Assignement 3

Sofia Davoli 813479

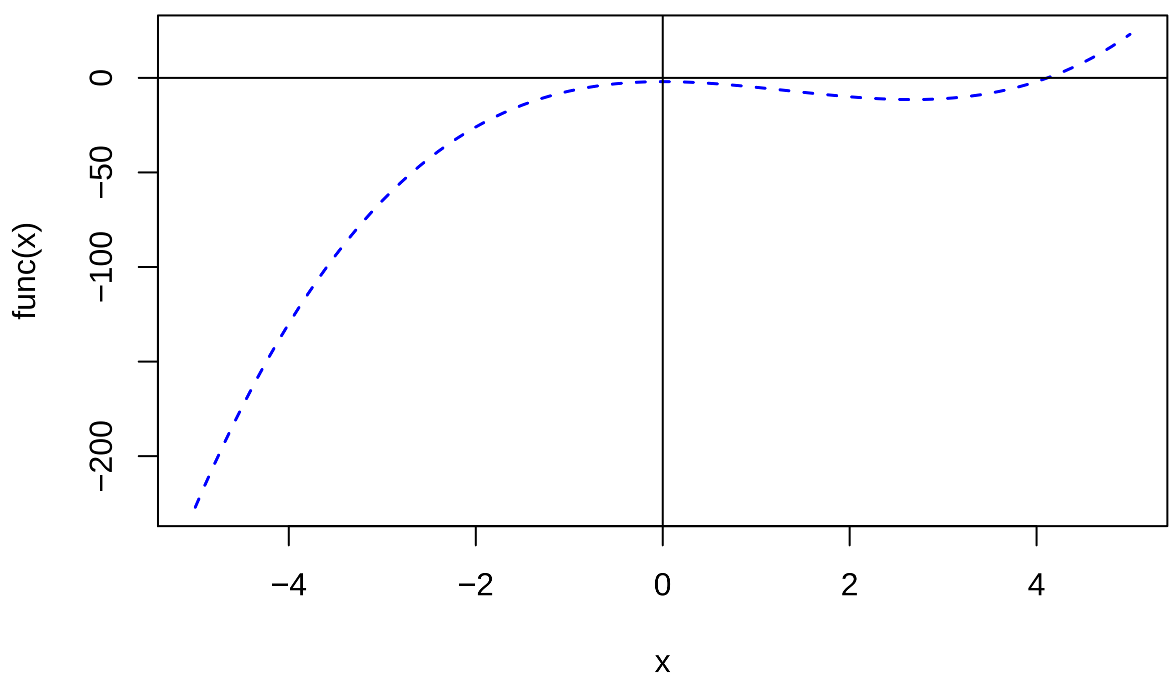
25/5/2020

Problem 1

```
library(NLRoot)
```

```
func <- function(x) {
          x^3 - 4 * x^2 - 2
        }

curve(func, xlim = c(-5,5), col = 'blue',
      lwd = 1.5, lty = 2)
abline(h = 0)
abline(v = 0)
```



As we can see in the plot, the solution of the function is in the interval (3,5). Using a bisection function we can find the exact solution.

```
bisection <- function(f, a, b, n = 1000, tol = 1e-7) {
  #-- Se i segni di a e b sono uguali, si stampa la soluzione
  if (sign(f(a) == sign(f(b)))) {
    stop('signs of f(a) and f(b) must differ')
  }

  for (i in 1:n) {
    c <- (a + b) / 2 #-- Calcolo punto medio

    #-- Se f(x) = 0 nel punto medio o il punto medio Ã¨ piÃ¹ basso
    #-- della tolleranza, si stampa la soluzione
    if ((f(c) == 0) || ((b - a) / 2) < tol) {
      return(c)
    }

    #-- Controllare i segni dn c ed a e si riassegna il valore a o b al punto medio
    ifelse(sign(f(c)) == sign(f(a)),
           a <- c,
           b <- c)
  }
  #-- Se troppe iterazioni
  print('Too many iterations')
}

bisection(func, 3, 5)
```

```
## [1] 4.117942
```

Bisection method find the solution in 4.1179. Using BFfzero we found the solution of the function which is equal to the one find using bisrection method.

```
BFfzero(func, 3, 5)
```

```
## [1] 1
## [1] 4.117941
## [1] -1.770047e-05
## [1] "finding root is successful"
```

Problem 2

2.1

For the use of gradient Descent Algorithm gradient of the funcion is needed.

```
func <- function(x){
  y <- 2*x[1]^2 + x[1]*x[2] + 2*(x[2]-3)^2
  y
}
```

```
fun.gr <- function(x){
  c(4*x[1]+x[2], x[1] + 4*(x[2] - 3))
}
```

Applying gradient Descent Algorithm to point A (-1,4) and using an lr=0.1

```
gradientDescent <- function(a, fun.gr, lr){
  search_direction <- fun.gr(a)
  punto_successivo <- a - lr * search_direction
  return(punto_successivo)
}

gradientDescent(a = c(-1,4), fun.gr = fun.gr, lr=0.1)
```

```
## [1] -1.0  3.7
```

2.2

For the use of Newton method hessian matrix must be calculated.

```
fun.hess <- function(x){
  return(matrix(c(4, 1, 1, 4), 2, 2))
}
```

```
Newton <- function(a, fun.gr, fun.hess){
  punto_successivo <- a - as.vector(solve(fun.hess(a)) %*% as.matrix(fun.gr(a)))
  return(punto_successivo)
}

Newton(c(-1, 4), fun.gr = fun.gr, fun.hess = fun.hess)
```

```
## [1] -0.8  3.2
```

minimum of the function is found in (-0.8, 3.2)

2.3

Newton function need only one iteration to find the minimum of this function because hessian matrix is a positive difined and quadratic matrix.

Problem 3

Creating the function which need to be minimized

```
fun.loss <- function(x){
  y <- 34 * exp(-1/2 * ((x-88)/2)^2) + (x/10 - 2 * sin(x/10))^2
  return(y)
}
```

Creating candidate solutions

```
NewSolution <- function(solution){
    new_solution <- solution + rnorm(1, 0,10)
    return(new_solution)
}
```

Creating acceptation Criterion

```
acceptanceCriterion <-function(fcandidate, fcurrent,fbest, temperature){

  if(fcandidate <= fbest){ #-- Se Soluzione Migliore
      return(TRUE)
    }
  else {
      if(exp((fcurrent-fcandidate)/temperature)>runif(1, 0, 1)){
    return(TRUE)
    }
    return(FALSE)
  }

}
```

Main function using a cooling rate =0.99

```
SA<- function(start, maxIterNoChange=200){

  # Initialization step
  solution <- start                          #-- Soluzione Iniziale
  tmin <- 0.0001                             #-- Minima Temperatura
  coolingRate <- 0.99                       #-- Rapporto di Raffreddamento
  Temp <- tini <- 1000                          #-- Temperatura Iniziale
  loss <- fun.loss(solution)  #-- Funzione Obiettivo della Soluzione Iniziale
  bestLoss <- loss                         #-- Valore Iniziale per la Miglior Variabile
  traceBest <- c(loss)                       #-- Traccia con Miglior Risultato
  traceCurrentLoss <- c(loss)              #-- Traccia con Risultato Corrente
  iterNoChange = 0                           #-- Step

  #-- Loop Principale
  while(Temp >= tmin){                          #-- Se Temperatura non Minima

    iterNoChange = iterNoChange+1

    #-- Nuova Soluzione Generata con Scambio della Soluzione Corrente
    newsolution <- NewSolution(solution)
    loss_new <- fun.loss(newsolution)

    #-- Controllo su Criterio di Accettazione
    if(acceptanceCriterion(loss_new, loss, bestLoss, Temp)){

      if(loss_new <= bestLoss)
        bestLoss <- loss_new             #-- Aggiornamento della Soluzione Migliore
```

```r
      solution <- newsolution              #-- Assegnamento della Soluzione Migliore
      loss <- loss_new
      iterNoChange <- 0                    #-- Numero di Iterazioni Settato a 0

    }

    #-- Aggiornamento della Temperatura
    Temp <- Temp*coolingRate

    traceBest <- append(traceBest, bestLoss)
    traceCurrentLoss <- append(traceCurrentLoss, loss)

    #-- Se Nessun Cambiamento per Tanto Tempo STOP
    if(iterNoChange >= maxIterNoChange){ break}
  }

  res = list(x=solution,
             traceBest = traceBest,
             trace = traceCurrentLoss)

  class(res) = "SAObj"
  return(res)
}
```

Creating a loop to start at different starting point. Function

```r
result <- list()
for(i in 1:3){
  result[[i]] <- SA(sample(c(-1000:1000), 1))
}
```

```r
result[[1]]$x
```

```
## [1] -450.4226
```

```r
result[[2]]$x
```

```
## [1] -0.01478527
```

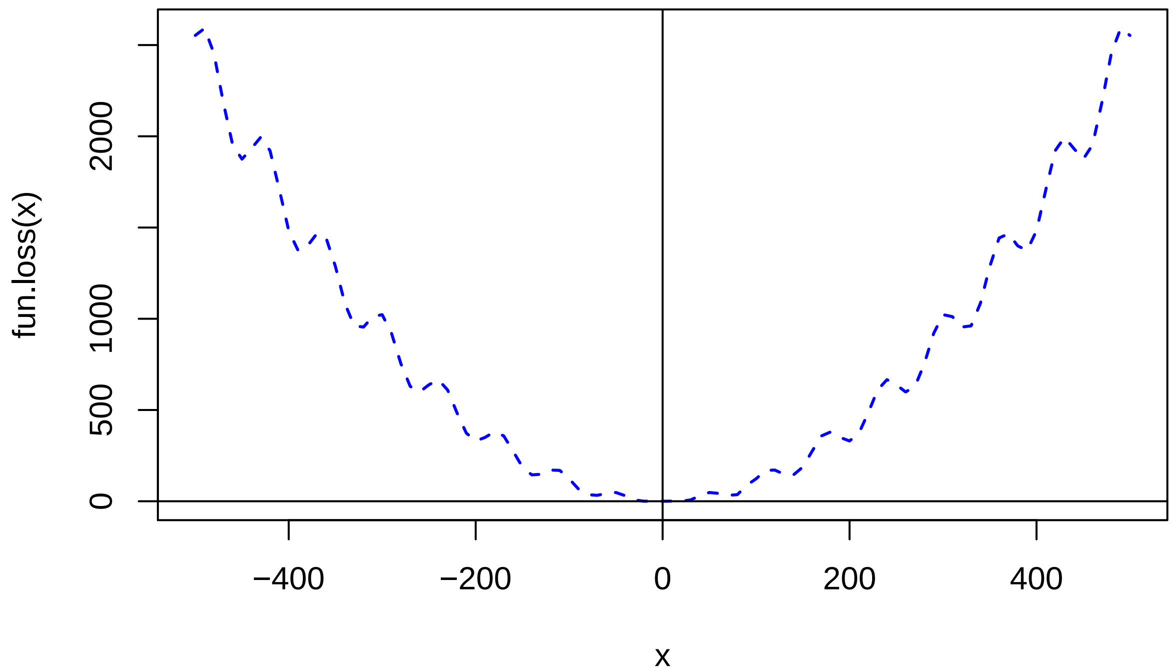
```r
result[[3]]$x
```

```
## [1] 261.7501
```

3 local minimum are find.

From the plot of this function, it's evident that global minimum it's close to zero value in x axes. Since function present many local minimum, simulated annealing starting from a random point will not find the global minimum. (only if the random point is close to zero)

```r
curve(fun.loss, xlim = c(-500,500), col = 'blue',
      lwd = 1.5, lty = 2)
abline(h = 0)
abline(v = 0)
```


fun.loss(x)
0
500
1000
2000
−400
−200
0
200
400
x