8 Quarta lezione

8.1 Importare file Python

Abbiamo visto come utilizzare le librerie, tutto a partire dal comando import. Oltre alle librerie possiamo importare anche altri file Python scritti da noi, magari perchè in quel file è implementata una funzione che ci serve. Facciamo un esempio:

```
def f(x, n):
    """
    restituisce la potenza n-esima di un numero x
    Parametri
    ---------
    x, n : float

    Return
    ---------
    v : float
        x**n
    """

    v = x**n

    return v

if __name__ == '__main__':
    #test
    print(f(5, 2))

[Output]
25
```

Abbiamo questo codice che chiamiamo "elevamento.py" che ha implementato la funzione di elevamento a potenza e supponiamo di voler utilizzare questa funzione in un altro codice, possiamo farlo grazie ad import:

```
import elevamento

print(elevamento.f(3, 3))

[Output]
27
```

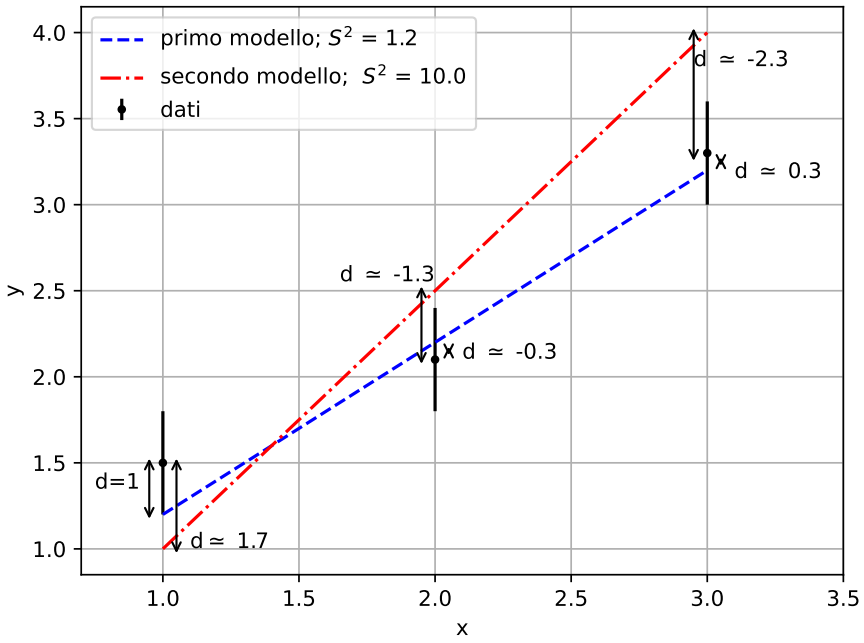
Notiamo nel codice iniziale la presenza dell' if, esso serve per far si che tutto ciò che sia scritto sotto venga eseguito solo se il codice viene lanciato come 'main' appunto e non importato come modulo su un altro codice. In genere l'utilizzo di questa istruzione è buona norma quando si vuol scrivere un codice da importare altrove.

8.2 Fit

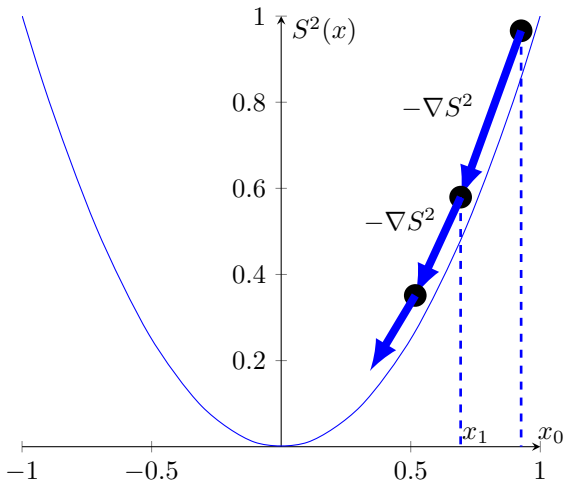
Nell'ambito della statistica un fit, cioè una regressione lineare o non che sia (dove la linearità è riferita ai parametri della funzione), è un metodo per trovare la funzione che meglio descrive l'andamento di alcuni dati. Nel caso di regressione lineare la procedura da eseguire non è troppo complicata, mentre per la regressione non lineare le cose si fanno parecchio complicate e si utilizzano algoritmi di ottimizzazione. Se noi abbiamo quindi un modello teorico che ci dice che un corpo cade con una legge oraria della forma $y(t) = h_0 - \frac{1}{2}gt^2$, grazie al fit possiamo trovare i valori dei parametri della leggere oraria, h_0 e g, che meglio adattano la curva ai dati (nella speranza che escano valori fisicamente sensati, dato che in genere i dati sono di origine sperimentale o simulativa). In ogni caso comunque l'idea di ciò che va fatto è trovare il minimo della seguente funzione:

$$S^2(\{\theta\}_j) = \sum_i \frac{(y_i - f(x_i; \{\theta\}_j))^2}{\sigma_{y_i}^2} \tag{1}$$

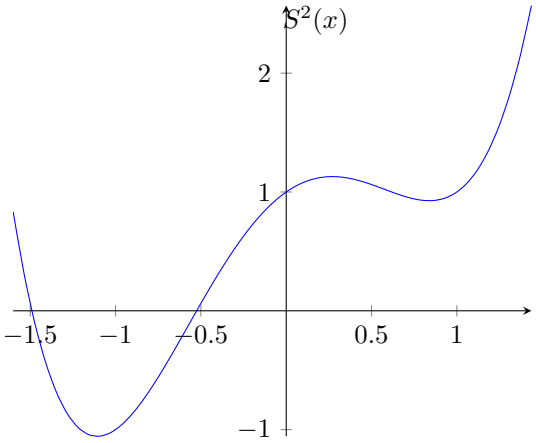
che nel caso in cui il termine dentro la somma sia distribuito in modo gaussiano allora la quantità S^2 è distribuita come un chiquadro, e da qui si potrebbe fare tutta una discussione sulla significatività statistica di quello che andiamo a fare, che ovviamente noi non facciamo. Analizziamo un attimo questa formula: S^2 è in linea di principio una funzione a molte variabili e che resituisce un numero reale. Il termine dentro la somma rappresenta la distanza tra valore del dato e valore della funzione in unità della barra d'errore del dato. Facciamo un esempio visivo per rendere più chiaro il concetto. Consideriamo giusto a titolo di esempio tre punti e due possibili rette che noi possiamo pensare che più o meno approssimino i dati.



Nel grafico d è proprio la distanza del modello dal dato in unità di barre di errore, e quindi la somma di tutte queste quantità elevate al quadrato è il valore di S^2 che è riportato nella legenda. Notiamo quindi che effettivamente la retta che presenta un valore di S^2 minore è quella che ad occhio meglio approssima i dati. Ora uno potrebbe pensare che quindi basta calcolare S^2 su una griglia e vedere dove assume il valore più piccolo. Questo è un metodo abbastanza brute force e il linea di principio funziona, ma quello che in genere si fa è un po' diverso. In linea di principio per capire come funziona un'algoritmo di minimo basta pensare ad una palline che cade in una ciotola. Precisiamo che si vuole fare tutta questa trattazione per far capire che la parte più delicata di questa procedura è scegliere quello che noi chiameremo nel codice "init" e che esso violentemente aggiusta o complica la nosta situazione. Consideriamo per semplicità, didattica e grafica, una funzione di una singola variabile.



Quel che noi facciamo è scegliere un x_0, (il nostro init) ed aggiornare questa posizione considerando la pendenza della funzione, che altro non sarebbe che la derivata della funzione che vogliamo considerare. Concedetemi, per maggiore generalità, si sostituire il termine derivata con il termine gradiente, indicato dal simbolo ∇. Quindi quello che il codice fa è diciamo analogo ad una pallina che si muove sotto l'azione di un potenziale, che sarebbe S^2 e la sua derivata, il suo gradiente, non è altro che la forza che la pallina sente. Facendo così troviamo una serie di x_i ierativamente, fino ad arrivare al minimo dove il gradiente, la forza esterna, è zero. Questo metodo è chiamato gradiente discendente. Finche abbiamo un solo minimo quindi va tutto bene. Lo troviamo senza problemi a prescindere da dove partiamo. Supponiamo ora una situazione più brutta:



In questo caso vediamo subito che abbiamo due punti in cui la derivata è nulla, quindi due minimi (questo sarebbe il caso di una regressione non lineare, a differenza di quella lineare di sopra, un solo minimo), ma quello a cui siamo interessati noi è il minimo assoluto. Se utilizzassimo il metodo precedente è facile vedere che se partiamo per esempio con $x > 1$ ci incastraimo nel minimo locale. Le uniche zone buone sono soltanto quelle con $x < 0$. Vedete quindi che una piccola complicazione riduce di molto le nostre possibilità e dobbiamo quindi selezionare il nostro punto di partenza con delicatezza. Questo perchè una volta arrivato al minimo locale la nosta "pallina" non ci arriva con una velocità con accadrebbe nella realtà e quindi non riesce a scavallare la collinetta. Fondamentalmente per migliorare la cosa dobbiamo spiegare al computer il concetto di inerzia e anche di attrito (se l'energia si conservasse la pallina oscillerebbe all'infinito e il codice non terminerebbe). Un esempio di ciò, chiamato gradiente discendente con momento, e anche di quanto visto sopra è disponibile in una delle appendici. Inoltre in questa stessa lezione andremo a vedere cosa fa effettivamente "curve_fit" che è un pochino diverso. Un caso ancora peggiore lo vediamo adesso con un problema fisico, dove ora non abbiamo un semiasse da poter scegliere, ma solo una piccola e precisa zona, dovuto al fatto che ora non abbiamo due minimi ma molti di più. Prima di vedere il codice vediamo brevemente due grafici della quantità S^2, che con un po' di abuso di notazione chiamiamo chiquadro, nel caso di regressione lineare e non:

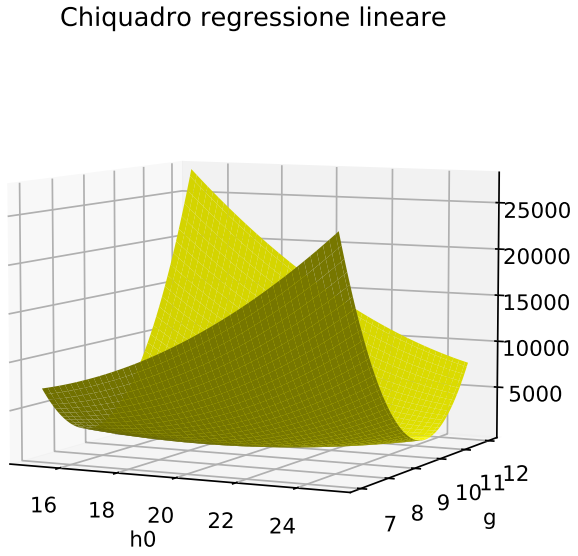


Figura 1: modello lineare $y(t) = h_0 - \frac{1}{2}gt^2$. Unico minimo, qualunque punto iniziale va bene.

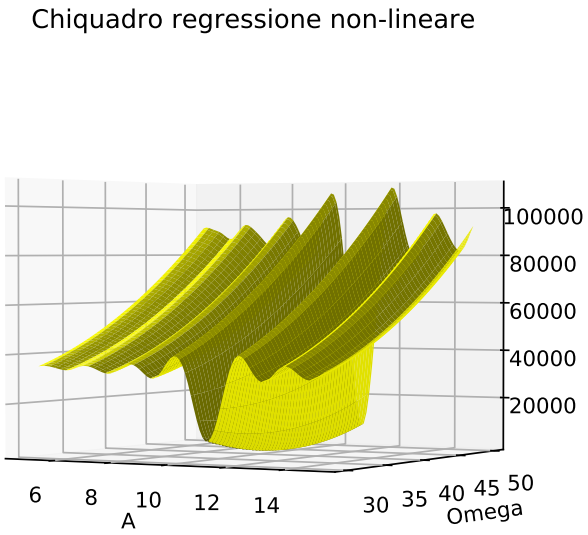
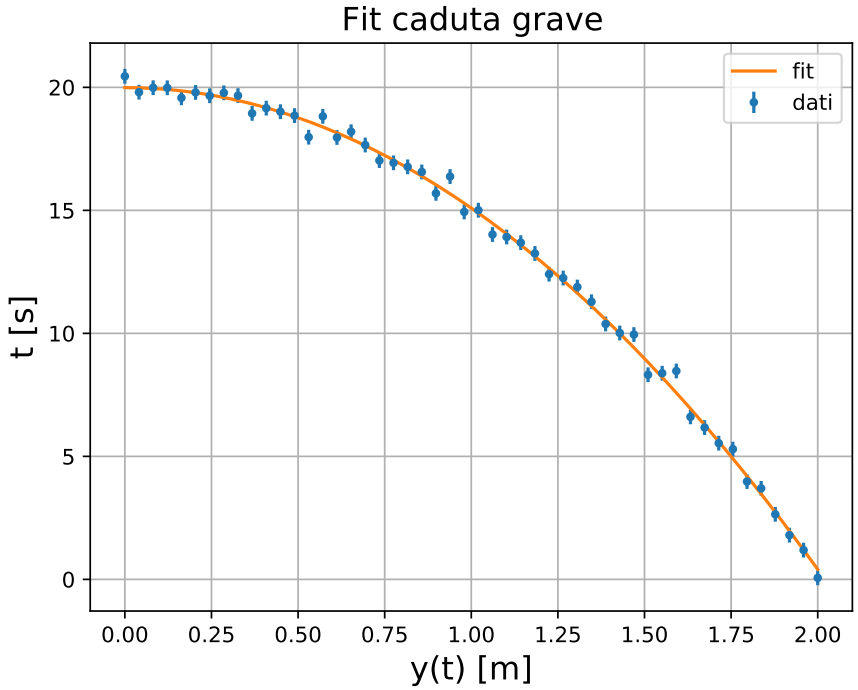


Figura 2: modello non lineare $y(t) = Acos(\omega t)$. Tanti minimi locali bisogna stare attante a dove partire altrimenti l'algoritmo si blocca su soluzioni non fisiche. Solo una piccola regione va bene come valori inziali.

I codici per generare i grafici che abbiamo visto non sono riportati per brevità ma sono presenti nella cartella. Vediamo ora un semplice esempio di codice:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def Legge_oraria(t, h0, g):
    """
    Restituisce la legge oraria di caduta
    di un corpo che parte da altezza h0 e
    con una velocita' inziale nulla
    """
    return h0 - 0.5*g*t**2

"""""
dati misurati:
xdata : fisicamemnte i tempi a cui osservo
        la caduta del corpo non affetti da
        errore
ydata : fisicamente la posizione del corpo
        misurata a dati tempi xdata afetta
        da errore
"""

#misuro 50 tempi tra 0 e 2 secondi
xdata = np.linspace(0, 2, 50)

#legge di caduta del corpo
y = Legge_oraria(xdata, 20, 9.81)
rng = np.random.default_rng()
y_noise = 0.3 * rng.normal(size=xdata.size)
#dati misurati afferri da errore
ydata = y + y_noise
dydata = np.array(ydata.size*[0.3])

#funzione che mi permette di vedere anche le barre d'errore
plt.errorbar(xdata, ydata, dydata, fmt='.', label='dati')

#array dei valori che mi aspetto, circa, di ottenere
init = np.array([15, 10])
#eseguo il fit
```

```
popt, pcov = curve_fit(Legge_oraria, xdata, ydata, init, sigma=dydata, absolute_sigma=False)

h0, g = popt
dh0, dg = np.sqrt(pcov.diagonal())
print(f'Altezza inziale h0 = {h0:.3f} +- {dh0:.3f}')
print(f"Accelerazione di gravita' g = {g:.3f} +- {dg:.3f}")

#garfico del fit
t = np.linspace(np.min(xdata), np.max(xdata), 1000)
plt.plot(t, Legge_oraria(t, *popt), label='fit')

plt.grid()
plt.title('Fit caduta grave', fontsize=15)
plt.xlabel('y(t) [m]', fontsize=15)
plt.ylabel('t [s]', fontsize=15)
plt.legend(loc='best')
plt.show()

[Output]
Altezza inziale h0 = 19.988 +- 0.065
Accelerazione di gravita' g = 9.790 +- 0.071
```



L'utilizzo dell'array init ci aiuta a trovare il minimo assoluto in modo che il codice vada a cercare intorno a quei valori, evitando che il codice si incastri altrove; anche se in questo caso non era necessario in quanto regressione lineare, è comunque buona norma utilizzarlo. Provate a fittare il modello non lineare visto sopra e vi accorgerete come solo una piccola regione dei parametri conduca alla soluzione coretta e che basti spostarvi di poco per ottenere risultati poco sensati.

8.3 Dietro curve fit: Levenberg-Marquardt

Vogliamo ora provare ad andare dietro la libreria e vedere cosa fa effettivamente curve fit. Chiaramente i metodi di fit implementati sono molti e diversi, a seconda delle esigenze; per semplicità perciò andiamo a vedere quello che viene usato di default: Levenberg-Marquardt. Questo è un metodo iterativo, il che spiega la sensibilità ai valori iniziali, caratteristica di ogni metodo iterativo. Consideriamo la nostra funzione di fit f la quale dipende da una variabile indipendente e da un insieme di parametri θ, il quale fondamentalmente è un vettore di $\mathbb{R}^m$. Possiamo espandere f in serie di taylor intorno ad un valore dei nostri parametri:

$$f(x_i, \theta_j + \delta_j) \simeq f(x_i, \theta_j) + J_{ij}\delta_j \tag{2}$$

dove δ_j è lo spostamento che vine fatto ad ogni passo dell'iterazione e J_{ij} è il gradiente di f, o jacobinao se volete:

$$J_{ij} = \frac{\partial f(x_i, \theta_j)}{\partial \theta_j} = \begin{bmatrix} \frac{\partial f(x_1, \theta_1)}{\partial \theta_1} & \cdots & \frac{\partial f(x_1, \theta_m)}{\partial \theta_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x_n, \theta_1)}{\partial \theta_1} & \cdots & \frac{\partial f(x_n, \theta_m)}{\partial \theta_m} \end{bmatrix} \tag{3}$$

Che è una matrice $m \times n$ con $m < n$ altrimenti il metodo non funziona e dobbiamo adottare altre strategie. Per trovare il valore di δ espandiamo la (1):

$$\begin{aligned} S^2(\theta + \delta) &\simeq \sum_{i=1}^{n} \frac{(y_i - f(x_i, \beta) - J_{ij}\delta_j)^2}{\sigma_{y_i}^2} \\ &= (y - f(x, \theta) - J\delta)^T W (y - f(x, \theta) - J\delta) \\ &= (y - f(x, \theta))^T W (y - f(x, \theta)) - (y - f(x, \theta))^T W J\delta - (J\delta)^T W (y - f(x, \theta)) + (J\delta)^T W (J\delta) \\ &= (y - f(x, \theta))^T W (y - f(x, \theta)) - 2(y - f(x, \theta))^T W J\delta + \delta^T J^T W (J\delta) \end{aligned} \tag{4}$$

Dove W è tale che $W_{ii} = 1/\sigma_{y_i}^2$ e derivando rispetto a delta otteniamo il metodo di Gauss-Newton:

$$\frac{\partial S^2(\theta + \delta)}{\partial \delta} = -2(y - f(x, \theta))^T W J + 2\delta^T J^T W J = 0 \tag{5}$$

per cui facendo il trasposto a tutto otteniamo:

$$(J^T W J)\delta = J^T W (y - f(x, \theta)) \tag{6}$$

La quale si risolve per δ. Per migliorare la convergenza del metodo si introduce un parametro di damping λ e l'equazione diventa:

$$(J^T W J - \lambda \operatorname{diag}(J^T W J))\delta = J^T W (y - f(x, \theta)) \tag{7}$$

Il valore di λ viene cambiato a seconda se ci avviciniamo o meno alla soluzione giusta. Se ci stiamo avvicinando ne riduciamo il valore, andando verso il metodo di Gauss-Newton; mentre se ci allontaniamo ne aumentiamo il valore in modo che l'algoritmo si comporti più come un gradiente discendente (di cui in appendice ci sarà un esempio). La domanda è: come capiamo se ci stiamo avvicinando alla soluzione? Calcoliamo:

$$\begin{aligned} \rho(\delta) &= \frac{S^2(x, \theta) - S^2(x, \theta + \delta)}{|(y - f(x, \theta) - J\delta)^T W (y - f(x, \theta) - J\delta)|} \\ &= \frac{S^2(x, \theta) - S^2(x, \theta + \delta)}{|\delta^T (\lambda \text{diag}(J.TWJ)\delta + J.TW(y - f(x, \theta)))|} \end{aligned} \tag{8}$$

se $\rho(\delta) > \varepsilon_1$ la mossa è accetta e riduciamo λ senno rimaniamo nella vecchia posizione. Altra domanda a cui rispondere è: quando siamo arrivati a convergenza? definiamo:

$$R1 = \max(|J.TW(y - f(x, \theta))|) \tag{9}$$

$$R2 = \max(|\delta/\theta|) \tag{10}$$

$$R3 = |S^2(x, \theta)/(n - m) - 1| \tag{11}$$

Se una di queste quantità è minore di una certa tolleranza allora l'algoritmo termina. Rimane ora un ultima domanda a cui rispondere e possiamo passare al codice. Dato che ci servono gli errori sui parametri di fit: come calcoliamo la matrice di covarianza? Basta calcolare:

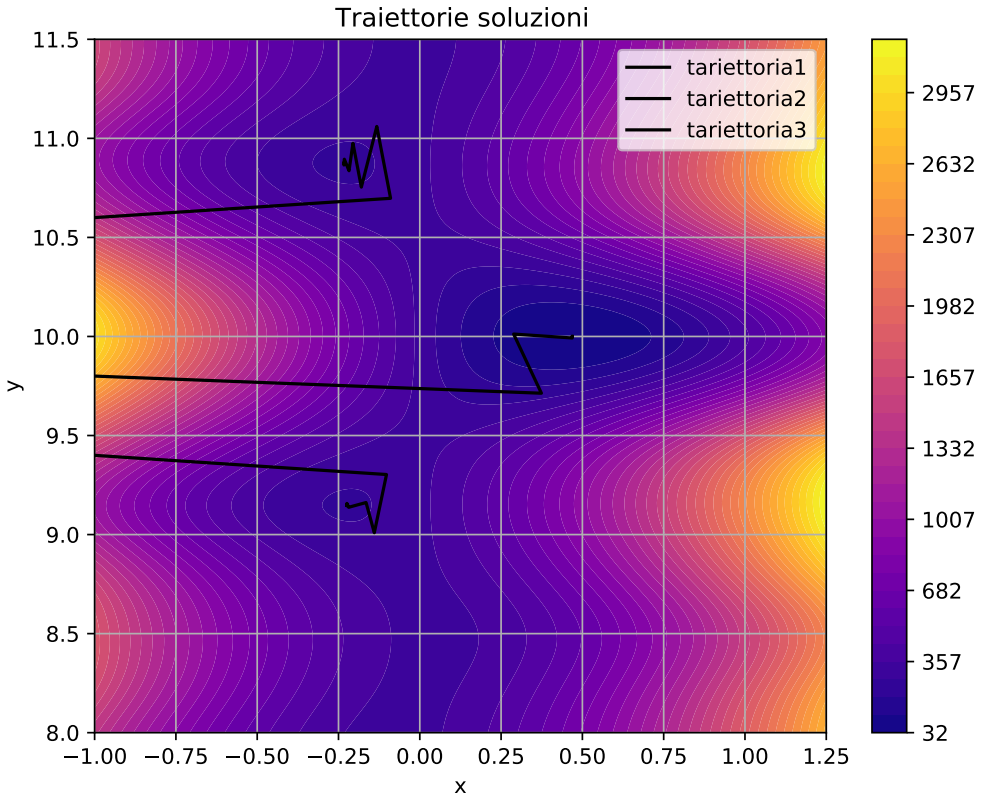
$$\text{Cov} = (J^T W J)^{-1} \tag{12}$$

quindi gli errori saranno semplicemente la radice degli elementi sulla diagonale, e le altre entrate le correlazioni fra parametri.
Passiamo ora al codice:

```
"""
the code performs a linear and non linear regression
Levenberg-Marquardt algorithm. You have to choose
some parameters delicately to make the result make sense
"""

import numpy as np
import matplotlib.pyplot as plt


def lm_fit(func, x, y, x0, sigma=None, tol=1e-6, dense_output=False, absolute_sigma=False):
    """
    Implementation of Levenberg-Marquardt algorithm
    for non-linear least squares. This algorithm interpolates
    between the Gauss-Newton algorithm and the method of
    gradient descent. It is iterative optimization algorithms
    so finds only a local minimum. So you have to be careful
    about the values you pass in x0

    Parameters
    ----------
    f : callable
        fit function
    x : 1darray
        the independent variable where the data is measured.
    y : 1darray
        the dependent data, y <= f(x, {\theta})
    x0 : 1darray
        initial guess
    sigma : None or 1darray
        the uncertainty on y, if None sigma=np.ones(len(y)))
    tol : float
        required tollerance, the algorithm stop if one of this quantities
        R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
        R2 = np.max(abs(d/x0))
        R3 = sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1
        is smaller than tol

    dense_output : bool, optional dafult False
        if true all iteration are returned
    absolute_sigma : bool, optional dafult False
        If True, sigma is used in an absolute sense and
        the estimated parameter covariance pcov reflects
        these absolute values.
        pcov(absolute_sigma=False) = pcov(absolute_sigma=True) * chisq(popt)/(M-N)

    Returns
    -------
    x0 : 1d array or ndarray
        array solution
    pcov : 2darray
        The estimated covariance of popt
    iter : int
        number of iteration
    """

    iter = 0                #initialize iteration counter
    h = 1e-7                #increment for derivatives
    l = 1e-3                #damping factor
    f = 10                  #factor for update damping factor
    M = len(x0)             #number of variable
    N = len(x)              #number of data
    s = np.zeros(M)         #auxiliary array for derivatives
    J = np.zeros((N, M))    #gradient
    #some trashold
    eps_1 = 1e-1
    eps_2 = tol
    eps_3 = tol
    eps_4 = tol

    if sigma is None :      #error on data
        W  = np.diag(1/np.ones(N))
        dy = np.ones(N)
    else :
        W  = np.diag(1/sigma**2)
        dy = sigma
```

```
77
78
79      if dense_output:       #to store solution
80          X = []
81          X.append(x0)
82
83      while True:
84          #jacobian computation
85          for i in range(M):                                      #loop over variables
86              s[i] = 1                                            #we select one variable at a time
87              dz1 = x0 + s*h                                      #step forward
88              dz2 = x0 - s*h                                      #step backward
89              J[:,i] = (func(x, *dz1) - func(x, *dz2))/(2*h)  #derivative along z's direction
90              s[:] = 0                                            #reset to select the other
    variables
91
92          JtJ = J.T @ W @ J                           #matrix multiplication, JtJ is an MxM matrix
93          dia = np.eye(M)*np.diag(JtJ)                #dia_ii = JtJ_ii ; dia_ij = 0
94          res = (y - func(x, *x0))                    #residuals
95          b   = J.T @ W @ res                         #ordinate or dependent variable values of
    system
96          d   = np.linalg.solve(JtJ + l*dia, b)       #system solution
97          x_n = x0 + d                                #solution at new time
98
99          # compute the metric
100         chisq_v = sum((res/dy)**2)
101         chisq_n = sum(((y - func(x, *x_n))/dy)**2)
102
103         rho = chisq_v - chisq_n
104         den = abs( d.T @ (l*np.diag(JtJ)@d + J.T @ W @ res))
105         rho = rho/den
106         # acceptance
107         if rho > eps_1 :          #if i'm closer to the solution
108             x0 = x_n              #update solution
109             l /= f                #reduce damping factor
110         else:
111             l *= f                #else magnify
112
113         # Convergence criteria
114         R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
115         R2 = np.max(abs(d/x0))
116         R3 = abs(sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1)
117
118         if R1 < eps_2 or R2 < eps_3 or R3 < eps_4:          #break condition
119             break
120
121         iter += 1
122
123         if dense_output:
124             X.append(x0)
125
126     #compute covariance matrix
127     pcov = np.linalg.inv(JtJ)
128
129     if not absolute_sigma:
130         s_sq = sum(((y - func(x, *x0))/dy)**2)/(N - M)
131         pcov = pcov * s_sq
132
133     if not dense_output:
134         return x0, pcov, iter
135     else :
136         X = np.array(X)
137         return X, pcov, iter
```

Il parametro dense_output è stato inserito per fare un plot interessante per far vedere la dipendenza dalle condizioni iniziali. Non riportiamo l'intero codice per non appesantire, la restante parte trattava solo di fare il plot delle curve di livello. In ogni caso è disponibile nell'apposita cartella il codice intero. Questo è il primo vero codice che fa qualcosa di molto complicato esso usa tutto quanto spiegato fin'ora e adesso è evidente l'importanza di mettere commenti, dare nomi sensati e rendere leggibile il codice. Bisogna ricordarsi che i codici in genere vengono scritti una volta ma letti tante volte quindi la chiarezza non va dosata con parsimonia.

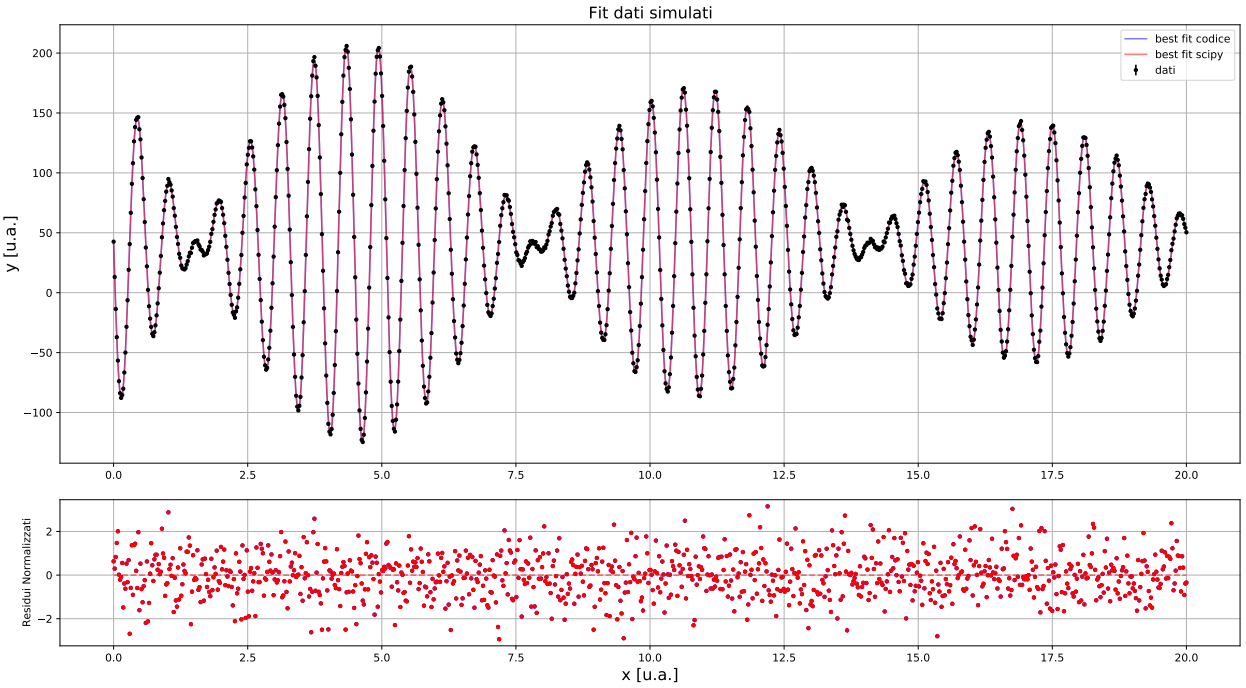


Questo grafico rappresenta le curve di livello del modello non lineare $y(t) = A\cos(\omega t)$ ed è facile vedere come partendo da condizioni diverse il fit si incastri in minimo locali. L'asse y corrisponde a ω mentre l'asse x corrisponde ad A. Vediamo ora di testare i risultati del codice fittando qualcosa di un po' più bruttino.

```
"""
Test
"""
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from Lev_Maq import lm_fit


def f(t, A, o1, o2, f1, f2, v, tau):
    """fit function
    """
    return A*np.cos(t*o1 + f1)*np.cos(t*o2 + f2)*np.exp(-t/tau) + v

##data
x = np.linspace(0, 20, 1000)
y = f(x, 200, 10.5, 0.5, np.pi/2, np.pi/4, 42, 25)
rng = np.random.default_rng(seed=69420)
y_noise = 1 * rng.normal(size=x.size)
y  = y + y_noise
dy = np.array(y.size*[1])

##confronto

init = np.array([101, 10.5, 0.475, 1.5, 0.6, 35, 20])

pars1, covm1, iter = lm_fit(f, x, y, init, sigma=dy, tol=1e-8)
dpar1 = np.sqrt(covm1.diagonal())
pars2, covm2 = curve_fit(f, x, y, init, sigma=dy)
dpar2 = np.sqrt(covm2.diagonal())
print("        ------codice---------|--------scipy----------")
for i, p1, dp1, p2, dp2 in zip(range(len(init)), pars1, dpar1, pars2, dpar2):
    print(f"pars{i} = {p1:.5f} +- {dp1:.5f} ; {p2:.5f} +- {dp2:.5f}")

print(f"numero di iterazioni = {iter}")

chisq1 = sum(((y - f(x, *pars1))/dy)**2.)
```

```
chisq2 = sum(((y - f(x, *pars2))/dy)**2.)
ndof = len(y) - len(pars1)
print(f'chi quadro codice = {chisq1:.3f} ({ndof:d} dof)')
print(f'chi quadro numpy  = {chisq2:.3f} ({ndof:d} dof)')


c1 = np.zeros((len(pars1),len(pars1)))
c2 = np.zeros((len(pars1),len(pars1)))
#Calcoliamo le correlazioni e le inseriamo nella matrice:
for i in range(0, len(pars1)):
    for j in range(0, len(pars1)):
       c1[i][j] = (covm1[i][j])/(np.sqrt(covm1.diagonal()[i])*np.sqrt(covm1.diagonal()[j]))
       c2[i][j] = (covm2[i][j])/(np.sqrt(covm2.diagonal()[i])*np.sqrt(covm2.diagonal()[j]))
#print(c1) #matrice di correlazione
#print(c2)

##Plot
#Grafichiamo il risultato
fig1 = plt.figure(1)
#Parte superiore contenetnte il fit:
frame1=fig1.add_axes((.1,.35,.8,.6))
#frame1=fig1.add_axes((trasla lateralmente, trasla verticamente, larghezza, altezza))
frame1.set_title('Fit dati simulati',fontsize=15)
plt.ylabel('y [u.a.]',fontsize=15)
plt.grid()


plt.errorbar(x, y, dy, fmt='.', color='black', label='dati') #grafico i punti
t = np.linspace(np.min(x), np.max(x), 10000)
plt.plot(t, f(t, *pars1), color='blue', alpha=0.5, label='best fit codice') #grafico del best
    fit
plt.plot(t, f(t, *pars2), color='red' , alpha=0.5, label='best fit scipy') #grafico del best
    fit scipy
plt.legend(loc='best')#inserisce la legenda nel posto migliorte

#Parte inferiore contenente i residui
frame2=fig1.add_axes((.1,.1,.8,.2))

#Calcolo i residui normalizzari
ff1 = (y - f(x, *pars1))/dy
ff2 = (y - f(x, *pars2))/dy
frame2.set_ylabel('Residui Normalizzati')
plt.xlabel('x [u.a.]',fontsize=15)

plt.plot(t, 0*t, color='red', linestyle='--', alpha=0.5) #grafico la retta costantemente zero
plt.plot(x, ff1, '.', color='blue') #grafico i residui normalizzati
plt.plot(x, ff2, '.', color='red') #grafico i residui normalizzati scipy
plt.grid()
plt.show()

[Output]
        ------codice---------|--------scipy----------
pars0 = 199.85504 +- 0.17712 ; 199.85504 +- 0.17712
pars1 = 10.50005 +- 0.00009 ; 10.50005 +- 0.00009
pars2 = 0.49990 +- 0.00008 ; 0.49990 +- 0.00008
pars3 = 1.57040 +- 0.00087 ; 1.57040 +- 0.00087
pars4 = 0.78579 +- 0.00067 ; 0.78579 +- 0.00067
pars5 = 41.92350 +- 0.03125 ; 41.92350 +- 0.03125
pars6 = 24.99194 +- 0.05652 ; 24.99194 +- 0.05652
numero di iterazioni = 6
chi quadro codice = 969.017 (993 dof)
chi quadro numpy  = 969.017 (993 dof)
```

Non abbiamo stampato la matrice di covarianza per avere un po' più di ordine. Vediamo che tra i due non ci sono differenze, siamo felici. Vediamo anche il grafico:


Fit dati simulati
best fit codice
best fit scipy
dati
y [u.a.]
Residui Normalizzati
x [u.a.]