Homework: Efficient Computing, Linear Algebra & Optimization

Runyan Xin

September 10, 2025

Instructions

- Work **individually**; discussion is encouraged, but submit your own write-up.
- Show your **reasoning** (math and code). Comment your code.
- Set a seed for reproducibility: `set.seed(4210)`.
- Please Upload your knitted **PDF** and **.Rmd** files to canvas before the due date

Setup

You may find these packages useful (install if needed):

```r
library(tidyverse)
library(MASS)          # Boston data, mvrnorm
library(Matrix)        # sparse/linear algebra helpers
library(microbenchmark)
library(dplyr)
```

Problem 1 — Vectorization, `apply()`, and quick EDA (10 pts)

We'll use the **Boston** housing data.

```r
data("Boston", package = "MASS")
glimpse(Boston)
```

```
## Rows: 506
## Columns: 14
## $ crim    <dbl> 0.00632, 0.02731, 0.02729, 0.03237, 0.06905, 0.02985, 0.08829,~
## $ zn      <dbl> 18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 12.5, 12.5, 12.5, 12.5, 12.5, 1~
## $ indus   <dbl> 2.31, 7.07, 7.07, 2.18, 2.18, 2.18, 7.87, 7.87, 7.87, 7.87, 7.~
## $ chas    <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ nox     <dbl> 0.538, 0.469, 0.469, 0.458, 0.458, 0.458, 0.524, 0.524, 0.524,~
## $ rm      <dbl> 6.575, 6.421, 7.185, 6.998, 7.147, 6.430, 6.012, 6.172, 5.631,~
## $ age     <dbl> 65.2, 78.9, 61.1, 45.8, 54.2, 58.7, 66.6, 96.1, 100.0, 85.9, 9~
## $ dis     <dbl> 4.0900, 4.9671, 4.9671, 6.0622, 6.0622, 6.0622, 5.5605, 5.9505~
## $ rad     <int> 1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4,~
## $ tax     <dbl> 296, 242, 242, 222, 222, 222, 311, 311, 311, 311, 311, 311, 31~
## $ ptratio <dbl> 15.3, 17.8, 17.8, 18.7, 18.7, 18.7, 15.2, 15.2, 15.2, 15.2, 15~
## $ black   <dbl> 396.90, 396.90, 392.83, 394.63, 396.90, 394.12, 395.60, 396.90~
## $ lstat   <dbl> 4.98, 9.14, 4.03, 2.94, 5.33, 5.21, 12.43, 19.15, 29.93, 17.10~
## $ medv    <dbl> 24.0, 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15~
```

(a) Compute the 90th percentile of `medv`. Construct a factor `price_tier` `{expensive, inexpensive}` indicating whether each town is above/below this threshold. Using **vectorized** code (no explicit loops), compute the mean `medv` by tier. Briefly interpret the difference.

```r
quantile(Boston$medv,0.9) #the 90th percentile of 'medv'
```

```
##  90%
## 34.8
```

```r
price_tier<-factor(ifelse(Boston$medv>=quantile(Boston$medv,0.9),"expensive","inexpensive"))
#use ifelse to construct a factor `price_tier  {expensive, inexpensive}`
price_tier #the value indicates whether each town is above/below this threshold
```

```
##   [1] inexpensive inexpensive inexpensive inexpensive expensive   inexpensive
##   [7] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [13] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [19] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [25] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [31] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [37] inexpensive inexpensive inexpensive inexpensive expensive   inexpensive
##  [43] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [49] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [55] inexpensive expensive   inexpensive inexpensive inexpensive inexpensive
##  [61] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [67] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [73] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [79] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [85] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [91] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
##  [97] inexpensive expensive   expensive   inexpensive inexpensive inexpensive
## [103] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [109] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [115] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [121] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [127] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [133] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [139] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [145] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [151] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [157] inexpensive expensive   inexpensive inexpensive inexpensive expensive
## [163] expensive   expensive   inexpensive inexpensive expensive   inexpensive
## [169] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [175] inexpensive inexpensive inexpensive inexpensive inexpensive expensive
## [181] expensive   expensive   expensive   inexpensive inexpensive inexpensive
## [187] expensive   inexpensive inexpensive expensive   expensive   inexpensive
## [193] expensive   inexpensive inexpensive expensive   inexpensive inexpensive
## [199] inexpensive expensive   inexpensive inexpensive expensive   expensive
## [205] expensive   inexpensive inexpensive inexpensive inexpensive inexpensive
## [211] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [217] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [223] inexpensive inexpensive expensive   expensive   expensive   inexpensive
## [229] expensive   inexpensive inexpensive inexpensive expensive   expensive
## [235] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [241] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [247] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [253] inexpensive expensive   inexpensive inexpensive expensive   expensive
## [259] expensive   inexpensive inexpensive expensive   expensive   inexpensive
## [265] expensive   inexpensive inexpensive expensive   expensive   inexpensive
## [271] inexpensive inexpensive inexpensive expensive   inexpensive inexpensive
## [277] inexpensive inexpensive inexpensive expensive   expensive   expensive
## [283] expensive   expensive   inexpensive inexpensive inexpensive inexpensive
## [289] inexpensive inexpensive inexpensive expensive   inexpensive inexpensive
## [295] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [301] inexpensive inexpensive inexpensive inexpensive expensive   inexpensive
## [307] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [313] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [319] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [325] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [331] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [337] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [343] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [349] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [355] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [361] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [367] inexpensive inexpensive expensive   expensive   expensive   expensive
## [373] expensive   inexpensive inexpensive inexpensive inexpensive inexpensive
## [379] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [385] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [391] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [397] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [403] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [409] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [415] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [421] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [427] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [433] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [439] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [445] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [451] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [457] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [463] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [469] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [475] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [481] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [487] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [493] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [499] inexpensive inexpensive inexpensive inexpensive inexpensive inexpensive
## [505] inexpensive inexpensive
## Levels: expensive inexpensive
```

```
mean(Boston$medv[price_tier == "expensive"]) # use 'vectorized' function( mean() ) to compute
```

```
## [1] 43.13137
```

```
mean(Boston$medv[price_tier == "inexpensive"])
```

```
## [1] 20.22396
```

```
#The difference:towns classified as "expensive" (above the 90th percentile of medv)
#have, on average, significantly higher median home values than towns classified as
#"inexpensive"
```

(b) With a single `apply()` (or `sapply()`), compute the **(Q1, median, Q3)** for every **numeric** column. Return the results.

```
apply(Boston[,c("crim","zn","indus","nox","rm","age","dis","rad","tax","ptratio","black","lsta
```

```
##         crim   zn indus   nox     rm    age      dis rad tax ptratio    black
## 25% 0.082045  0.0  5.19 0.449 5.8855 45.025 2.100175   4 279   17.40 375.3775
## 50% 0.256510  0.0  9.69 0.538 6.2085 77.500 3.207450   5 330   19.05 391.4400
## 75% 3.677083 12.5 18.10 0.624 6.6235 94.075 5.188425  24 666   20.20 396.2250
##      lstat   medv
## 25%  6.950 17.025
## 50% 11.360 21.200
## 75% 16.955 25.000
```

(c) Write **two** functions that, given a numeric matrix, return the column means: 1. a **for-loop** version; 2. a **vectorized** version using `colMeans`. Briefly interpret the speed difference.

```
mean_loop<-function(x){ #the first function" a for-loop" version
  start1<-Sys.time()
  mean_col <- numeric(ncol(x)) #create function 'mean_col'
  for(i in 1:ncol(x)){ #for every column
    sum<-0
    for(j in 1:nrow(x)){ #compute the sum of the column i
      sum<-sum+x[j,i]
    }
    mean_col[i]<-sum/nrow(x) #compute the mean of the column i
  }
  return( mean_col) #return the column means
}
set.seed(4210)
mat_1<-matrix(rnorm(20), nrow = 5, ncol = 4) #given a numeric matrix
start1<-Sys.time() #the start time of 'a for-loop' version
mean_loop(mat_1) #the means generated by 'a for-loop' version
```

```
## [1]  0.15091041 -0.04250015  0.09913646  0.43454217
```

```
print(Sys.time()-start1) #the time difference of 'a for-loop' version
```

```
## Time difference of 0.003369093 secs
```

```
start2<-Sys.time() #the start time of 'vectorized' version
colMeans(mat_1) #the means generated by 'vectorized' version
```

```
## [1]  0.15091041 -0.04250015  0.09913646  0.43454217
```

```
print(Sys.time()-start2) #the time difference of 'vectorized' version
```

```
## Time difference of 0.000369072 secs
```

```
#Interpretation of the speed difference: Vectorization is faster because it operates
#on the entire matrix in a single batch operation, unlike for-loops which require
#repetitive individual operations on each element.
```

(d) Choose four columns and compute the singular value decomposition of the resulting data matrix. Make sure that the variables are appropriately scaled. Is there any interpretation based on how the variables load onto the right singular vectors? Explain.

```
data_d<-Boston[,c("crim","ptratio","medv","rm")] #chose"crim","ptratio","medv","rm" as four co
center_scale_data_d<-scale(data_d,center = T,scale = T) # center and scale the matric
svd_data<-svd(center_scale_data_d) #SVD
```

```r
V_svd<-svd_data$v #the right singular vectors
U_svd<-svd_data$u #the left singular vectors
D_svd<-svd_data$d #the singluar values
X_svd<-center_scale_data_d%*%V_svd[,1:3] #X*
V_svd #print the right singular vectors
```

```
##            [,1]       [,2]       [,3]       [,4]
## [1,]  0.3816613 0.85802292 -0.3054624  0.1575567
## [2,]  0.4733236 0.06187493  0.8594879  0.1828028
## [3,] -0.5951091 0.17738701  0.1513880  0.7690648
## [4,] -0.5254947 0.47801883  0.3808609 -0.5918600
```

```r
#the interpretation based on V_svd :For the first principal component, the loadings
#are [0.38, 0.47, -0.60, -0.53]. The coefficients for 'crim' and 'ptratio' are large
#and positive, while those for 'medv' and 'rm' are large and negative. This indicates
#a negative relationship between "high crime rate & high pupil-teacher ratio" and
#"high housing prices & larger houses." In other words, neighborhoods with poorer
#education and worse safety conditions tend to have lower housing prices and smaller
#houses, whereas neighborhoods with better education and safer conditions tend to have
#higher housing prices and larger houses.
```

Problem 2 — Linear regression via normal equations, QR, and Cholesky (20 pts)

We'll compare several ways to solve least squares $\min_\beta \|y - X\beta\|_2^2$.

Generate data with **correlated predictors**:

(a) The ordinary least squares solution to the above optimization problem is obtained by solving the normal equations $X^\top X\beta - X^\top y = 0$ for β. Using a QR decomposition for X, show that the normal equations may be reduced to solving a linear system $R\beta = Q^\top y$. Hint: under the QR decomposition $Q^\top Q = I_p$, and $(QR)^\top = R^\top Q^\top$.

Proof.

$$X^T X\beta - X^T y = 0$$

which implies

$$X^T X\beta = X^T y$$

Let

$$X = QR$$

Then

$$(QR)^T(QR)\beta = (QR)^T y$$

$$R^T Q^T(QR)\beta = R^T Q^T y$$

$$R^T R\beta = R^T Q^T y$$
$$R\beta = Q^T y$$

(b) Solve the normal equations via

- **Direct**: compute $\hat{\beta} = (X^\top X)^{-1} X^\top y$ with matrix inversion

```
# X: design matrix, y: response vector
set.seed(4210)
n<-1000 #number of observation
p<-10 #number of variables
start_direct<-Sys.time()
X<-matrix(rnorm(n*p),n,p) # create a matrix X(n*p)
y<-rnorm(n) #create a y
beta_hat_direct <- solve(t(X) %*% X,t(X) %*% y) #use slove() to solve the normal equations
time_direct<-Sys.time()-start_direct
print(beta_hat_direct) # the bate_hat
```

```
##               [,1]
##  [1,]  0.048598151
##  [2,] -0.019416787
##  [3,]  0.022484611
##  [4,] -0.044250933
##  [5,]  0.067023784
##  [6,] -0.020070048
##  [7,] -0.008329842
##  [8,]  0.022357539
##  [9,]  0.005438889
## [10,] -0.033789238
```

- **QR Decomposition**: use `qr(X)` and triangular solves to get $\hat{\beta}$

```
start_qr<-Sys.time()
qr_X<-qr(X)
Q<-qr.Q(qr_X) #QR decomposition
R<-qr.R(qr_X)
beta_hat_qr<-backsolve(R,t(Q)%*%y) #triangular solves
time_qr<-Sys.time()-start_qr
print(beta_hat_qr)
```

```
##               [,1]
##  [1,]  0.048598151
##  [2,] -0.019416787
##  [3,]  0.022484611
##  [4,] -0.044250933
##  [5,]  0.067023784
##  [6,] -0.020070048
##  [7,] -0.008329842
##  [8,]  0.022357539
##  [9,]  0.005438889
## [10,] -0.033789238
```

- **Cholesky Decomposition**: form $G = X^\top X$ and use `chol(G)`, then **forward/back** solves via `forwardsolve`/`backsolve` to recover $\hat{\beta}$.

```r
start_Cholesky<-Sys.time()
G<-t(X)%*%X #G=X^T X
C<-chol(G) #G=C^T C, C is upper triangular matrix
Z<-forwardsolve(t(C),t(X)%*%y) # C^T C beta = X^T y -> Z=C beta
beta_hat_Cholesky<-backsolve(C,Z) # C beta=Z
time_Cholesky<-Sys.time()-start_Cholesky
print(beta_hat_Cholesky)
```

```
##              [,1]
##  [1,]  0.048598151
##  [2,] -0.019416787
##  [3,]  0.022484611
##  [4,] -0.044250933
##  [5,]  0.067023784
##  [6,] -0.020070048
##  [7,] -0.008329842
##  [8,]  0.022357539
##  [9,]  0.005438889
## [10,] -0.033789238
```

Provide concise code for each path.

(b) Compare **timings** and **numerical accuracy** by reporting $\|\hat{\beta}_{\text{method}} - \hat{\beta}_{\text{true}}\|_2$.

```r
time_direct
```

```
## Time difference of 0.001087904 secs
```

```r
time_qr
```

```
## Time difference of 0.001816034 secs
```

```r
time_Cholesky
```

```
## Time difference of 0.0009829998 secs
```

```
#the speed of Cholesky is the most faster, the second one is Direct, and the lowest one is QR
```

Problem 3 — Multivariate normal simulation at scale (40 pts)

Part (a) Let $p = 1000$, $n = 1000$. Construct a **valid** covariance Σ (e.g., $\Sigma_{ij} = 0.8^{|i-j|}$). Compare two approaches to simulate n samples $X \sim \mathcal{N}(0, \Sigma)$:

- `MASS::mvrnorm`
- **Cholesky** decomposition

Measure the compute time for each and verify empirically that `cov(X)` is close to Σ (use Frobenius norm of the difference). Which approach is faster on your machine?

```
p<-1000
n<-1000
A<-matrix(rnorm(n*p),nrow=1000) #create matirx A
Sigma <- matrix(0, nrow = p, ncol = p) #create Sigma
for (i in 1:p) {
  for (j in 1:p) {
    Sigma[i, j] <- 0.8^abs(i - j)
  }
}

#MASS::mvrnorm
library(MASS)
set.seed(4210)
start_mv<-Sys.time() # start time of mvrnorm
X_mv<-mvrnorm(n=n,mu=rep(0,p),Sigma = Sigma) #use MASS::mvrnorm to create X_mv
print(Sys.time()-start_mv) # compute time of mvrnorm
```

```
## Time difference of 1.405761 secs
```

```
cov_mv<-cov(X_mv) #compulate the covariance of X_mv

# **Cholesky** decomposition
start_chol <- Sys.time() # start time of Cholesky
Z<-matrix(rnorm(n*p),nrow = 1000) #Z
L<-chol(Sigma) #Sigma=L'*L, L is an upper-triangular.
X_chol<-Z%*%L #cov(X_xhol)=cov(Z*L)=L'*cov(Z)*L=Sigma
print(Sys.time()-start_chol) # compute time of Cholesky
```

```
## Time difference of 0.3969531 secs
```

```
cov_chol<-cov(X_chol) #compulate the covariance of X_chol

#compare
diff_mv<-cov_mv-Sigma
diff_chol<-cov_chol-Sigma
Frobenius<-function(matrix){ #create a function to compute Frobenius norm of the difference
  dim_row<-dim(matrix)[1]
  dim_col<-dim(matrix)[2]
  sum_frobenius<-0
  for(i in 1:dim_row){
    for(j in 1:dim_col){
      sum_frobenius=sum_frobenius+matrix[i,j]^2
    }
  }
  return(sqrt(sum_frobenius))
}
F_mv<-Frobenius(diff_mv)
F_chol<-Frobenius(diff_chol)
print(F_mv) #the Frobenius norm of the difference of MASS::mvrnorm
```

```
## [1] 32.16917
```

```
print(F_chol) #the Frobenius norm of the difference of  Cholesky decomposition
```

```
## [1] 31.61577
```

```
#according to the result of comparsion, the Cholesky decomposition is faster
#the Frobenius norm of the difference of Cholesky decomposition is smaller
#the Cholesky decomposition is better
```

Part (b): **Gaussian process (GP)** regression is one of the most powerful and flexible Bayesian methods, and is employed for a variety of applications. However, estimation of Gaussian process regression is limited by inversion and storage of large matrices, which is what you will explore in this problem.

A Bayesian Gaussian process regression model is written hierarchically as

$$
\begin{aligned}
f &\sim GP(m, K),\\
y &= f(x) + \epsilon
\end{aligned}
$$

Where a Gaussian process prior $f \sim GP(m, K)$ is a prior distribution over the unknown regression function, such that for any finite dimensional vector $(x_1, \ldots, x_n)$,

$$(f(x_1), \ldots, f(x_n)) \sim \text{Multivariate Normal}(m, K),\ m_i = m(x_i),\ K_{ij} = k(x_i x_j; \theta)$$

Here, $k(.,.;\theta)$ is a positive definite kernel (covariance function) with hyperparameters θ. A common choice is the **squared-exponential (SE)** (a.k.a. radial basis function, RBF), which in one dimension, is given by:

$$k(x, x'; \theta) = \sigma_f^2 \exp\Big(-\frac{(x-x')^2}{2\ell^2} \Big),$$

where: - $\sigma_f^2 > 0$ controls **vertical scale** (signal variance), - $\ell > 0$ is the **length-scale** controlling smoothness/correlation range,

In Bayesian Gaussian process regression, we estimate a) the *posterior* distribution of f, evaluated at our $(x_1, \dots, x_n)$, and b) usde this posterior to predict at new inputs x^*.

In this exercise, you'll begin to understand some of the computational bottlenecks of this popular machine learning method, as well as its flexibility.

Task 1

Over an evenly spaced grid of $n = 100$ x-values between (-2,2), write a function to compute the squared-exponential kernel where the user can control both $\sigma_f^2$and ℓ. Return the covariance kernel. What is the dimension?

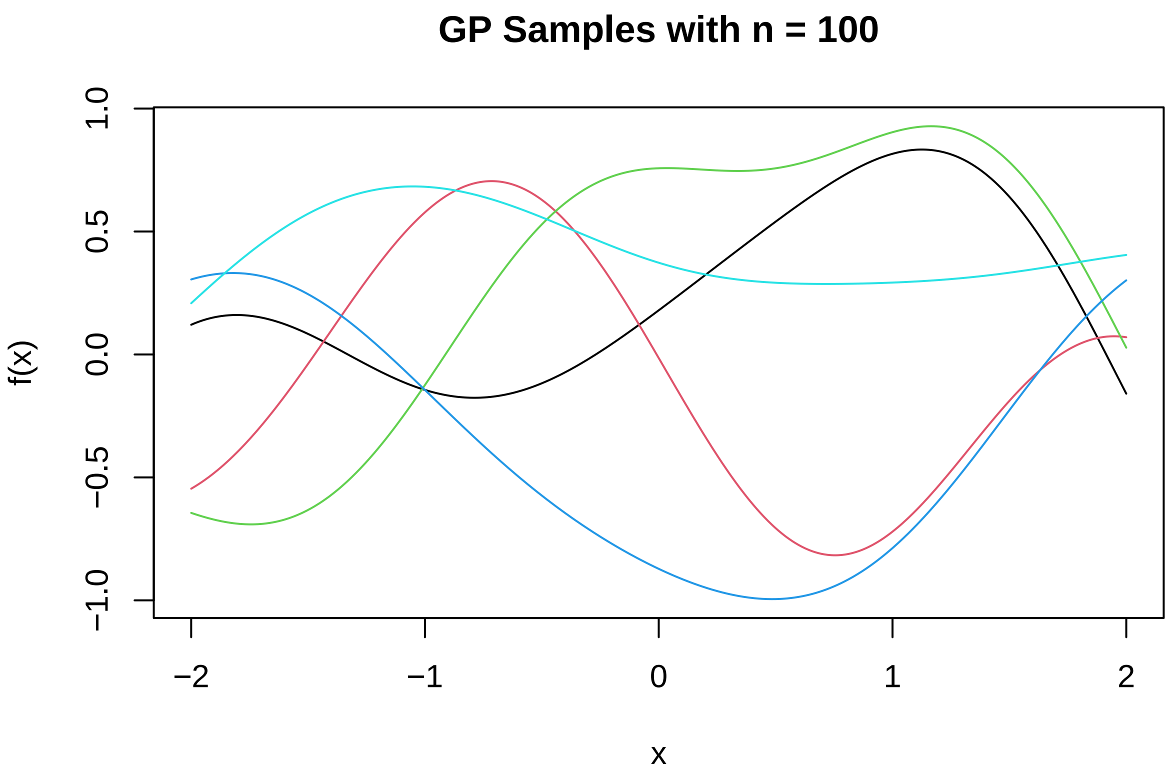
```r
n_task1<-100
n<-n_task1
x_values<-seq(-2, 2, length.out = 100) #an evenly spaced grid of   = 100 x-values between (-2,2
SE<-function(a,b){
  Sigma_square<- a
  l<-b
  k<-matrix(,nrow=n,ncol=n) #create an empty 100*100 matrix k
  for(i in 1:n){ #row i
    for(j in 1:n){ #col j
      k[i,j]<-Sigma_square*exp(-(x_values[i]-x_values[j])^2/(2*l^2))
    }
  }
  return(k)
}
#the dimension of k is 100*100
```

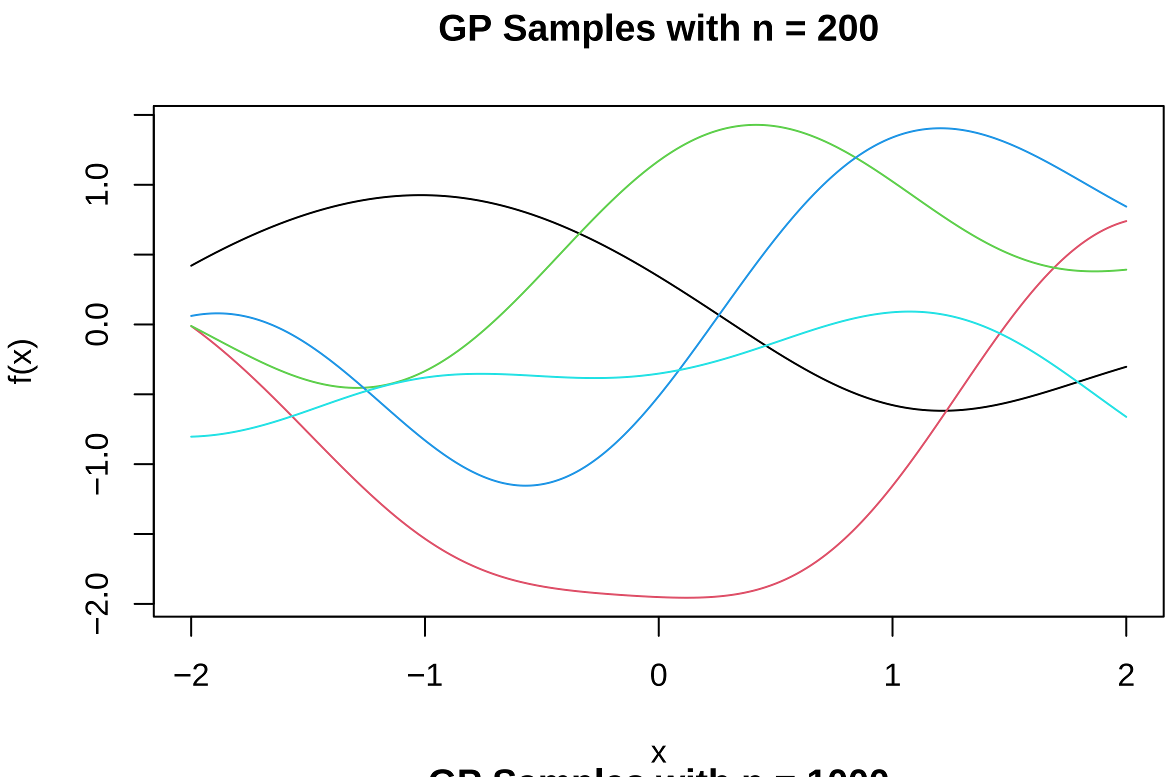
Task 2

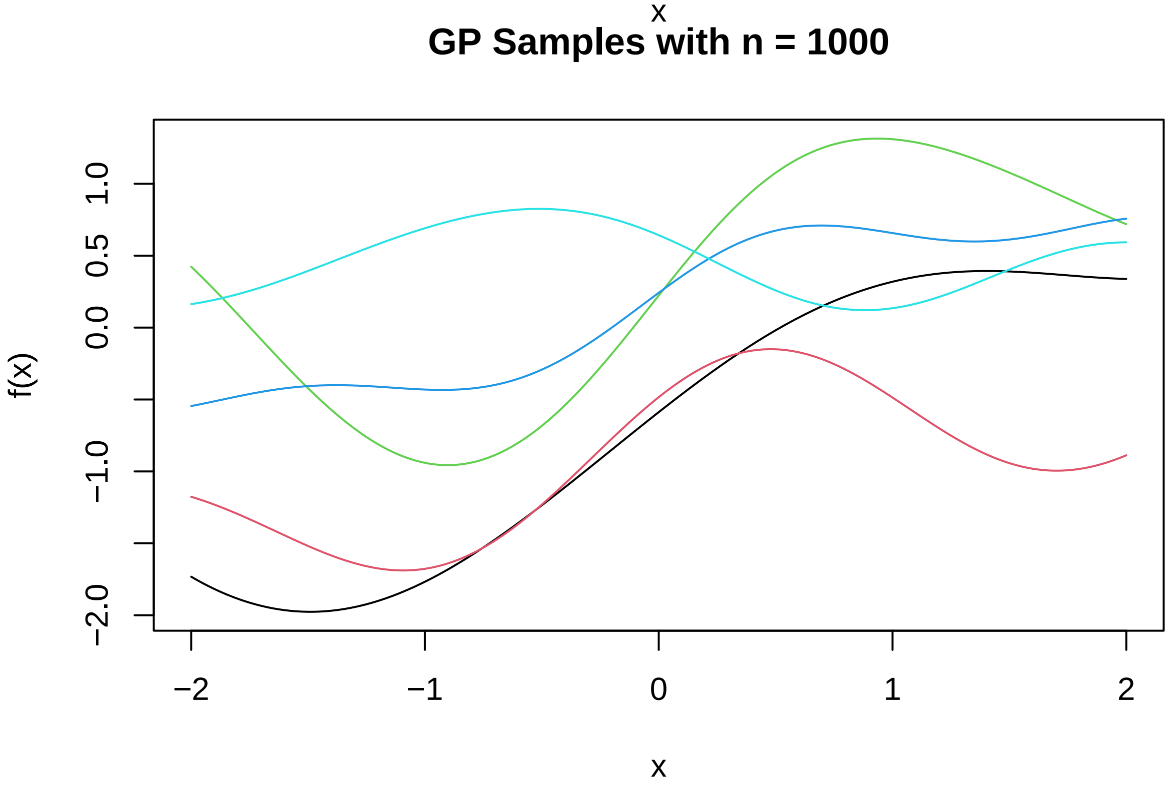
Across an evenly spaced grid of x-values between (-2,2), simulate 5 realizations of f from a mean-zero Gaussian process prior with squared-exponential kernel (i.e., $(f(x_1), \dots, f(x_n)) \sim$ Multivariate Normal(m, K), $m_i = 0$, $K_{ij} = k(x_i x_j; \theta)$), fixing $\theta = (\sigma_f = 1, \ell = 1)$ for samples of size $n \in \{100, 200, 1000\}$. To do so, use the Cholesky decomposition of K and the `MASS::mvrnorm` package. Use for loops to iterate over the sample sizes. Plot the realizations.

```r
library(MASS)
set.seed(4210)
n_task2<-c(100,200,1000)
start_task2=Sys.time()
```

```r
for(p in n_task2){
  x_values<-seq(-2, 2, length.out = i)
  n<-i
  SE_Kernel<-SE(1,1)
  f_task2 <- mvrnorm(5, mu = rep(0, i), Sigma = SE_Kernel)
  matplot(x_values, t(f_task2), type="l", lty=1, col=1:5,main=paste("GP Samples with n =", p),
          xlab="x", ylab="f(x)")

}
```





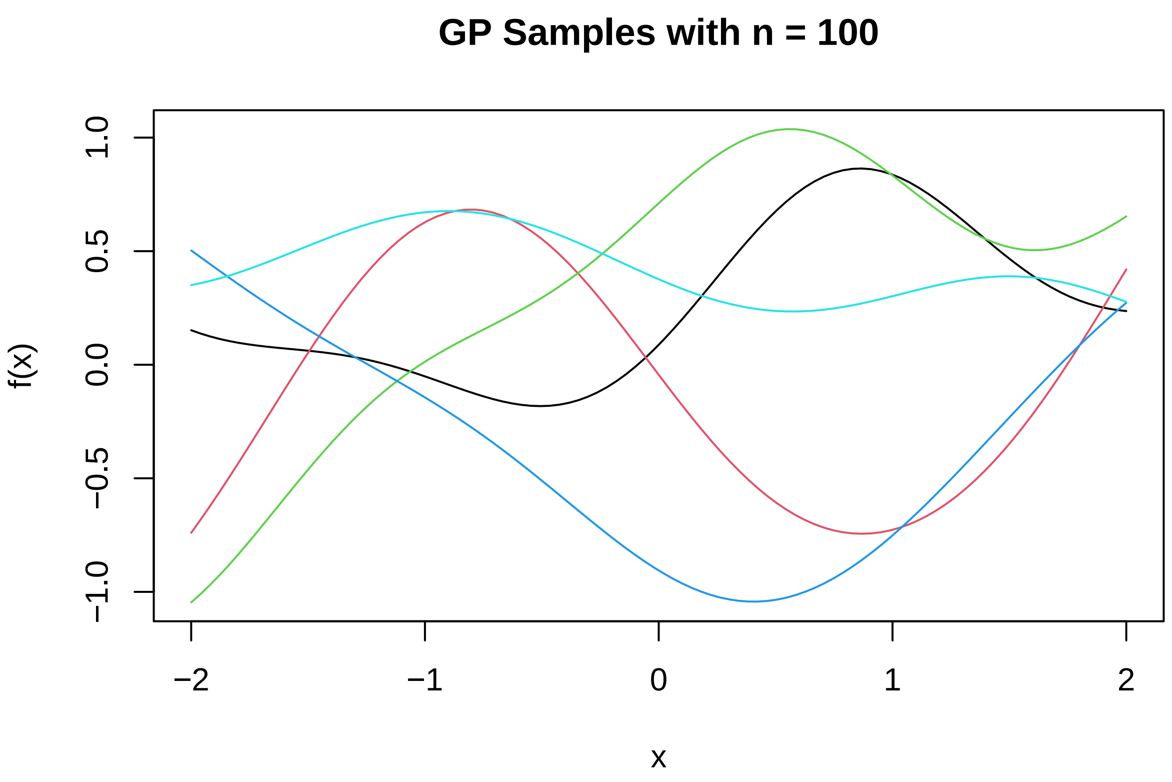


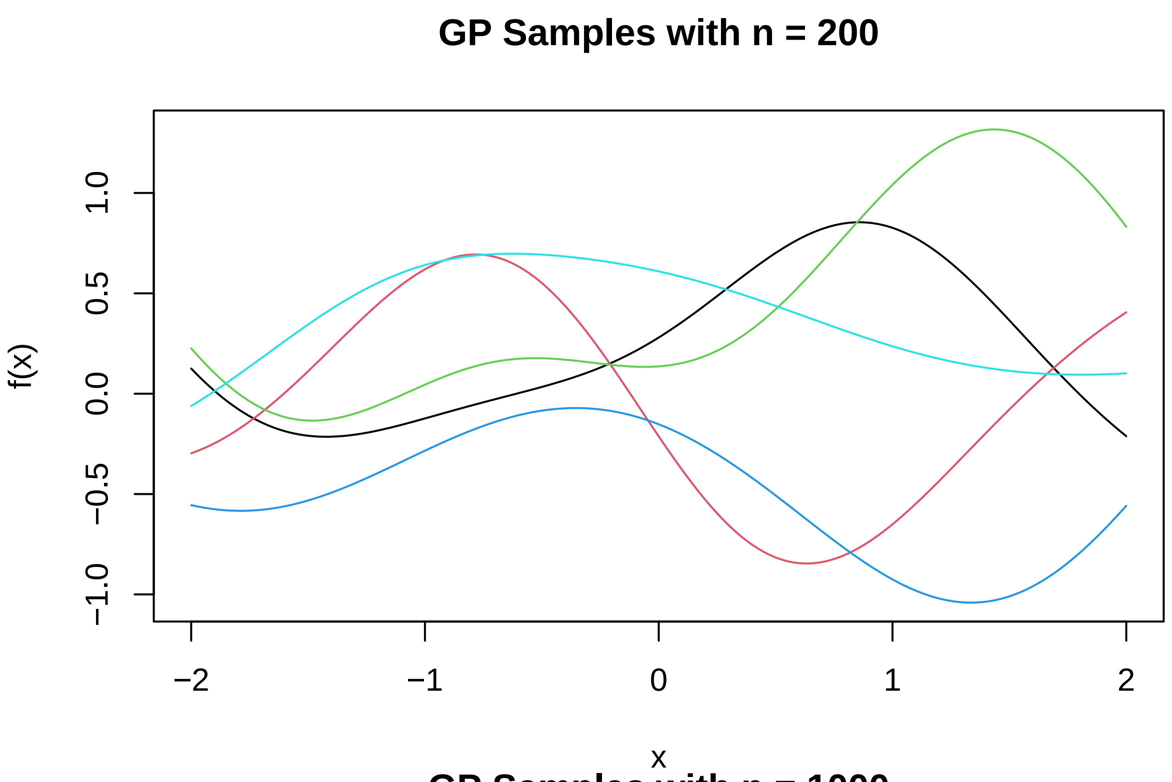
```
time_2<-Sys.time()-start_task2
```

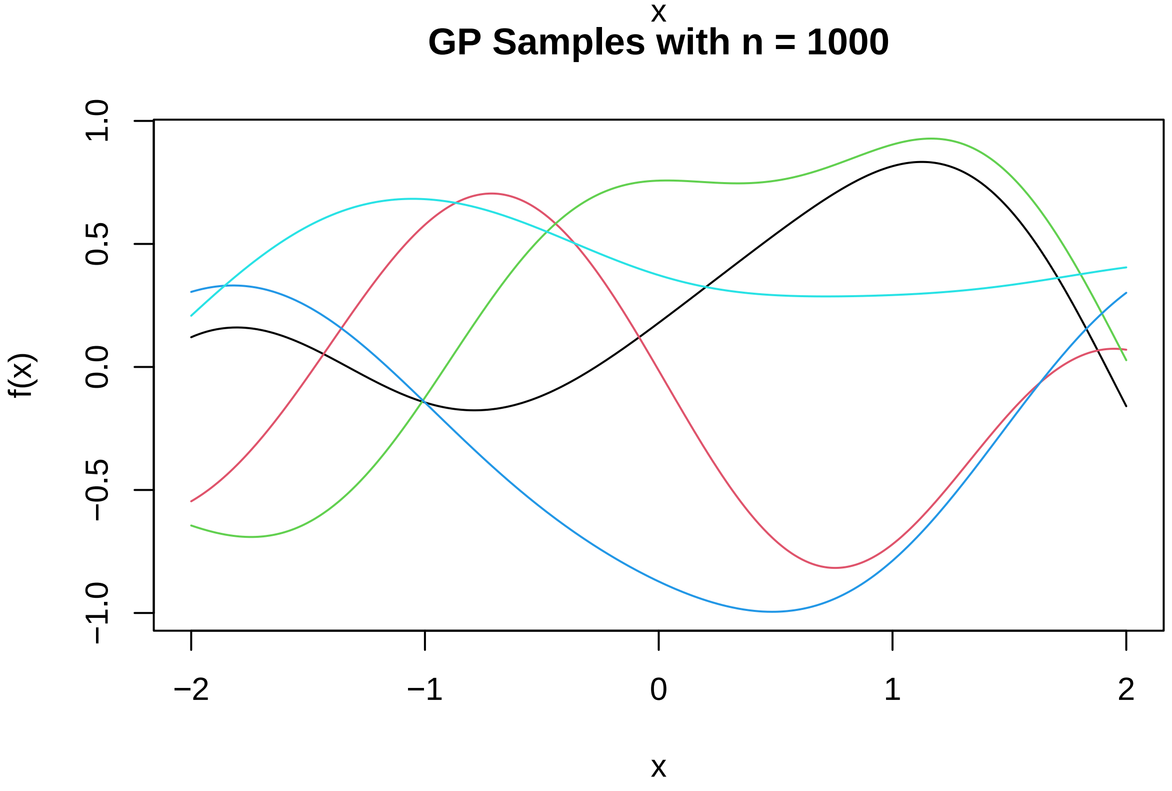
Task 3

Do the same thing as in Task 2, but use parallel computing instead of for loops. Save the realized values of f for each sample size and plot them. What is the payoff in compute time?

```r
library(foreach)
library(doParallel)
num_cores<-parallel::detectCores()
cl<-makeCluster(num_cores)
registerDoParallel(cl)
n_iter=3
start_task3=Sys.time()
results<-foreach(i=1:3)%dopar%{
  library(MASS)
  set.seed(4210)
  x_values<-seq(-2, 2, length.out = n_task2[i])
  n<-n_task2[i]
  SE_Kernel<-SE(1,1)
  f_task2 <- mvrnorm(5, mu = rep(0, n_task2[i]),Sigma = SE_Kernel)
  f_task2
}
stopCluster(cl)

for(i in 1:3){
   x_values<-seq(-2, 2, length.out = n_task2[i])
   matplot(x_values, t(results[[i]]), type="l", lty=1, col=1:5,main=paste("GP Samples with n ="
          xlab="x", ylab="f(x)")
}
```







```r
time_3<-Sys.time()-start_task3
payoff_time<-time_2-time_3
payoff_time
```

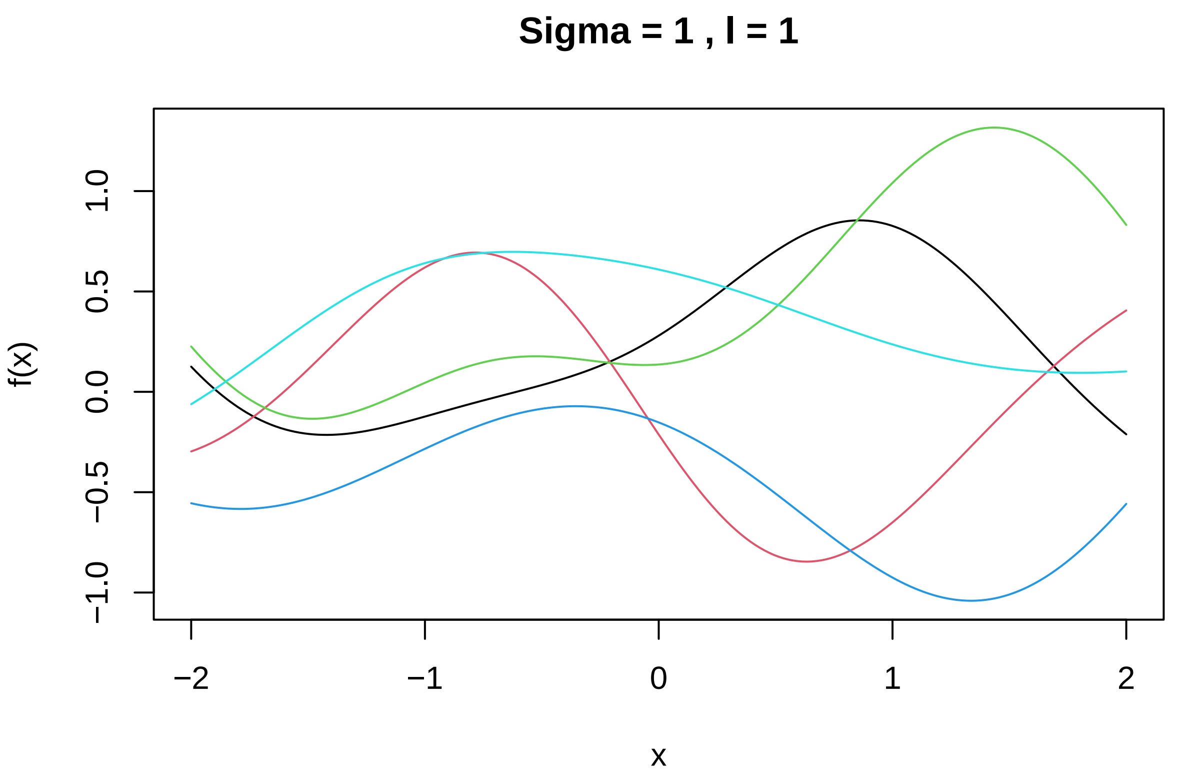
```
## Time difference of 1.471509 secs
```

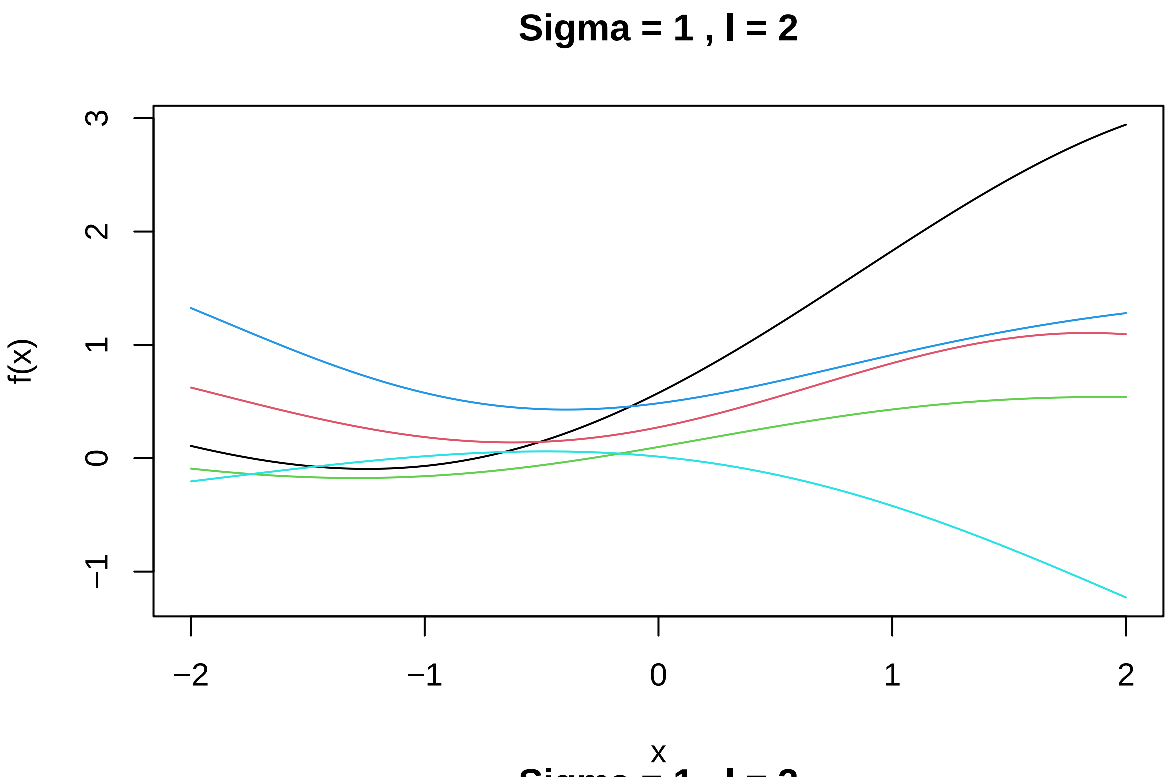
```
#the time of task3 is shorter than task2
```

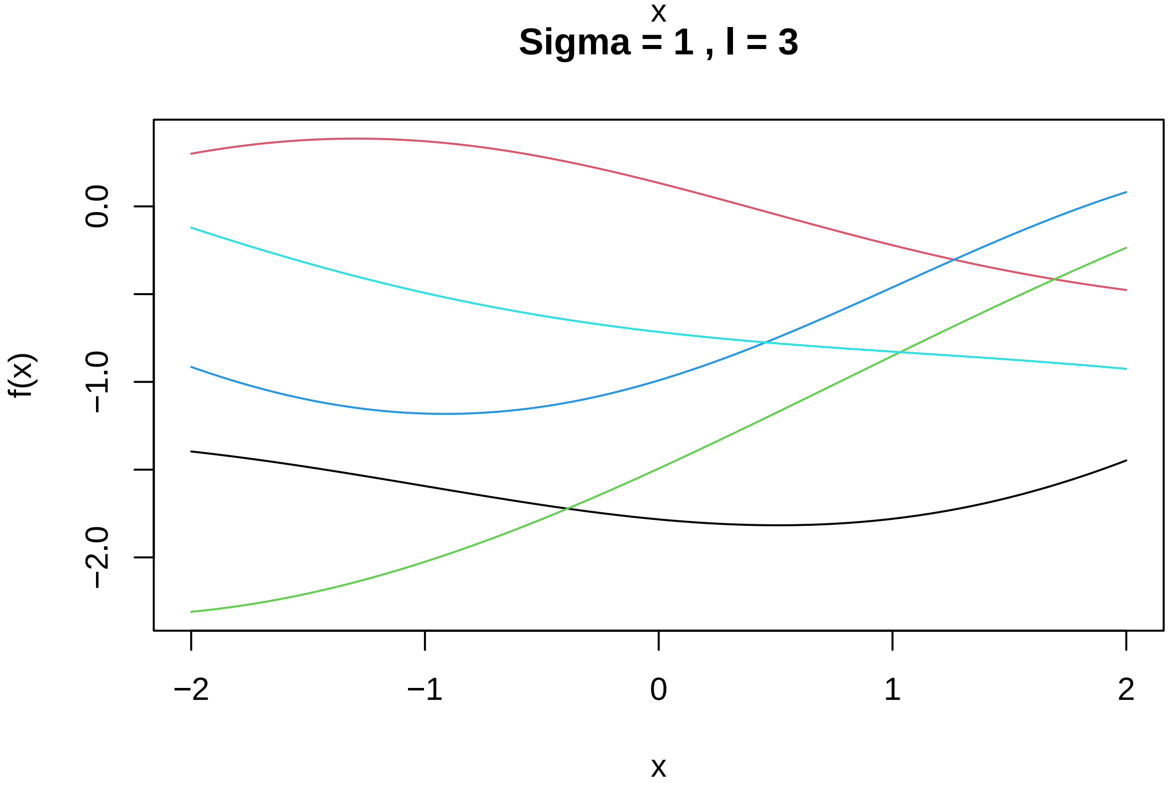
Task 4

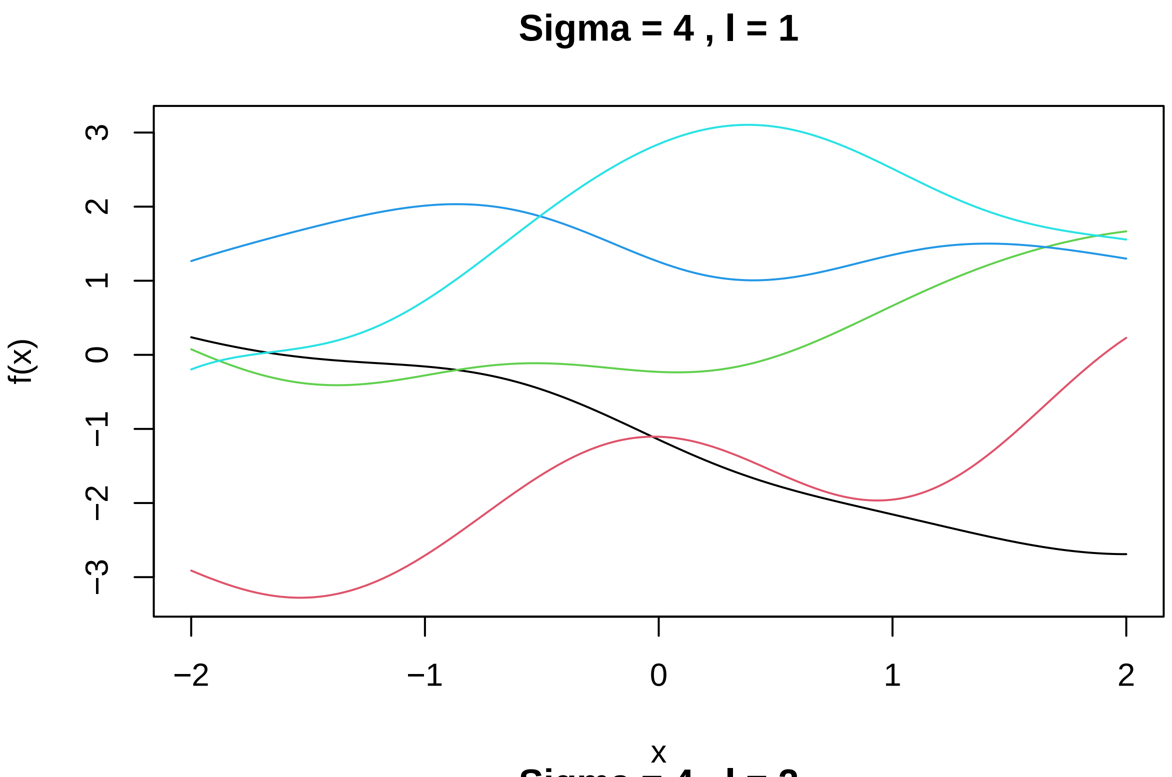
Fixing $n = 200$, simulate samples of f at $(x_1, \ldots, x_n)$ from a GP prior, this time for different combinations of σ_f, ℓ. What happens to the simulated functions?

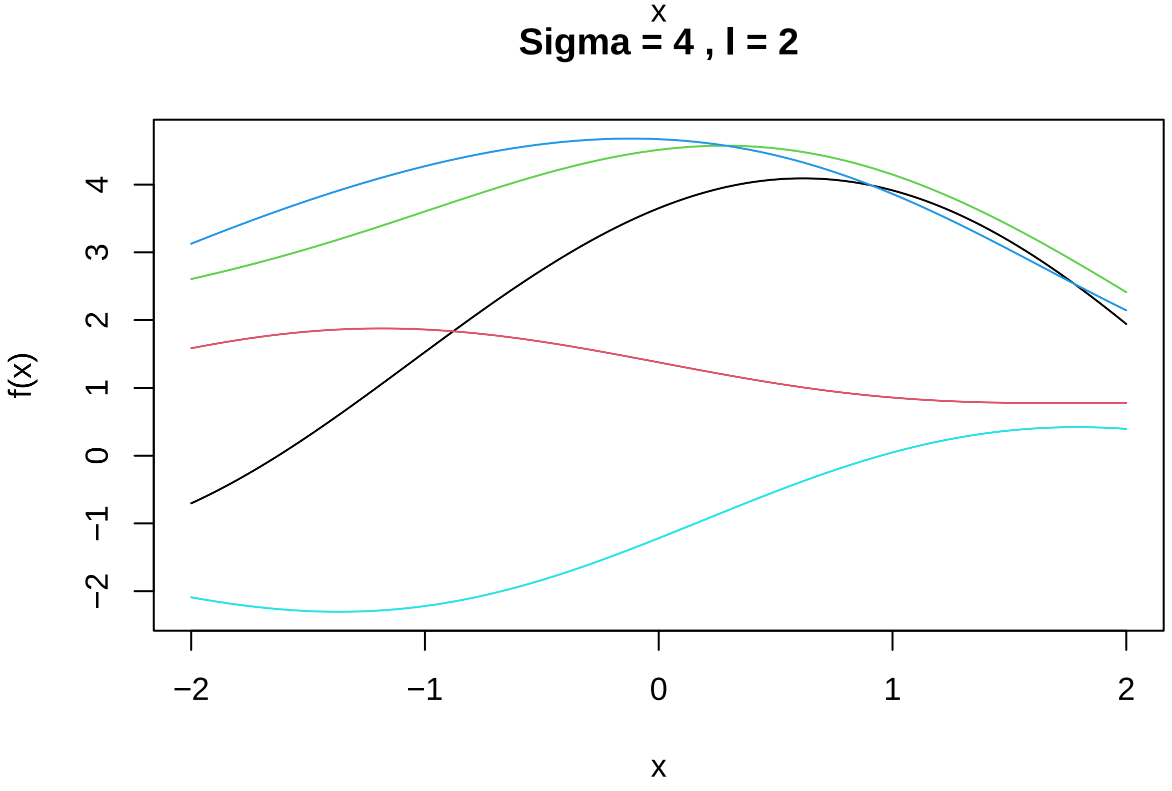
```
library(MASS)
set.seed(4210)
n_task4<-200
n<-n_task4
Sigma_task4<-c(1,4,9)
l_task4<-c(1,2,3)
for(i in 1:3){
  for(j in 1:3){
  x_values<-seq(-2, 2, length.out = n_task4)
  SE_Kernel<-SE(Sigma_task4[i],l_task4[j])
  f_task4 <- mvrnorm(5, mu = rep(0, n_task4), Sigma = SE_Kernel)
  matplot(x_values, t(f_task4), type="l", lty=1, col=1:5,main = paste("Sigma =", Sigma_task4[i
          xlab="x", ylab="f(x)")
  }
}
```

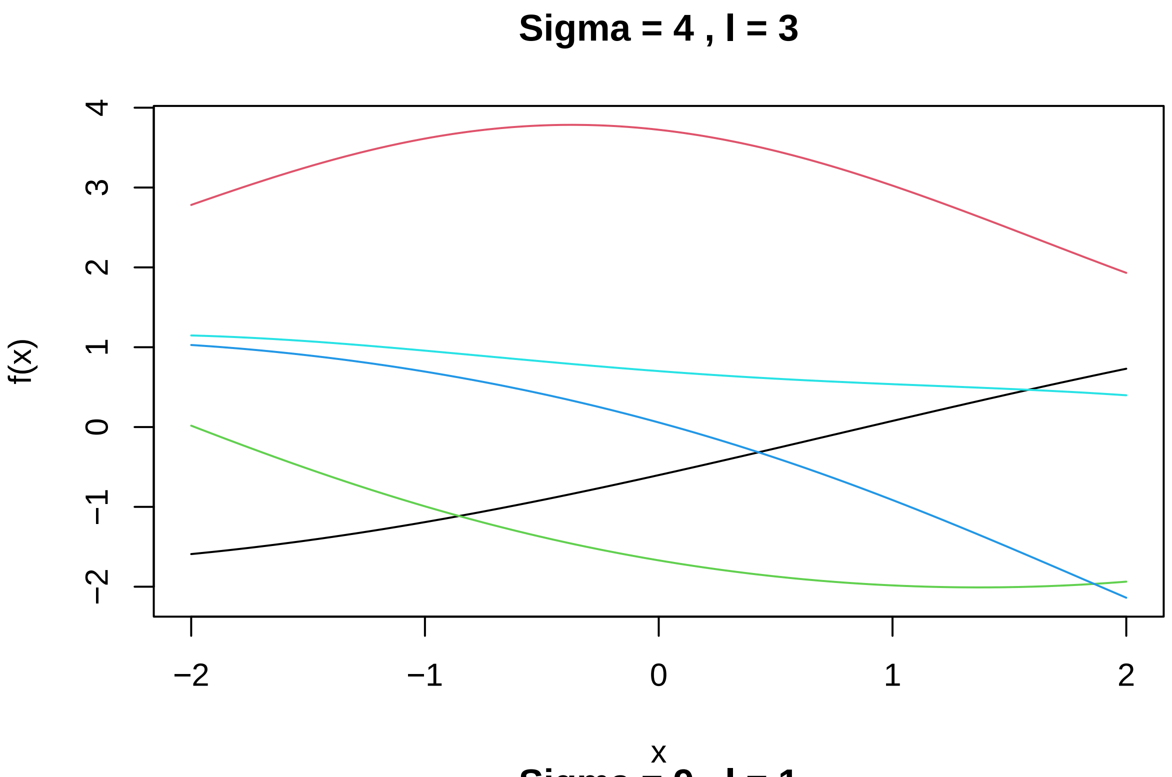


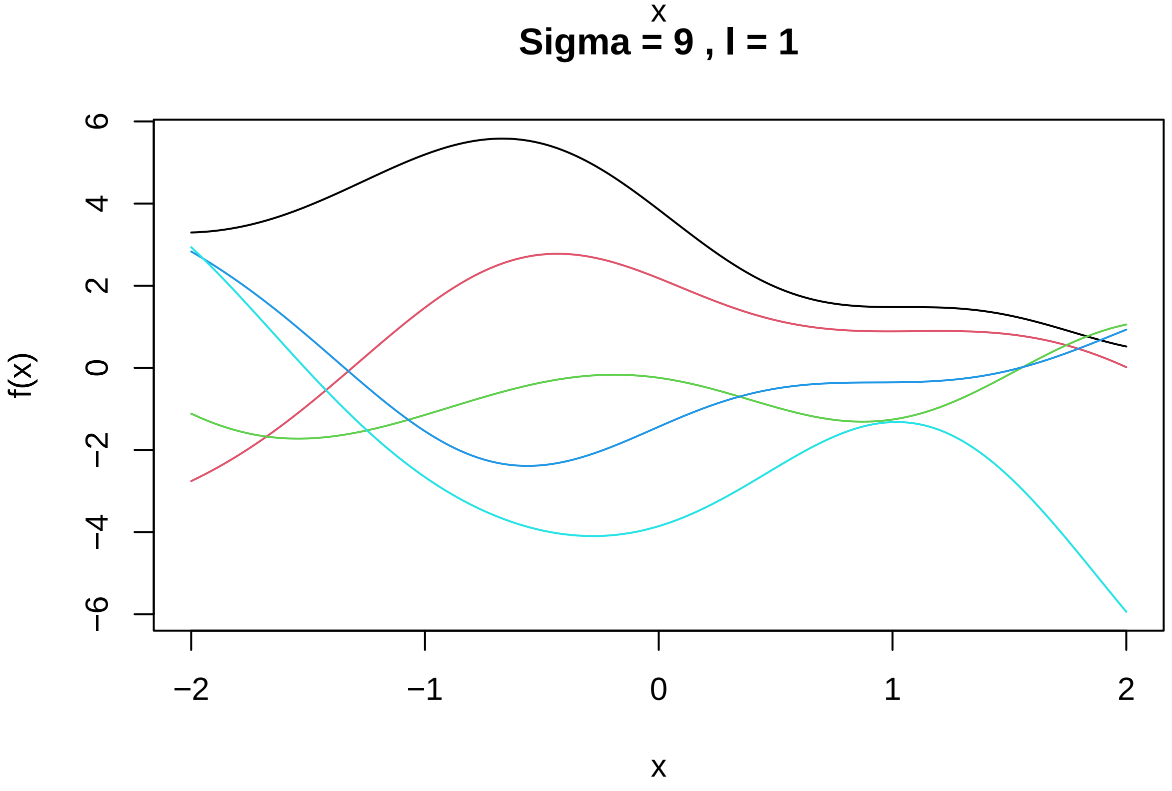

Sigma = 1 , l = 2
f(x)
x
3
2
1
0
−1
−2
−1
0
1
2

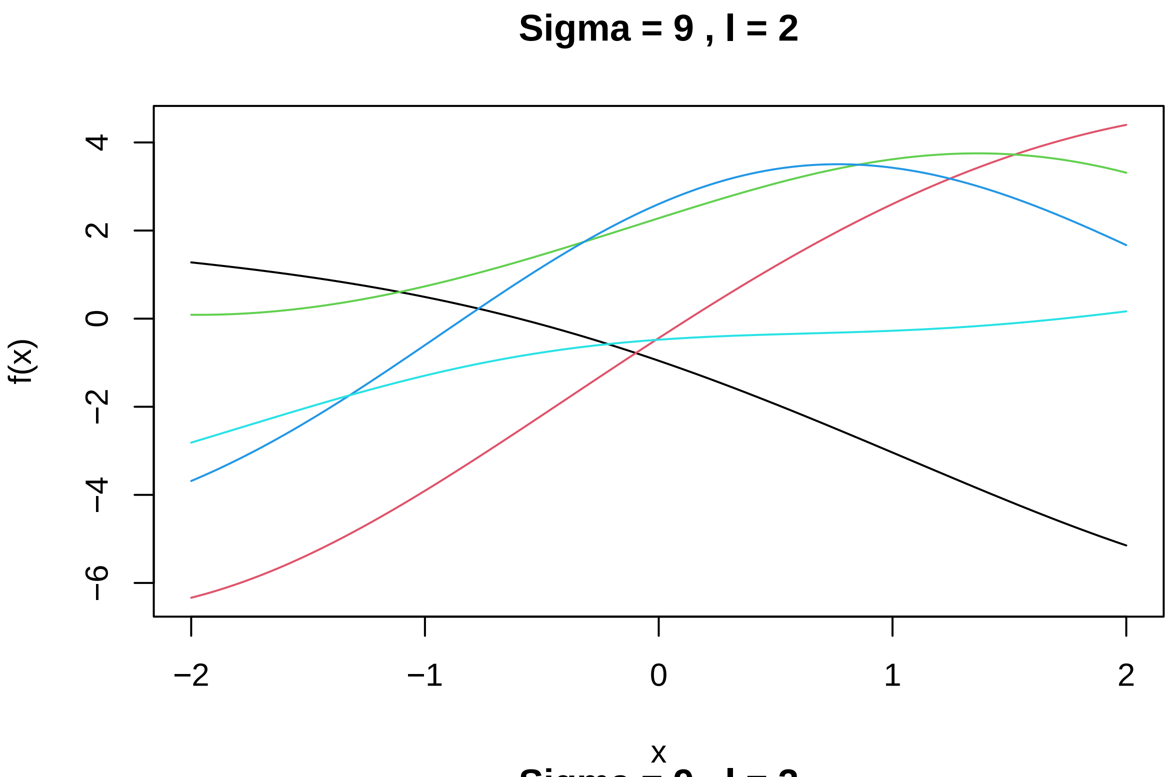

Sigma = 1 , l = 3
f(x)
x
0.0
−1.0
−2.0
−2
−1
0
1
2

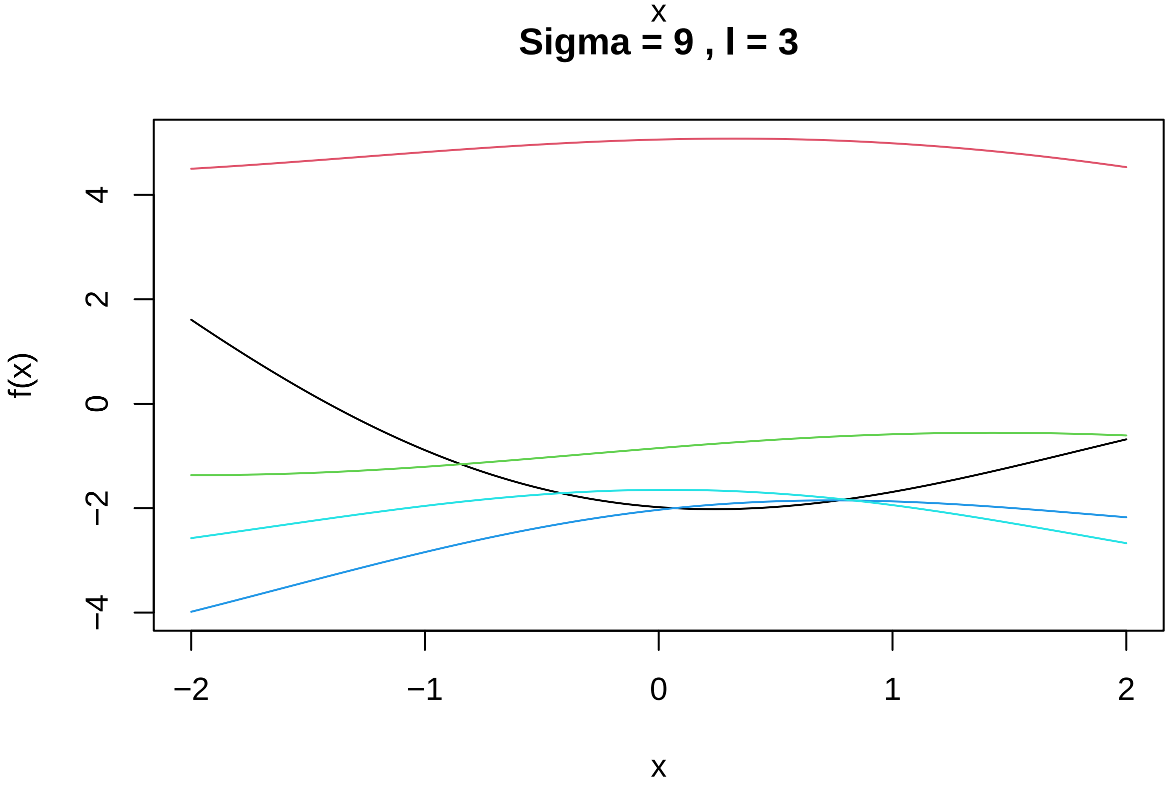

Sigma = 4 , l = 1
f(x)
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
x


Sigma = 4 , l = 2
f(x)
4
3
2
1
0
−1
−2
−2
−1
0
1
2
x


Sigma = 4 , l = 3
f(x)
x
4
3
2
1
0
−1
−2
−2
−1
0
1
2


Sigma = 9 , l = 1
f(x)
x
6
4
2
0
−2
−4
−6
−2
−1
0
1
2





Task 5

Simulate $n_{\text{train}} = 200$ and $n_{\text{test}} = 200$ covariates $\{x_i\}_{i=1}^{200}$ independently from a standard normal distribution. Then, simulate $y_i \mid x_i \sim N(2 * x_i + 2 * sin(\pi x_i), 2)$ for both the training and testing covariates.

Under the zero-mean Gaussian process prior, the *posterior predictive distribution* for y_{test} at x_{test}

is given by

$$y_{\text{test}} \sim \text{Multivariate Normal}(\mu^*, \Sigma^* + \sigma^2 \text{I}_{200\times 200}),$$

where

$$\mu^* = K(x_{\text{test}}, x_{\text{train}})(K(x_{\text{train}}, x_{\text{train}}) + \sigma^2 \text{I}_{n_{\text{train}}\times n_{\text{train}}})^{-1} y_{\text{train}}$$

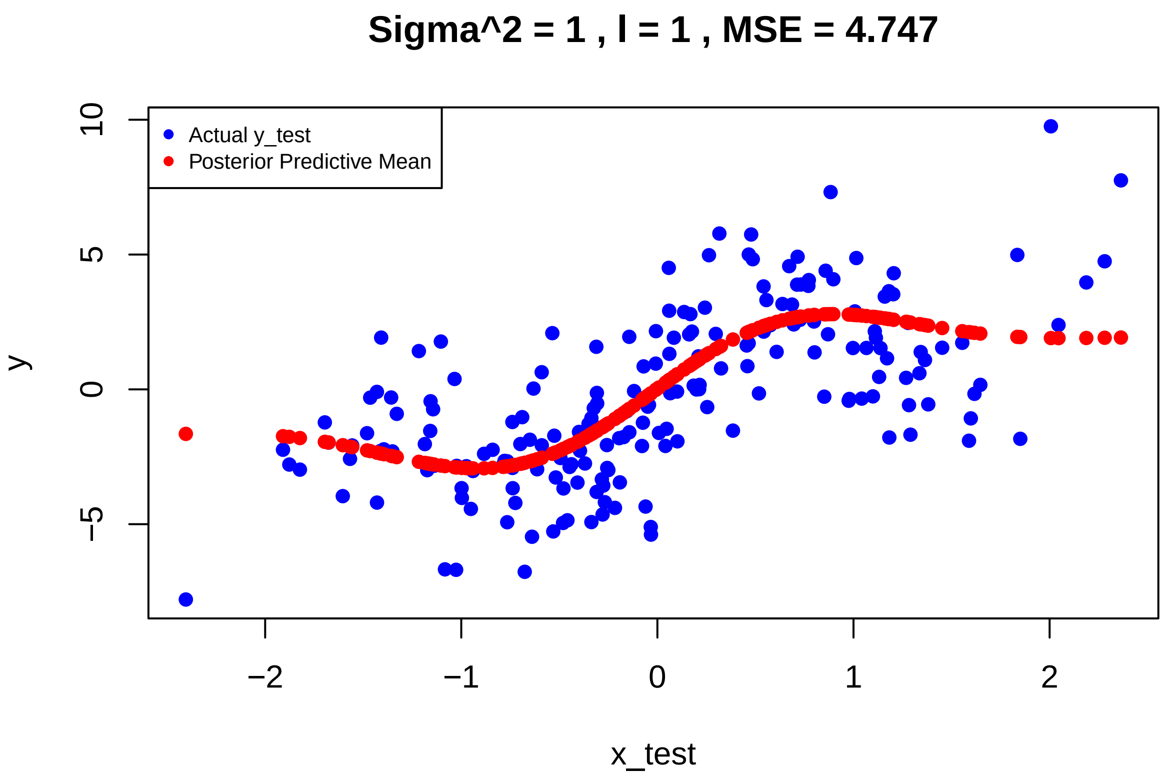
and

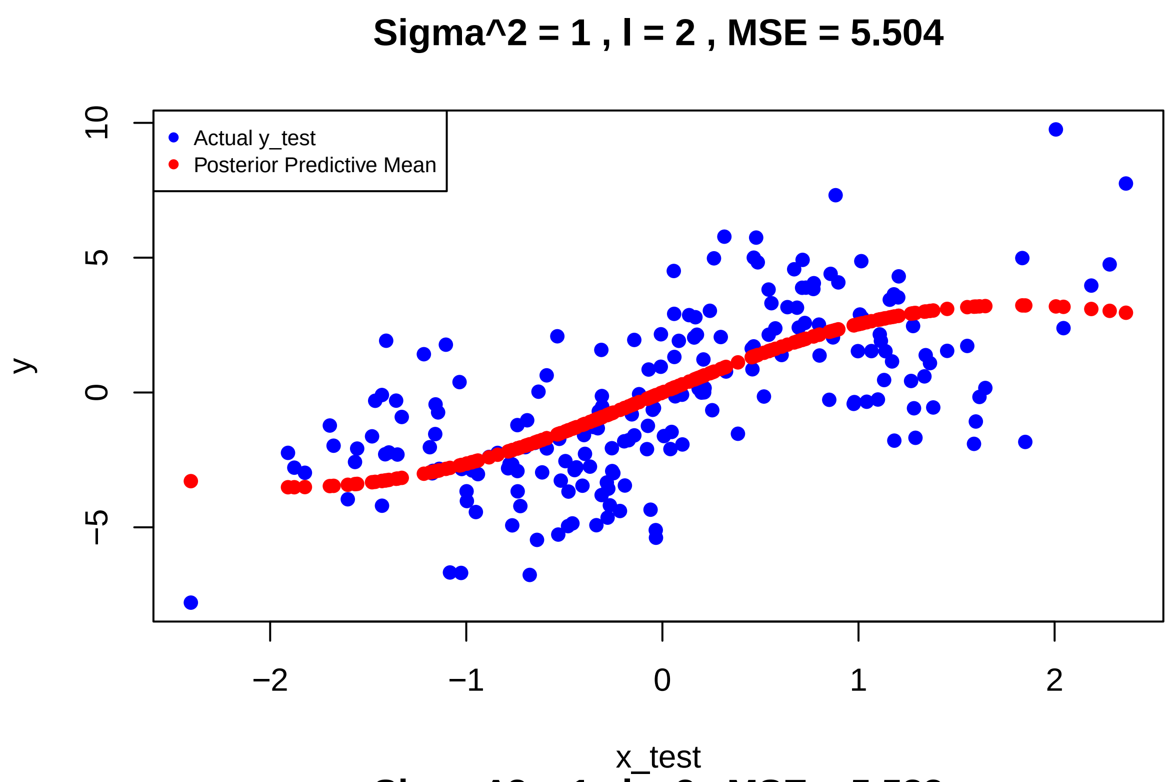
$$\Sigma^* = K(x_{\text{test}}, x_{\text{test}}) + K(x_{\text{test}}, x_{\text{train}})(K(x_{\text{train}}, x_{\text{train}} + \sigma^2 \text{I}_{200\times 200}))^{-1} K(x_{\text{train}}, x_{\text{test}}).$$

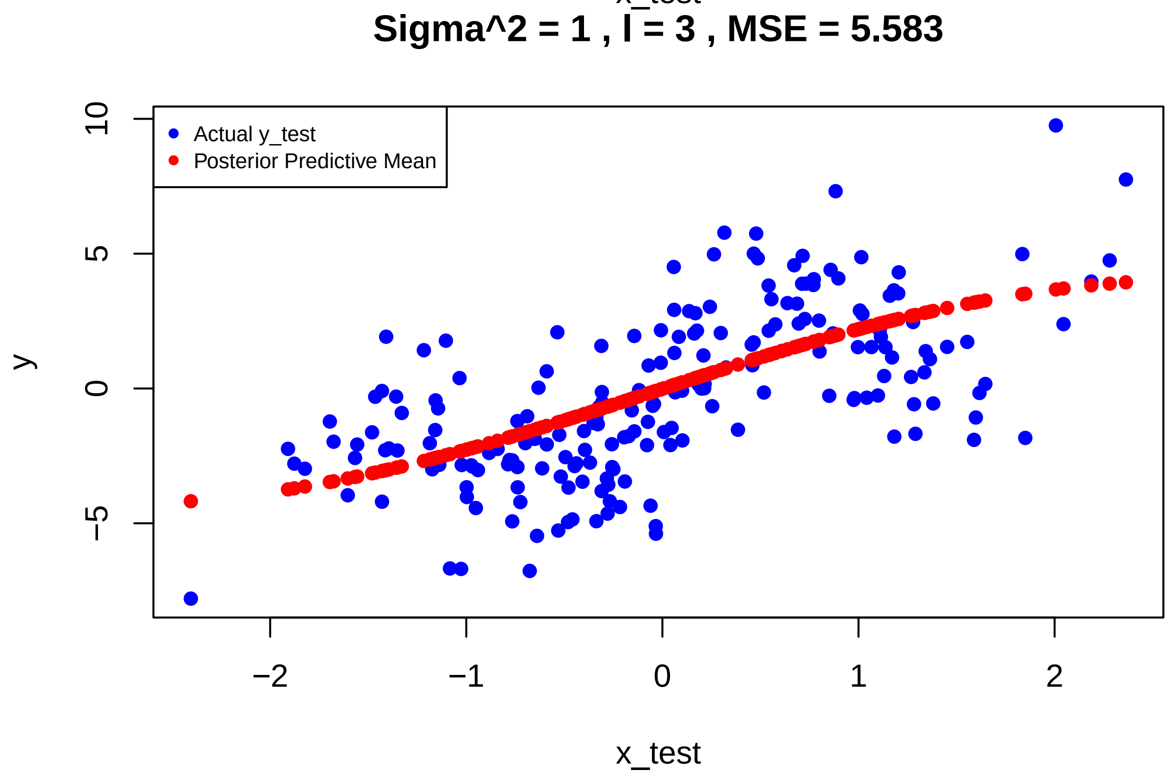
Based on your simulations above $\sigma^2 = 2$.

Across different combinations of σ_f, ℓ for your covariance kernel, use the Cholesky Decomposition of the Σ^* to simulate 100 posterior predictive samples of y_{test}. Your resulting matrix should be 200×100 for each combination of parameters. Plot the posterior predictive mean and compare to the actual testing values. Which combination of hyperparameters provides the best predictions?

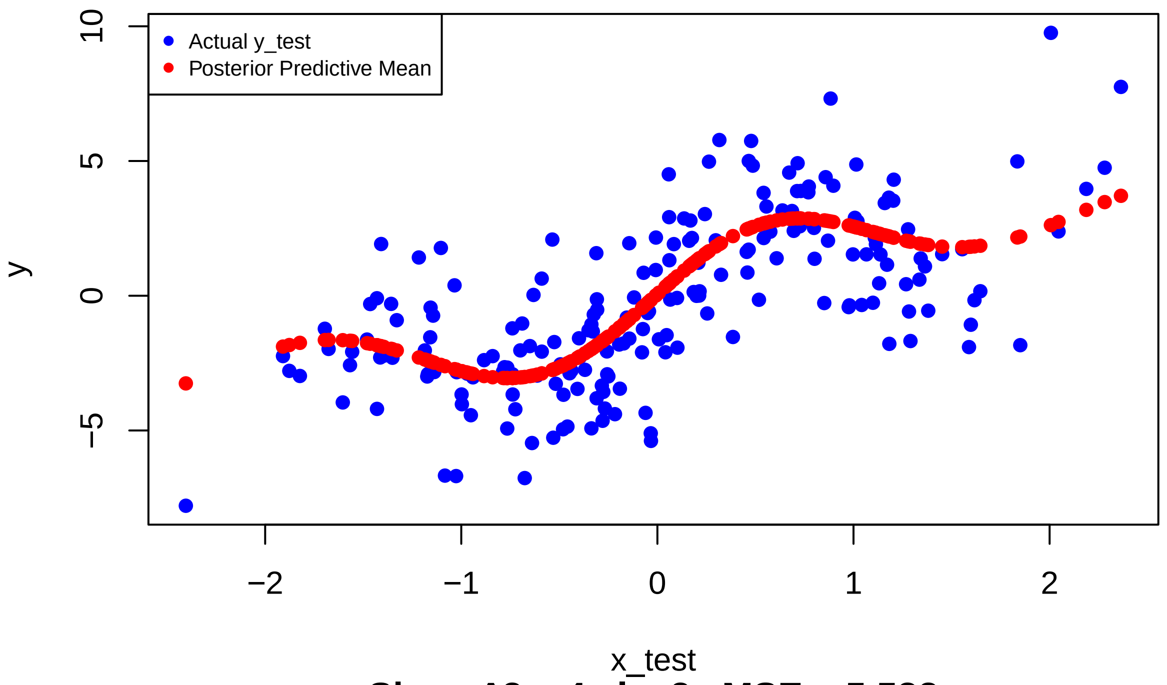
```r
library(MASS)
set.seed(4210)

n_train<-200
n_test<-200
Sigma_square_task_5<-2
x_train<-rnorm(n_train)
x_test<-rnorm(n_test)

y_train<-rnorm(n_train,2*x_train+2*sin(pi*x_train),2)
y_test<-rnorm(n_test,2*x_test+2*sin(pi*x_test),2)

#construct more flexible function SE_task_5, return K
SE_task_5<-function(a,b,x_1,x_2,n){
  Sigma_square<- a
  l<-b
  k<-matrix(,nrow=n,ncol=n) #create an empty 100*100 matrix k
  for(i in 1:n){ #row i
    for(j in 1:n){ #col j
      k[i,j]<-Sigma_square*exp(-(x_1[i]-x_2[j])^2/(2*l^2))
    }
  }
  return(k)
}
Sigma_task4<-c(1,4,9)
l_task4<-c(1,2,3)

#different combinations of  ,
for(i in 1:3){
  for(j in 1:3){

#compute  *
posterior_predictive_mean<-SE_task_5(Sigma_task4[i],l_task4[j],x_test,x_train,n_test)%*%solve(
#compute Sigma*
posterior_predictive_Sigma<-SE_task_5(Sigma_task4[i],l_task4[j],x_test,x_test,n_test)+SE_task_5

t_y_result<-mvrnorm(100,t(posterior_predictive_mean),Sigma = posterior_predictive_Sigma+Sigma_s

y_result<-t(t_y_result)

diff_task_5<-mean((posterior_predictive_mean - y_test)^2)

#plot the posterior predictive mean and compare to the actual testing values
plot(x_test, y_test, col="blue", pch=16,xlab="x_test", ylab="y", main=paste("Sigma^2 =", Sigma_
points(x_test, posterior_predictive_mean, col="red", pch=16)
legend("topleft", legend=c("Actual y_test", "Posterior Predictive Mean"),col=c("blue", "red"),


  }
}
```



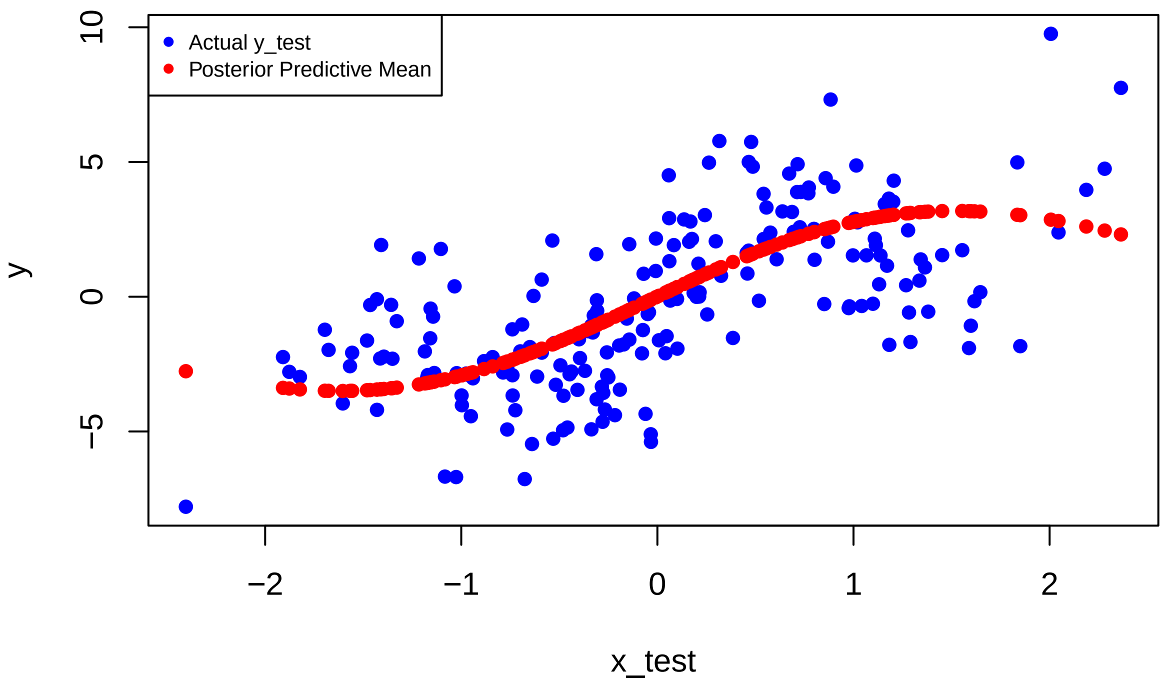

Sigma^2 = 1 , l = 2 , MSE = 5.504
Actual y_test
Posterior Predictive Mean
y
x_test
10
5
0
−5
−2
−1
0
1
2

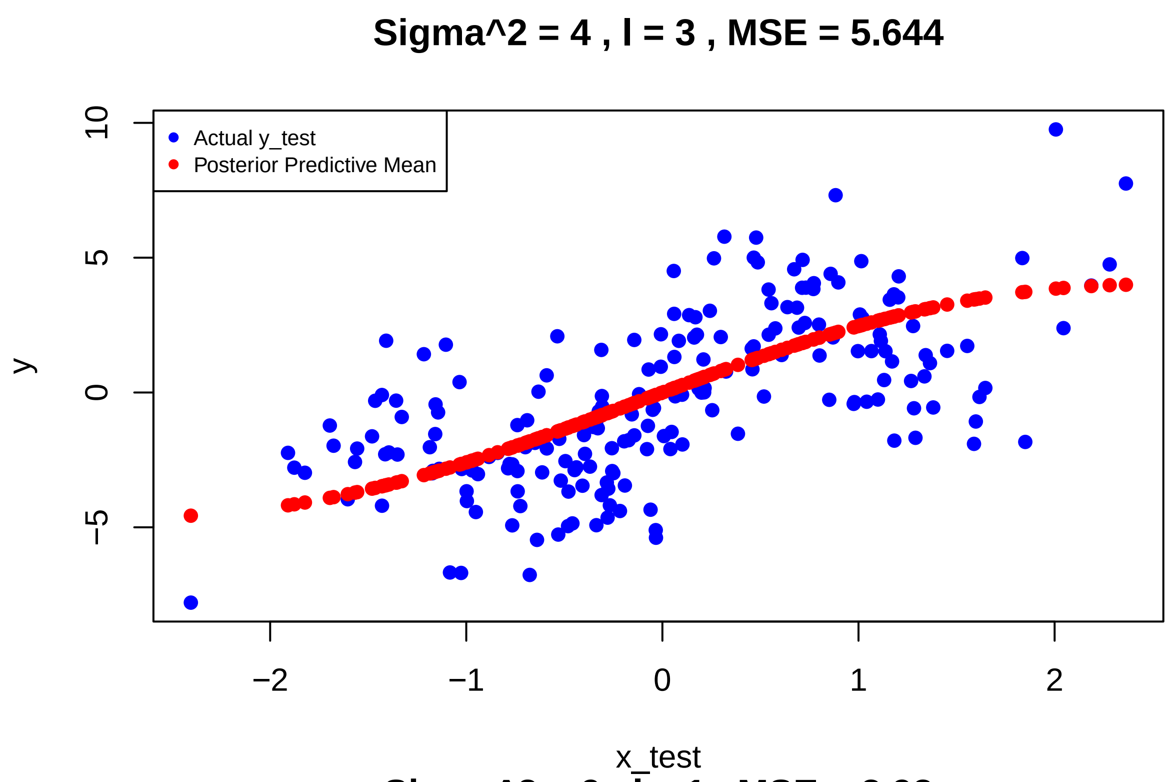

Sigma^2 = 1 , l = 3 , MSE = 5.583
Actual y_test
Posterior Predictive Mean
y
x_test
10
5
0
−5
−2
−1
0
1
2

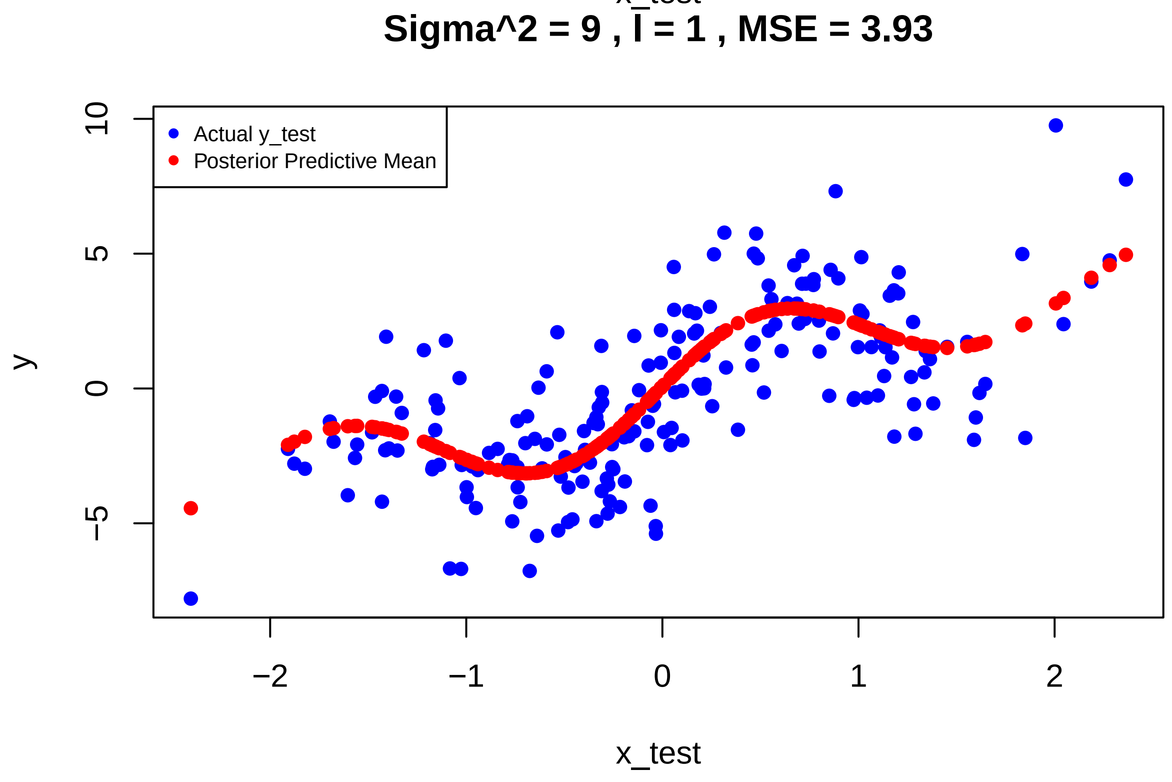
Sigma^2 = 4 , l = 1 , MSE = 4.175

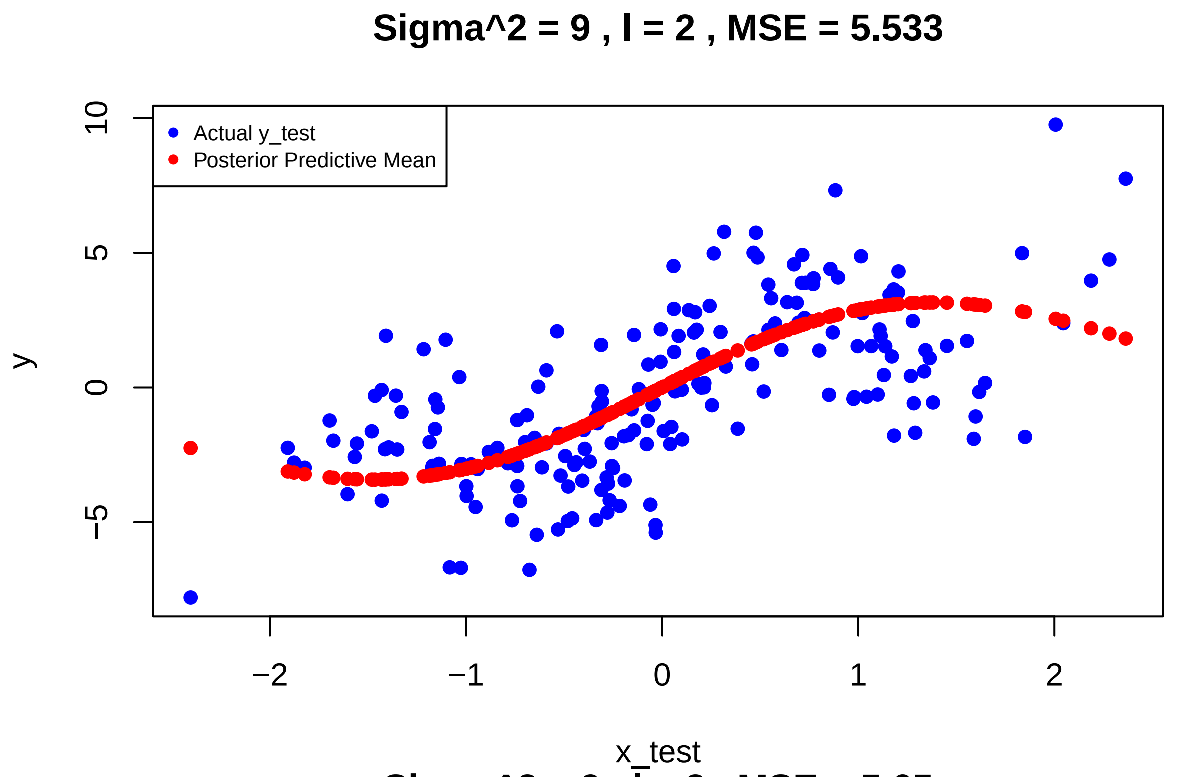


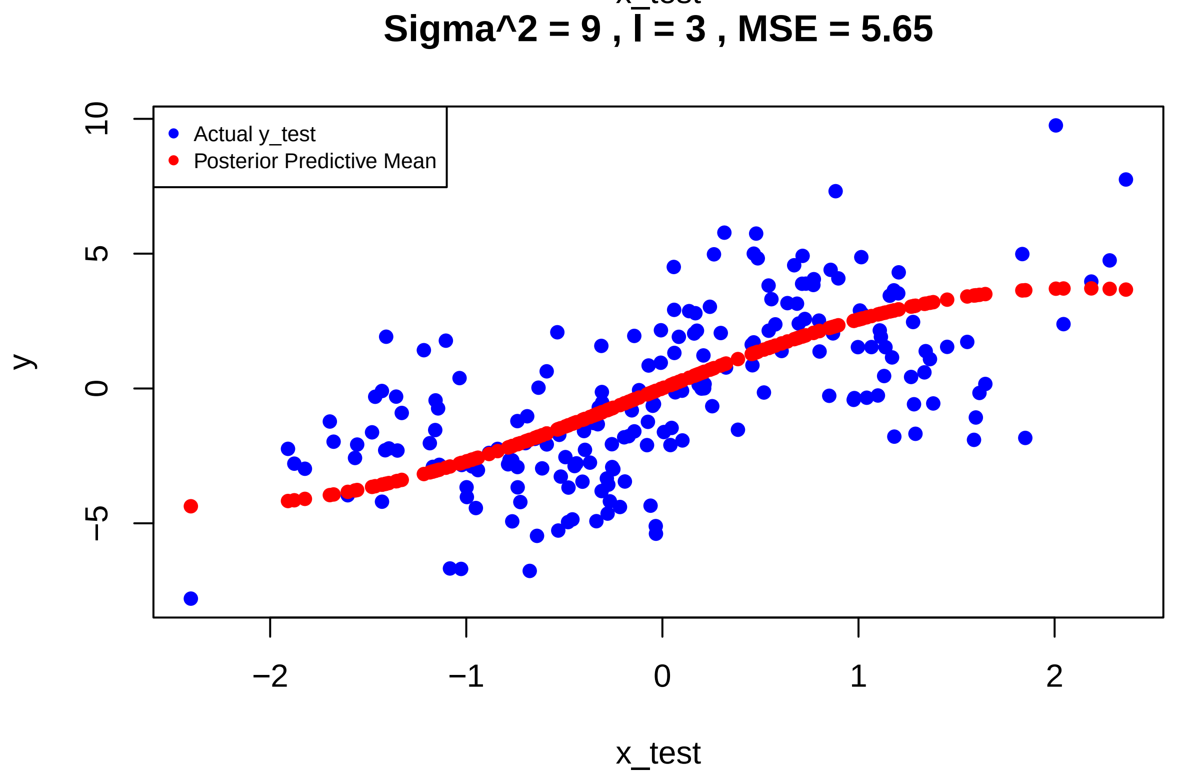
Sigma^2 = 4 , l = 2 , MSE = 5.532




Sigma^2 = 4 , l = 3 , MSE = 5.644
Actual y_test
Posterior Predictive Mean
y
x_test
10
5
0
−5
−2
−1
0
1
2


Sigma^2 = 9 , l = 1 , MSE = 3.93
Actual y_test
Posterior Predictive Mean
y
x_test
10
5
0
−5
−2
−1
0
1
2





```
#Sigma^2=9,l=1 is the best combination of hyperparameters
```

Problem 4 — One-dimensional root finding: bisection vs. Newton (30 pts)

(a) Show that the bisection algorithm has linear convergence.

Proof. Let the true root be x^*, and the midpoint at iteration k be c_k.
Define the error as

$$e_k = |c_k - x^*|.$$

Since the root always lies in the current interval $[a_k, b_k]$, we have

$$e_k \le \frac{b_k - a_k}{2}.$$

The bisection method halves the interval at each step:

$$b_{k+1} - a_{k+1} = \tfrac{1}{2}(b_k - a_k).$$

Therefore, the error satisfies

$$e_{k+1} \le \tfrac{1}{2} e_k.$$

Now, consider the function

$$f(x) = x^5 - x - 1$$

.

(b) Implement a **bisection** solver `bisect(f, a, b, tol, maxit)` that returns the root approximation and the iteration log. Use $[a, b] = [1, 2]$. Include **stopping criteria** by (i) interval width and (ii) function value.

```
biect<-function(f,a,b,tol,maxit){
  stopifnot(is.function(f), is.numeric(a), is.numeric(b), a < b)
  f_a<-f(a)
  f_b<-f(b)
  log<-NULL
  if (f_a*f_b>= 0) stop("Interval does not have a root: f(a)*f(b) >= 0")
  for(i in 1:maxit){
    c<-(a+b)/2
    f_c<-f(c)
    log <- rbind(log, data.frame(iter=i, a=a, b=b, c=c, f_c=f_c))
    if(abs(a-b)/2<tol | abs(f(c))<tol){
      return(list(root=c, log=log))
    }
    if(f(a)*f(c)<0){
      a<-a
      b<-c
      f_b<-f_c
    }
    else{
      a<-c
      b<-b
      f_a<-f_c
    }
  }
  return(list(root=c, log=log))
}

biect(function(x)x^5-x-1,1,2,0.001,500)
```

```
## $root
## [1] 1.166992
##
## $log
##    iter        a        b        c          f_c
## 1     1 1.000000 2.000000 1.500000  5.093750000
## 2     2 1.000000 1.500000 1.250000  0.801757812
## 3     3 1.000000 1.250000 1.125000 -0.322967529
## 4     4 1.125000 1.250000 1.187500  0.173892021
## 5     5 1.125000 1.187500 1.156250 -0.089639038
## 6     6 1.156250 1.187500 1.171875  0.038197125
## 7     7 1.156250 1.171875 1.164062 -0.026683718
## 8     8 1.164062 1.171875 1.167969  0.005513586
## 9     9 1.164062 1.167969 1.166016 -0.010645540
## 10   10 1.166016 1.167969 1.166992 -0.002581134
```

(c) Implement **Newton's method** `newton(f, fprime, x0, tol, maxit)` with $x_0 = 1$. Track iterates and residuals.

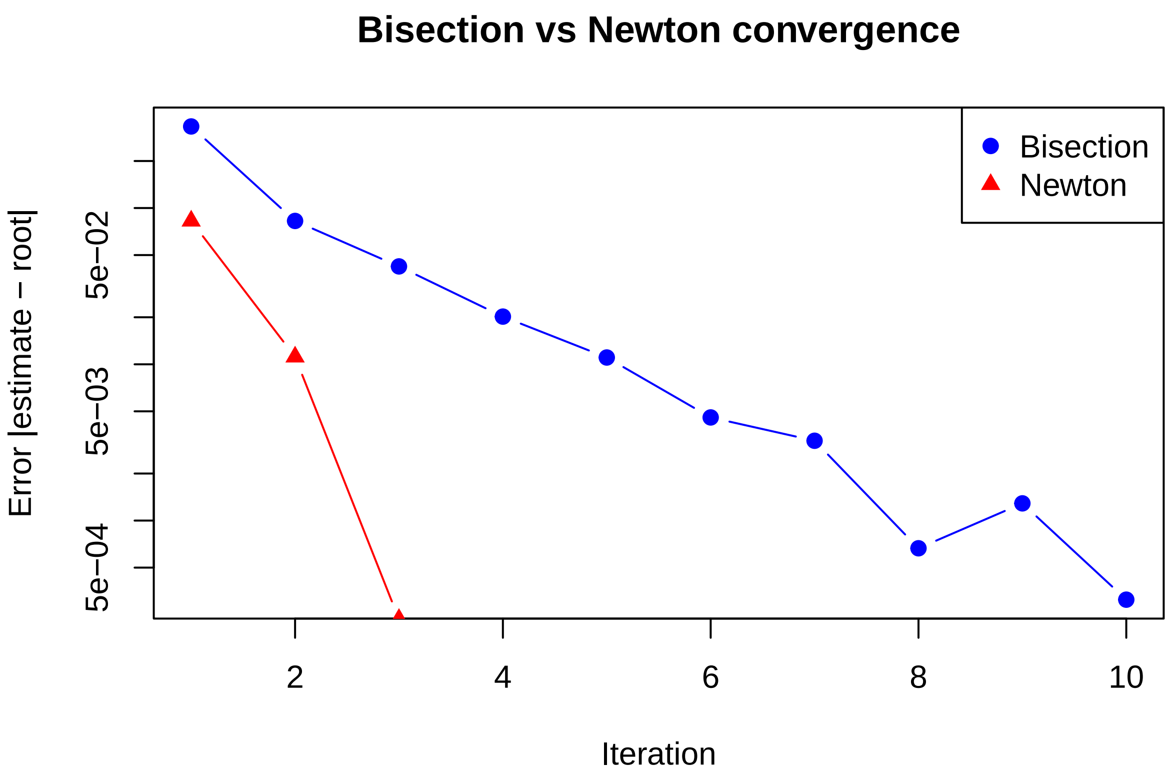
```
newton<-function(f,fprime,x0,tol,maxit){
  x<-x0
  log<-NULL
  for(i in 1:maxit){
    if(abs(f(x))<tol){
      log<-rbind(log,data.frame(iter=i,x=x,f_x=f(x)))
      return(list(root=x, log=log))
      break
    }
    x<-x-f(x)/fprime(x)
    log<-rbind(log,data.frame(iter=i,x=x,f_x=f(x)))
  }
  return(list(root=x, log=log))
}
newton(function(x)x^5-x-1,function(x)5*x^4-1,1,0.001,500)
```

```
## $root
## [1] 1.167304
##
## $log
##   iter        x          f_x
## 1    1 1.250000 8.017578e-01
## 2    2 1.178459 9.440284e-02
## 3    3 1.167537 1.934299e-03
## 4    4 1.167304 8.661230e-07
## 5    5 1.167304 8.661230e-07
```

(d) Empirically assess convergence rate. Which algorithm converges more quickly?

```
root_ <- uniroot(function(x)x^5-x-1, lower=1, upper=2, tol=1e-14)$root


biect<-function(f,a,b,tol,maxit){
  stopifnot(is.function(f), is.numeric(a), is.numeric(b), a < b)
  f_a<-f(a)
  f_b<-f(b)
  path_biect<-NULL
  if (f_a*f_b>= 0) stop("Interval does not have a root: f(a)*f(b) >= 0")
  for(i in 1:maxit){
    c<-(a+b)/2
    f_c<-f(c)
    path_biect <- rbind(path_biect, data.frame(iter=i, a=a, b=b, c=c, f_c=f_c,e=abs(c-root_)))
    if(abs(a-b)/2<tol | abs(f(c))<tol){
      return(list(root=c, path_biect=path_biect))
    }
    if(f(a)*f(c)<0){
      a<-a
      b<-c
      f_b<-f_c
    }
    else{
      a<-c
      b<-b
      f_a<-f_c
    }
  }
  return(list(root=c, path_biect=path_biect))
}
b <- biect(function(x)x^5-x-1,1,2,0.001,500)


newton<-function(f,fprime,x0,tol,maxit){
  x<-x0
```

```r
  path_newton<-NULL
  for(i in 1:maxit){
    if(abs(f(x))<tol){
      path_newton<-rbind( path_newton,data.frame(iter=i,x=x,f_x=f(x),e=abs(x-root_)))
      return(list(root=x,  path_newton= path_newton))
      break
    }
    x<-x-f(x)/fprime(x)
     path_newton<-rbind( path_newton,data.frame(iter=i,x=x,f_x=f(x),e=abs(x-root_)))
  }
  return(list(root=x,  path_newton= path_newton))
}
n<-newton(function(x)x^5-x-1,function(x)5*x^4-1,1,0.001,500)



plot(b$path_biect$iter, b$path_biect$e, type="b", log="y", col="blue", pch=19, xlab="Iteration"
lines(n$path_newton$iter, n$path_newton$e, type="b", col="red", pch=17)
legend("topright", legend=c("Bisection", "Newton"), col=c("blue", "red"), pch=c(19,17))
```



```r
#Newton converges more quickly
```