

7 Quarta lezione

7.1 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:

```
1 def f(x, n):
2     """
3     restituisce la potenza n-esima di un numero x
4     Parametri
5     -----
6     x, n : float
7
8     Return
9     -----
10    v : float
11        x**n
12    """
13
14    v = x**n
15
16    return v
17
18 if __name__ == '__main__':
19     #test
20     print(f(5, 2))
21
22 [Output]
23 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`:

```
1 import elevamento
2
3 print(elevamento.f(3, 3))
4
5 [Output]
6 27
```

Notiamo nel codice iniziale la presenza dell' `if`, esso serve per far sì 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.

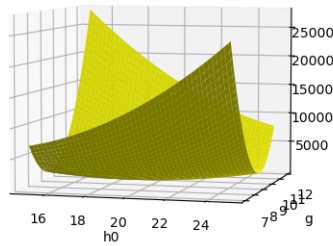
7.2 Fit

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 legge 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). Nella nostra pigrizia deleghiamo tutto il da fare alla funzione "`scipy.optimize.curve_fit()`". In ogni caso comunque l'idea di ciò che va fatto è trovare il minimo della seguente funzione:

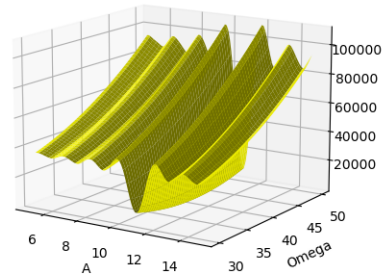
$$S^2(\{\theta_i\}) = \sum_i \frac{(y_i - f(x_i; \{\theta_j\}))^2}{\sigma_{y_i}^2} \quad (1)$$

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. Il problema della non linearità fondamentalmente si può esprimere nell'esistenza di minimi locali che potrebbero bloccare il fit dando valori per i parametri θ_i non realistici; mentre per una regressione lineare il minimo è solo uno e assoluto. 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:

Chiquadro regressione lineare



Chiquadro regressione non-lineare



modello: $y(t) = h_0 - \frac{1}{2}gt^2$

modello: $y(t) = A \cos(\omega t)$

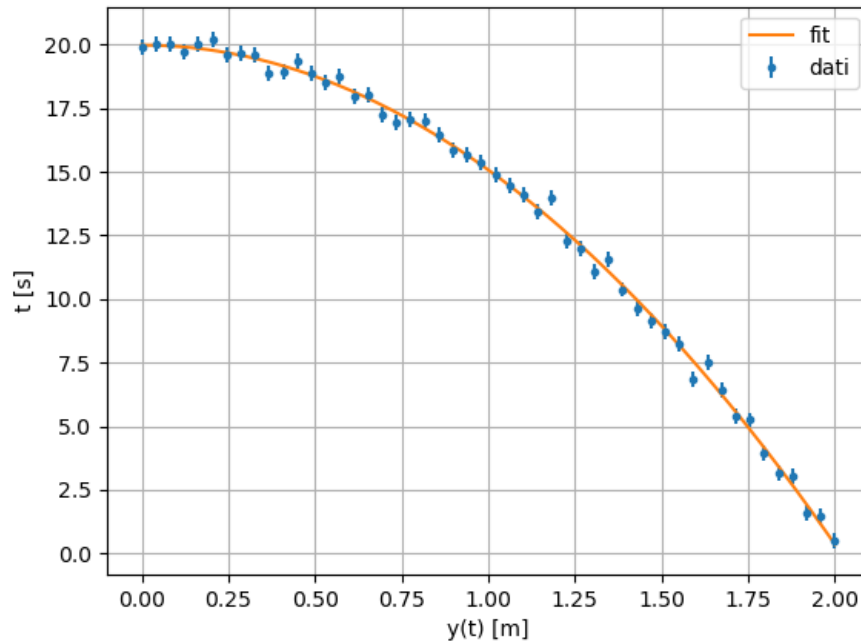
Vediamo come effettivamente siano presenti nel caso non lineare una serie di minimi locali che sarebbe meglio evitare (i codici per la realizzazione di grafici sono riportati a fine sezione). Vediamo ora un semplice esempio di codice:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4
5 def Legge_oraria(t, h0, g):
6     """
7     Restituisce la legge oraria di caduta
8     di un corpo che parte da altezza h0 e
9     con una velocita' iniziale nulla
10    """
11    return h0 - 0.5*g*t**2
12
13 """
14 dati misurati:
15 xdata : fisicamemnte i tempi a cui osservo
16         la caduta del corpo non affetti da
17         errore
18 ydata : fisicamente la posizione del corpo
19         misurata a dati tempi xdata afetta
20         da errore
21 """
22
23 #misuro 50 tempi tra 0 e 2 secondi
24 xdata = np.linspace(0, 2, 50)
25
26 #legge di caduta del corpo
27 y = Legge_oraria(xdata, 20, 9.81)
28 rng = np.random.default_rng()
29 y_noise = 0.3 * rng.normal(size=xdata.size)
30 #dati misurati afferri da errore
31 ydata = y + y_noise
32 dydata = np.array(len(ydata)*[0.3])
33
34 #funzione che mi permette di vedere anche le barre d'errore
35 plt.errorbar(xdata, ydata, dydata, fmt='.', label='dati')
36
37 #array dei valori che mi aspetto, circa, di ottenere
38 init = np.array([15, 10])
39 #eseguo il fit
40 popt, pcov = curve_fit(Legge_oraria, xdata, ydata, init, sigma=dydata, absolute_sigma=False)
41
42 h0, g = popt
43 dh0, dg = np.sqrt(pcov.diagonal())
44 print(f'Altezza iniziale h0 = {h0:.3f} +- {dh0:.3f}')
45 print(f"Accelerazione di gravita' g = {g:.3f} +- {dg:.3f}")
46
47 #garfico del fit
48 t = np.linspace(np.min(xdata), np.max(xdata), 1000)
49 plt.plot(t, Legge_oraria(t, *popt), label='fit')
50
51 plt.grid()
52 plt.xlabel('y(t) [m]')
```

```

53 plt.ylabel('t [s]')
54 plt.legend(loc='best')
55 plt.show()
56
57 [Output]
58 Altezza iniziale h0 = 19.975 +- 0.064
59 Accelerazione di gravita' g = 9.810 +- 0.070

```



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. Di seguito riportiamo i codici usati per costruire i grafici del chiquadro mostrati sopra:

Caso lineare

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def Legge_oraria(t, h0, g):
5     """
6     Restituisce la legge oraria di caduta
7     di un corpo che parte da altezza h0 e
8     con una velocita' iniziale nulla
9     """
10    return h0 - 0.5*g*t**2
11
12 """
13 dati misurati:
14 xdata : fisicamemnte i tempi a cui osservo
15         la caduta del corpo non affetti da
16         errore
17 ydata : fisicamente la posizione del corpo
18         misurata a dati tempi xdata afetta
19         da errore
20 """
21
22 #misuro 50 tempi tra 0 e 2 secondi
23 xdata = np.linspace(0, 2, 50)
24
25 #legge di caduta del corpo
26 y = Legge_oraria(xdata, 20, 9.81)
27 rng = np.random.default_rng()
28 y_noise = 0.3 * rng.normal(size=xdata.size)
29 #dati misurati afferri da errore

```

```

30 ydata = y + y_noise
31 dydata = np.array(ydata.size*[0.3])
32
33 N = 100
34 S2 = np.zeros((N, N))
35 h0 = np.linspace(15, 25, N)
36 g = np.linspace(7, 12, N)
37 for i in range(N):
38     for j in range(N):
39         S2[i, j] = (((ydata - Legge_oraria(xdata, h0[i], g[j]))/dydata)**2).sum()
40
41 #grafico del chi quadro
42 fig = plt.figure(1)
43 gridx, gridy = np.meshgrid(h0, g)
44 ax = fig.add_subplot(projection='3d')
45 ax.plot_surface(gridx, gridy, S2, color='yellow')
46 ax.set_title('Chiquadro regressione lineare')
47 ax.set_xlabel('h0')
48 ax.set_ylabel('g')
49 plt.show()

```

Caso non lineare

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4
5 def Legge_oraria(t, A, omega):
6     """
7     Restituisce la legge oraria di un corpo che
8     oscilla con ampiezza A e frequenza omega
9     """
10    return A*np.cos(omega*t)
11
12    """
13    dati misurati:
14    xdata : fisicamente i tempi a cui osservo
15            l'oscillazione del corpo non
16            affetti da errore
17    ydata : fisicamente la posizione del corpo
18            misurata a dati tempi xdata affetta
19            da errore
20    """
21
22    #misuro 50 tempi tra 0 e 2 secondi
23    xdata = np.linspace(0, 2, 50)
24
25    #legge di oscillazione del corpo
26    y = Legge_oraria(xdata, 10, 42)
27    rng = np.random.default_rng()
28    y_noise = 0.3 * rng.normal(size=xdata.size)
29    #dati misurati affetti da errore
30    ydata = y + y_noise
31    dydata = np.array(ydata.size*[0.3])
32
33    N = 100
34    S2 = np.zeros((N, N))
35    A = np.linspace(5, 15, N)
36    O = np.linspace(30, 50, N)
37    for i in range(N):
38        for j in range(N):
39            S2[i, j] = (((ydata - Legge_oraria(xdata, A[i], O[j]))/dydata)**2).sum()
40
41 #grafico chiquadro
42 fig = plt.figure(1)
43 gridx, gridy = np.meshgrid(A, O)
44 ax = fig.add_subplot(projection='3d')
45 ax.plot_surface(gridx, gridy, S2, color='yellow')
46 ax.set_title('Chiquadro regressione non-lineare')
47 ax.set_xlabel('A')
48 ax.set_ylabel('Oomega')
49 plt.show()

```

7.3 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:

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

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

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

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):

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

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

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

per cui facendo il trasposto a tutto otteniamo:

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

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

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

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:

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

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

$$R1 = \max(|J^T W (y - f(x, \theta))|) \quad (9)$$

$$R2 = \max(|\delta/\theta|) \quad (10)$$

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

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:

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

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

Passiamo ora al codice:

```

1  """
2  the code performs a linear and non linear regression
3  Levenberg-Marquardt algorithm. You have to choose
4  some parameters delicately to make the result make sense
5  """
6
7  import numpy as np
8  import matplotlib.pyplot as plt
9
10
11 def lm_fit(func, x, y, x0, sigma=None, tol=1e-6, dense_output=False, absolute_sigma=False):
12     """
13     Implementation of Levenberg-Marquardt algorithm
14     for non-linear least squares. This algorithm interpolates
15     between the Gauss-Newton algorithm and the method of
16     gradient descent. It is iterative optimization algorithms
17     so finds only a local minimum. So you have to be careful
18     about the values you pass in x0
19
20     Parameters
21     -----
22     f : callable
23         fit function
24     x : 1darray
25         the independent variable where the data is measured.
26     y : 1darray
27         the dependent data, y <= f(x, {\theta})
28     x0 : 1darray
29         initial guess
30     sigma : None or 1darray
31         the uncertainty on y, if None sigma=np.ones(len(y))
32     tol : float
33         required tollerance, the algorithm stop if one of this quantities
34         R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
35         R2 = np.max(abs(d/x0))
36         R3 = sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1
37         is smaller than tol
38
39     dense_output : bool, optional dafult False
40         if true all iteration are returned
41     absolute_sigma : bool, optional dafult False
42         If True, sigma is used in an absolute sense and
43         the estimated parameter covariance pcov reflects
44         these absolute values.
45         pcov(absolute_sigma=False) = pcov(absolute_sigma=True) * chisq(popt)/(M-N)
46
47     Returns
48     -----
49     x0 : 1d array or ndarray
50         array solution
51     pcov : 2darray
52         The estimated covariance of popt
53     iter : int
54         number of iteration
55     """
56
57     iter = 0           #initialize iteration counter
58     h = 1e-7          #increment for derivatives
59     l = 1e-3          #damping factor
60     f = 10            #factor for update damping factor
61     M = len(x0)        #number of variable
62     N = len(x)         #number of data
63     s = np.zeros(M)    #auxiliary array for derivatives
64     J = np.zeros((N, M)) #gradient
65     #some trashold
66     eps_1 = 1e-1
67     eps_2 = tol
68     eps_3 = tol
69     eps_4 = tol
70
71     if sigma is None :    #error on data
72         W = np.diag(1/np.ones(N))
73         dy = np.ones(N)
74     else :
75         W = np.diag(1/sigma**2)
76         dy = sigma

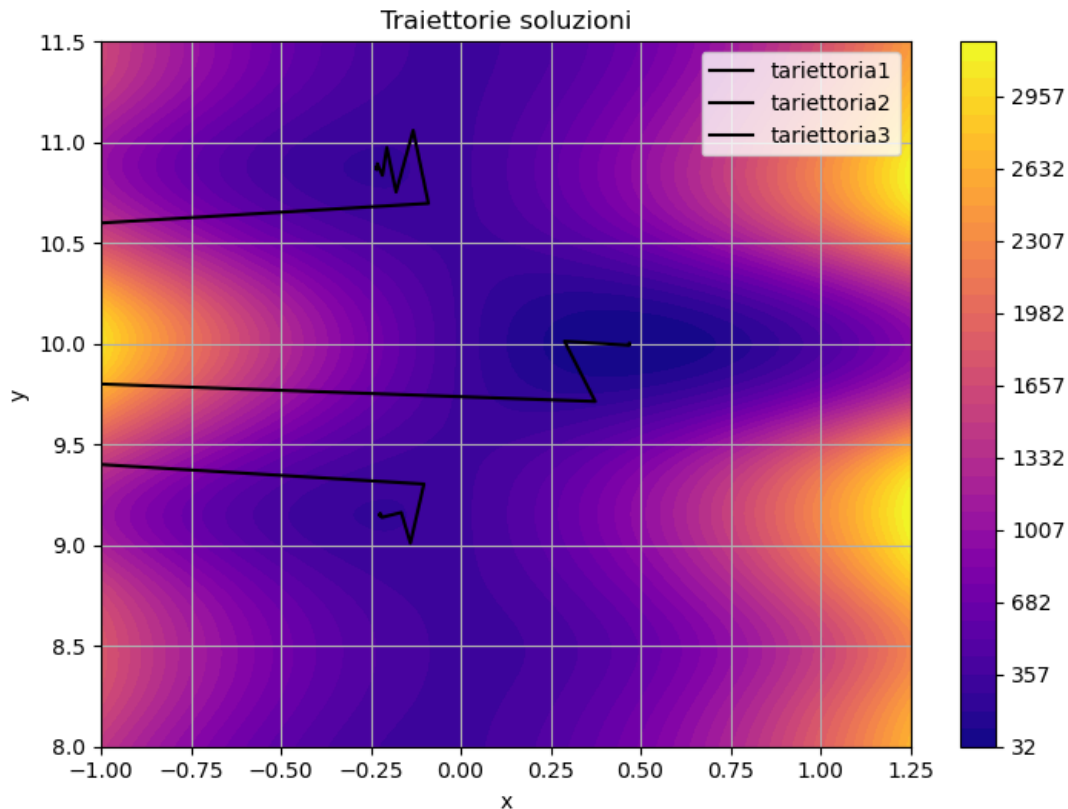
```

```

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
91     variables
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
96     system
97     d = np.linalg.solve(JtJ + l*dia, b) #system solution
98     x_n = x0 + d                #solution at new time
99
100     # compute the metric
101     chisq_v = sum((res/dy)**2)
102     chisq_n = sum(((y - func(x, *x_n))/dy)**2)
103
104     rho = chisq_v - chisq_n
105     den = abs( d.T @ (l*np.diag(JtJ)*d + J.T @ W @ res))
106     rho = rho/den
107     # acceptance
108     if rho > eps_1 :             #if i'm closer to the solution
109         x0 = x_n                #update solution
110         l /= f                   #reduce damping factor
111     else:
112         l *= f                  #else magnify
113
114     # Convergence criteria
115     R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
116     R2 = np.max(abs(d/x0))
117     R3 = abs(sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1)
118
119     if R1 < eps_2 or R2 < eps_3 or R3 < eps_4:    #break condition
120         break
121
122     iter += 1
123
124     if dense_output:
125         X.append(x0)
126
127     #compute covariance matrix
128     pcov = np.linalg.inv(JtJ)
129
130     if not absolute_sigma:
131         s_sq = sum(((y - func(x, *x0))/dy)**2)/(N - M)
132         pcov = pcov * s_sq
133
134     if not dense_output:
135         return x0, pcov, iter
136     else :
137         X = np.array(X)
138         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 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 incastra 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.

```

1  """
2  Test
3  """
4  import numpy as np
5  import matplotlib.pyplot as plt
6  from scipy.optimize import curve_fit
7  from Lev_Maqq import lm_fit
8
9
10 def f(t, A, o1, o2, f1, f2, v, tau):
11     """fit function
12     """
13     return A*np.cos(t*o1 + f1)*np.cos(t*o2 + f2)*np.exp(-t/tau) + v
14
15 ##data
16 x = np.linspace(0, 20, 1000)
17 y = f(x, 200, 10.5, 0.5, np.pi/2, np.pi/4, 42, 25)
18 rng = np.random.default_rng(seed=69420)
19 y_noise = 1 * rng.normal(size=x.size)
20 y = y + y_noise
21 dy = np.array(y.size*[1])
22
23 ##confronto
24
25 init = np.array([101, 10.5, 0.475, 1.5, 0.6, 35, 20])
26
27 pars1, covm1, iter = lm_fit(f, x, y, init, sigma=dy, tol=1e-8)
28 dpar1 = np.sqrt(covm1.diagonal())
29 pars2, covm2 = curve_fit(f, x, y, init, sigma=dy)
30 dpar2 = np.sqrt(covm2.diagonal())
31 print("-----codice-----|-----scipy-----")
32 for i, p1, dp1, p2, dp2 in zip(range(len(init)), pars1, dpar1, pars2, dpar2):
33     print(f"pars{i} = {p1:.5f} +- {dp1:.5f} ; {p2:.5f} +- {dp2:.5f}")
34
35 print(f"numero di iterazioni = {iter}")
36

```



```

37 chisq1 = sum(((y - f(x, *pars1))/dy)**2.)
38 chisq2 = sum(((y - f(x, *pars2))/dy)**2.)
39 ndof = len(y) - len(pars1)
40 print(f'chi quadro codice = {chisq1:.3f} ({ndof:d} dof)')
41 print(f'chi quadro numpy = {chisq2:.3f} ({ndof:d} dof)')
42
43
44 c1 = np.zeros((len(pars1),len(pars1)))
45 c2 = np.zeros((len(pars1),len(pars1)))
46 #Calcoliamo le correlazioni e le inseriamo nella matrice:
47 for i in range(0, len(pars1)):
48     for j in range(0, len(pars1)):
49         c1[i][j] = (covm1[i][j])/(np.sqrt(covm1.diagonal()[i])*np.sqrt(covm1.diagonal()[j]))
50         c2[i][j] = (covm2[i][j])/(np.sqrt(covm2.diagonal()[i])*np.sqrt(covm2.diagonal()[j]))
51 #print(c1) #matrice di correlazione
52 #print(c2)
53
54 ##Plot
55 #Grafichiamo il risultato
56 fig1 = plt.figure(1)
57 #Parte superiore contenente il fit:
58 frame1=fig1.add_axes((.1,.35,.8,.6))
59 #frame1=fig1.add_axes((trasla lateralmente, trasla verticalmente, larghezza, altezza))
60 frame1.set_title('Fit dati simulati',fontsize=10)
61 plt.ylabel('y [u.a.]',fontsize=10)
62 plt.grid()
63
64
65 plt.errorbar(x, y, dy, fmt='.', color='black', label='dati') #grafico i punti
66 t = np.linspace(np.min(x), np.max(x), 10000)
67 plt.plot(t, f(t, *pars1), color='blue', alpha=0.5, label='best fit codice') #grafico del best
68     fit
69 plt.plot(t, f(t, *pars2), color='red', alpha=0.5, label='best fit scipy') #grafico del best
70     fit scipy
71 plt.legend(loc='best')#inserisce la legenda nel posto migliore
72
73 #Parte inferiore contenente i residui
74 frame2=fig1.add_axes((.1,.1,.8,.2))
75
76 #Calcolo i residui normalizzati
77 ff1 = (y - f(x, *pars1))/dy
78 ff2 = (y - f(x, *pars2))/dy
79 frame2.set_ylabel('Residui Normalizzati')
80 plt.xlabel('x [u.a.]',fontsize=10)
81
82 plt.plot(t, 0*t, color='red', linestyle='--', alpha=0.5) #grafico la retta costantemente zero
83 plt.plot(x, ff1, '.', color='blue') #grafico i residui normalizzati
84 plt.plot(x, ff2, '.', color='red') #grafico i residui normalizzati scipy
85 plt.grid()
86 plt.show()
87
88 [Output]
89
90 -----codice-----|-----scipy-----
91 pars0 = 199.85504 +- 0.17712 ; 199.85504 +- 0.17712
92 pars1 = 10.50005 +- 0.00009 ; 10.50005 +- 0.00009
93 pars2 = 0.49990 +- 0.00008 ; 0.49990 +- 0.00008
94 pars3 = 1.57040 +- 0.00087 ; 1.57040 +- 0.00087
95 pars4 = 0.78579 +- 0.00067 ; 0.78579 +- 0.00067
96 pars5 = 41.92350 +- 0.03125 ; 41.92350 +- 0.03125
97 pars6 = 24.99194 +- 0.05652 ; 24.99194 +- 0.05652
98 numero di iterazioni = 6
99 chi quadro codice = 969.017 (993 dof)
100 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:

