Exam solutions

Task 1

D-separation

Load libraries and data.

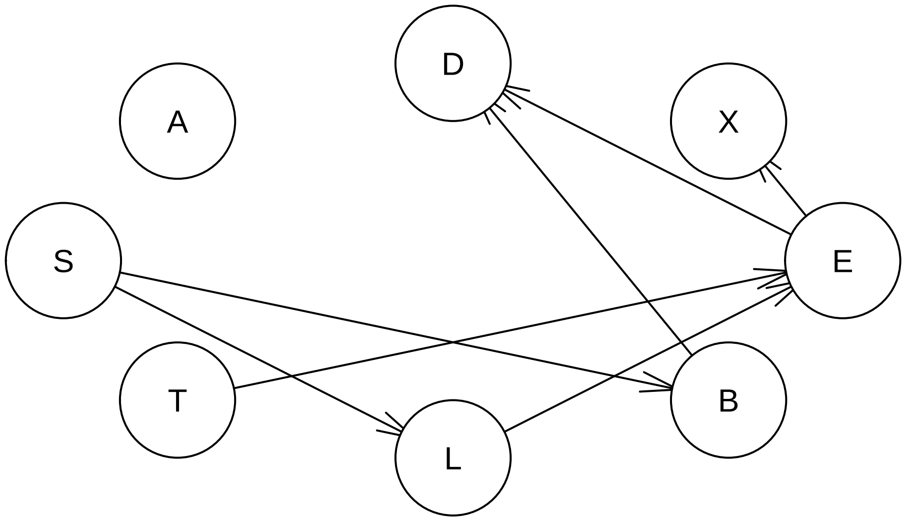
```r
suppressPackageStartupMessages(library(bnlearn))
suppressPackageStartupMessages(library(gRain))

data("asia")
```

First, let us learn structure and parameters.

```r
set.seed(12345)
bn <- hc(x=asia, score="bde", iss=1)
bn_fitted <- bn.fit(bn, asia)
compiled_bn <- compile(as.grain(bn_fitted))
```

Now identify a d-separation by plotting the learned structure

```r
plot(bn)
```



It looks like D and X are d-separated by E, because arrows on the path from D to x meet tail-to-tail at the node E. Let us check whether this carries to the probability distribution as well.

First we query $P(D \mid E)$ and $P(X \mid E)$, then we check whether $P(D \mid E, X) = P(D \mid E)$ and $P(X \mid E, D) = P(X \mid E)$, because if they are equal, that implies $D \perp X \mid E$, which is the same as D is d-separated from X by E.

```r
# $P(D \mid E)$ and $P(X \mid E)$
bn_w_evidence <- setEvidence(compiled_bn, c("E"), c("yes"))
querygrain(bn_w_evidence, c("D", "X"))
```

```
## $D
## D
##        no       yes
## 0.1912653 0.8087347
##
## $X
## X
##          no         yes
## 0.005405405 0.994594595
```

```
#  $P(X \mid E, D)$
bn_w_evidence <- setEvidence(compiled_bn, c("E", "D"), c("yes", "yes"))
querygrain(bn_w_evidence, c("D", "X"))
```

```
## $X
## X
##          no         yes
## 0.005405405 0.994594595
```

```
# $P(D \mid E, X)$
bn_w_evidence <- setEvidence(compiled_bn, c("E", "X"), c("yes", "yes"))
querygrain(bn_w_evidence, c("D", "X"))
```

```
## $D
## D
##        no       yes
## 0.1912653 0.8087347
```

Since $P(D \mid E, X) = P(D \mid E)$ and $P(X \mid E, D) = P(X \mid E)$, we conclude that indeed $D \perp X \mid E$ in the probability distribution, just as in the graph.

For a sense check, let us see if D is dependent on X without E:

```
# $P(D), P(X)$
querygrain(compiled_bn, c("D", "X"))
```

```
## $D
## D
##        no       yes
## 0.5309833 0.4690167
##
## $X
## X
##        no       yes
## 0.8859915 0.1140085
```

```
# $P(D \mid X)$
bn_w_evidence <- setEvidence(compiled_bn, c("X"), c("yes"))
querygrain(bn_w_evidence, c("D", "X"))
```

```
## $D
## D
##        no       yes
## 0.3206243 0.6793757
```

```r
# $P(X \mid D)$
bn_w_evidence <- setEvidence(compiled_bn, c("D"), c("yes"))
querygrain(bn_w_evidence, c("D", "X"))
```

```
## $X
## X
##        no       yes
## 0.8348574 0.1651426
```

We see that $P(D \mid X) \neq P(D)$ and $P(X \mid D) \neq P(X)$, therefore, without E they are dependent.

Fraction of essential DAGs

DAG is not equivalent to any other DAG (hence, essential), if none of the links can be reversed while keeping the same probability distribution. In terms of code below, it means that a DAG is essential, if the corresponding essential DAG (DAG with all arrows that can be removed removed produced by `cpdag()`) has no links without direction.

```r
n <- 10000
counter <- 0
for (i in 1:n) {
  dag <- random.graph(nodes = c("A", "B", "C", "D", "E"),
                      method = "ic-dag",
                      burn.in = 10000)
  cpdag <- cpdag(dag)
  if (all.equal(dag, cpdag) == TRUE) {
    counter <- counter + 1
  }
}
fraction_of_essential <- counter / n
```

Thus, the approximate fraction of essential DAGs among dags is 0.0676.

Task 2

First we specify the robot. Assume that once it hits state 100, it cannot go back to 1 (path is not round).

```
library(HMM)

States <- 1:100
Symbols <- c("door", "no door")

startProbs <- rep(1/100, 100)

transProbs <- diag(0.1, 100) +
  diag(0.9, 100)[, c(100, 1:99)]
# Correct for not being able to move from 100 to 1
transProbs[100, 1] <- 0
transProbs[100, 100] <- 1

# Most cases: no door
emissionProbs <- t(rbind(
  rep(0.1, 100),
  rep(0.9, 100)
))

# Cases with doors
door_segments <- c(10, 11, 12, 20, 21, 22, 30, 31, 32)
emissionProbs[c(door_segments), 1] <- 0.9
emissionProbs[c(door_segments), 2] <- 0.1

robot_hmm <- initHMM(States=States,
                     Symbols=Symbols,
                     startProbs=startProbs,
                     transProbs=transProbs,
                     emissionProbs=emissionProbs)
```

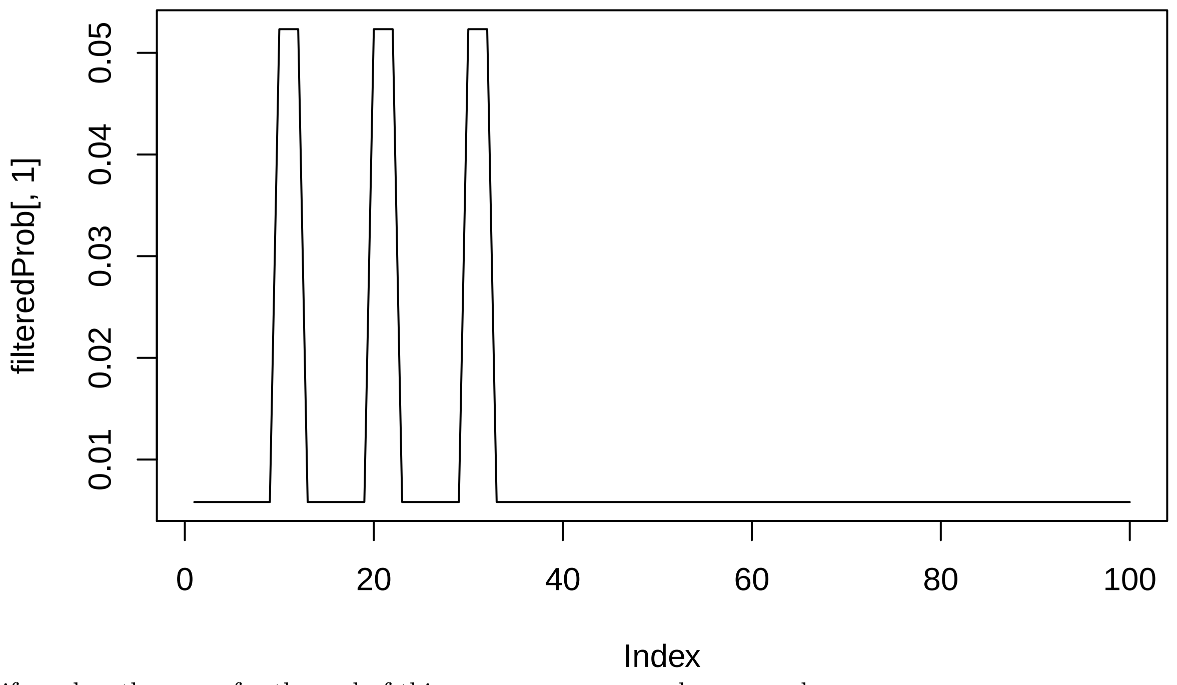
Sequence that will return a unimodal distribution is the one that has witnessed all three doors. This happens because when we observe the third door, then we know it is the third door (disregarding noisy measurements for a moment), since our robot can move only in one direction. While if we have observed, for example, 2 doors, then this can be both the second or the third door. Hence, the distribution will be with 2 equal peaks.

The simplest such sequence is the one that we would observe by moving from position 10 to 32. It corresponds to the readings specified below (with no noise, because this is just an illustration and we want to be obvious).

```
observation <- rep("no door", 23)
observation[c(1,2,3,11,12,13,21,22,23)] <- "door"
filteredProbLog <- forward(robot_hmm, observation)
filteredProb <- prop.table(exp(filteredProbLog), 2)
```

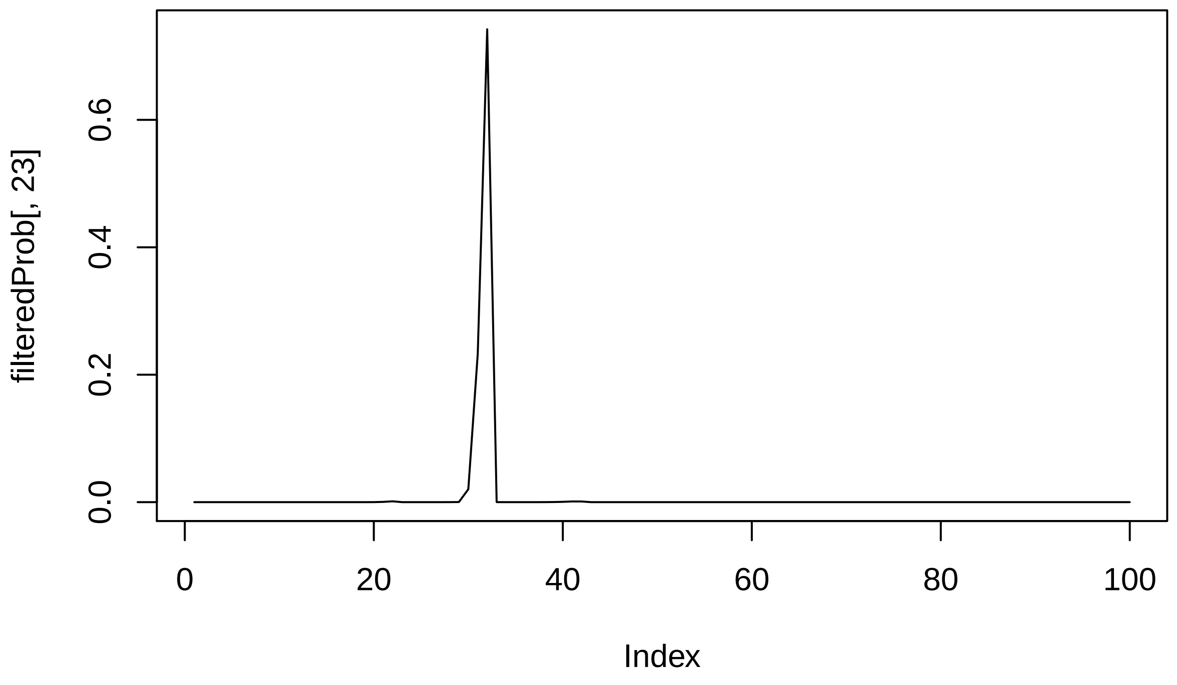
And indeed, if we plot the distribution of $p(z_t \mid x_{1:t})$ at $t = 1$ we see 3 modes:

```
plot(filteredProb[, 1], type="l")
```



while if we dew the same for the end of this sequence, we see only one mode:

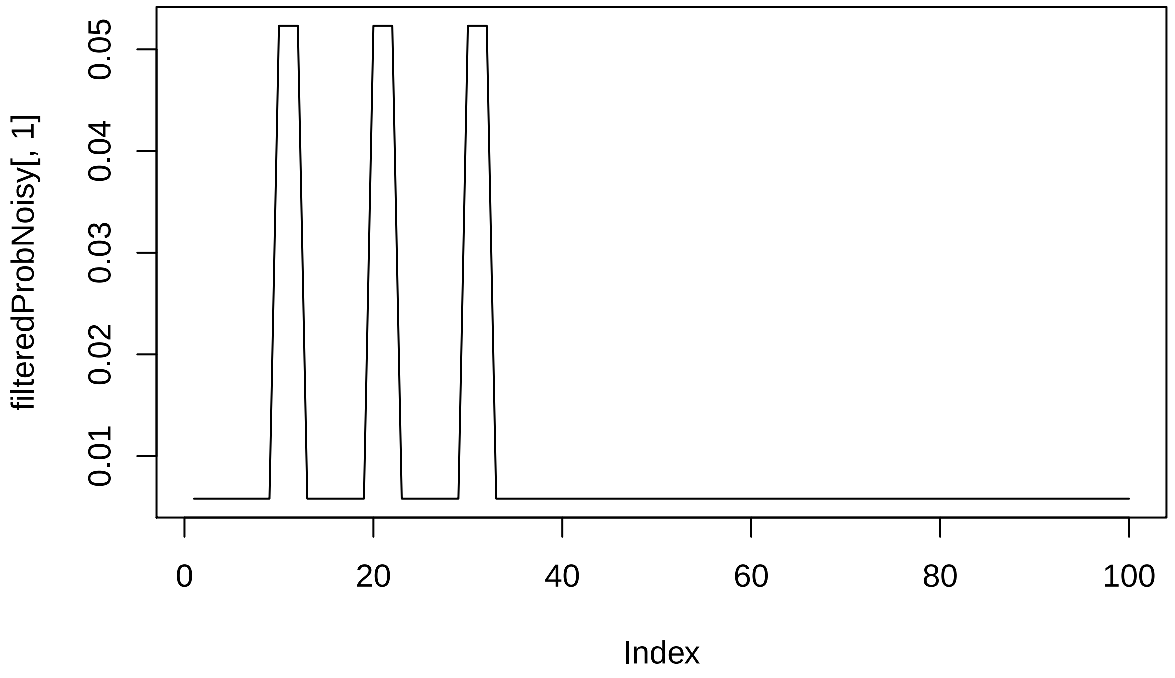
```r
plot(filteredProb[, 23], type="l")
```



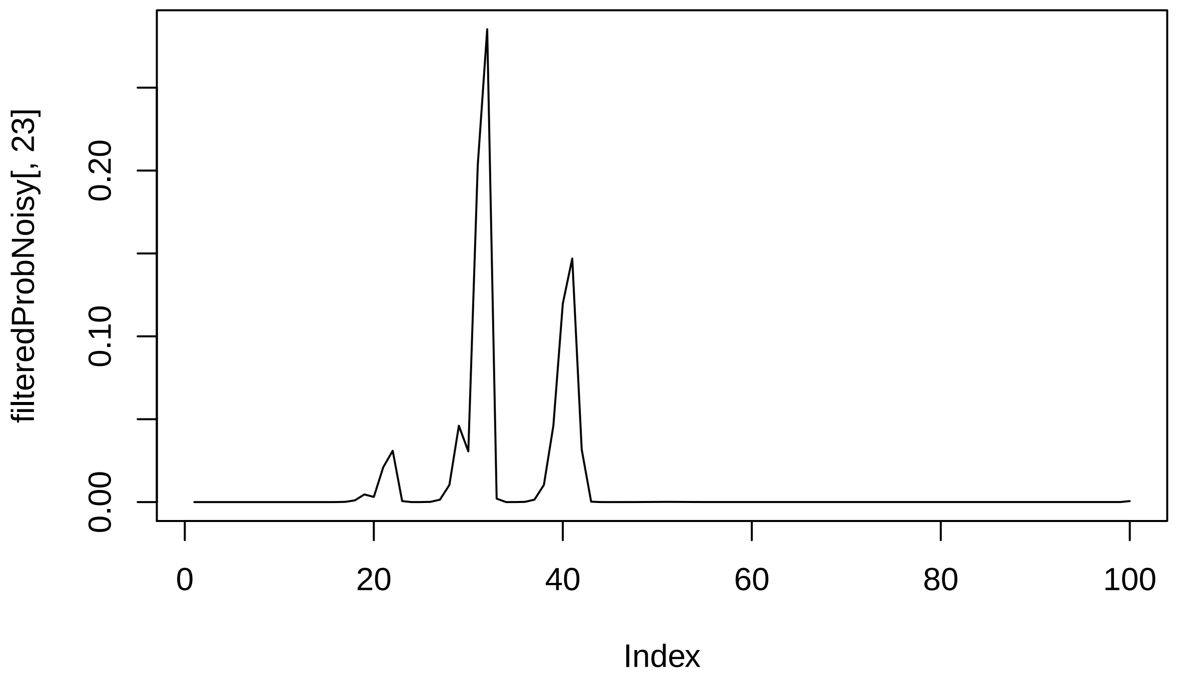
If we add noise, the overall result is the same, but the distribution becomes not exactly unimodal, but rather with one pronounced mode, and 2 significantly shorter modes:

```r
# Add noise
observation_noisy <- rep(NA, 23)
for (i in 1:length(observation)) {
  if (runif(1,0,1) < 0.1) {
    # Flip the observation
    if (observation[i] == "door") {
      observation_noisy[i] <- "no door"
    } else {
      observation_noisy[i] <- "door"
    }
  } else {
    observation_noisy[i] <- observation[i]
  }
}

filteredProbLogNoisy <- forward(robot_hmm, observation_noisy)
filteredProbNoisy <- prop.table(exp(filteredProbLogNoisy), 2)
```

```r
plot(filteredProbNoisy[, 1], type="l")
```



```r
plot(filteredProbNoisy[, 23], type="l")
```


filteredProbNoisy[, 23]
0.00
0.10
0.20
0
20
40
60
80
100
Index

Task 3

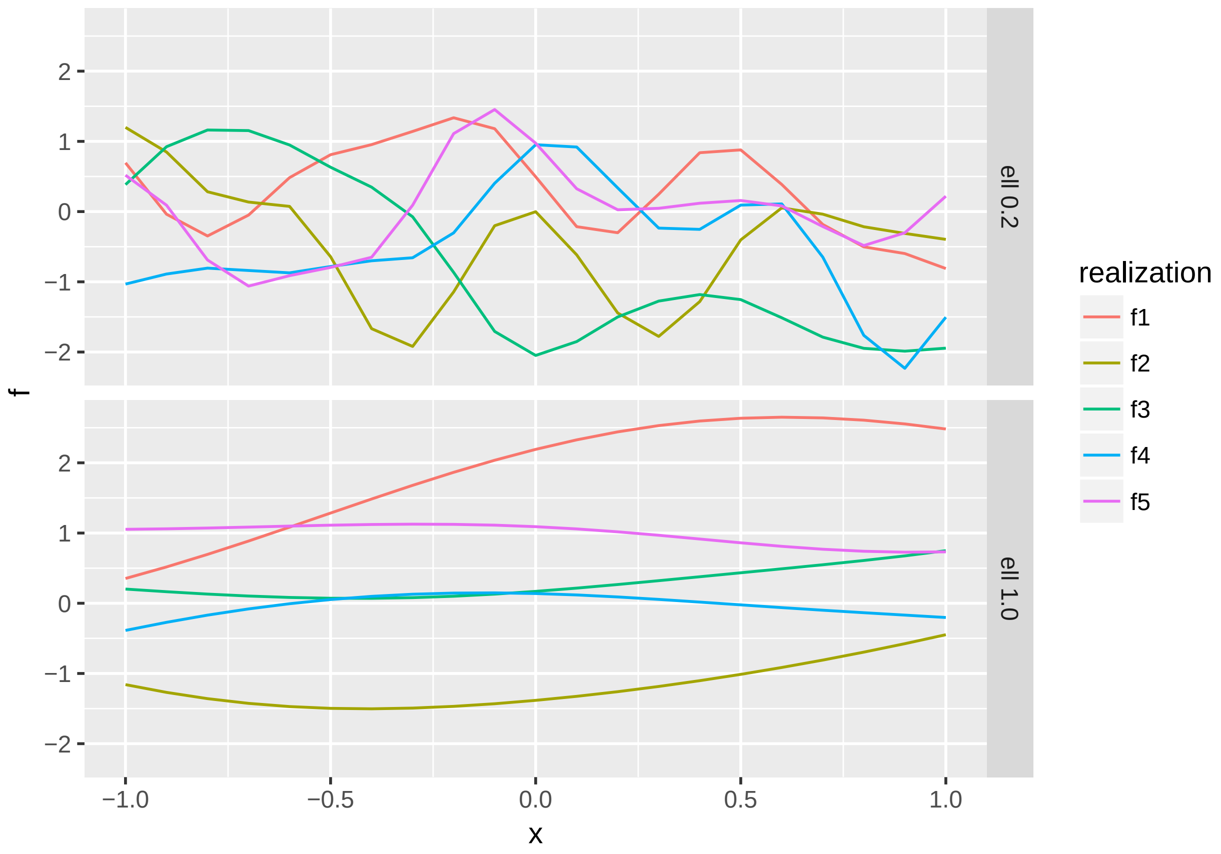
(a)

```r
suppressPackageStartupMessages(library(kernlab))
suppressPackageStartupMessages(library(mvtnorm))
suppressPackageStartupMessages(library(ggplot2))
suppressPackageStartupMessages(library(tidyr))
suppressPackageStartupMessages(library(dplyr))

# Squared exponential kernel
k <- function(sigmaf = 1, ell = 1)
{
  rval <- function(x, y = NULL)
  {
    r = sqrt(crossprod(x-y))
    return(sigmaf^2*exp(-r^2/(2*ell^2)))
  }
  class(rval) <- "kernel"
  return(rval)
}
```

First, let us plot the 5 realizations from $l = 0.2$ and from $l = 1$.

```r
# Simulate and plot 5 realizations
grid <- seq(-1, 1, by=0.1)

# Simulate for ell = 0.2
ell <- 0.2
sq_exp_k <- k(sigmaf=1, ell=ell)
K <- kernelMatrix(sq_exp_k, grid)
x <- t(rmvnorm(5, sigma=K))
plot_df <- cbind(grid, x)
colnames(plot_df) <- c("x", "f1", "f2", "f3", "f4", "f5")
plot_df %>%
  as.data.frame() %>%
  gather(realization, f, -x) ->
  plot_df_02


# Simulate for ell = 0.1
ell <- 1
sq_exp_k <- k(sigmaf=1, ell=ell)
K <- kernelMatrix(sq_exp_k, grid)
x <- t(rmvnorm(5, sigma=K))
plot_df <- cbind(grid, x)
colnames(plot_df) <- c("x", "f1", "f2", "f3", "f4", "f5")
plot_df %>%
  as.data.frame() %>%
  gather(realization, f, -x) ->
  plot_df_10
```

```
# Plot
plot_df_02 <- cbind(plot_df_02, ell="ell 0.2")
plot_df_10 <- cbind(plot_df_10, ell="ell 1.0")

plot_df <- rbind(plot_df_02, plot_df_10)

ggplot(plot_df, aes(x=x, y=f, col=realization)) +
  geom_line() +
  facet_grid(ell~.)
```



Now us compute the correlations of $f(0), f(0.1)$ and $f(0), f(0.5)$ for the both length-scales.

```
# Compute correlations
ell <- 0.2
sq_exp_k <- k(sigmaf=1, ell=ell)
K <- kernelMatrix(sq_exp_k, c(0, 0.1, 0.5))

corr_0_01_ell_02 <- K[1, 2] / sqrt(sqrt(K[1, 1]) * sqrt(K[2, 2]))
corr_0_05_ell_02 <- K[1, 3] / sqrt(sqrt(K[1, 1]) * sqrt(K[3, 3]))

ell <- 1
sq_exp_k <- k(sigmaf=1, ell=ell)
K <- kernelMatrix(sq_exp_k, c(0, 0.1, 0.5))

corr_0_01_ell_1 <- K[1, 2] / sqrt(sqrt(K[1, 1]) * sqrt(K[2, 2]))
corr_0_05_ell_1 <- K[1, 3] / sqrt(sqrt(K[1, 1]) * sqrt(K[3, 3]))
```

Thus, for $l = 0.2$:

- $Corr(f(0), f(0.1))$: 0.8824969
- $Corr(f(0), f(0.5))$: 0.0439369

And for $l = 1$:

- $Corr(f(0), f(0.1))$: 0.9950125
- $Corr(f(0), f(0.5))$: 0.8824969

Plot for realizations of $l = 1$ is significantly smoother than the plot of $l = 0.2$. Correlations for the same points are higher when we use $l = 1$ compared to $l = 0.2$.

Both plots and correlations suggest the same thing: parameter l governs the smoothness of the lines, because it ensures that the larger the l the larger the correlation between functions at any 2 points on the grid.

In particular, larger l means that the kernel function returns higher covariances for the covariance matrix of the Gaussian process. This forces the realizations from the functions to be closer together, which does not allow for function values at different points on the grid to be too far apart. Which, in its turn, implies smoother curves.

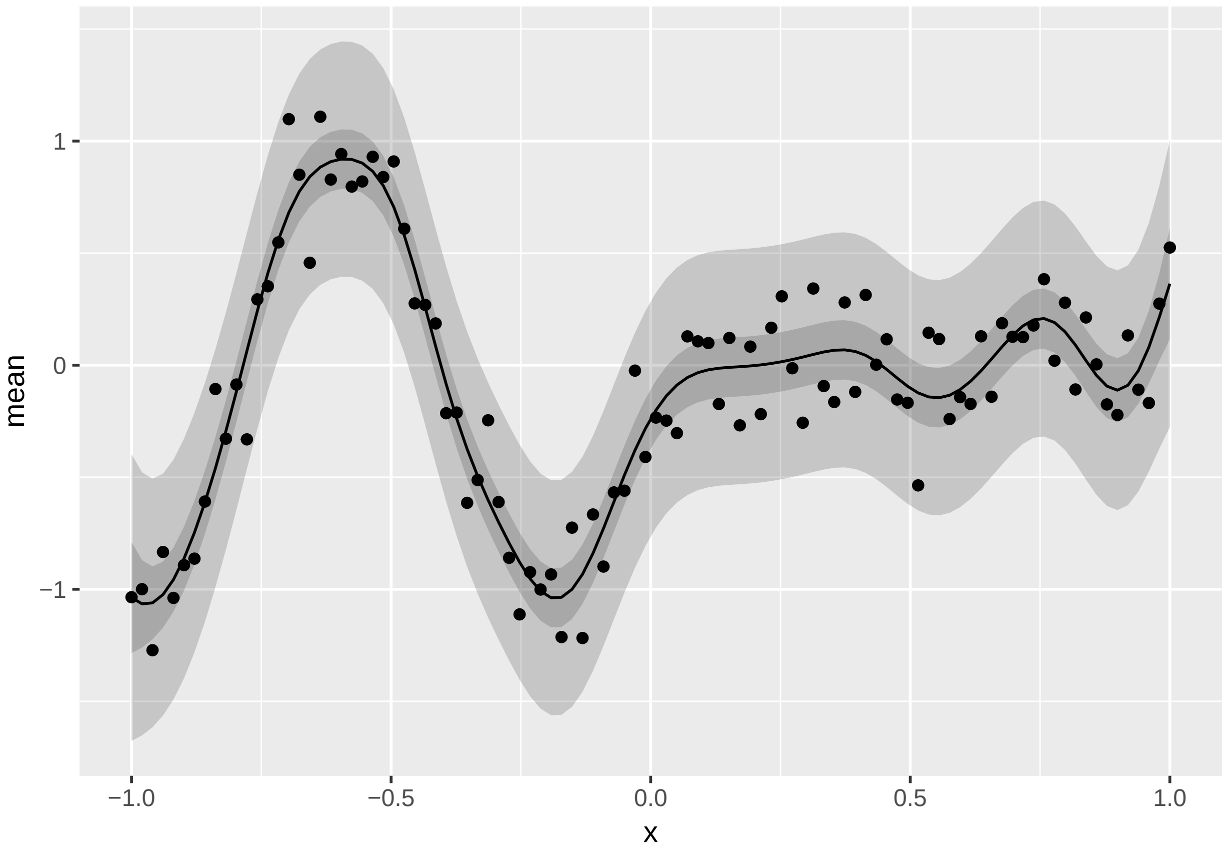
In our case, we see that for $l = 1$ the correlation between "distant" points of 0 and 0.5 is the same as the correlation between "close" points of 0 and 0.1 for $l = 0.2$.

(b)

```r
load("../data/GPdata.RData")
grid <- seq(min(x), max(x), length.out=100)

original_data_df <- data.frame(
  x=x,
  y=y
)
```

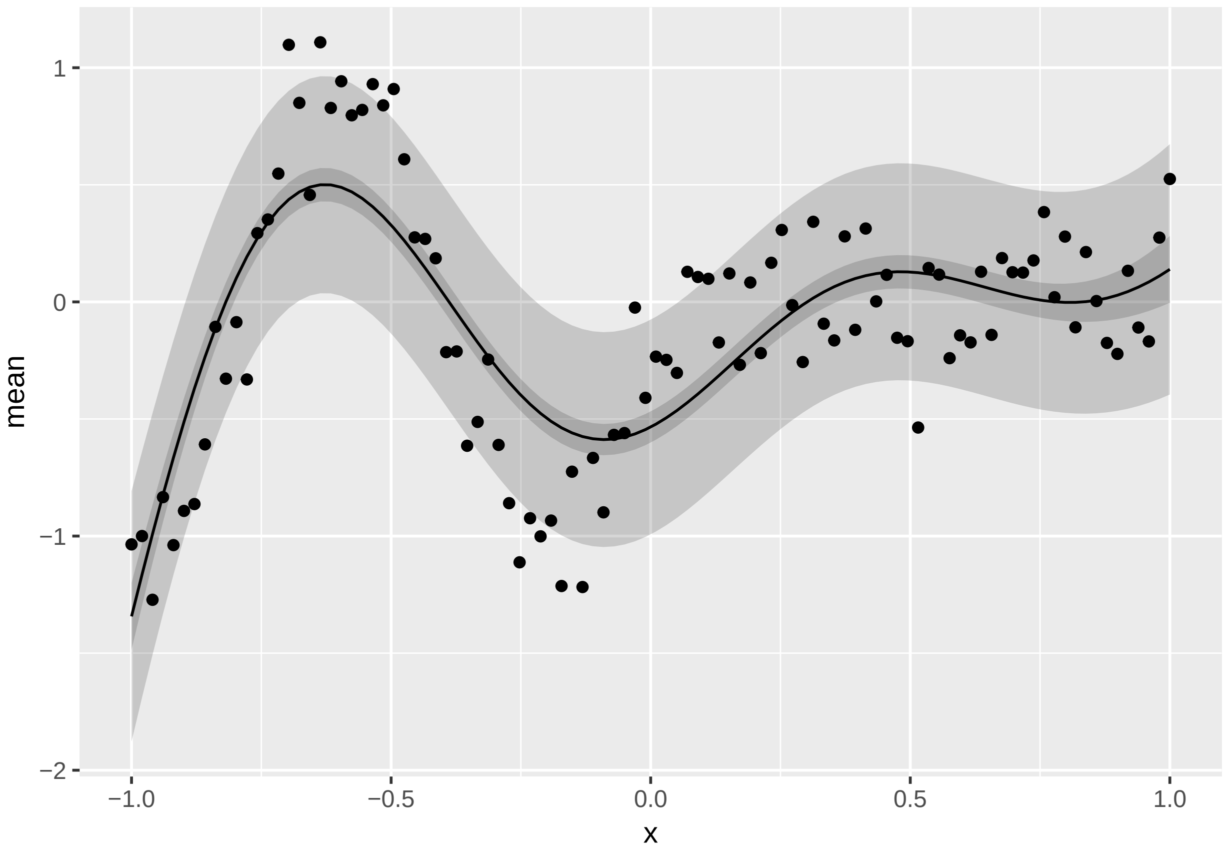
First, let us plot the posterior for $l = 0.2$. Lightly shaded area is the prediction interval for the data points, dark shaded areas are the probability intervals for f and line is the posterior mean.

```r
ell <- 0.2
gp_fit <- gausspr(x=x, y=y, kernel=k, kpar=list(sigmaf=1,ell=ell),var=0.2^2)
gp_posterior_mean <- predict(gp_fit, grid)

sq_exp_k <- k(sigmaf=1, ell=ell)
posterior_var <- kernelMatrix(sq_exp_k, grid, grid) -
  kernelMatrix(sq_exp_k, grid, x) %*%
  solve(kernelMatrix(sq_exp_k, x, x) + 0.2^2 * diag(length(x))) %*%
  kernelMatrix(sq_exp_k, x, grid)
posterior_var <- diag(posterior_var)
plot_df <- data.frame(
  x=grid,
  mean=gp_posterior_mean,
  lower_f=gp_posterior_mean - 1.96 * sqrt(posterior_var),
  upper_f=gp_posterior_mean + 1.96 * sqrt(posterior_var),
  lower_prediction=gp_posterior_mean - 1.96 * sqrt(posterior_var) - 1.96*0.2,
  upper_prediction=gp_posterior_mean + 1.96 * sqrt(posterior_var) + 1.96*0.2
)

ggplot(plot_df, aes(x=x)) +
  geom_ribbon(aes(ymin=lower_prediction, ymax=upper_prediction), alpha=0.2) +
  geom_ribbon(aes(ymin=lower_f,          ymax=upper_f),          alpha=0.2) +
  geom_line(aes(y=mean)) +
  geom_point(data=original_data_df, aes(x=x, y=y))
```



And for $l = 1$.

```r
ell <- 1
gp_fit <- gausspr(x=x, y=y, kernel=k, kpar=list(sigmaf=1,ell=ell),var=0.2^2)
gp_posterior_mean <- predict(gp_fit, grid)

sq_exp_k <- k(sigmaf=1, ell=ell)
posterior_var <- kernelMatrix(sq_exp_k, grid, grid) -
  kernelMatrix(sq_exp_k, grid, x) %*%
  solve(kernelMatrix(sq_exp_k, x, x) + 0.2^2 * diag(length(x))) %*%
  kernelMatrix(sq_exp_k, x, grid)
posterior_var <- diag(posterior_var)
plot_df <- data.frame(
  x=grid,
  mean=gp_posterior_mean,
  lower_f=gp_posterior_mean - 1.96 * sqrt(posterior_var),
  upper_f=gp_posterior_mean + 1.96 * sqrt(posterior_var),
  lower_prediction=gp_posterior_mean - 1.96 * sqrt(posterior_var) - 1.96*0.2,
  upper_prediction=gp_posterior_mean + 1.96 * sqrt(posterior_var) + 1.96*0.2
)

ggplot(plot_df, aes(x=x)) +
  geom_ribbon(aes(ymin=lower_prediction, ymax=upper_prediction), alpha=0.2) +
  geom_ribbon(aes(ymin=lower_f,          ymax=upper_f),          alpha=0.2) +
  geom_line(aes(y=mean)) +
  geom_point(data=original_data_df, aes(x=x, y=y))
```



We can see that the posterior mean (and the corresponding confidence bands) are smoother in case of $l = 1$, which is expected given the discussion in (a). Squared exponential kernel is probably a reasonable choice for this data, because the kernel not periodic, and we do not see much periodicity in the data either (and we do not know what this data represents, therefore cannot not expect periodicity apriori). However, the relationship seems to be highly non-linear, which makes it a good use case for the Gaussian processes (as opposed to linear model, for example).

(c)

To find the best set of hyper parameters for Gaussian processes we can, for example, maximize the marginal likelihood of the model, thus choosing the model (GP with the set of hyper parameters), which is the most likely given our, data marginalizing over all possible functions.

If the noise variance is unknown, we can handle it in the same way as any other hyper parameters. Or alternatively, we could try estimating it from a simpler model, e.g. by fitting a polynomial model, and then use the estimate.

Task 4

First, let us specify transitions, emissions, etc.

```r
t_total <- 10000

A <- 1
B <- 1
C <- 1

R <- 1^2
Q <- 5^2
Sigma_0 <- 10^2

mu_0 <- 50
```

Then simulate.

```r
u <- rep(1, t_total)  # actions (always 1)
x <- rep(NA, t_total) # hidden
z <- rep(NA, t_total) # observed

x[1] <- rnorm(1, mu_0, sqrt(Sigma_0))
z[1] <- rnorm(1, x[1], sqrt(Q))

for (t in 2:t_total) {
  x[t] <- rnorm(1, x[t-1] + B*u[t], sqrt(R))
  z[t] <- rnorm(1, x[t], sqrt(Q))
}
```

Then implement (but not run) Kalman filter.

```r
kalman_filter <- function(A, B, C, R, Q, Sigma_0, mu_0, u, z) {
  # Arguments follow notation on the slides, Lecture SSM 1, slide 11, except for
  # all them are scalars and not matrices/vectors.
  #
  # Returns:
  #   List:
  #     $mu
  #     $sigma

  t_total <- length(z)

  mu <- rep(NA, t_total)
  sigma <- rep(NA, t_total)

  mu[1] <- mu_0
  sigma[1] <- Sigma_0

  for (t in 2:t_total) {
    mu_bar <- A * mu[t-1] + B * u[t]
    sigma_bar <- A * sigma[t-1] * A + R
    K <- sigma_bar * C / (C * sigma_bar * C + Q)

    mu[t] <- mu_bar + K * (z[t] - C * mu_bar)
    sigma[t] <- (1 - K * C) * sigma_bar
  }
  return(list(mu = mu, sigma = sigma))
}
```

Then implement (but not run) Kalman filter and the auxiliary functions used by it.

```
particle_filter <- function(reading,
                            sample_transition,
                            density_emission,
                            n_particles = 100) {
  # Perform particle filtering given readings and functions that sample from
  # transition probabilities and compute density of emission probabilities.
  #
  # Args:
  #  reading           Vector of readings.
  #  sample_transition Function that takes pervious state and samples the new
  #                    one.
  #  density_emission  Function that computes density of emission.
  #  n_particles       Number of particles to use.
  #
  # Returns:
  #  List:
  #    $particles
  #    $weights

  sample_particles <- function(mu, weigths, n) {
    # Sample particles, given previous particles and their weights.
    mus <- sample(mu, n, replace=TRUE, prob=weigths)
    # For each mean, sample an observation from that distribution
    particles <- sapply(mus, function(mu) {
      sample_from_mixture <- sample_transition(mu)
    })
    return(particles)
  }
  weight_particles <- function(particles, reading) {
    # Compute weight of particles, given current reading.
    weights <- sapply(particles, function(particle) {
      density_emission(reading, particle)
    })
    return(weights / sum(weights))
  }
  t_total <- length(reading)
  particles <- matrix(NA, nrow=t_total, ncol=n_particles)
  weights <- matrix(NA, nrow=t_total, ncol=n_particles)
  particles[1, ] <- runif(n_particles, 0, 100)
  weights[1, ] <- weight_particles(particles[1, ], reading[1])

  for (t in 2:t_total) {
    particles[t, ] <- sample_particles(particles[t-1, ], weights[t-1, ], n_particles)
    weights[t, ] <- weight_particles(particles[t, ], reading[t])
  }
  return(list(particles = particles, weights = weights))
}

get_expected_state <- function(particles, weights, t) {
  # Compute expected state at t, given matrix of particles and weights.
  #
  # Args:
  #  particles Matrix of weights (T x N).
  #  weights   Matrix of particles (T x N).
  #  t         Numeric, period for which to compute expected state.
  #
  # Returns:
  #  Expected state.

  return(as.numeric(t(weights[t, ]) %*% particles[t, ]))
}


sample_transition <- function(x_previous) {
  u_t <- 1 # in our task it is always 1
  # B and R are set in the parent environment
  return(rnorm(1, x_previous + B*u_t, sqrt(R)))
}

density_emission <- function(x_t, particle) {
  # Q is set in the parent environment
  return(dnorm(particle, x_t, sqrt(Q)))
}

# Perform filtering
particle_filter_expectation <- function(z, sample_transition, density_emission, ...) {
  # Perform particle filtering given readings and functions that sample from
  # transition probabilities and compute expected filtered values.
  #
  # Args:
  #  reading           Vector of readings.
  #  sample_transition Function that takes pervious state and samples the new
  #                    one.
  #  density_emission  Function that computes density of emission.
  #  ...               Other arguments to particle_filter.
  #
  # Returns:
  #   Vector of expectations.

  p_filter <- particle_filter(z, sample_transition, density_emission, ...)
  particles <- p_filter$particles
  weights <- p_filter$weights

  mu_particle <- rep(NA, t_total)
  for (t in 1:t_total) {
    mu_particle[t] <- get_expected_state(particles, weights, t)
  }
  return(mu_particle)
}
```

Then compare the performance.

```
performance <- data.frame(
  type = c("Kalman", "Observations", rep("Particle", 3)),
  particles = c(rep(NA, 2), c(10, 50, 100)),
  error_mean = rep(NA, 5),
  error_sd = rep(NA, 5),
  runtime = rep(NA, 5)
)

performance[1, "runtime"] <- 0
performance[1, "error_mean"] <- mean(abs(x - z))
performance[1, "error_sd"] <- sd(abs(x - z))

start_time <- Sys.time()
mu_kalman <- kalman_filter(A, B, C, R, Q, Sigma_0, mu_0, u, z)$mu
end_time <- Sys.time()
performance[2, "runtime"] <- end_time - start_time
performance[2, "error_mean"] <- mean(abs(x - mu_kalman))
performance[2, "error_sd"] <- sd(abs(x - mu_kalman))

for (n_particles in c(10, 50, 100)) {
  start_time <- Sys.time()
  mu_particle <- particle_filter_expectation(z, sample_transition,
                                              density_emission,
                                              n_particles = n_particles)
  end_time <- Sys.time()
  performance_table_row <- performance$type == "Particle" &
    performance$particles == n_particles
  performance[performance_table_row, "runtime"] <- end_time - start_time
  performance[performance_table_row, "error_mean"] <- mean(abs(x - mu_particle))
  performance[performance_table_row, "error_sd"] <- sd(abs(x - mu_particle))
}
```

```
knitr::kable(performance, digits=3,
             col.names = c("Type", "# particles", "Error mean", "Error s.d.",
                           "Runtime"))
```

Type	# particles	Error mean	Error s.d.	Runtime
Kalman	NA	3.983	3.020	0.000
Observations	NA	1.750	1.283	0.029
Particle	10	2.219	1.705	1.160
Particle	50	1.841	1.361	3.887
Particle	100	1.785	1.328	6.609

As we see, the best performance in terms mean and standard deviation of errors is for Kalman filter. This logical, since Kalman filter uses exact analytic solution of the linear Gaussian state space model, which is our true underlying model. Performance of particle filters are worse, because they approximate posterior distributions. Therefore, it is reasonable to expect that performance of approximations is worse than the performance of the exact analytic solutions.

The more particles we use for a particle filter, the better the accuracy (lower mean and standard deviation of errors). This is also expected, because the more particles we use, the better we approximate the true posterior, because we approximate it at more points.

Runtime-wise, Kalman filter is the fastest, because it does not have to sample particles and compute their likelihoods for each step. Particle filters are slower, and the more particles we use, the slower they become.

It is worth noting, however, that although particle filtering is a slower and approximate solution, it is still much more accurate than using (noisy) readings directly.