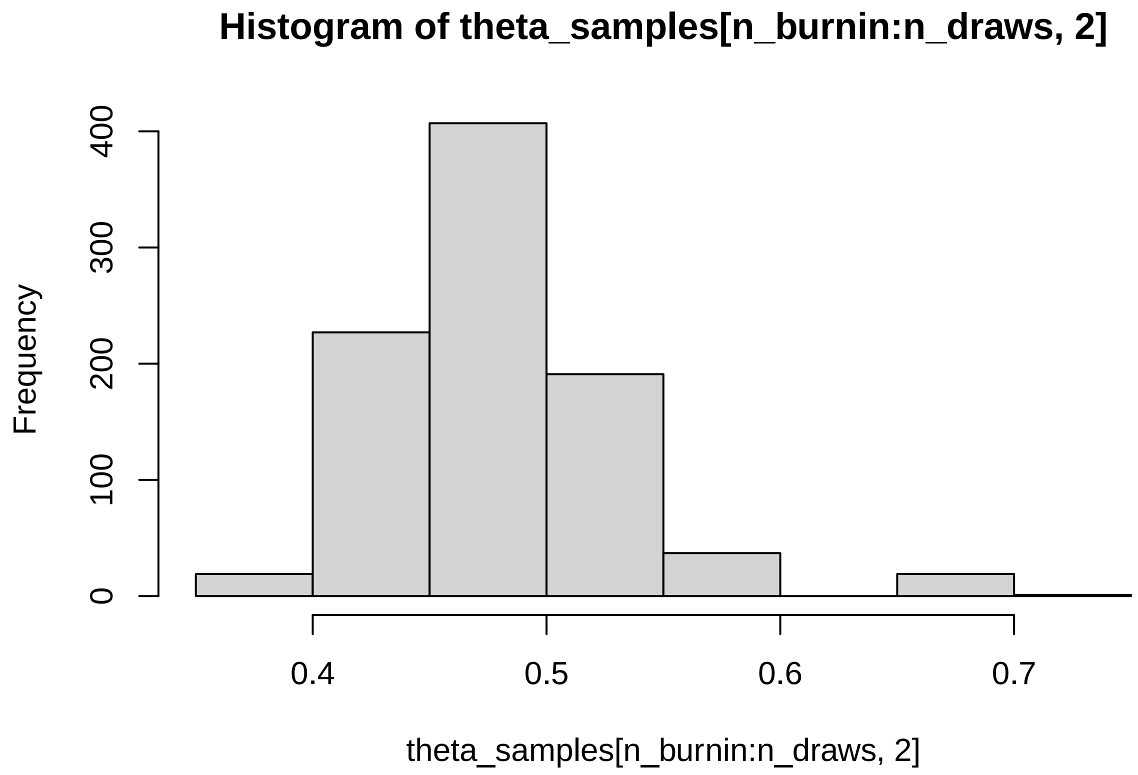
We make use of the previously discussed function `log.posterior` to implement this procedure. The function implemented makes use of the following inputs:

- `x`: the independent values of the regression problem
- `y`: dependent values of the regression problem
- `n_draws`: the total number of times to run through the Monte Carlo procedure
- `n_burning`: the total number of draws to consider as "burn-in" samples before the algorithm (hopefully) converges to the stationary distribution.
- `proposal_sigma`: the standard deviation of the normal distribution used to draw proposals

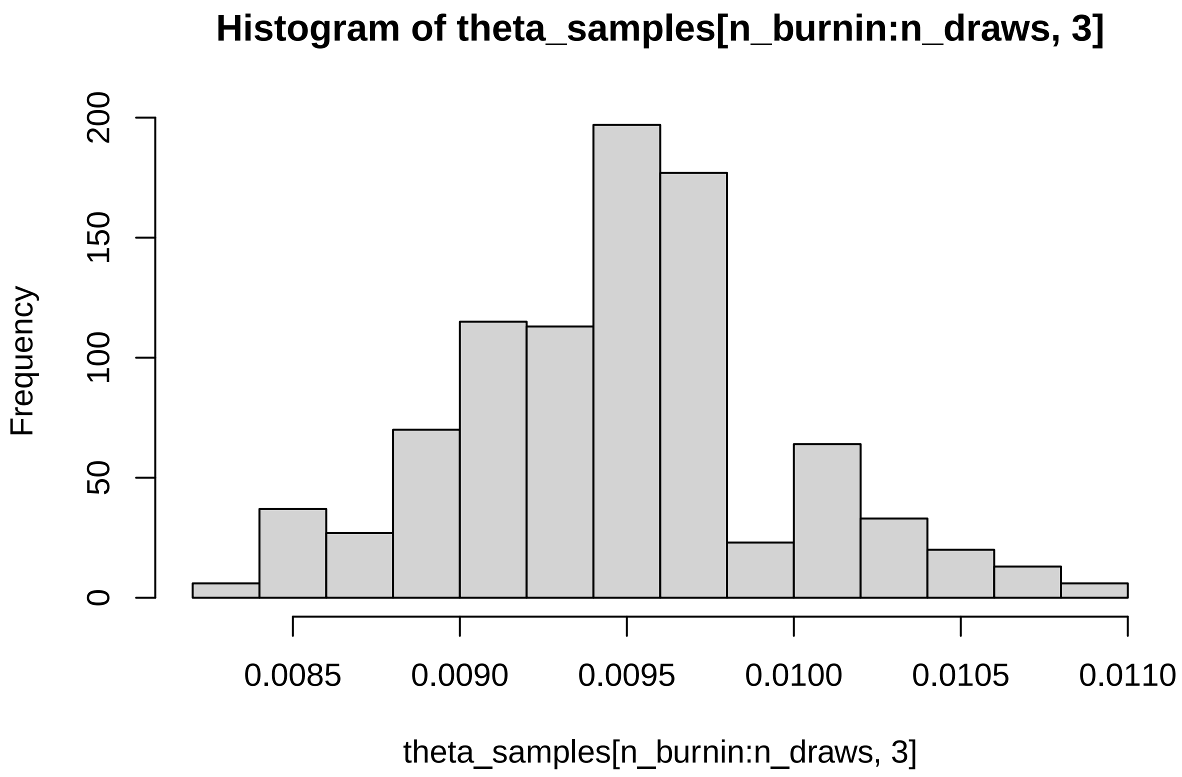
```r
# Simulation infrastructure
n_draws <- 1000
n_burnin <- 100
theta_samples <- matrix(NA, nrow = n_draws+1, ncol = 3)
proposal_sigma <- 1

# Initial guess
# Exponentiated normal
theta_samples[1,] <- exp(rnorm(3, 0, proposal_sigma))

for (i in 1:n_draws){
  for (j in 1:3){
    # Store the log of the current value of theta
    theta_current <- log(theta_samples[i,])
    # Draw a proposal
    theta_proposed <- theta_current
    theta_proposed[j] <- rnorm(1, theta_current[j], proposal_sigma)
    # Evaluate the posterior probability of the proposal
    p_proposal <- log.posterior(theta_proposed[1], theta_proposed[2],
                                theta_proposed[3], x, y, a = 1, b = 1.25)
    p_current <- log.posterior(theta_current[1], theta_current[2],
                               theta_current[3], x, y, a = 1, b = 1.25)
    # Determine whether or not to accept the proposal
    alpha_proposal <- exp(p_proposal - p_current)
    acceptance_prob <- min(1, alpha_proposal)
    # Accept or reject
    verdict <- rbinom(1, 1, acceptance_prob)
    if (verdict == 1){
      theta_samples[i+1,j] <- exp(theta_proposed[j])
    } else{
      theta_samples[i+1,j] <- exp(theta_current[j])
    }
  }
}
hist(theta_samples[n_burnin:n_draws,1])
```



```
hist(theta_samples[n_burnin:n_draws,2])
```



```
hist(theta_samples[n_burnin:n_draws,3])
```



2.c. Bayesian Optimization

Another approach we can use to investigate and tune the hyperparameters of our Gaussian process is Bayesian optimization (Mockus et al. [1978]).

The irony of using a Gaussian process to pick hyperparameters for another Gaussian process is not lost on us, but we were curious how this method would perform on the task at hand.

Bayesian optimization refers to not one technique but an entire family of techniques and here we implement the Expected Inprovement (EI) method, first introduced by Schonlau [1997] and discussed in depth by Gramacy [2020].

However, our prior implementation of the Gaussian process was for univariate independent variables. In this case, we have three parameters to tune in our univariate Gaussian process regression, so the "data" in the second-order Gaussian process are multivariate (with three dimensions). Thus, we must implement a version of the Gaussian process which computes a kernel similarity matrix on three-dimensional inputs. We can do this fairly simply using (15.20) in Murphy [2012].

We use the squared exponential kernel as above, where the distance between two points is now a matrix operations

$$k(x_1, x_2) = \sigma_f^2 \exp\left(-(x_1 - x_2)' L (x_1 - x_2)\right) + \sigma_y^2 I$$

where L is a matrix of hyperparameters. Murphy [2012] note that two common choices are $L = \ell^{-2} I$ for one fixed ℓ hyperparameter or $L = (\ell_1, \ell_2, \cdots, \ell_p) I_p$ where each ℓ_i is different for each hyperparameter in the original regression problem. This is particularly helpful if the hyperparameters are on different scales, although it comes at a cost of having to optimize more hyperparameters in the second order surrogate Gaussian process.

Rather than implement these modifications directly, we rely on external libraries to fit a Gaussian process to multivariate independent values as well as to draw samples from a space-filling design.

We implement the approach described in Algorithm 6.1 of Gramacy [2020], which we present below in the direct verbiage of our problem.

0. (Not technically part of the algorithm, but we note ahead of time that the data x and y referenced above are treated as fixed values of the original Gaussian process regression problem. That is, we do not seek to augment or modify this dataset, we instead treat the hyperparameters of our original Gaussian process as "data" and the log-likelihood or the log-posterior of that Gaussian process, evaluated using different hyperparameters as "outcomes.")
1. Generate some initial values of the hyperparameters.

a. This can be done using any method, but a common choice is a space-filling design, such as a latin hypercube, which we can implement in R using `lhs`

b. For each of the sets of hyperparameters, evaluate the log-likelihood or log-posterior.

2. Fit a Gaussian Process to the dataset of hyperparameters and log-probabilities (likelihood or posterior), tuning the hyperparameters on that second order GP by any method (SGD / maximum likelihood).
3. Choose a "new" set of hyperparameters by maximizing an objective function, such as Expected Improvement (described in section 7 of Gramacy [2020] and later in this writeup).
4. Evaluate the log-likelihood or log-posterior at the new hyperparameter values, refit the Gaussian process with one additional pair of data samples.

This process is repeated until some pre-specified stopping point at which point the hyperparameters in the dataset with the largest log-probability are chosen to be the "optimal" hyperparameters.

Here we demonstrate the inner workings of the method before wrapping it in a function.

```r
# Generate some initial hyperparameters
n_guesses <- 50
num_hparam <- 3
hparam_dataset <- log(randomLHS(n_guesses, num_hparam)*2)
hparam_post <- apply(
  hparam_dataset, 1, function(z) log.posterior(
    z[1], z[2], z[3], x, y, a = 1, b = 1.25
  ))

# Fit a Gaussian process to the "data" (rescaled to be in [0,1])
my_model <- GP_fit(exp(hparam_dataset)/2, hparam_post)
```

Below we define the Expected Improvement (EI) function which scores GP predictions for new hyperparameters according to their expected ability to "improve" on the current optimum in the dataset.

Equation (7.3) in Gramacy [2020] expresses this function as

$$EI(x) = (f^* - \hat{\mu}(x))\, \Phi\left(\frac{f^* - \hat{\mu}(x)}{\hat{\sigma}(x)}\right) + \hat{\sigma}(x)\phi\left(\frac{f^* - \hat{\mu}(x)}{\hat{\sigma}(x)}\right)$$

where f^* is the current value of the log posterior at the optimal set of hyperparameters in the dataset of candidate hyperparameters, $\hat{\mu}(x)$ is the estimated log posterior at a new candidate sample of hyperparameter, as predicted by a Gaussian process trained on the existing hyperparameter candidates, $\hat{\sigma}(x)$ is the estimated variance of the log posterior predictions for the Gaussian process surrogate model, Φ is the standard normal CDF, and ϕ is the standard normal pdf.

We implement the function with an additional ϵ term subtracted from $f^* - \hat{\mu}(x)$.

```r
ei_calc <- function(m, s, y_opt_prev, eps) {
  if (s == 0) {
    return(0)
  }
  Z <- (m - y_opt_prev - eps)/s
  expected_imp <- (m - y_opt_prev - eps) * pnorm(Z) + s * dnorm(Z)
  return(expected_imp)
}
```

Now we define a function that evaluates EI for each set of candidate hyperparameters and returns the index of the hyperparameter set that maximizes EI.

```r
BO.iter <- function(hparam_candidates, surrogate_model, eps, y_opt_prev){
  # Predict the outcome on the candidates
  pred <- predict.GP(my_model, xnew = data.frame(x = exp(hparam_candidates)/2))
  mu <- pred$Y_hat
  sigma <- sqrt(pred$MSE)

  # Evaluate the "Expected Improvement" on each candidate point
  EI_values <- apply(cbind(mu, sigma), 1, function(z) ei_calc(z[1], z[2], y_opt_prev = y_opt_prev, eps
  new_sample_ind <- which(EI_values == max(EI_values))
  return(new_sample_ind)
}
```

We run two iterations of this algorithm as a quick check that it works

```r
# Initial values for the algorithm
hparam_best <- min(-hparam_post)
eps <- 0.01

# Generate a large set of "candidate" values
n_candidates <- 300
num_hparam <- 3
hparam_candidates <- log(randomLHS(n_candidates, num_hparam)*2)

# Run the algorithm
new_sample_ind <- unname(BO.iter(hparam_candidates, my_model, eps, hparam_best))
new_sample <- hparam_candidates[new_sample_ind, ]
hparam_candidates <- hparam_candidates[(1:nrow(hparam_candidates) != new_sample_ind), ]

# Evaluate the log-posterior
new_lp <- log.posterior(new_sample[1], new_sample[2], new_sample[3],
                        x, y, a = 1, b = 1.25)
# Add to the "dataset" of hyperparam / log-posterior pairs
hparam_dataset <- rbind(hparam_dataset, new_sample)
hparam_post <- c(hparam_post, new_lp)

# Refit the model
my_model <- GP_fit(exp(hparam_dataset)/2, hparam_post)

# Run the algorithm
new_sample_ind <- unname(BO.iter(hparam_candidates, my_model, eps, hparam_best))
new_sample <- hparam_candidates[new_sample_ind, ]
hparam_candidates <- hparam_candidates[(1:nrow(hparam_candidates) != new_sample_ind), ]

# Evaluate the log-posterior
new_lp <- log.posterior(new_sample[1], new_sample[2], new_sample[3],
                        x, y, a = 1, b = 1.25)
# Add to the "dataset" of hyperparam / log-posterior pairs
hparam_dataset <- rbind(hparam_dataset, new_sample)
hparam_post <- c(hparam_post, new_lp)
```

```r
# Refit the model
my_model <- GP_fit(exp(hparam_dataset)/2, hparam_post)

# Observe the value of the "best" point after two iterations
exp(new_sample)
```

```
## [1] 1.98853950 0.58885996 0.07675146
```

We now define a function to run this algorithm for a given number of iterations.

```r
surrogate.opt <- function(n_guesses, num_hparam, x, y, eps,
                          prior_a, prior_b, n_candidates, n_iter){
  hparam_dataset <- log(randomLHS(n_guesses, num_hparam)*2)
  hparam_post <- apply(
    hparam_dataset, 1, function(z) log.posterior(
      z[1], z[2], z[3], x, y, a = prior_a, b = prior_b
  ))

  # Fit a Gaussian process to the "data" (rescaled to be in [0,1])
  my_model <- GP_fit(exp(hparam_dataset)/2, hparam_post)

  # Pick the "best" set of hyperparameters from the initial dataset
  hparam_best <- min(-hparam_post)

  # Generate a set of "candidate" hyperparameters
  hparam_candidates <- log(randomLHS(n_candidates, num_hparam)*2)

  # Prepare to iterate the optimization routine
  opt_results <- matrix(NA, nrow = n_iter, ncol = 3)
  for (i in 1:n_iter){
    # Run the algorithm
    new_sample_ind <- unname(
      BO.iter(hparam_candidates, my_model, eps, hparam_best)
    )
    new_sample <- hparam_candidates[new_sample_ind, ]
    hparam_candidates <- hparam_candidates[
      (1:nrow(hparam_candidates) != new_sample_ind), ]

    # Evaluate the log-posterior
    new_lp <- log.posterior(new_sample[1], new_sample[2],
                            new_sample[3], x, y, a = prior_a,
                            b = prior_b)
    # Add to the "dataset" of hyperparam / log-posterior pairs
    hparam_dataset <- rbind(hparam_dataset, new_sample)
    hparam_post <- c(hparam_post, new_lp)

    # Refit the model
    my_model <- GP_fit(exp(hparam_dataset)/2, hparam_post)
  }
  # Return the latest optimum
  return(hparam_dataset[which(-hparam_post == min(-hparam_post)),])
}
```

Now, we can run this optimization program for 50 iterations

```r
best_params_2c <- exp(surrogate.opt(
  n_guesses = 100, num_hparam = 3, x = x, y = y,
  eps = 0.01, prior_a = 1, prior_b = 1.25,
  n_candidates = 500, n_iter = 50
))
```

And check the results to see if it's close to the "right" answers ($\sigma_f = 1$, $\ell = 0.5$ and $\sigma_y = 0.01$)

```r
best_params_2c
```

```
## [1] 0.66738903 0.80642022 0.01660696
```

3. GP Evaluation

We rely on a combination of simulation studies and real data to evaluate our approach(es) to selecting hyperparameters for Gaussian Process regression. First, we define some helper functions that use the above optimization procedures and produce hyperparameter results or samples from a conditional distribution of a holdout set.

This function runs SGD to optimize hyperparameters and then draws a number of samples from the conditional distribution of a holdout set.

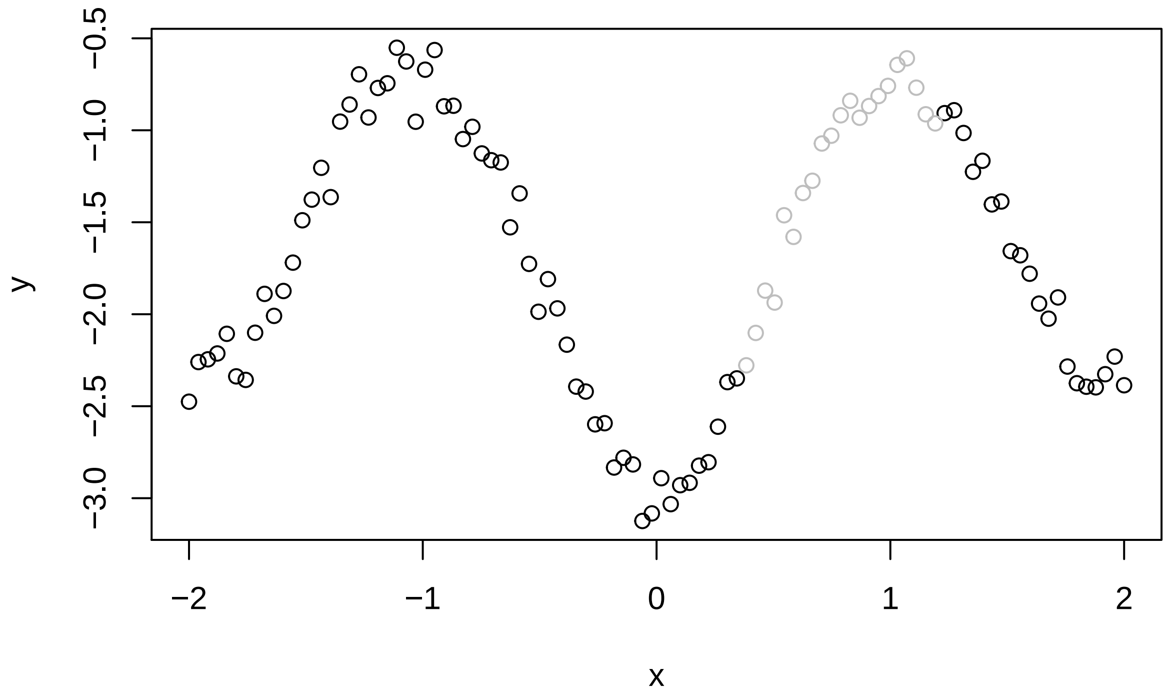
```r
fit.interp <- function(
    x, y, pred_inds, n_interp = 50, batch_size = 20, n_iter = 1000,
    stop_iter = 10, convergence_limit = 0.0001, n_restarts = 5, step_size = 0.001,
    momentum_gamma = 0.9,  opt_type = "likelihood", prior_parameter_a = 1,
    prior_parameter_b = 1.25
){
    # Split into a holdout sample and plot
    x_pred <- x[pred_inds]
    y_pred <- y[pred_inds]
    x_est <- x[!(1:n %in% pred_inds)]
    y_est <- y[!(1:n %in% pred_inds)]

    # Estimate hyperparameters for a GP with squared exponential kernel
    # on the `est` data
    opt_results <- sgd.optimize(
        x_est, y_est, batch_size = batch_size, n_iter = n_iter, stop_iter = stop_iter,
        convergence_limit = convergence_limit, n_restarts = n_restarts, step_size = step_size,
        momentum_gamma = momentum_gamma, opt_type = opt_type,
        prior_parameter_a = prior_parameter_a, prior_parameter_b = prior_parameter_b
    )
    result_mat <- opt_results[[1]]
    iter_mat <- opt_results[[2]]
    num_plot_points <- nrow(iter_mat)
    n_restarts <- nrow(result_mat)

    # Store the "final" hyperparameters in a vector
    gp_hyperparams <- exp(result_mat[nrow(result_mat),])
    sigma.f.hat <- gp_hyperparams[1]
    ell.hat <- gp_hyperparams[2]
    sigma.y.hat <- gp_hyperparams[3]

    # Interpolate the missing data
    gp_mu <- gp_pred_mean(x_est, x_pred, y_est, sigma.f.hat, ell.hat, sigma.y.hat)
    gp_sigma <- gp_pred_var(x_est, x_pred, y_est, sigma.f.hat, ell.hat, sigma.y.hat)
    Y_interp <- mvrnorm(n = n_interp, mu = gp_mu, Sigma = gp_sigma)
    return(Y_interp)
}
```

3.a. Simulated Data

We are interested in the effect of various datasets on hyperparameter selection and the resulting interpolation / prediction task.

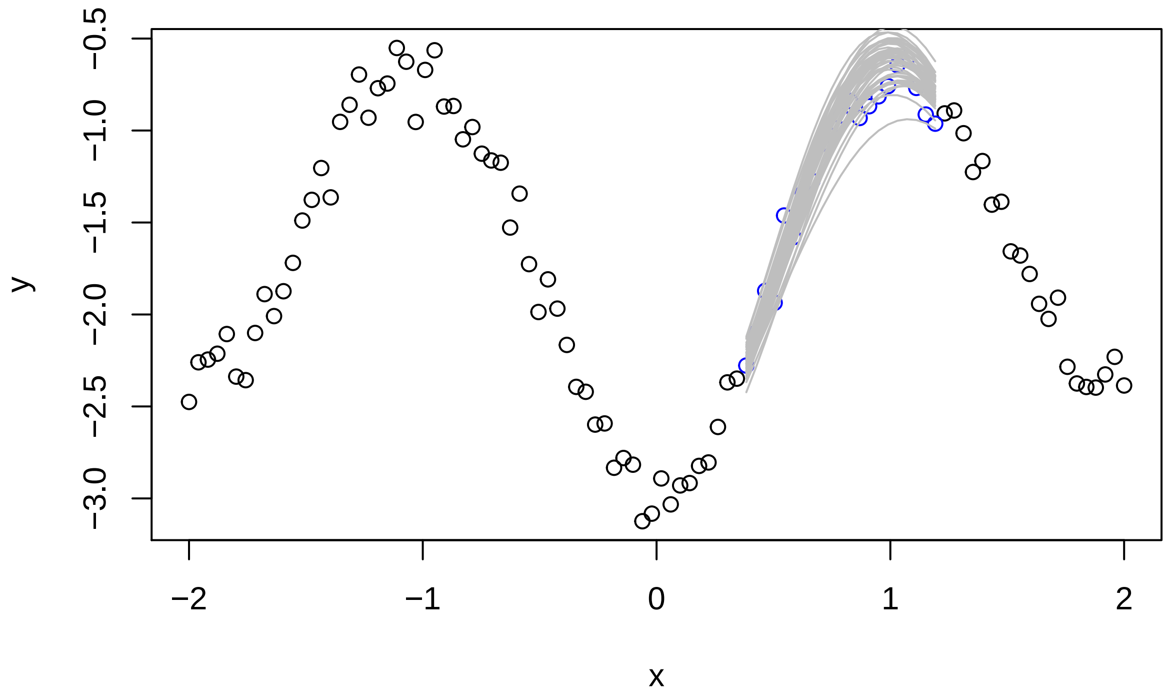
3.a.1. Smooth function, low noise, low n

```
# Generate some independent values
n <- 100
x <- seq(-2, 2, length.out = n)

# Generate y = -2 + 0.3*abs(x) + cos(pi x) + eps
eps <- rnorm(n, mean = 0, sd = 0.1)
y <- -2 + 0.3*abs(x/1) - cos(pi*x) + eps
pred_inds <- (floor(n*0.6):floor(n*0.8))

# Plot the data
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "gray")
```



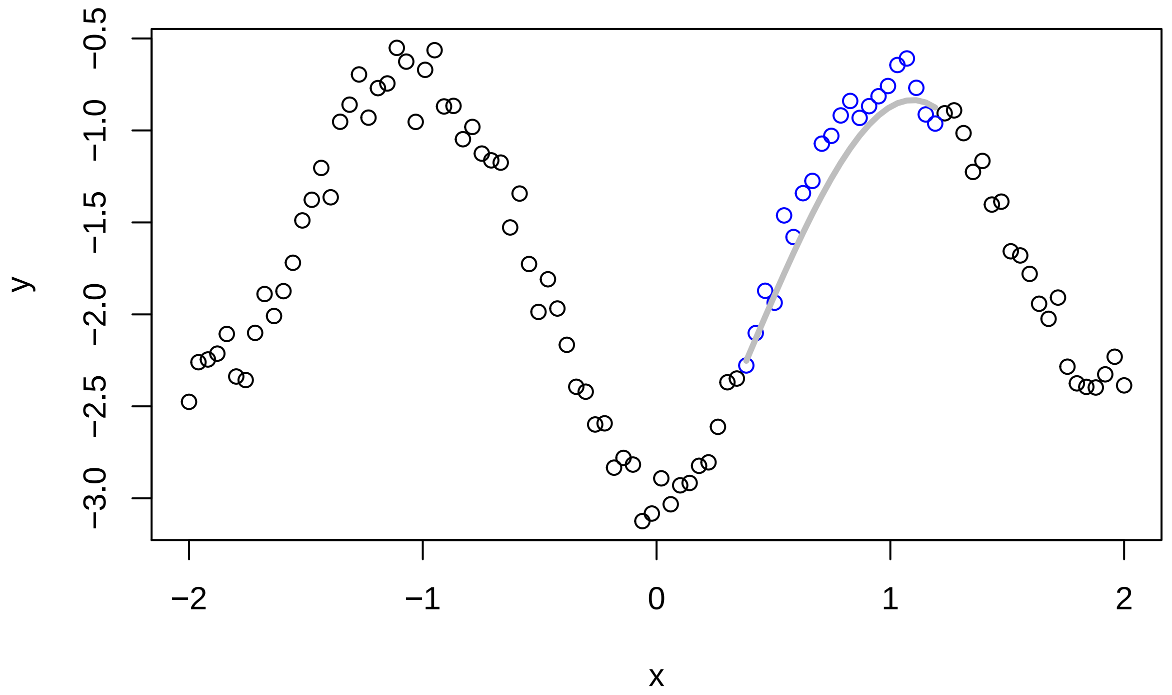
Optimize the Gaussian process hyperparameters and plot the interpolation formed by the results.

```
# Optimize the hyperparameters on the observed data and then
# use those hyperparameters to interpolate the holdout data with a GP
Y.interp <- fit.interp(
    x, y, pred_inds = pred_inds, n_interp = 50, batch_size = 20, n_iter = 1000,
    stop_iter = 10, convergence_limit = 0.0001, n_restarts = 1, step_size = 0.001,
    momentum_gamma = 0.9,  opt_type = "posterior", prior_parameter_a = 1,
    prior_parameter_b = 1.25
)

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
for (i in 1:50){
    lines(x[pred_inds], Y.interp[i,], col = "gray")
}
```



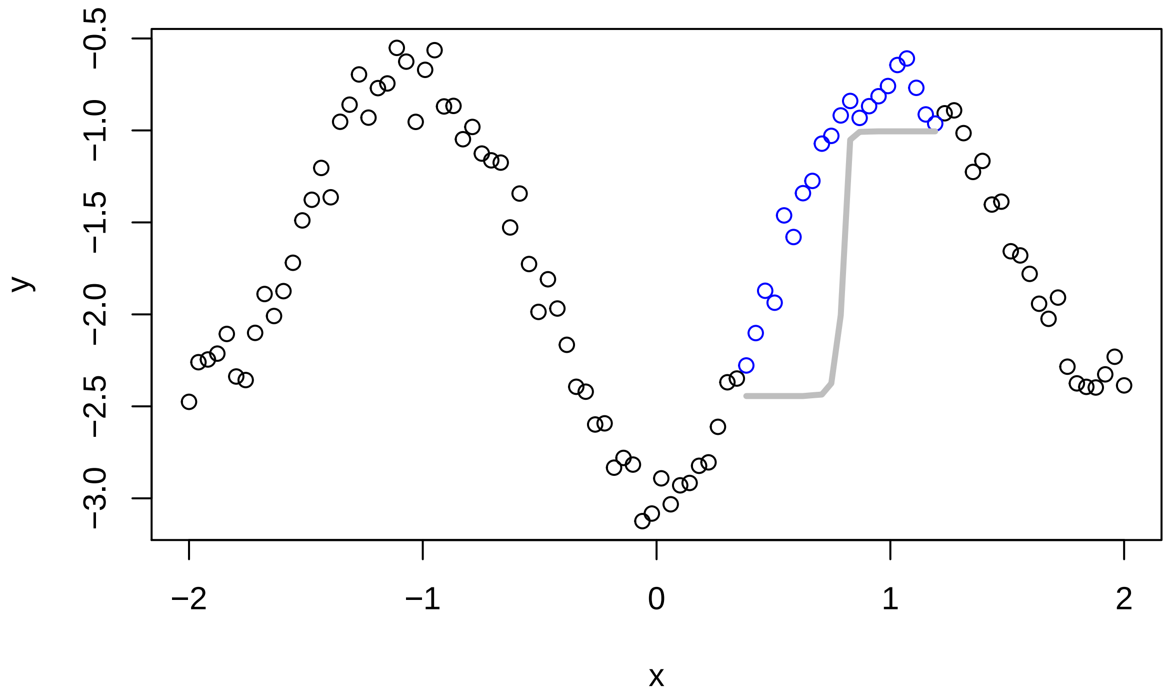
Compare this to smoothing spline interpolation

```
# Fit and predict the model
spline.model <- smooth.spline(
  x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)]
)
y_pred <- predict(spline.model, x[pred_inds])$y

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y_pred, col = "gray", lwd = 3)
```



Also compare to random forest interpolation.

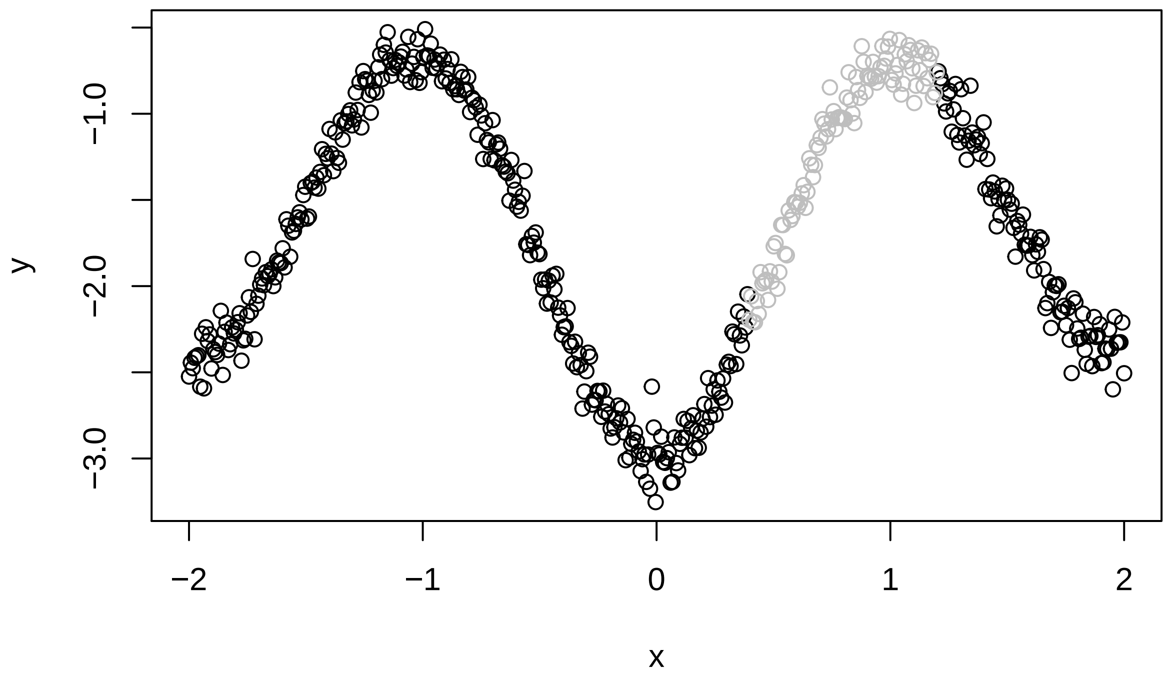
```r
# Fit and predict the model
rf.model <- randomForest(as.matrix(x[!(1:n %in% pred_inds)]), y[!(1:n %in% pred_inds)])
y.interp <- predict(rf.model, as.matrix(x[pred_inds]))

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y.interp, col = "gray", lwd = 3)
```



In this case, both the spline and GP interpolations are quite close, and the random forest interpolation is not great.

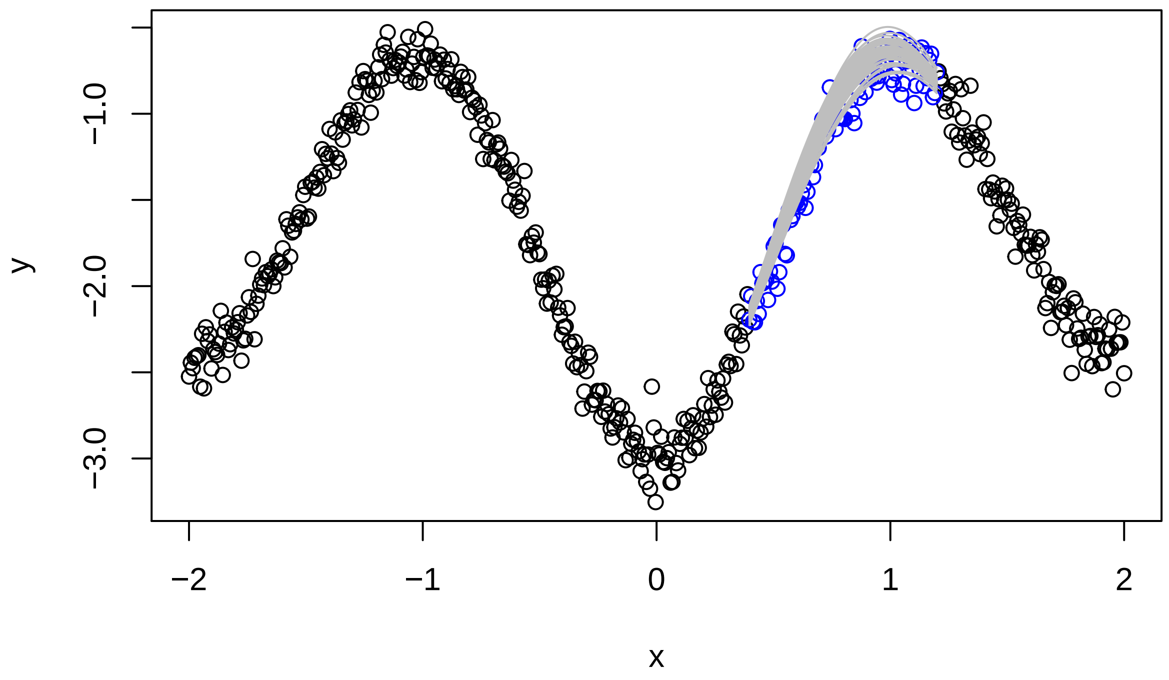
3.a.2. Smooth function, low noise, moderate n

```
# Generate some independent values
n <- 500
x <- seq(-2, 2, length.out = n)

# Generate y = -2 + 0.3*abs(x) + cos(pi x) + eps
eps <- rnorm(n, mean = 0, sd = 0.1)
y <- -2 + 0.3*abs(x/1) - cos(pi*x) + eps
# pred_inds <- c(101:125,201:225,301:325,401:425)
pred_inds <- (floor(n*0.6):floor(n*0.8))

# Plot the data
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "gray")
```



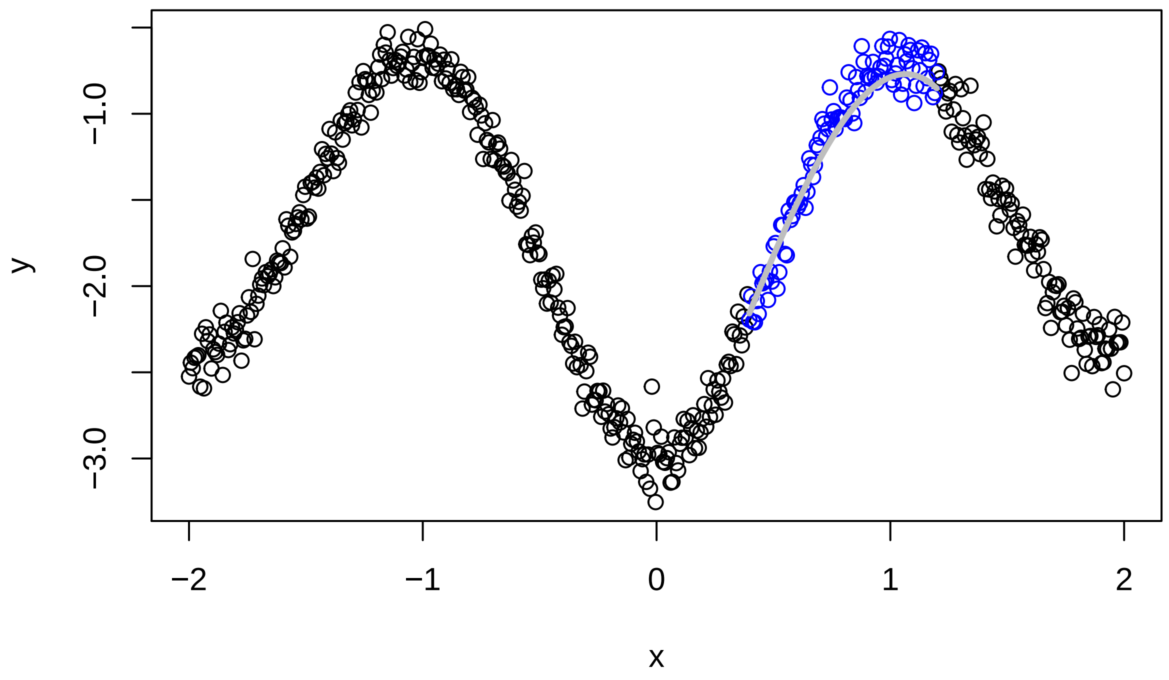
Optimize the Gaussian process hyperparameters and plot the interpolation formed by the results.

```
# Optimize the hyperparameters on the observed data and then
# use those hyperparameters to interpolate the holdout data with a GP
Y.interp <- fit.interp(
    x, y, pred_inds = pred_inds, n_interp = 50, batch_size = 20, n_iter = 1000,
    stop_iter = 10, convergence_limit = 0.0001, n_restarts = 1, step_size = 0.001,
    momentum_gamma = 0.9,  opt_type = "posterior", prior_parameter_a = 1,
    prior_parameter_b = 1.25
)

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
for (i in 1:50){
    lines(x[pred_inds], Y.interp[i,], col = "gray")
}
```



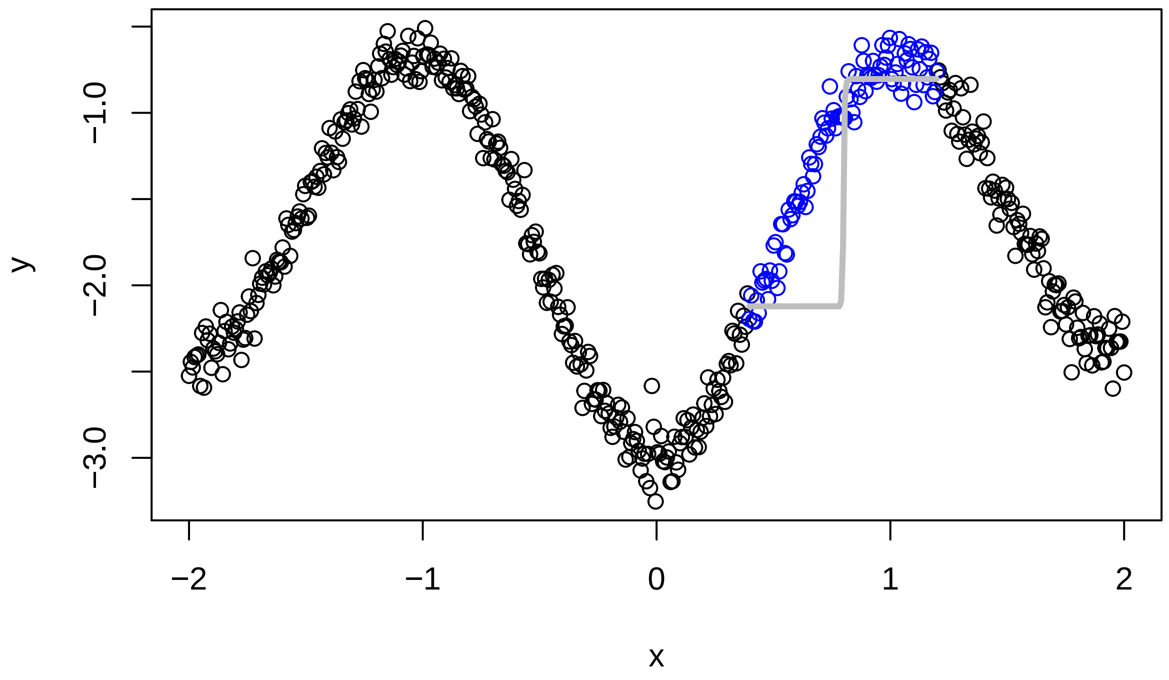
Compare this to smoothing spline interpolation

```r
# Fit and predict the model
spline.model <- smooth.spline(
  x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)]
)
y_pred <- predict(spline.model, x[pred_inds])$y

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y_pred, col = "gray", lwd = 3)
```



Also compare to random forest interpolation.

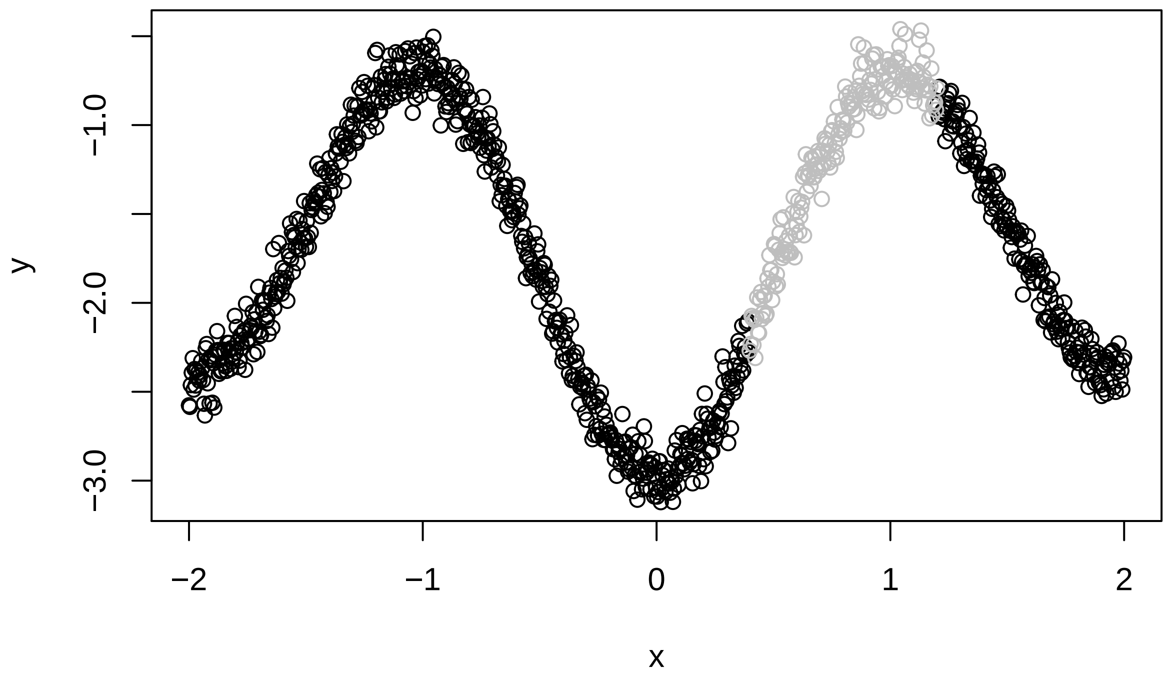
```
# Fit and predict the model
rf.model <- randomForest(as.matrix(x[!(1:n %in% pred_inds)]), y[!(1:n %in% pred_inds)])
y.interp <- predict(rf.model, as.matrix(x[pred_inds]))

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y.interp, col = "gray", lwd = 3)
```



In this case, the GP interpolation is quite good as is the spline interpolation.

3.a.3. Smooth function, low noise, high n

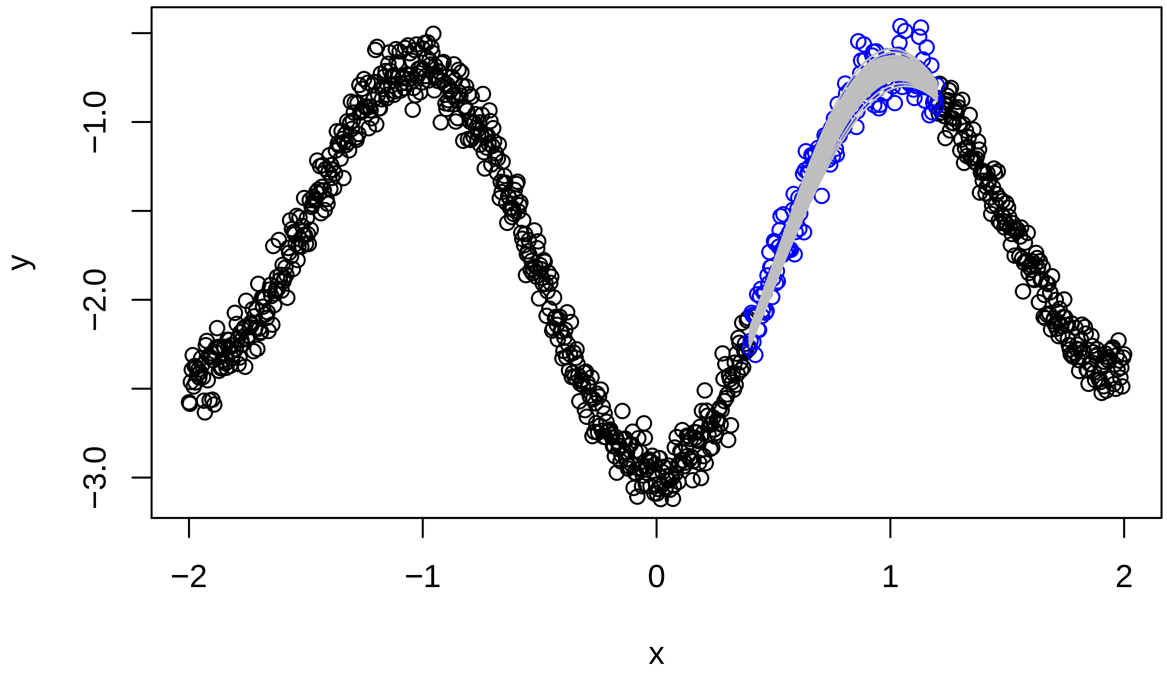
```r
# Generate some independent values
n <- 1000
x <- seq(-2, 2, length.out = n)

# Generate y = -2 + 0.3*abs(x) + cos(pi x) + eps
eps <- rnorm(n, mean = 0, sd = 0.1)
y <- -2 + 0.3*abs(x/1) - cos(pi*x) + eps
# pred_inds <- c(101:125,201:225,301:325,401:425)
pred_inds <- (floor(n*0.6):floor(n*0.8))

# Plot the data
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "gray")
```



Optimize the Gaussian process hyperparameters and plot the interpolation formed by the results.

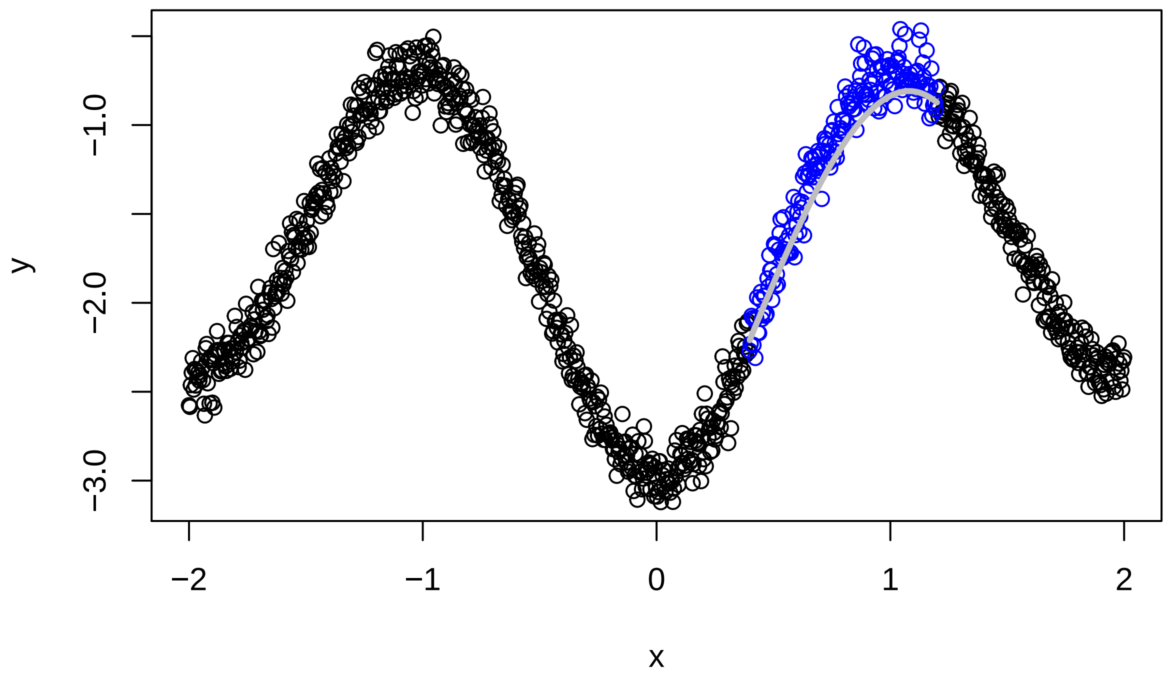
```r
# Optimize the hyperparameters on the observed data and then
# use those hyperparameters to interpolate the holdout data with a GP
Y.interp <- fit.interp(
    x, y, pred_inds = pred_inds, n_interp = 50, batch_size = 20, n_iter = 1000,
    stop_iter = 10, convergence_limit = 0.0001, n_restarts = 1, step_size = 0.001,
    momentum_gamma = 0.9,  opt_type = "posterior", prior_parameter_a = 1,
    prior_parameter_b = 1.25
)

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
```

```
points(x[pred_inds], y[pred_inds], col = "blue")
for (i in 1:50){
    lines(x[pred_inds], Y.interp[i,], col = "gray")
}
```



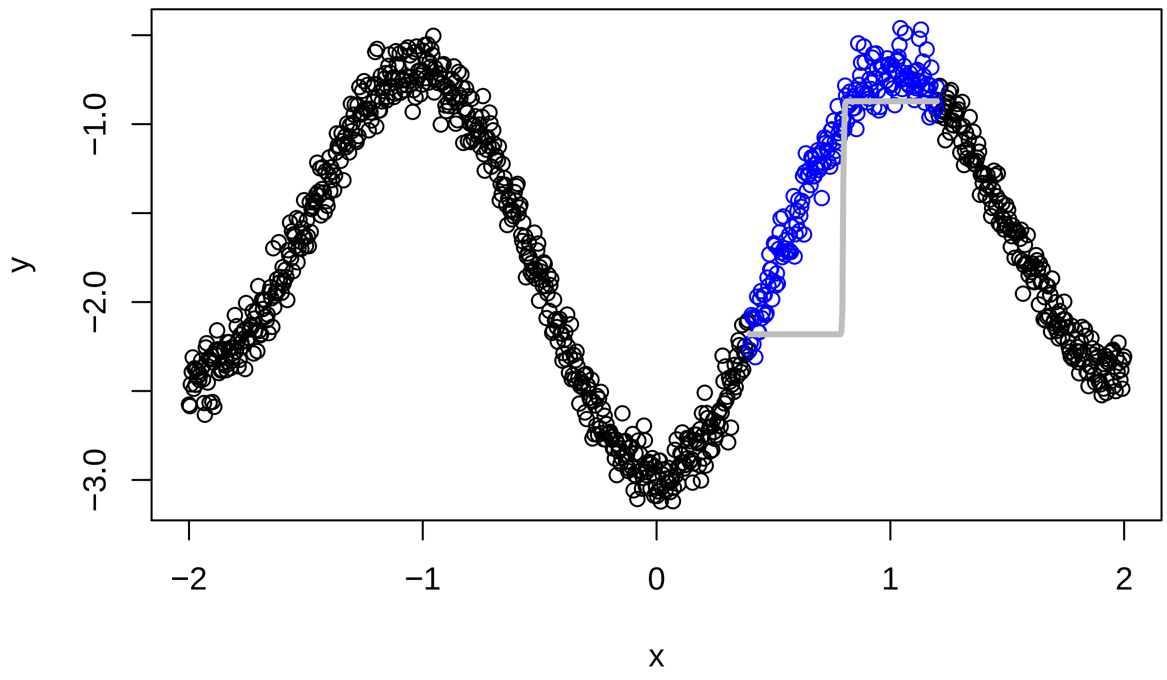
Compare this to smoothing spline interpolation

```
# Fit and predict the model
spline.model <- smooth.spline(
  x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)]
)
y_pred <- predict(spline.model, x[pred_inds])$y

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y_pred, col = "gray", lwd = 3)
```



Also compare to random forest interpolation.

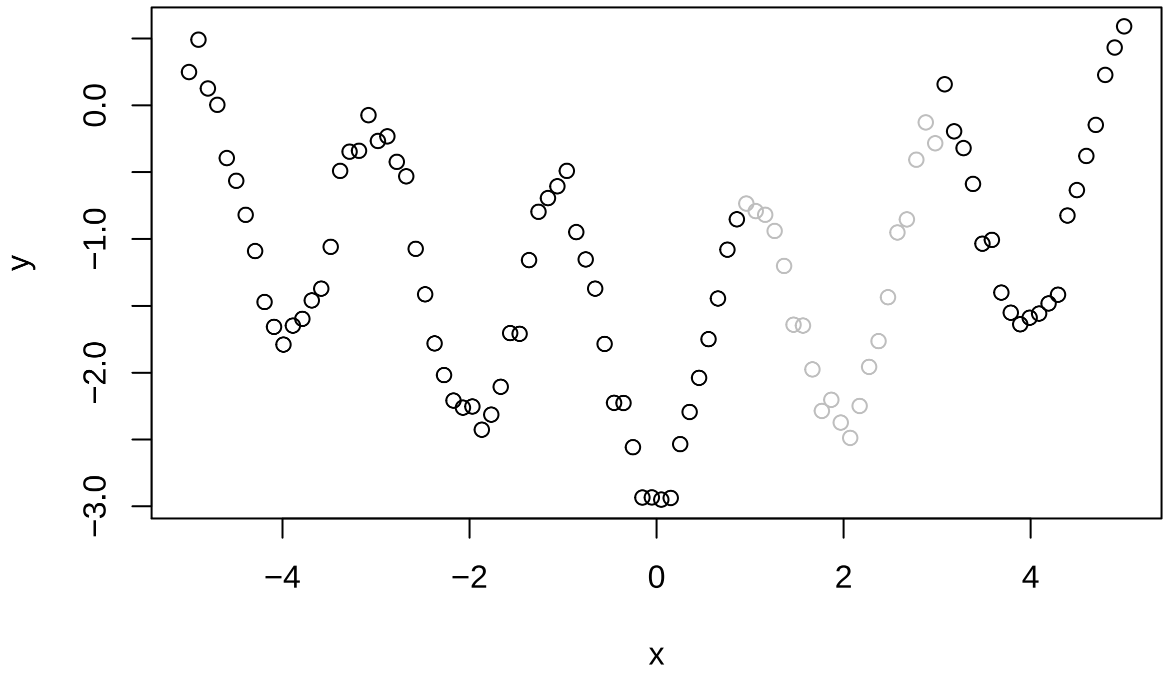
```r
# Fit and predict the model
rf.model <- randomForest(as.matrix(x[!(1:n %in% pred_inds)]), y[!(1:n %in% pred_inds)])
y.interp <- predict(rf.model, as.matrix(x[pred_inds]))

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y.interp, col = "gray", lwd = 3)
```



As above, the GP interpolation is quite good as is the spline interpolation.

3.a.4. Highly periodic function, low noise, low n

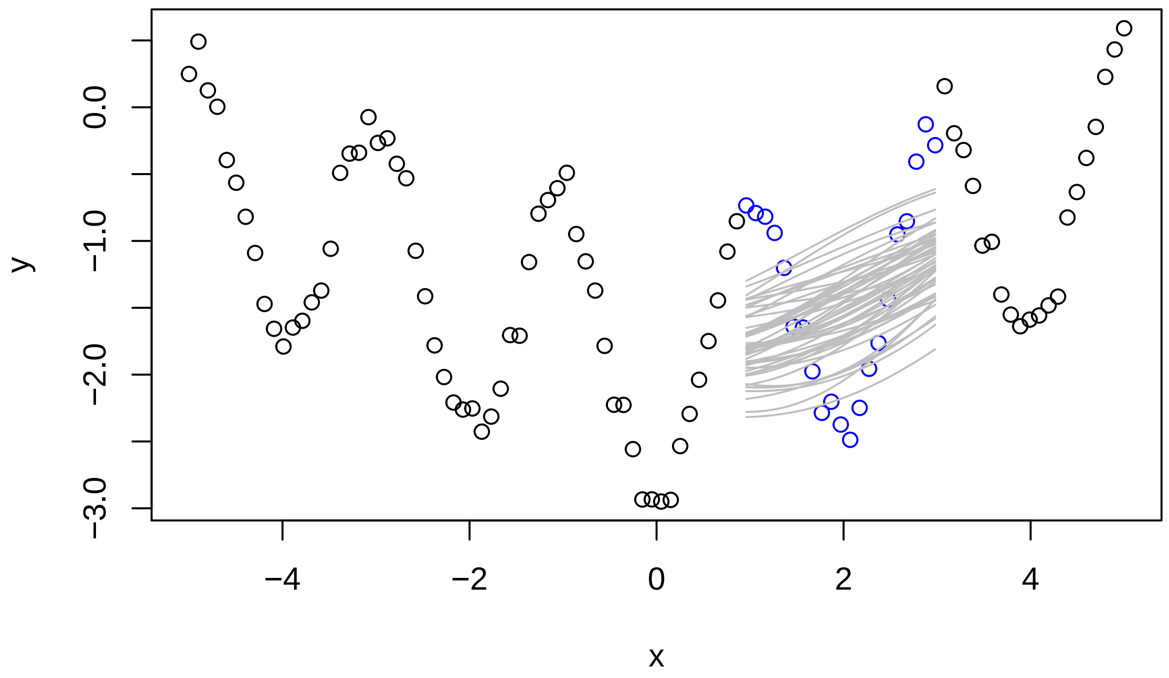
```r
# Generate some independent values
n <- 100
x <- seq(-5, 5, length.out = n)

# Generate y = -2 + 0.3*abs(x) + cos(pi x) + eps
eps <- rnorm(n, mean = 0, sd = 0.1)
y <- -2 + 0.3*abs(x/1) - cos(pi*x) + eps
pred_inds <- (floor(n*0.6):floor(n*0.8))

# Plot the data
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "gray")
```



Optimize the Gaussian process hyperparameters and plot the interpolation formed by the results.

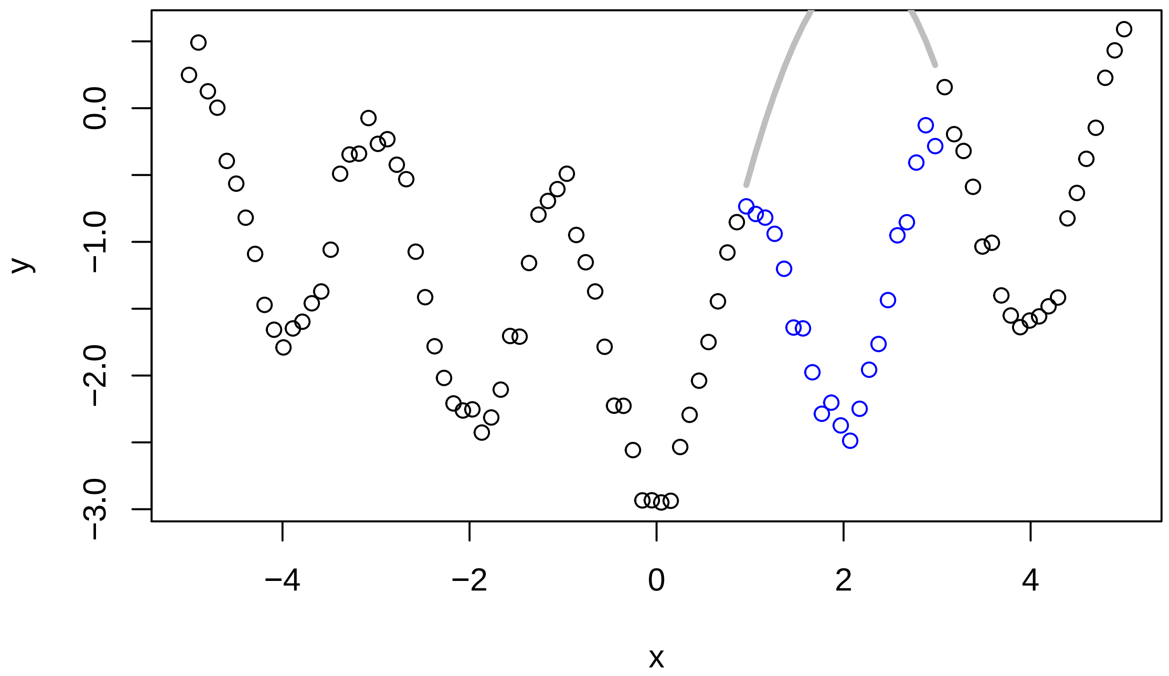
```r
# Optimize the hyperparameters on the observed data and then
# use those hyperparameters to interpolate the holdout data with a GP
Y.interp <- fit.interp(
    x, y, pred_inds = pred_inds, n_interp = 50, batch_size = 20, n_iter = 1000,
    stop_iter = 10, convergence_limit = 0.0001, n_restarts = 1, step_size = 0.001,
    momentum_gamma = 0.9,  opt_type = "posterior", prior_parameter_a = 1,
    prior_parameter_b = 1.25
)

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
```

```r
for (i in 1:50){
    lines(x[pred_inds], Y.interp[i,], col = "gray")
}
```



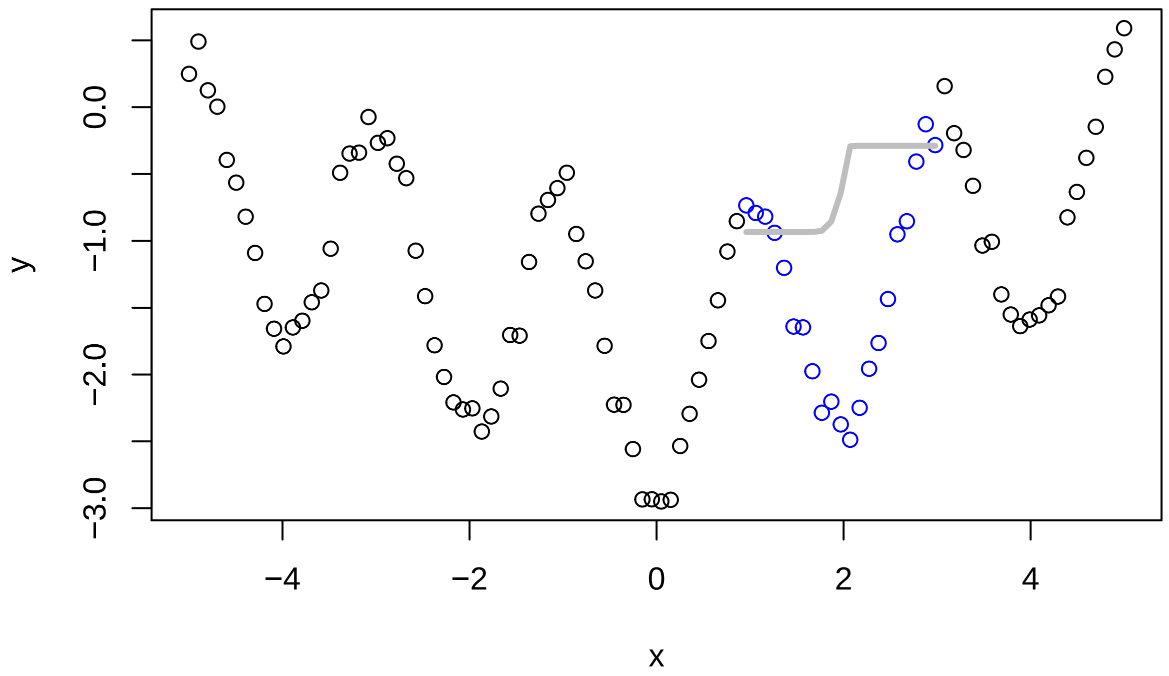
Compare this to smoothing spline interpolation

```r
# Fit and predict the model
spline.model <- smooth.spline(
  x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)]
)
y_pred <- predict(spline.model, x[pred_inds])$y

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y_pred, col = "gray", lwd = 3)
```



Also compare to random forest interpolation.

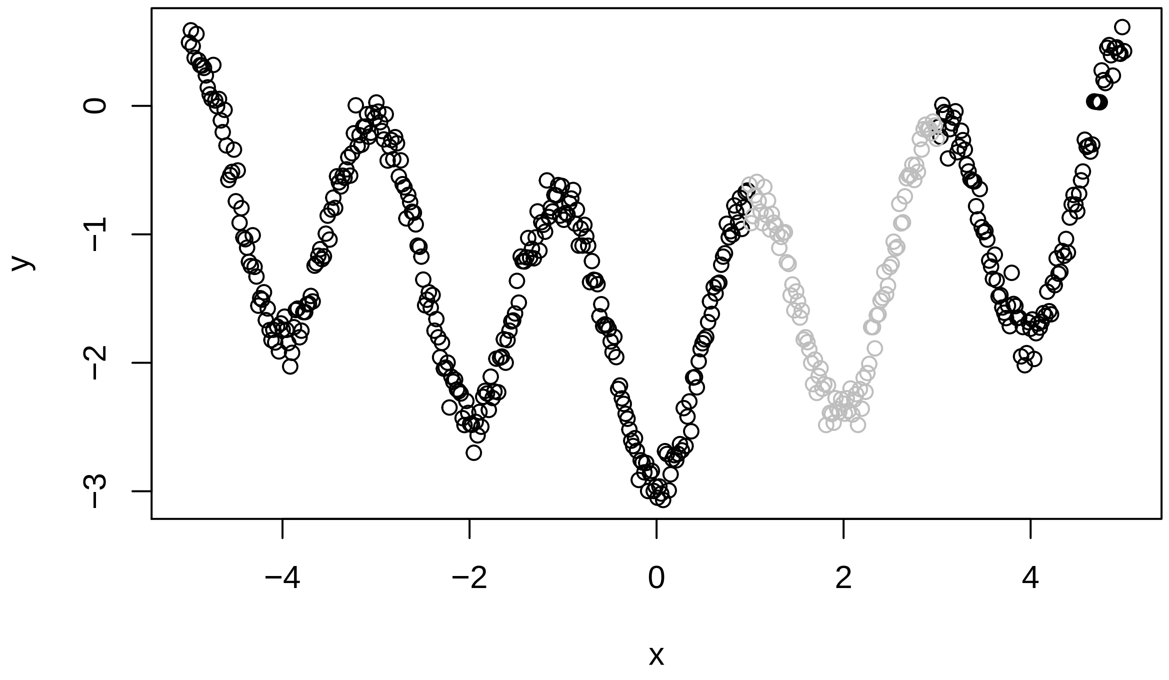
```
# Fit and predict the model
rf.model <- randomForest(as.matrix(x[!(1:n %in% pred_inds)]), y[!(1:n %in% pred_inds)])
y.interp <- predict(rf.model, as.matrix(x[pred_inds]))

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y.interp, col = "gray", lwd = 3)
```



As above, the GP interpolation does not appear to work well, at least in terms of capturing the uncertainty about the true function. The spline interpolation misses the true functional form completely and the random forest is as usual a relatively uncomplicated step function.

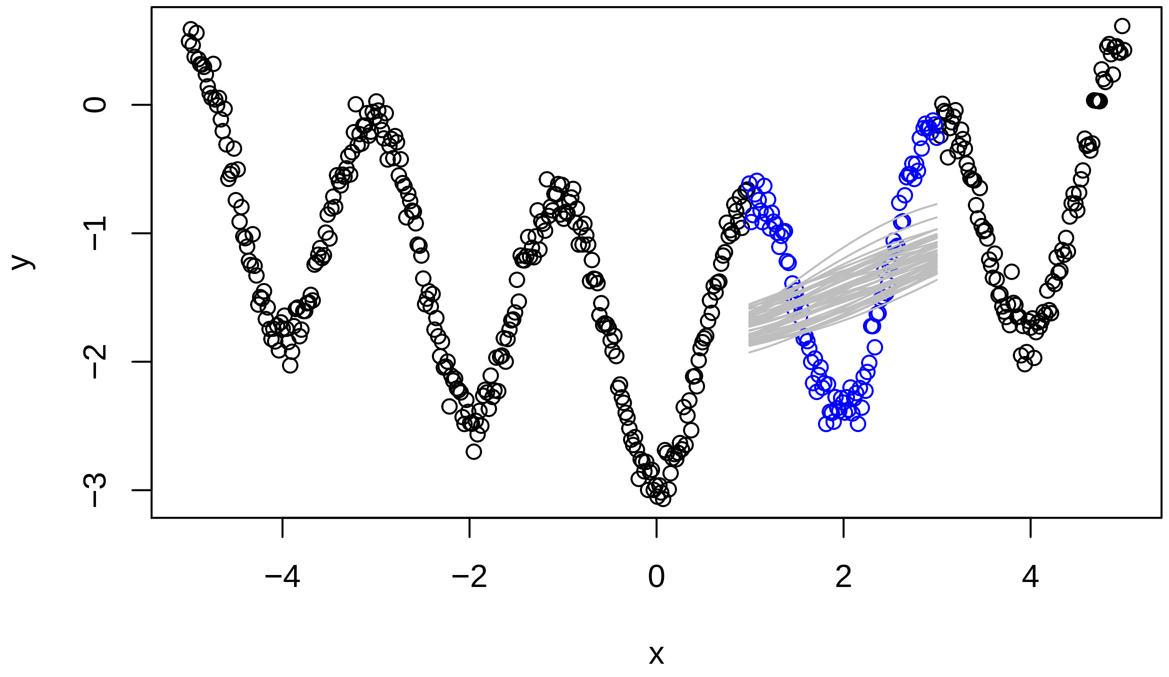
3.a.5. Highly periodic function, low noise, moderate n

```
# Generate some independent values
n <- 500
x <- seq(-5, 5, length.out = n)

# Generate y = -2 + 0.3*abs(x) + cos(pi x) + eps
eps <- rnorm(n, mean = 0, sd = 0.1)
y <- -2 + 0.3*abs(x/1) - cos(pi*x) + eps
pred_inds <- (floor(n*0.6):floor(n*0.8))

# Plot the data
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "gray")
```



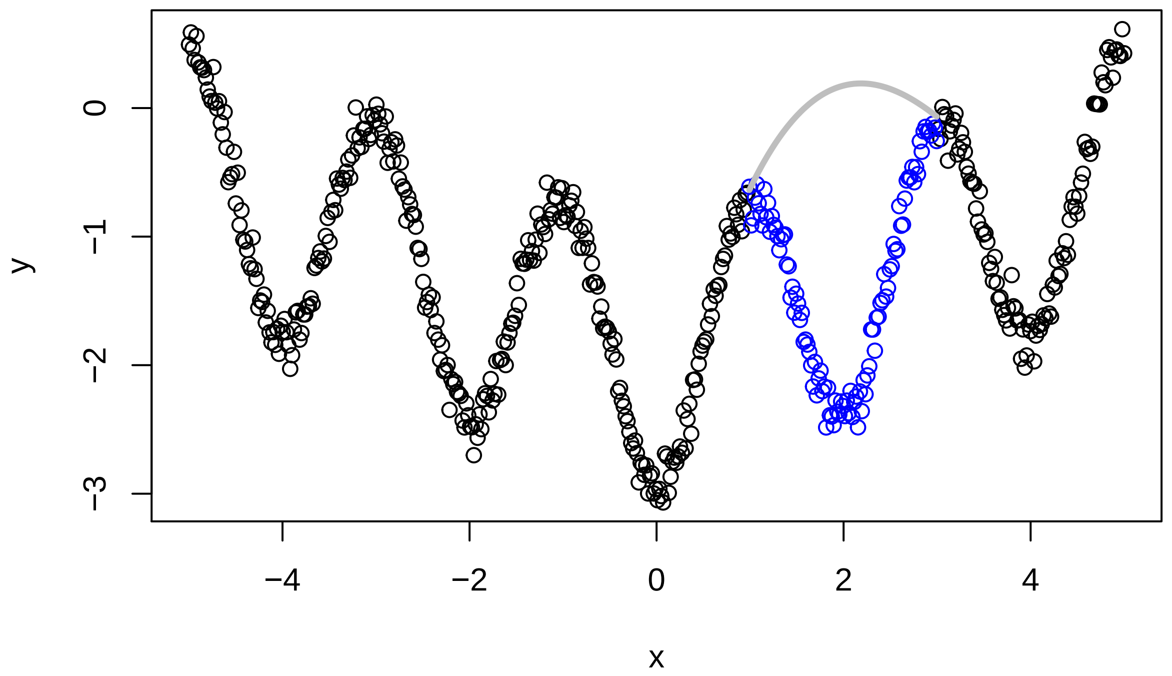
Optimize the Gaussian process hyperparameters and plot the interpolation formed by the results.

```
# Optimize the hyperparameters on the observed data and then
# use those hyperparameters to interpolate the holdout data with a GP
Y.interp <- fit.interp(
    x, y, pred_inds = pred_inds, n_interp = 50, batch_size = 20, n_iter = 1000,
    stop_iter = 10, convergence_limit = 0.0001, n_restarts = 1, step_size = 0.001,
    momentum_gamma = 0.9,  opt_type = "posterior", prior_parameter_a = 1,
    prior_parameter_b = 1.25
)

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
for (i in 1:50){
    lines(x[pred_inds], Y.interp[i,], col = "gray")
}
```



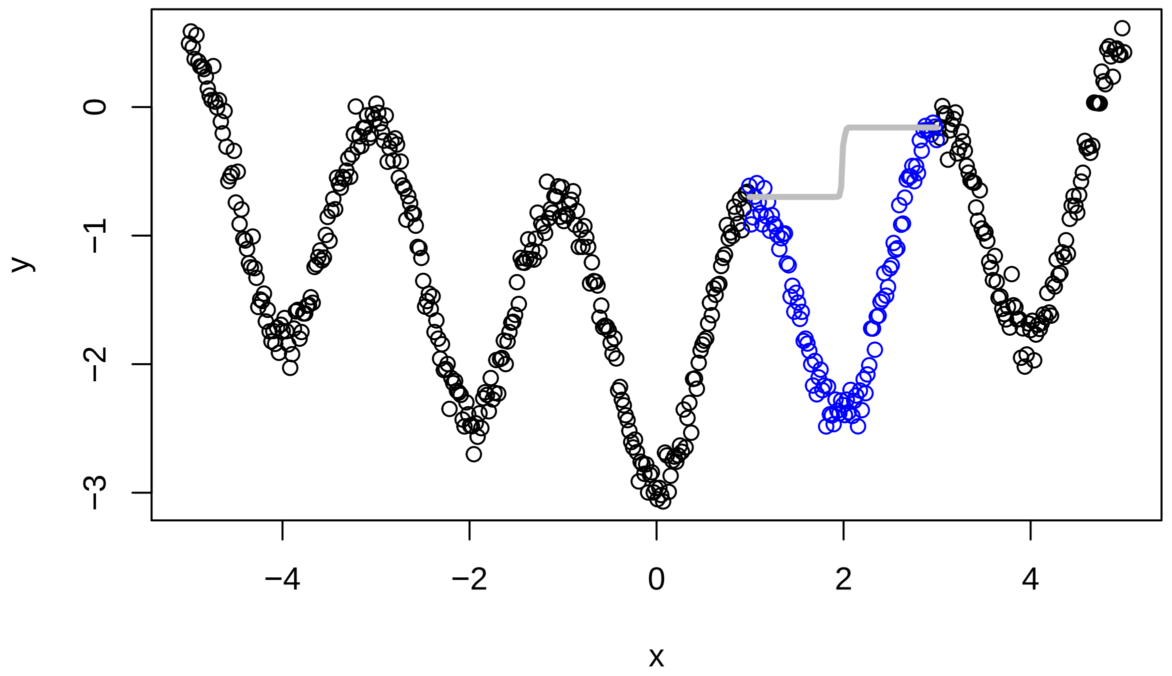
Compare this to smoothing spline interpolation

```
# Fit and predict the model
spline.model <- smooth.spline(
  x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)]
)
y_pred <- predict(spline.model, x[pred_inds])$y

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y_pred, col = "gray", lwd = 3)
```



Also compare to random forest interpolation.

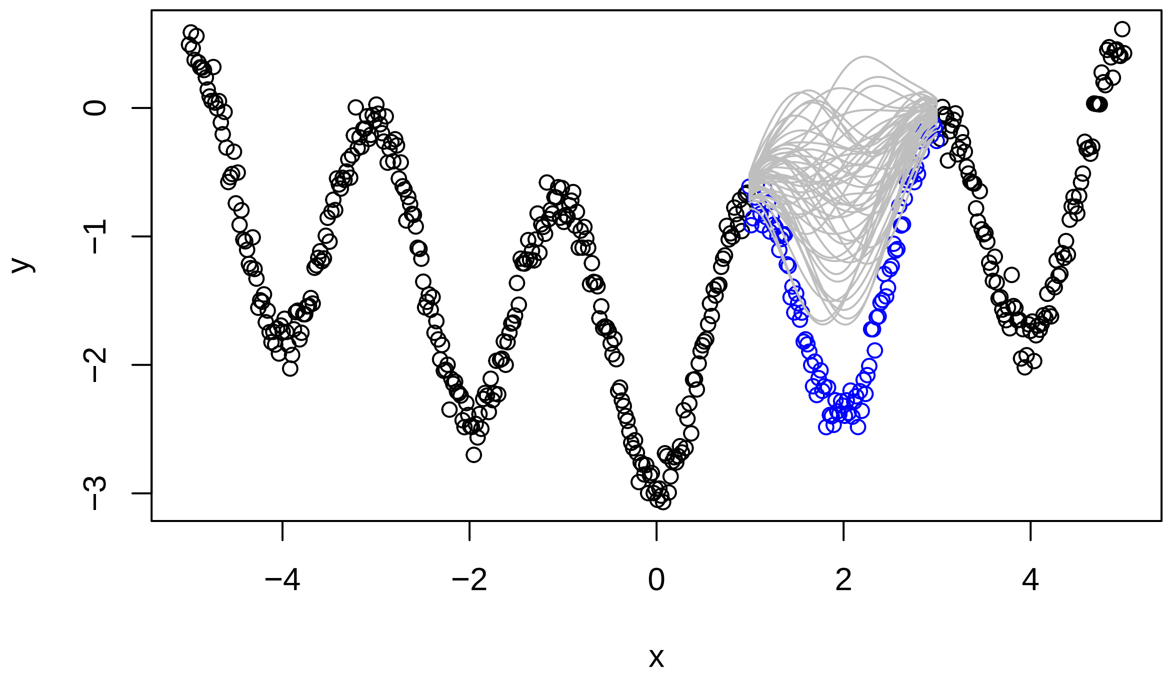
```r
# Fit and predict the model
rf.model <- randomForest(as.matrix(x[!(1:n %in% pred_inds)]), y[!(1:n %in% pred_inds)])
y.interp <- predict(rf.model, as.matrix(x[pred_inds]))

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y.interp, col = "gray", lwd = 3)
```



For the GP, even with more data, the resulting estimated hyperparameters appear to generate poor interpolations. It's possible that the periodicity of the true function is high enough that the optimization process views it as noise and fits a less-complex shallow curve through the middle of the true dataset.

Now, we can consider whether the Bayesian optimization procedure will yield a drastically different set of hyperparameters and resulting interpolation set.

```r
# First, optimize the hyperparameters using BO
best_params_3a5 <- exp(surrogate.opt(
  n_guesses = 50, num_hparam = 3,
  x = x[!(1:n %in% pred_inds)], y = y[!(1:n %in% pred_inds)],
  eps = 0.01, prior_a = 1, prior_b = 1.25,
  n_candidates = 200, n_iter = 50
))

# Then draw some interpolations using those parameters
n_interp <- 50
sigma.f.hat.3a5 <- best_params_3a5[1]
ell.hat.3a5 <- best_params_3a5[2]
sigma.y.hat.3a5 <- best_params_3a5[3]
gp_mu_3a5 <- gp_pred_mean(
  x[!(1:n %in% pred_inds)], x[pred_inds], y[!(1:n %in% pred_inds)],
  sigma.f.hat.3a5, ell.hat.3a5, sigma.y.hat.3a5
)
gp_sigma_3a5 <- gp_pred_var(
  x[!(1:n %in% pred_inds)], x[pred_inds], y[!(1:n %in% pred_inds)],
  sigma.f.hat.3a5, ell.hat.3a5, sigma.y.hat.3a5
)
Y_interp_3a5 <- mvrnorm(n = n_interp, mu = gp_mu_3a5, Sigma = gp_sigma_3a5)

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
for (i in 1:n_interp){
    lines(x[pred_inds], Y_interp_3a5[i,], col = "gray")
}
```

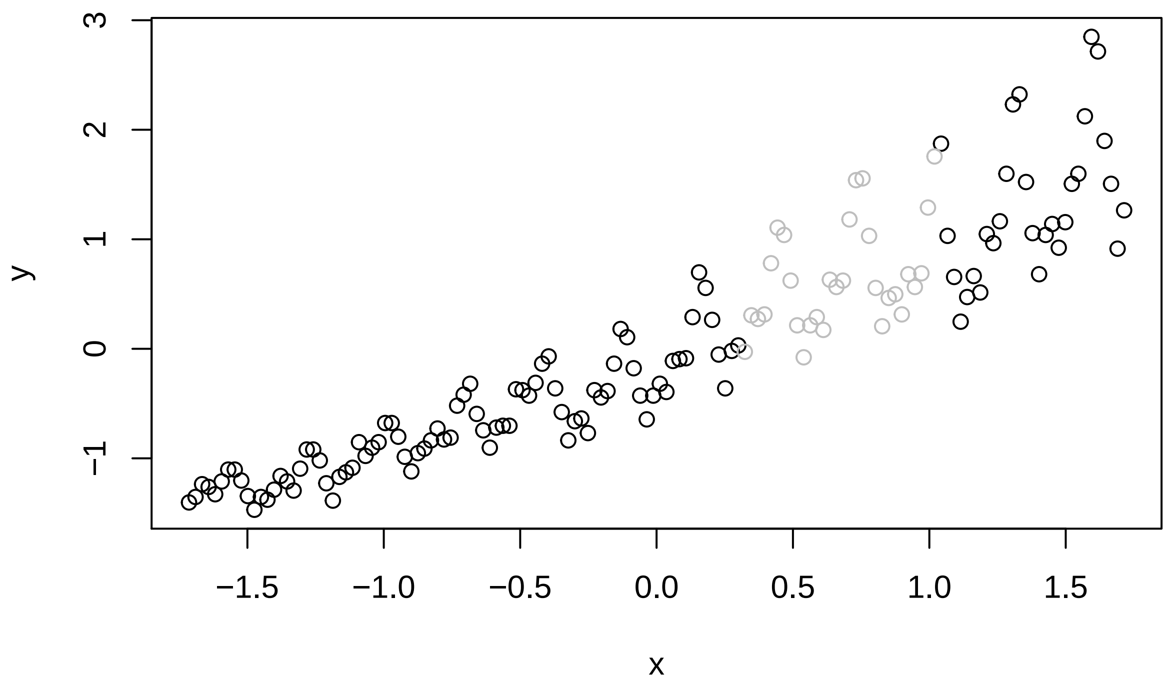


This looks better, though it was rather computationally expensive!

3.b. Real Data

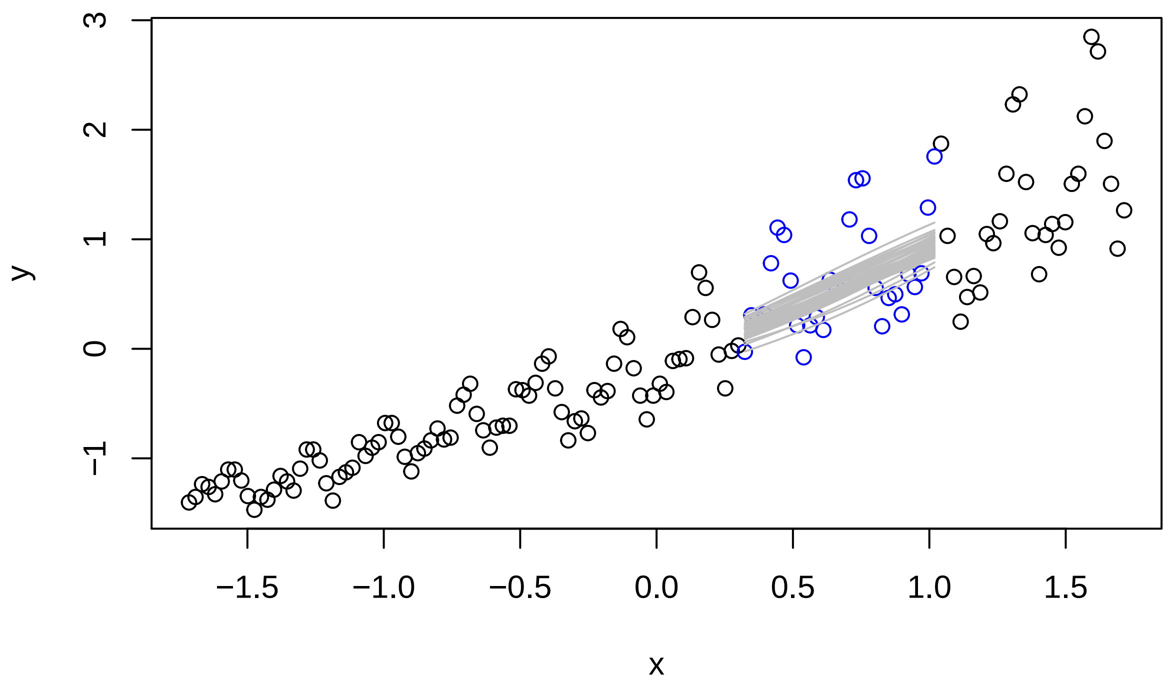
We also test these methods on the `AirPassengers` time series dataset provided by R.

```r
# Load the AirPassengers data
y <- (AirPassengers - mean(AirPassengers))/sd(AirPassengers)
x <- (1:length(y) - mean(1:length(y)))/sd(1:length(y))
n <- length(y)
pred_inds <- (floor(n*0.6):floor(n*0.8))

# Plot the data
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)],
     xlab = "x", ylab = "y", xlim = c(min(x), max(x)),
     ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "gray")
```



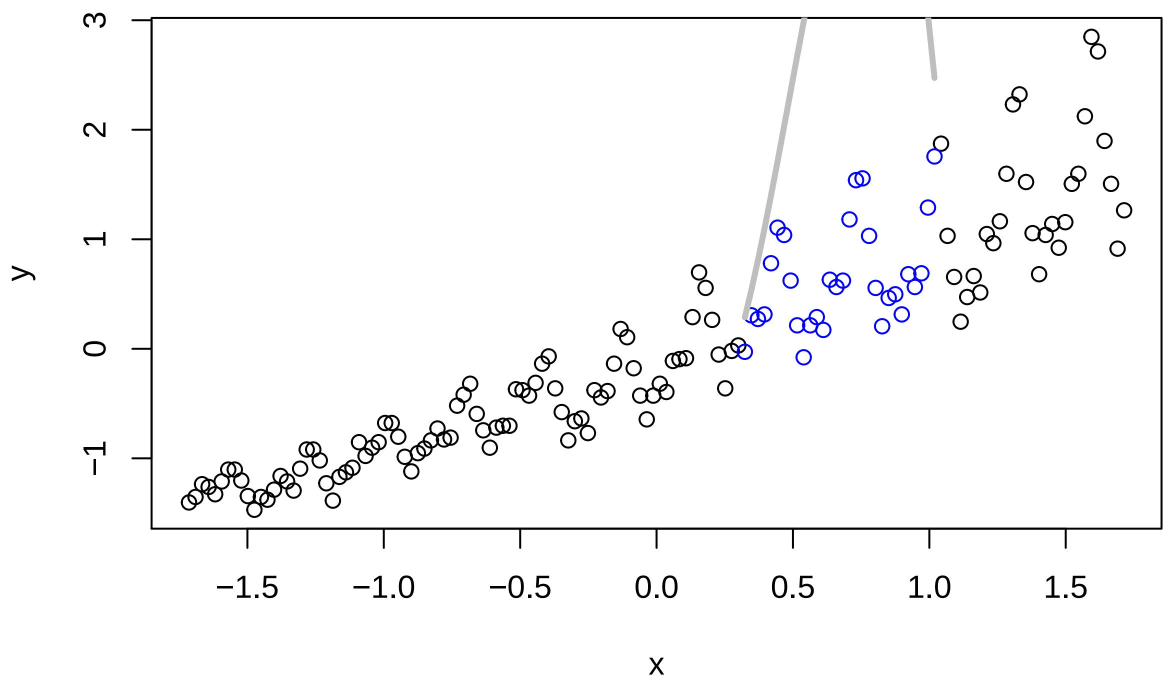
Optimize the Gaussian process hyperparameters and plot the interpolation formed by the results.

```r
# Optimize the hyperparameters on the observed data and then
# use those hyperparameters to interpolate the holdout data with a GP
Y.interp <- fit.interp(
    x, y, pred_inds = pred_inds, n_interp = 50, batch_size = 20, n_iter = 1000,
    stop_iter = 10, convergence_limit = 0.0001, n_restarts = 1, step_size = 0.001,
    momentum_gamma = 0.9,  opt_type = "posterior", prior_parameter_a = 1,
    prior_parameter_b = 1.25
)

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
for (i in 1:50){
    lines(x[pred_inds], Y.interp[i,], col = "gray")
}
```



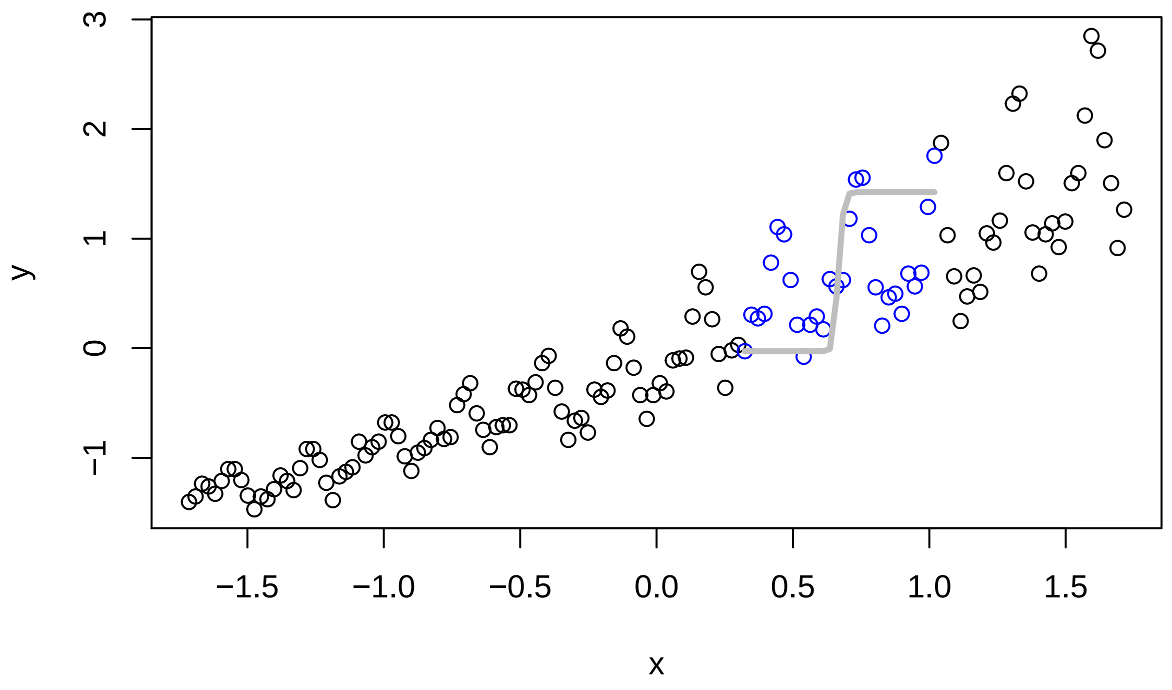
Compare this to smoothing spline interpolation

```r
# Fit and predict the model
spline.model <- smooth.spline(
  x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)]
)
y_pred <- predict(spline.model, x[pred_inds])$y

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y_pred, col = "gray", lwd = 3)
```



Also compare to random forest interpolation.

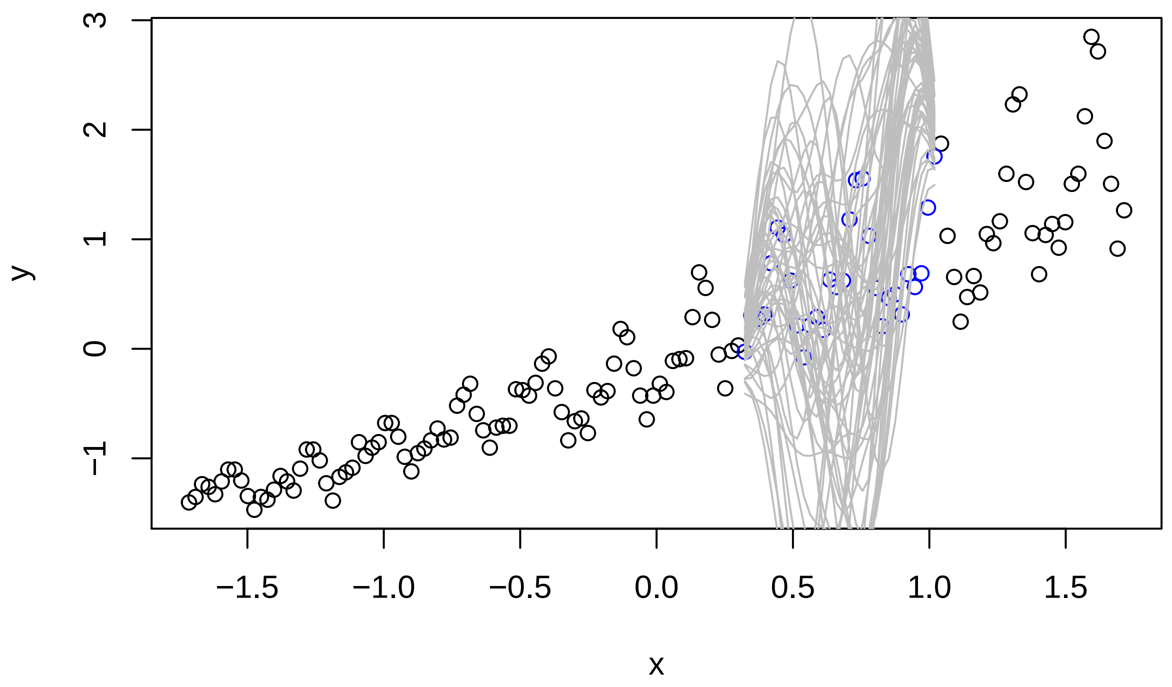
```r
# Fit and predict the model
rf.model <- randomForest(as.matrix(x[!(1:n %in% pred_inds)]), y[!(1:n %in% pred_inds)])
y.interp <- predict(rf.model, as.matrix(x[pred_inds]))

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
lines(x[pred_inds], y.interp, col = "gray", lwd = 3)
```



The GP approach appears to have the same pathology observed in 3.a.5 where the periodicity is ignored. Something to consider is that for data like this which have periodicity, the squared exponential kernel is likely inappropriate. It would thus make for fascinating future work to consider the role of the GP kernel on different data generating process.

Finally, we also consider whether the Bayesian optimization procedure will yield a drastically different set of hyperparameters and resulting interpolation set.

```r
# First, optimize the hyperparameters using BO
best_params_3b <- exp(surrogate.opt(
  n_guesses = 50, num_hparam = 3,
  x = x[!(1:n %in% pred_inds)], y = y[!(1:n %in% pred_inds)],
  eps = 0.01, prior_a = 1, prior_b = 1.25,
  n_candidates = 200, n_iter = 50
))

# Then draw some interpolations using those parameters
n_interp <- 50
sigma.f.hat.3b <- best_params_3b[1]
ell.hat.3b <- best_params_3b[2]
sigma.y.hat.3b <- best_params_3b[3]
gp_mu_3b <- gp_pred_mean(
  x[!(1:n %in% pred_inds)], x[pred_inds], y[!(1:n %in% pred_inds)],
  sigma.f.hat.3b, ell.hat.3b, sigma.y.hat.3b
)
gp_sigma_3b <- gp_pred_var(
  x[!(1:n %in% pred_inds)], x[pred_inds], y[!(1:n %in% pred_inds)],
  sigma.f.hat.3b, ell.hat.3b, sigma.y.hat.3b
)
Y_interp_3b <- mvrnorm(n = n_interp, mu = gp_mu_3b, Sigma = gp_sigma_3b)

# Plot the results
plot(x[!(1:n %in% pred_inds)], y[!(1:n %in% pred_inds)], xlab = "x", ylab = "y",
     xlim = c(min(x), max(x)), ylim = c(min(y), max(y)))
points(x[pred_inds], y[pred_inds], col = "blue")
for (i in 1:n_interp){
    lines(x[pred_inds], Y_interp_3b[i,], col = "gray")
}
```



While the interpolation using SGD was slightly over-smooth, the Bayesian optimization procedure appears to have returned something that is under-smooth.

References

Robert B. Gramacy. *Surrogates: Gaussian Process Modeling, Design and Optimization for the Applied Sciences*. Chapman Hall/CRC, Boca Raton, Florida, 2020. http://bobby.gramacy.com/surrogates/.

Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

Kevin P Murphy. *Machine Learning: a Probabilistic Perspective*. MIT press, 2012.

Matthias Schonlau. Computer experiments and global optimization. 1997.