Nel seguito delle note saranno presenti codici in dei riquadri e, per completezza, dopo la riga [Output] viene presentato anche il risultato degli stessi nel caso ci fossero (i.e. ciò che viene stampato su shell).

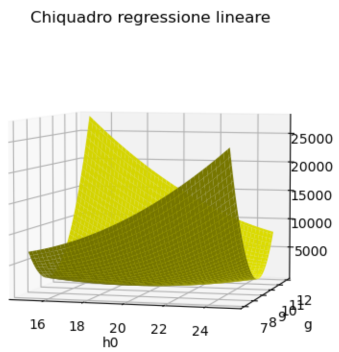
1 Quarta lezione

1.1 Fit

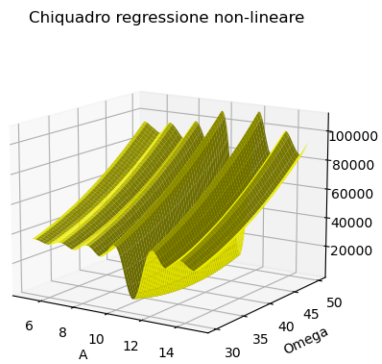
Nell'ambito della statistica un fit, cioè una regressione lineare o non che sia (dove la linearità è riferita ai parametri della funzione), è un metodo per trovare la funzione che meglio descrive l'andamento di alcuni dati. Nel caso di regressione lineare La procedura da eseguire non è troppo complicata, 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). Nella nostra pigrizia deleghiamo tutto il da fare alla funzione "scipy.optimaze.curve_fit()". In ogni caso comunque l'idea di ciò che va fatto è trovare il minimo della seguente funzione:

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

che nel caso in chi 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:



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

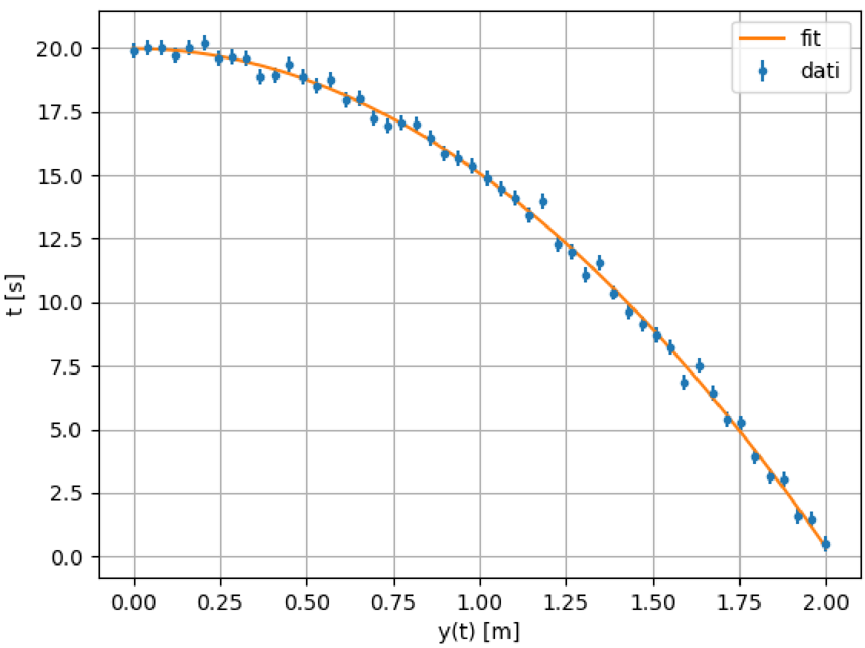


modello:$y(t) = Acos(\omega t)$

Vediamo come effettivamente siano presenti nel caso non lineare una serie di minimi locali che sarebbe meglio evitare. 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(len(ydata)*[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.xlabel('y(t) [m]')
plt.ylabel('t [s]')
plt.legend(loc='best')
plt.show()

[Output]
Altezza inziale h0 = 19.975 +- 0.064
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.

1.2 Risolvere numericamente le ODE

In fisica è prassi che spuntino fuori equazioni differenziali che non ammettano soluzione analitica; piuttosto che lamentarci di questo ringraziamo quando ciò capita con le ODE, cioè le equazione differenziali ordinarie, perché spesso e volentieri madre natura preferisce l'utilizzo delle equazioni differenziali alle derivate parziali(dette PDE) che in genere da risolvere sono abbastanza più complicate. Qui vedremo semplici esempi per risolvere un'ode. I metodi mostrati saranno per brevità solo due: l'utilizzo delle funzione "odeint()" di scipy e il metodo di eulero, basato sulla definizione di derivata, che mostriamo brevemente: sia $\frac{df(t)}{dt} = g(t, f(t))$ l'equazione da risolvere, allora:

$$\frac{df}{dt} \xrightarrow{discretizzando} \frac{f(t+dt) - f(t)}{dt}$$

Dove la forma ottenuta discretizzando non è altro che il rapporto incrementale di $f(t)$, di cui per definizione la derivata ne è il limite per $dt \to 0$. Sapendo la forma funzionale della derivata, ovvero la $g(t, f(t)$, data dall'equazione differenziale, possiamo ottenere la soluzione dell'equazione per passi:

$$f(t+dt) = f(t) + dtg(t, f(t))$$

quindi possiamo trovare la soluzione al tempo t+dt, sapendo quella al tempo t. inoltre nella g non compare la dipendenza da $f(t+dt)$ ma solo da $f(t)$, per questo il metodo è chiamato esplicito.

1.2.1 Esponenziale

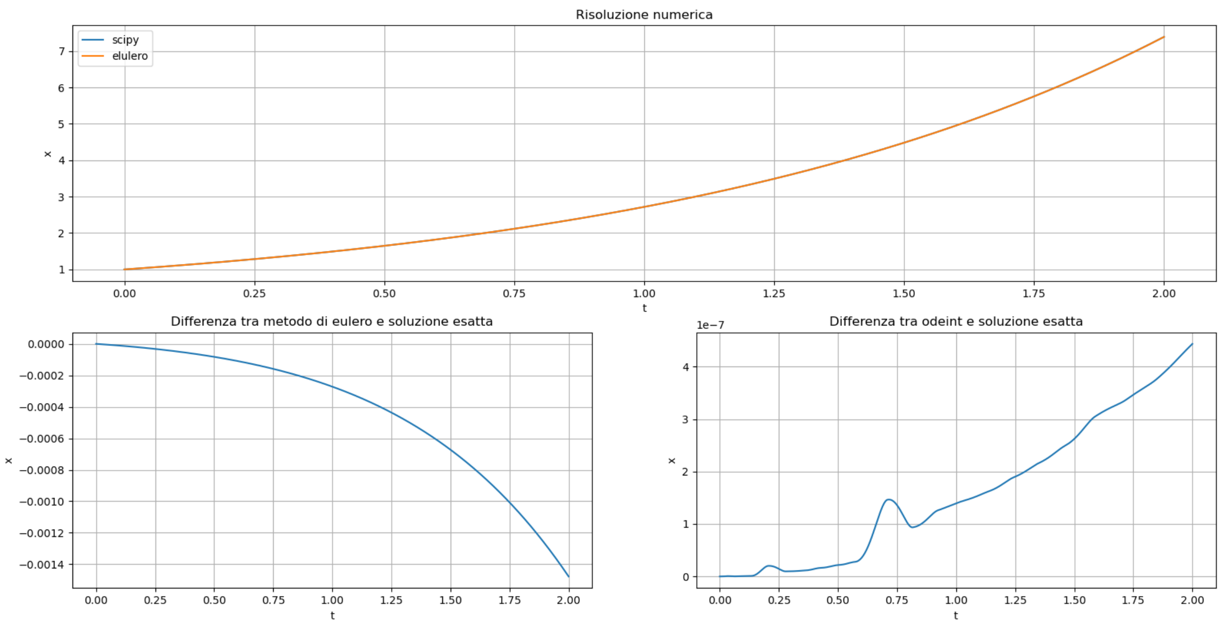
Cominciamo con il problema di Cauchy:

$$\begin{cases} \frac{dx(t)}{dt} = x(t) \\ x(t=0) = 1 \end{cases}$$

Abbiamo Una funzione incognita x(t) di cui sapiamo che la derivata è uguale a se stessa e che calcolata in zero restituisce uno. Nella fattispecie la soluzione è semplice, si stratta di un esponenziale crescente, tuttavia vediamo come risolvere numericamente tale equazione.

```
import numpy as np
import scipy.integrate
import  matplotlib.pyplot as plt

#parametri
x0 = 1      #condizione inizale
tf = 2      #fino a dove integrare
N = 10000   #numero di punti

#odeint
def ODE_1(y, t):
    """
    equzione da risolvere per odeint
    """
    x = y
    dydt = x
    return dydt


y0 = [x0] #x(0)
t = np.linspace(0, tf, N+1)
sol = scipy.integrate.odeint(ODE_1, y0, t)

x_scipy = sol[:,0]

#metodo di eulero
def ODE_2(x):
    """
    equzione da risolvere per eulero
    """
    x_dot = x
    return x_dot

def eulero(N, tf, x0):
    """
    si usa che dx/dt = (x[i+1]-x[i])/dt
    che e' praticamente la definizione di rapporto incrementale
    discretizzata la derivata sappiamo a cosa eguagliarla
    perche dx/dt = g(x(t)) nella fattispecie g(x) = x
    quindi discretizzando tutto:
    (x[i+1]-x[i])/dt = x[i]
```

```
    da cui si isola x[i+1]
    """
    dt = tf/N #passo di integrazione
    x = np.zeros(N+1)
    x[0] = x0

    for i in range(N):
        x[i+1] = x[i] + dt*ODE_2(x[i])

    return x

x_eulero = eulero(N, tf, x0)

plt.figure(1)

ax1 = plt.subplot(211)
ax1.set_title('Risoluzione numerica')
ax1.set_xlabel('t')
ax1.set_ylabel('x')
ax1.plot(t, x_scipy, label='scipy')
ax1.plot(t, x_eulero, label='elulero')
ax1.legend(loc='best')
ax1.grid()

ax2 = plt.subplot(223)
ax2.set_title('Differenza tra metodo di eulero e soluzione esatta')
ax2.set_xlabel('t')
ax2.set_ylabel('x')
ax2.plot(t, x_eulero-np.exp(t))
ax2.grid()


ax3 = plt.subplot(224)
ax3.set_title('Differenza tra odeint e soluzione esatta')
ax3.set_xlabel('t')
ax3.set_ylabel('x')
ax3.plot(t, x_scipy-np.exp(t))
ax3.grid()

plt.show()
```



Vediamo che entrambi i metodi sembrano funzionare bene, scipy usa un integratore migliore rispetto ad eulero infatti vediamo che la differenza fra le due soluzioni e dell'ordine di 10^{-7}, ma costruirne uno analogo non è difficile, si può provare con i metodi di Runge-kutta; famoso e molto usato è quello di ordine 4. Abbiamo risolto un'ode del primo ordine, e per ordine più elevati la cosa è analoga perché con cambi di variabili si può abbassare l'ordine fino ad ottenere un sistema di ode accoppiate di ordine 1;