TDDE15 Tenta 2020-10-27

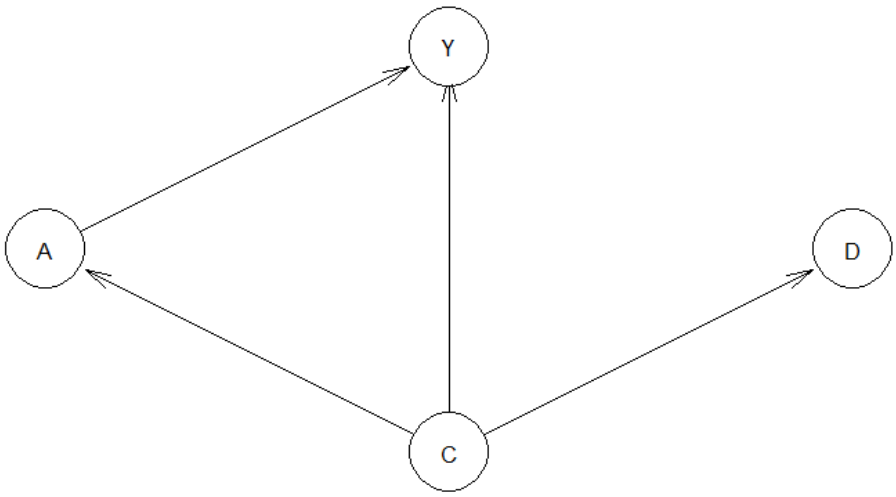
Assignment 1

Grade: 6/7

Task (1)

The binary variables are taken into account when parameterizing the model.

```
  structure = model2network("[C][A|C][Y|A:C][D|C]")
```



Task (2)

I make a function getRandom to get a random parameterization of the graph model. I do this by sampling a random number **r** from a uniform distribution in [0,1] for each conditional probability e.g p(y|a=0, b=1) and setting e.g p(y=1|a=0, b=1) = **r** and p(y=0|a=0, b=1)=1-**r.**

```
getRandom = function(){
  structure = model2network("[C][A|C][Y|A:C][D|C]")


  # C
  cparams = runif(1)
  par.C = matrix(c(cparams, 1-cparams), ncol = 1, nrow = 2,
dimnames=list(C=c(0,1)))

  # y
  yparams = runif(4)
```

```
  par.Y = c(yparams[1], 1-yparams[1],
            yparams[2], 1-yparams[2],
            yparams[3], 1-yparams[3],
            yparams[4], 1-yparams[4])
  dim(par.Y) = c(2,2,2)
  dimnames(par.Y) = list("Y"=c(0,1), "C"=c(0,1), "A" =c(0,1))

  # A
  aparams = runif(2)
  par.A = matrix(c(aparams[0], 1-aparams[0],
                   aparams[1], 1-aparams[1]), ncol = 2, nrow = 2,
dimnames=list(A=c(0,1), C=c(0,1)))

  # D
  dparams = runif(2)
  par.D = matrix(c(dparams[1], 1-dparams[1],
                   dparams[2], 1-dparams[2]), ncol = 2, nrow = 2)
  dimnames(par.D) = list("D"=list(0,1), "C"=list(0,1))

  # Y
  yparams = runif(4)
  par.Y = c(yparams[1], 1-yparams[1],
            yparams[2], 1-yparams[2],
            yparams[3], 1-yparams[3],
            yparams[4], 1-yparams[4])
  dim(par.Y) = c(2,2,2)
  dimnames(par.Y) = list("Y"=c(0,1), "C"=c(0,1), "A" =c(0,1))
  fitted_bn = custom.fit(structure, dist = list("C" = par.C, "Y" = par.Y,
"A" = par.A, "D"=par.D))
  return(fitted_bn)
}
```

I make a helper function to get probabilities from querygrain.

```
get_probs = function(grainTree, nodes, states, goalNode){

  # Probability for first door
  evidence <- setEvidence(object = grainTree,
                          nodes = nodes,
                          states = states)

  probs = querygrain(object = evidence,
                     nodes = goalNode)[goalNode]
  return(probs)
}
```

I make a helper function to find if the function is monotone over a given variable given A.

```
check_monotone =function(grainTree, over){
  # p(Y=1|A=1, over=1)
  prob1 = get_probs(grainTree, c("A",over), c("1","1"), "Y")$Y[2]

  # p(Y=1|A=1, over=0)
  prob2 = get_probs(grainTree, c("A",over), c("1","0"), "Y")$Y[2]

  # p(Y=1|A=0, over=0)
  prob3 = get_probs(grainTree, c("A",over), c("1","0"), "Y")$Y[2]

  # p(Y=1|A=0, over=1)
  prob4 = get_probs(grainTree, c("A",over), c("1","0"), "Y")$Y[2]
  non_decreasing = prob1>=prob2 && prob2>=prob3
  non_increasing = prob1<=prob2 && prob2<=prob3
  monotone = non_decreasing | non_increasing
  return(monotone)
}
```

I then set the given variable to C to check monotone in C and to D to check monotone in D. I do this for every randomized graph and check cases (i) and (ii). I count these up and print them out below.

```
ss = 1000

# counts of graphs that are monotone in C but not D
count1 = 0

# counts of graphs that are monotone in D but not C
count2 = 0

for(i in 1:ss){
  fitted_bn = getRandom()

  grainBN = as.grain(fitted_bn)
  grainTree <- compile(grainBN)

  # check i)
  monoC = check_monotone(grainTree, "C")
  monoD = check_monotone(grainTree, "D")

  is_1 = monoC && !monoD
  is_2 = monoD && !monoC
  count1 = count1 + as.numeric(is_1)
  count2 = count2 + as.numeric(is_2)
}

print(sprintf("%i of case (i)   %i of case (ii)", count1, count2))
```

```
## [1] "0 of case (i)   0 of case (ii)"
```

There are 0 cases found for both case (i) and (ii). This implies that either the model is monotone over in C and D or not monotone over any of them for all cases of randomized models.

Assignment 2

Grade: 3/7

Task (1)

I implement the algorithm by changing the correction by swapping the max q table of the next state given the optimal policy to the value of the q table in the next state and action from the EpislonGreedy strategy.

Also, as the environments now give negative rewards for all states except the reward, we terminate the algorithm once we find the positive reward. We could have added a limit for the amount of actions also, to adjust for iterations with high exploration where the agent takes many actions before finding the reward. We also sum up the rewards instead of giving the final reward, as rewards are collected throughout the iterations.

```
alt_q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma =
0.95,
                           beta = 0){

  current_state = start_state
  summed_rewards = 0
  repeat{
    # Follow policy, execute action, get reward.

    # Q-table update.
    episode_correction = 0

    ## Get action and reward for current state



    action = alt_EpsilonGreedyPolicy(current_state[1],current_state[2],
epsilon)
    new_state = transition_model(current_state[1],current_state[2],
action=action, beta=beta)
    reward = reward_map[new_state[1],new_state[2]]

    # get a' from new_states current action and s' from new state to get
    # g(s', a')

      ##### This is the main change if the algorithm compared to the
      q_learning algorithm. We also use a different policy method to use
      the alt_q_table instead


    next_state_action = alt_EpsilonGreedyPolicy(new_state[1],new_state[2],
epsilon)
    qnew = alt_q_table[new_state[1], new_state[2], next_state_action]

    # Calculate correction
    # correction is altered to the alternative algorithm where the max
```

```
action value
    # of the next state in the q table is replaced by the epsilonGreedy
policy action
    # of the new state (qnew)
    correction = alpha*(gamma*qnew +
                            reward-
alt_q_table[current_state[1],current_state[2],action])

    # Update current Q-table
    alt_q_table[current_state[1],current_state[2],action] <<-
      alt_q_table[current_state[1],current_state[2],action] + correction

    #Sum all corrections
    episode_correction = episode_correction + correction
    current_state = new_state

    summed_rewards = summed_rewards + reward
    if(reward>=0)
      # End episode.
      return (c(summed_rewards,episode_correction))

  }
}
```

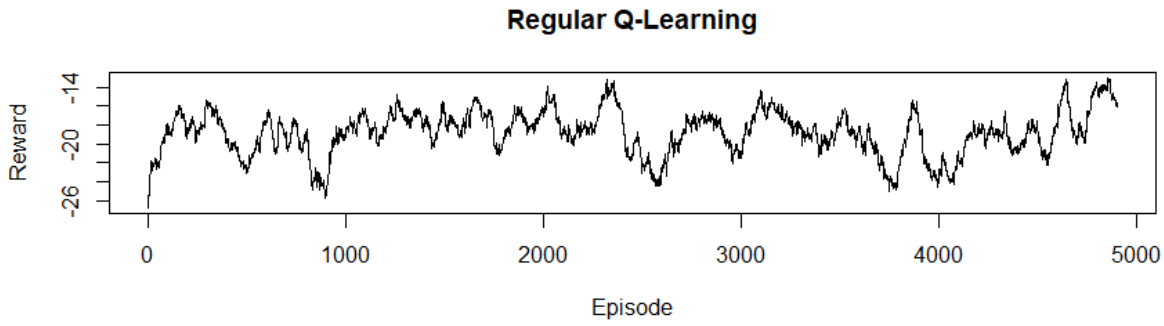
Task (2)

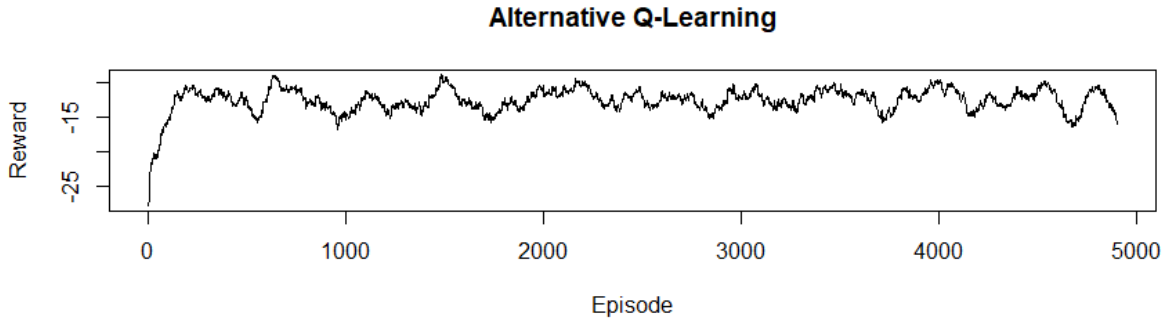
We change the reward map to the specifications. The observed alternative q table has more negative q values, probably since randomization is used in the correction step for the next state which can result in more negative rewards for instance by randomly stepping into the -10 rewards.

```
H <- 3
W <- 6

epsilon = 0.5
gamma = 1
beta = 0
alpha = 0.1

# -1 for all positions, except 1,2:5 = -10 and 1,6 = 10
reward_map <- matrix(-1, nrow = H, ncol = W)
reward_map[1,2:5] <- -10
reward_map[1,6] <- 10

q_table <- array(0,dim = c(H,W,4))
alt_q_table <- array(0,dim = c(H,W,4))

rewards = c()
alt_rewards = c()


# Train (2)
for(i in 1:5000){
```

```
  rew <- q_learning(epsilon=epsilon, alpha=alpha, gamma = gamma,  beta = be
ta, start_state = c(1,1))
  alt_rew <- alt_q_learning(epsilon=epsilon, alpha=alpha, gamma = gamma,  b
eta = beta, start_state = c(1,1))
  rewards = c(rewards, rew[1])
  alt_rewards = c(alt_rewards, alt_rew[1])
}

graphics.off()
par(mfrow=c(2,1))
plot(MovingAverage(rewards,100),type = "l", xlab="Episode", ylab="Reward",
main= "Regular Q-Learning")
plot(MovingAverage(alt_rewards,100),type = "l", xlab="Episode", ylab="Rewar
d", main="Alternative Q-Learning")

print("q_table:")

## [1] "q_table:"

print(q_table)

## , , 1
##
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    4    5    6    7    8    0
## [2,]    3    4    5    6    7    8
## [3,]    3    4    5    6    7    8
##
## , , 2
##
##      [,1] [,2] [,3]         [,4] [,5]     [,6]
## [1,]   -5   -4   -3 -3.10449e-12   10 0.000000
## [2,]    5    6    7  8.00000e+00    9 9.000000
## [3,]    4    5    6  7.00000e+00    8 7.999999
##
## , , 3
##
##      [,1] [,2] [,3] [,4]          [,5] [,6]
## [1,]    3   -5   -4   -3 -9.278212e-10    0
## [2,]    3   -5   -4   -3 -7.105427e-15   10
## [3,]    4    5    6    7  8.000000e+00    9
##
## , , 4
##
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    3    3   -5   -4   -3    0
## [2,]    4    4    5    6    7    8
## [3,]    3    3    4    5    6    7

print("alt_q_table")

## [1] "alt_q_table"

print(alt_q_table)
```

```
## , , 1
##
##           [,1]      [,2]      [,3]       [,4]     [,5]     [,6]
## [1,] -10.521038 -9.434012 -9.155213 -4.1180093 2.349063 0.000000
## [2,]  -8.428915 -5.957634 -3.371387  0.3007393 1.591015 4.898164
## [3,]  -8.344501 -6.576628 -2.877220 -1.2538672 2.765009 4.557827
##
## , , 2
##
##           [,1]       [,2]        [,3]       [,4]      [,5]     [,6]
## [1,] -22.16582 -20.954944 -18.3925458 -8.6449333 10.000000 0.000000
## [2,] -10.74621  -6.558365  -4.6569266  0.3686293  7.027088 7.530734
## [3,]  -6.39576  -3.172907  -0.1141331  2.9235819  4.775272 5.187422
##
## , , 3
##
##           [,1]      [,2]       [,3]       [,4]       [,5]      [,6]
## [1,] -14.24423 -23.79752 -20.544466 -19.748926  -5.382054  0.000000
## [2,] -13.40624 -23.21277 -23.517670 -16.793607 -13.682036 10.000000
## [3,] -10.05648 -13.54616  -7.507567  -3.499379  -0.107331  7.847388
##
## , , 4
##
##           [,1]       [,2]       [,3]       [,4]       [,5]     [,6]
## [1,] -12.649810 -13.798517 -22.354394 -19.814985 -16.541041 0.000000
## [2,]  -9.997564 -10.348039 -10.430238  -8.784474  -3.771863 1.861648
## [3,]  -8.068342  -7.918723  -5.380988  -3.329890  -1.105741 1.676616
```





As we can see, the alternative Q-learning method performs slightly better under training. This might be because the algorithm has more exploration built in, as it explores the next state when updating the q_table for the current state, which means it might find the positive rewards faster. It also optimizes the Q_table for the epsilon strategy of randomizing actions, as it will assign q_values according to the prior belief that the actions will always be randomized with probability epsilon and therefor the agent will learn a q_table that better predicts expected future rewards under training with an epsilon greedy policy.

Task (3)

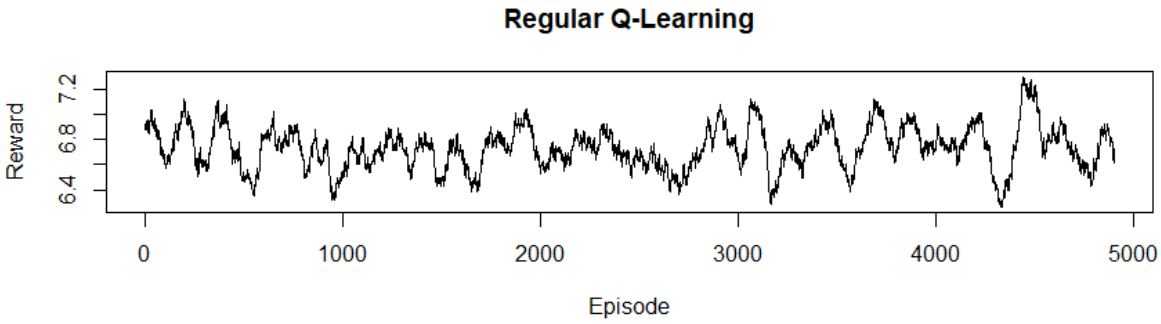
I make two helper functions for testing, where the Greedy policies are used to find the next state and the rewards are summed.

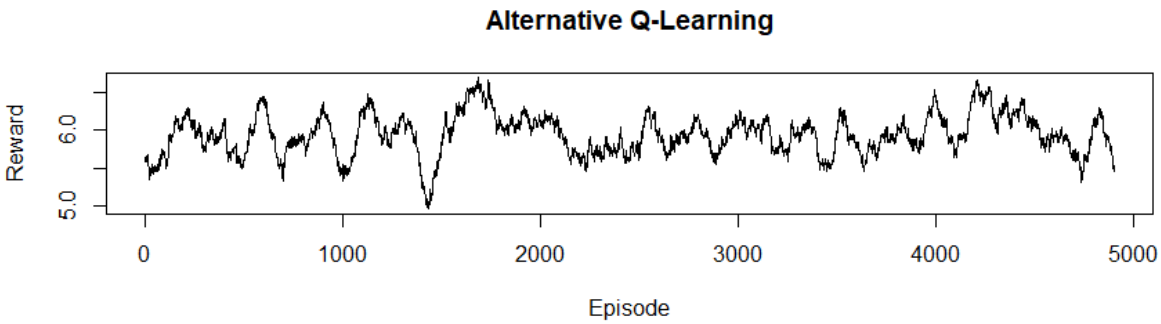
```
alt_q_test <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma =
0.95,
                           beta = 0){

  summed_rewards = 0
  current_state = start_state
  repeat{
    # Follow policy, execute action, get reward.


    ## Get action and reward for current state
    action = alt_GreedyPolicy(current_state[1],current_state[2])
    new_state = transition_model(current_state[1],current_state[2],
action=action, beta=beta)
    reward = reward_map[new_state[1],new_state[2]]

    current_state = new_state

    summed_rewards = summed_rewards +  reward
    if(reward>=0)
      # End episode.
      return (summed_rewards)

  }
}

q_test <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0){

  summed_rewards = 0
  current_state = start_state
  repeat{
    # Follow policy, execute action, get reward.


    ## Get action and reward for current state
    action = GreedyPolicy(current_state[1],current_state[2])
```

```
    new_state = transition_model(current_state[1],current_state[2],
action=action, beta=beta)
    reward = reward_map[new_state[1],new_state[2]]

    current_state = new_state

    summed_rewards = summed_rewards +  reward
    if(reward>=0)
      # End episode.
      return (summed_rewards)

  }
}
```

We then run testing over 5000 iterations by calling the functions. We also randomize the starting point, as we otherwise would get the same reward for all episodes and thus produce a flat plot and would not observe how the agent behaves from different positions. We also plot the mean reward to get a clear number on which agent produces better rewards on average.

```
# Test (3)

test.rewards = c()
test.alt_rewards = c()
for(i in 1:5000){
  test.rew <- q_test(epsilon=epsilon, alpha=alpha, gamma = gamma,  beta = b
eta, start_state = c(runif(1,1,H),runif(1,1,W)))

  test.alt_rew <- alt_q_test(epsilon=epsilon, alpha=alpha, gamma = gamma,
beta = beta, start_state =  c(runif(1,1,H),runif(1,1,W)))

   test.rewards = c(test.rewards, test.rew)
  test.alt_rewards = c(test.alt_rewards, test.alt_rew)
}

graphics.off()
par(mfrow=c(2,1))
plot(MovingAverage(test.rewards,100),type = "l", xlab="Episode", ylab="Rewa
rd", main= "Regular Q-Learning")
plot(MovingAverage(test.alt_rewards,100),type = "l", xlab="Episode", ylab="
Reward", main="Alternative Q-Learning")

mean(test.rewards)

mean(test.alt_rewards)
```






```
> mean(test.rewards)
[1] 6.7338
> mean(test.alt_rewards)
[1] 5.9344
```

As we can see, the regular Q-learning algorithm performs better. This is probably because the alternative Q-learning “expects” a policy where the actions are randomized, so the values for the Q-table does not hold under a deterministic policy. It has therefor not learned the optimal policy, but the optimal policy given when actions are randomized by epsilon.

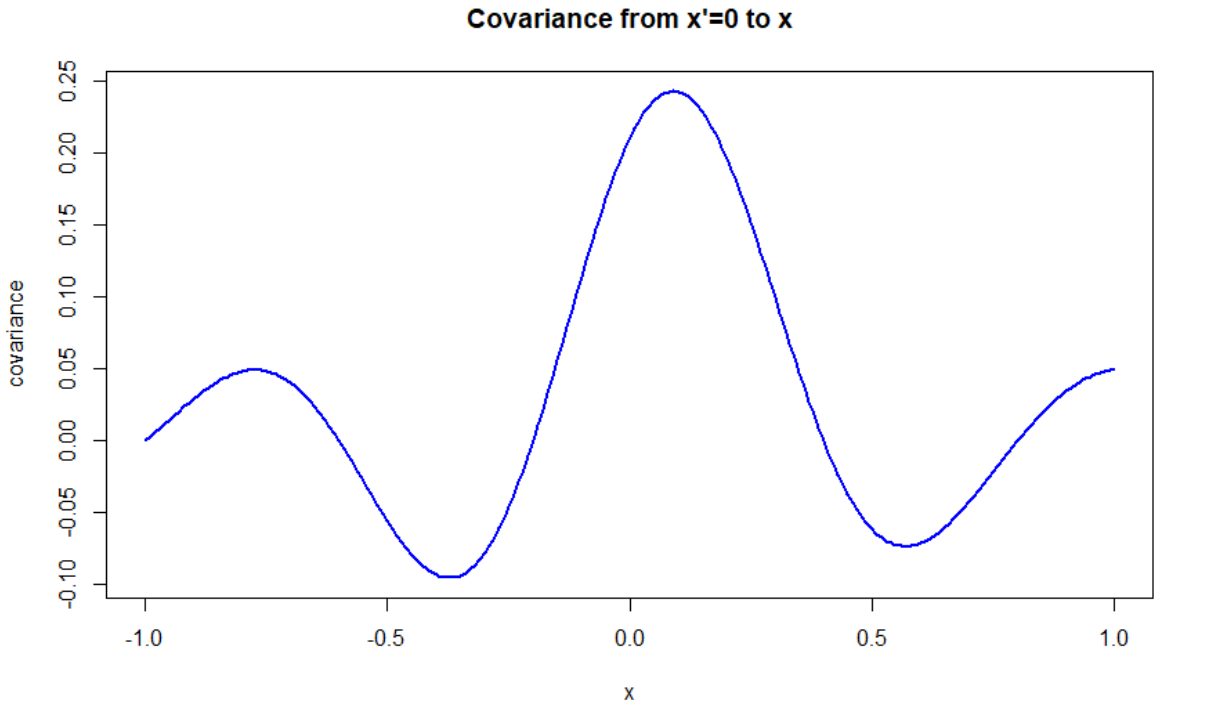
Assignment 3

Grade: 6/7

Task (1)

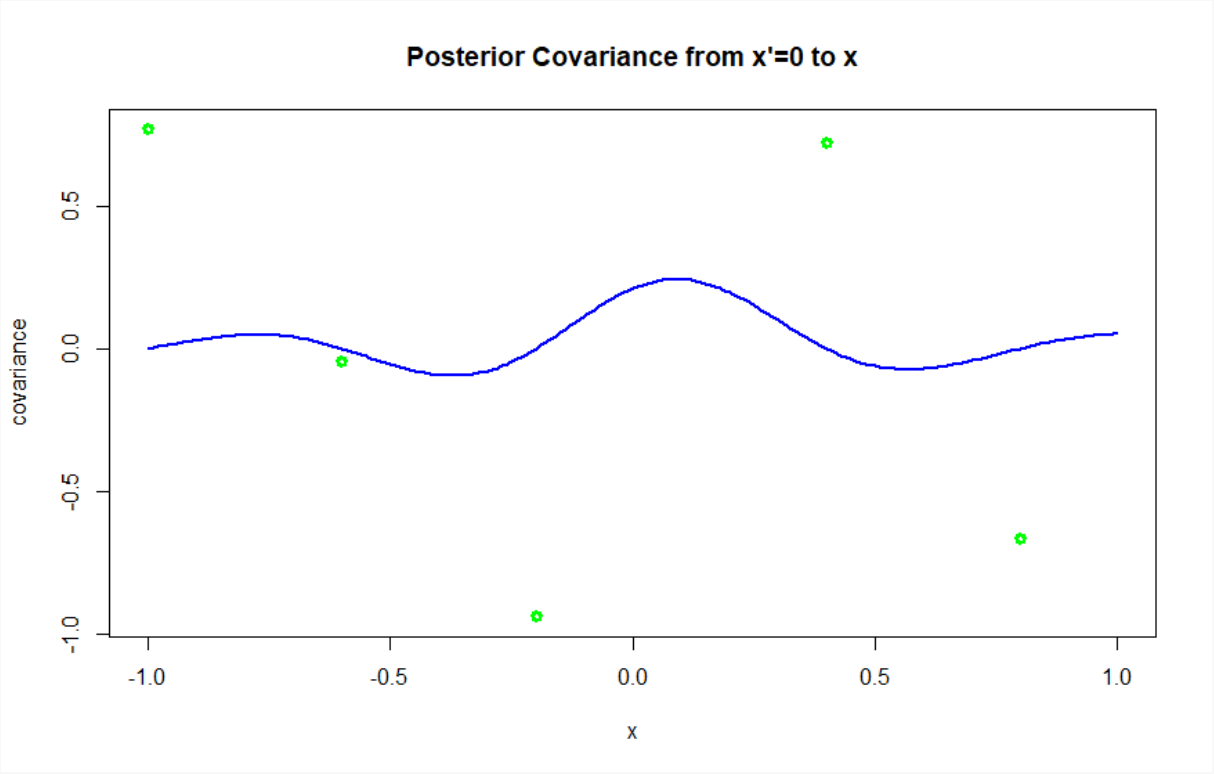
The questions are answered in the code with bullet points corresponding to 1-4.

```
graphics.off()
par(mfrow=c(1,1))
X = c(-1.0, -0.6, -0.2, 0.4, 0.8)
y = c(0.768, -0.044, -0.940, 0.719, -0.664)

sigmaNoise = 0
Xstar = seq(-1,1,by=0.01)

gpParam = posteriorGP(X, y, Xstar, sigmaNoise, kernelMaker(sigmaF=1, l=0.3)
)
plotGP(gpParam$fmean, gpParam$fvar, Xstar, X, y)


# (1)

## The covariance from x=0 to all other xs is found in the variance for the
corresponding column of
# x=0
plot(x=Xstar, y=gpParam$fvar[which(Xstar==0),], type="l", xlab="x", ylab="c
ovariance", main="Covariance from x'=0 to x", col="blue", lwd=2)
```



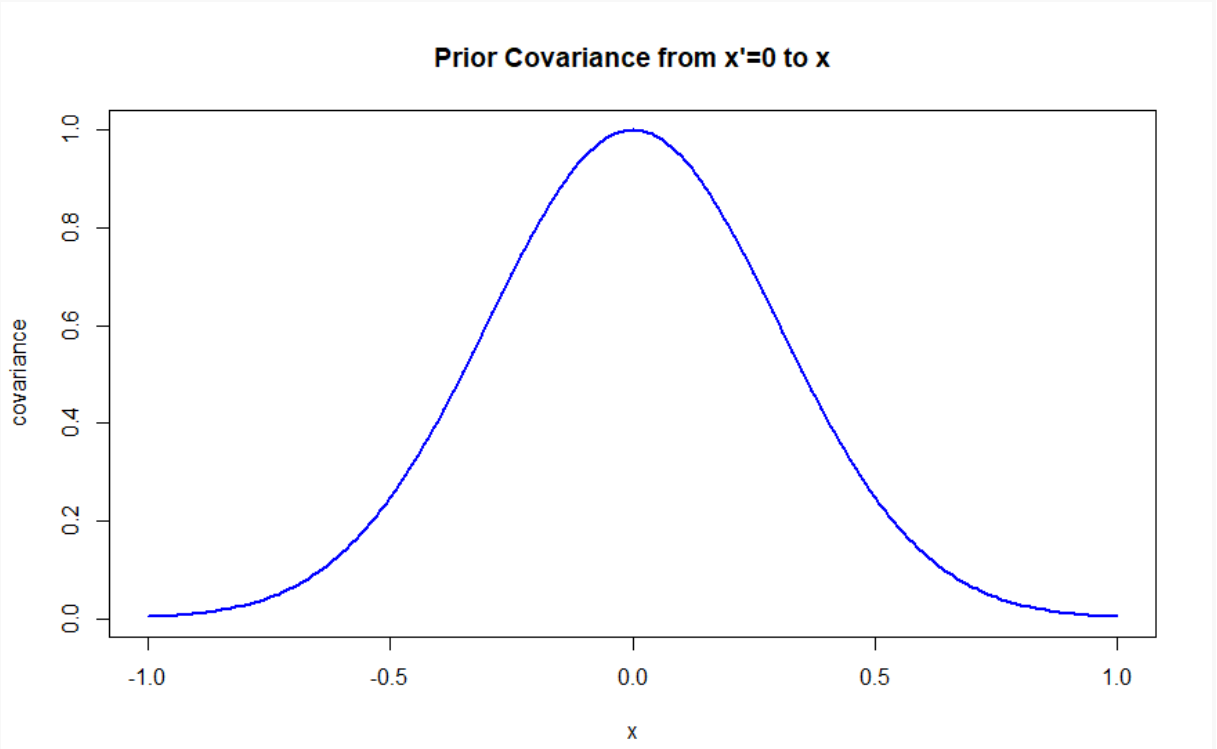

```
# as we can see, maximum covariance is found around 0, but slightly shifted
.
# this is because the points in the data influences the covariance by makin
g x values with similar y values more correlated
```

```r
# for instance, the training point at 0.8 is ~ 0.3 away from the training p
oint at -0.2 in y value. This means that the
# corresponding x values for theese points are slightly correlated, explain
ging the curvature upwards towards x=1 in the covariance plot


# (2)
plot(x=Xstar, y=gpParam$fvar[which(Xstar==0),], ylim=c(min(y),max(y)),
     type="l", xlab="x", ylab="covariance", main="Posterior Covariance from
x'=0 to x", col="blue", lwd=2)
points(x=X, y=y, col="green", lwd=3)
```




```r
# At all the plotted training points, the covariance is 0. This because the
the estimated f value at 0
# has no correlation with the x values that has a training point, as this c
ovariance only depends on the
# training point at that x value

# (3)
# because as we mentioned earlier, the covariance will be influenced by the
training points, and thus training
# points close to 0 will have correlation with other points where the train
ing points for 0 and that x value
# are similiar


# (4) the prior for the covariance of x and x' is given by the exponential
kernel. This will result in a decreasing correlation over distance from the
x value. The rate of decrease is related to the l value, a lower rate of de
crease would mean a larger l value and vice-versa.
varPrior = kernelMaker(sigmaF=1, l=0.3)(Xstar, 0)
```

```
plot(x=Xstar, y=varPrior, xlab="x", ylab="covariance", type="l", main="Prio
r Covariance from x'=0 to x", col="blue", lwd=2)
```



Task (2)

I use a grid search over the parameter of sigma. I then use the marginal likelihood given from Algorithm 2.1 from the book by Rasmussen and Williams of each given sigma, compare it to the current best marginal likelihood, and if it is better, I store that sigma and its likelihood.

The best sigma found was 0.1. The resulting posterior mean is plotted below.

```
posteriorGP = function (X, y, Xstar, sigmaNoise, k){
  kstar = k(X,Xstar)
  L = chol(k(X,X)+sigmaNoise^2*diag(length(X)))
  L = t(L)
  alpha = solve(t(L), solve(L,y))
  fMean = t(kstar) %*% alpha
  v = solve(L,kstar)
  fVar = k(Xstar, Xstar)-t(v)%*%v
  logMarginalLike = -0.5*(t(y)%*%alpha)-sum(diag(L))-length(y)/2*log(2*pi)
  return (list(fmean=fMean, fvar=fVar, logLike=logMarginalLike))
}

getMargLike = function (X, y, Xstar, sigmaNoise, k){
  post = posteriorGP(X, y, Xstar, sigmaNoise, k)
  return ((post$logLike))
}

k = kernelWrap(sigmaF = 20, l = 0.2)
```

```r
## Grid search over sigma param
best_sigma = 0
best_marg = -Inf


sigmas_grid = seq(0.1, 10, by=0.01)
for (sigma_test in sigmas_grid){
  currMarg = getMargLike(small.data$time, small.data$temp, small.data$time,
sigma_test, k)

  if(currMarg>best_marg){
    best_sigma = sigma_test
    best_marg = currMarg
  }
}

posterior = posteriorGP(small.data$time, small.data$temp,small.data$time, b
est_sigma, k)
plotGP(posterior$fmean, posterior$fvar, small.data$time, small.data$time, s
mall.data$temp)

print(best_sigma)

## 0.1
```