Assignment 2B

Niccolò Puccinelli - 881395

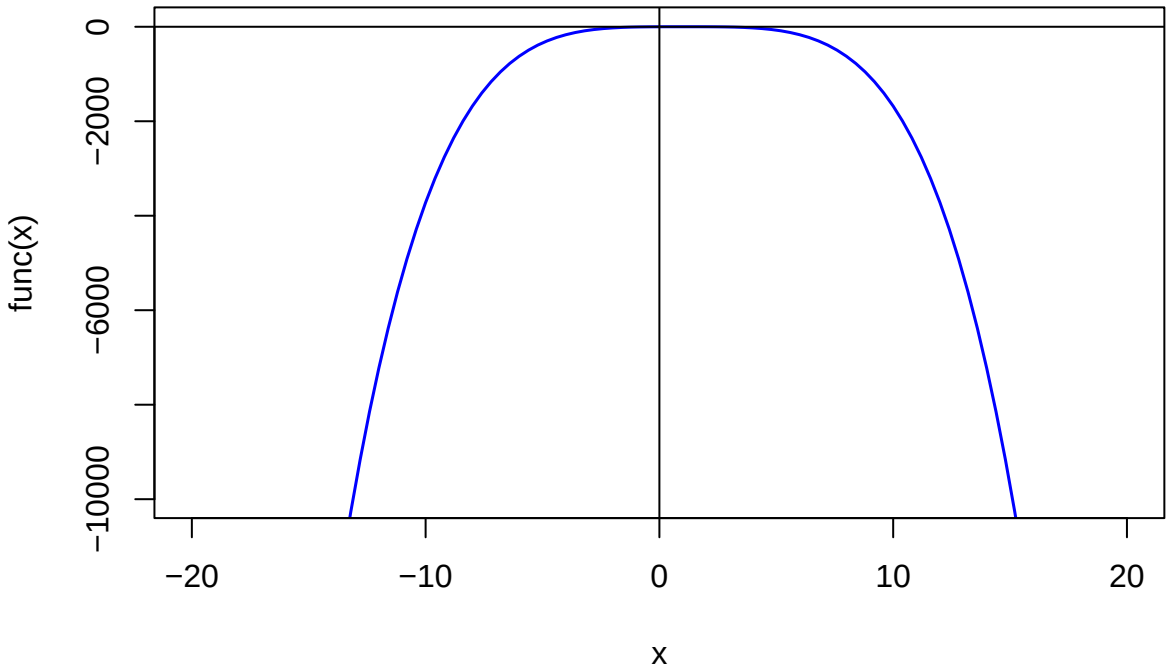
2022-06-01

Funzione 1

Innanzitutto vediamo il comportamento della prima funzione da massimizzare.

```
func <- function(x) {
  x^3 + 2 * x - 2 * x^2 - 0.25 * x^4
}

curve(func, xlim = c(-20,20), ylim = c(-10000, 10), col = 'blue', lwd = 1.5)
abline(h = 0)
abline(v = 0)
```



La funzione appare quasi simmetrica rispetto a x=0 (leggermente asimmetrica verso destra). Sembra inoltre esserci un plateau vicino y=0. In particolare, sembra che il segno della y non cambi mai. Questo potrebbe far fallire i metodi.

Applichiamo i metodi di bisezione, di Newton e delle secanti.

Metodo di bisezione

Effettuiamo una prima iterazione scegliendo come intervallo di partenza (-1, 1.5).

```
lower <- -1
upper <- 1.5
print(func(lower))
```

```
## [1] -5.25
```

```
print(func(upper))
```

```
## [1] 0.609375
```

I segni sono discordi, quindi c'è effettivamente un punto in cui la funzione cambia segno.

```
mid <- (lower + upper) / 2
print(mid)
```

```
## [1] 0.25
```

```
print(func(mid))
```

```
## [1] 0.3896484
```

Il nuovo punto è molto vicino a 0. Qualora volessimo avvicinarci di più, dovremmo effettuare un'altra iterazione, assegnando a upper il valore di mid (0.25).

Ora costruiamo l'algoritmo.

```
bisezione <- function(func, lower, upper, n = 1000, tol = 1e-7) {
  if (sign(func(lower)) == sign(func(upper))) {
    print('I segni non possono essere uguali!')
  }
  else{
    for (i in 1:n) {
      mid <- (lower + upper) / 2
      # Se midpoint = 0 oppure è sotto la tolleranza, abbiamo trovato il punto e fermiamo l'algoritmo
      if ((func(mid) == 0) || ((upper - lower) / 2) < tol) {
        return(mid)
        }
      # Altrimenti si fa un'altra iterazione, controllando di nuovo i segni dei due bound
      # e assegniamo i nuovi valori a seconda del valore di mid.
      if(sign(func(mid)) == sign(func(lower))){
```

```
        lower <- mid
        }
      else{
        upper <- mid
      }
      if (sign(func(lower)) == sign(func(upper))) {
        stop('I segni non possono essere uguali!')
      }
    }
  # Se il punto non è stato trovato entro 1000 iterazioni ritorna un messaggio di errore
  print('Raggiunte 1000 iterazioni')
  }
}
```

Proviamo diversi valori:

- (-10, 10);
- (-1, 1);
- (-0.1, 0.3).

```
bisezione(func, -10, 10, n = 1000, 1e-7)
```

```
## [1] "I segni non possono essere uguali!"
```

La funzione restituisce errore, in quanto func(-10) e func(10) hanno segno uguale. Proviamo con valori più piccoli dei bounds.

```
bisezione(func, -1, 1, n = 1000, 1e-7)
```

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

Abbiamo individuato il plateau, la funzione ci ha restituito il valore 0. Modifichiamo ulteriormente l'intervallo di partenza e vediamo cosa succede.

```
bisezione(func, -0.1, 0.3, n = 1000, 1e-7)
```

```
## [1] 9.536743e-08
```

La funzione restituisce ancora un numero vicino a 0, anche se di poco maggiore. La funzione cambia dunque segno per una porzione molto piccola del grafo. Il plateau supera di pochissimo l'asse x.

Verifichiamo ulteriormente le nostre assunzioni tramite la funzione predefinita *BFfzero*.

```
BFfzero(func, -10, 10)
```

```
## [1] 0
## [1] 0
## [1] 0
## [1] 0
## [1] 2
## [1] 0
## [1] "finding root is fail"
```

Anche in questo caso l'algoritmo fallisce.

```
BFfzero(func, -1, 1)
```

```
## [1] 1
## [1] -3.051758e-06
## [1] -6.103534e-06
## [1] 0
## [1] 0
## [1] "finding root is successful"
```

Qui l'algoritmo predefinito trova la nostra stessa soluzione.

```
BFfzero(func, -0.1, 0.3)
```

```
## [1] 1
## [1] 4.882812e-06
## [1] 9.765577e-06
## [1] "finding root is successful"
```

La soluzione qui è leggermente diversa dalla nostra, ma comunque molto vicina a 0.

Metodo di Newton

Effettuiamo una prima iterazione scegliendo come punto iniziale 4.

```
f0 <- func(4)
print(f0)
```

```
## [1] -24
```

Il punto è ben lontano dalla convergenza. Applichiamo la prima iterazione del metodo.

```
# Calcolo della derivata prima
df1 <- genD(func = func, x = 4)$D[1]
# Calcolo di x_i+1
x_i1 <- 4 - (f0 / df1)
print(x_i1)
```

```
## [1] 3.2
```

```
# Calcolo della differenza tra x_i+1 e il punto iniziale
print(abs(x_i1 - 4))
```

```
## [1] 0.8
```

Ci siamo avvicinati allo 0, tuttavia sono necessarie altre iterazioni, sostituendo x_i1 a 4 (i.e. il nostro punto iniziale).

Ora costruiamo l'algoritmo.

```
newton <- function(func, start,  n = 1000, tol = 1e-7) {
  f0 <- func(start)
  # Controllo se ho già trovato il punto
  if (abs(f0) < tol) {
    return(start)
  }
  x_i <- start
  for (i in 1:n) {
    # Calcolo della derivata prima f'(x_i)
    df1 <- genD(func = func, x = x_i)$D[1]
    # Calcolo di x_i+1
    x_i1 <- x_i - (func(x_i) / df1)
    print(x_i1)
    # Controllo se la differenza tra x_i+1 e x_i è < tol
    if (abs(x_i1 - x_i) < tol) {
      return(x_i1)
    }
    # Se non ho raggiunto la convergenza vado avanti
    x_i <- x_i1
  }
   # Se il punto non è stato trovato entro 1000 iterazioni ritorna un messaggio di errore
  print('Raggiunte 1000 iterazioni')
}
```

Proviamo diversi valori:

- 10;
- -1;
- 0.1.

```
newton(func, 10, n= 1000, 1e-7)
```

```
## [1] 7.723577
## [1] 6.008719
## [1] 4.71428
## [1] 3.736598
## [1] 3.004141
## [1] 2.477827
## [1] 2.151727
## [1] 2.019939
## [1] 2.00039
## [1] 2
## [1] 2
## [1] 2

## [1] 2
```

Il metodo converge a 2 in 13 iterazioni. Le ultime 4 sono uguali probabilmente perché abbiamo specificato una tolleranza molto piccola. Vediamo gli altri valori.

```
newton(func, -1, n= 1000, 1e-7)
```

```
## [1] -0.475
## [1] -0.1502491
## [1] -0.01957832
## [1] -0.0003759324
## [1] -1.412721e-07
## [1] -1.995779e-14
## [1] -4.007413e-28

## [1] -4.007413e-28
```

Qui il metodo converge a 0 (e non a 2) in 8 iterazioni, confermando l'esistenza del plateau già individuato graficamente e col metodo di bisezione. Proviamo l'ultimo valore.

```
newton(func, 0.1, n= 1000, 1e-7)
```

```
## [1] -0.01109576
## [1] -0.0001217631
## [1] -1.482444e-08
## [1] -2.197639e-16

## [1] -2.197639e-16
```

Anche in questo caso abbiamo convergenza in 0, in un numero minore di iterazioni (5), segno della dipendenza del metodo dalla scelta del punto iniziale.

Metodo delle secanti

Effettuiamo una prima iterazione scegliendo come punti iniziali (-4, 4).

```
fa <- func(-4)
fb <- func(4)
print(fa)
```

```
## [1] -168
```

```
print(fb)
```

```
## [1] -24
```

Ora applichiamo la formula.

```
c <- 4 - fb / ((fb - fa) / (4 - (-4)))
print(c)
```

```
## [1] 5.333333
```

```
print(c - 4)
```

```
## [1] 1.333333
```

Sono necessarie ulteriori iterazioni per la convergenza, assegnando b ad a e c a b.

Ora costruiamo l'algoritmo.

```
secant <- function(func, a, b, n = 1000, tol = 1e-7) {
  for (i in 1:n) {
    # Applicazione della formula
    c <- b - func(b) / ((func(b) - func(a)) / (b - a))
    # Controllo se ho già trovato il punto
    if (abs(c - b) < tol) {
      print(paste0(i, ' iterazioni'))
      return(c)
    }
    # Se non ho raggiunto la convergenza vado avanti
    a <- b
    b <- c
  }
   # Se il punto non è stato trovato entro 1000 iterazioni ritorna un messaggio di errore
  print('Raggiunte 1000 iterazioni')
}
```

Proviamo diversi valori:

- (-10, 10);
- (-1, 1);
- (-0.1, 0.3).

```
secant(func, -10, 10, n = 1000, 1e-7)
```

```
## [1] "18 iterazioni"
```

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

L'algoritmo converge a 2 in 18 iterazioni, come per il metodo di Newton, che però sembra godere di una convergenza più veloce. Proviamo gli altri valori.

```
secant(func, -1, 1, n = 1000, 1e-7)
```

```
## [1] "16 iterazioni"
```

```
## [1] 1.62297e-13
```

Qui l'algoritmo, come per gli altri metodi, converge a 0 in 16 iterazioni. Vediamo l'ultimo intervallo.

```
secant(func, -0.1, 0.3, n = 1000, 1e-7)
```

```
## [1] "6 iterazioni"
```

```
## [1] 7.674468e-15
```

Anche qui abbiamo convergenza a 0, ma in un numero di iterazioni molto minore (6), vicino al metodo di Newton (5). Pertanto, anche il metodo delle secanti dipende dalla scelta del punto iniziale.

Verifichiamo il funzionamento dell'algoritmo tramite la funzione predefinita *SMfzero*.

```
SMfzero(func, -10, 10)
```

```
## [1] 2
## [1] -2.902567e-12
## [1] "finding root is successful"
```

A differenza del nostro algoritmo, la funzione predefinita converge a 0.

```
SMfzero(func, -1, 1)
```

```
## [1] 1.62297e-13
## [1] 3.245939e-13
## [1] "finding root is successful"
```

Come nel nostro caso, qui abbiamo convergenza a 0.

```
SMfzero(func, -0.1, 0.3)
```

```
## [1] -1.719854e-09
## [1] -3.439708e-09
## [1] "finding root is successful"
```

Anche per l'ultimo intervallo la convergenza è a 0.

Funzione 2

```
func2 <- expression(2 * x1 * x2 + x2 - x1^2 - 2 * x2^2)
```

La funzione si presenta in 2 variabili. Applichiamo dunque il metodo del gradiente, che fa uso delle derivate parziali.

Metodo del gradiente

Effettuiamo una prima iterazione scegliendo come punto iniziale x0 = (-1, 3). Calcoliamo anzitutto il vettore gradiente, i.e. le derivate parziali della funzione secondo x1 e x2.

```
func2_x1 <- as.expression(D(func2, "x1"))
print(func2_x1)
```

```
## expression(2 * x2 - 2 * x1)
```

```
func2_x2 <- as.expression(D(func2, "x2"))
print(func2_x2)
```

```
## expression(2 * x1 + 1 - 2 * (2 * x2))
```

Calcoliamo il gradiente nel punto x0 e valutiamo se fare un'iterazione del metodo.

```
grad0 = c(eval(func2_x1[1], envir = list(x1 = -1, x2 = 3)), eval(func2_x2[1], envir = list(x1 = -1, x2 =
print(grad0)
```

```
## [1]   8 -13
```

```
# Criterio di stop
print(sqrt(grad0[1]^2 + grad0[2]^2))
```

```
## [1] 15.26434
```

Facciamo un'iterazione.

Calcoliamo anzitutto il learning rate. Consideriamo che il problema è di massimizzazione.

x1 = x0 + lr(grad(f(x0))) = [-1, 3] + lr[8, -13] = [-1 + lr * 8, 3 + lr * (-13)].

Poniamo dunque x1 = -1 + lr * 8 e x2 = 3 + lr * (-13) e sostituiamoli alla funzione originaria. Successivamente, ponendo la derivata di quest'ultima uguale a 0, possiamo calcolare il learning rate.

```
# Sostituiamo i parametri alla funzione originaria
func2_lr <- as.expression(do.call('substitute', list(func2[[1]], list(x1 = quote(-1 + 8 * x), x2 = quot
print(func2_lr)
```

```
## expression(2 * (-1 + 8 * x) * (3 + (-13) * x) + (3 + (-13) *
##     x) - (-1 + 8 * x)^2 - 2 * (3 + (-13) * x)^2)
```

```
# Semplifichiamo
func2_lr <- as.expression(Simplify(func2_lr))
print(func2_lr)
```

```
## expression(-610 * x^2 + 233 * x - 22)
```

```
# Calcoliamo la derivata rispetto a lr
d_lr <- as.expression(D(func2_lr, 'x'))
print(d_lr)
```

```
## expression(233 - 610 * (2 * x))
```

```r
# Semplifichiamo
d_lr <- as.expression(Simplify(d_lr))
print(d_lr)
```

```
## expression(-1220 * x + 233)
```

```r
# Poniamo la derivata uguale a 0 e troviamo il learning rate
x <- Sym("x")
res <- Solve(d_lr, x)
print(res)
```

```
## Yacas vector:
## [1] x == 233/1220
```

```r
# Il metodo precedente ci restituisce uno Yacas vector, pertanto dobbiamo trasformarlo in espressione e
res <- eval(yacas(paste0("x Where ", res))$text)
print(res)
```

```
## [1] 0.1909836
```

Abbiamo trovato il nuovo learning rate. Ricaviamoci il nuovo punto x1 e valutiamolo.

```r
new_x1 <- -1 + 8 * res
new_x2 <- 3 + (-13) * res
grad1 = c(eval(func2_x1[1], envir = list(x1 = new_x1, x2 = new_x2)), eval(func2_x2[1], envir = list(x1 =
print(grad1)
```

```
## [1] -0.02131148 -0.01311475
```

```r
# Criterio di stop
print(sqrt(grad1[1]^2 + grad1[2]^2))
```

```
## [1] 0.0250235
```

Ci siamo avvicinati (seppur non a sufficienza) allo 0.

Ora costruiamo l'algoritmo.

```r
gradient <- function(func, start, n = 1000, tol = 1e-5){

  print("x0:")
  print(start)

  # Anzitutto mi servono le derivate parziali della funzione (i.e. il gradiente)
  func_x1 <- as.expression(D(func, "x1"))
  func_x2 <- as.expression(D(func, "x2"))

  # Poi calcolo il gradiente in x0
  grad = c(eval(func_x1[1], envir = list(x1 = bquote(.(start[1])), x2 = bquote(.(start[2])))), eval(func
  print("Vettore gradiente di x0:")
```

```
  print(grad)

  # Criterio di stop: se è già sotto tol mi fermo
  if (sqrt(grad[1]^2 + grad[2]^2) < tol){
    return(grad)
  }

  # Altrimenti comincio con le iterazioni
  for (i in 1:n){

    # Sostituiamo i parametri alla funzione originaria
    func_lr <- as.expression(do.call('substitute', list(func[[1]], list(x1 = bquote(.(start[1]) + .(gra
    # Semplifichiamo
    func_lr <- as.expression(Simplify(func_lr))

    # Calcoliamo la derivata rispetto a lr
    d_lr <- as.expression(D(func_lr, 'x'))
    # Semplifichiamo
    d_lr <- as.expression(Simplify(d_lr))

    # Poniamo la derivata uguale a 0 e troviamo il learning rate
    x <- Sym("x")
    lr <- Solve(d_lr, x)
    # Il metodo precedente ci restituisce uno Yacas vector, pertanto dobbiamo trasformarlo in espressio
    lr <- eval(yacas(paste0("x Where ", lr))$text)

    # Abbiamo trovato il nuovo learning rate. Ricaviamoci il nuovo punto e verifichiamo il criterio di

    new_x1 <- start[1] + grad[1] * lr
    new_x2 <- start[2] + grad[2] * lr

    print(paste0("Nuovo punto x", i, ":"))
    print(c(new_x1, new_x2))
    grad <- c(eval(func_x1[1], envir = list(x1 = new_x1, x2 = new_x2)), eval(func_x2[1], envir = list(x

    print(paste0("Vettore gradiente di x", i, ":"))
    print(grad)

    # Criterio di stop: se è sotto tol mi fermo
    err <- sqrt(grad[1]^2 + grad[2]^2)
    if (err < tol){
      return(grad)
    }


    # Altrimenti setto il nuovo punto trovato come punto iniziale per la prossima iterazione
    start <- c(new_x1, new_x2)
  }

   # Se il punto non è stato trovato entro 1000 iterazioni ritorna un messaggio di errore
  print('Raggiunte 1000 iterazioni')
}
```

Proviamo diversi punti:

- (-10, 10);
- (-1, 1);
- (-0.1, 0.3).

```
gradient(func2, c(-10, 10), n = 1000, 1e-5)
```

```
## [1] "x0:"
## [1] -10  10
## [1] "Vettore gradiente di x0:"
## [1]  40 -59
## [1] "Nuovo punto x1:"
## [1] -2.349044 -1.285160
## [1] "Vettore gradiente di x1:"
## [1] 2.127767 1.442554
## [1] "Nuovo punto x2:"
## [1] 0.4075048 0.5836861
## [1] "Vettore gradiente di x2:"
## [1]  0.3523627 -0.5197350
## [1] "Nuovo punto x3:"
## [1] 0.4749026 0.4842744
## [1] "Vettore gradiente di x3:"
## [1] 0.01874364 0.01270755
## [1] "Nuovo punto x4:"
## [1] 0.4991852 0.5007372
## [1] "Vettore gradiente di x4:"
## [1]  0.003103986 -0.004578382
## [1] "Nuovo punto x5:"
## [1] 0.4997789 0.4998615
## [1] "Vettore gradiente di x5:"
## [1] 0.0001651122 0.0001119431
## [1] "Nuovo punto x6:"
## [1] 0.4999937 0.5000071
## [1] "Vettore gradiente di x6:"
## [1]  2.680284e-05 -4.094580e-05
## [1] "Nuovo punto x7:"
## [1] 0.4999988 0.4999992
## [1] "Vettore gradiente di x7:"
## [1] 9.101856e-07 5.958027e-07

## [1] 9.101856e-07 5.958027e-07
```

L'algoritmo converge per il punto (0.5, 0.5) in 7 iterazioni. Verifichiamo anche gli altri punti.

```
gradient(func2, c(-1, 1), n = 1000, 1e-5)
```

```
## [1] "x0:"
## [1] -1  1
## [1] "Vettore gradiente di x0:"
```

```
## [1]  4 -5
## [1] "Nuovo punto x1:"
## [1] -0.22641509  0.03301887
## [1] "Vettore gradiente di x1:"
## [1] 0.5188679 0.4150943
## [1] "Nuovo punto x2:"
## [1] 0.3992786 0.5335738
## [1] "Vettore gradiente di x2:"
## [1]  0.2685905 -0.3357381
## [1] "Nuovo punto x3:"
## [1] 0.4512230 0.4686433
## [1] "Vettore gradiente di x3:"
## [1] 0.03484074 0.02787259
## [1] "Nuovo punto x4:"
## [1] 0.4932368 0.5022544
## [1] "Vettore gradiente di x4:"
## [1]  0.01803521 -0.02254401
## [1] "Nuovo punto x5:"
## [1] 0.4967247 0.4978945
## [1] "Vettore gradiente di x5:"
## [1] 0.002339474 0.001871578
## [1] "Nuovo punto x6:"
## [1] 0.4995459 0.5001514
## [1] "Vettore gradiente di x6:"
## [1]  0.001211014 -0.001513785
## [1] "Nuovo punto x7:"
## [1] 0.4997801 0.4998586
## [1] "Vettore gradiente di x7:"
## [1] 0.0001570754 0.0001256780
## [1] "Nuovo punto x8:"
## [1] 0.4999701 0.5000106
## [1] "Vettore gradiente di x8:"
## [1]  8.111997e-05 -1.024019e-04
## [1] "Nuovo punto x9:"
## [1] 0.4999858 0.4999909
## [1] "Vettore gradiente di x9:"
## [1] 1.020423e-05 8.083502e-06
## [1] "Nuovo punto x10:"
## [1] 0.4999981 0.5000007
## [1] "Vettore gradiente di x10:"
## [1]  5.058252e-06 -6.385257e-06

## [1]  5.058252e-06 -6.385257e-06
```

L'algoritmo converge per lo stesso punto di prima, ma in più iterazioni. Questo a causa della dipendenza dalla scelta del punto iniziale. Proviamo l'ultimo punto.

```
gradient(func2, c(-0.1, 0.3), n = 1000, 1e-5)
```

```
## [1] "x0:"
## [1] -0.1  0.3
## [1] "Vettore gradiente di x0:"
## [1]  0.8 -0.4
```

```
## [1] "Nuovo punto x1:"
## [1] 0.1 0.2
## [1] "Vettore gradiente di x1:"
## [1] 0.2 0.4
## [1] "Nuovo punto x2:"
## [1] 0.2 0.4
## [1] "Vettore gradiente di x2:"
## [1]  0.4 -0.2
## [1] "Nuovo punto x3:"
## [1] 0.30 0.35
## [1] "Vettore gradiente di x3:"
## [1] 0.1 0.2
## [1] "Nuovo punto x4:"
## [1] 0.35 0.45
## [1] "Vettore gradiente di x4:"
## [1]  0.2 -0.1
## [1] "Nuovo punto x5:"
## [1] 0.400 0.425
## [1] "Vettore gradiente di x5:"
## [1] 0.05 0.10
## [1] "Nuovo punto x6:"
## [1] 0.425 0.475
## [1] "Vettore gradiente di x6:"
## [1]  0.10 -0.05
## [1] "Nuovo punto x7:"
## [1] 0.4500 0.4625
## [1] "Vettore gradiente di x7:"
## [1] 0.025 0.050
## [1] "Nuovo punto x8:"
## [1] 0.4625 0.4875
## [1] "Vettore gradiente di x8:"
## [1]  0.050 -0.025
## [1] "Nuovo punto x9:"
## [1] 0.47500 0.48125
## [1] "Vettore gradiente di x9:"
## [1] 0.0125 0.0250
## [1] "Nuovo punto x10:"
## [1] 0.48125 0.49375
## [1] "Vettore gradiente di x10:"
## [1]  0.0250 -0.0125
## [1] "Nuovo punto x11:"
## [1] 0.487500 0.490625
## [1] "Vettore gradiente di x11:"
## [1] 0.00625 0.01250
## [1] "Nuovo punto x12:"
## [1] 0.490625 0.496875
## [1] "Vettore gradiente di x12:"
## [1]  0.01250 -0.00625
## [1] "Nuovo punto x13:"
## [1] 0.4937500 0.4953125
## [1] "Vettore gradiente di x13:"
## [1] 0.003125 0.006250
## [1] "Nuovo punto x14:"
## [1] 0.4953125 0.4984375
```

```
## [1] "Vettore gradiente di x14:"
## [1]  0.006250 -0.003125
## [1] "Nuovo punto x15:"
## [1] 0.4968750 0.4976563
## [1] "Vettore gradiente di x15:"
## [1] 0.0015625 0.0031250
## [1] "Nuovo punto x16:"
## [1] 0.4976563 0.4992187
## [1] "Vettore gradiente di x16:"
## [1]  0.0031250 -0.0015625
## [1] "Nuovo punto x17:"
## [1] 0.4984375 0.4988281
## [1] "Vettore gradiente di x17:"
## [1] 0.00078125 0.00156250
## [1] "Nuovo punto x18:"
## [1] 0.4988281 0.4996094
## [1] "Vettore gradiente di x18:"
## [1]  0.00156250 -0.00078125
## [1] "Nuovo punto x19:"
## [1] 0.4992187 0.4994141
## [1] "Vettore gradiente di x19:"
## [1] 0.0003906442 0.0007812244
## [1] "Nuovo punto x20:"
## [1] 0.4994141 0.4998047
## [1] "Vettore gradiente di x20:"
## [1]  0.0007812756 -0.0003907338
## [1] "Nuovo punto x21:"
## [1] 0.4996094 0.4997071
## [1] "Vettore gradiente di x21:"
## [1] 0.0001953477 0.0003905354
## [1] "Nuovo punto x22:"
## [1] 0.4997071 0.4999023
## [1] "Vettore gradiente di x22:"
## [1]  0.0003905354 -0.0001951877
## [1] "Nuovo punto x23:"
## [1] 0.4998047 0.4998535
## [1] "Vettore gradiente di x23:"
## [1] 0.0000975970 0.0001953702
## [1] "Nuovo punto x24:"
## [1] 0.4998535 0.4999511
## [1] "Vettore gradiente di x24:"
## [1]  1.952997e-04 -9.756177e-05
## [1] "Nuovo punto x25:"
## [1] 0.4999023 0.4999267
## [1] "Vettore gradiente di x25:"
## [1] 4.881609e-05 9.772028e-05
## [1] "Nuovo punto x26:"
## [1] 0.4999267 0.4999756
## [1] "Vettore gradiente di x26:"
## [1]  9.768502e-05 -4.879847e-05
## [1] "Nuovo punto x27:"
## [1] 0.4999511 0.4999634
## [1] "Vettore gradiente di x27:"
## [1] 2.441684e-05 4.887775e-05
```

```
## [1] "Nuovo punto x28:"
## [1] 0.4999633 0.4999878
## [1] "Vettore gradiente di x28:"
## [1]  4.886011e-05 -2.440801e-05
## [1] "Nuovo punto x29:"
## [1] 0.4999756 0.4999817
## [1] "Vettore gradiente di x29:"
## [1] 1.221282e-05 2.444769e-05
## [1] "Nuovo punto x30:"
## [1] 0.4999817 0.4999939
## [1] "Vettore gradiente di x30:"
## [1]  2.443886e-05 -1.220840e-05
## [1] "Nuovo punto x31:"
## [1] 0.4999878 0.4999908
## [1] "Vettore gradiente di x31:"
## [1] 6.108629e-06 1.222823e-05
## [1] "Nuovo punto x32:"
## [1] 0.4999908 0.4999969
## [1] "Vettore gradiente di x32:"
## [1]  1.222384e-05 -6.106431e-06
## [1] "Nuovo punto x33:"
## [1] 0.4999939 0.4999954
## [1] "Vettore gradiente di x33:"
## [1] 3.055409e-06 6.116310e-06

## [1] 3.055409e-06 6.116310e-06
```

Anche qui l'algoritmo converge per (0.5, 0.5), ma in un numero di iterazioni più alto rispetto alle precedenti (33). Questo anche a causa della tolleranza molto bassa. Infatti, se ci fossimo limitati ad una tolleranza pari a 0.01, avremmo ottenuto la convergenza già alla tredicesima iterazione.

Meta-euristiche

Tramite meta-euristiche siamo in grado di trovare una soluzione buona (ottimo locale) facendo molte meno assunzioni rispetto, ad esempio, al gradiente, che richiede la derivabilità della funzione. Inoltre, lo spazio di ricerca non viene completamente esplorato e dobbiamo trovare un trade-off tra exploitation (ricerca nel vicinato di una soluzione candidata) ed exploration (ricerca in parti più lontane rispetto alla soluzione candidata).

Potremmo usare, ad esempio, il tabu-search.

Tabu-search

Questo algoritmo classifica un set di mosse nel vicinato come proibite e le inserisce in una lista tabù. Tuttavia, una mossa può uscire dalla lista tabù se presenta un elevato aspiration criterion. L'algoritmo ammette quindi soluzioni non migliorative, permettendo di uscire dagli ottimi locali.

In riferimento alla funzione, bisogna anzitutto selezionare un punto di partenza (da cui, in generale, le meta-euristiche dipendono molto), creare la lista tabù e la lista di soluzioni candidate, i.e. il vicinato, un insieme di soluzioni raggiungibili dalla soluzione corrente.
La raggiungibilità dipende dalla rappresentazione della soluzione. Per il nostro problema potremmo costruire un intervallo di punti intorno al punto scelto: ((x1-d, x1+d), (x2-d, x2+d)).
Per quanto riguarda la lista tabù invece, potremmo implementare una lista FIFO di lunghezza 5, anche

in base alla lunghezza della lista delle soluzioni candidate. Più lunga è la lista tabù, maggiore sarà la diversificazione, favorendo l'esplorazione dello spazio di ricerca. Il viceversa favorisce l'exploitation. Come aspiration criterion potremmo usare la regola FIFO: il primo che entra nella lista, una volta che quest'ultima è piena, è anche il primo ad uscire qualora dovesse aggiungersi un nuovo valore.
Le soluzioni candidate sono quindi valutate secondo una specifica funzione (nel nostro caso una che privilegia le soluzioni che massimizzano la funzione di partenza), in base alla quale viene scelta la miglior soluzione ammissibile.
A questo punto, se la soluzione rispetta i criteri di stop, ci fermiamo, altrimenti aggiorniamo di conseguenza la lista tabù e l'aspiration criterion, in base ai valori di x1 e x2 scelti, per poi creare la nuova lista di soluzioni candidate e applicare di nuovo l'algoritmo.
Un criterio di stop potrebbe essere il numero massimo di iterazioni senza miglioramenti o la non ammissibilità delle soluzioni nel vicinato.