ALGORITMI METODI NUMERICI

Metodi per Zeri di Equazioni non Lineari

- Metodo di Bisezione:

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.

```
def metodo_bisezione(fname, a, b, tolx):

 fa=fname(a)
 fb=fname(b)
 if sign(fa * fb) >= 0:#to do
     print("Non è possibile applicare il metodo di bisezione \n")
     return None, None,None

 it = 0
 v_xk = []

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

    if sign(fa * fxk) > 0:# to do
      a = xk#to do
      fa= fxk#to do
    elif sign(fb * fxk) > 0:# to do
      b = xk
      fb= fxk

 return xk, it, v_xk
```

Il metodo di bisezione converge globalmente alla soluzione La convergenza è garantita qualunque sia l'ampiezza dell'intervallo iniziale [a,b]. E' un metodo molto lento. ordine di convergenza lineare, p=1, e fattore di convergenza c=1/2.

- Metodo Regula Falsi:

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.

```
def falsa_posizione(fname,a,b,tolx,tolf,maxit):
    fa=fname(a)
    fb=fname(b)
    if sign(fa * fb) >= 0:#to do:
       print("Metodo di bisezione non applicabile")
       return None,None,None

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

        if sign(fb * fxk) > 0:#to do
           b= xk#to do
           fb= fxk#to do
        elif sign(fa * fxk) > 0:#to do
           a= xk#to do
           fa= fxk#to do
        if xk!=0:
            errore= abs(xk - xprec) / abs(xk)#to do
        else:
            errore=abs(xk - xprec)#to do
        xprec=xk
    return xk,it,v_xk
```

tiene conto dei valori della funzione agli estremi dell'intervallo, mentre bisezione tiene conto solo dei segni. Convergenza globale e più veloce di bisezione (convergenza superlineare)

- Metodo delle Corde:

Implementa il metodo delle corde per il calcolo degli zeri di un'equazione non lineare.

Parametri:
fname: La funzione da cui si vuole calcolare lo zero.
m: coefficiente angolare della retta che rimane fisso per tutte le iterazioni
tolx: La tolleranza di errore tra due iterati successivi
tolf: tolleranza sul valore della funzione
nmax: numero massimo di iterazione

Restituisce:
Lo zero approssimato della funzione, il numero di iterazioni e la lista degli iterati intermedi.

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

    # coeff_ang è il coefficiente angolare della retta che rimane fisso per tutte le
iterazioni
        xk=[]
        it=0
        errorex=1+tolx
        erroref=1+tolf
        while it < nmax and errorex >= tolx and erroref >= tolf:#to do
           fx0=fname(x0)# to do
           d=fx0 / coeff_ang# to do

           x1=x0 - d#to do
           fx1=fname(x1)#
           if x1!=0:
                errorex=abs(d) / abs(x1)#to do
           else:
                errorex=abs(d)#to do

           erroref=np.abs(fx1)#to do

           x0=x1
           it=it+1
           xk.append(x1)

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

        return x1,it,xk
```

Ho un punto (x_0, $f(x_0)$) su una curva e voglio trovare il punto in cui la retta con pendenza m passante per quel punto interseca l'asse x. La retta ha equazione: $y - f(x_0) = m(x - x_0)$
Pongo $y = 0$ (intersezione con asse x) e viene $x = x_0 - (f(x_0) / m)$
QUINDI nel codice x1 = x0 - fx0 / m DOVE d = fx0 / m

Metodo di linearizzazione (approssima funzione con una retta). Usa sempre lo stesso coeff. angolare.

- Metodo di Newton:

Implementa il metodo di Newton per il calcolo degli zeri di un'equazione non lineare.

Parametri:
fname: La funzione di cui si vuole calcolare lo zero.
fpname: La derivata prima della funzione di cui si vuole calcolare lo zero.
x0: iterato iniziale
tolx: La tolleranza di errore tra due iterati successivi
tolf: tolleranza sul valore della funzione
nmax: numero massimo di iterazione
Restituisce:
Lo zero approssimato della funzione, il numero di iterazioni e la lista degli iterati intermedi.

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

        xk=[]
        it=0
        errorex=1+tolx
        erroref=1+tolf
        while it < nmax and errorex >= tolx and erroref >= tolf:#to do
           fx0=fname(x0)
           if abs(fpname(x0)) <= np.spacing(1):#to do
                print(" derivata prima nulla in x0")
                return None, None,None
           d=fx0 / fpname(x0)#to do
           x1=x0 - d#to do
           fx1=fname(x1)
           erroref=np.abs(fx1)
           if x1!=0:
                errorex=abs(d) / abs(x1)#to do
           else:
                errorex=abs(d)#to do

           it=it+1
           x0=x1
           xk.append(x1)

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

        return x1,it,xk
```

Si considera a ogni passo la retta tangente alla funzione e passante per $(x_k, f(x_k))$ cioè il polinomio di Taylor (retta che approssima meglio la funzione in un intorno). Se lo zero ha molteplicità m > 1, metodo non ha convergenza quadratica e si usa il modificato.

- Metodo delle Secanti:

Implementa il metodo delle secanti per il calcolo degli zeri di un'equazione non lineare.

Parametri:
fname: La funzione di cui si vuole calcolare lo zero.
xm1, x0: primi due iterati
tolx: La tolleranza di errore tra due iterati successivi
tolf: tolleranza sul valore della funzione
nmax: numero massimo di iterazione

Restituisce:
Lo zero approssimato della funzione, il numero di iterazioni e la lista degli iterati intermedi.

```
def secanti(fname,xm1,x0,tolx,tolf,nmax):
        xk=[]
        it=0
        errorex=1+tolx
        erroref=1+tolf
        while it < nmax and errorex >= tolx and erroref >= tolf:#to do
            fxm1=fname(xm1)#to do
            fx0=fname(x0)#to do
            d=fx0 * (x0 - xm1) / (fx0 - fxm1)#to do
            x1=x0 - d#to do
            fx1=fname(x1)
            xk.append(x1);
            if x1!=0:
                errorex=abs(d) / abs(x1)#to do
            else:
                errorex=abs(d)#to do

            erroref=np.abs(fx1)#to do
            xm1=x0#to do
            x0=x1#to do
            it=it+1;

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

        return x1,it,xk
```

Dati due punti iniziali x_0 e x_1 si traccia la retta che li unisce. L'intersezione tra la retta e l'asse x diventa x_2 e si traccia la retta secante tra x_1 e x_2 ecc…
COnvergenza locale e superlineare

- Metodo di Newton Modificato:

Implementa il metodo di Newton modificato da utilizzato per il calcolo degli zeri di un'equazione non lineare nel caso di zeri multipli.

Parametri:

fname: La funzione di cui si vuole calcolare lo zero.
fpname: La derivata prima della funzione di cui si vuole calcolare lo zero.
m: molteplicità della radice
x0: iterato iniziale
tolx: La tolleranza di errore tra due iterati successivi
tolf: tolleranza sul valore della funzione
nmax: numero massimo di iterazione

Restituisce:

Lo zero approssimato della funzione, il numero di iterazioni e la lista degli iterati intermedi.

```
def newton_modificato(fname,fpname,m,x0,tolx,tolf,nmax):
    #m è la molteplicità dello zero
        xk=[]
        it=0
        errorex=1+tolx
        erroref=1+tolf
        while it < nmax and errorex >= tolx and erroref >= tolf:#to do
           fx0=fname(x0)
           if abs(fpname(x0)) <= np.spacing(1):#to do
                print(" derivata prima nulla in x0")
                return None, None,None
           d=fx0 / fpname(x0)#to do
           x1=x0 - m * d#to do
           fx1=fname(x1)
           erroref=np.abs(fx1)
           if x1!=0:
                errore=abs(d) / abs(x1)#to do
           else:
                errore=abs(d)#to do

           it=it+1
           x0=x1
           xk.append(x1)

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

        return x1,it,xk
```

Cambia solo che quando calcoli x1 moltiplichi m * d

Metodi per Sistemi non Lineari

- Metodo di Newton-Raphson:

Funzione per la risoluzione del sistema F(x)=0 mediante il metodo di Newton.

Parametri:
fun: funzione vettoriale contenente ciascuna equazione non lineare del sistema.
jac: funzione che calcola la matrice Jacobiana della funzione vettoriale.
x0: Vettore contenente l'approssimazione iniziale della soluzione.
tolx: float - Parametro di tolleranza per l'errore assoluto.
tolf: float - Parametro di tolleranza per l'errore relativo.
nmax: int - Numero massimo di iterazioni.

Restituisce:
x: Vettore soluzione del sistema (o equazione) non lineare.
it: Numero di iterazioni fatte per ottenere l'approssimazione desiderata.
Xm: Vettore contenente la norma dell'errore relativo tra due iterati successivi.

```python
def newton_raphson(initial_guess, F_numerical, J_Numerical, tolX, tolF, max_iterations):
    X= np.array(initial_guess, dtype=float)
    it=0
    erroreF=1+tolF
    erroreX=1+tolX
    errore=[]
    while it < max_iterations and erroreX >= tolX and erroreF >= tolF:#to do
        jx = J_Numerical(X[0], X[1])#to do
        if np.linalg.det(jx) == 0:#to do
            print("La matrice dello Jacobiano calcolata nell'iterato precedente non è a rango massimo")
            return None, None,None

        fx = F_numerical(X[0], X[1])#To do
        fx = fx.squeeze()
        s = np.linalg.solve(jx, -fx)#to do
        Xnew= X + s#to do
        normaXnew=np.linalg.norm(Xnew,1)
        if normaXnew !=0:
            erroreX= np.linalg.norm(s, 1) / normaXnew#to do
        else:
            erroreX=np.linalg.norm(s, 1)#to do

        errore.append(erroreX)
        fxnew=F_numerical(Xnew[0], Xnew[1])#to do
        erroreF= np.linalg.norm(fxnew.squeeze(),1)
        X=Xnew
        it=it+1

    return X,it,errore
```

Metodo a convergenza locale e ordine di convergenza quadratico

- Metodo delle Corde:

Funzione per la risoluzione del sistema f(x)=0 mediante il metodo di Newton, con variante delle corde, in cui lo Jacobiano non viene calcolato ad ogni iterazione, ma rimane fisso, calcolato nell'iterato iniziale x0.

Parametri:

fun: funzione vettoriale contenente ciascuna equazione non lineare del sistema.
jac: funzione che calcola la matrice Jacobiana della funzione vettoriale.
x0: Vettore contenente l'approssimazione iniziale della soluzione.
tolx : float - Parametro di tolleranza per l'errore tra due soluzioni successive.
tolf: float - Parametro di tolleranza sul valore della funzione.
nmax: Numero massimo di iterazioni.

Restituisce:

x: Vettore soluzione del sistema (o equazione) non lineare.
it: Numero di iterazioni fatte per ottenere l'approssimazione desiderata.
Xm: Vettore contenente la norma dell'errore relativo tra due iterati successivi.

```python
def newton_raphson_corde(initial_guess, F_numerical, J_Numerical, tolX, tolF, max_iterations):
    X= np.array(initial_guess, dtype=float)
    it=0
    erroreF=1+tolF
    erroreX=1+tolX
    errore=[]
    while it < max_iterations and erroreX >= tolX and erroreF >= tolF:#to do
        if it == 0:#to do
            jx = J_Numerical(X[0], X[1])#to do
            if np.linalg.det(jx) == 0:#to do
                print("La matrice dello Jacobiano calcolata nell'iterato precedente non è a rango massimo")
                return None, None,None

        fx = F_numerical(X[0], X[1])#To do
        fx = fx.squeeze()
        s = np.linalg.solve(jx, -fx)#to do
        Xnew= X + s#to do
        normaXnew=np.linalg.norm(Xnew,1)
        if normaXnew !=0:
            erroreX=np.linalg.norm(s, 1) / normaXnew#to do
        else:
            erroreX=np.linalg.norm(s, 1)#to do

        errore.append(erroreX)
        fxnew=F_numerical(Xnew[0], Xnew[1])#to do
        erroreF= np.linalg.norm(fxnew.squeeze(),1)
        X=Xnew
        it=it+1

    return X,it,errore
```

Calcolare ogni volta lo jacobiano chiede di valutare n^2 derivate parziali.

- Metodo di Shamanskii:

Funzione per la risoluzione del sistema f(x)=0 mediante il metodo di Newton, con variante delle shamanski, in cui lo Jacobiano viene aggiornato ogni un tot di iterazioni, deciso dall'utente.

Parametri:

fun: funzione vettoriale contenente ciascuna equazione non lineare del sistema.
jac: funzione che calcola la matrice Jacobiana della funzione vettoriale.
x0: Vettore contenente l'approssimazione iniziale della soluzione.
tolx: float - Parametro di tolleranza per l'errore tra due soluzioni successive.
tolf: float - Parametro di tolleranza sul valore della funzione.
nmax: Numero massimo di iterazioni.

Restituisce:

x: Vettore soluzione del sistema (o equazione) non lineare.
it: Numero di iterazioni fatte per ottenere l'approssimazione desiderata.
Xm: Vettore contenente la norma dell'errore relativo tra due iterati successivi.

```python
def newton_raphson_sham(initial_guess, update, F_numerical, J_Numerical, tolX, tolF, max_iterations):
    X= np.array(initial_guess, dtype=float)
    it=0
    erroreF=1+tolF
    erroreX=1+tolX
    errore=[]
    while it < max_iterations and erroreX >= tolx and erroreF >= tolf:#to do
        if it % update == 0:# to to:
            jx = J_Numerical(X[0], X[1])#to do
            if np.linalg.det(jx) == 0:#to do
                print("La matrice dello Jacobiano calcolata nell'iterato precedente non è a rango massimo")
                return None, None,None

        fx = F_numerical(X[0], X[1])#To do
        fx = fx.squeeze()
        s = np.linalg.solve(jx, -fx)#to do
        Xnew= X + s#to do
        normaXnew=np.linalg.norm(Xnew,1)
        if normaXnew !=0:
            erroreX= np.linalg.norm(s, 1) / normaXnew#to do
        else:
            erroreX=np.linalg.norm(s, 1)#to do

        errore.append(erroreX)
        fxnew=F_numerical(Xnew[0], Xnew[1])#to do
        erroreF= np.linalg.norm(fxnew.squeeze(),1)
        X=Xnew
        it=it+1

    return X,it,errore
```

- Metodo di Newton-Raphson per Calcolo del Minimo:

Funzione di newton-raphson per calcolare il minimo di una funzione in più variabili

Parametri:
fun: Nome della funzione che calcola il gradiente della funzione non lineare.
Hess: Nome della funzione che calcola la matrice Hessiana della funzione non lineare.
x0: Vettore contenente l'approssimazione iniziale della soluzione.
tolx: float - Parametro di tolleranza per l'errore assoluto.
tolf: float - Parametro di tolleranza per l'errore relativo.
nmax: Numero massimo di iterazioni.

Restituisce:
x: Vettore soluzione del sistema (o equazione) non lineare.
it: Numero di iterazioni fatte per ottenere l'approssimazione desiderata.
Xm: Vettore contenente la norma del passo ad ogni iterazione.

```python
def newton_raphson_minimo(initial_guess, grad_func, Hessian_func, tolX, tolF, max_iterations):
    X= np.array(initial_guess, dtype=float)
    it=0
    erroreF=1+tolX
    erroreX=1+tolF
    errore=[]
    while it < max_iterations and erroreX >= tolX and erroreF >= tolF:#to do:
        Hx = Hessian_func(X[0], X[1])#to do
        if np.linalg.det(Hx) == 0:#to do
            print("La matrice Hessiana calcolata nell'iterato precedente non è a rango massimo")
            return None, None,None

        gfx = grad_func(X[0], X[1])#to do
        gfx = gfx.squeeze()
        s = np.linalg.solve(Hx, -gfx)#to do
        Xnew= X + s#to do
        normaXnew=np.linalg.norm(Xnew,1)
        if normaXnew!=0:
            erroreX= np.linalg.norm(s, 1) / normaXnew#to do
        else:
            erroreX=np.linalg.norm(s, 1)#to do

        errore.append(erroreX)
        gfxnew=grad_func(Xnew[0], Xnew[1])#to do
        erroreF= np.linalg.norm(gfxnew.squeeze(),1)
        X=Xnew
        it=it+1

    return X,it,errore
```

Metodi Diretti per Sistemi Lineari

- m = n
  - Piccola e Densa
    - Ben condizionata → GAUSS (LU)
    - Mal condizionata → HOUSEHOLDER (QR)
    - Simmetrica e def. pos. → CHOLESKY (LL^T o R^TR)
  - Grande e Sparsa:
    - Diagonale Dominante → Jacobi (anche gauss-seidel)
    - Simmetrica e def. pos. → GS, GS SOR, discesa
- m > n
  - Ben condizionata e rango massimo → eqnorm
  - Mediamente mal condizionata e rango massimo → QR
  - Non rango massimo → SVD

```python
dati = loadmat()
A = dati['A']
b = dati['b']

m, n = A.shape
print("dimensioni matrice: ", m, n)

nz = np.count_nonzero(A)/(n*m)
perc_nz = nz * 100
print("Percentuale elementi diversi da 0: ", perc_nz)

flag = A==A.T
if np.all(flag) == 0:
    print("Matrice non simmetrica")
else:
    print("matrice simmetrica")

#se simmetrica, per sapere se è definita positiva:
autovalori = np.linalg.eigvals(A)
def_pos = np.all(autovalori > 0)
print(def_pos)
#se torna False non è definita positiva
```

```python
#diagonale dominante?
def verifica_dd(A):
    n = A.shape[0]
    flag = True
    for i in range(n):
        el_diag = np.abs(A[i, i])
        sum_extradiag = np.sum(np.abs(A[i, :])) - np.abs(A[i, i])
        if el_diag < sum_extradiag:
            print("Matrice non a diagonale dominante")
            flag = False
            return flag
    return flag

dd = verifica_dd(A)
print("Matrice a diagonale dominante?: ", dd)
```

PT, L, U = scipy.linalg.lu(A) poi P = PT.T.copy() oppure L=scipy.linalg.cholesky(A,lower=True) oppure Q, R = scipy.linalg.qr(A)

- Fattorizzazione di Gauss LU

Algoritmo stabile in senso debole perché la costante che maggiora gli elementi di L non dipende dall'ordine della matrice, mentre la costante che maggiora gli elementi di U dipende in maniera esponenziale dall'ordine della matrice

```
def LUsolve(P,A,L,U,b):
    pb=np.dot(P,b)
    y,flag=Lsolve(L,pb)
    if flag == 0:
        x,flag=Usolve(U,y)
    else:
        return [],flag
    return x,flag
```

- Fattorizzazione LL^T di Cholesky

Algoritmo stabile in senso forte

- Fattorizzazione QR

Algoritmo stabile in senso debole ma + stabile di Gauss perchè non esponenziale ma sqrt(n)

```
Q,R=sp.linalg.qr(A)
y=np.dot(Q.T,b)
xqr,flag=Usolve(R,y)
```

MATRICE PICCOLA SE VA DA 10 A 100
MATRICE GRANDE SE VA DA 300 A 500
SPARSA SE ZERI NELLA MATRICE < 33%
Per metodi di discesa:
PER FARLO O USO METODO SYLVESTER , I MINORI PRINCIPALI HANNO TUTTI DETERMINANTE POSITIVO OPPURE SE AUTOVALORI SONO REALI E POSITIVI

A BEN CONDIZIONATA SE INDICE DI CONDIZIONAMENTO FINO A 100
A MEDIAMENTE MAL CONDIZIONATA DA 10^3, 10^4
A MAL CONDIZIONATA SE > 10^5
USO np.linalg.cond PER INDICE DI CONDIZIONAMENTO MA SOLO SE SO LA FORMULA PERCHE LA CHIEDE si calcola come:

$\kappa(A) = \|A\|_2 \cdot \|A^{-1}\|_2$ ossia k = np.linalg.norm(A, 2) * np.linalg.norm(np.linalg.inv(A), 2)

Metodi Iterativi per Sistemi Lineari

Condizione necessaria e sufficiente per la convergenza del metodo iterativo è che il raggio spettrale (autovalore di modulo massimo) della matrice di iterazione T sia minore di 1

$$\rho(T) < 1$$

Il metodo è tanto più veloce quanto il raggio spettrale è più piccolo

- Jacobi

A = M - N si ottiene con M = D e N = - (E + F) con D diagonale, E tr. inferiore F tr. superiore

```
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+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
```

Si può usare questo algoritmo se gli elementi diagonali di A sono diversi da 0. Se non lo sono e A è non singolare, si possono scambiare le colonne e rispettivamente le incognite. Inoltre il metodo è programmabile in parallelo

- Gauss-Seidel

A = M - N si ottiene con M = E + D e N = - F con D diagonale, E tr. inferiore F tr. superiore

```python
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
    invM=np.linalg.inv(M)
    T=invM@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
```

Uguale a Jacobi ma utilizzando i risultati intermedi per aggiornare i risultati mano a mano.
Non si presta a essere parallelizzato perchè dipende da ris. precedenti

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

TEOREMA:
Se la matrice A è a diagonale strettamente dominante
Allora sia il metodo di Jacobi che quello di Gauss-Seidel convergono

- Gauss-Seidel SOR

Metodo Successive Over-Relaxation: usati per accellerare la convergenza di Gauss-Seidel per sistemi dove converge lentamente.

Gauss-Seidel diventa: $x^{(k)} = x^{(k-1)} + \omega r^{(k)}$ e il problema principale diventa scegliere in modo ottimale omega

```
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
```

Metodi di Discesa per Sistemi Lineari

- Steepest Descent

Scelgo a ogni passo k, p (direzione) come l'antigradiente di F, quindi la direzione di max. decrescita. Grafico sarà a zig-zag. alpha è lo step-size. convergenza lineare, metodo lento

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

    iterates_array = np.vstack([arr.T for arr in vec_sol])
    return x,vet_r,iterates_array,it
```

Risolve un sistema di equazioni lineari.
Per matrici quadrate (m=n) sparse con grandi dimensioni e simmetrica e definita positiva.
Usato per meno tempo di convergenza

$$\alpha^{(k)} = -\frac{<r^{(k)},p^{(k)}>}{<Ap^{(k)},p^{(k)}>} = -\frac{(r^{(k)})^T p^{(k)}}{(p^{(k)})^T A p^{(k)}}$$ alpha è lo step-size (quanto mi devo muovere nella direzione p)

La nuova soluzione x = x + alpha * p

$F(x^{(k)} + \alpha^{(k)} p^{(k)}) < F(x^{(k)})$ deve valere questa condizione

Fattore di convergenza dipende dall'indice di condizionamento di A. Tanto più K(A) è alto tanto più è lenta la convergenza.
Ad una matrice A mal condizionata corrisponde un' iperellissoide molto allungato, mentre ad un K(A) piccolo corrisponde un iperellissoide più arrotondato

- Conjugate Gradient

Questo metodo tiene conto sia del gradiente r (il residuo coincide con il gradiente) che della direzione di discesa dell'iterazione precedente. Con gamma si fa in modo che la nuova direzione sia coniugata a quella precedente ossia punti verso il centro

```python
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.copy())
    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.copy())
        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)
    iterates_array = np.vstack([arr.T for arr in vec_sol])
    return x,vet_r,iterates_array,it
```

Risolve un sistema di equazioni lineari.
Per matrici quadrate (m=n) sparse con grandi dimensioni e simmetrica e definita positiva.
Usato per meno iterazioni e più efficienza

Sia A€R^nxn e x,b€R^n
Allora la soluzione del sistema Ax=b coincide a minimizzare la seguente funzione quadratica
F(x)=½<Ax,x>-<b,x>

Metodi per Sistemi Sovradeterminati

- Metodo delle Equazioni Normali

Per risolvere sistema sovradeterminato uso equazioni normali: **Gx = A^Tb** dove G = A^TA matrice nxn
Dato il sistema lineare sovradeterminato $Ax = b$
$x^* = arg\ min\ ||Ax - b||_2^2 \Leftrightarrow x^*$ 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
La soluzione del problema dei minimi quadrati mediante equazioni normali richiede solo che la matrice A del sistema sovradeterminato Ax=b abbia rango massimo.
La matrice G = A^TA è simmetrica e definita positiva e quindi il sistema può essere risolto utilizzando il metodo di Cholesky

import scipy.linalg as spLin

```
def eqnorm(A,b):

    G=A.T@A
    condG=np.linalg.cond(G)
    print("Indice di condizionamento di G ",condG)
    f=A.T@b

    L=spLin.cholesky(G,lower=True)
    U=L.T

    z,flag=SolveTriangular.Lsolve(L,f)
    if flag==0:
        x,flag=SolveTriangular.Usolve(U,z)

    return x
```

Si usa per la retta di regressione (retta di approssimazione ai minimi quadrati) o la parabola
Si definiscono i punti sperimentali e si crea un numpy array chiamati x e y
Si definisce n come il grado del polinomi di regressione (1 per la retta 2 per la parabola)
Costruisco A come la matrice di Vandermonde: A = np.vander(x, increasing = True)[:,:n+1]
Posso calcolarne il condizionamento (aumenta all'aumentare delle colonne di Vandermonde)
Implemento eqnorm → a = eqnorm(A, y)
a è il vettore dei coefficienti del polinomio di regressione, dopo uso flip perchè sono al contr.
xv = np.linspace(iniz, fine, numero di elementi)
Ora calcolo il polinomio con coefficienti np.flip(a) in xv → pol1 = np.polyval(np.flip(a), xv)
Disegno i punti e la retta / parabola con plt.plot(x, y, ‘bo’, xv, pol1, ‘r-’)

- Metodo QR

Fattorizzo A = QR dove Q ortogonale e R triangolare superiore

Il problema diventa: $\min_x \|Ax - b\| = \min_x \|QRx - b\| = \min_x \|Rx - Q^T b\|$

Si risolve un sistema triangolare Rx = Q^Tb più stabile ed efficiente

Non stabile in senso forte ma mediamente

Il residuo è l'errore di approssimazione

```
def qrLS(A,b):
    n=A.shape[1]  # numero di colonne di A
    Q,R=spLin.qr(A)
    h=Q.T@b
    x,flag=SolveTriangular.Usolve(R[0:n,:],h[0:n])
    residuo=np.linalg.norm(h[n:])**2
    return x,residuo
```

- Decomposizione in Valori Singolari: SVD

Il metodo delle equazioni normali e il metodo QR richiedono rango massimo. Se A non ha rango massimo si usa SVD

$A = U\Sigma V^T$ U ortogonale, V ortogonale, S matrice diagonale con valori singolari >=0

Valori singolari sono i valori sulla diagonale di S:

- σ_1 è il massimo valore singolare
- σ_{max} / σ_{min} dà l'indice di condizionamento di A
- Numero di valori singolari non nulli è il rango di A

```
def SVDLS(A,b):
    m,n=A.shape  #numero di righe e  numero di colonne di A
    U,s,VT=spLin.svd(A)  #Attenzione : Restituisce U, il numpy-array 1d che contiene la diagonale della matrice Sigma e VT=VTrasposta)
    #Quindi
    V=VT.T
    thresh=np.spacing(1)*m*s[0] ##Calcolo del rango della matrice, numero dei valori singolari maggiori di una soglia
    k=np.count_nonzero(s>thresh)
    print("rango=",k)
    d=U.T@b
    d1=d[:k].reshape(k,1)
    s1=s[:k].reshape(k,1)
    #Risolve il sistema diagonale di dimensione kxk avene come matrice dei coefficienti la matrice Sigma
    c=d1/s1
    x=V[:,:k]@c
    residuo=np.linalg.norm(d[k:])**2
    return x,residuo
```

Metodi per Interpolazione Polinomiale

```python
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) #Calcola i coefficienti del polinomio di grado n che si annulla nel vettore xzeri
    den=np.polyval(num,xnodi[j]) #Lo valuta nel nodo escluso (-jesimo)

    p=num/den

    return p
```

```python
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
```

Costruisco due numpy array x e y con i punti → grado del polinomio di Lagrange avrà grado numero di elementi di x - 1
Implemento i due metodi
xv = np.linspace() è in quanti punti voglio valutare
polL = InterpL(x, y, xv)
plt.plot(x, y, ‘ro’, xv, polL)
Da cosa dipende l’errore di interpolazione?
Regolarità della funzione
Disposizione dei punti di interpolazione sull’asse delle ascisse

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

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

ERRORE E' NULLO SE PRENDO COME PUNTO DI CALCOLO DELL'ERRORE UNO DEI PUNTI SPERIMENTALI E SE LA FUNZIONE HA LO STESSO GRADO DEL POLINOMIO INTERPOLATORE

La costante di Lebesgue risulta essere il coefficiente di amplificazione degli errori relativi sui dati e pertanto identifica il numero di condizionamento del problema di interpolazione polinomiale.

```
Lebesgue = 0
for i in range(x_new.size):
    p = plagr(x_new, i)
    Lebesgue += np.abs(np.polyval(p, x_new))
leb_const = np.linalg.norm(Lebesgue, ord = np.inf)
print(leb_const)
```

```
A = np.array([[4.5, 1, 3, 2], [1, -8, 2, 1], [-1, -2, -3, -1], [2, 6, 0, 1]])
PT, L, U = sp.linalg.lu(A)
P = PT.T
detA = np.linalg.det(P) * np.prod(np.diag(U))
print("Determinante di A: ", detA) # Il determinante di A = det(LU) = det(L) * det(U) = det(P) * produttoria di U
print("Determinante con linalg di A: ", np.linalg.det(A))
# Ora risolvo n sistemi lineari Axi = ei
```

```
n = 4
I = np.eye(n)
X = np.zeros_like(A)
for i in range(n):
    b = I[:,i]
    y, flag = Lsolve(L, P@b)
    if flag == 0:
        x, flag = Usolve(U, y)
    X[:,i] = x.reshape(n, )
print(X)
print("Inversa con linalg di A: \n", np.linalg.inv(A))
```

Per calcolare l'inversa e il determinante sfruttando la fattorizzazione LU

```
for i in range(1, n):
    print(np.linalg.det(A[:i, :i]))
```

Per verificare se si può fare fattorizzazione LU

b=np.sum(A,axis=1).reshape(4,1)
Per far sì che il vettore soluzione sia tutti 1

```
rango = np.linalg.matrix_rank(A1)
massimo_rango = min(A1.shape)  # il ra
if rango == massimo_rango:
    print("A1 è a rango massimo")
else:
    print("A1 non è a rango massimo")
```

Se sistema sovradeterminato