Problem Set

Alberto José Díaz Cuenca and Ignacio Almodóvar Cárdenas

21/3/2022

Exercise 5.11.

The challenger.txt dataset contains information regarding the state of the solid rocket boosters after launch for 23 shuttle flights prior the Challenger launch. Each row has, among others, the variables fail.field (indicator of whether there was an incident with the O-rings), nfail.field (number of incidents with the O-rings), and temp (temperature in the day of launch, measured in Celsius degrees).

a. Fit a local logistic regression (first degree) for fails.field ~ temp, for three choices of bandwidths: one that oversmooths, another that is somehow adequate, and another that undersmooths. Do the effects of temp on fails.field seem to be significant?

First of all, we start by reading the dataset "challenger.txt". We have stored the response variable fail.field in a variable called "Y" and the predictor variable temp in "X".

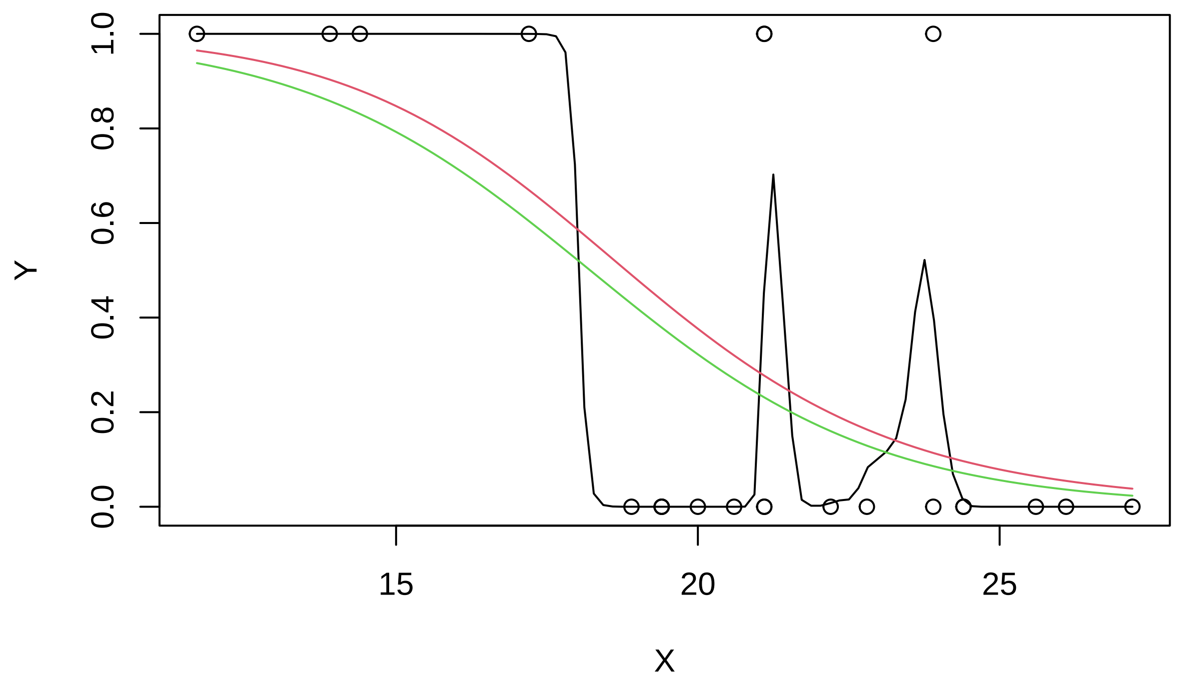
We are going to employ locfit in order to fit the logistic regression models. Here, the bandwidth needs to be specified within the "lp" argument.

We started with the lowest bandwidth allowed by the model, which is equal to h=0.22. That can allow us to believe that it will undersmooth the function. To find a somehow adequate bandwith, we have used h=0.6, which creates an estimation that seems correct. Finally, we have used a bandwith h=2, which assures us an oversmoothing.

```r
h=0.22
fit_locfit <- locfit::locfit(Y ~ locfit::lp(X, deg = 1, nn = h),
                              family = "binomial", kern = "gauss")
h2 = 1.5
fit_locfit2 <-locfit::locfit(Y ~ locfit::lp(X, deg = 1, nn = h2),
                                family = "binomial", kern = "gauss")

h3 = 10
fit_locfit3 <-locfit::locfit(Y ~ locfit::lp(X, deg = 1, nn = h3),
                                family = "binomial", kern = "gauss")
```

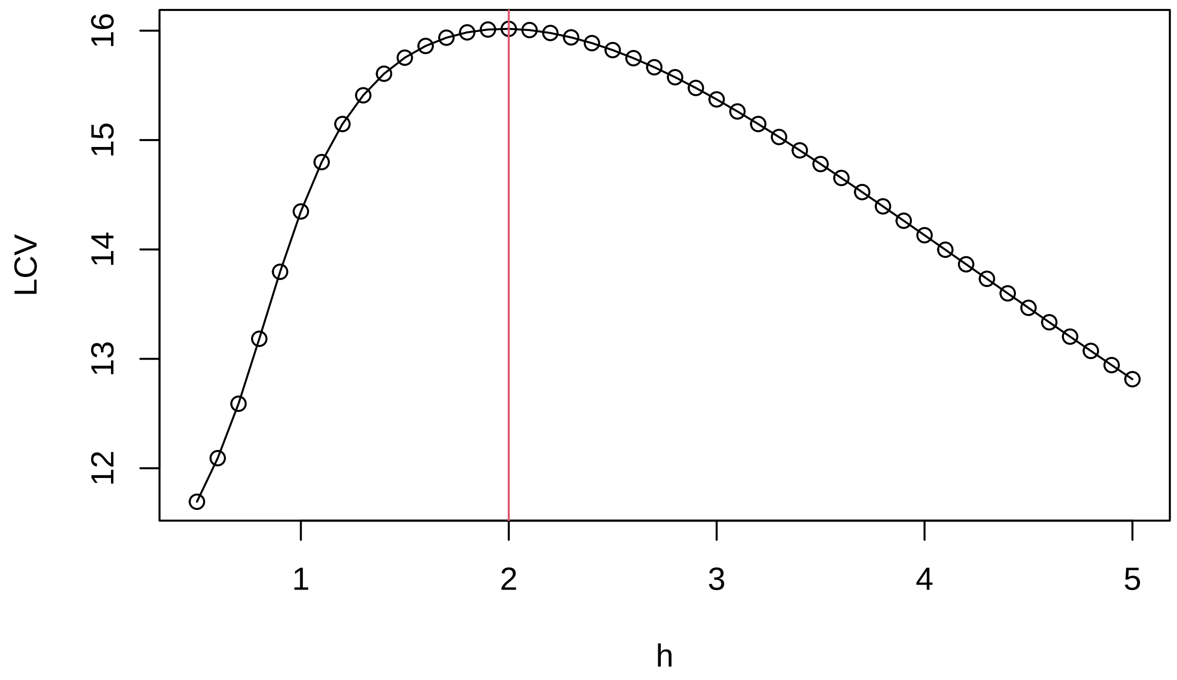
Plotting the results.

```r
plot(fit_locfit, col=1) + lines(fit_locfit2, col=2)  + lines(fit_locfit3, col=3) + points(X,Y, type="p"
```



b. Obtain hLCV and plot the LCV function with a reasonable accuracy.

```
# Exact LCV
h <- seq(0.5, 5, by = 0.1)
n=23
suppressWarnings(
  LCV <- sapply(h, function(h) {
  sum(sapply(1:n, function(i) {
    K <- dnorm(x = X[i], mean = X[-i], sd = h)
    nlm(f = function(beta) {
      -sum(K * (Y[-i] * (beta[1] + beta[2] * (X[-i] - X[i])) -
                  log(1 + exp(beta[1] + beta[2] * (X[-i] - X[i])))))
      }, p = c(0,0))$minimum
    }))
  })
)
plot(h, LCV, type = "o") + abline(v = h[which.max(LCV)], col = 2)
```



```
h[which.max(LCV)]
```

```
## [1] 2
```

For this section we have used the function that computes the Least Cross Validation bandwidth estimator through likelihood cross validation. The function basically minimizes the regression model thanks to the nlm() function, using p=c(0,0) as a starting point. Both the plot and the h[which.max(LCV)] return an estimation of h=2 (relatively close to the somehow adequate h=1.5 that we used in the previous section).

c. Using hLCV, predict the probability of an incident at temperatures -0.6 (launch temperature of the Challenger) and 11.67 (specific recommendation by the vice president of engineers).

```
hlcv <- 2
fit_locfit_lcv <- locfit::locfit(Y ~ locfit::lp(X, deg = 1, nn = hlcv),
                             family = "binomial", kern = "gauss")
prediction <- predict(fit_locfit_lcv, c(-0.6,11.67))
prediction
```

```
## [1] 0.9998206 0.9535512
```

We have obtained probabilities of 0.9998206 and 0.9535512 respectively for -0.6 and 11.67 degrees. Obviously, it is highly probable to have an incident with any of these temperatures. If we take a look at the plot of the regression model, we can see that, for temperatures under approximately 15 degrees, the response variable will always return "1" as the outcome, i.e., an incident occurring. Therefore, it holds that the probabilities obtained for such low temperatures are these high.

d. What are the local odds at -0.6 and 11.67? Show the local logistic models about these points, in spirit of Figure 5.1, and interpret the results.

Now, we can compute the local odds as follows:

```
local_odds1 <- prediction[1]/(1-prediction[1])
local_odds1
```

```
## [1] 5573.327
```

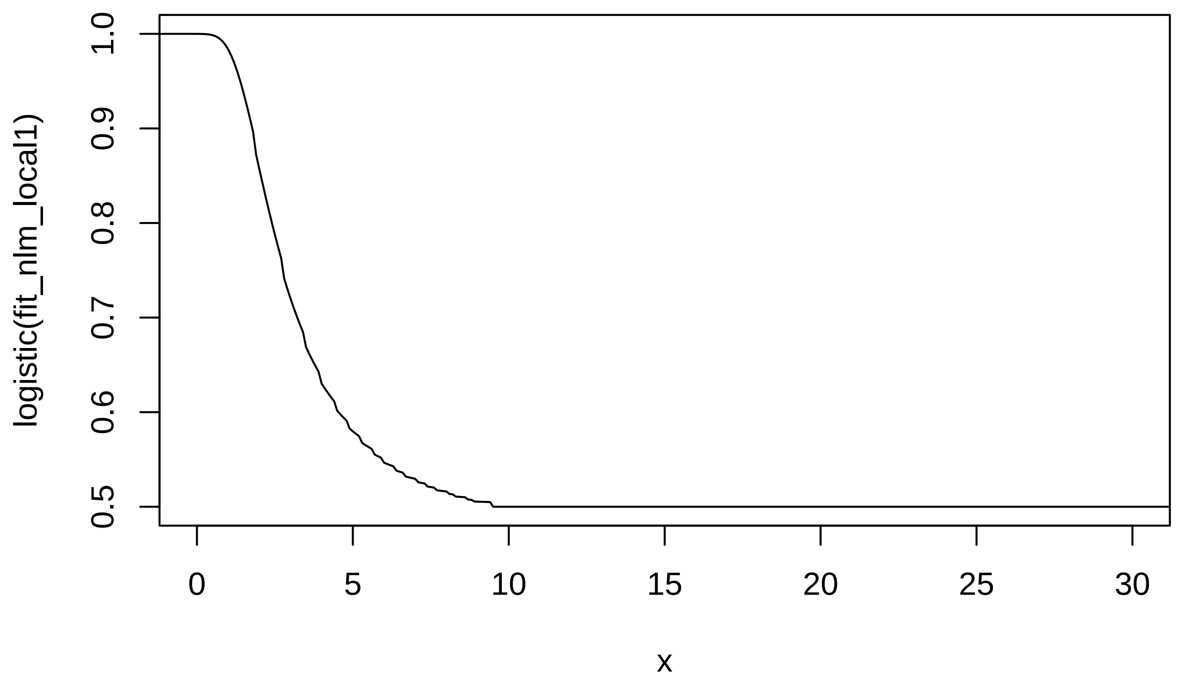
```
local_odds2 <- prediction[2]/(1-prediction[2])
local_odds2
```

```
## [1] 20.52908
```

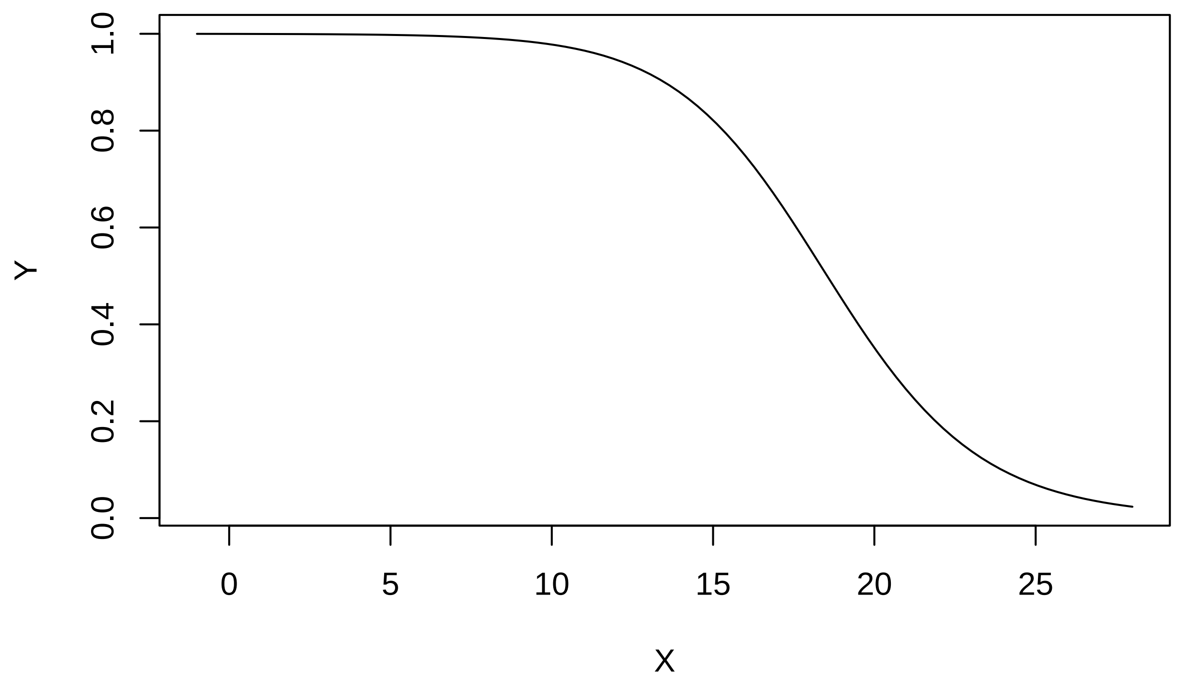
In order to show the local logistic model for these points, we need to give a different approach to implement the logistic models, this time using glm.fit().

First, we use X=-0.6 and Y=1.

```
logistic <- function(x) 1 / (1 + exp(-x))
Xev = -0.6
Yev=1
x = seq(-3, 40, by = 0.1)
suppressWarnings(
  fit_nlm_local1 <- sapply(x, function(x) {
    K <- dnorm(x = x, mean = Xev, sd = 1.9)
    nlm(f = function(beta) {
      -sum(K * (Yev * (beta[1] + beta[2] * (Xev - x)) -
                  log(1 + exp(beta[1] + beta[2] * (Xev - x)))))
    }, p = c(0, 0))$estimate[1]
  })
)
plot(x,logistic(fit_nlm_local1), type="l", xlim = c(0,30)) + plot(fit_locfit_lcv, xlim = c(-1,28))
```

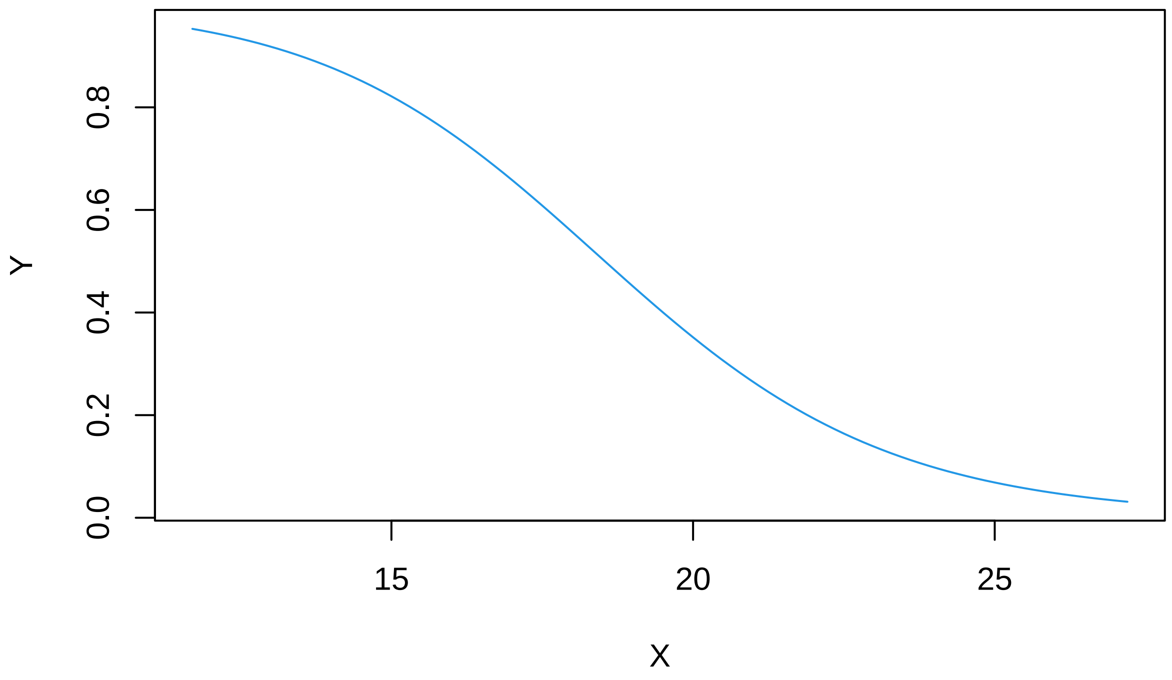


logistic(fit_nlm_local1)
1.0
0.9
0.8
0.7
0.6
0.5
0
5
10
15
20
25
30
x

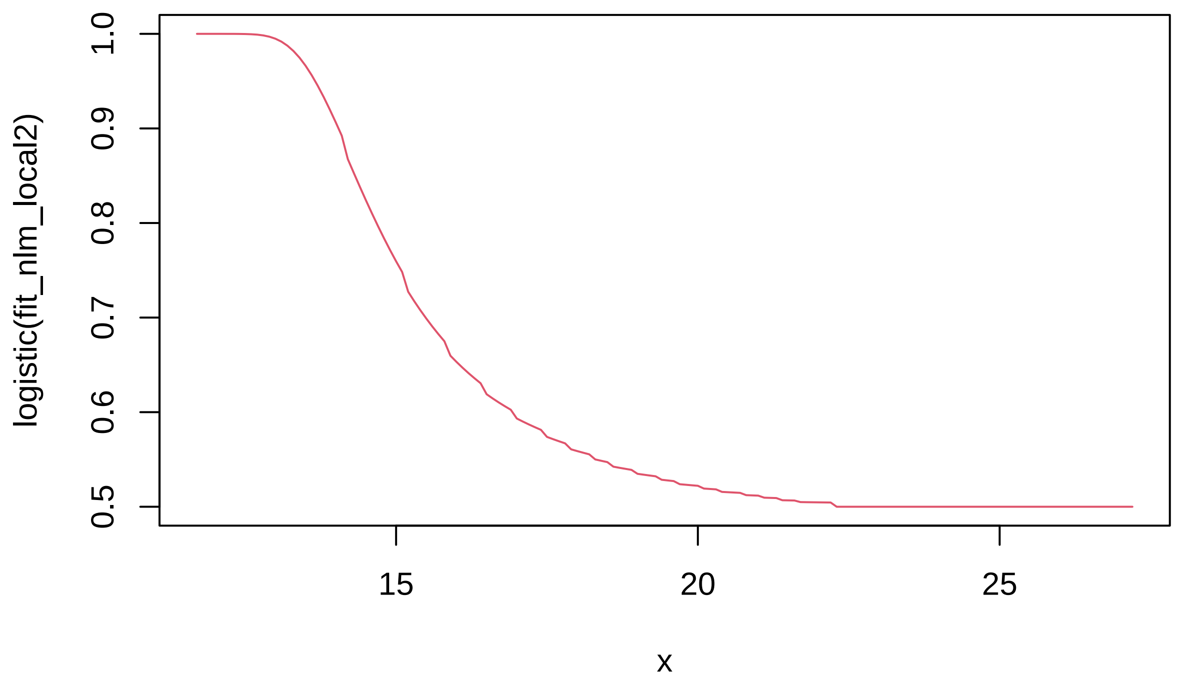


Now, the same but with X = 1.67 and Y = 1.

```
Xev2 <- 11.67
logistic <- function(x) 1 / (1 + exp(-x))
Yev=1
x = seq(min(X), max(X), by = 0.1)
suppressWarnings(
  fit_nlm_local2 <- sapply(x, function(x) {
    K <- dnorm(x = x, mean = Xev2, sd = 2)
    nlm(f = function(beta) {
      -sum(K * (Yev * (beta[1] + beta[2] * (Xev2 - x)) -
                  log(1 + exp(beta[1] + beta[2] * (Xev2 - x)))))
    }, p = c(0, 0))$estimate[1]
  })
)

plot(fit_locfit_lcv, col = 4) + plot(x,logistic(fit_nlm_local2), col=2, type="l")
```



Y
0.0
0.2
0.4
0.6
0.8
15
20
25
X



Thanks to all these plots we can see that, as in Figure 5.1, the local logistic estimated model for the points evaluated and the points in the original plots coincide, which is what we were looking for.

Exercise 4.9. Perform the following tasks:

a. Code your own implementation of the local cubic estimator. The function must take as input the vector of evaluation points x, the sample data, and the bandwidth h. Use the normal kernel. The result must be a vector of the same length as x containing the estimator evaluated at x.

For this exercise, we have used the derivation of a general local linear estimator (for any p). From there, we have implemented the equations needed to obtain the local cubic estimator. These equations result mainly from the multiplication of the matrices X, W and Y and we had to create them.

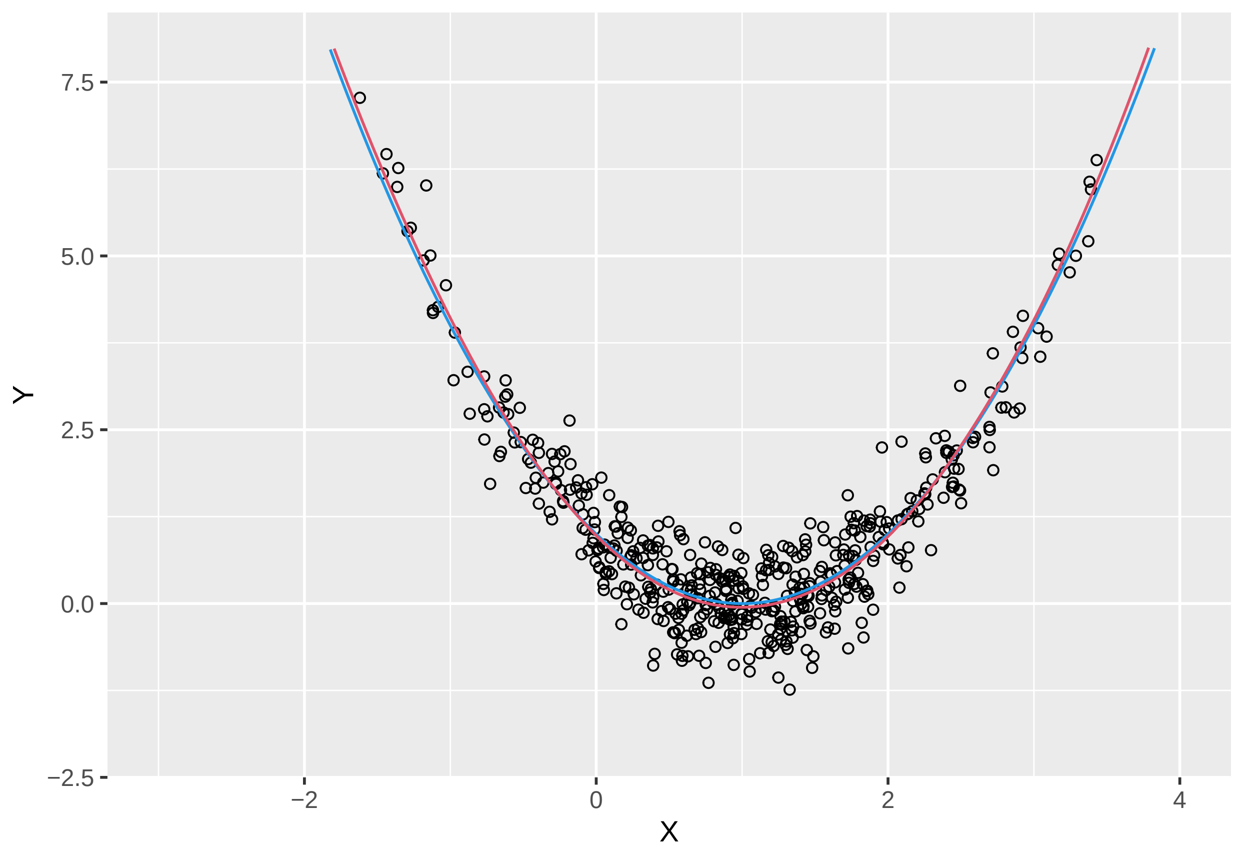
In the creation of the matrix X, we have fixed the value p=3 and computed the matrix. After that, inside the same loop (from 1 to the length of the vector of evaluation points), we created the diagonal matrix W and the final estimation obtained from the multiplication of the matrices (and the multiplication of t(e1) at the beginning of the calculations in order to keep just the first row).

```r
set.seed(1233)

local_cubic=function(DF,x,h){
  X=matrix(nrow=nrow(DF),ncol=4)
  estim_x=c()
  for (i in 1:length(x)) {
    for(p in 0:3){
        X[,(p+1)]=(DF$X-x[i])^p
    }
    W=diag(dnorm(DF$X-x[i]/h)/h)
    estim_x[i]= t(c(1,0,0,0)) %*% (t(X)%*%W%*%X %>% ginv()) %*% (t(X) %*% W %*% DF$Y)
  }

  return(estim_x)
}
```

b. Test the implementation by estimating the regression function in the location model $Y = m(X) + \epsilon$, where $m(x) = (x-1)^2$, $X \sim N(1,1)$, and $\epsilon \sim N(0, 0.5)$. Do it for a sample of size n $= 500$.

We created a data frame with the variables X and Y. After that, we implemented the function and plotted the results to see how good the estimation is.

```
n=500
X=rnorm(n,1,1)
x=seq(-4,4,l=600)

m=function(x){
  return((x-1)^2)
}

err=rnorm(n,0,0.5)

Y=m(X)+err
DF=data.frame(X,Y)


Te=local_cubic(DF,x,h)
m=m(x)

ggplot() + geom_point(data = DF,aes(x=X,y=Y),shape=1) + geom_line(aes(x,m),col="4") +
  geom_line(aes(x,Te),col="2") + xlim(-3,4) + ylim(-2,8)
```



First, we plotted the estimated regression function of the evaluation points. After that, we added the real function m applied to the evaluation points and the estimation of this function. We can clearly see that the estimation of m(x) (m(x,3,h)) is quite accurate, since it is very similar to the real m(x) and follows the same path as the estimated regression function.

Exercise 3.31. Implement the Euler method for f being f_oval, f_croissant, and f_sin, in order to reproduce Figure 3.18.

Let's implement the whole exercise for the function f_oval first and then adapt it to the rest of the functions. Of course, we start by computing this function.

```
# Our functions

# "Oval" density
f_oval <- function(x, mu = 2, sigma = 0.35,
                   Sigma = rbind(c(1, -0.71), c(-0.71, 2))) {

  # x always as a matrix
  x <- rbind(x)

  # Rotate x with distortion
  Sigma_inv_sqrt <- solve(chol(Sigma))
  x <- x %*% Sigma_inv_sqrt

  # Polar coordinates
```

```r
  r <- sqrt(rowSums(x^2))

  # Density as conditional * marginal
  f_theta <- 1 / (2 * pi)
  f_r_theta <- dnorm(x = r, mean = mu, sd = sigma)
  jacobian <-  det(Sigma_inv_sqrt) / r
  f <- f_r_theta * f_theta * jacobian
  return(f)

}
```

Now, we have to implement the projection of the gradient into the Hessian s-th eigenvector subspace. In order to do so, we have made a few changes to the original function.

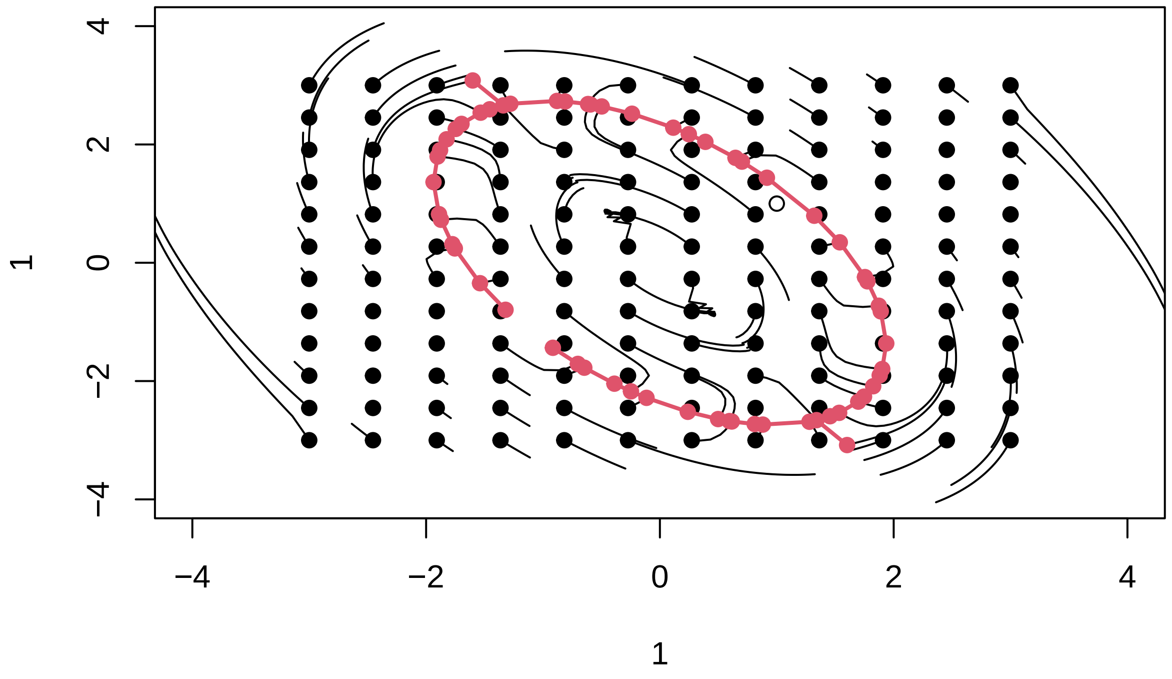
First of all, we have used the functions numDeriv::grad and numDeriv::hessian in order to obtain the gradient and the hessian of the function. Since the objects return from these functions were not the same as the ones from the grad_norm() and hess_norm() functions used before, we had to adjust them in order to make the main function work. For instance, the object grad was not a matrix at the beginning and, therefore, we could not choose the elements grad[i,], so we had to adjust this. A few other variations were made and they can be seen in the function:

```r
# Projected gradient into the Hessian s-th eigenvector subspace
proj_grad_norm_oval <- function(x, mu, Sigma, s = 2) {

  # Gradient
  grad <- numDeriv::grad(f_oval,x) %>% as.matrix() %>% t()

  # Hessian
  Hess <- numDeriv::hessian(f_oval,x)

  # Eigenvectors Hessian
  eig_Hess <- function(A){
    t(as.matrix(eigen(x = A, symmetric = TRUE)$vectors[, s]))
  }
  eig_hess <- eig_Hess(Hess)

  # Projected gradient
  proj_grad <- t(sapply(1:nrow(eig_hess), function(i) {
    tcrossprod(eig_hess[i,]) %*% grad[i,]
  }))

  # As an array
  return(proj_grad)

}
```

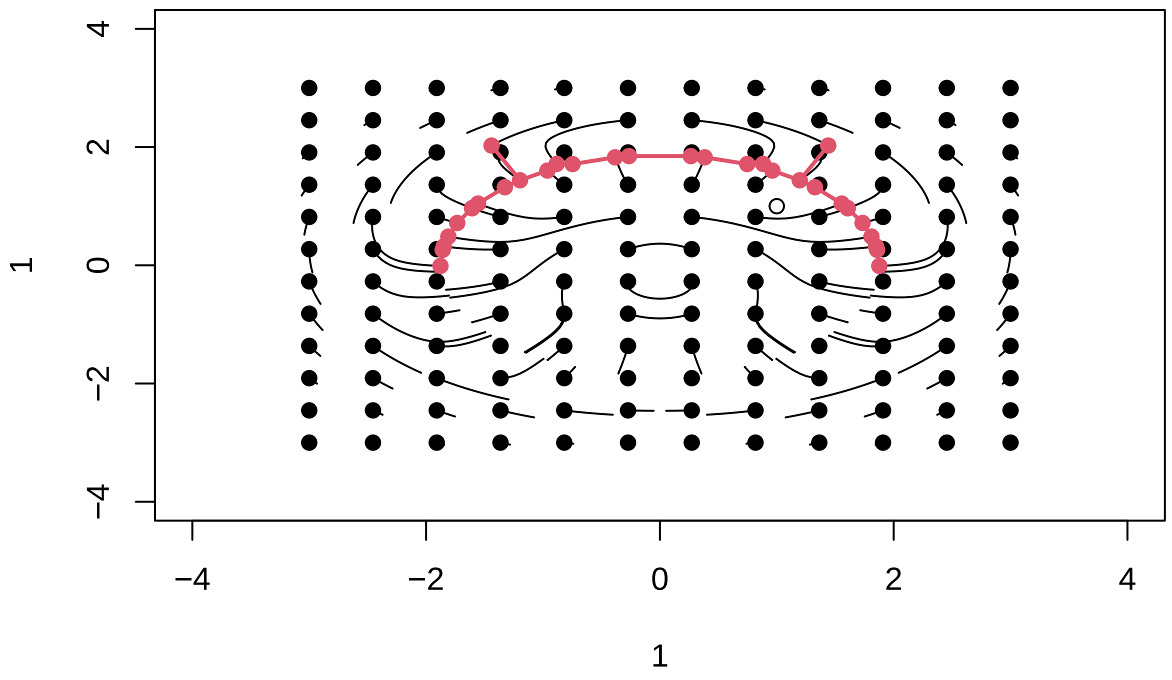
Now, we had to implement the main function for the Euler solution. Here, we had some constraints that stopped the iteration if $\|x_{t+1} - x_t\|_\infty < \epsilon \;\; or \;\; \|x_{t+1} - x_t\|_\infty / \|x_t\|_\infty < \epsilon$, but we added another constraint that disregarded the final points if $f(x) < 50 * \delta$. For this, we implemented this constraint before plotting the final points, allowing to plot the points that only satisfy this constraint.

```r
# Euler solution
plot(x=1,y=1, xlim=c(-4,4), ylim=c(-4,4))
x0 <- as.matrix(expand.grid(seq(-3, 3, l = 12), seq(-3, 3, l = 12)))
```

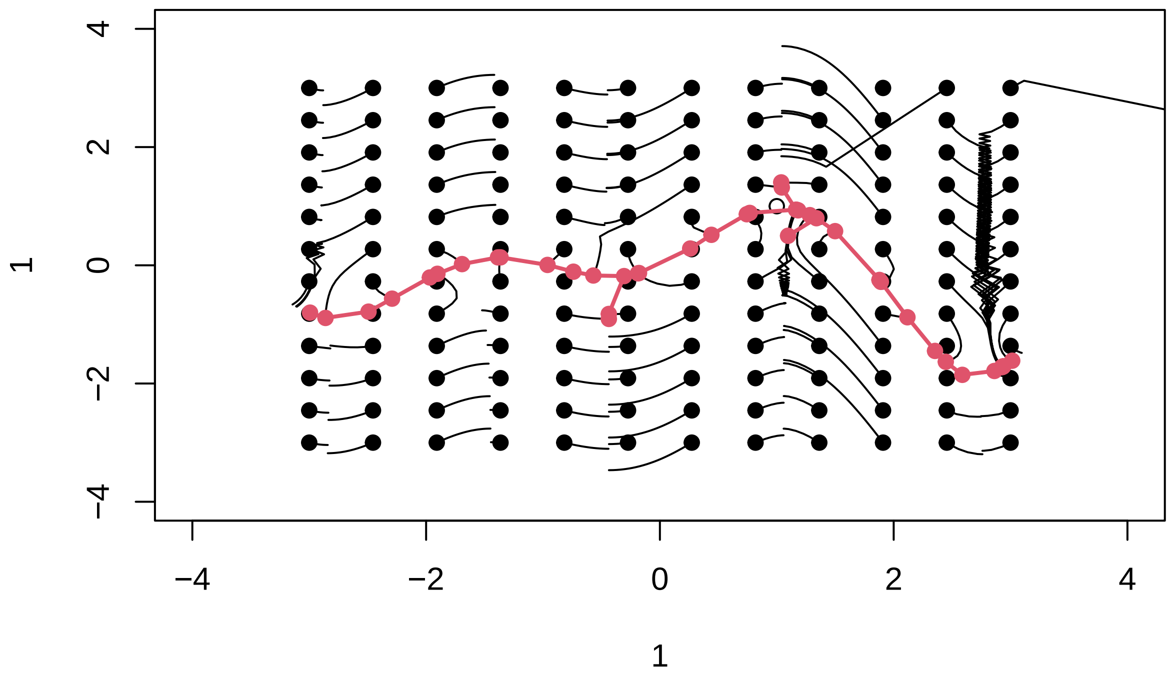
```r
x <- matrix(NA, nrow = nrow(x0), ncol = 2)
N <- 100
h <- 0.06
phi <- matrix(nrow = N + 1, ncol = 2)
eps <- 1e-4
mu <- c(0,0)
Sigma <- rbind(c(1, -0.71), c(-0.71, 2))
delta <- 1.e-3
count=0

for (i in 1:nrow(x0)) {

  # Move along the flow curve
  phi[1, ] <- x0[i, ]
  for (t in 1:N) {

    # Euler update
      phi[t + 1, ] <- phi[t, ] +
        h * proj_grad_norm_oval(phi[t, ], mu = mu, Sigma = Sigma) /
        f_oval(x = phi[t, ])

      # Stopping criterion (to save computing time!)
      abs_tol <- max(abs(phi[t + 1, ] - phi[t, ]))
      rel_tol <- abs_tol / max(abs(phi[t, ]))
      if (abs_tol < eps | rel_tol < eps) break

  }

  # Save final point
  x[i, ] <- phi[t + 1, , drop = FALSE]


  # Plot lines and x0
  lines(phi[1:(t + 1), ], type = "l")
  points(x0[i, , drop = FALSE], pch = 19)
  # if(f_crois(x[i,]) > 50*delta){
  #   points(x=x[i,2],y=x[i,1],pch=19,col=2)
  # }
}

# Plot final points
for (i in 1:nrow(x)) {
  if (f_oval(x)[i] < 50*delta){
    x[i,] <- NA
  }
}
x <- na.omit(x)
points(x, pch = 19, col = 2)
# Join the ridge points with lines in an automatic and sensible way:
# an Euclidean Minimum Spanning Tree (EMST) problem!

emst <- emstreeR::ComputeMST(x = x, verbose = FALSE)
segments(x0 = x[emst$from, 1], y0 = x[emst$from, 2],
        x1 = x[emst$to, 1], y1 = x[emst$to, 2], col = 2, lwd = 2)
```



Now, we do the same thing for the f_crois function.



Now, same thing for f_sin.



1
1
−4
−2
0
2
4
−4
−2
0
2
4