Assignment 3 - The free position facility location problem

Pranav Kasela 846965

Contents

The Problem

In this assignment we will look at the **Facility Location Problem**. The problem can be described in general terms as follows:

We must decide the placement of a facility that distributes a certain good to a group of consumers that need it.

The placement must to be chosen in order to **minimize** the total compound distance from the facility and the customers.

The following assumptions has to be taken into account:

1. The possible location for the facility is unknown, that is the problem is to find the right spot to build it.
2. The facility building costs are fixed and independent from the position of the building site.

Notice that in this scenario there is one possible decision to make:

- where to build the facility, that is find the position (χ, υ) that minimises the compound distance of the facility with respect to all custumers.

Data

A file with the locations of the consumers can be found in the `Data` folder.

Distance function

Given the position of the facility $f = (\chi, \upsilon)$ and of a consumer $p_i = (x_i, y_i)$ use the following formula to calculate the distance between them.

$$d(f, p_i) = log((\chi - x_i)^2 + 1) + log((\upsilon - y_i)^2 + 1)$$

The assignment and the solution

1. Formulate the objective function to minimize for the described problem.

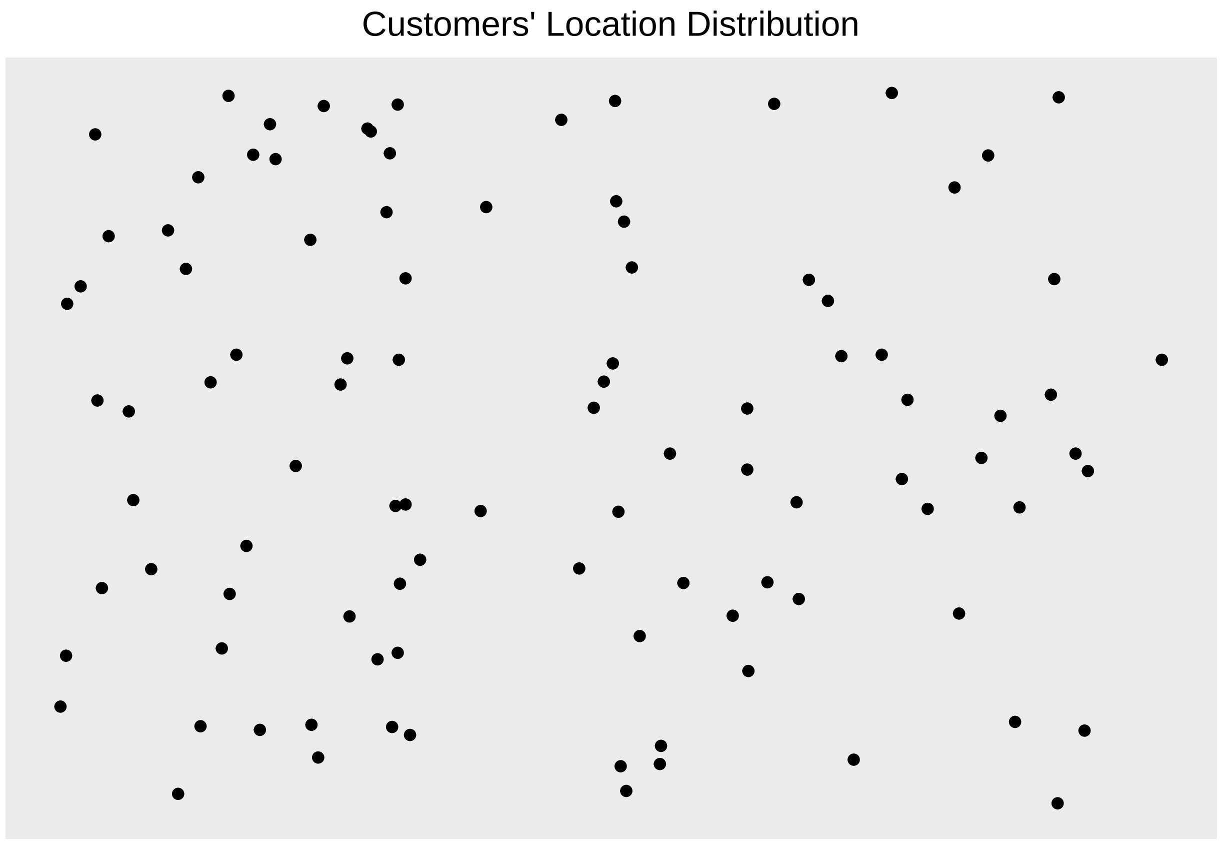
Our Problem, in this case, is to minimize the function:

$$\min \qquad d(\bar{f}) = \sum_{j=1}^{n} \text{distance}(\bar{f}, \bar{p}_j)$$

Where $\bar{f}$ is the location of the facility while $\bar{p}_j$ is the location of the j-th customer, with the distance as defined above.

```r
require(ggplot2)
require(gganimate)
require(curry)
require(optimx)
require(dplyr)
```

```r
customer_locations <- read.csv("consumer_locations.csv")

ggplot(customer_locations, aes(x=x,y=y)) +
    geom_point() +
    ggtitle("Customers' Location Distribution") +
    theme(axis.ticks = element_blank(),
        axis.text  = element_blank(),
        panel.grid = element_blank(),
        axis.title = element_blank(),
        plot.title = element_text(hjust = 0.5))
```



We see that the points are randomly distributed among the plane, this indicates the the minimum of the function (the facility location) will not be near border otherwise the facility will be very far from the location on the opposite border.

2. Express in analytical form the gradient for the objective to minimize.

We start calculating the gradient of the function $d(\bar{f}) = d(\chi, \upsilon)$:

$$\nabla_d = \Big(\sum_i \frac{2(\chi - x_i)}{(\chi - x_i)^2 + 1}, \sum_i \frac{2(\upsilon - y_i)}{(\upsilon - y_i)^2 + 1}\Big)^T$$

This gradient will be used in the numerical methods to search for a local minimum.

We calculate the Hessian matrix as well, we will use it to determine if the stationary point found is a minimum:

$$H = \begin{bmatrix} d_{\chi\chi} & d_{\chi\upsilon} \\ d_{\upsilon\chi} & d_{\upsilon\upsilon} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} \frac{2((\chi-x_i)^2+1)-4(\chi-x_i)^2}{((\chi-x_i)^2+1)^2} & 0 \\ 0 & \sum_{i=1}^{n} \frac{2((\upsilon-y_i)^2+1)-4(\upsilon-y_i)^2}{((\upsilon-y_i)^2+1)^2} \end{bmatrix}$$

Since it has a simple (diagonal) form we also immediatly know that it's inverse is:

$$H^{-1} = \begin{bmatrix} \frac{1}{d_{\chi\chi}} & 0 \\ 0 & \frac{1}{d_{\upsilon\upsilon}} \end{bmatrix}$$

3. Implement the `Gradiend Descent` method and solve the problem with it.

In the Gradient Descendent method, we decide to decay the learning rate (lr) with the number of iterations, based also on a coeffiecent (lr.decay) ≥ 0, the decay is not drastic if the lr.decay is chosen accordingly, but if the lr.decay is wrongly chosen the method might not converge and slow down a lot until it stops long before the zero. The smaller the value of lr.decay the smaller will be the decay, and for lr.decay $= 0$ the is no decay. It is similar to the concept of the stickiness, if stickiness coefficient $= 0$ then the movement is purely based on the gradient of the function, if it is > 0 the ball loses it ability to move after a while and sticks to the surface.

The idea behind it is to slow down the gradient descendent after a while in order to achieve a greater precision in finding the zero and avoiding the situation of the estimated "bouncing" back and forth near the minimum (or maximum) of the function. Usually the learning rate or the step must be calculated at each step, as another minimization or maximization problem (in the steepest gradient descendent method), but it can become really expensive computationally, another method is to use the Newton method calculating the inverse of the Hessian matrix and choosing it as the learning rate.

Another change is that we added a stopping criterion, in our case we decided to stop when the squared norm of the gradient vector $||\nabla||^2$ is smaller than a certain tollerance (which is by default 10^{-6}), this choice was made to make the code a little bit more efficient since it was useless to continue any funther because the iteration step is defined as the previous value - lr*gradient, so if the gradient is too low the movement will be minimal and a gradient near to 0 indicated that a stationary point has been found.

The function returns a list consisting of all the points created with the iteration, a vector containg the change in value of the objective function (the distance), the optimal point and the value of the objective function in the optimal point, a boolean value indicating if the method conveges or not (it is done by checking if the square of the norm of the gradient was smaller than the given tollerance), the determinant of the Hessian matrix in the optimal point and finally number of iterations taken, it return also the value of the final learing rate to check if the lr.decay was choses correctly and lr didn't drop too much.

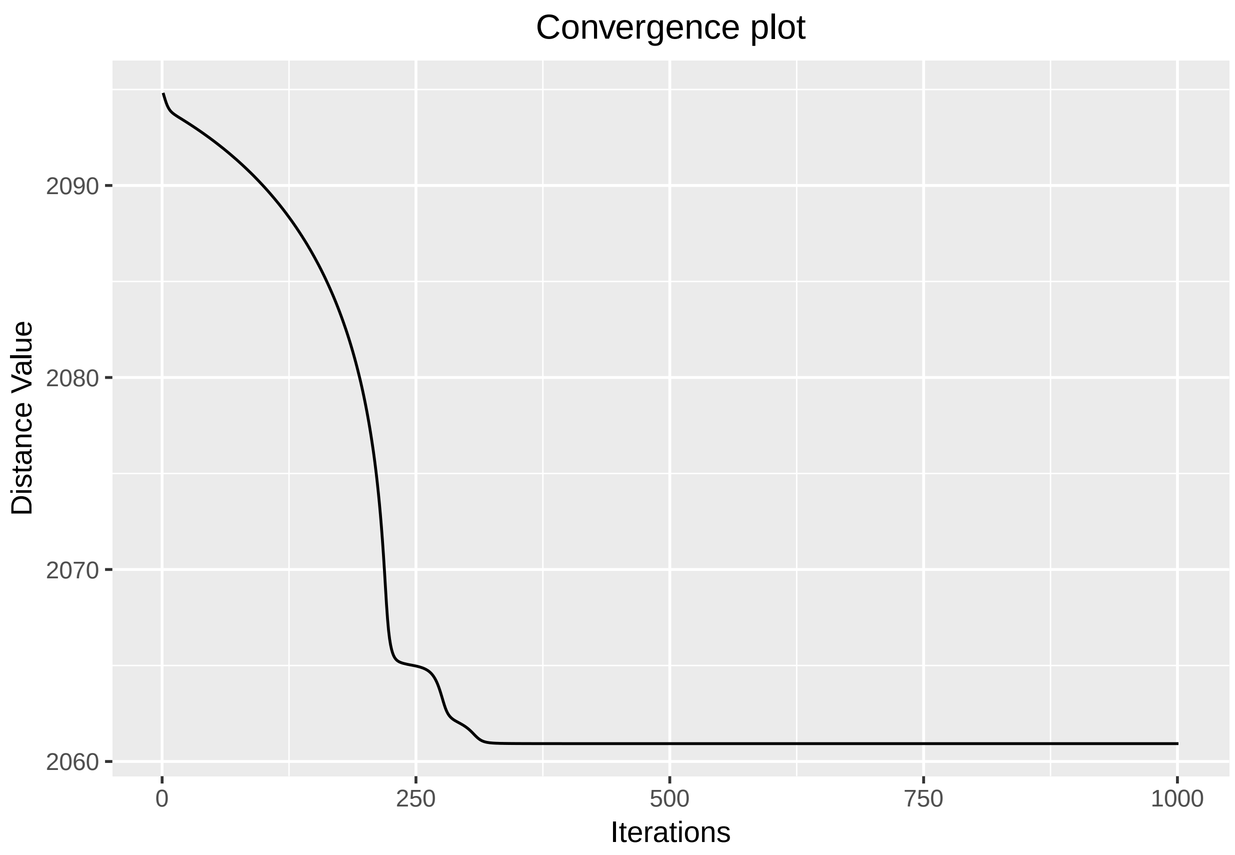
```r
distance <- function(data,f){
  sum(log((f[1]-data[,1])^2+1) + log((f[2]-data[,2])^2+1))
}

#fix the customer_location parameter in the distance function
fn.distance <- curry(distance, customer_locations)
```

```r
gradient <- function(data,f){
  c(sum(2*(f[1]-data[,1])/((f[1]-data[,1])^2 +1)),
    sum(2*(f[2]-data[,2])/((f[2]-data[,2])^2 +1)))
}
#fix the customer_location parameter in the gradient function
fn.gr <- curry(gradient,customer_locations)

Hessian <- function(data,f){
  dxx <- sum(((2*((f[1]-data[,1])^2+1))-4*(f[1]-data[,1])^2)/
    ((f[1]-data[,1])^2+1)^2)
  dxy <- 0
  dyx <- 0
  dyy <- sum(((2*((f[2]-data[,2])^2+1))-4*(f[2]-data[,2])^2)/
    ((f[2]-data[,2])^2+1)^2)
  H <- matrix(c(dxx,dyx,dxy,dyy),nrow=2)
  # Should create a matrix like
  #          |dxx   dxy|
  #          |dyx   dyy|
  return(H)
}
fn.H <- curry(Hessian,customer_locations)

# gradient descent function
gradientDescent <- function(f.init, fn.gr=fn.gr, fn.H=fn.H,
                            lr = 0.1, max_iters=1000,
                            lr.decay = 0, toll=1e-6){
  f       <- matrix(NA, nrow = max_iters+1, ncol = 2)
  grad    <- matrix(NA, nrow = max_iters+1, ncol = 2)
  f[1,]   <- f.init
  lr.init <- lr
  grad[1,]<- fn.gr(f[1,])
  for(k in 1:max_iters){
    f[k+1,]   <- f[k,] - lr*grad[k,]
    #decay the lr as the iteration increases
    #another solution is the Newton solution H.inverse(par[k,])
    lr        <- lr.init*exp(-k*lr.decay)
    grad[k+1,]  <- fn.gr(f[k+1,])
    #stop if norm of gradient^2 is smaller than toll practically 0
    if(sum(grad[k+1,]^2) < toll)
      break
    iter <- k
  }
  f = f[!is.na(rowSums(f)),]
  distance <- apply(f,1,fn.distance)
  if (sum(grad[k+1,]^2) < toll & det(fn.H(tail(f,1)))>0)
    convergence = TRUE
  else
    convergence = FALSE
  return(list(f           = f,
              dist        = distance,
              opt_point   = tail(f,1),
              min_dist    = tail(distance,1),
              det_H       = det(fn.H(tail(f,1))),
              convergence = convergence,
              lr          = lr,# only for testing and development
              iters       = iter))
}

# A function to print the solution
print.solution <- function(result){
  P      <- result$opt_point
  P_dist <- result$min_dist
  f      <- result$f
  f_dist <- result$dist
  if (result$convergence)
    converge <- "The method converges to the minimum"
  else
    converge <- "The method doesn't converge to the minimum"

   cat(paste0(" ",converge," in ", result$iters,
       " iterations."),'\n',
       paste0("The minimum is the point of coordinates (",
       round(P[1],2), ",",round(P[2],2),
       ")"),'\n',paste0("The value of the function is ",
       round(P_dist,2)),"\n",paste0("The determinant of",
                                   " the Hessian is ",
                                   result$det_H))

  ggplot(data=data.frame(distance=f_dist)) +
    aes(x=distance) +
    geom_line(aes(x = 1:length(distance),y=distance)) +
    labs(title = "Convergence plot",
         x = "Iterations", y="Distance Value") +
    theme(plot.title = element_text(hjust = 0.5))
}
```

Here is a first attempt to find a solution using the gradiente descendent method using some random points.

```r
res <- gradientDescent(c(runif(1,0,1000),runif(1,0,1000)),
                       fn.gr=fn.gr, fn.H=fn.H, lr = 0.1,
                       max_iters = 1000, lr.decay = 0.001)


print.solution(res)
```

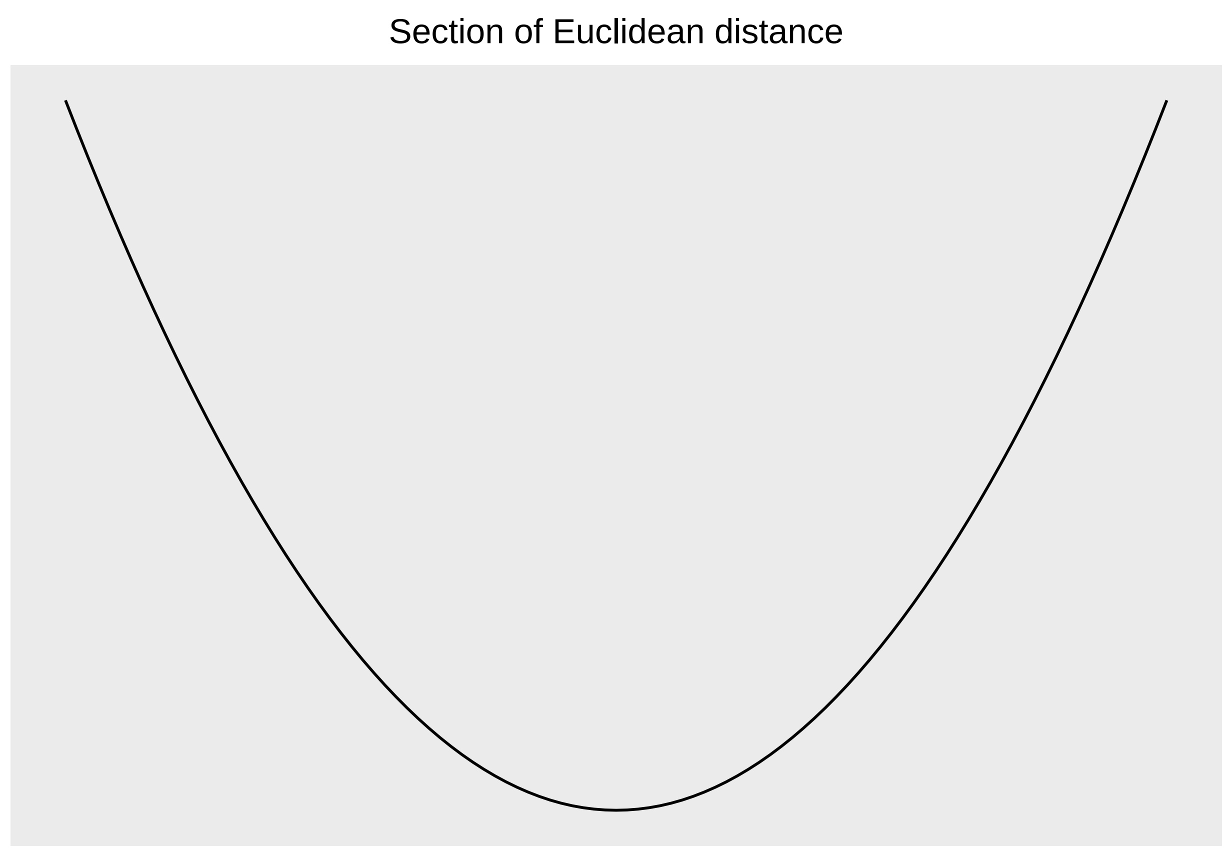
```
##  The method doesn't converge to the minimum in 1000 iterations.
##  The minimum is the point of coordinates (175.2,429.66)
##  The value of the function is 2060.93
##  The determinant of the Hessian is -0.271341426505545
```



We notice that the solution is heavily influenced by the initial point (we noticed it on different run of the latter code), to verify if we have a local minima problem we try to do a 3D plot and a contour plot of the distance function, we can expect it to have a lot of minimum function since the function distance from single point has only 1 minimum which is the point itself (it has distance 0 from itself and distance for defintion is positive), but as we sum all these distances from all points and it might create a lot of minimum between the point.

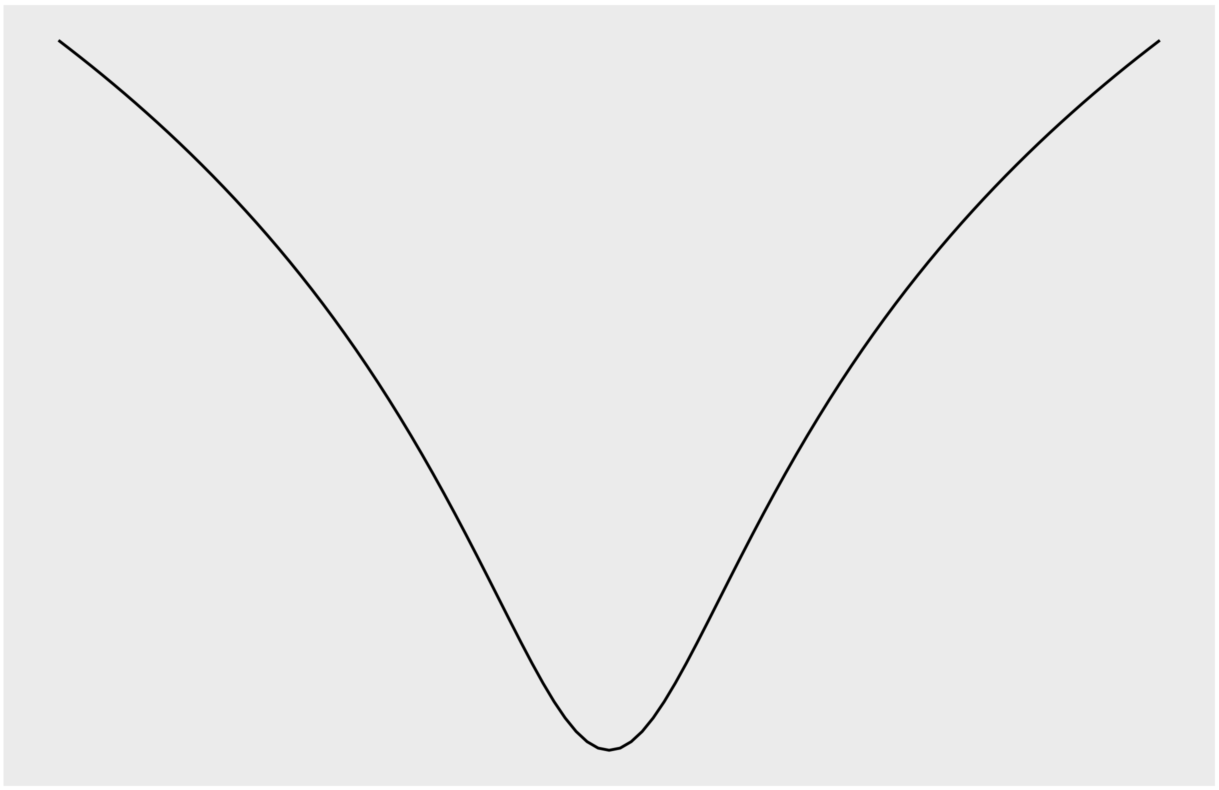
It is more intuitive to think a point as a heavy mass which deforms the plane and the deformation is based on the distance itself, in the case of a euclidean distance the deformation is a paraboloid, in this case we have a logarithmic tranformation of the euclidean distance traslated by +1, thus changing the concavity of the distance function. Now as we put more points on the plane we have that the deformation of this plane creates a lot of local minimum and also a lot of maximum.

```r
quadratic <- function(x) x^2

ggplot(data = data.frame(x=0), mapping = aes(x=x)) +
  stat_function(fun = quadratic) + xlim(-5,5) +
  labs(title = "Section of Euclidean distance") +
  theme(axis.ticks = element_blank(),
        axis.text  = element_blank(),
        panel.grid = element_blank(),
        axis.title = element_blank(),
        plot.title = element_text(hjust = 0.5))
```



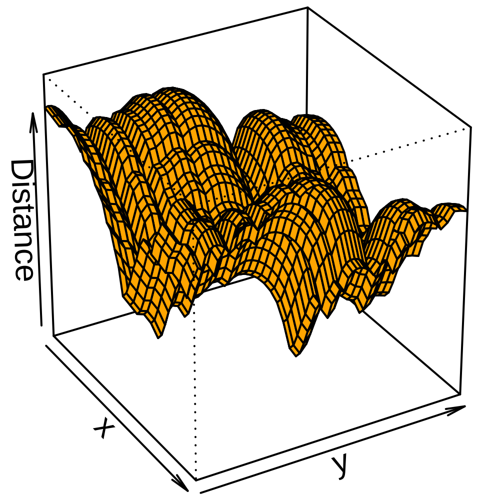
```
logarithmic <- function(x) log(x^2+1)
ggplot(data = data.frame(x=0), mapping = aes(x=x)) +
  stat_function(fun = logarithmic) + xlim(-5,5) +
  labs(title = "Section of log transformation of Euclidean dist.") +
  theme(axis.ticks = element_blank(),
        axis.text  = element_blank(),
        panel.grid = element_blank(),
        axis.title = element_blank(),
        plot.title = element_text(hjust = 0.5))
```

Section of log transformation of Euclidean dist.

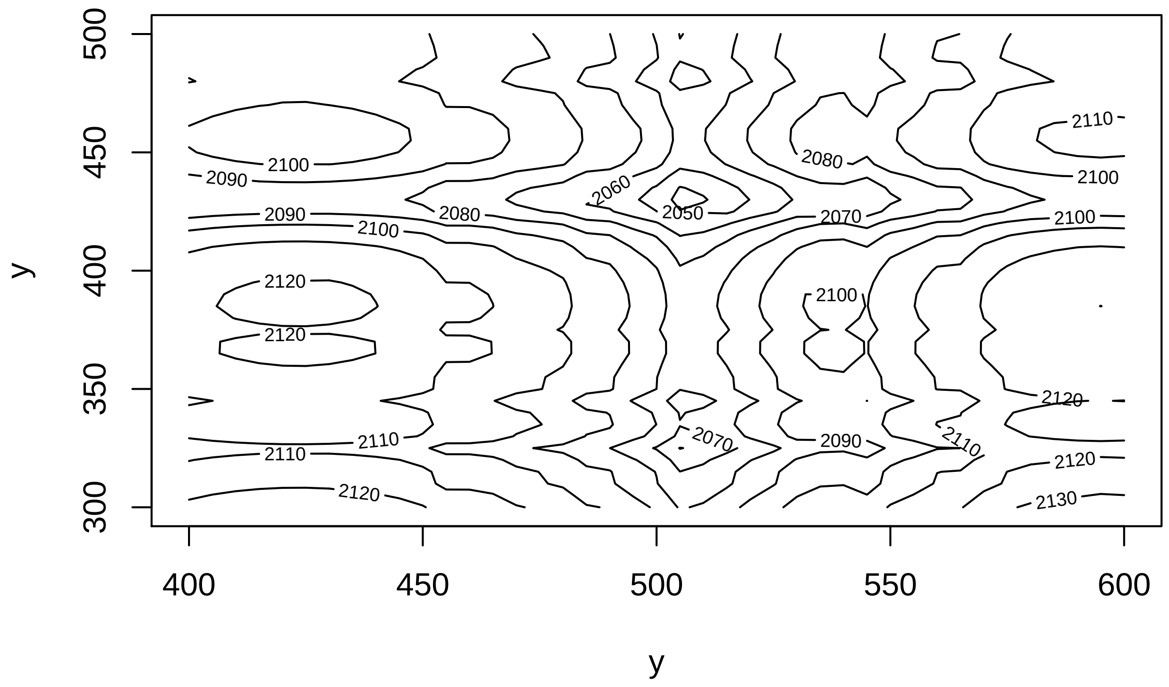


Let's try to see it with a 3D plot helped by a contour plot, in order to avoid complex and black plot each point is distant 5 units from each other and we plot only between: $400 < x < 600$ and $300 < y < 500$ to show the problem of a lot a local minima and maximum.

```r
helper.dist <- function(x,y){ #m and c are vectors
dist = matrix(nrow = length(x),ncol = length(y))
for (i in seq(1, length(x), by = 1)) {
 for (j in seq(1, length(y), by = 1)) {
   dist[i,j] = fn.distance(c(x[i], y[j]))
 }
}
dist
}

x=seq(400, 600, 5)
y=seq(300, 500, 5)
z <- helper.dist(x,y)

persp(x, y, z, phi = 30, theta = 60,col = "orange",xlab = "x",
      ylab = "y", zlab = "Distance", r=10, d=5)
```



```
contour(x,y,z, xlab="y",ylab="y")
```

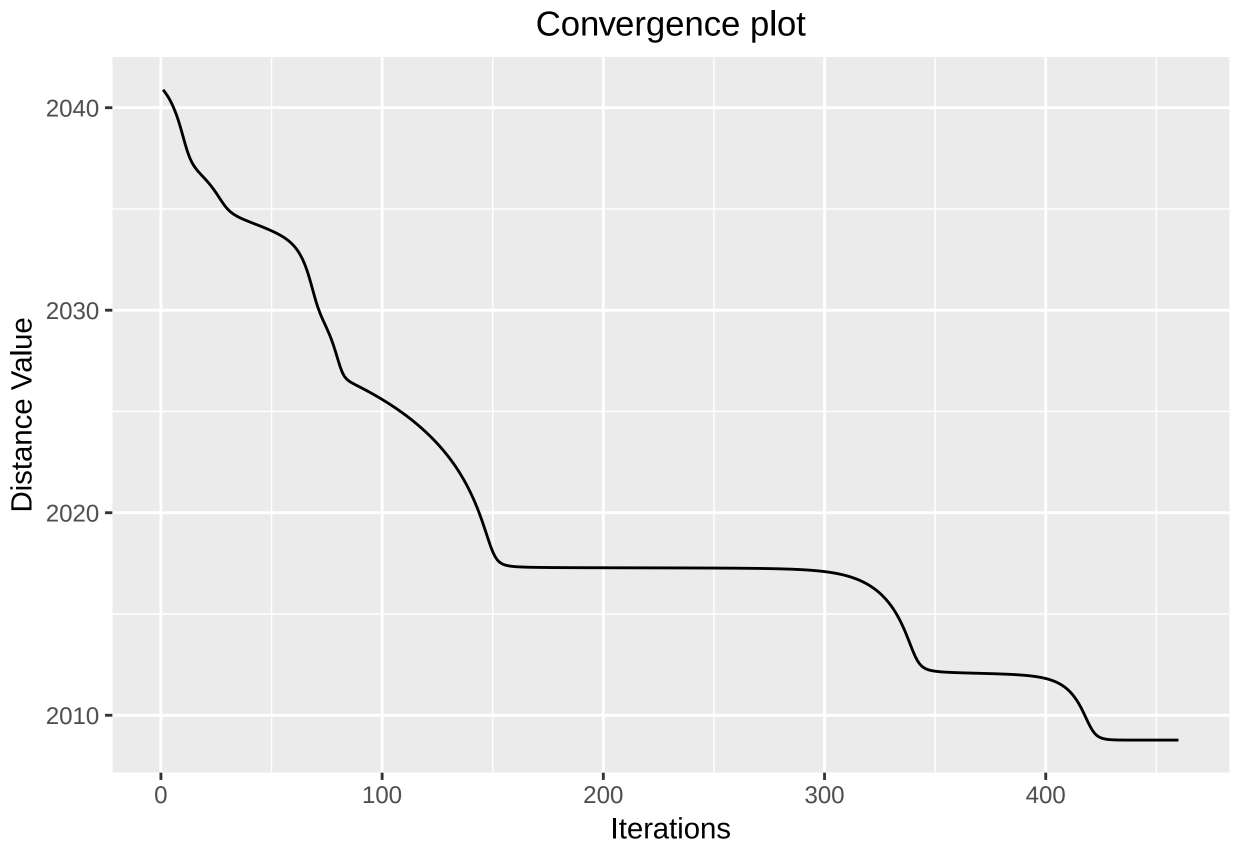


A great method to solve this problem, in this case, is to initiate a lot of random points from which we start the solution (a multistart method), we start by using a uniform distribuition of the point.

```
set.seed(123456789)
x <- runif(1000,0,1000)
y <- runif(1000,0,1000)
random.points <- cbind(x,y)

ris_temp <- lapply(1:length(y), function(x) {
  gradientDescent(random.points[x,],fn.gr,fn.H,lr = 0.1,
                  max_iters = 1000,lr.decay = 0.0001)})

opt_points <- matrix(NA,ncol=2, nrow=length(x))
distances  <- rep(NA,length(x))
```

```
for (i in 1:nrow(random.points)){
  opt_points[i,] <- ris_temp[[i]]$opt_point
  distances[i]   <- ris_temp[[i]]$min_dist
}

print.solution(ris_temp[[which.min(distances)]])
```

```
##  The method converges to the minimum in 458 iterations.
##  The minimum is the point of coordinates (310.16,565.48)
##  The value of the function is 2008.77
##  The determinant of the Hessian is 5.6186117284341
```

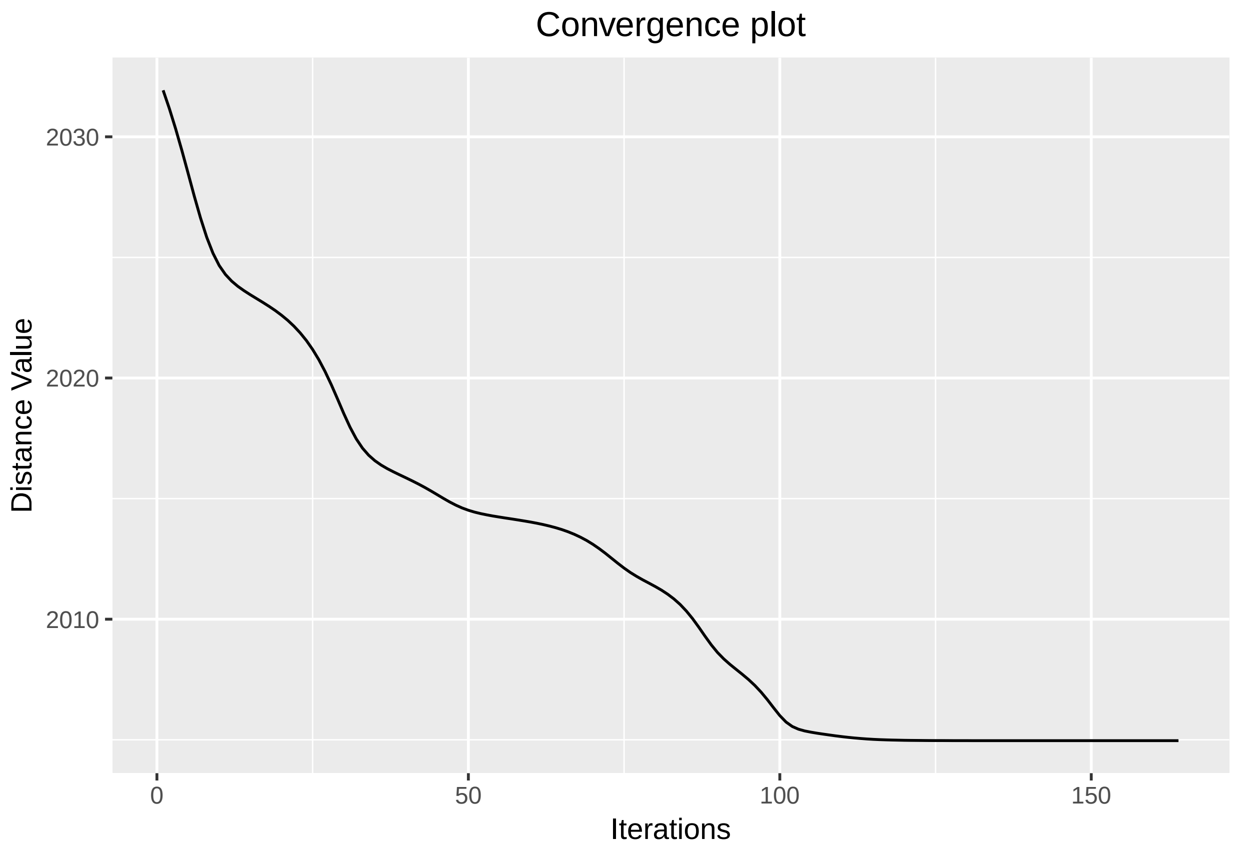


Another choice could have been to generate the random point using a normal distribution with the standard deviations and means of the x,y components of the customer locations. This way the points are concentrated in the middle where we suspect the global minimum to be (more density → less distance between points).

```
set.seed(123456789)
sd.x   <- sd(customer_locations$x)
mean.x <- mean(customer_locations$x)
sd.y   <- sd(customer_locations$y)
mean.y <- mean(customer_locations$y)
x <- rnorm(500,mean.x,sd.x)
y <- rnorm(500,mean.y,sd.y)
random.points <- cbind(x,y)

ris_temp <- lapply(1:length(y), function(x) {
  gradientDescent(random.points[x,],fn.gr,fn.H,lr = 0.1,
                max_iters = 1000,lr.decay = 0.0001)})
```

```r
opt_points <- matrix(NA,ncol=2, nrow=length(x))
distances  <- rep(NA,length(x))

for (i in 1:nrow(random.points)){
  opt_points[i,] <- ris_temp[[i]]$opt_point
  distances[i]   <- ris_temp[[i]]$min_dist
}

print.solution(ris_temp[[which.min(distances)]])
```

```
##  The method converges to the minimum in 162 iterations.
##  The minimum is the point of coordinates (310.16,430.87)
##  The value of the function is 2004.96
##  The determinant of the Hessian is 3.62126677851931
```



```r
# P     <- ris_temp[[which.min(distances)]]$f
# final <- ris_temp[[which.min(distances)]]$opt_point
# z     <- seq(1,nrow(P),2)
#
# ggif_minimal <- data.frame(x=P[z,1],y=P[z,2],z=z)%>%
#   ggplot(aes(x = x, y = y))+
#   geom_point(col="magenta",alpha=0.5) +
#   labs(title = "Iteration of the point towards the minimum")+
#   geom_point(data=data.frame(x=final[1],y=final[2]),col="blue")+
#   geom_text(data=data.frame(z=z),
#             mapping = aes(x = 300, y = 435,
#                           label = paste0("ITERATION N.  ",z)))+
#   theme(plot.title = element_text(hjust = 0.5))+
#   transition_reveal(z)+
#   ease_aes("linear")+
#   enter_appear()+
#   exit_fade()
#
# animate(ggif_minimal, fps=90)
```

The best solution found is (310.16,430.87) with a value of 2004.96.

4. Solve the problem with a package provided by R (for instance, using the function `optimr` within the package `optimx`). Note that it is not required to use the `gradient descent` algorithm to solve the problem, other algorithms can be used as well.

In this case there is nothing to comment, we just use the function provided by optimr on the random points generated randomly using a normal distribution. We use as the method `Rcgmin` which is an updated version of the `Conjugate Gradient(CG)`

```
ris_temp <- lapply(1:length(y), function(x) {
  optimr(random.points[x,],fn.distance, fn.gr,method = "Rcgmin")})

opt_points <- matrix(NA,ncol=2, nrow=length(x))
distances  <- rep(NA,length(x))

for (i in 1:nrow(random.points)){
  opt_points[i,] <- ris_temp[[i]]$par
  distances[i]   <- ris_temp[[i]]$value
}

proptimr(ris_temp[[which.min(distances)]])
```

```
## Result  ris_temp[[which.min(distances)]]   proposes optimum function value = 2004.964  at parameters
##        x        y
## 310.1627 430.8702
## After  37  fn evals, and  22  gr evals
## Termination code is  0 : NA
## ---------------------------------------
```

5. Implement the `Stochastic gradient descent` algorithm with mini-batches and use it to solve the problem.

In the stochastic method, since our data has the same scale there is no need to normalize. For the stopping creterion we choose to stop if the $||\nabla||^2$ <toll. The subset of the points is chosen as the 20% of the initial data randomly sampled with replacement. Also in this case we adapt the decaying learning rate.

```
stoch_gradDescent <- function(xy,f.init, fun.gr, lr,
                              max_iters=1000, lr.decay=0){
  f       <- f.init
  # Initialize a matrix to store values of the point
  # and distance for each iteration
  P       <- matrix(NA, nrow = max_iters + 1, ncol = 2)
  dist    <- rep(NA,times=max_iters)
  P[1,]   <- f
```

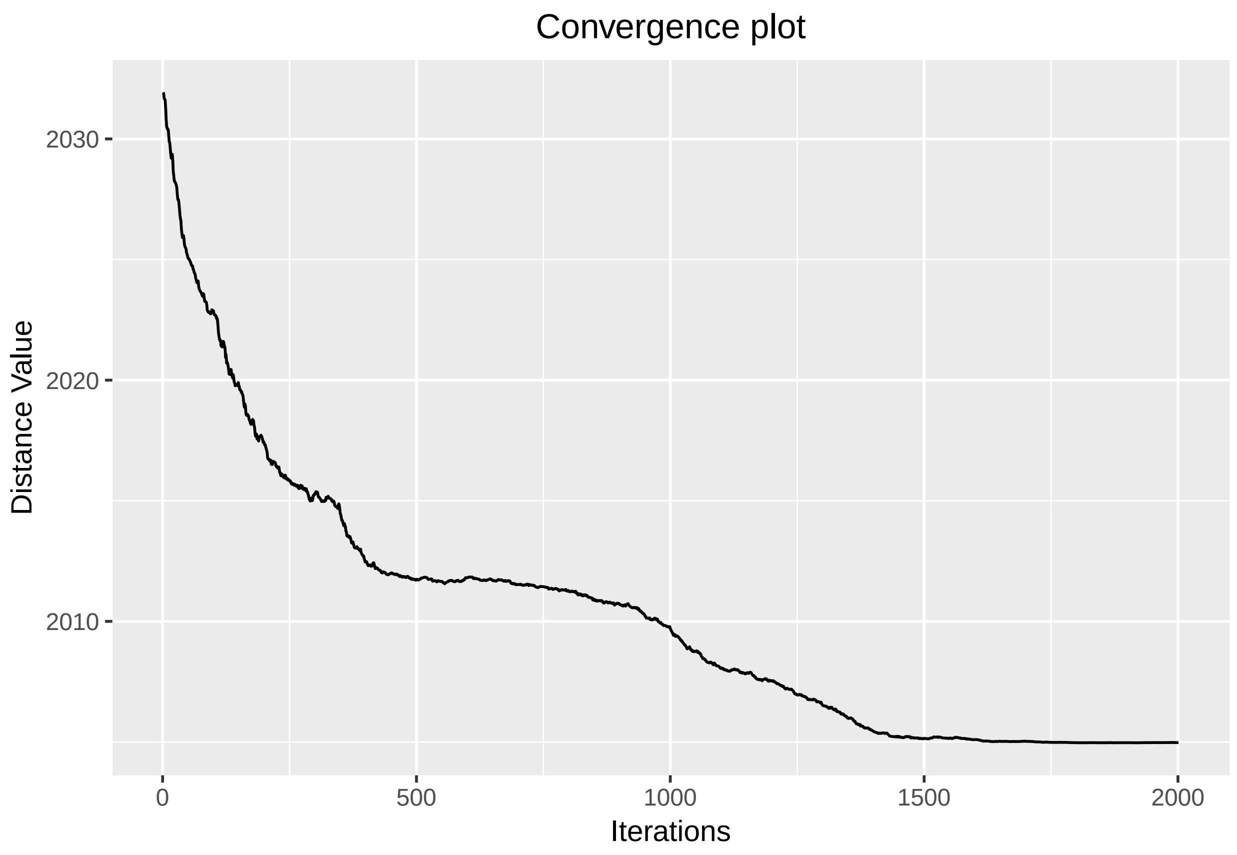
```r
  dist[1] <- fn.distance(P[1,])
  # set seed value for random sampling
  set.seed(42)
  lr.init <- lr
  # now iterate using mini batches of randomly sampled  data,
  for (i in 1:max_iters) {
    # randomly sample 20% of data from the xy data frame
    xysamp <- xy[sample(nrow(xy), floor(nrow(xy)*0.2),
                                  replace = TRUE), ]
    # update point using mini batches
    f   <- f - lr  * fun.gr(xysamp, f)
    lr  <- lr.init * exp(-i*lr.decay)
    # save the beta values for iteration i to a matrix for plotting
    P[i+1,]  <- f
    iter      <- i
    dist[i+1] <- fn.distance(P[i+1,])
    if(sum((fun.gr(xy,f))^2)<1e-6)
      break
 } # end for loop
 P     <- P[!rowSums(is.na(P)),]
 dist  <- dist[!is.na(dist)]
 if (sum((P[i+1,]-P[i,])^2)<1e-6)
    convergence <- TRUE
  else
    convergence <- FALSE
 return(list( f           = P,
              dist        = dist,
              opt_point   = P[which.min(dist),],
              min_dist    = dist[which.min(dist)],
              convergence = convergence,
              lr          = lr,# only for testing and development
              iters       = iter))
}
```

The best choise is always to start randomly, so we choose the random values already created, for the Point 3. Here we see that the method does not converge exactly at the desired point, one of the reasons is how the stochastic method is defined, but also the shape of the function is so that the gradient rapidly becomes zero very near the minimum (as we can see in the plot in point 3 of the section of the log transformation), so even if the solution is near to the minimum it can have a gradient not even close to zero. Infact we can see that the difference between the given solution from the stochastic method and the gradient method is small (the difference is of an order of 10^{-2}) but the gradient is not small enough to be considered zero.

```r
# We take randoms from the point 3.
ris_temp <- lapply(1:nrow(random.points), function(x) {
  stoch_gradDescent(customer_locations,random.points[x,],
                    gradient,lr=0.1,max_iters=2000,
                    lr.decay=0.0015)})

opt_points <- matrix(NA,ncol=2, nrow=length(x))
distances  <- rep(NA,length(x))

for (i in 1:nrow(random.points)){
  opt_points[i,] <- ris_temp[[i]]$opt_point
  distances[i]   <- ris_temp[[i]]$min_dist
}

f      <- ris_temp[[which.min(distances)]]$opt_point
f_dist <- ris_temp[[which.min(distances)]]$min_dist
P_dist <- ris_temp[[which.min(distances)]]$dist

cat(paste0(" The minimum is the point of coordinates (",
   round(f[1],2), ",",round(f[2],2),
   ")"),'\n',paste0("and the value of the function is ",
   round(f_dist,2)),'\n',"The gradient value in the",
    " optimal point is (",fn.gr(f),")")
```

```
##  The minimum is the point of coordinates (310.13,430.93)
##  and the value of the function is 2004.97
##  The gradient value in the  optimal point is ( -0.09365178 0.07093146 )
```

```
ggplot(data=data.frame(distance=P_dist)) +
  aes(x=distance) +
  geom_line(aes(x = 1:length(distance),y=distance)) +
  labs(title = "Convergence plot",
       x = "Iterations", y="Distance Value") +
  theme(plot.title = element_text(hjust = 0.5))
```



In the following plot we can see the chaotic movement of the iteration reaching finally the solution where it start to stablize, here the decaying learing rate is helping the method to stablize the solution after a certain number of iterations.

```
# P     <- ris_temp[[which.min(distances)]]$f
# final <- ris_temp[[which.min(distances)]]$opt_point
# z     <- seq(1,nrow(P),2)
#
# ggif_minimal <- data.frame(x=P[z,1],y=P[z,2],z=z)%>%
#   ggplot(aes(x = x, y = y))+
#   geom_point(col="magenta",alpha=0.5) +
#   labs(title = "Iteration of the point towards the minimum")+
#   geom_point(data=data.frame(x=final[1],y=final[2]),col="blue")+
#   geom_text(data=data.frame(z=z),
#             mapping = aes(x = 306, y = 430,
#                           label = paste0("ITERATION N.  ",z)))+
#   theme(plot.title = element_text(hjust = 0.5))+
#   transition_reveal(z)+
#   ease_aes("linear")+
#   enter_appear()+
#   exit_fade()
#
# animate(ggif_minimal, fps=90)
```

Extra: Implementation of the `Newton Method` and `Conjugate Descendent Method`

Since the inverse of the Hessian Matrix is so easy to calculate, we can try to implement the newton method (usually it is way more faster than the gradient descendent with fixed learning rate), but we must note that the Newton method searches only for the solution of $\nabla = 0$, and is not obligated to move towards the minimum, unlike the the gradient descendent method which can be used either to move upward or downward in the function.

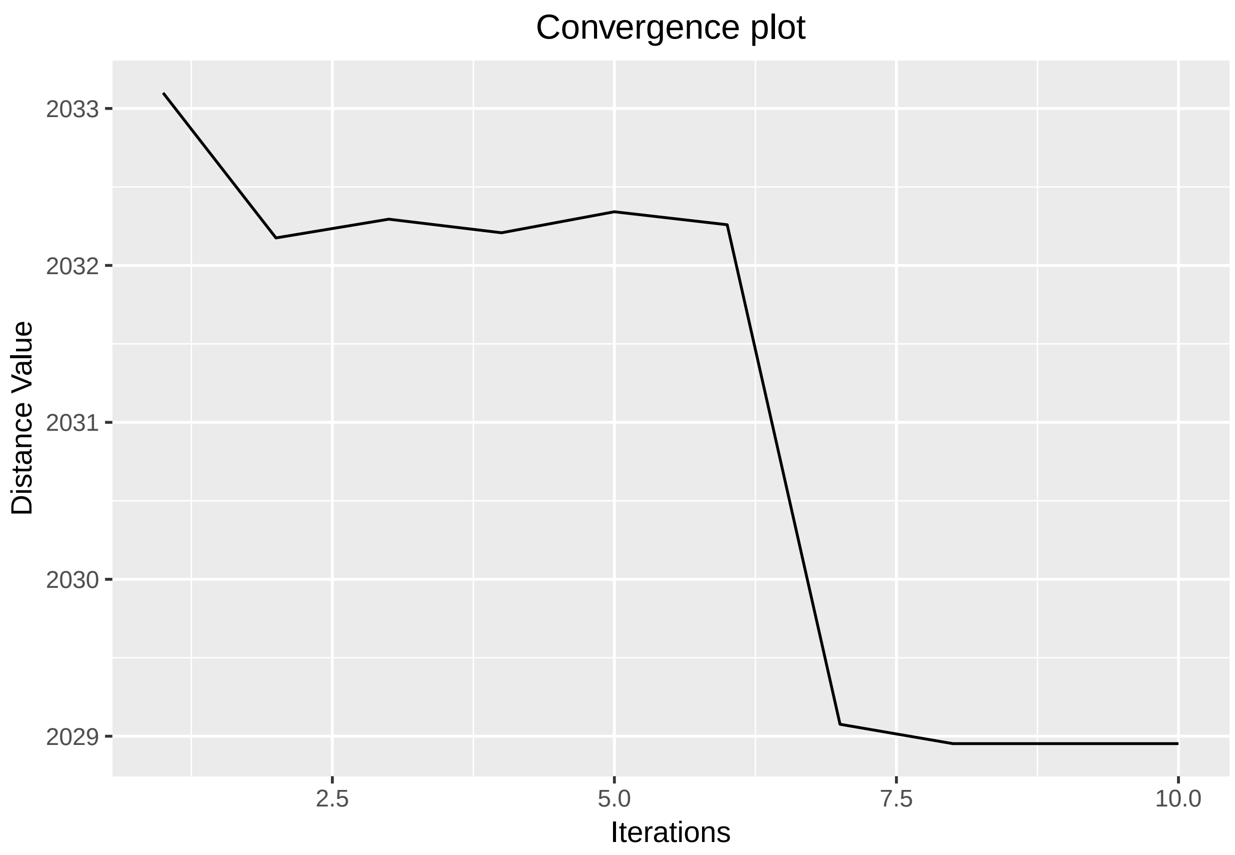
So we might see that the optimal results starts rising instead of going down, and stablizes at a point, whose objective value is higher even than the starting point. This happens because the negative gradient vector points towards a lower value of the function but if the function is concave it's Hessian is definitive negative, so is it's inverse in our case (since it's diagonal), thus the function starts moving in the opposite direction of the gradient, and if we start far enough from the zero of the gradient the method might even not converge.

```
inv.fn.H <- function(f){
  H <- fn.H(f)
  inv.dxx <- 1/H[1,1]
  inv.dyy <- 1/H[2,2]
  inv.H <- matrix(c(inv.dxx,0,0,inv.dyy),nrow=2)
}
Newton <- function(f.init, fn.gr=fn.gr, inv.fn.H=inv.fn.H,
                   max_iters=1000,toll=1e-6){
  f       <- matrix(NA, nrow = max_iters+1, ncol = 2)
  f[1,]   <- f.init
  for(k in 1:max_iters){
    f[k+1,]   <- f[k,] - inv.fn.H(f[k,])%*%fn.gr(f[k,])
    #another solution is the Newton solution H.inverse(par[k,])
    #stop if norm of gradient^2 is smaller than toll practically 0
    if(sum(fn.gr(f[k,])^2) < toll)
      break
    iter <- k
  }
  f = f[!is.na(rowSums(f)),]
```

```r
  distance <- apply(f,1,fn.distance)
  if (sum(fn.gr(f[k+1,])^2) < toll)
    convergence = TRUE
  else
    convergence = FALSE
  return(list(f           = f,
              dist        = distance,
              opt_point   = tail(f,1),
              min_dist    = tail(distance,1),
              det_H       = det(fn.H(tail(f,1))),
              convergence = convergence,
              iters       = iter))
}

ris_temp <- lapply(1:length(y), function(x) {
  Newton(random.points[x,],fn.gr,inv.fn.H,
                max_iters = 1000)})

opt_points <- matrix(NA,ncol=2, nrow=length(x))
distances  <- rep(NA,length(x))

for (i in 1:nrow(random.points)){
  opt_points[i,] <- ris_temp[[i]]$opt_point
  distances[i]   <- ris_temp[[i]]$min_dist
}

f      <- ris_temp[[which.min(distances)]]$opt_point
f_dist <- ris_temp[[which.min(distances)]]$min_dist
P_dist <- ris_temp[[which.min(distances)]]$dist

cat(paste0("The stationary is the point of coordinates (",
   round(f[1],2), ",",round(f[2],2),
   ")"),'\n',paste0("and the value of the function is ",
   round(f_dist,2)))
```

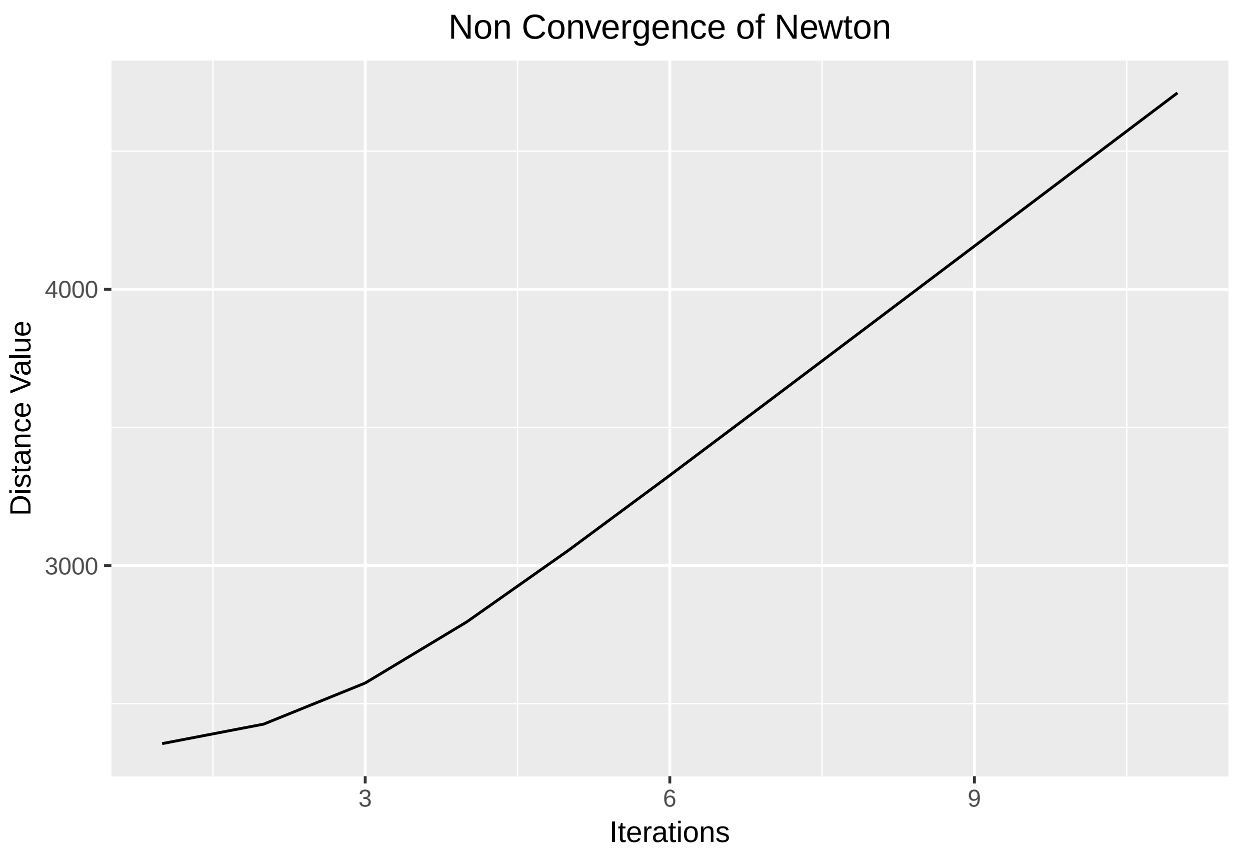
```
## The stationary is the point of coordinates (292.3,565.48)
##  and the value of the function is 2028.95
```

```r
ggplot(data=data.frame(distance=P_dist)) +
  aes(x=distance) +
  geom_line(aes(x = 1:length(distance),y=distance)) +
  labs(title = "Convergence plot",
       x = "Iterations", y="Distance Value") +
  theme(plot.title = element_text(hjust = 0.5))
```



Here is an example on how the Newton method can fail, and rapidly diverge to points far far away.

```
no_converge <- Newton(c(0,0),fn.gr,inv.fn.H,
                       max_iters = 10)

ggplot(data=data.frame(distance=no_converge$dist)) +
  aes(x=distance) +
  geom_line(aes(x = 1:length(distance),y=distance)) +
  labs(title = "Non Convergence of Newton",
       x = "Iterations", y="Distance Value") +
  theme(plot.title = element_text(hjust = 0.5))
```



```r
cat("The final point is (",
    no_converge$opt_point[1],",",no_converge$opt_point[2],")")
```

```
## The final point is ( -92200.66 , -182516.1 )
```

Here we try to implement the Conjugate Gradient Descendent method, in the first step we basically calculate the second point as $x_1 = x_0 - \nabla_d(x_0)$, from here on we calculate for each step $\Delta_n = \nabla_d(x_n) = (\delta_\chi, \delta_v)^T$ and define the step "lr" as $\beta = \max\left(\frac{t(\Delta_{n+1})\Delta_{n+1}}{t(\Delta_n)\Delta_n}, 0\right)$ where $t(\Delta_n)$ is the transpose of the vector Δ_n and the next point is calculated as $x_{n+1} = x_n - \beta\Delta_n$, if $\beta < 0$, so we will move in the opposite direction of the one we desire, the β will be reseted to 0, so the process will start again since we will have $\Delta_n = \Delta_{n+1}$ thus the next $\beta = 1$, which was how we started the method. This estimation of β was proposed by Fletcher and Reeves, there are other ways of calculating the moving rate, for example another one is: $\beta = \frac{t(\Delta_{n+1})(\Delta_{n+1}-\Delta_n)}{t(\Delta_n)\Delta_n}$ proposed by Polak and Ribiere.

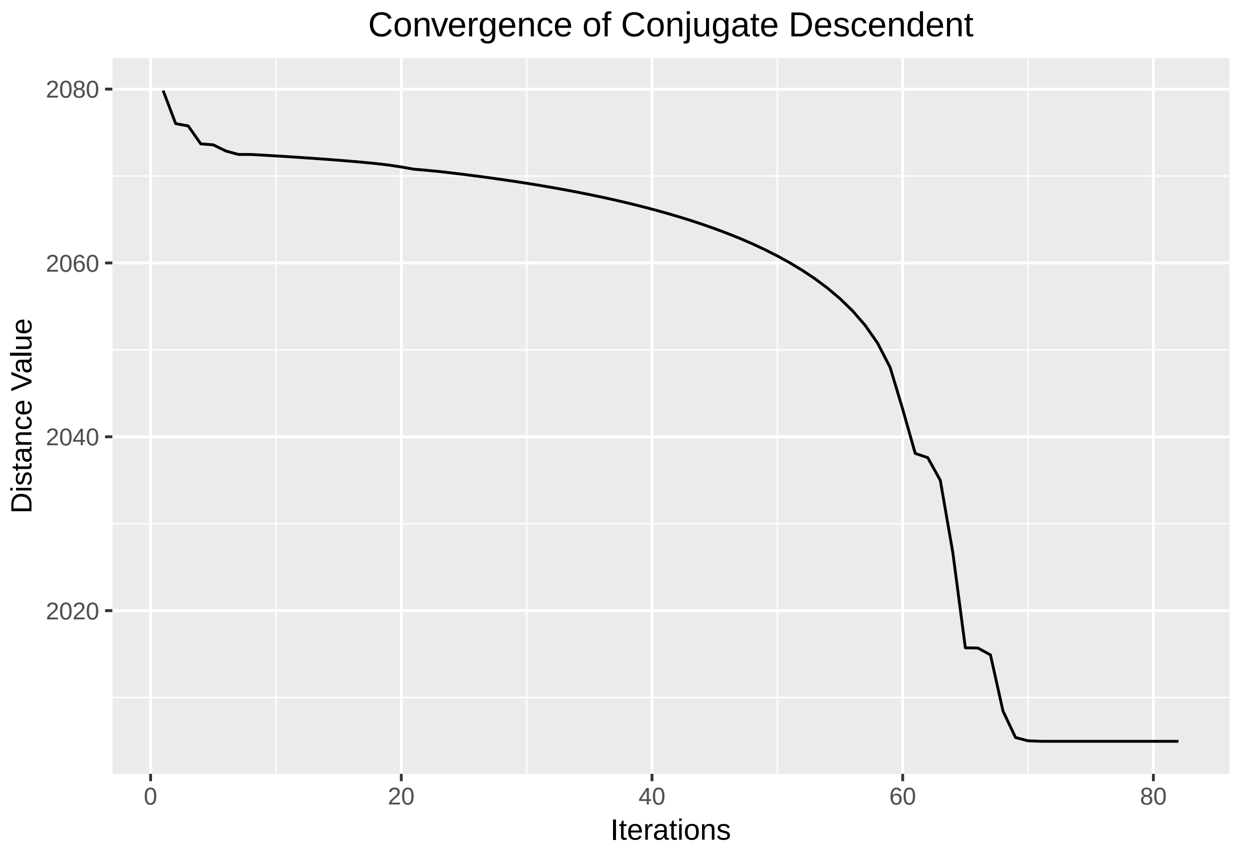
```r
conjugate <- function(f.init, fn.gr=fn.gr, fn.H=fn.H,
                      max_iters=1000,toll=1e-6){
  f          <- matrix(NA, nrow = max_iters+1, ncol = 2)
  f[1,]      <- f.init
  delta      <- matrix(NA, nrow = max_iters+1, ncol = 2)
  delta[1,] <- fn.gr(f[1,])
  #first step moves in the direction of minima
  f[2,]      <- f[1,] - delta[1,]
  delta[2,] <- fn.gr(f[2,])
  beta <- max(as.numeric((delta[2,]%*%delta[2,])/
                         (delta[1,]%*%delta[1,])),0)
  for(k in 2:max_iters){
    f[k+1,]      <- f[k,] - beta*delta[k,]
    delta[k+1,] <- fn.gr(f[k+1,])
    #beta <- max(as.numeric((delta[k+1,]%*%delta[k+1,])/
    #                      (delta[k,]%*%delta[k,])),0)

    beta <- max(as.numeric((delta[k+1,]%*%delta[k+1,])/
                           (delta[k,]%*%delta[k,])),0)
    #stop if norm of gradient^2 is smaller than toll practically 0
    if(sum(fn.gr(f[k,])^2) < toll)
      break
    iter <- k
  }
  f = f[!is.na(rowSums(f)),]
  distance <- apply(f,1,fn.distance)
  if ((sum(fn.gr(f[k+1,])^2) < toll) && (fn.H(f[k+1,])>0))
    convergence = TRUE
  else
    convergence = FALSE
  return(list(f           = f,
              dist        = distance,
              opt_point   = tail(f,1),
              min_dist    = tail(distance,1),
              det_H       = det(fn.H(tail(f,1))),
              convergence = convergence,
              iters       = iter))
}
```

This algorithm seems to take less iteration and less time than the other procedures (except for the Newton method, which unfortunately does not guarantee the convergerce towards the minima). We generate random points between 300 and 600 because now we know that the global minima or atleast one the best points for minima is in this area.

We can see that is sometimes the value of the function increases in this method this is due to due to the fact that the method is based on the ratio between the gradient on the current point and the precedent one, so if the precedent point had a gradient lower than 1 we will see an increase in the value of β, which, if increases a lot, can create a strong launch of the point the direction of movement (some times even skipping the local minima).

```r
set.seed(123456789)
x <- runif(300,300,600)
y <- runif(300,300,600)
random.points <- cbind(x,y)

ris_temp <- lapply(1:nrow(random.points), function(x) {
          conjugate(random.points[x,],
                    fn.gr,fn.H,max_iters = 1000)})

opt_points <- matrix(NA,ncol=2, nrow=length(x))
distances  <- rep(NA,length(x))

for (i in 1:nrow(random.points)){
  opt_points[i,] <- ris_temp[[i]]$opt_point
  distances[i]   <- ris_temp[[i]]$min_dist
}

f      <- ris_temp[[which.min(distances)]]$opt_point
f_dist <- ris_temp[[which.min(distances)]]$min_dist
```

```
P_dist <- ris_temp[[which.min(distances)]]$dist

ggplot(data=data.frame(distance=P_dist)) +
  aes(x=distance) +
  geom_line(aes(x = 1:length(distance),y=distance)) +
  labs(title = "Convergence of Conjugate Descendent",
       x = "Iterations", y="Distance Value") +
  theme(plot.title = element_text(hjust = 0.5))
```



```
cat(" The best point found is (",
    f[1],",",f[2],") \n with distance value: ", f_dist)
```

```
##  The best point found is ( 310.1627 , 430.8702 )
##  with distance value:  2004.964
```

Concluding we can say that the Conjugate Gradient Method seems to be the best one so far, since it was able to calculate a minimum of 2004.964 at the points $(310.16, 430.87)$, way more faster than the others. The stochastic method is more useful in cases where the dataset is bigger than the one given to us, so in this case, where the gradient was not so complicated, this method didn't have any extra usefulness than the classical gradient descendent.

The idea of decreasing the learning rate, helped us into stablizing the solution after a certain number of iterations, of course there are more efficient algorithm to adapt the learning rate in a more intelligent way, for example calculating the "pace of movement" of the point towards the minima, and adjust accordingly.

One great idea could have been to let the initial point explore the area around it, using some heuristics (which will be used the next assignment), instead seeing the "low" complexity of the problem we decided to randomly start a different locations (the multistart method) and keep the best solution it found.