METODI NUMERICI

CALCOLO DEGLI ZERI SU FUNZIONI NON LINEARI

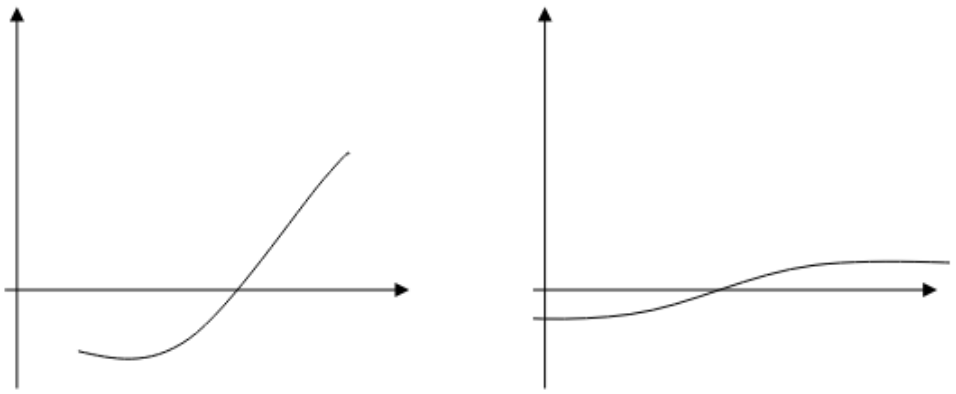
Data una funzione f : R→R non lineare consideriamo il problema di determinare i valori $\alpha \in R$ tali che f(α) = 0. Tali valori sono solitamente chiamati zeri o radici della funzione f

L'approssimazione numerica di una radice α di $f(x)$ si basa sull'uso di metodi iterativi che consistono nella costruzione di una successione di iterati che tende alla soluzione α del problema, cioè

$$lim\ k\rightarrow+\infty\ xk = \alpha \quad \textbf{(1)}$$

E' inoltre necessario rendere il problema ben posto, cioè individuare un intervallo I contenente una sola radice e poi applicare il metodo iterativo fino a convergenza alla soluzione (radice).

$\boldsymbol{K = 1\ |f'(\alpha)|}$ rappresenta l'indice di condizionamento del problema di calcolare la radice della funzione f(x). Se $|f'(\alpha)|$ è molto piccolo, allora il problema è mal condizionato.



Ben Condizionato Mal Condizionato

METODI ITERATIVI PER CALCOLO DEGLI ZERI DI SISTEMI NON LINEARI

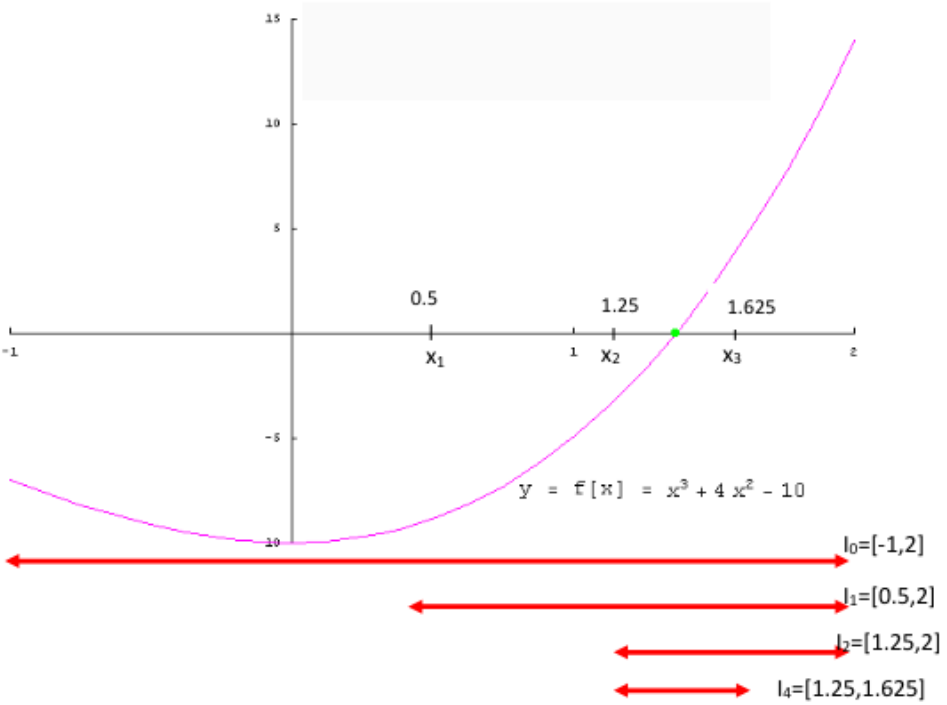
Metodo di Bisezione

Il metodo di Bisezione si basa sul teorema degli zeri di funzioni continue. Sia f(a)f(b) < 0.
Il metodo di bisezione consiste nel generare una successione di sottointervalli tali che racchiudano sempre lo zero cercato, cioè $f(ak) \cdot f(bk) < 0$ e $Ik \subset Ik-1$

Il metodo di bisezione ha ordine di convergenza lineare, p=1, e fattore di convergenza c=½

Un criterio di arresto del metodo di bisezione è chiedere che l'errore al passo k, $|ek|$, sia minore di una precisione fissata , cioè

$$\left|\frac{b-a}{2^{k+1}}\right| \leq \varepsilon.$$



Implementazione:

```
def metodo_bisezione(fname, a, b, tolx):
 """
 Implementa il metodo di bisezione per il calcolo degli zeri di
un'equazione non lineare.

 Parametri:
  f: La funzione da cui si vuole calcolare lo zero.
  a: L'estremo sinistro dell'intervallo di ricerca.
  b: L'estremo destro dell'intervallo di ricerca.
  tol: La tolleranza di errore.

 Restituisce:
  Lo zero approssimato della funzione, il numero di iterazioni e la
lista di valori intermedi.
 """
 fa=fname(a)
 fb=fname(b)
 if sign(fa)*sign(fb)>=0:
     print("Non è possibile applicare il metodo di bisezione \n")
     return None, None,None

 it = 0
```

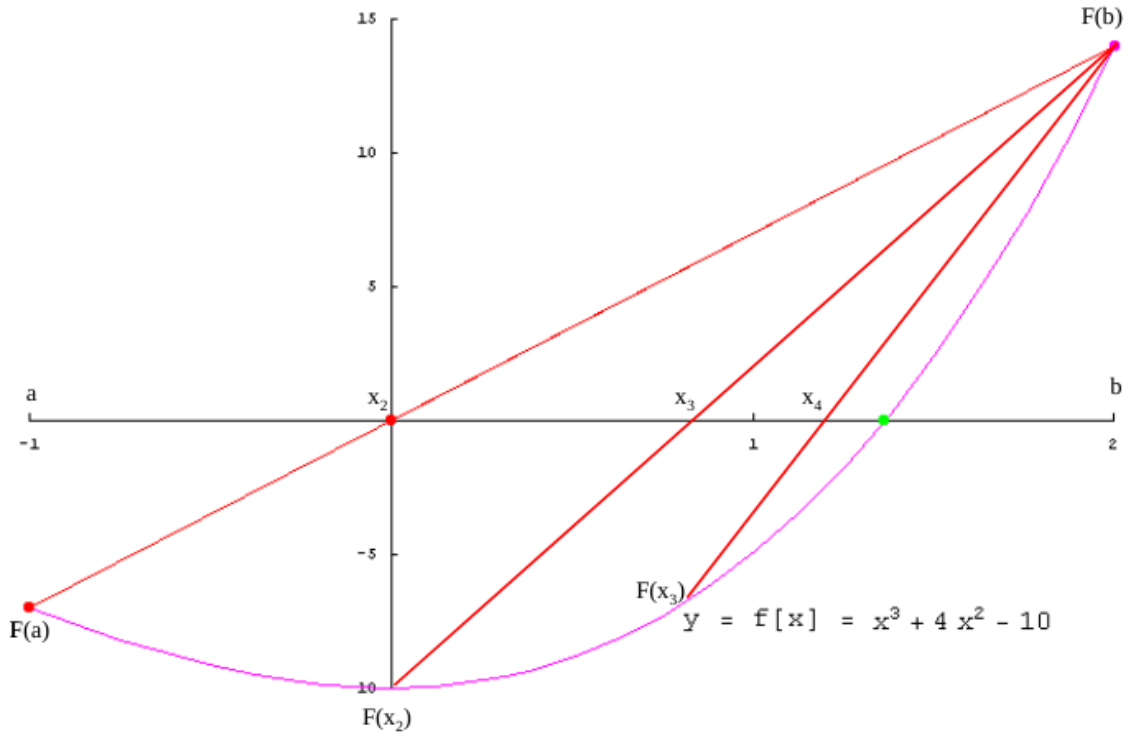
```
v_xk = []


maxit = math.ceil(math.log2((b - a) / tolx))-1


while abs(b - a) > tolx:
   xk = a+(b-a)/2
   v_xk.append(xk)
   it += 1
   fxk=fname(xk)
   if fxk==0:
     return xk, it, v_xk

   if sign(fa)*sign(fxk)>0:  #continua su [xk,b]
     a = xk
     fa=fxk
   elif sign(fxk)*sign(fb)>0:   #continua su [a,xk]
     b = xk
     fb=fxk


return xk, it, v_xk
```

Metodo di Regula Falsi

Un modo naturale per migliorare il metodo di bisezione è quello di considerare anche i valori che la funzione assume negli estremi dell'intervallo. Si considera come nuova approssimazione della soluzione l'intersezione dell'asse delle ascisse con la retta passante per $(a, f(a))$ $(b, f(b))$

$$x = a - f(a)\frac{(b-a)}{f(b)-f(a)}$$

E' più veloce rispetto al metodo di bisezione (convergenza superlineare). In generale l'ampiezza dell'intervallo [ai , bi] non tende a zero pertanto il criterio di arresto basato sull'ampiezza dell'intervallo non è applicabile.



Implementazione:

```
def falsi(fname, a, b, maxit, tolx,tolf):
 """
 Implementa il metodo di falsa posizione per il calcolo degli zeri di
un'equazione non lineare.

 Parametri:
  f: La funzione da cui si vuole calcolare lo zero.
  a: L'estremo sinistro dell'intervallo di ricerca.
  b: L'estremo destro dell'intervallo di ricerca.
  tol: La tolleranza di errore.

 Restituisce:
  Lo zero approssimato della funzione, il numero di iterazioni e la
lista di valori intermedi.
 """
 fa=fname(a);
 fb=fname(b);
 if sign(fa)*sign(fb)>=0:
     print("Non è possibile applicare il metodo di falsa posizione \n")
     return None, None,None

 it = 0
 v_xk = []

 fxk=10


 while it < maxit and abs(b - a) > tolx and abs(fxk) > tolf:
    xk = a-fa*(b-a)/(fb-fa)
    v_xk.append(xk)
    it += 1
    fxk=fname(xk)
    if fxk==0:
      return xk, it, v_xk

    if sign(fa)*sign(fxk)>0:  #continua su [xk,b]
      a = xk
      fa=fxk
    elif sign(fxk)*sign(fb)>0:   #continua su [a,xk]
      b = xk
      fb=fxk
```

```
return xk, it, v_xk
```

Metodi di Linearizzazione

$Data\ f(x)$, $x0$, $f(x0)$: si approssima la funzione con una retta per ($x0$, $f(x0)$) $y = f(x0) + m(x - x0)$ Si ottiene una versione linearizzata del problema $f(x) = 0$

In generale:

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

A seconda della scelta di mk si ottengono:
metodo delle corde (mk= m = costante), metodo delle secanti e metodo di Newton

Metodo di Corde

Utilizza un valore costante m ≠ 0. Il metodo assume la forma

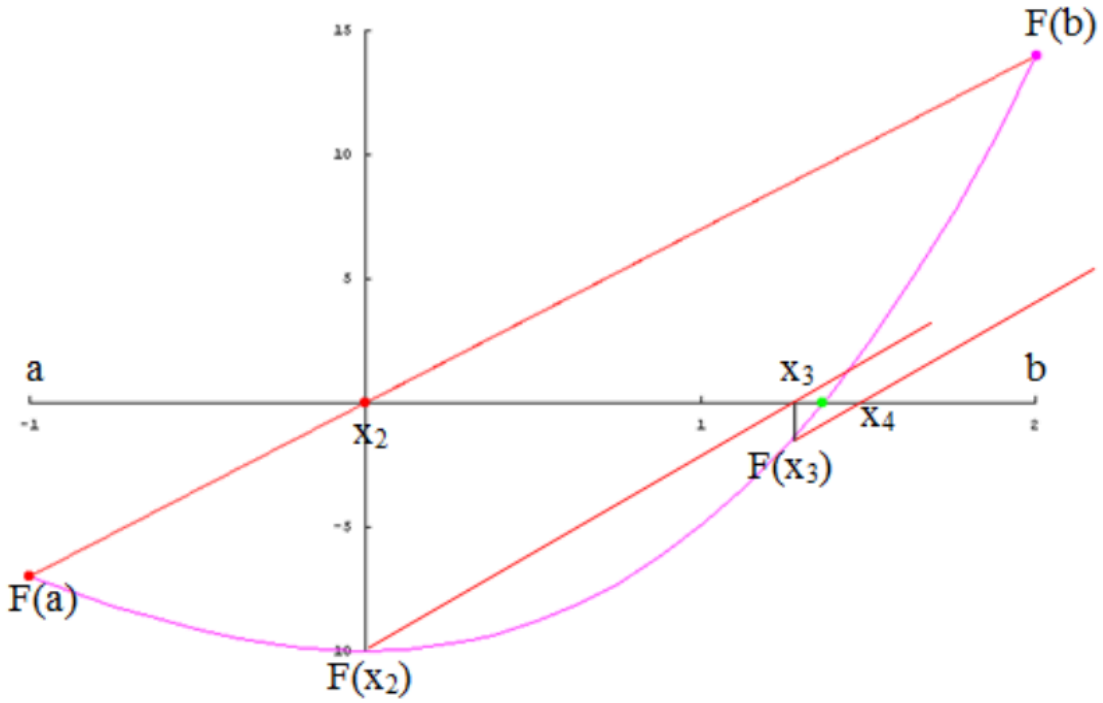
$$x_{k+1} = x_k - \frac{f(x_k)}{m}$$

Una scelta classica è quella di utilizzare il coefficiente angolare della retta che congiunge i punti $(a, f(a))$ e $(b, f(b))$.

$$m = \frac{f(b) - f(a)}{b - a}$$

da cui si ottiene la formula ricorsiva:

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



Implementazione:

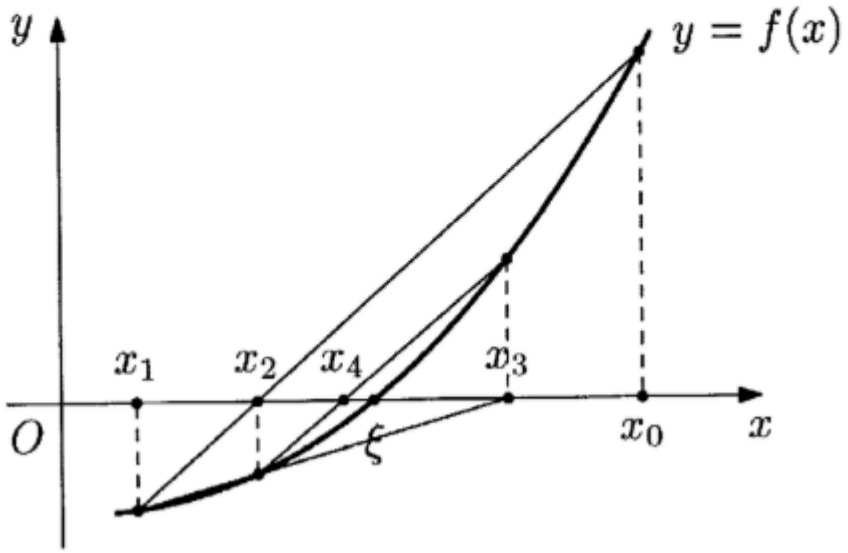
```
def corde(fname,m,x0,tolx,tolf,nmax):

    # m è il coefficiente angolare della retta che rimane fisso per
tutte le iterazioni
        xk=[]
        fx0=fname(x0)
        d=fx0/m
        x1=x0-d
        fx1=fname(x1)
        xk.append(x1)
        it=1

        while it<nmax and  abs(fx1)>=tolf and abs(d)>=tolx*abs(x1) :
```

```
            x0=x1
            fx0=fname(x0)
            d=fx0/m
            '''
            #x1= ascissa del punto di intersezione tra  la retta che
passa per il punto
            (xi,f(xi)) e ha pendenza uguale a m  e l'asse x
            '''
            x1=x0-d
            fx1=fname(x1)
            it=it+1

            xk.append(x1)

        if it==nmax:
            print('raggiunto massimo numero di iterazioni \n')


        return x1,it,xk
```

Metodo delle Secanti

Assegnati i due valori iniziali x0,x1 , al passo k l'approssimazione della funzione f nell'intervallo [xk-1,xk] è la retta che passa per i punti (xk-1,f(xk-1)),(xk,f(xk)) con coefficiente angolare

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



Implementazione:

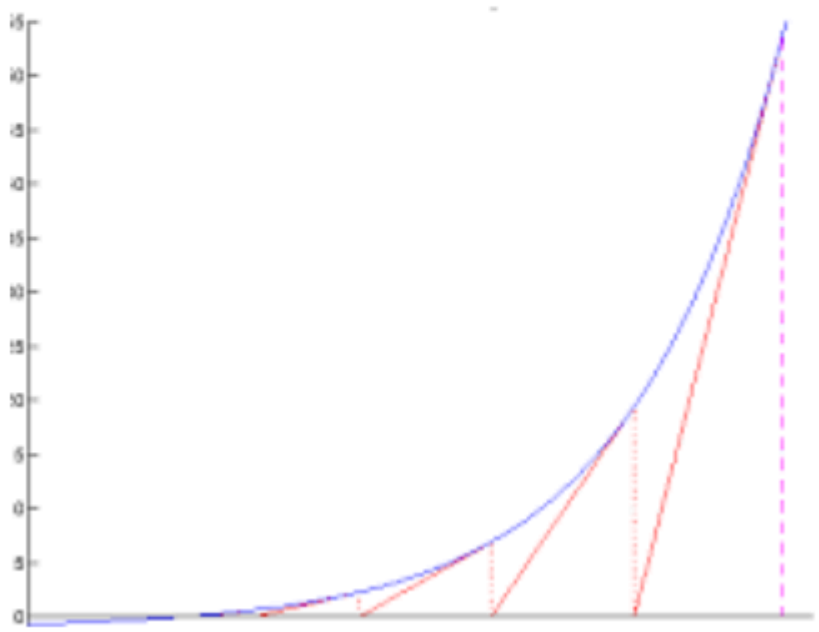
```
def secanti(fname,xm1,x0,tolx,tolf,nmax):
        xk=[]
        fxm1=fname(xm1);
        fx0=fname(x0);
        d=fx0*(x0-xm1)/(fx0-fxm1)
        x1=x0-d;
        xk.append(x1)
        fx1=fname(x1);
        it=1

        while it<nmax and abs(fx1)>=tolf and abs(d)>=tolx*abs(x1):
            xm1=x0
            x0=x1
            fxm1=fname(xm1)
            fx0=fname(x0)
            d=fx0*(x0-xm1)/(fx0-fxm1)
            x1=x0-d
```

```
            fx1=fname(x1)
            xk.append(x1);
            it=it+1;


        if it==nmax:
           print('Secanti: raggiunto massimo numero di iterazioni \n')

        return x1,it,xk
```

Metodo di Newton

Nel metodo di Newton, ad ogni passo k, si considera la retta passante per il punto(xk, $f(xk)$) e tangente alla curva f(x) e si determina il nuovo iterato come il punto di incontro tra questa retta e l'asse delle x. Per far ciò, nella formula ricorsiva (4), si pone mk = $f'(xk)$, ottenendo la formula ricorsiva

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



Implementazione:

```
def newton(fname,fpname,x0,tolx,tolf,nmax):

        xk=[]
        fx0=fname(x0)
        if abs(fpname(x0))<=np.spacing(1): #Se la derivata prima e' pià
piccola della precisione di macchina stop
            print(" derivata prima nulla in x0")
            return None, None,None

        d=fx0/fpname(x0)
        x1=x0-d

        fx1=fname(x1)
        xk.append(x1)
        it=1
```

```
        while it<nmax and  abs(fx1)>=tolf and abs(d)>=tolx*abs(x1) :
            x0=x1
            fx0=fname(x0)
            if abs(fpname(x0))<=np.spacing(1): #Se la derivata prima e'
più piccola della precisione di macchina stop
                 print(" derivata prima nulla in x0")
                 return None, None,None
            d=fx0/fpname(x0)
            '''
            #x1= ascissa del punto di intersezione tra  la retta che
passa per il punto
            (xi,f(xi)) ed è tangente alla funzione f(x) nel punto
(xi.f(xi))  e l'asse x
            '''
            x1=x0-d
            fx1=fname(x1)
            it=it+1

            xk.append(x1)

        if it==nmax:
            print('raggiunto massimo numero di iterazioni \n')


        return x1,it,xk
```

SISTEMI DI EQUAZIONI NON LINEARI

calcolare la soluzione del sistema, cioè calcolare il vettore α = [α1, α2, … αn] $T \in R\, n$ che annulla contemporaneamente le equazioni equivale a calcolare $\boldsymbol{\alpha}$ = [$\boldsymbol{\alpha 1}$, $\boldsymbol{\alpha 2}$, … $\boldsymbol{\alpha n}$] $\boldsymbol{T \in R\, n}$ tale che F($\boldsymbol{\alpha}$)=0

Il gradiente di una funzione f: $R\ n \rightarrow$ R, differenziabile è dato da

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \\ \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

Lo Jacobiano di F(X) è la matrice

$$J(X) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

Metodo di Newton Raphson

L'algoritmo di Newton Raphson si può così schematizzare
L'algoritmo si può così schematizzare:
Dato $X0 \in R\ n$ ed F, per ogni iterazione k
1. Valutare J(Xk-1)
2.Risolvere il sistema lineare $J(Xk-1)sk-1 = -F(Xk-1)$
3. Porre $Xk = Xk-1 + sk-1$

E’ un metodo a convergenza locale e ordine di convergenza quadratico.
Implementazione:

```
def newtonRaphson(fname,Jacname,x0,tolx,tolf,nmax):
   errors = []
   it = 0

   if np.linalg.det(Jacname(x0))==0:
       print("jacobiano di x0 nullo")
       return None,None,None

   #Effettuo la prima iterazione inizializzando x0=(xk) ed x1=(xk1)
   xk = np.array(x0)
   jac = Jacname(x0)
   fxk = fname(x0)
   xk1 = xk - np.linalg.inv(jac)@fxk
   """
   Soluzione per xk1 più veloce:
   s = np.linalg.solve(matjac,fun(x0))
   x1 = x0+s
   fx1 = fun(x1)
   """

   #Criteri di arresto nel ciclo while: il primo capisce se l'errore
del valore iterato è sufficentemente piccolo.
   while it < nmax and np.linalg.norm(xk1 - xk) / np.linalg.norm(xk) >=
tolx and np.linalg.norm(fname(xk)) >= tolf:
       #Ogni ciclo calcola il nuovo iterato xk1
       xk = xk1
       jac = Jacname(xk)
       fxk = fname(xk)
       xk1 = xk - np.linalg.inv(jac)@fxk
       it = it + 1
       error = np.linalg.norm(xk1 - xk) / np.linalg.norm(xk)
       errors.append(error)


   if not np.linalg.norm(xk1 - xk) / np.linalg.norm(xk) >= tolx:
       print("Arresto sulla tolleranza dell'incremento")

   if not np.linalg.norm(fname(xk) >= tolf):
       print("Arresto sulla tolleranza del residuo")
```

```
   print("\n soluzione: ",xk1,"\n errore: ",errors, "\n Iterazioni
eseguite: ", it)
   return xk,errors,it
```

Metodo di Newton Raphson Minimo di funzione

Data f: $R\,n \rightarrow R, f \in C\,2$ (differenziabile due volte con continuità), trovare $x* \in R\,n$ tale che $x*$ = arg $min\ x{\in}Rn\ f(x)$. I punti di stazionarietà locale $x*$ sono soluzione del seguente sistema non lineare
$\nabla f(x*\,)$ = 0
il procedimento iterativo:

$$x_{k+1} - x_k = -H^{-1}(X_k)\nabla f(x_k)$$

Si osserva che $-H\,{-}1\ (Xk\,)\nabla f(xk)$ è la soluzione del sistema lineare

$$x_{k+1} = x_k + s_k$$

Implementazione:

```
def newtonRaphsonMinimo(grad_name ,Hess_name, x0, tolx, tolf, nmax):
   errors = []
   xk = np.array(x0)
   hk = Hess_name(xk)
   gk = grad_name(xk)
   s = np.linalg.solve(hk,gk)
   xk1 = xk+s
   it = 0

   while it < nmax and np.linalg.norm(xk1 - xk) / np.linalg.norm(xk) >=
tolx and np.linalg.norm(gk) >= tolf:
       xk = xk1
       hk = Hess_name(xk)
       gk = grad_name(xk)
```

```
        s = np.linalg.solve(hk,gk)
        xk1 = xk+s
        error = np.linalg.norm(xk1 - xk) / np.linalg.norm(xk)
        errors.append(error)
        it = it + 1

    return xk,it,errors
```

RISOLUZIONE DI SISTEMI LINEARI

Definizione:

A (nxn) è detta **NON singolare** se soddisfa una delle seguenti condizioni equivalenti:

1) $det(A) \neq 0$
2) Esiste la matrice inversa A-1 di A
3) rank(A)=n

Altrimenti A è singolare.

Teorema: *Condizione necessaria e sufficiente affinché il sistema lineare* $Ax=b$, $A \in M(n \times n)$, $x,b \in Rn$ *ammetta una ed una sola soluzione, comunque si scelga b, è che la matrice A sia a rango massimo (cioè che la matrice A sia invertibile); si ha perciò:*

$x = A-1b$

Se il sistema è omogeneo (**b=0**), ed A è non singolare allora esiste solo la soluzione nulla **x=0**.

La risoluzione di Ax = b con A ortogonale è sempre un problema ben condizionato

K indice di condizionamento

K(A) piccolo (ordine 10^p p=0,1,2,3): **il problema/matrice è ben condizionato**

K(A) grande (ordine > 10^3): **Il problema/matrice è mal condizionato**

METODI DIRETTI SISTEMI LINEARI

Essi sono adatti per la
soluzione di sistemi con **matrice dei coefficienti densa e di moderate dimensioni**.

Fattorizzazione LU (Gauss)

ridursi ad un sistema triangolare superiore con matrice dei coefficienti U, e termine noto trasformato y, e quindi costruire la matrice L, triangolare inferiore, memorizzando i moltiplicatori utilizzati per trasformare la matrice del sistema A in triangolare superiore.

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

Implementazione:

```
A=np.array([[2,1],[3,4]])
PT,L,U=lu(A) #Restituisce in output la trasposta
della matrice di Permutazione
P=PT.T.copy() #P è la matrice di permutazione
print("A=",A)
print("L=",L)
print("U=",U)
print("P=",P)
#LU è la fattorizzazione di P*A (terorema 2)
A1=P@A # equivale al prodotto matrice x matrice
np.dot(P,A)
```

```
A1Fatt=L@U # equivale a np.dot(L,U)
print("Matrice P*A \n", A1)
print("Matrice ottenuta moltipicando Le ed U
\n",A1Fatt)
```

Fattorizzazione Cholesky

Sia A una matrice di ordine n simmetrica e definita positiva, allora esiste una matrice
triangolare inferiore L con elementi diagonali positivi, (($lii > 0$ $i = 1,...n$) tale che
$A = L \cdot LT$
Sfruttando questo teorema, la soluzione del sistema lineare Ax=b, con A matrice
simmetrica e definita positiva, si riduce alla soluzione di due sistemi lineari

$$\begin{cases} Ly = b \\ L^T x = y \end{cases}$$

Implementazione:

```
L=cholesky(A,lower=True)
print(L)
A1=L@L.T
print("A1=\n",A1)
```

Fattorizzazione QR

Sia $A \in M(m \times n)$ con $m \geq n$ e rank(A) = n (ossia le colonne di A sono linearmente indipendenti). Allora esistono una matrice $Q \in M(m \times m)$ ortogonale e
una matrice $R = (R1\ 0\ ) \in M(m \times n)$ dove $R1 \in M(n \times n)$ è una matrice triangolare
superiore non singolare, tale che A = QR.

$$\begin{cases} Qy = b \\ Rx = y \end{cases}$$

Essendo Q ortogonale la soluzione del sistema Qy=b si riduce a $y = QT\ B$, (poiché l'inversa di una matrice ortogonale coincide con la sua inversa).

Matrice inversa con LU

La soluzione dell'i-esimo sistema lineare rappresenta l'i-esima colonna della matrice
Inversa.
Si può effettuare la fattorizzazione LU della matrice PA, che rimane la stessa per tutti
i sistemi lineari e poi utilizzarla per risolvere gli n sistemi lineari:

$$Ax_i = e_i \qquad \begin{cases} Ly_i = Pe_i \\ Ux_i = y_i \end{cases}$$

La soluzione *xi* dell'i-esimo sistema lineare così ottenuta rappresenta l'i-esima
colonna della matrice inversa
Implementazione:

```
def solvensis(A,B):
```

```
    PT,L,U = spl.lu(A)
    P = PT.T.copy()
    X = np.zeros((3,3))
    print(X)
    for i in range(0, A.shape[0]):
    ris = LUsolve(P, L, U, B[:,i])
    A[:,i] = ris[0].ravel()
    return A, P, U
```

METODI ITERATIVI SISTEMI LINEARI

Una famiglia di metodi iterativi per la soluzione del sistema lineare (1) si ottiene
utilizzando una **decomposizione della matrice** ***A*** nella forma
$A = M - N$ con $\det M \neq 0$.
In tal modo il sistema (1) diventa
$(M-N)\,x = b,$
cioè
$Mx = Nx + b$
e quindi si ottiene
$x = M^{-1}Nx + M^{-1}b$

Metodo di Jacobi

Nel metodo di Jacobi la decomposizione di A nella forma A=M-N si ottiene
scegliendo **M=D** e **N= - (E+F)**.
Pertanto il procedimento iterativo (2) diviene

$$x^{(k)} = -D^{-1}(E+F)\,x^{(k-1)} + D^{-1}b \qquad \text{per } k=1,2,..$$

In termini di componenti la (3), equivale a calcolare la i-esima componente
dell'iterato k-esimo come

$$x_i^{(k)} = \frac{\sum_{\substack{j=1 \\ j\neq i}}^{n} -a_{ij}x_j^{(k-1)} + b_i}{a_{ii}} \qquad i = 1,2..,n,$$

La matrice di iterazione del metodo di Jacobi è quindi data da

$$T_J = M^{-1}N = -D^{-1}(E+F)$$

Implementazione:

```
def jacobi(A,b,x0,toll,it_max):
    errore=1000
    d=np.diag(A)
    n=A.shape[0]
    invM=np.diag(1/d)
    E=np.tril(A,-1)
    F=np.triu(A,1)
    N=-(E+F)
    T=np.dot(invM,N)
    autovalori=np.linalg.eigvals(T)
    raggiospettrale=np.max(np.abs(autovalori))
    print("raggio spettrale jacobi", raggiospettrale)
    it=0

    er_vet=[]
    while it<=it_max and errore>=toll:
        x=(b+np.dot(N,x0))/d.reshape(n,1)
        errore=np.linalg.norm(x-x0)/np.linalg.norm(x)
        er_vet.append(errore)
        x0=x.copy()
        it=it+1
    return x,it,er_vet
```

#metodo di arresto:

$$\frac{||x^{(k)}-x^{(k-1)}||}{||x^{(k)}||} \leq \varepsilon.$$

Metodo di Gauss Siedel

Nel metodo di Gauss-Seidel la decomposizione di A nella forma $A=M-N$ si ottiene
scegliendo $M=E+D$ e $N=-F$. In questo caso la matrice di iterazione del metodo di

Gauss Seidel è data da ***TG = M-1N = -(E+D)-1F e la soluzione al passo k si ottiene come***

$$x^{(k)} = -(E+D)^{-1}F\,x^{(k-1)} + (E+D)^{-1}b \qquad \text{per } k=1,2,..$$

attraverso passaggi algebrici:

$$x^{(k)} = -\,D^{-1}\,(E\,x^{(k)} + F\,x^{(k-1)} - b\,) \qquad \text{per } k=1,2,..$$

Implementazione Gauss Siedel:

```
def gauss_seidel(A,b,x0,toll,it_max):
    errore=1000
    d=np.diag(A)
    D=np.diag(d)
    E=np.tril(A,-1)
    F=np.triu(A,1)
    M=D+E
    N=-F
    T=np.dot(np.linalg.inv(M),N)
    autovalori=np.linalg.eigvals(T)
    raggiospettrale=np.max(np.abs(autovalori))
    print("raggio spettrale Gauss-Seidel ",raggiospettrale)
    it=0
    er_vet=[]
```

```
    while it<=it_max and errore>=toll:
        temp=b-F@x0
        x,flag=Lsolve(M,temp)  #Calcolare la soluzione al passo k
equivale a calcolare la soluzione del sistema triangolare con matrice
M=D+E
                               # e termine noto b-F@x0
        errore=np.linalg.norm(x-x0)/np.linalg.norm(x)
        er_vet.append(errore)
        x0=x.copy()
        it=it+1
    return x,it,er_vet
```

Implementazione Gauss Siedel SOR:

```
def gauss_seidel_sor(A,b,x0,toll,it_max,omega):
    errore=1000
    d=np.diag(A)
    D=np.diag(d)
    Dinv=np.diag(1/d)
    E=np.tril(A,-1)
    F=np.triu(A,1)
    Momega=D+omega*E
    Nomega=(1-omega)*D-omega*F
    T=np.dot(np.linalg.inv(Momega),Nomega)
    autovalori=np.linalg.eigvals(T)
    raggiospettrale=np.max(np.abs(autovalori))
    print("raggio spettrale Gauss-Seidel SOR ", raggiospettrale)

    M=D+E
    N=-F
    it=0
    xold=x0.copy()
    xnew=x0.copy()
    er_vet=[]
    while it<=it_max and errore>=toll:
        temp=b-np.dot(F,xold)
        xtilde,flag=Lsolve(M,temp)
        xnew=(1-omega)*xold+omega*xtilde
        errore=np.linalg.norm(xnew-xold)/np.linalg.norm(xnew)
        er_vet.append(errore)
        xold=xnew.copy()
        it=it+1
```

```
    return xnew,it,er_vet
```

Teorema: condizione convergenza necessaria sufficente

Sia A=M-N una matrice di ordine n, con det(A) 0, e T=M -1N la matrice di iterazione del procedimento iterativo (2).
Condizione necessaria e sufficiente per la convergenza del procedimento iterativo, comunque si scelga il vettore iniziale x(0), al vettore soluzione x del sistema Ax=b , è che $\varrho(T) < 1$ *ovvero che il raggio spettrale (che corrisponde all'autovalore di modulo massimo) della matrice di iterazione T sia minore di 1.*

Teorema: condizione convergenza sufficiente

Calcolare il raggio spettrale su grosse matrici risulta oneroso, si ricorre perciò al seguente teorema:

Se, per una qualche norma, risulta $||T||<1$, *allora il processo iterativo*

$$x^{(k)} = Tx^{(k-1)} + M^{-1}b \qquad per\ k=1,2,....$$

è convergente per ogni x(0)

.

METODI DISCESA SISTEMI LINEARI

Nota: Ricordiamo che dati due vettori colonna *x,y* ∈ *Rn con la notazione <x,y> si intende il prodotto scalare x^Ty.*

Teorema 1:

*Sia A Rnxn , **matrice simmetrica e definita positiva**, b,x ∈ Rn, allora la soluzione del sistema lineare*

Ax = b (1)

coincide con il punto di minimo della seguente funzione quadratica

$$F(x) = \frac{1}{2} < Ax, x > - < b, x > = \frac{1}{2} x^T Ax - b^T x = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j - \sum_{i=1}^{n} b_i x_i$$

In generale ad ogni passo iterativo

$$\nabla F = Ax\text{-}b$$

Ponendo l'anti gradiente uguale ad r, ovvero la direzione in cui muoversi per trovare il minimo si effettua la scelta dello STEP SIZE.

determinarne il valore di $\alpha(k)$ che rende minima F nella direzione *p(k)*, cioè:

$$arg \min_{\alpha^{(k)}} F(x^{(k)} + \alpha^{(k)} p^{(k)})$$

Si arriva a determinare il valore

$$\alpha^{(k)} = - \frac{<r^{(k)}, p^{(k)}>}{<Ap^{(k)}, p^{(k)}>} = - \frac{(r^{(k)})^T p^{(k)}}{(p^{(k)})^T Ap^{(k)}}$$

Questo significa che, una volta determinata la direzione p(k), ovvero scelto il metodo di

minimizzazione, la relazione ci fornisce il valore da assegnare allo stepsize $\alpha(k)$ per ottenere il minimo valore possibile della F lungo la direzione p(k).

Teorema:
Nel punto di minimo $x(k+1) = x(k) + \alpha(k)p(k)$*, ottenuto muovendosi lungo la*
direzione $p(k)$ *con* $\alpha(k)$*dato dalla (3), il vettore residuo* $r(k+1) = Ax(k+1) - b$ *risulta ortogonale alla direzione* $p(k)$ *, cioè*

$$< r^{(k+1)}, p^{(k)} >= 0$$

Steep Descent

Il metodo di Discesa più Ripida (Steepest Descent) è caratterizzato dalla scelta,
ad ogni passo k, della direzione p(k) come l'antigradiente della F calcolato nell'iterato k -esimo, ovvero

$$p^{(k)} = -\nabla F(x^{(k)}) = -Ax^{(k)} + b = -r^{(k)}.$$

Si ricorda che *p* è la direzione di discesa ed *r* è il residuo.

Algoritmo steepest descent (Algoritmo del gradiente)

1. Parti con qualche $x^{(0)}$, $k = 0$
2. Calcola la direzione di discesa più ripida

$$p^{(k)} = -\nabla f(x^{(k)})$$

3. Scegli lo stepsize $\alpha^{(k)}$ tale che

$$F\left(x^k + \alpha^{(k)}\, p^{(k)}\right) < F\left(x^{(k)}\right)$$

$$\alpha^{(k)} = \frac{< r^{(k)}, r^{(k)} >}{< Ar^{(k)}, r^{(k)} >}$$

4. Aggiorna l'iterato

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$$

$$r^{(k+1)} = r^{(k)} + \alpha^{(k)} A\, p^{(k)}$$

5. Incrementa il contatore k=k+1

Fino a convergenza

Si considera che il procedimento iterativo ha raggiunto la convergenza quando
$||r(k+1)|| \leq$ *tolleranza*

Implementazione:

```
def steepestdescent(A,b,x0,itmax,tol):

    n,m=A.shape
    if n!=m:
        print("Matrice non quadrata")
        return [],[]


   # inizializzare le variabili necessarie
    x = x0


    r = A@x-b
    p = -r
    it = 0
    nb=np.linalg.norm(b)
    errore=np.linalg.norm(r)/nb
```

```
    vec_sol=[]
    vec_sol.append(x)
    vet_r=[]
    vet_r.append(errore)

# utilizzare il metodo del gradiente per trovare la soluzione
    while errore>= tol and it< itmax:
        it=it+1
        Ap=A@p

        alpha = -(r.T@p)/(p.T@Ap)

        x = x + alpha*p  #aggiornamento della soluzione nella direzione
opposta a quella del gradiente: alpha mi dice dove fermarmi
        #nella direzione del gradiente affinche F(xk+t p ) <F(xk)



        vec_sol.append(x)
        r=r+alpha*Ap
        errore=np.linalg.norm(r)/nb
        vet_r.append(errore)
        p = -r #Direzione opposta alla direzione del gradiente


    return x,vet_r,vec_sol,it
```

Metodo Gradiente Coniugato

In questo metodo la scelta della direzione di discesa p(k) tiene conto non solo del
gradiente della F(x(k)) cioè di r(k) , ma anche della direzione di discesa dell'iterazione
precedente p(k-1).

$$p^{(k)} = -r^{(k)} + \gamma_k p^{(k-1)}, \quad k = 1,2,....$$

Poiché il punto di minimo nel piano k coincide con il centro dell'ellisse, il parametro
k sarà scelto in modo che la direzione p(k) punti verso il centro dell'ellisse, cioè sia il
coniugato, rispetto all'ellisse, di p(k-1)

$$\gamma_k = \frac{<r^{(k)}, Ap^{(k-1)}>}{<p^{(k-1)}, Ap^{(k-1)}>}.$$

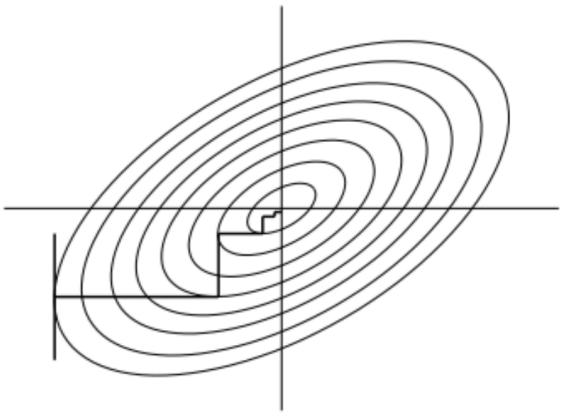
$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$$

$$\alpha_k = -\frac{<r^{(k)}, p^{(k)}>}{<Ap^{(k)}, p^{(k)})} > \qquad k = 1,2,\ldots .$$

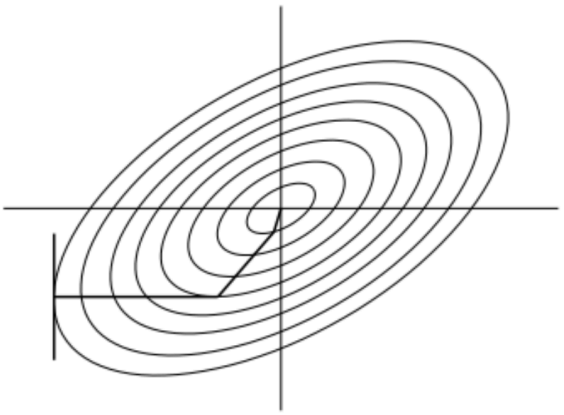
Teorema:
Nel metodo del ***gradiente coniugato*** ***le direzioni di discesa p(k) , con k=0,1,.., formano un sistema di direzioni coniugate****, mentre i* ***vettori residui r(k) , con k=0,1,.., formano un sistema ortogonale***

Implementazione:

```
def conjugate_gradient(A,b,x0,itmax,tol):
    n,m=A.shape
    if n!=m:
        print("Matrice non quadrata")
        return [],[]


   # inizializzare le variabili necessarie
    x = x0

    r = A.dot(x)-b
    p = -r
    it = 0
    nb=np.linalg.norm(b)
    errore=np.linalg.norm(r)/nb
    vec_sol=[]
    vec_sol.append(x0)
    vet_r=[]
    vet_r.append(errore)
# utilizzare il metodo del gradiente coniugato per calcolare la
soluzione
    while errore >= tol and it< itmax:
        it=it+1
        Ap=A.dot(p)
        alpha = -(r.T@p)/(p.T@Ap)
        x = x + alpha *p
        vec_sol.append(x)
        rtr_old=r.T@r
        r=r+alpha*Ap
        gamma=r.T@r/rtr_old
        errore=np.linalg.norm(r)/nb
        vet_r.append(errore)
        p = -r+gamma*p  #La nuova direzione appartiene al piano
individuato da -r e p. gamma è scelto in maniera tale che la nuova
direzione
        #sia coniugata rispetto alla direzione precedente( che
geometricamente significa che punti verso il centro)


    return x,vet_r,vec_sol,it
```



Steepest descent



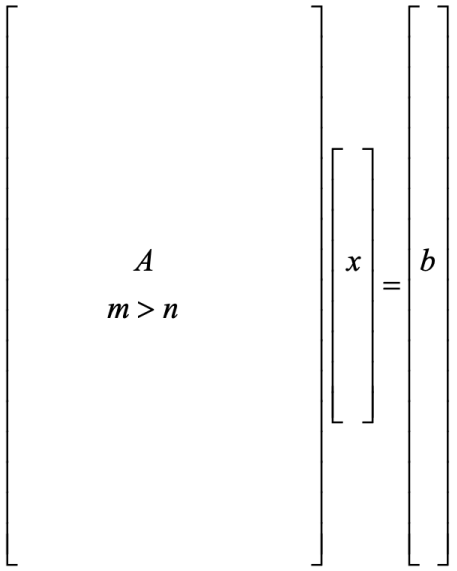
Gradiente Coniugato

SISTEMI SOVRADETERMINATI

I sistemi lineari sovradeterminati hanno un numero di equazioni superiore al numero di incognite, ossia sono della forma

$$Ax = b$$

dove $A \in R^{m \times n}$, $x \in R^n$ e $b \in R^m$, $m > n$



se $rank(A) \neq rank(A|b)$ → sistema incompatibile nessuna soluzione
se $rank(A) = rank(A|b)$ → sistema compatibile

La risoluzione di un sistema lineare sovradeterminato risulta, quindi, essere un
problema mal posto in quanto potrebbe accadere che la soluzione non esista o, se esiste, non sia unica.
Si tende quindi a far diventare A una matrice quadrata.

Last squares / Equazioni normali

Problema ben posto:
Cerchiamo la soluzione del sistema lineare sovradeterminato
$Ax=b$
dove $A \in Rm \times n$, $x \in Rn$ e $b \in Rm$, $m > n$

cerchiamo il vettore $x* \in Rn$ che rende minima la norma 2 al quadrato del residuo, cioè

$$x^* = \arg\min_{x \in R^n} ||r(x)||_2^2 = \arg\min_{x \in R^n} ||Ax - b||_2^2$$

$$F(x) = ||Ax - b||_2^2 = (Ax - b)^T (Ax - b) = x^T A^T Ax - x^T A^T b - b^T A\, x + b^T b = x^T A^T Ax - 2x^T A^T b + b^T b$$

Poniamo $G = A^T A$. Si ha che G è simmetrica, infatti $G^T = (A^T A)^T = A^T A = G$.

$$F(x) = x^T G\, x - 2x^T A^T b + b^T b$$

Per calcolare il valore $x* \in Rn$ che rende minimo $F(x)$, calcoliamo il gradiente della funzione $F(x)$ ed imponiamo che si annulli.
Ricordiamo che il gradiente rispetto ad x della forma quadratica $xTG\,x$, con G simmetrica, è uguale a $2\,G\,x$.
Allora si ha:

$$\nabla F(x) = 2\,G\,x - 2A^T b = 0$$

Il vettore $x* \in Rn$ che annulla il gradiente della funzione $F(x)$ è la soluzione del sistema lineare

$$\boldsymbol{G\,x = A^T b}$$

Ovviamente affinché il sistema quadrato di dimensione $n \times n$ sia risolubile è necessario che il determinante della matrice A.TA sia diverso da zero, e quindi è necessario che la matrice A abbia rango n .

Implementazione:

```
def eqnorm(A,b):
```

```
    # G@x = A.T@b
    G = A.T @ A
    b1 = A.T @  b
    #Risolvo Gx = b1 nuovo_termine con Cholesky
    L = spl.cholesky(G, lower=True)
    y,flag = SolveTriangular.Lsolve(L,b1)
    #risolvo Ux = y
    U = L.T
    x, flag = SolveTriangular.Usolve(U,y)
    return x
```

Inoltre, affinché la soluzione cercata sia proprio un minimo è necessario che la matrice Hessiana valutata nella soluzione sia una matrice definita positiva.
Poichè $\nabla 2F(x)=2\ G=2\ ATA$ occorre che la matrice $G= ATA$, che abbiamo visto essere simmetrica, sia definita positiva.

Verifico anche che sia definita positiva nel seguente modo:

$$\| Ax \|_2^2 > 0 \quad \forall\ Ax \neq 0.$$

TEOREMA
Dato il sistema lineare sovradeterminato
$Ax=b$

$$x^* = \arg\min_{x \in R^n} \ ||Ax - b||_2^2 \ \Leftrightarrow \text{ è la soluzione di } \ A^T Ax = A^T b$$

può essere sempre calcolato, teoricamente, come soluzione delle equazioni normali
$A^T Ax=A^T b$.
La matrice G = A^T A è simmetrica e definita positiva e quindi il sistema può essere risolto utilizzando il metodo di Cholesky.
ATTENZIONE AL MALCONDIZIONAMENTO
Se quindi **A è ben condizionata** il metodo delle equazioni normali è un ottimo metodo per calcolare la soluzione del problema dei minimi

quadrati. Se A è 'mediamente mal condizionata' il metodo delle equazioni normali è ***numericamente pericoloso*** perché può fornire risultati non affidabili.

Proprietà: Una matrice ortogonale $Q \in Rm \times m$ applicata ad un vettore $y \in Rm$, ne mantiene inalterata la norma 2 al quadrato

$$||y||_2^2 = ||Qy||_2^2$$

QRLS Minimi Quadrati

Una volta calcolata la fattorizzazione QR di A, poiché Q è ortogonale e quindi anche QT è ortogonale, consideriamo

$$\|Q^T(Ax-b)\|_2^2 = \|Q^TAx - Q^Tb\|_2^2 = \|R_1x - Q^Tb\|_2^2.$$

Posto

$$h = Q^Tb = \begin{bmatrix} h_1 \\ \cdots \\ h_2 \end{bmatrix} \begin{matrix} \}n \\ \\ \}m-n \end{matrix}$$

si ha

$$\left\| \overbrace{\begin{bmatrix} \ddots & & R_1 \\ & \ddots & \\ 0 & & \ddots \\ \cdots & \cdots & \cdots \\ & 0 & \end{bmatrix}}^{n} x - \begin{bmatrix} h_1 \\ \\ \cdots \\ h_2 \end{bmatrix} \begin{matrix} \}n \\ \\ m-n \end{matrix} \right\|_2^2 = \left\| \begin{matrix} (R_1x - h_1) \\ \\ -h_2 \end{matrix} \right\|_2^2$$

$$\arg\min_{x \in R^n} \|Ax-b\|_2^2 = \arg\min_{x \in R^n} \{\|R_1x - h_1\|_2^2 + \|h_2\|_2^2\}$$

$$= \arg\min_{x \in R^n} \|R_1x - h_1\|_2^2 + \|h_2\|_2^2.$$

Quindi il minimo sarà ottenuto per x* che risolve il sistema lineare

$$R_1x = h_1$$

x è quindi il minimo del sistema lineare sovradeterminato
Questo elemento sfruttando l'ortogonalità risulta efficiente anche su una matrice mal condizionata.

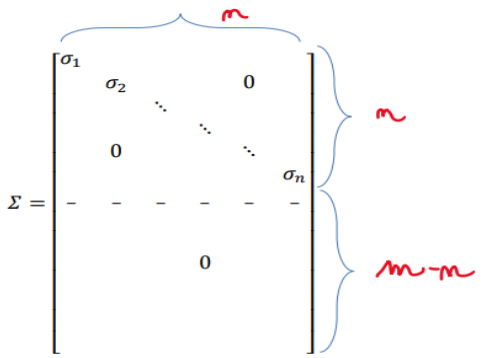
Implementazione:

```
def qrLS(A,b,n):
    #A rango massimo
    Q,R = spl.qr(A)
    R1 = R[0:n,0:n]
    #risolvo il sistema R1x=h1 dove h1 = Q.T b
    h = Q.T@b
    x = spl.solve(R1,h[0:n])
    return x
```

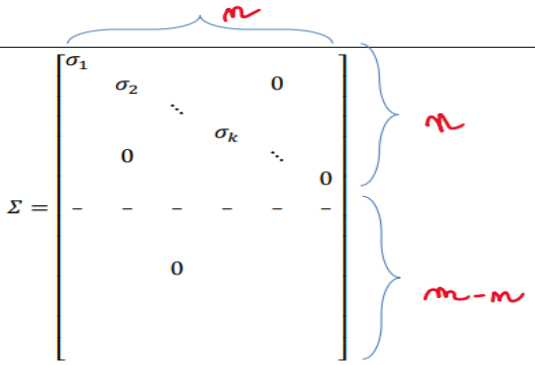
SVD: decomposizione ai valori singolari

Sia A R mxn a rango $k \leq min(m, n)$. Allora esistono due matrici ortogonali U mxm e V nxn tali che $U.TAV = \Sigma = diag(\sigma 1, \sigma 2, \cdots, \sigma k, 0, \ldots, 0)$

Se A ha rango k=n



Se A ha rango k<n



$A = U\Sigma V^T$

$$c = V^T x \quad \text{e} \quad d = U^T b = \begin{bmatrix} u_1{}^T b \\ u_2{}^T b \\ u_n{}^T b \\ \\ u_m{}^T b \end{bmatrix} = \begin{bmatrix} d_1 \\ \\ d_2 \end{bmatrix} \begin{matrix} \} k \\ \\ \} m-k \end{matrix}$$

$$\arg\min_{x \in R^n} ||Ax - b||_2^2 = \arg\min_{c \in R^n} ||\Sigma c - d||_2^2 = \arg\min_{c \in R^n} | \, |\Sigma c - d_1||_2^2 + ||d_2||_2^2$$

In realtà quindi per trovare sigma mi basta trovare la matrice diagonale che ha come valori ($\sigma 1, \sigma 2, \cdots, \sigma k$) e risolvere il sistema associato `Σc = d dove c=V.T x e d = U.T b`

Implementazione:

```
"""
npl.svd spiegazione:
    Una matrice unitaria U: una matrice di dimensioni (m, m) se
full_matrices è True, altrimenti di dimensioni (m, min(m, n)).
    Un array contenente i valori singolari s, ordinati in ordine
decrescente.
    La trasposizione coniugata della matrice unitaria V: una matrice di
dimensioni (n, n) se full_matrices è True, altrimenti di dimensioni
(min(m, n), n).
    U e V ortogonali
"""

def svdLS(A,b):
    # A senza rango massimo => esistono U mxm V nxn ortogonale
    # Risolvo il sistema Σc = d dove c=V.T x e d = U.T b
    U, sig, VT = np.linalg.svd(A, full_matrices=False)

    # Calcola l'inversa del reciproco della matrice diagonale s
    SIG = np.diag(1/sig)

    # Calcola la soluzione x tramite la formula pseudoinversa
    x = VT.T @ SIG @ U.T @ b

    # Calcola il residuo
    residual = np.linalg.norm(b - A @ x)**2

    return x, residual
```

INTERPOLAZIONE POLINOMIALE

Definiamo $\mathbb{P}n[x]$ lo spazio vettoriale dei polinomi nella variabile x di grado minore o uguale ad n e a coefficienti reali.
$\mathbb{P}n[x]:= \{\alpha 0 + \alpha 1x + \alpha 2x2 + \cdots \alpha nxn \mid \alpha i \in R\}$
Imporre $Pn(xi)= yi$, i=0,..n, significa
$Pn(xi) = \alpha 0 + \alpha 1xi + \alpha 2xi\text{^}2+...+\alpha nxi\text{^}n = yi$ per ogni i=0,…,n

Se scriviamo questa relazione per ogni *i*, ricaveremo il seguente sistema lineare.

$$\begin{cases} \alpha_0 + \alpha_1 x_0 + \alpha_2 x_0^2 + \ldots + \alpha_n x_0^n = y_0 \\ \alpha_0 + \alpha_1 x_1 + \alpha_2 x_1^2 + \ldots + \alpha_n x_1^n = y_1 \\ \ldots \\ \alpha_0 + \alpha_1 x_n + \alpha_2 x_n^2 + \ldots + \alpha_n x_n^n = y_n \end{cases}$$

In termini matriciali, questo sistema lineare si può scrivere nel seguente modo: A =y (1)
dove la matrice dei coefficienti A di questo sistema lineare è la matrice di dimensione (*n+1*) (*n+1*)

$$A = \begin{bmatrix} 1 & x_0^1 & x_0^2 & \ldots & x_0^n \\ 1 & x_1^1 & x_1^2 & \ldots & x_1^n \\ 1 & x_2^1 & x_2^2 & \ldots & x_2^n \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_n^1 & x_n^2 & \ldots & x_n^n \end{bmatrix}$$

Riassumendo:

1) trovo la matrice V di Vandermond che ha i valori assunti da x1,....,xn
2) risolvo il sistema Vα=y dove y è un vettore delle ascisse y1,...,yn
3) α è il vettore che racchiude i coefficenti del mio polinomio α1 + α2x2 + … + αnxn

Implementazione:

```
x = np.array([-3.5, -3, -2, -1.5, -0.5, 0.5, 1.7, 2.5, 3])
y = np.array([-3.9, -4.8, -3.3, -2.5, 0.3, 1.8, 4, 6.9, 7.1])

m = x.shape
n = 1 #grado del polinomio di regressione
n1 = n + 1 #gradi di libertà o n umero dei coefficenti del
polinomio
B = np.vander(x, increasing=True)[:,:n1]

a_EQN = eqnorm(B,y)

a_QR = qrLS(B,y,n1)

a_SV = svdLS(B,y)


xv = np.linspace(np.min(x), np.max(x),200)
pol_EQN = np.polyval(np.flip(a_EQN),xv)
plt.plot(x,y,'*',xv,pol_EQN)
```

Interpolazione di Lagrange

La matrice dei coefficienti di Vandermonde, è una matrice molto mal condizionata, bisogna cambiare base per lo spazio $\mathbb{P}n[x]$.
Quindi il polinomio $P\,n(x) \in \mathbb{P}n[x]$ nella base di Lagrange si esprime nella forma

$$P_n(x) = \alpha_0\, L_0^{(n)}(x) + \alpha_1\, L_1^{(n)}(x) + \alpha_2\, L_2^{(n)}(x) + \ldots + \alpha_n\, L_n^{(n)}(x)$$

che scritto in termini matriciali diventa:

$$\begin{bmatrix} L_0^{(n)}(x_0) & L_1^{(n)}(x_0) & L_2^{(n)}(x_0) & \dots & L_n^{(n)}(x_0) \\ L_0^{(n)}(x_1) & L_1^{(n)}(x_1) & L_2^{(n)}(x_1) & \dots & L_n^{(n)}(x_1) \\ \vdots & \vdots & \vdots & \dots & \vdots \\ \dots & \dots & \dots & \dots & \dots \\ L_0^{(n)}(x_n) & L_1^{(n)}(x_n) & L_2^{(n)}(x_n) & \dots & L_n^{(n)}(x_n) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \dots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}$$

I polinomi L(x) sono detti polinomi base di Lagrange

Riassumendo:

1) Trovo $L(x)_i$ per i da 0 a n.
2) $L(x)_i$ è trovato tramite numeratore uguale alla produttoria di $(x - x_J)$ con j compreso fra 0 ed n, $J \neq i$
3) $L(x)_i$ è trovato tramite denominatore uguale alla produttoria di $(x_i - x_J)$ con j compreso fra 0 ed n, $J \neq i$
4) $L(x)_i$ = numeratore ÷ denominatore
5) vado a sostituire i polinomi di Lagrange trovati nel seguente polinomio: $P(x)_n = \alpha_0 + \alpha_1 L(x)_1 + \dots + \alpha_n L(x)_n$

Es:

(-1,2), (1,1), (2,1)

Dobbiamo quindi individuare il polinomio di grado 2 che interpola i dati assegnati:

$$P_2(x) = \sum_{j=0}^{2} y_j L_j^{(2)}(x)$$

$$P_2(x) = y_0 L_0^{(2)}(x) + y_1 L_1^{(2)}(x) + y_2 L_2^{(2)}(x) \qquad (3)$$

$$L_0^{(2)}(x) = \frac{(x-x_1).(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{(x-1).(x-2)}{(-1-1)(-1-2)} = \frac{1}{6}(x-1)\cdot(x-2)$$

$$L_1^{(2)}(x) = \frac{(x-x_0).(x-x_2)}{(x_1-x_0)(x_1-x_2)} = \frac{(x+1).(x-1)}{(1+1)(1-2)} = -\frac{1}{2}(x+1)\cdot(x-2)$$

$$L_2^{(2)}(x) = \frac{(x-x_0).(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{(x+1).(x-1)}{(2+1)(2-1)} = \frac{1}{3}(x+1)\cdot(x-1)$$

Sostituendo in (3) queste espressioni per gli $L_j^{(2)}(x)$ j=0,1,2 e y_0= 2 y_1=1 y_2=1, si ha

$$P_2(x) = 2\cdot\frac{1}{6}(x-1)(x-2) + 1\cdot\left(-\frac{1}{2}(x+1)(x-2)\right) + 1\left(\frac{1}{3}(x+1)(x-1)\right)$$

Effettuando le opportune moltiplicazioni e raccogliendo si ottiene

$$P_2(x) = \frac{4}{3} - \frac{1}{2}x + \frac{1}{6}x^2$$

Implementazione:

```
def plagr(xnodi,j):
   """
   Restituisce i coefficienti del k-esimo pol di
   Lagrange associato ai punti del vettore xnodi
   """
   xzeri=np.zeros_like(xnodi)
   n=xnodi.size
   if j==0:
      xzeri=xnodi[1:n]
   else:
      xzeri=np.append(xnodi[0:j],xnodi[j+1:n])

   num=np.poly(xzeri)
   den=np.polyval(num,xnodi[j])

   p=num/den

   return p


def InterpL(x, y, xx):
    """"
```

```
       %funzione che determina in un insieme di punti il valore del
polinomio
       %interpolante ottenuto dalla formula di Lagrange.
       % DATI INPUT
       %  x  vettore con i nodi dell'interpolazione
       %  f  vettore con i valori dei nodi
       %  xx vettore con i punti in cui si vuole calcolare il polinomio
       % DATI OUTPUT
       %  y vettore contenente i valori assunti dal polinomio
interpolante
       %
    """
    n=x.size
    m=xx.size
    L=np.zeros((m,n))
    for j in range(n):
       p=plagr(x,j)
       L[:,j]=np.polyval(p,xx)


    return L@y
```

Errore di Interpolazione

Stimiamo adesso l'errore che si compie quando al posto della funzione che ha
generato i dati si sostituisce la formula del polinomio interpolatore.
$E(\bar{x}) = f(\bar{x}) - P\,n(\bar{x})$.

Risulta allora che

$$E(\bar{x}) = f(\bar{x}) - P_n(\bar{x}) = \frac{1}{(n+1)!}\omega_{n+1}(\bar{x})f^{(n+1)}(\xi)$$

dove $\zeta \in (a,b)$ *e* $\omega_{n+1}(\bar{x}) = (\bar{x} - x_0)(\bar{x} - x_1)\dots(\bar{x} - x_n)$

Se $xi=\bar{x}$ allora l'errore è nullo perchè si annulla il fattore
$\omega n+1(xi) = (xi-x0)(xi- x1)..,(xi- xi).....(xi- xn) =0$.

Inoltre, l’errore risulta nullo anche nel caso di dati provenienti da funzioni che hanno
la derivata n+1 nulla, cioè per funzioni che sono polinomi di grado n.

Condizionamento del problema di Interpolazione

Siano date le coppie $(xi,yi),i = 0,..,n$ con xi,i = 0,..n appartenenti all’intervallo[a,b].
Consideriamo le perturbazioni sui dati $\tilde{yi} = yi + \epsilon i$,$i = 0,\ldots,n$ e l’errore relativo sui dati:

$$\frac{\|\tilde{y} - y\|_\infty}{||y||_\infty}$$

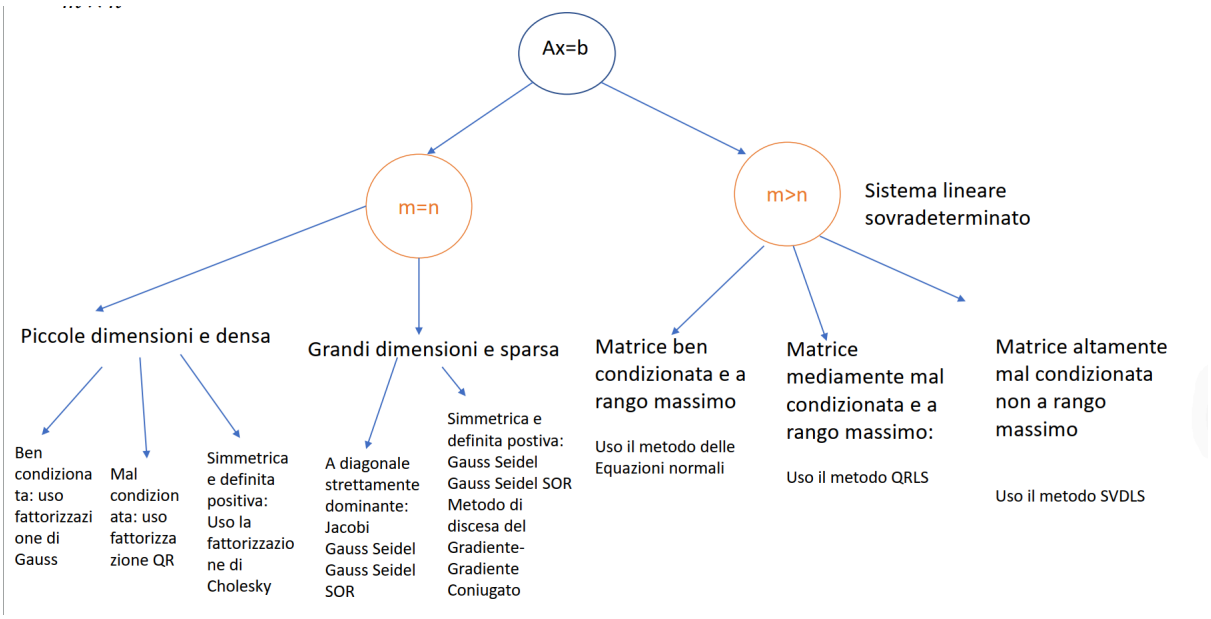
TABELLA DI UTILIZZO ALGORITMI

Nome	Finalità	Caso d'uso	Errore relativo
Metodo bisezione	zeri di funzione su equazioni non lineari	converge globalmente alla soluzione con la sola ipotesi che f sia continua nell'intervallo [a;b], è molto lento.	| iterati - α | lab: α = 0
Regula Falsi	zeri di funzione su equazioni non lineari	convergenza globale. E' più veloce rispetto al metodo di bisezione (convergenza superlineare).	| iterati - α | lab: α = 0
Metodo Corde	zeri di funzione su equazioni non lineari	non ho accesso alla derivata della f.	| iterati - α | lab: α = 0
Metodo Secanti	zeri di funzione su equazioni non lineari	più veloce ma non converge sempre.	| iterati - α | lab: α = 0
Metodo Newton	zeri di funzione su equazioni non lineari	convergenza di ordine quadratico e globale.	| iterati - α | lab: α = 0
Newton Raphson	Soluzione di sistemi non lineari	Convergenza locale e di ordine quadratico. Uso per minimo del sistema / soluzione	|| xk1 - xk || / || xk || lab: xk1 è l'iterato successivo.
Fattorizzazione LU Gauss	Soluzione di sistemi lineari (diretto)	Convergenza lenta, matrice con det ≠ 0. Matrice ben condizionata	|| x - xlu || / || x ||
Fattorizzazione Cholesky	Soluzione di sistemi lineari (diretto)	matrice di ordine n simmetrica e definita positiva. Più veloce di LU	|| x - xch || / || x ||
Fattorizzazione QR	Soluzione di sistemi lineari (diretto)	matrice quadrata, mal condizionata	|| x - xqr || / || x ||

Metodo Jacobi	Soluzione di sistemi lineari (iterativo)	più veloce del metodo diretto. det ≠ 0 e elementi della diagonale della matrice ≠ 0	|| x - x0 || / || x || lab: x0 è il primo iterato
Metodo Gauss Siedel	Soluzione di sistemi lineari (iterativo)	più veloce del metodo diretto. Più veloce di Jacobi	|| x - x0 || / || x ||
Steep Descent	Minimo di funzioni lineari	più semplice da implementare e richiedere meno memoria. È utile quando non si può garantire la convessità del problema di ottimizzazione.	|| r || / || b ||
Metodo Gradiente Congiunto	Minimo di funzioni lineari	Più veloce dello steep descent. È efficace quando la matrice del problema è simmetrica e definita positiva.	|| r || / || b ||
Equazioni Normali / Last Squares	Soluzione di sistemi sovradeterminati	Matrice ben condizionata	
QRLS Minimi Quadrati	Soluzione di sistemi sovradeterminati	Matrice mal condizionata	
SVD Decomposizione ai minimi valori	Soluzione di sistemi sovradeterminati	Matrice mxn a rango $k \leq min(m, n)$	
Interpolazione di Lagrange	Approssimazione a polinomio di una funzione		|| y+α - y ||$_\infty$ / || y ||$_\infty$

ANALISI DI MATRICE

```
n,m = A.shape
nz = np.count_nonzero(A)/(n*m)
perc_nz = nz * 100
K = npl.cond(A)
if np.all(A == A.T) == 0:
    print("Matrice non simmetrica")
else:
    print("Matrice è simmetrica")
    autovalori = npl.eigvals(A)
    flag_dp = np.all(autovalori>0)
    print("Matrice definita positiva", flag_dp)

if K < 10**3:
    print("matrice ben condizionata")
else:
    print("Matrice mal condizionata")

if perc_nz > 66:
    print("Matrice sparsa")
else:
    print("Matrice non sparsa")
plt.spy(A)
```



TEOREMI

Teorema degli zeri di funzioni continue

Sia f(x) continua nell'intervallo [a, b] e sia tale che f(a)·f(b)<0, allora f ammette almeno uno zero in (a,b), cioè esiste almeno un punto α in (a,b) tale che f(α)=0.

Teorema di convergenza locale

Se $f : [a: b] \rightarrow R$ soddisfa le seguenti ipotesi
i) $f(a)\, f(b) < 0$
ii) f, f', f'' *sono continue in* $[a; b]$, *ossia* $f \in C\,2\,[a; b]$
iii) $f'(x) \neq 0 \;\forall\; x \in [a, b]$
allora esiste un intorno $I \subset [a, b]$ dell'unica radice $\alpha \in (a, b)$ tale che, se $x \in I$, allora la successione di Newton $\{xi\}i\geq 1$ converge ad α.

Teorema di convergenza globale

Sia $f(x) \in C\,2\,[a, b]$, [a,b] intervallo chiuso e limitato. Se sono verificate le seguenti condizioni

1. $f(a)f(b)<0$
2. $f'(x)\neq 0 \quad \forall x\in[a,b]$
3. $f''(x)>0 \quad$ *oppure* $\quad f''(x)<0 \quad \forall x\in[a,b]$
4. $\left|\frac{f(a)}{f'(a)}\right|<b-a \quad \left|\frac{f(b)}{f'(b)}\right|<b-a$

allora il metodo di Newton converge all'unica soluzione α in [a,b], per ogni scelta di x0 in [a,b].

Teorema f convessa e differenziabile

Se la funzione f è convessa e differenziabile, ($x0$, $y0$) è un minimo $\Leftrightarrow \nabla f(x0, y0) = 0$,

Teorema di Rouchè-Capelli

I sistema lineare Ax=b ammette soluzioni se e solo se la matrice dei coefficienti A e la matrice completa [A b] hanno lo stesso rango: Rank(A)=r=Rank([A b]) Altrimenti se (Rank(A)rank([A b])) il sistema non ammette soluzioni

Teorema esistenza fattorizzazione LU

Data $A \in M(nxn)$, sia Ak la sottomatrice principale di testa di A ottenuta considerando le prime k righe e le prime k colonne di A. Se Ak è non singolare per ogni k=1,..,n-1 allora esiste ed è unica la fattorizzazione LU di A

Teorema matrice permutazione per LU

Data una qualunque matrice A non singolare, esiste una matrice di permutazione P non singolare t.c. PA = LU. N.B. P non è unica, ossia possono esistere diverse matrici P t.c. PA soddisfa le ipotesi del Teorema precedente

Teorema di Cholesky

Sia A una matrice di ordine n simmetrica e definita positiva, allora esiste una matrice triangolare inferiore L con elementi diagonali positivi, ((lii > 0 i = 1, . . . n) tale che $A = L \cdot L.T$

Teorema della convergenza

Se il sistema (1) ammette un' unica soluzione x e se il processo iterativo (2) è convergente, allora il vettore limite y coincide con la soluzione x, cioè x(k) = x per k -> infinito

Teorema condizione sufficiente alla convergenza

Se, per una qualche norma, risulta ||T||<1, allora il processo iterativo x (k) = Tx(k-1) + M -1b per k=1,2,…. è convergente per ogni x(0) .

Teorema convergenza Gauss-Siedel

Se la matrice A è simmetrica e definita positiva, il metodo di Gauss-Seidel è convergente.

Soluzione sistema a matrice simmetrica definita positiva

Sia A $R\,nxn$, matrice simmetrica e definita positiva, $b, x \in R\,n$, allora la soluzione del sistema lineare Ax = b coincide con il punto di minimo della seguente funzione quadratica

$$F(x) = \frac{1}{2} < Ax, x > - < b, x > = \frac{1}{2} x^T Ax - b^T x = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j - \sum_{i=1}^{n} b_i x_i$$

Teorema di Sylvester

Per le matrici simmetriche e definite positive. Il criterio di Sylvester è un teorema che fornisce una condizione necessaria e sufficiente affinché una matrice simmetrica sia definita positiva. Stabilisce che una matrice simmetrica è definita positiva se e solo se i determinanti di tutte le sottomatrici principali di testa Ak, k = 1, . . , n sono positivi. Poiché la sottomatrice principale di testa di ordine 1 è l'elemento a11 e la sottomatrice principale di testa di ordine n coincide con la matrice A, segue che a11 > 0 ed il determinante dell'Hessiano è positivo. Quindi il punto che annulla il gradiente è il minimo della forma quadratica. Quindi il vettore x * che minimizza la funzione F(x) coincide con la soluzione del sistema lineare (1); viceversa la soluzione del sistema (1) con matrice A simmetrica definita positiva minimizza la corrispondente funzione quadratica (2).

Teorema ortogonalità del residuo

Nel punto di minimo x (k+1) = x (k) + α (k)p (k) , ottenuto muovendosi lungo la direzione p (k) con α (k), il vettore residuo r (k+1) = Ax (k+1) − b risulta ortogonale alla direzione p (k) , cioè < r (k+1) , p (k) >= 0

Teorema del gradiente congiunto

Nel metodo del gradiente coniugato le direzioni di discesa p(k) , con k=0,1,.., formano un sistema di direzioni coniugate, mentre i vettori residui r(k) , con k=0,1,.., formano un sistema ortogonale, cioè

$$<r^{(k)}, r^{(j)}>=0 \qquad k\neq j\ ,\ j=0,1,..,k-1$$

$$<Ap^{(k)}, p^{(j)}>=0 \qquad k\neq j,\ \ j=0,1,..,k-1.$$

Teorema esistenza soluzione in un sistema sovradeterminato

Dato il sistema lineare sovradeterminato $Ax = b$

dove $A \in R^{m\times n},\ x \in R^n \ \ e\ b \in R^m,\ m > n$

$x^* = \arg\min_{x\in R^n} ||Ax - b||_2^2 \Leftrightarrow$ è la soluzione di $A^TAx = A^Tb$

La soluzione è unica se e solo se la matrice A ha rango massimo $rank(A) = n$ (le colonne di A sono linearmente indipendenti).

La soluzione del problema dei minimi quadrati mediante equazioni normali richiede solo che la matrice A del sistema sovradeterminato Ax=b abbia rango massimo.

Se questa condizione è verificata il vettore x* tale che

$$x^* = \arg\min_{x\in R^n} \|Ax - b\|_2^2$$

può essere sempre calcolato, teoricamente, come soluzione delle equazioni normali

$$A^TAx = A^Tb.$$

La matrice $G = A^TA$ è simmetrica e definita positiva e quindi il sistema può essere risolto utilizzando il metodo di Cholesky.

Teorema esistenza matrice singolare di SVD

Sia A R mxn a rango $k \leq min(m, n)$. Allora esistono due matrici ortogonali UR mxm e VR nxn tali che $U\,TAV = \Sigma = diag(\sigma 1, \sigma 2, \cdots, \sigma k, 0, \ldots, 0)$
I valori sulla diagonale di sono detti valori singolari di A e soddisfano $\sigma 1 \geq \sigma 2 \geq \cdots \geq \sigma k > 0$.
Se la matrice A ha rango $k = n$, avremo n > 0, $\sigma 1 \geq \sigma 2 \geq \cdots \geq \sigma n > 0$ se ha rango $k < n$ avremo che $\sigma k > 0$ e $\sigma k+1 = \sigma k+2 = \cdots \sigma n = 0$. $\sigma 1 \geq \sigma 2 \geq \cdots \geq \sigma k > 0$

Teorema dell'errore interpolazione

Siano assegnate le coppie (xi,yi) i=0,…,n $a \equiv x0 < x1 < \ldots < xn \equiv b$ e yi=f(xi) siano i valori assunti in questi punti da una funzione f(x) definita in [a,b] e continua insieme alle sue derivate fino a quella di ordine n+1, (f(x) C n+1[a,b]).
Sia Pn(x) il polinomio di grado n che interpola tali coppie di dati. Sia x [a,b] , indichiamo con $E(\bar{x}) = f(\bar{x}) - Pn(\bar{x})$.
Se i x = x allora l'errore è nullo perchè si annulla il fattore $\omega n+1(xi) = (xi - x0)(xi - x1). . , (xi - xi). . . . . (xi - xn) = 0$.

Teorema gradiente Lipschitziano

Ipotizziamo che f: $Rn \to R$ sia una funzione convessa e differenziabile con gradiente ∇f(x) continuo Lipschitz con costante L, e che $x*$ tale che
$\nabla f(x*) = 0$. Consideriamo l'iterazione del metodo della discesa del gradiente (GD).
$x\,(k+1) = x\,(k) - \eta \nabla f(x\,(k))$
dove
$0 < \eta < 2/L$
allora la sequenza di iterati $x\,(k)$ converge ad $x*$