Assignment 2 - Ottimizzazione non lineare

Si consideri il seguente problema di massimizzazione:

$$max f(x) = x^3 + 2x - 2x^2 - 0.25x^4$$

Si applichino il metodo di bisezione, delle tangenti (o di Newton) e delle secanti per la risoluzione del problema. Si effettui un'iterazione a mano, dopodiché si realizzino delle procedure in Python o in R che implementino gli algoritmi. La scelta del punto iniziale, della tolleranza, del numero massimo di iterazioni e delle condizioni di terminazione è libera. Idem per la seguente funzione obiettivo utilizzando, il metodo del gradiente:

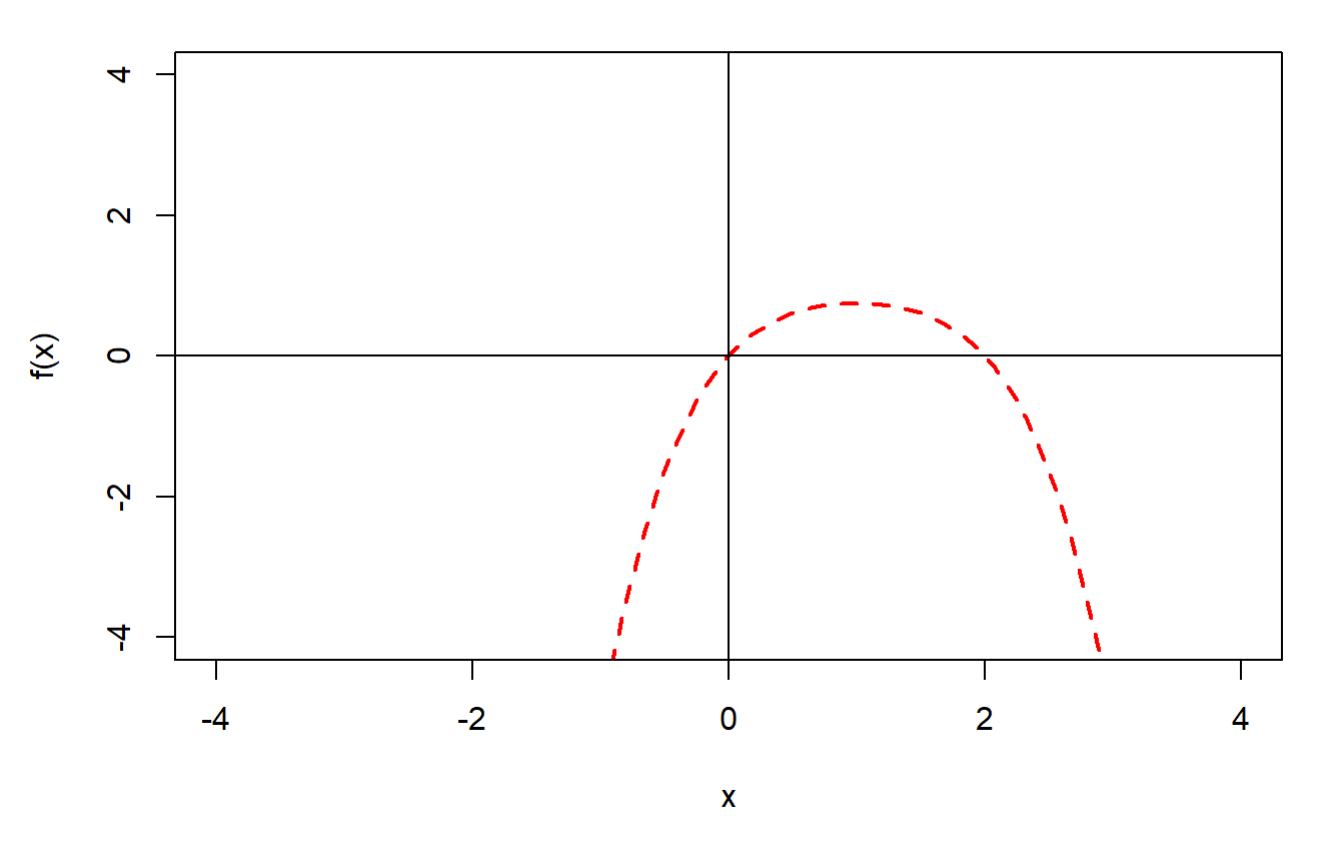
$$max f(x) = 2x_1x_2 + x_2 + -x_1^2 - 2x_2^2$$

Costruisco la funzione

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

Mostro graficamente la funzione, per capire meglio la forma che ha e per avere delle informazioni approssimativi sul punto di massimo.

```
curve(f, xlim=c(-4,4), ylim=c(-4,4), col = 'red', lwd= 2, lty=2)
abline(h=0)
abline(v=0)
```



L'obiettivo è massimizzare la funzione obiettivo ma essendo una funzione non lineare non possiamo applicare le metodologie classiche di programmazione lineare, quindi applicherò il metodo della bisezione, il metodo di Newton,il metodo del gradiente e il metodo della secante.

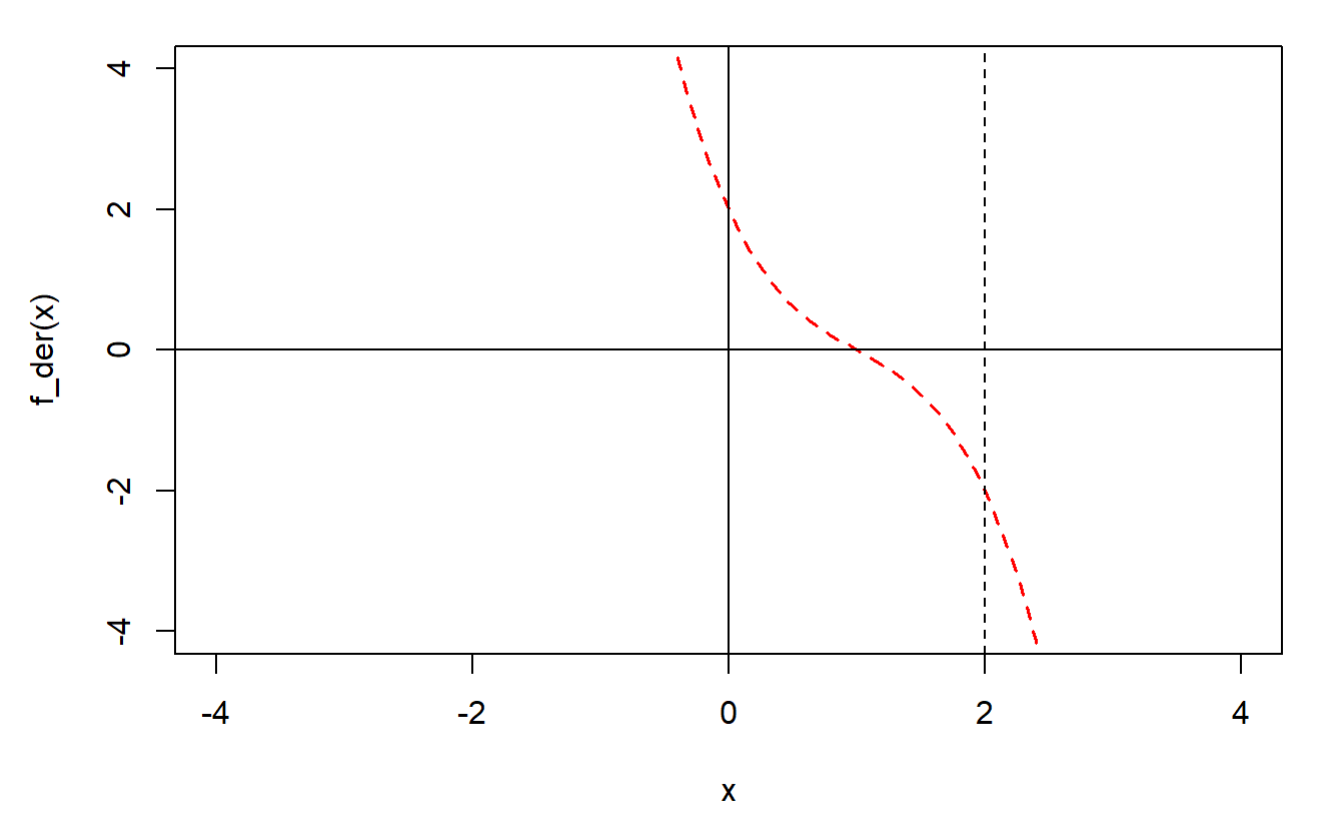
Metodo della bisezione

Il metodo della bisezione è un approccio algoritmico per trovare la radice di una funzione continua su un intervallo [a,b]. Il metodo sfrutta un corollario del teorema dei valori intermedi chiamato teorema di Bolzano, che afferma che se i valori di f(a) e f(b) hanno segni opposti, l'intervallo deve contenere almeno una radice. Il metodo di bisezione è un metodo iterativo che, passo dopo passo, restituisce una migliore approssimazione del valore della radice generando intervalli sempre più piccoli che racchiudono la radice. Le fasi di iterazione del metodo di bisezione sono relativamente semplici, tuttavia la convergenza verso la soluzione è lenta rispetto ad altri metodi di ricerca delle radici.

Quindi calcolo la derivata della funzione

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

curve(f_der, xlim=c(-4,4), ylim=c(-4, 4), col='red', lwd=1.5, lty=2)
abline(h=0)
abline(v=0)
abline(v=2, col="black", lty=2)
```



Sucessivamente seleziono i punti di partenza x_l e x_r e notiamo come la derivata si annulla tra 0 e 2 e che la funzione è positiva in 0 e negativa in 2, quindi scegliero 0 e 2 come punti di partenza.

Per costruire l'algoritmo settiamo il numero di iterazioni uguale a 1000 e un valore di tolleranza pari a 10^-7.

```
bis <- function(f, xl, xr, n = 1000, tol = 1e-7) {

  if (sign(f(xl) == sign(f(xr)))) {
    stop('Il segno di f(xl) e f(xr) deve essere diverso')
  }
# Faccio  un ciclo che continua fino al massimo delle iterazioni
  for (i in 1:n) {
    xm <- (xl + xr) / 2
# Calcolo il punto medio
# Se il valore della funzione è 0 nel punto xm oppure se il punto medio è  più basso dell'indice di  tolleranza f
ermiamo la funzione e stampiamo la radice
    if ((f(xm) == 0) || ((xr - xl) / 2) < tol) {
      return(xm)
    }

    #aggiorno i valori di xl e xr:
    ifelse(sign(f(xm)) == sign(f(xl)),
           xl <- xm,
           xr <- xm)
  }
# Se non trovo la radice in tutte le iterazioni:
  print('Troppe iterazioni')
}
```

Quindi vado a controllare il valore della radice della derivata:

```
bis(f_der,0,2)
```

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

Alla prima iterazione trovo il punto che equivale a 1 ed osservando il grafico iniziale, sembra essere il punto che stavo cercando.

Metodo di Newton

Il metodo di Newton (chiamato anche Newton-Raphson) è un approccio per trovare le radici di equazioni non lineari ed è uno degli algoritmi di ricerca delle radici più comuni per la sua relativa semplicità e velocità. Sfrutta l'idea che una funzione continua e differenziabile può essere approssimata da una retta ad essa tangente, che è una funzione di cui siamo in grado di trovare facilmente gli zeri.

La radice di una funzione è il punto in cui f(x)=0. Molte equazioni hanno più di una radice (infatti, ogni polinomio reale di grado dispari ha un numero dispari di radici reali).

Questo è un metodo iterativo che parte da un'ipotesi iniziale della radice. Il metodo utilizza la derivata prima della funzione f'(x) così come la funzione originale f(x), e quindi funziona solo quando la derivata può essere calcolata.

Come prima cosa utilizzo la libreria numDeriv per il cacolo della derivata (ricordando che stiamo calcolando la derivata seconda della prima funzione impostata).

```
 require(numDeriv)
```

```
## Caricamento del pacchetto richiesto: numDeriv
```

```
newton <- function(f, x0, tol = 1e-7, n = 1000) {

  #fisso il valore iniziale
  x_i <- x0
  # creo k da utilizzare per arcihiviare  i risultati delle iterazioni
  k <- n

  fa <- f(x0)
  if (abs(fa) < tol) {
    return(x0)
  }
# derivata primanel punto
  for (i in 1:n) {
    dx <- genD(func = f, x = x_i)$D[1]
#calcolo il valore successivo
    x_next <- x_i - (f(x_i) / dx)
    k[i] <- x_next
#approssimo la radice all'ultimo elemento
    if (abs(x_next - x_i) < tol) {
      root.appr <- tail(k, n=1)

      res <- list('Radice approssimata' = root.appr, 'iterazione' = k)
      return(res)
    }
#fin quando non raggiungiamo la convergenza il ciclo continua
    x_i <- x_next
  }
#se non raggiungo la convergenza dopo tutte le iterazioni:
  print('Troppe iterazioni')
}
```

```
newton(f_der,0)
```

```
## $`Radice approssimata`
## [1] 1
##
## $iterazione
## [1] 0.5000000 0.8571429 0.9945055 0.9999997 1.0000000 1.0000000
```

Come per il metodo precedente la radice corrisponde al punto 1, ma il punto viene trovato dopo 6 iterazioni.

Metodo del gradiente

Il metodo di discesa del gradiente è un metodo iterativo che cerca di identificare un punto di minimo (o di massimo) di una funzione utilizzando il gradiente (nel caso univariato la derivata prima) della funzione come indicatore di aumento o diminuzione della funzione. Come per altri metodi, è quindi necessario che la derivata prima della funzione da ottimizzare esista nell'intero dominio.

Imposto la funzione multivariata (

$$f(x_1, x_2) = 2x_1x_2 + x2 - x_1^2 - 2x_2^2$$

)

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

Quindi imposto il gradiente (

$$gradf(x) = 2x_2 - 2x_1, 2x_1 - 4x_2 + 1)$$

)

```
grad_f <- function(x) {
  c(2*x[2] - 2*x[1], 2*x[1] + 1 - 4*x[2])
}
```

Imposto l'algoritmo del gradiente:

```
library(pracma)
```

```
##
## Caricamento pacchetto: 'pracma'
```

```
## Il seguente oggetto è mascherato _da_ '.GlobalEnv':
##
##     newton
```

```
## I seguenti oggetti sono mascherati da 'package:numDeriv':
##
##     grad, hessian, jacobian
```

```
gradiente <- function(func, grad_func, x0, lr = 0.05, tol = 1e-7, n = 1000) {
# Imposto  il valore iniziale
  x_i <- x0
# Vettore per la componente x_1
  k_1 <- n # Vettore per la componente
# Vettore per la componente x_2
  k_2 <- n
# Vettore per il valore della norma del gradiente
  grad_hist <- n
#iterazioni
  i = 1

#Costruisco  un ciclo  che continua finchè la norma del gradiente è maggiore del valore di  tolleranza e non si è
raggiunto il massimo numero di iterazioni
  while (Norm(grad_func(x_i))>tol && i<n) {
#Archivio norma del gradiente e le coordinate dei punti x_1,x_2
    grad_hist[i] <- Norm(grad_func(x_i))
    k_1[i] <- x_i[1]
    k_2[i] <- x_i[2]
#Calcolo il punto successivo(positivo perchè voglio il massimo)
    x_next <- x_i + lr*grad_func(x_i)

    x_i <- x_next
#iterazioni
    i <- i + 1 #
  }
  grad_hist[i] <- Norm(grad_func(x_i)) # Salvo la norma del gradiente nel punto
                                       # finale
 #Controllo che al termine del ciclo il valore della norma del gradiente sia minore della tolleranza
  if (Norm(grad_func(x_i))<tol) {
    return(x_i)

  }
  # Se non siamo arrivati a convergenza:
  print('Troppe iterazioni')
}
```

```
gradiente(f_multivar, grad_f, c(0,0))
```

```
## [1] 0.4999999 0.4999999
```

Il punto di massimo della funzione corrisponde (0.499, 0.499)

Metodo della secante

Il metodo della secante per trovare le radici delle equazioni non lineari è una variante comune e popolare del metodo di Newton Il metodo della secante è un metodo iterativo che richiede due ipotesi iniziali della radice, rispetto a una sola ipotesi richiesta da Newton. A causa dei calcoli aggiuntivi dovuti alla necessità di due approssimazioni, il metodo della secante spesso converge più lentamente rispetto al metodo di Newton; tuttavia, di solito è più stabile.

Il metodo delle secanti presenta un altro vantaggio rispetto a Newton, in quanto non richiede che la derivata della funzione in questione (priva di derivata) sia nota o presunta per essere facilmente calcolata. Il metodo delle secanti utilizza le rette secanti (da cui la necessità di due valori iniziali) per trovare la radice di una funzione, mentre il metodo di Newton approssima la radice con una retta tangente quindi otteremo la seguente funzione:

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} f(x_k)$$

```
secante <- function(f, x0, x1, tol = 1e-7, n = 1000) {
  for (i in 1:n) {
    x2 <- x1 - f(x1) / ((f(x1) - f(x0)) / (x1 - x0))
# Se la differenza fra il nuovo valore e precedente è abbastanza piccola, termino il ciclo
    if (abs(x2 - x1) < tol) {

      return(x2)
    }
# Se non si è arrivati a convergenza, aggiorno i valori di x e continuo
    x0 <- x1
    x1 <- x2
  }
# Se non si raggiunge la convergenza entro il numero massimo iterazioni:
  print('Troppe iterazioni')
}
```

```
secante(f_der, -1, 3)
```

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

Anche in questo caso otteniamo lo stesso punto,ottenuto con i metodi precedenti.

Per controllare i risultati carico delle librerie con funzioni già impostate:

- Metodo della bisezione:

```
# Utilizzo la funzione BFfzero della libreria NLroot
library(NLRoot)
BFfzero(f_der, -3, 3)
```

```
## [1] 1
## [1] 0.9999985
## [1] 1.525879e-06
## [1] "finding root is successful"
```

- Metodo di Newton:

```
# Utilizzo la funzione netwonRaphson della libreria pracma
library(pracma)
newtonRaphson(f_der, 0)
```

```
## $root
## [1] 1
##
## $f.root
## [1] 0
##
## $niter
## [1] 6
##
## $estim.prec
## [1] 0
```

- Metodo della gradiente:

```
#res$par
#I risultati ottenuti sono x=0.498 ed y = 0.499 ma durante il knit la cella non funziona, ho provato a risolvere
 ma non ho trovato soluzione
```

- Metodo della secante:

```
#Utilizzo ka funzione SMfzero della libreria NLroot
SMfzero(f_der, -1, 3)
```

```
## [1] 1
## [1] 0
## [1] "finding root is successful"
```

Per tutte le metodologie applicate, i risultati anche in modo approssimato combaciano ed anche osservando il primo grafico notiamo che il punto di massimo si avvicina a 1, questo conferma i risultati ottenuti.

Com'è possibile risolvere quest'ultimo problema di ottimizzazione utilizzando le metaeuristiche?

Quando si ha a che fare con funzioni complesse è necessario ideare dei metodi risolutivi che facciano assunzioni più deboli o non ne facciano quando parliamo di questo ci riferiamo all'ottimizzazione stocastica, una classe di algoritmi che approcciano ai problemi di ottimizzazione introducendo un grado di casualità per trovare la soluzione ottimale. Le metaeuristiche fanno parte di questa classi di algoritmi , essi sono molto utili quando cerchiamo soluzioni ottimali per problemi complicati o in mancanza di informazioni circa la natura del problema. L'idea alla base di questi algoritmi è quella di partire da una o più soluzioni (procedura di inizializzazione), valutarne la qualità (procedura di valutazione) e poi creare delle copie da andare a modificare per cercare delle soluzioni differenti (procedura di modifica) ed infine, di queste nuove soluzioni si vanno a mantenere solo le migliori (procedura di selezione).

In questo caso si potrebbe utilizzare quello dell' hill climbing. Partiamo da una soluzione S e apportiamo delle modifche a tale soluzione. Le nuove soluzioni verranno valutate, aggiornando il valore della soluzione corrente (quando si trova una soluzione migliore), questo processo viene svolto ciclimanete per ogni iterazione per trovare sia il punto di ottimo locale che quello di ottimo globale

Spesso le soluzioni che vengono generate non sono migliori di quella corrente. In questo caso, si può utilizzare una metodologià diversa cioè lo steepest ascent hill climbing, in questo caso, a ogni iterazione, si va a campionare l'intorno della soluzione candidata scegliendo la direzione migliore.

Un'altra soluzione potrebbe essere quella di utilizzare l'algorimto chiamato Tabu Search che è un altro algoritmo della classe trajectory based e rappresenta un'evoluzione del metodo del gradiente. L'algoritmo del Tabu Search si rifà al concetto di mosse tabu (in italiano mosse proibite), che impediscono all'algoritmo di tornare sui propri Una delle caratteristiche più importanti di questo algoritmo riguarda l'uso della memoria infatti per aumentare l'efficacia del processo di ricerca si tiene traccia di alcune informazioni relative all'intero itinerario percorso. Questo permette di eseguire mosse proibite laddove queste siano particolarmente attrattive:l'algoritmo si sposta fra nuove soluzioni da cui, applicando una mossa vietata, si potrebbe arrivare a una soluzione particolarmente buona. Un altro vantaggio è che l'algoritmo può essere opportunamente modificato e regolato andando a scegliere la lunghezza della lista tabu, le caratteristiche del "vicinato" in cui trovare la nuova soluzione candidata e il criterio di terminazione.

Altra soluzione potrebbe essere quella di utilizzare l'algoritmo simulated annealing,(anch'essa appartiene alla categoria degli algoritmi trajectory based), si parte da un punto iniziale, si generano ripetutamente delle nuove soluzioni e si mantengono queste soluzioni fino ad arrivare alla convergenza (utilizza valori casuali per poter esplorare lo spazio di ricerca e di evitare gli ottimi locali, favorendo la possibilità di individuare l'ottimo globale ad ogni iterazione) La soluzione può essere accettata o rifiutata su base probabilistica:la casualità viene gestita attraverso il concetto di temperatura questa quantità parte da un valore grande, che porterà ad accettare la nuova soluzione con grande probabilità, e viene progressivamente diminuita, seguendo un annealing schedule. I passi dell'algoritmo di simulated annealing sono i seguenti:

1. si parte da un punto iniziale $x = x_0$ e da $k = 0$;
2. si valuta una funzione di costo $F = f(x_k)$;
3. si genera casualmente la soluzione x_{k+1};
4. se $f(x_{k+1}) < F$ si accetta la soluzione: $x = x_{k+1}$ e $F = f(x_{k+1})$;
5. se $f(x_{k+1}) \geq F$ e $rand() < \epsilon$ si accetta la soluzione: $x = x_{k+1}$ e $F = f(x_{k+1})$;
6. $k = k + 1$ e si ritorna al punto 2.

Il punto fondamentale è il punto 5, dove si introduce ϵ determina la probabilità di accettare la soluzione. Una soluzione molto usata è: $\epsilon = exp[-\frac{f(x_{k+1}) - F}{T_{k+1}}]$ dove T_{k+1} è la temperatura che va a descrescere nel tempo La scelta della temperatura è importante perchè determina la velocità di convergenza dell'algoritmo. Man mano che la temperatura diminuisce, la probabilità di accettare una soluzione peggiorativa si abbassa. Il simulated annealing non garantisce di trovare l'ottimo globale, ma che cerca di evitare le soluzioni locali e di solito fornisce prestazioni migliori di risolutori locali, nonostante sia una procedura molto più costosa.

Concludiamo dicendo che le la classe di algoritmi metaerustici sono molto dinamici e possono essere modificati e adattati rispetto ad un problema di ottimizzazione, secondo le nostre necessità quindi quando abbiamo a che fare con problemi non lineari, se le classiche metodologie non portano a buoni risultati allora possiamo usare questi algoritmi.

Università degli Studi di Milano-Bicocca Corso di Laurea in Data Science Simone Farallo 889719