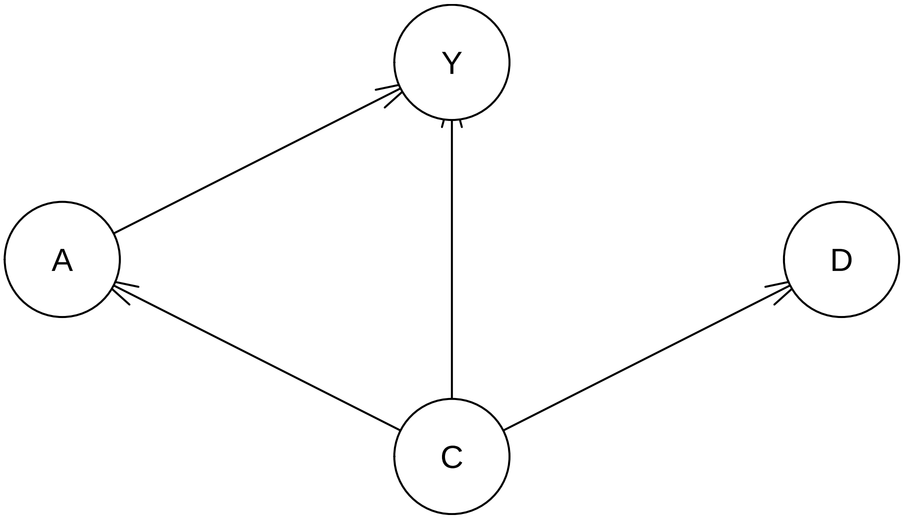
Exam: TDDE15

David Björelind, davbj395

10/27/2020

1. Graphical Models

```r
# Function to produce random parameters
rand_para = function(){
  # Making the Conditional Probability Tables
  r1 = runif(1, 0, 1)
  cptC = matrix(c(r1, 1-r1), ncol = 2, dimnames = list(NULL, c("0", "1")))

  r1 = runif(1, 0, 1)
  r2 = runif(1, 0, 1)
  cptD = matrix(c(r1, 1-r1,
                  r2, 1-r2))
  dim(cptD) = c(2,2)
  dimnames(cptD) = list("D" = c("0", "1"), "C" =  c("0", "1"))

  r1 = runif(1, 0, 1)
  r2 = runif(1, 0, 1)
  cptA = matrix(c(r1, 1-r1,
                  r2, 1-r2))
  dim(cptA) = c(2,2)
  dimnames(cptA) = list("A" = c("0", "1"), "C" =  c("0", "1"))

  r1 = runif(1, 0, 1)
  r2 = runif(1, 0, 1)
  r3 = runif(1, 0, 1)
  r4 = runif(1, 0, 1)
  cptY = c(r1, 1-r1,
           r2, 1-r2,
           r3, 1-r3,
           r4, 1-r4)
dim(cptY) = c(2,2,2)
dimnames(cptY) = list("Y" = c("0", "1"), "A" = c("0", "1"), "C" = c("0", "1"))
dist = list("C" = cptC, "D"= cptD, "A"= cptA, "Y" = cptY)
return(dist)
}
# Function to calculate if a model is monotone on "nodes"
isMonotone = function(grain, nodes){
  # Do exact inference
  goal = c("Y")
  states1 = c("1", "1")
  states2 = c("1", "0")
  states3 = c("0", "1")
  states4 = c("0", "0")
  evi = setEvidence(grain, nodes = nodes, states = states1)
  dist1 = querygrain(evi, nodes = goal)
  evi = setEvidence(grain, nodes = nodes, states = states2)
  dist2 = querygrain(evi, nodes = goal)
  evi = setEvidence(grain, nodes = nodes, states = states3)
  dist3 = querygrain(evi, nodes = goal)
  evi = setEvidence(grain, nodes = nodes, states = states4)
  dist4 = querygrain(evi, nodes = goal)
  # Check if nondecreasing first, then nonincreasing
  if(dist1$Y[2] >= dist2$Y[2] && dist3$Y[2] >= dist4$Y[2]){
    return(TRUE)
  } else if(dist1$Y[2] <= dist2$Y[2] && dist3$Y[2] <= dist4$Y[2]){
    return(TRUE)
  } else {
    return(FALSE)
  }
}

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



```
monoACnotAD = 0 # (i)
monoADnotAC = 0 # (ii)
monoAC = 0
monoAD = 0

for (i in 1:1000){
  parameters = custom.fit(graph, dist = rand_para())
  grain = as.grain(parameters)
  ac = isMonotone(grain, nodes=c("A", "C"))
  ad = isMonotone(grain, nodes=c("A", "D"))
  if(ac == TRUE && ad == FALSE){
    monoACnotAD = monoACnotAD + 1
  }
  if(ad == TRUE && ac == FALSE){
    monoADnotAC = monoADnotAC + 1
  }
  if(ac == TRUE){
    monoAC = monoAC +1
  }
  if(ad == TRUE){
    monoAD = monoAD +1
  }
}
monoACnotAD
```

```
## [1] 0
```

```
monoADnotAC
```

```
## [1] 0
```

```
monoAC
```

```
## [1] 482
```

```
monoAD
```

```
## [1] 482
```

I found that there were no cases of (i) and (ii). This means that p(y|a,c) is always monotone in C when p(y|a,d) is monotone in C, and vice versa.

2. Reinforcement Learning

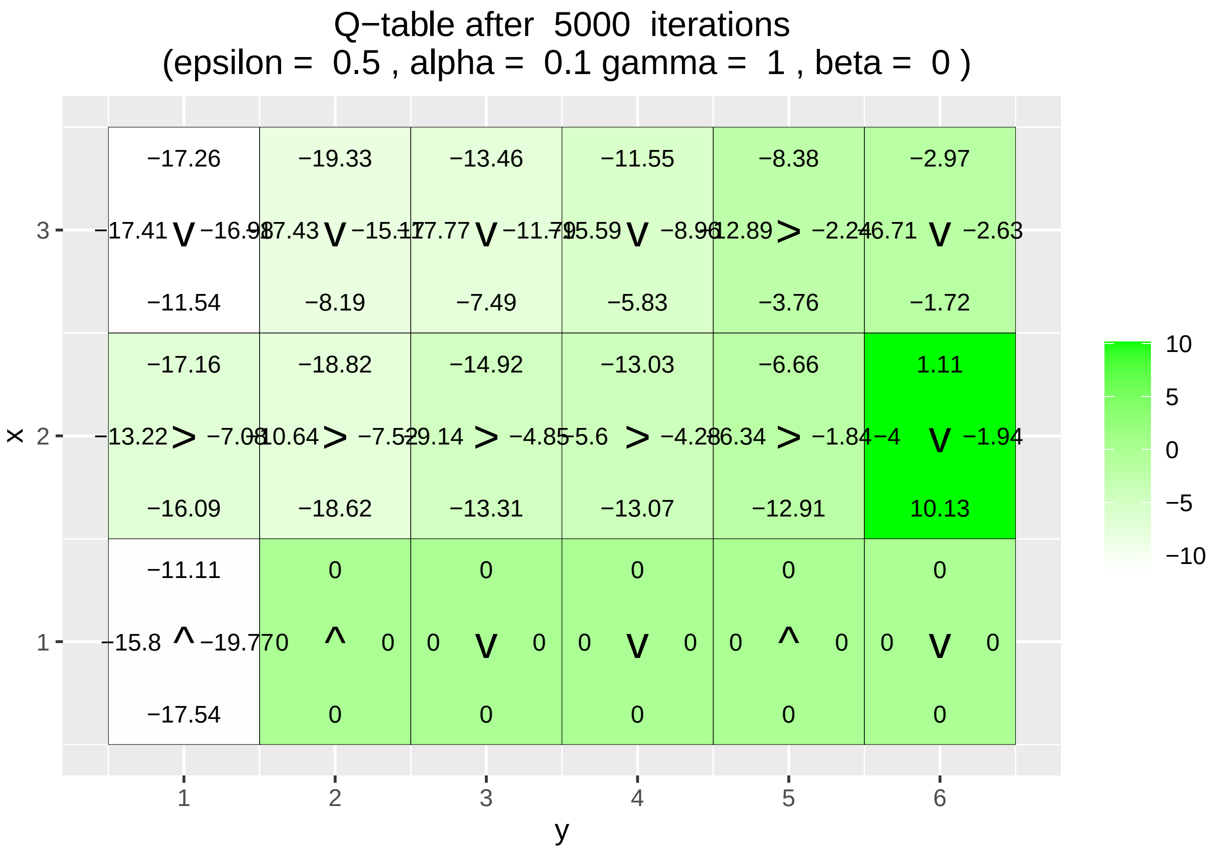
The functions vis_environment, GreedyPolicy, EpsilonGreedyPolicy and transition_model has not been altered and hence not included here.

a) Updating Q-learning

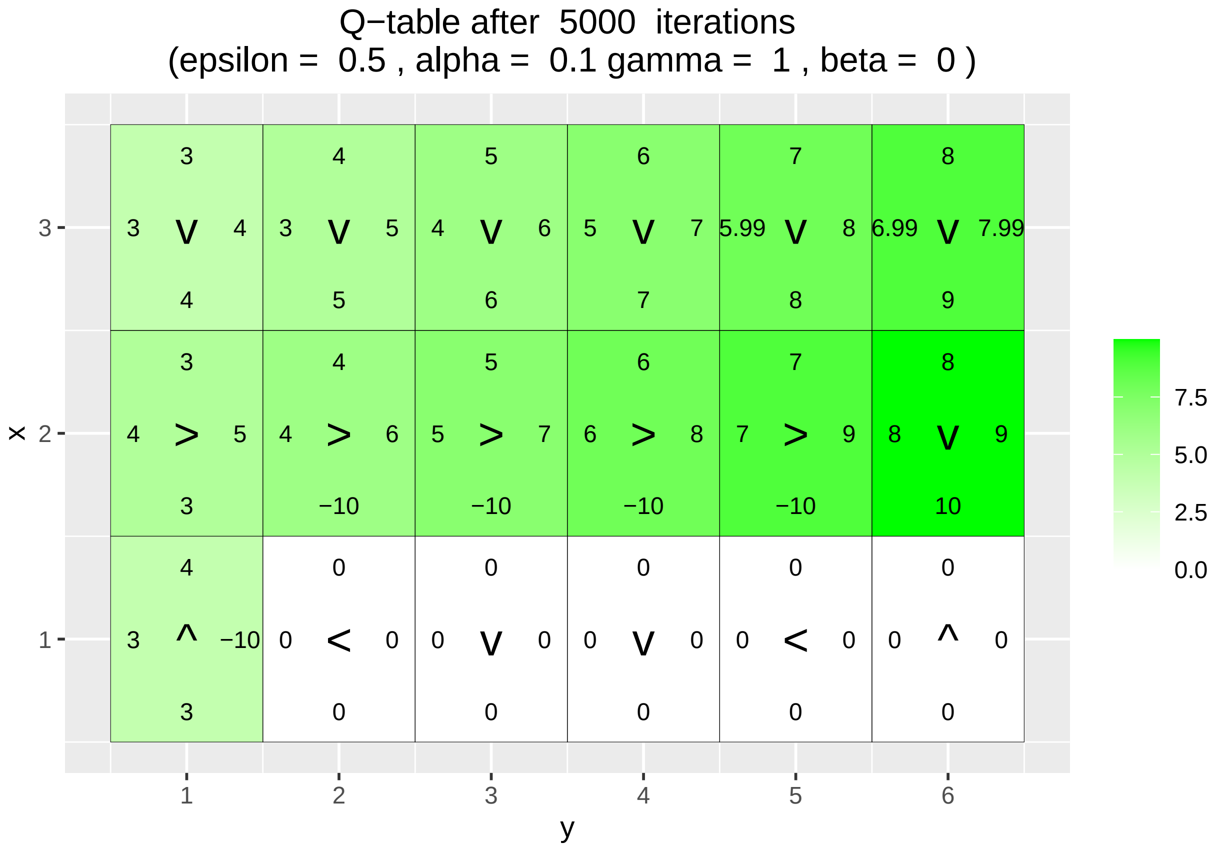
```r
q_learning_new <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95, beta = 0){

  state = start_state
  episode_correction = 0
  repeat{
    # Follow policy, execute action, get reward.
    action = EpsilonGreedyPolicy(state[1], state[2], epsilon) # Getting action
    new_state = transition_model(state[1], state[2], action, beta) # Updating state
    reward = reward_map[new_state[1], new_state[2]]

    # Old q-table update:
    #correction = alpha*(reward+gamma*max(q_table[new_state[1], new_state[2],]) - q_table[state[1], sta

    # New q-table update:
    new_action = EpsilonGreedyPolicy(new_state[1], new_state[2], epsilon) # Updating action
    next_state = transition_model(new_state[1], new_state[2], new_action, beta) # Updating state
    correction = alpha*(reward+gamma*q_table[next_state[1], next_state[2],][new_action] - q_table[state

    q_table[state[1], state[2],action] <<- q_table[state[1], state[2],action] + correction
    state = new_state
    episode_correction = episode_correction + correction
    #print(state)
    if(reward==10 || reward==-10)
      # End episode.
      return (c(reward,episode_correction))
  }
}
```

b) Modified environment C

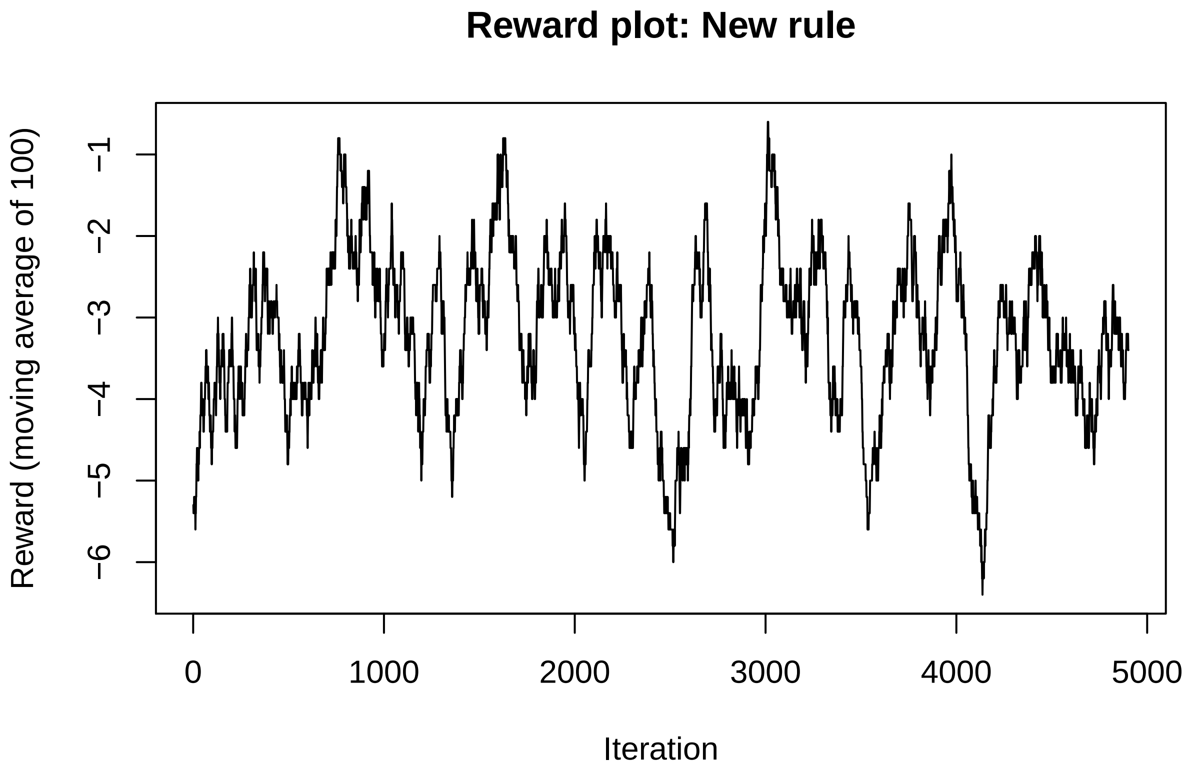
```r
# Environment C
H <- 3
W <- 6

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))
reward_new = 0

# Beta = 0, Epsilon = 0.5, gamma = 1, alpha = 0.1
for(i in 1:5000){
  foo <- q_learning_new(gamma = 1, beta = 0, start_state = c(1,1))
  reward_new <- c(reward_new,foo[1])
}
# Training new
train_reward_new = 0
for(i in 1:5000){
  foo <- q_learning_new(gamma = 1, beta = 0, start_state = c(1,1), alpha = 0, epsilon = 0)
  train_reward_new <- c(train_reward_new,foo[1])
}
reward_map <- matrix(0, nrow = H, ncol = W) # Removing rewards to be able to plot policy!
vis_environment(5000, gamma = 1, beta = 0)
```



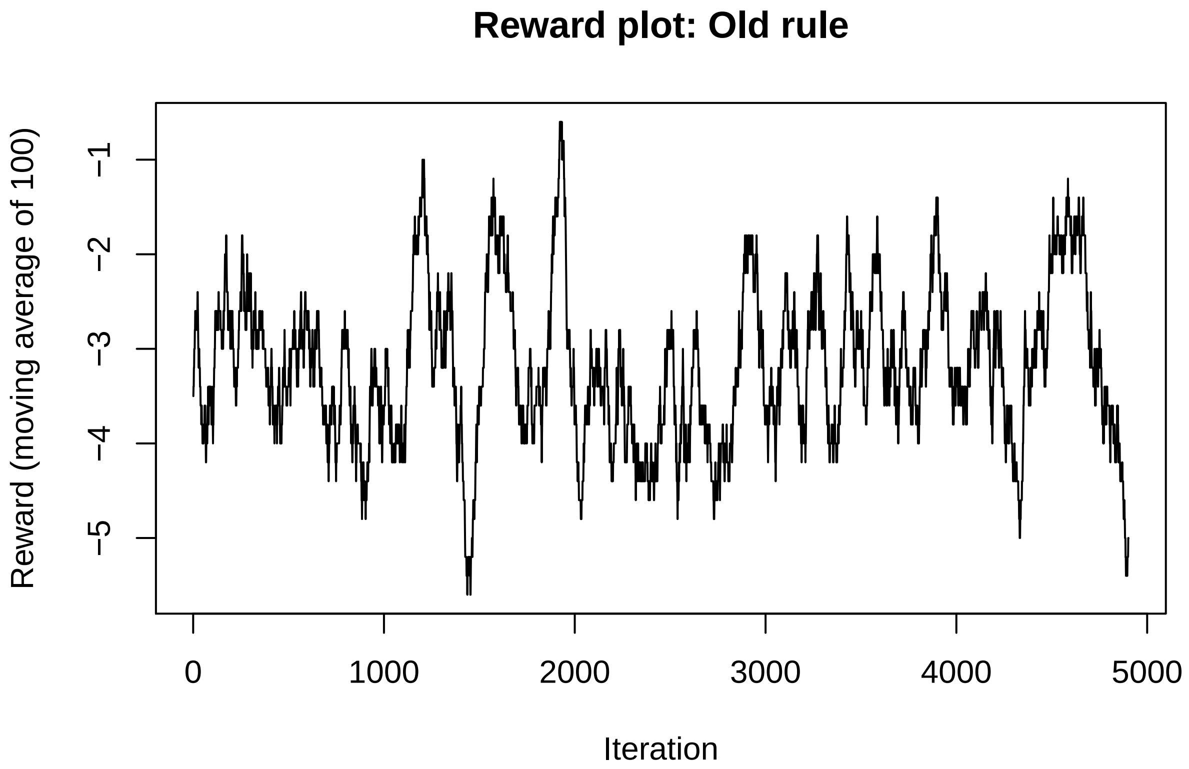
```
# OLD UPDATING RULE, resetting environment
reward_map <- matrix(-1, nrow = H, ncol = W)
reward_map[1,2:5] <- -10
reward_map[1,6] <- 10
q_table <- array(0,dim = c(H,W,4))
reward = 0

for(i in 1:5000){
  foo <- q_learning(gamma = 1, beta = 0, start_state = c(1,1))
  reward <- c(reward,foo[1])
}
# Training old
train_reward = 0
for(i in 1:5000){
  foo <- q_learning(gamma = 1, beta = 0, start_state = c(1,1), alpha = 0, epsilon = 0)
  train_reward <- c(train_reward,foo[1])
}
reward_map <- matrix(0, nrow = H, ncol = W) # Removing rewards to be able to plot policy!
vis_environment(5000, gamma = 1, beta = 0)
```



```
# Final q-table and policy
plot(MovingAverage(reward_new,100),type = "l", main="Reward plot: New rule", xlab="Iteration", ylab="Re
```



```
plot(MovingAverage(reward,100),type = "l", main="Reward plot: Old rule", xlab="Iteration", ylab="Reward
```



Note: I removed rewards in the policy/reward in order to show the policy produced. The model is still trained with the correct rewards.

The policy produced from the new updating rule works well. Comparing the policies between the two models, they are very similar. The robot walks along the "cliff" and in the last step (2,6) it walks down to the reward. A difference is that the new updating rule has mostly negative values in the q_table compared to the earlier rule. However, this does not seem to effect performance.

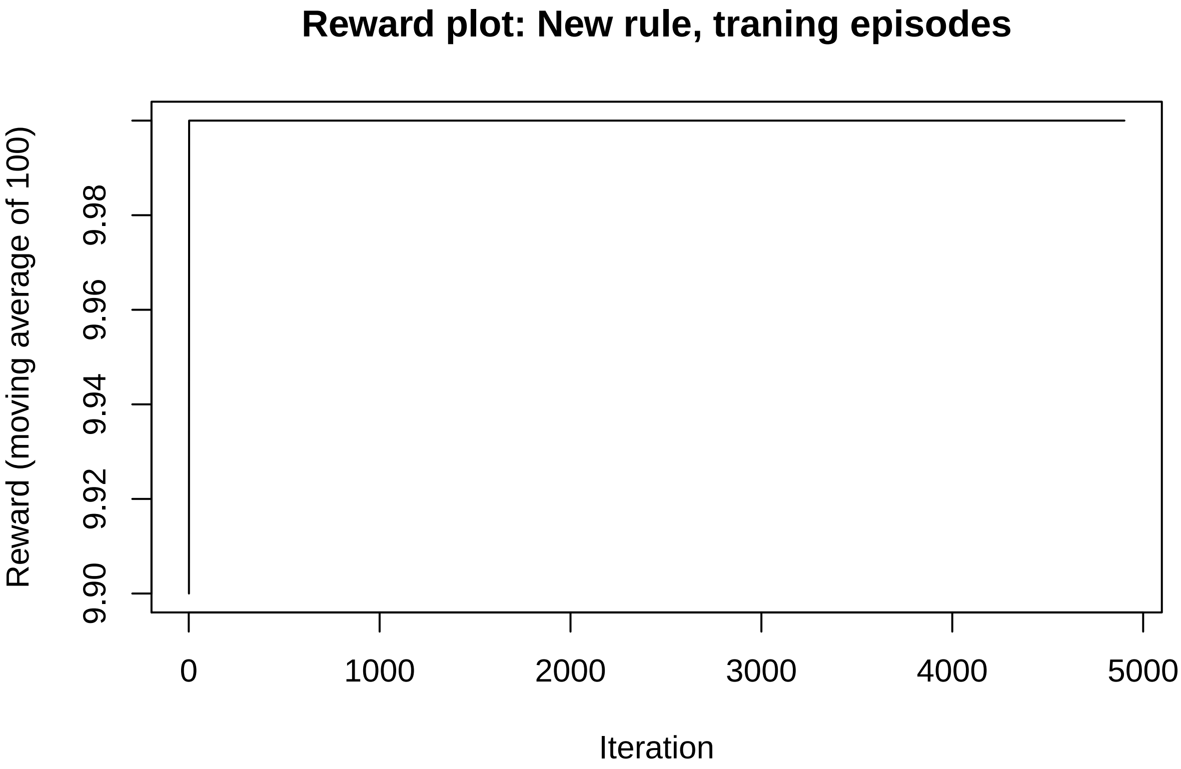
Looking at the reward plots, the two updating rules seems to behave the same. But looking at the final q_tables, the old rule produces clearer results in my opinion. The new updating rule is worse, since it seems it needs more episodes to learn the same policy "quality" as the old rule does. I say this because the old updating rule has more optimized q_values than the new rule.s

c) Testing the agent

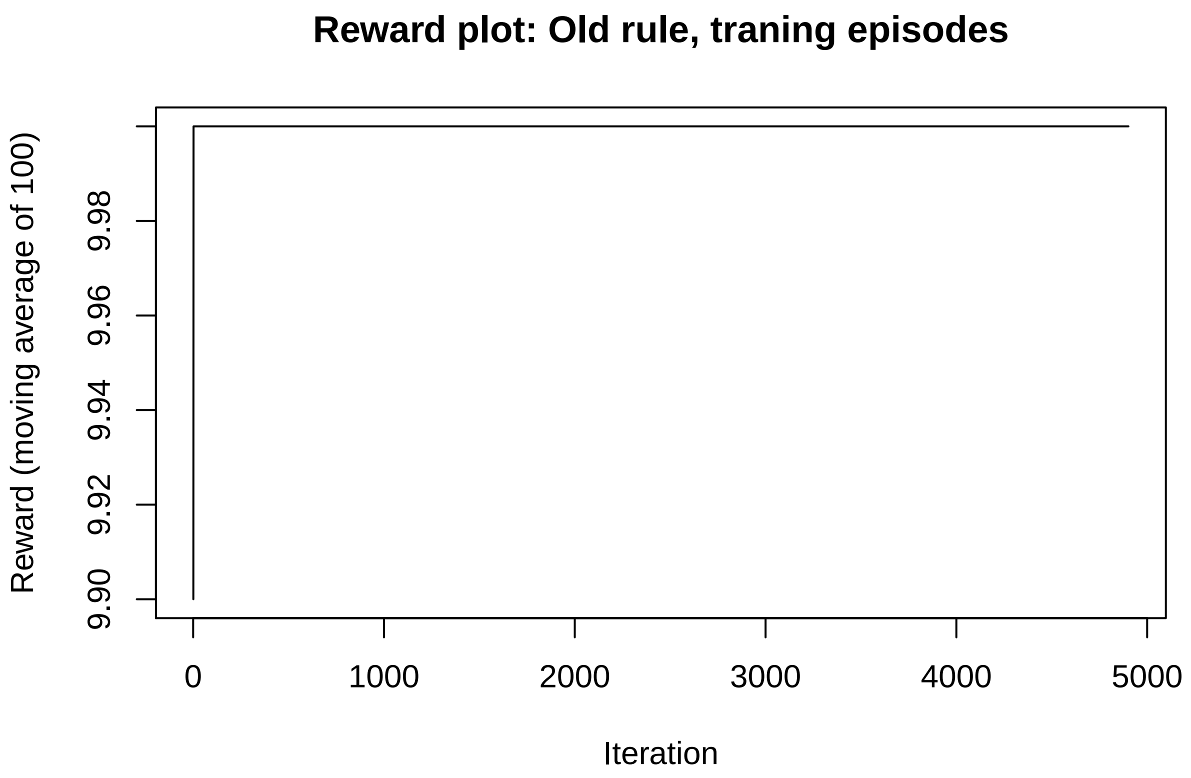
By setting alpha = 0, the robot will not learn anything new. Only using current policy. Also setting episilon = 0, the robot won't explore new ways and only follow the greedy policy. This can be interpreted as "testing".

The testing calculations are run in the previous chunk, but plotted here.

```
plot(MovingAverage(train_reward_new,100),type = "l", main="Reward plot: New rule, traning episodes", xla
```



```
plot(MovingAverage(train_reward,100),type = "l", main="Reward plot: Old rule, traning episodes", xlab="
```

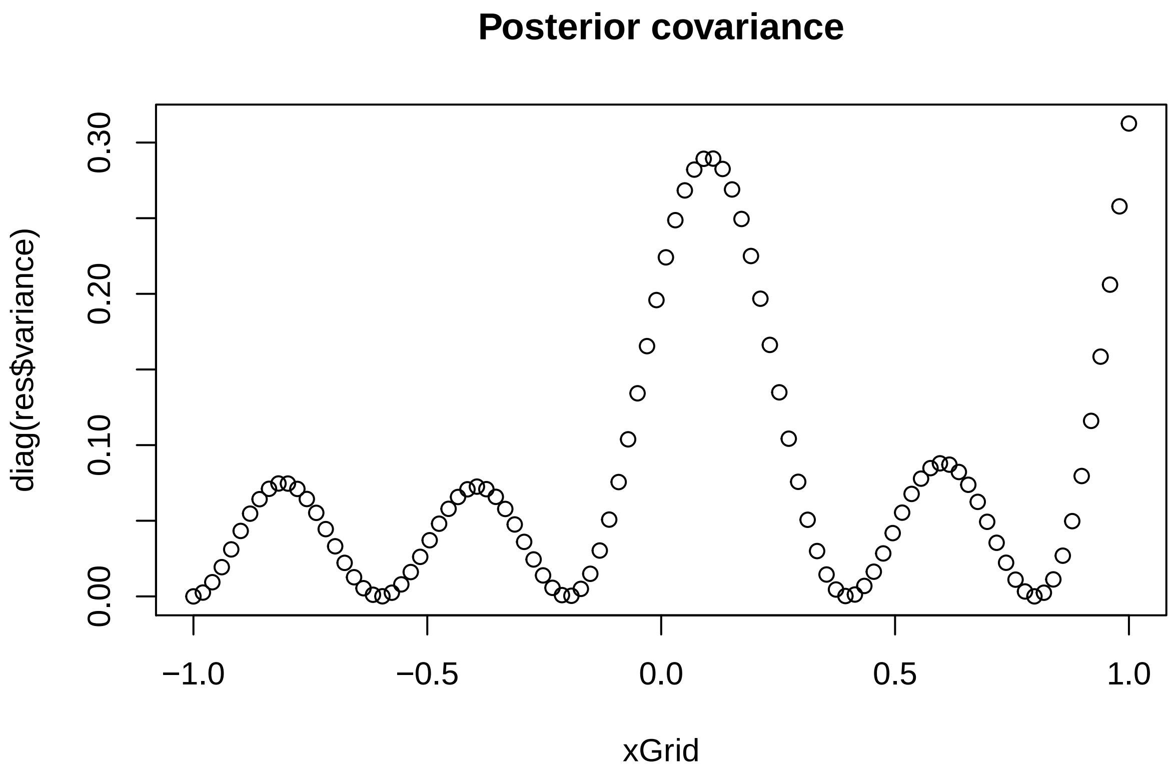


Both models has produced a good policy! They walk straight to reward 10! The number of iterations used was enough for the robot to learn a good policy.

3. Gaussian Processes

a) Posterior Covariance

```
# Covariance function
SquaredExpKernel <- function(x1,x2,sigmaF=1,l=3){
  n1 <- length(x1)
  n2 <- length(x2)
  K <- matrix(NA,n1,n2)
  for (i in 1:n2){
    K[,i] <- sigmaF^2*exp(-0.5*( (x1-x2[i])/l)^2 )
  }
  return(K)
}
# R & W Algorithm 2.1
posteriorGP = function(X, y, XStar, sigmaNoise, k, sigmaF, l){ # signaF NOT squared
  K = k(X,X, sigmaF^2, l)
  n = length(XStar)
  L = t(chol(K + sigmaNoise^2*diag(dim(K)[1])))
  kStar = k(X,XStar, sigmaF^2, l)
  alpha = solve(t(L), solve(L,y))
  FStar = t(kStar) %*% alpha
  v = solve(L, kStar)
  vf = k(XStar, XStar, sigmaF^2, l) - t(v)%*%v
  logmarglike = -t(y)%*%alpha/2 - sum(diag(L)) - n/2*log(2*pi)

  return(list("mean" = FStar,"variance" = vf,"logmarglike" = logmarglike))
}

sigmaF = 1
ell = 0.3
sigmaN = 0
x = c(-1.0, -0.6, -0.2, 0.4, 0.8)
y = c(0.768, -0.044, -0.940, 0.719, -0.664)

xGrid = seq(from=-1, to=1, length.out=100)

res = posteriorGP(X=x, y=y, XStar=xGrid, sigmaNoise = sigmaN, k = SquaredExpKernel, sigmaF = sigmaF, l=
plot(x = xGrid, y = diag(res$variance), main = "Posterior covariance")
```



```
SEKernel = function(x1, x2, ell = 1, sigmaf = 1){
  r = sqrt(sum((x1 - x2)^2))
  return(sigmaf^2*exp(-r^2/(2*ell^2)))
}
```

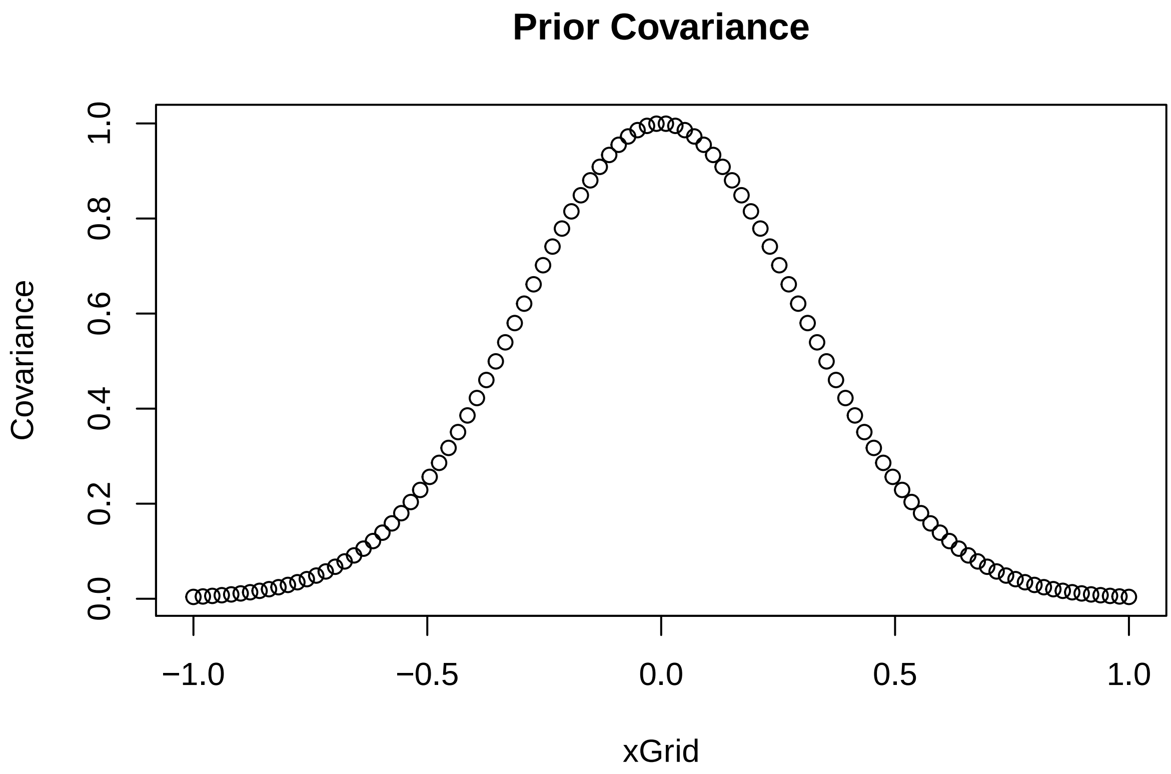
Covariance is 0 at training point because sigmaN = 0. This means that we are totally sure of the data at the training points. Hence, the model will produce means that go straight through the data points used for training.

We don't see decreasing posterior covariance because it is based on our training points. The model does not know if point far away should be correlated or not, even though the kernel itself believes so.

The prior covariance should be what the kernel itself produces.

Prior covariance

```r
Covariance = diag(SquaredExpKernel(xGrid, rep(0,length(xGrid)), sigmaF, ell))
plot(xGrid, Covariance, main="Prior Covariance")
```



b) Tullinge Temperatures

```r
time = seq(from=1, to=2190, by=5)
temps = temps[time]
day = rep(seq(from=1, to=361, by=5), times=6)

day_s = scale(day)
time_s = scale(time)
temps_s = scale(temps)
```

```
### Search grid for sigmaN ###
sigmaF = 20
ell = 0.2

top_sigmaN= 0
top_log = -1000000

for(sigmaN in seq(0.1, 1, length.out = 50)){
  log = posteriorGP(X=time_s, y=temps_s, XStar=time_s, sigmaNoise=sigmaN, k=SquaredExpKernel, sigmaF = 

  if(log > top_log){
    top_sigmaN = sigmaN
    top_log = log
  }
}
# Optimal parameters
top_sigmaN
```

```
## [1] 0.4857143
```

```
top_log
```

```
##           [,1]
## [1,] -1310.559
```

An “optimal” sigmaN was found to be around 0.48 for a log marginal likelihood of -1310.