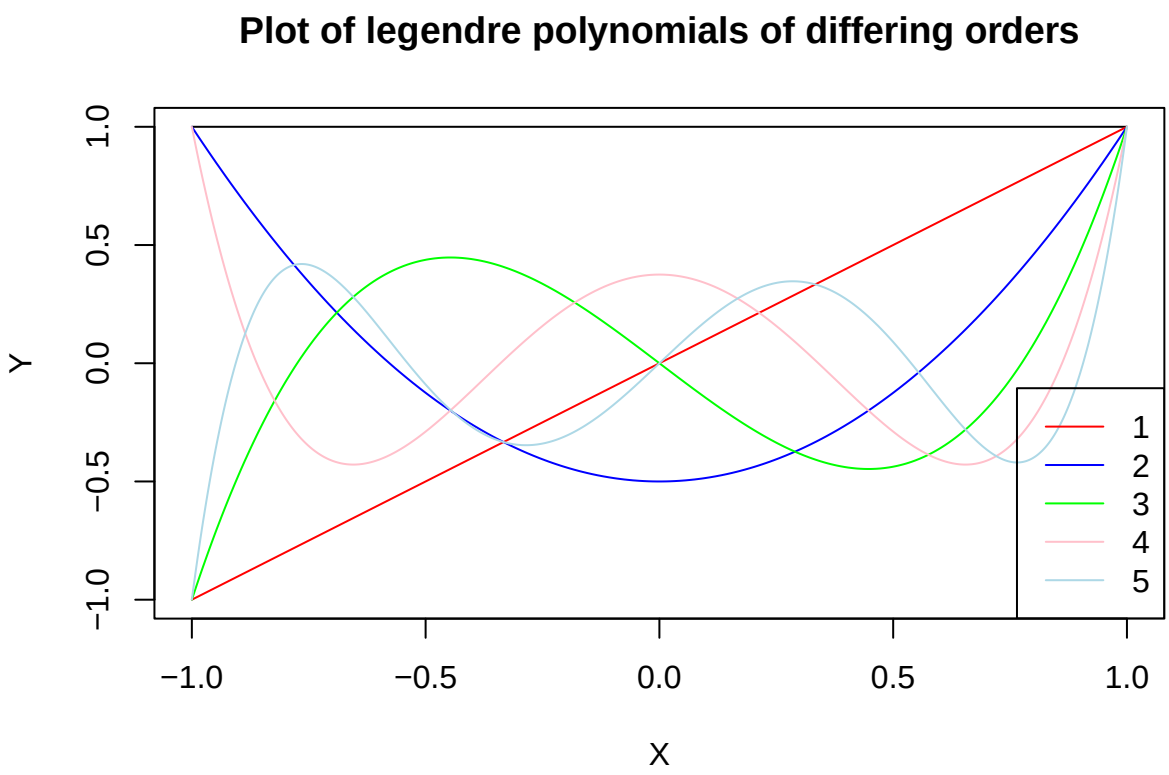
Assignment1

Michael Seebregts

2025-10-27

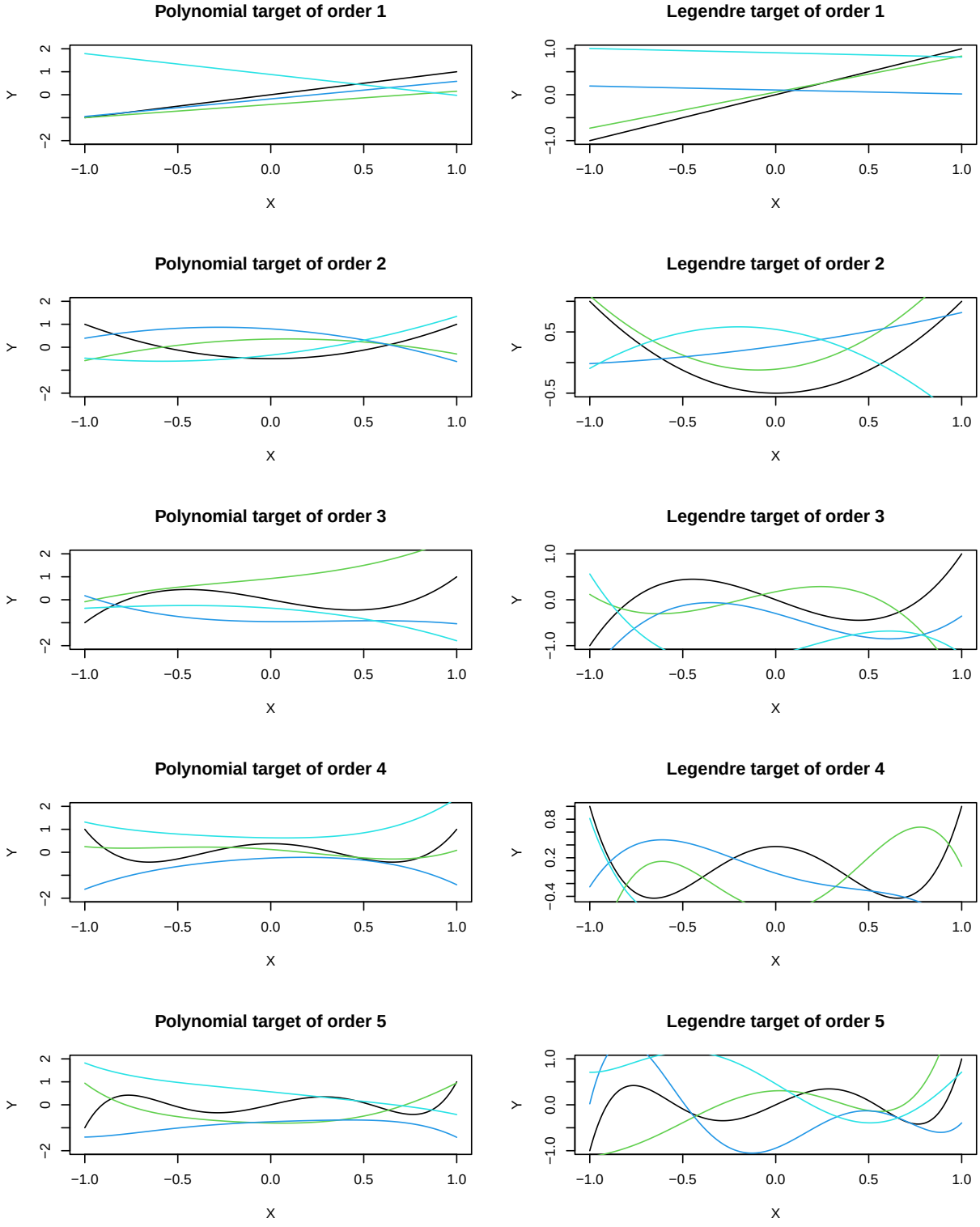
Question 1

1a



Can see that odd polynomials all start at -1, while even polynomials start at +1. The number of points of turning points is equal to degrees - 1. All polynomials end at +1

1b



Can see that around -1 and +1, the polynomial targets are much more limited in their ability to have multiple turning points, leading to them being nearly only increasing or only decreasing on the interval -1 to +1.

past order 2, the functions appear to be relatively similar. Meanwhile, the legendre targets are able to be much more wiggly between -1 and +1. However, this does make the legendre much more volatile than the polynomial over -1 to +1 from degree 3 on wards. Can see that the polynomial functions are much more stable even with the differing of the parameter values whereas the legendre are much more volatile with changing parameter values.

Question 2

2a

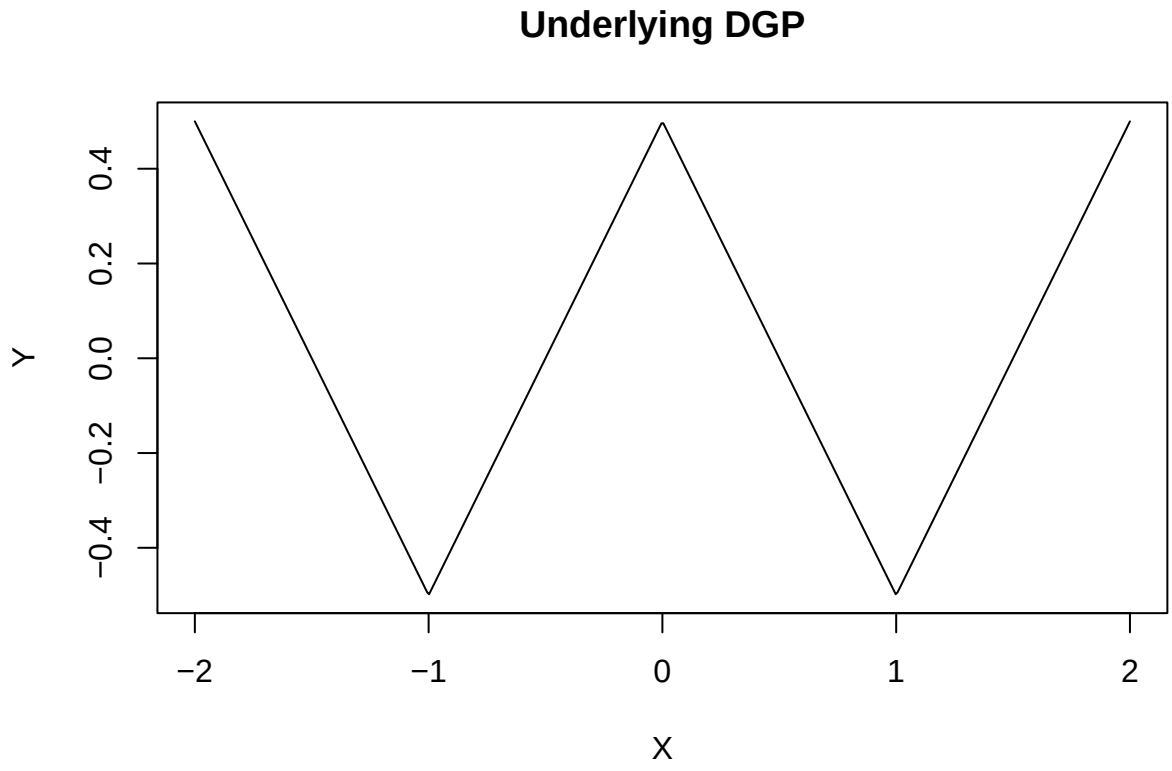
2b

2c

Question 3

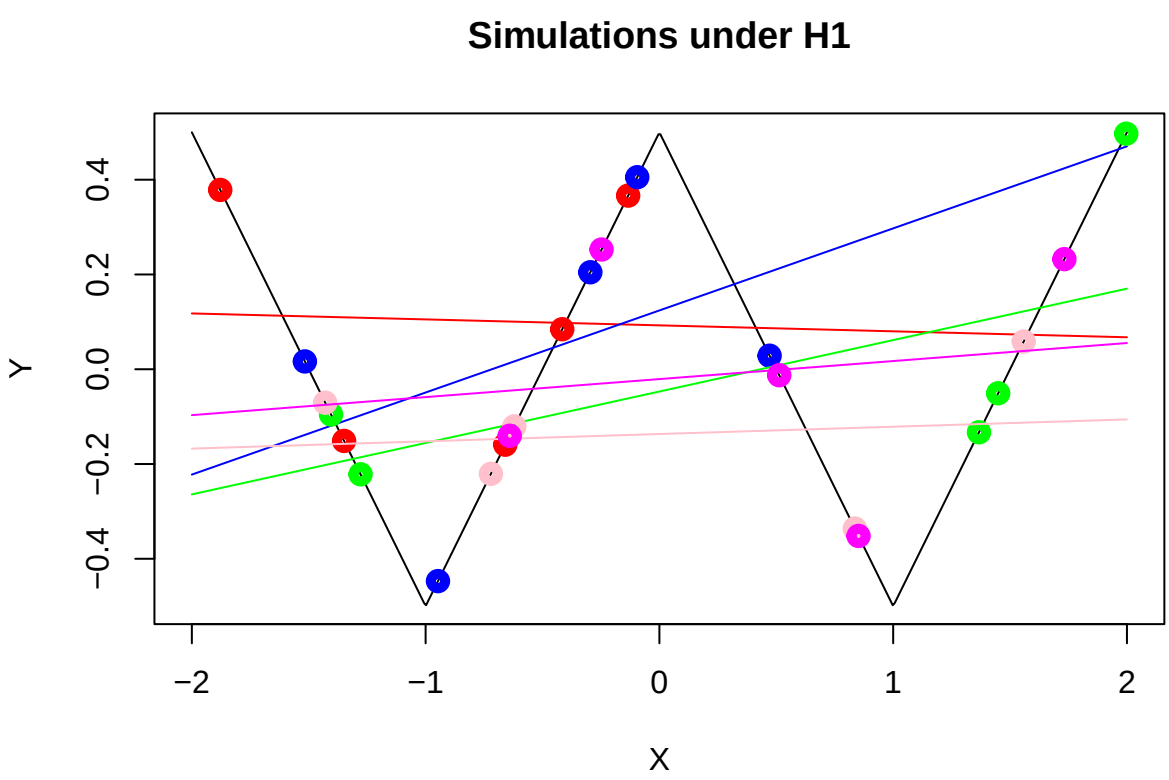
3a

```
set.seed(2023)

underlying = function(x)
{

  y = rep(0, length(x))
  for (j in 1:length(x))
  {
    if (x[j] < 0)
    {
      y[j] = abs(x[j] + 1) - 0.5
    }

    if (x[j] >= 0)
    {
      y[j] = abs(x[j] - 1) - 0.5
    }
  }

  return( y)
}

DGP = function(N, sigma)
{
  x = runif(N, -2, 2)
  e = rnorm(N, 0, sigma)
  y = underlying(x) + e
  return(list(x = x, y = y))
}

xBase = seq(-2, 2, length.out = 500)
runUnderlying = underlying(xBase)
plot(xBase, runUnderlying, type = "l", ylab = "Y", xlab = "X", main = "Underlying DGP")
```



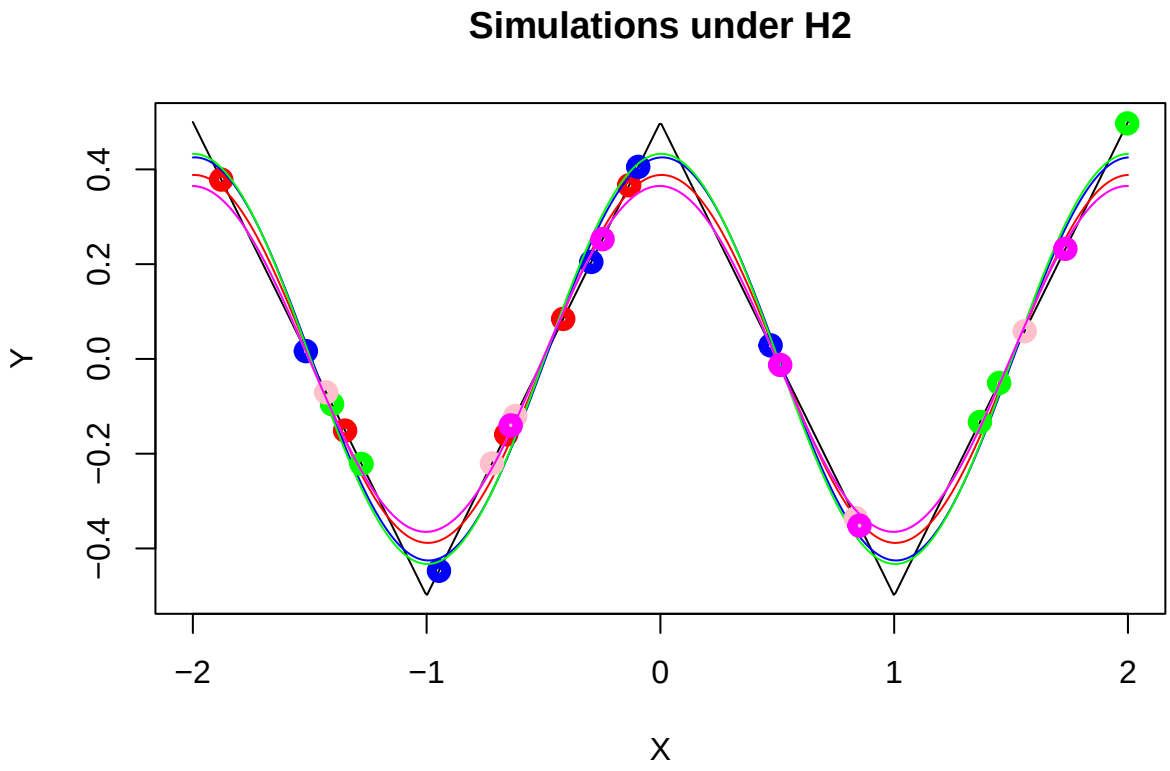

```
hyp1 = function(x, xPred, y, color, plt)
{
  mod = lm(y ~ x)

  pred = predict(mod, newdata = data.frame(x = xPred))

  if (plt)
  {
    lines(xPred, pred, col = color)
  }
  return(invisible(pred))
}

hyp2 = function(x, xPred, y, color, plt)
{
  mod = lm(y ~ sin(pi*x) + cos(pi*x) - 1)
  pred = predict(mod, newdata = data.frame(x = xPred))
  if (plt)
  {
    lines(xPred, pred, col = color)
  }
  return(invisible(pred))
}

dgp1 = DGP(5, 0)
dgp2 = DGP(5, 0)
```

```r
dgp3 = DGP(5, 0)
dgp4 = DGP(5, 0)
dgp5 = DGP(5, 0)

plot(xBase, runUnderlying, type = "l", ylab = "Y", xlab = "X", main = "Simulations under H1")
hyp1(dgp1$x, xBase, dgp1$y, "red", TRUE)
points(dgp1$x, dgp1$y, col = "red", type = "p", lwd = 5)
hyp1(dgp2$x, xBase, dgp2$y, "blue", TRUE)
points(dgp2$x, dgp2$y, col = "blue", type = "p", lwd = 5)
hyp1(dgp3$x, xBase, dgp3$y, "green", TRUE)
points(dgp3$x, dgp3$y, col = "green", type = "p", lwd = 5)
hyp1(dgp4$x, xBase, dgp4$y, "pink", TRUE)
points(dgp4$x, dgp4$y, col = "pink", type = "p", lwd = 5)
hyp1(dgp5$x, xBase, dgp5$y, "magenta", TRUE)
points(dgp5$x, dgp5$y, col = "magenta", type = "p", lwd = 5)
```




```r
plot(xBase, runUnderlying, type = "l", ylab = "Y", xlab = "X", main = "Simulations under H2")
hyp2(dgp1$x, xBase, dgp1$y, "red", TRUE)
points(dgp1$x, dgp1$y, col = "red", type = "p", lwd = 5)
hyp2(dgp2$x, xBase, dgp2$y, "blue", TRUE)
points(dgp2$x, dgp2$y, col = "blue", type = "p", lwd = 5)
hyp2(dgp3$x, xBase, dgp3$y, "green", TRUE)
points(dgp3$x, dgp3$y, col = "green", type = "p", lwd = 5)
hyp2(dgp4$x, xBase, dgp4$y, "pink", TRUE)
points(dgp4$x, dgp4$y, col = "pink", type = "p", lwd = 5)
```

```
hyp2(dgp5$x, xBase, dgp5$y, "magenta", TRUE)
points(dgp5$x, dgp5$y, col = "magenta", type = "p", lwd = 5)
```



3b

H2 clearly seems like the better fit, with the sin being able to capture the wave like nature much better, although not able to capture the sharp changes. H2 is able to go through many more points that it is fitted to, while H1 is generally not able to. H2 is also much less volatile than H1 under different simulations.

3c

```
bias_var = function(N, M, hyp, dx, sig)
{
  xx_lat = seq(-2, +2, dx)
  Nx = length(xx_lat)

  gBar = xx_lat*0

  G_D = matrix(0, M, Nx)

  testError = 0
  for (i in 1:M)
```

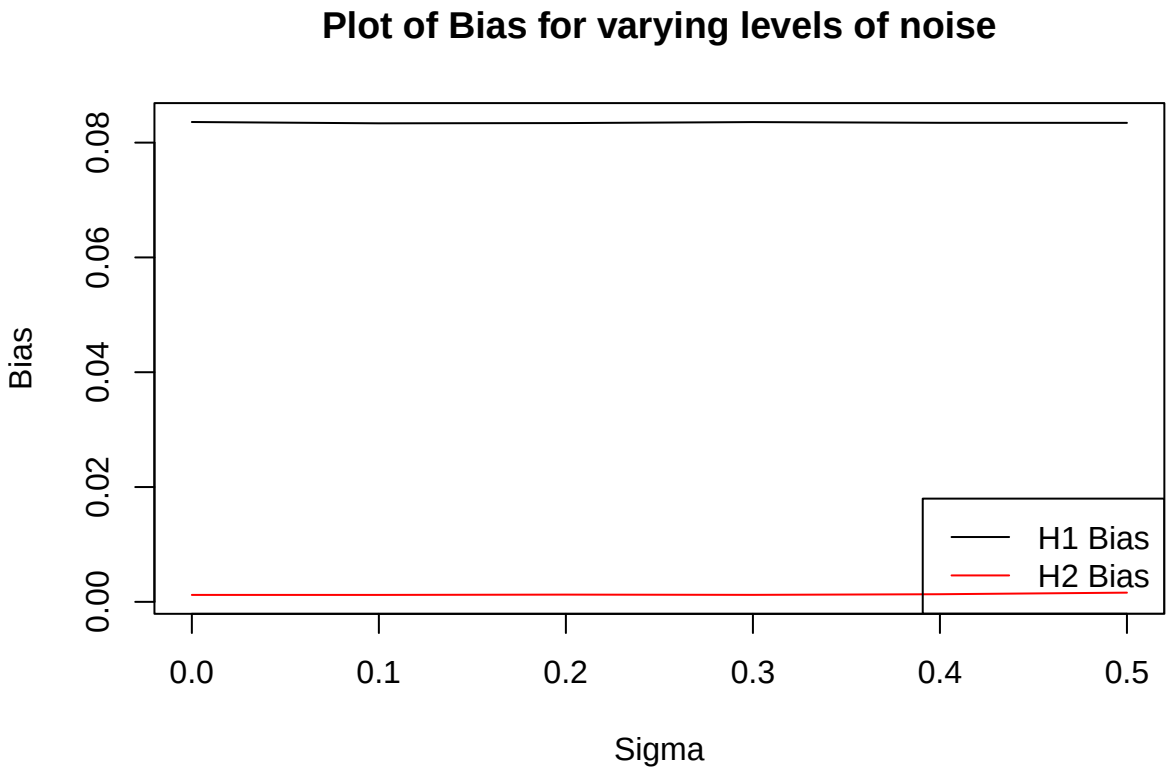
```
  {
    dat = DGP(N, sig)
    x = dat$x
    y = dat$y

    if (hyp == 1)
    {
      mod = lm(y ~ x)

    }
    if (hyp == 2)
    {
      mod = lm(y ~ sin(pi*x) + cos(pi*x) - 1)
    }
    g_D = predict(mod, newdata = data.frame(x = xx_lat))

    G_D[i, ] = g_D
    gBar = gBar + g_D

    datOOS = DGP(N, sig)
    pred_oos = predict(mod, data.frame(x = datOOS$x))
    testErrorD = mean((datOOS$y - pred_oos)^2)
    testError = testError + testErrorD
  }

  gBar = gBar/M

  phi_x = 1/4
  bias2 = sum((gBar - underlying(xx_lat))[-Nx]^2*phi_x*dx)
  bias2


  testError = testError/M

  ones = matrix(1, M, 1)
  varAtX = colSums((G_D - ones%*%gBar)^2)/M # calculate variance at each point x
  var = sum(varAtX[-Nx]*phi_x*dx) # Usual riemann integral
  var

  both = bias2 + var

  return(list("testError" = testError, "bias" = bias2, "variance" = var, "Var + Bias^2" = both))
}

biasVarH1 = as.data.frame(bias_var(5, 1000, 1, 1/100, 0))
biasVarH2 = as.data.frame(bias_var(5, 1000, 2, 1/100, 0))

biasVarH1 = round(biasVarH1, 4)
biasVarH2 = round(biasVarH2, 4)

kable(biasVarH1)
```

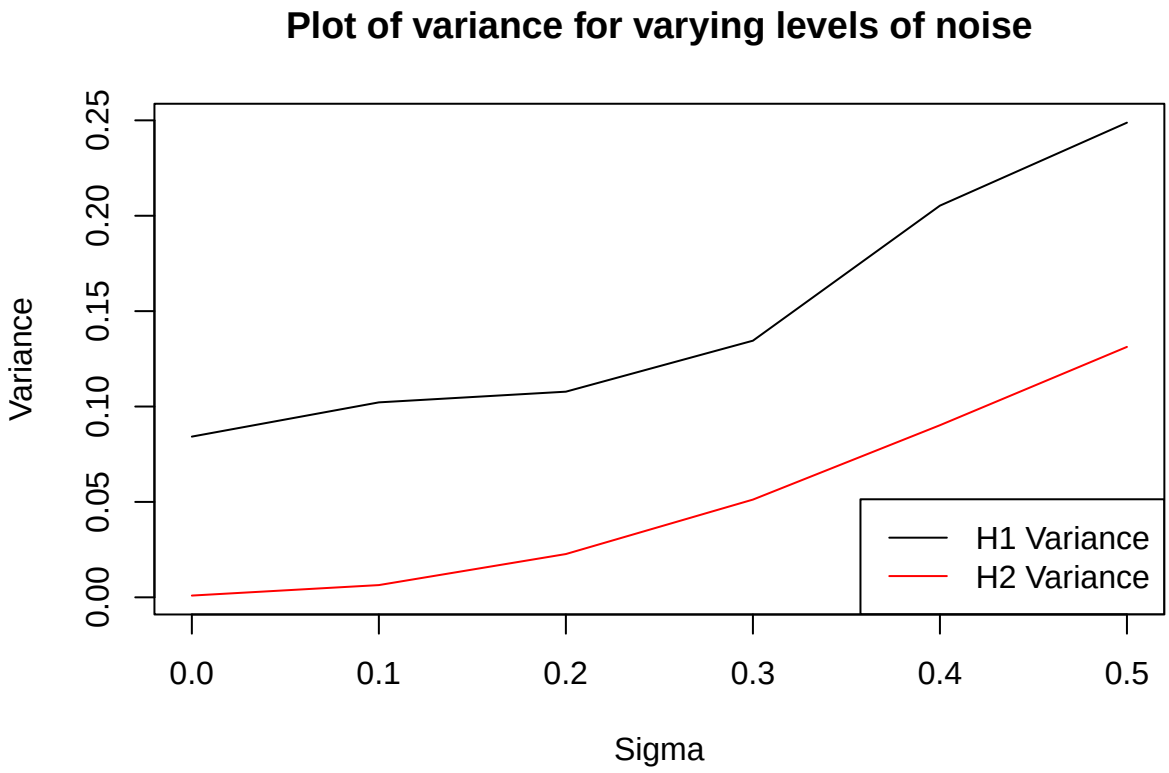
testError	bias	variance	Var. . . Bias.2
0.1766	0.0837	0.0864	0.1701

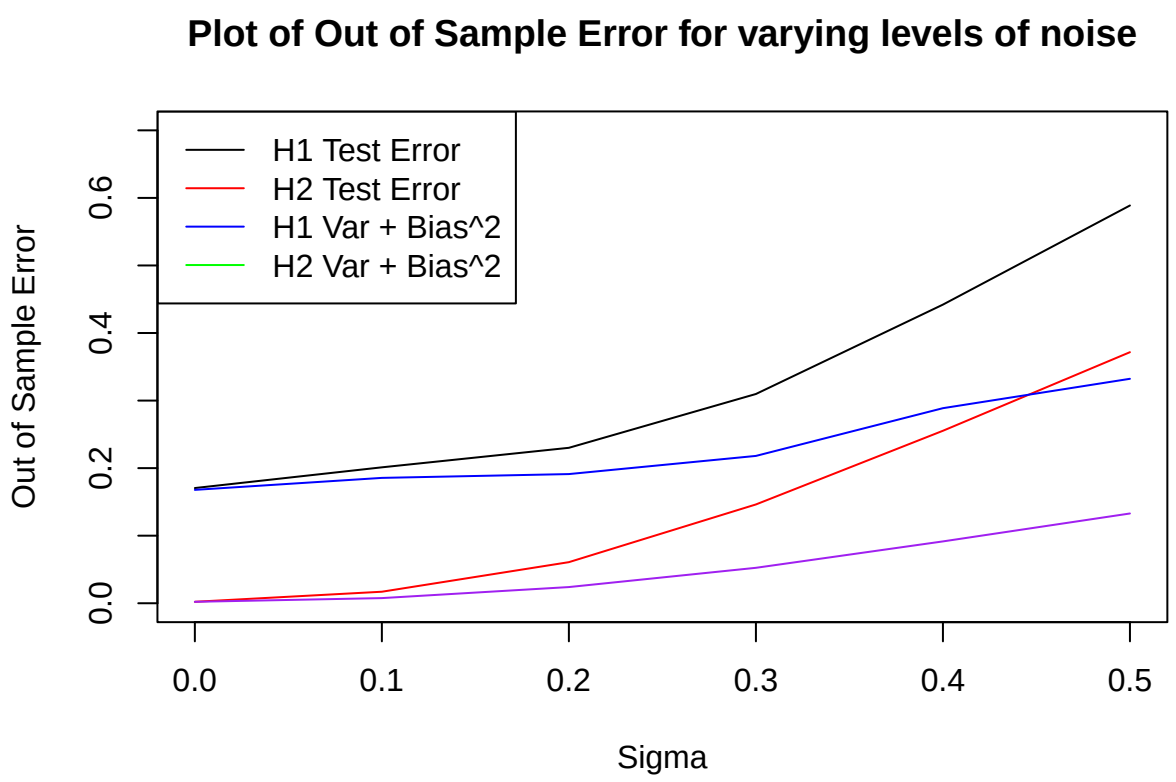
```
kable(biasVarH2)
```

testError	bias	variance	Var. . . Bias.2
0.002	0.0012	8e-04	0.002

3d


Plot of Bias for varying levels of noise
Bias
0.00
0.02
0.04
0.06
0.08
0.0
0.1
0.2
0.3
0.4
0.5
Sigma
H1 Bias
H2 Bias


Plot of variance for varying levels of noise
Variance
0.00
0.05
0.10
0.15
0.20
0.25
0.0
0.1
0.2
0.3
0.4
0.5
Sigma
H1 Variance
H2 Variance



3e

We can see that bias, variance and expected out of sample error are always greater under H1 than H2. The distance between the bias, variance and test error appear to have constant distances between H1 and H2 for varying levels of noise.

The bias stays essentially constant for varying levels of Sigma for both H1 and H2. This is because it depends on the true underlying function, and the average function. If there are sufficient functions fit, the average should stay relatively constant under different levels of noise. The bias under H2 is lower, due to it being closer to the underlying, minimizing $\bar{g}(x) - g(x)$ where $\bar{g}(x)$ is the average fit and g is the true function.

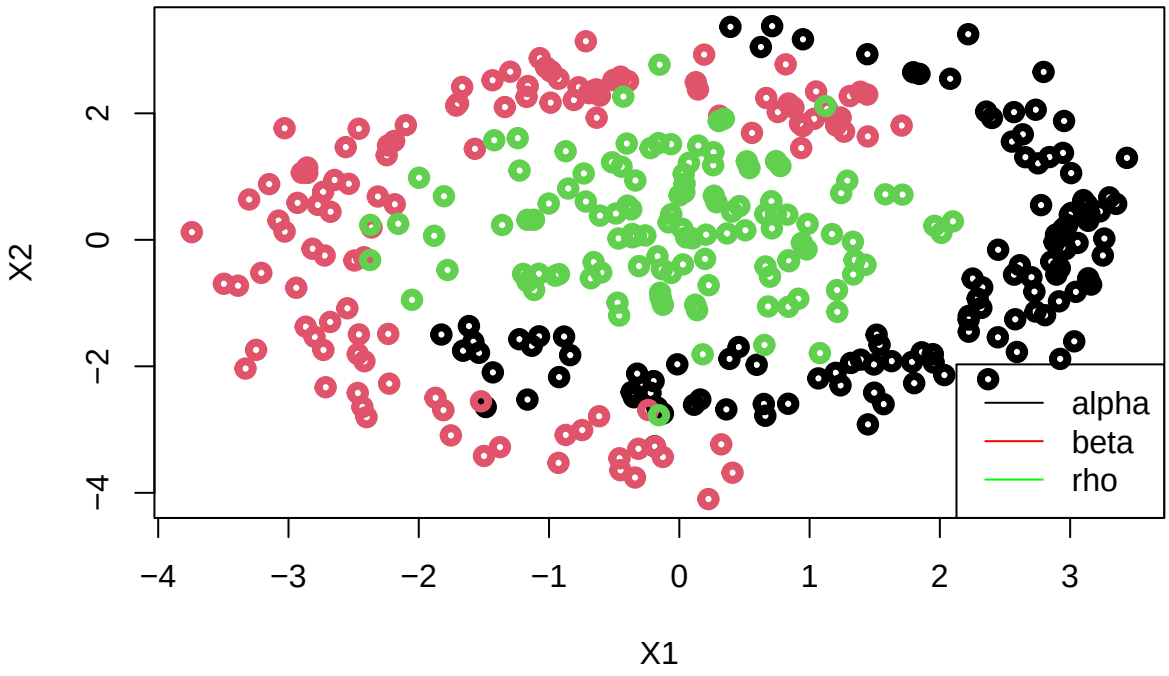
The variance is increasing for increasing levels of noise. This is because as we increase the noise the functions that can be fit increase and differ from the average function by more, i.e. $g^D(x)$ varies more and increases $g^D(x) - \bar{g}(x)$. The variance for H2 is lower, as the shape of the function remains similar no matter the fitting, which means that $g^D(x) - \bar{g}(x)$ will be lower than H1, $g^D(x)$ can vary much more.

Similarly, the test error increases for increasing levels of noise, can see this for both ways of measuring Expected Out of Sample Error. When estimated test error with bias and variance, due to bias staying the same while variance increases for both H1 and H2, we can explain the test error increasing. When calculating test error directly, as the noise increases $g^D(x)$ while change more variable, resulting in it differing from f(x) more, meaning $(\bar{g}(x) - f(x))^2$ will increase. H2 has a lower test error due to the sin and cos function simply follow the underlying distribution much more closely.

Question 4

4a

```
library(colorspace)

color.gradient = function(x, colors=c('cyan','orange','magenta'), colsteps=50)
{
  colpal = colorRampPalette(colors)
  return( colpal(colsteps)[ findInterval(x, seq(min(x),max(x), length=colsteps)) ] )
}

dat = read.table("Collider_Data_2022.txt", h = TRUE,stringsAsFactors =TRUE)

plot(dat$X1, dat$X2, col = max.col(dat[, 3:5]), type = "p", lwd = 4, ylab = "X2", xlab = "X1")
legend("bottomright", legend = c("alpha", "beta", "rho"), col = c("black", "red", "green"), lty = 1)
```



The data is clearly not linearly separable and therefore non linear machinery is likely to perform better. Likely able to partition the feature space better.

4b

```
X = cbind(dat[, 1:2])
Y = dat[, 3:5]

softMax = function(x)
{
  denom = sum(exp(x))
  softVec = rep(0, length.out = length(x))
  for(i in 1:length(x))
  {
    softVec[i] = exp(x[i])/denom
  }
  return(softVec)
}

softMaxMatrix = function(X)
{
  softMat = apply(X, 2, softMax)
  return(softMat)
}
```

4c

```
neural_net = function(X,Y,theta,m,nu)
{
  # Infer dimensions:
  N = dim(X)[1]
  p = dim(X)[2]
  q = dim(Y)[2]

  d = c(p,m,q)

  # Populate weight and bias matrices:
  index = 1:(d[1]*d[2])
  W1    = matrix(theta[index],d[1],d[2])
  index = max(index)+(1:(d[2]*d[3]))
  W2    = matrix(theta[index],d[2],d[3])
  index = max(index)+(1:d[2])
  b1    = matrix(theta[index],d[2],1)
  index =max(index)+(1:d[3])
  b2    = matrix(theta[index],d[3],1)

  ones = matrix(1,N,1)
  # Evaluate the updating equation in matrix form
  #for(i in 1:N)
  #{
  A0 = t(X)
  A1 = tanh(t(W1)%*%A0+b1%*%t(ones))

  #print(head(t(W2)%*%A1+b2%*%t(ones)))
  A2 = softMaxMatrix(t(W2)%*%A1+b2%*%t(ones))
  #print(head(A2))
```

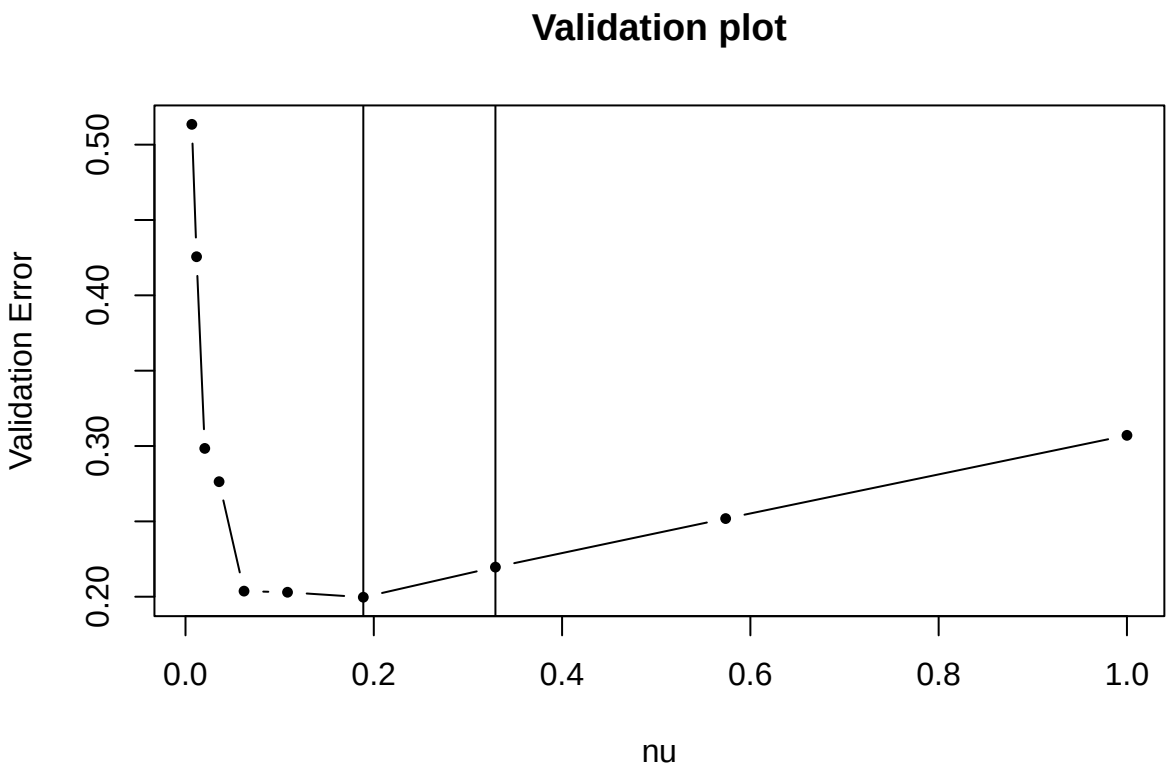
```
  #}
  yHat = A2
  # print(dim(yHat))
  # print(A2)

  error = rep(NA,N)    #-(Y*log(pi_hat)+(1-Y)*log(1-pi_hat))
  logA2 = log(A2)
  crossEnt = t(Y)*logA2
  #print(head(crossEnt))
  error = apply(crossEnt, 2, sum)
  # Evaluate an appropriate objective function and return some predictions:
  E1 = (-1)*sum(error)/N
  E2 = E1+nu/N*(sum(W1^2)+sum(W2^2))
  # Return a list of relevant objects:
  return(list(A2 = A2,A1 = A1, E1 = E1, E2 = E2))
}
```

Potential problem could come in the evaluation of the log in the softmax function. If the predicted yhat is 0, while the actual y is 1, the log will become -∞ becoming not evaluable.

4d

```
m   = 10
p     = dim(X)[2]
q     = dim(Y)[2]
npars = p*m+m*q+m+q
theta_rand = runif(npars,-1,+1)
res = neural_net(X,Y,theta_rand,m,0)

set.seed(2022)

M  = 100
x1 = seq(min(X[1, ]),max(X[1, ]),length = M)
x2 = seq(min(X[2, ]),max(X[2, ]),length = M)
xx1 = rep(x1,M)
xx2 = rep(x2, each = M)

XX = cbind(xx1,xx2)
YY = matrix(1,M^2,3)


N      = dim(X)[1]
set    = sample(1:N,0.5*N,replace = FALSE)
Xtrain = X[set,,drop = FALSE]
Ytrain = Y[set,,drop = FALSE]
Xval   = X[-set,,drop = FALSE]
Yval   = Y[-set,,drop = FALSE]


# nu = 5
obj_pen = function(pars)
{
```

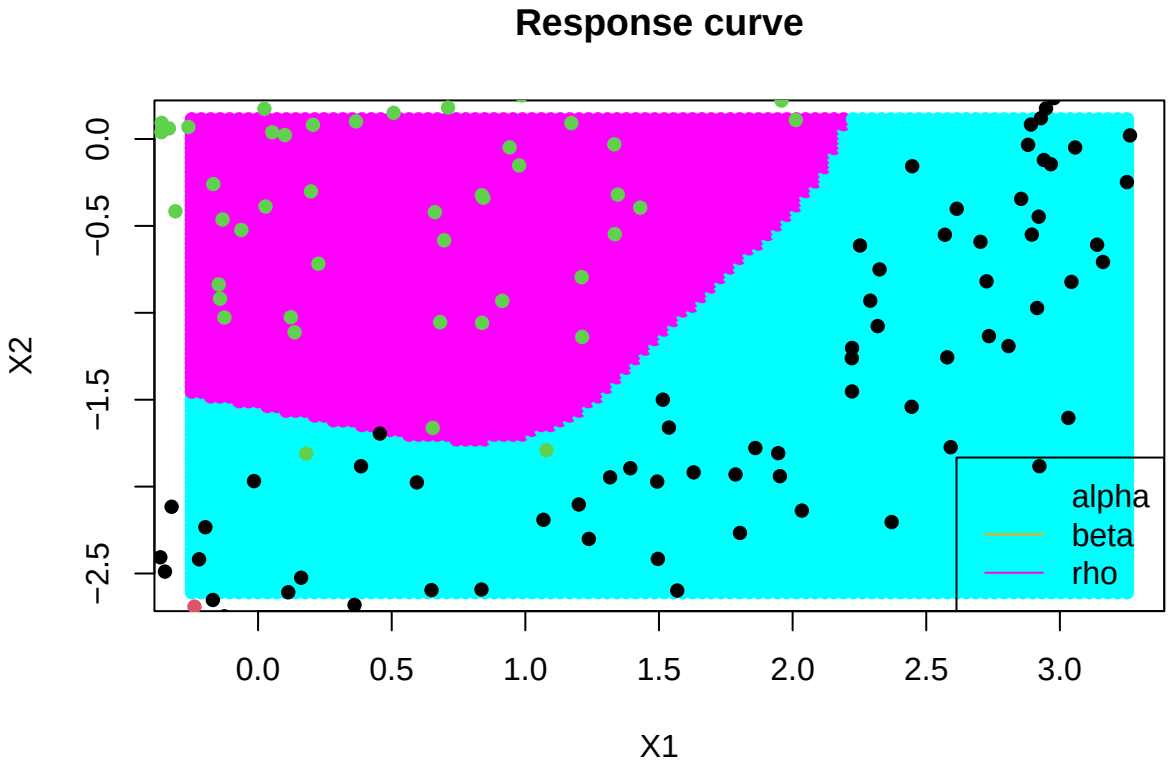
```
  res = neural_net(Xtrain,Ytrain,pars,m,nu)
  return(res$E2)
}
# obj_pen(theta_rand)


n_nus     = 10
nu_seq    = exp(seq(-5,0,length = n_nus))
val_error = rep(NA,n_nus)
for(i in 1:n_nus)
{
  nu  = nu_seq[i]
  # Fit the neural network using a standard optimizer in R:
  res_opt = nlm(obj_pen,theta_rand, iterlim = 1000)

  res_val      = neural_net(Xval,Yval,res_opt$estimate,m,0)
  val_error[i] = res_val$E1

  # Draw a response curve over the 2D input space to see
  # what pattern the neural network predicts

  # res_fitted = neural_net(XX,YY,res_opt$estimate,m,0)
  #
  # plot(XX[,2]~XX[,1], pch = 16, col = color.gradient(max.col(t(res_fitted$A2))))
  # points(X$X2~X$X1, col = max.col(Y), pch = 16)
  #
  # print(paste0('Validation run ', i,'| nu = ',round(nu,5)))
}

plot(val_error~nu_seq,type = 'b',pch = 16,cex = 0.75, ylab = "Validation Error", xlab = "nu", main = "Va

wh = which.min(val_error)

nu_final = nu_seq[wh]
nu_final_conservative = nu_seq[wh+1]
# print(nu_final)
# print(nu_final_conservative)
abline(v = nu_final)
abline(v = nu_final_conservative)
```



Looking at the plot, we can see that at nu = 0.188, there is a minimum in the validation error. This can be chosen as our validation level. In order to be conservative, we choose one level up at nu = 0.33

4e

```
obj_final = function(pars)
{
  res = neural_net(X,Y,pars,m,nu_final_conservative)
  return(res$E2)
}
res_opt_final = nlm(obj_final,theta_rand, iterlim = 1000)


res_fitted = neural_net(XX,YY,res_opt_final$estimate,m,0)

plot(XX[,2]~XX[,1], pch = 16, col = color.gradient(max.col(t(res_fitted$A2))), ylab = "X2", xlab = "X1"
points(X$X2~X$X1, col = max.col(Y), pch = 16)
legend("bottomright", legend = c("alpha", "beta", "rho"), col = c("cyan", "orange", "magenta"), lty = 1
```



New points can be classified by measuring their X1 and X2 variables. If the X1 and X2 is in cyan region, they can be labelled as α particles, if they are in the pink orange region, they can be labelled as β particles and if they are in the pink region, they can be labelled as ρ particles.