

Fisica computazionale: Relazione 2

Simone Rubiu

July 10, 2022

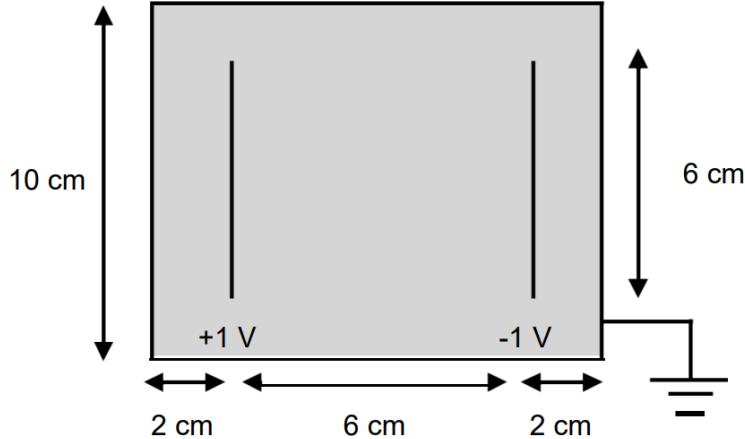
1 Condensatore 2D

1.1 Introduzione

Quando si ha una regione dello spazio in cui è presente un campo elettrico conservativo si può definire un potenziale scalare. Il potenziale può essere usato per calcolare il campo elettrico attraverso la relazione $\vec{E} = -\nabla V$. Per calcolare il potenziale si può risolvere l'equazione differenziale ellittica

$$v_{xx} + v_{yy} = \rho(x, y) \quad (1)$$

Il problema studiato con la simulazione consiste in un condensatore in 2 dimensioni all'interno di una scatola quadrata. Le due piastre sono rispettivamente a +1 e -1 volt, mentre i bordi della scatola si trovano a tensione zero volt (condizioni di Dirichlet).



1.2 Algoritmo SOR

Per risolvere l'equazione di poisson si usano i metodi iterativi, che permettono di trovare una soluzione in poco tempo. Quindi discretizzando l'equazione di poisson si ottiene

$$v_{xx} + v_{yy} + \rho(x, y) = v_t \quad (2)$$

Usando poi le formule FTCS si ottiene l'iterazione di Gauss-Jacobi, ovvero la funzione in un dato punto all'iterazione successiva è la media della funzione nei punti laterali.

$$v_{ij}^{n+1} = \frac{1}{4}[v_{i-1,j}^n + v_{i+1,j}^n + v_{i,j-1}^n + v_{i,j+1}^n + h^2 \rho_{ij}] \quad (3)$$

Un metodo per accelerare la convergenza è quello di Gauss-Seidel. Si ottiene osservando che quando, durante la scansione della griglia all'iterazione $n + 1$, si arriva ad aggiornare la funzione nel punto (i, j) , la soluzione è già quella nuova nei punti $(i, j - 1)$ e $(i - 1, j)$, mentre è quella vecchia negli altri due punti $(i + 1, j)$ e $(i, j + 1)$, oltre che nel punto (i, j) stesso.

$$v_{ij}^{n+1} = \frac{1}{4}[v_{i-1,j}^{n+1} + v_{i+1,j}^n + v_{i,j-1}^{n+1} + v_{i,j+1}^n + h^2 \rho_{ij}] \quad (4)$$

Infine si arriva al metodo iterativo più veloce tra i due, ovvero il SOR che consiste tirare ulteriormente la soluzione attuale verso la convergenza sovrapesando il termine di update al valore attuale:

$$v_{ij}^{n+1} = \frac{\omega}{4}[v_{i-1,j}^n + v_{i+1,j}^n + v_{i,j-1}^n + v_{i,j+1}^n + h^2 \rho_{ij}] + (1 - \omega)v_{ij}^n \quad (5)$$

dove ω misura il sovrarilassamento.

- $\omega = 0$: non succede niente perché $v_{ij}^{n+1} = v_{ij}^n$;
- $\omega < 1$: sottilassamento; la parte di update è minore di quella di Seidel e si reintroduce un po' della funzione all'istante precedente nella funzione all'istante successivo;
- ω compreso tra 1 e 2: sovrarilassamento, perchè si sottrae una porzione della funzione v_{ij}^n , e si sovrapesa la parte di update;
- $\omega = 2$: divergenza.

Si è quindi implementato l'algoritmo SOR in python per il calcolo del potenziale in una griglia quadrata:

```
[ ]: from cmath import sqrt
from turtle import color
from matplotlib import projections
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import scipy.constants as spc
from matplotlib import cm
from scipy.optimize import curve_fit
```

```
[ ]: # Funzione usata per controllare che si è al bordo della scatola
def IsOverBoundary(i, j, N, boundary):
    return i == 0 or j == 0 or i == N-1 or j == N-1
```

```
[ ]: # Funzione con le condizioni al contorno e algoritmo SOR per il calcolo del potenziale
def SORAlg(f, rho, h, w, i, j, N, boundary, pos1, pos2):
    # La lunghezza della piastra è data dalla differenza tra fine e inizio
    inizio = int((1/5) * N)
```

```

fine = int((4/5) * N)

if(IsOverBoundary(i, j, N, boundary)):
    return 0

# Identifica la posizione lungo le x della prima piastra e
# pone a 1 il potenziale nei punti che vanno da inizio a fine
if(j == pos1 and i <= fine and i >= inizio):
    return 1

# Identifica la posizione della seconda piastra e
# pone a -1 il potenziale nei punti che vanno da inizio a fine
if(j == pos2 and i <= fine and i >= inizio):
    return -1

# Calcola il nuovo potenziale nel punto conoscendo il vecchio potenziale
# nei punti attorno
newf = 0.25 * w * (f[i+1, j] + f[i-1, j] +\
                     f[i, j+1] + f[i, j-1] +\
                     rho[i, j] * h**2) + (1 - w) * f[i, j]
return newf

```

[]: # Funzione che calcola il potenziale in tutti i punti della griglia

```

def Step(f, rho, h, w, N, boundary, pos1, pos2):
    for i in range(1, N-1):
        for j in range(1, N-1):
            f[i, j] = SORAlg(f, rho, h, w, i, j, N, boundary, pos1, pos2)
    return f

```

[]: # Funzione che definisce la condizione di convergenza

```

# Raggiunta la convergenza l'algoritmo viene bloccato
def ConvergenceCondition(diff, threshold):
    return diff < threshold

```

[]: # Funzione che applica l'algoritmo SOR fino alla convergenza

```

def Run(f, rho, h, w, threshold, boundary, pos1, pos2):
    diff = 10 * threshold
    N = len(f)
    Nstep = 0
    while(not ConvergenceCondition(diff, threshold)):
        oldF = f.copy()
        f = Step(f, rho, h, w, N, boundary, pos1, pos2)
        diff = np.max(np.abs(f - oldF)) / (np.max(rho) * h**2)
        Nstep += 1
    return f, Nstep

```

1.3 Codice

Di seguito si è implementato il codice per analizzare il potenziale all'interno di una griglia quadrata:

```
[ ]: N = 100          # Grandezza griglia
      L = 0.1          # Grandezza scatola: 10 cm/0.1m
      h = L / N        # Spaziatura punti della griglia
      w = 1.92         # Peso per l'algoritmo SOR
      f = np.zeros((N, N))    # Potenziale
      rho = np.zeros((N, N)) # Densità superficiale/lineare, boh da vedere

      delta_V = 2       # Volt
      d = 0.06          # Metri
      l = 0.06          # Metri

      pos1 = int((1/5) * N)  # Posizione prima piastra
      pos2 = int((4/5) * N)  # Posizione della seconda piastra

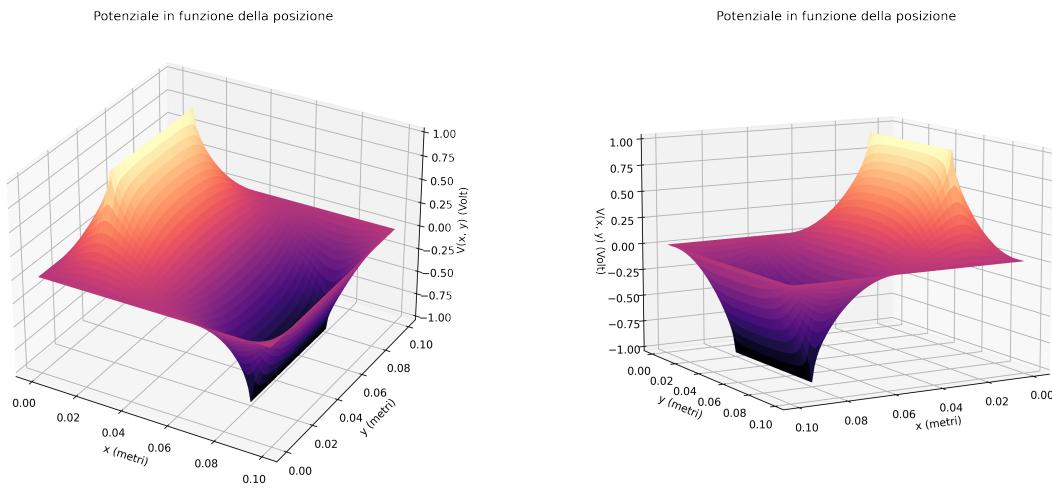
      rho[N//2, N//2] = 1
```

```
[ ]: # Calcola il potenziale nella griglia f,
      # inoltre restituisce anche il numero di iterazioni per arrivare a convergenza
      f, _ = Run(f, rho, h, w, 1e-4, N // 2, pos1, pos2)

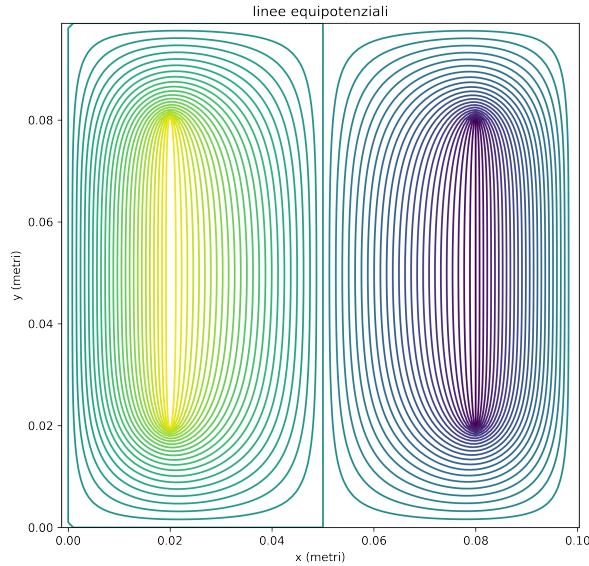
      # Crea una griglia a partire da due array, combinando tutti gli elementi di x e y
      x, y = np.meshgrid([i * h for i in range(N)], [j * h for j in range(N)])

      fig = plt.figure()
      ax = fig.add_subplot(111, projection = "3d")
      ax.plot_surface(x, y, f, cmap=cm.magma, linewidth=0, antialiased=False)

      ax.set_xlabel("x (metri)")
      ax.set_ylabel("y (metri)")
      ax.set_zlabel("V(x, y) (Volt)")
      ax.set_title("Potenziale in funzione della posizione")
      fig.set_figwidth(8)
      fig.set_figheight(8)
```



I grafici sopra rappresentano il potenziale nella griglia 100×100 . I due picchi si trovano rispettivamente a $+1$ e -1 volt in corrispondenza della posizione delle due sezioni delle piastre.



Sono riportate le linee di livello del grafico del potenziale, esse rappresentano le linee equipotenziali. Si può notare che il potenziale, nella retta che divide orizzontalmente la scatola in due parti uguali, decresce linearmente da $+1$ a -1 volt. Questo è supportato dalla teoria dei condensatori, il quale dice che lontano dai bordi il campo elettrico tra le armature è costante e quindi il potenziale varia linearmente come si vede dalla relazione $\vec{E} = -\nabla V$.

Più ci si allontana dal bordo del condensatore più la variazione del potenziale non è lineare come

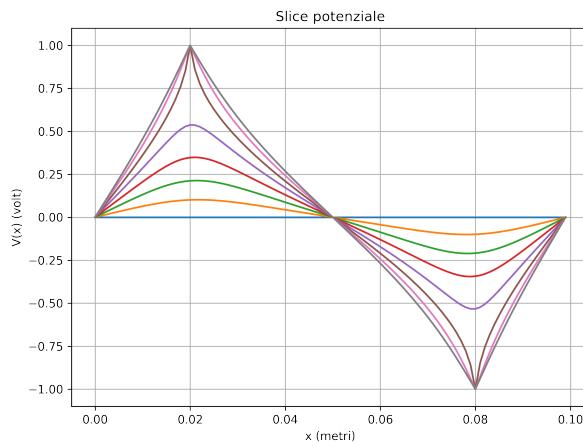
mostrato dal grafico successivo:

```
[ ]: fig3, ax3 = plt.subplots()
ax3.grid(True)

# Prende il potenziale in fette che dividono la scatola orizzontalmente
# man mano che ci si sposta verso il centro
for i in range(0, 30, 4):
    ax3.plot(y[:,0], f[i, :])

ax3.set_xlabel("x (metri)")
ax3.set_ylabel("V(x) (volt)")
ax3.set_title("Slice potenziale")

fig3.set_figwidth(8)
fig3.set_figheight(6)
# fig3.savefig("SlicePot.png", dpi=300)
```



Per arrivare a convergenza nel minor tempo possibile con il minor numero di iterazioni SOR si è calcolato il numero di iterazioni cambiando il peso, si è ottenuto il seguente grafico:

```
[ ]: # Crea un array di 30 pesi diversi tra 1.7 e 1.99
w1_array = np.linspace(1.7, 1.9, 20)
w2_array = np.linspace(1.9, 2.0, 10)
w_array = np.concatenate((w1_array, w2_array))
w_array[-1] = 1.99
iter_w = np.size(w_array)

# Conterrà il numero di iterazioni per ogni peso
NStep_array = np.zeros(iter_w)
```

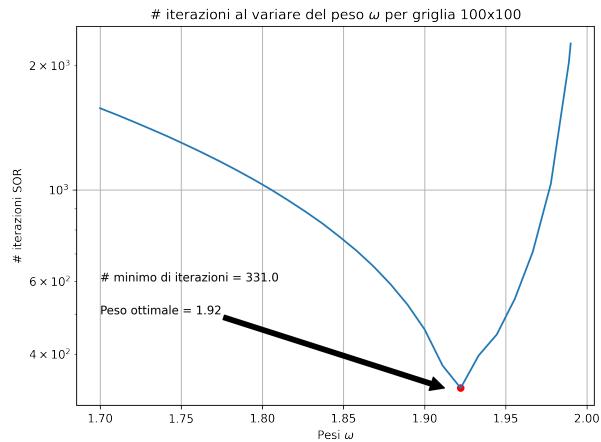
```
[ ]: # Grafico delle iterazioni di SOR rispetto ai pesi w
# Calcola la griglia per ogni peso e restituisce il numero di iterazioni
for i in range(iter_w):
    f = np.zeros((N, N))
    f, NStep_array[i] = Run(f, rho, h, w_array[i], 1e-4, N // 2, pos1, pos2)

[ ]: NStep_array = np.loadtxt("Nsteps(100x100).txt")
fig2, ax2 = plt.subplots()
ax2.plot(w_array, NStep_array)
ax2.set_xlabel("Pesi $\omega$")
ax2.set_ylabel("# iterazioni SOR")
ax2.set_yscale("Log")
ax2.set_title("# iterazioni al variare del peso $\omega$ per griglia 100x100")
plt.grid(True)

# Trova il peso ottimale cercando il numero minimo di iterazioni
# per arrivare a convergenza
min_steps = NStep_array[0]
for i in range(iter_w):
    if(NStep_array[i] < min_steps):
        min_steps = NStep_array[i]
        w_ottimale = w_array[i]

print("Peso ottimale :", w_ottimale)
print("# iterazioni minimo :", min_steps)
ax2.annotate(f'Peso ottimale = {round(w_ottimale, 2)}', xytext=(1.70, 500),
            xy=(w_ottimale-0.01, 331), arrowprops={'facecolor': 'black'})
ax2.annotate(f'# minimo di iterazioni = {min_steps}', xytext=(1.70, 600),
            xy=(w_ottimale-0.01, 331))
ax2.scatter(w_ottimale, min_steps, color="red", s=30)

fig2.set_figwidth(8)
fig2.set_figheight(6)
# fig2.savefig("iterconomega.png", dpi=300)
```



Si è studiato il potenziale al variare della distanza tra le piastre del condensatore ottenendo il grafico seguente:

```
[ ]: # Nomi dei file
nomi = ["1.png", "2.png", "3.png", "4.png", "5.png", "6.png", "7.png", "8.png",
        , "9.png", "10.png", "11.png", "12.png", "13.png", "14.png", "15.png", ↴
        →"16.png"]

fig4, ax4 = plt.subplots()
fig6, ax6 = plt.subplots()

fig4.set_figwidth(8)
fig4.set_figheight(8)

fig6.set_figwidth(8)
fig6.set_figheight(6)
ax4.grid(True)

ax6.set_xlabel("x (metri)")
ax6.set_ylabel("y (metri)")
ax6.axis("equal")
ax6.set_title("linee equipotenziali")

ax4.set_xlabel("x (metri)")
ax4.set_ylabel("V(x) (volt)")
ax4.set_title("Potenziale al centro per distanze diverse")

# array delle possibili posizioni nella griglia
pos = np.arange(0, 101, 1)
# Prende solo alcune posizioni per la prima piastra
pos1 = pos[2 : N//2 - 1 : 3]
# Calcola le posizioni per la seconda piastra
```

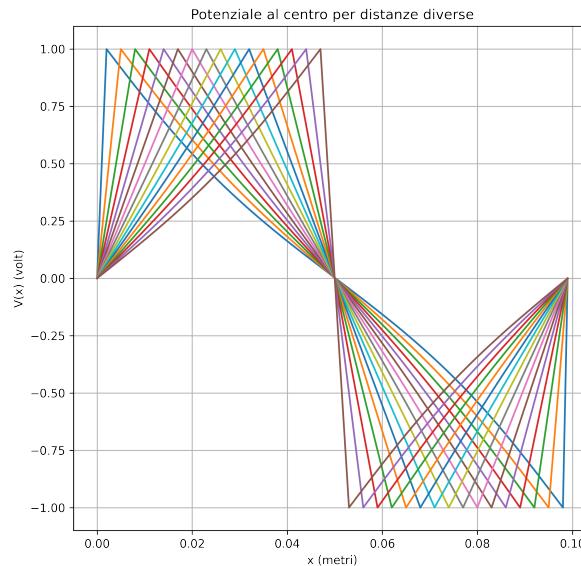
```

pos2 = pos[-pos1] - 1
ndist = np.size(pos2)
NStep_array = np.zeros(ndist)

# Crea il grafico del potenziale nella fetta centrale per diverse distanze
# Inoltre crea il grafico delle linee di livello per ogni distanza
for i in range(ndist):
    f = np.zeros((N, N))
    f, NStep_array[i] = Run(f, rho, h, w, 1e-4, N // 2, pos1[i], pos2[i])
    x, y = np.meshgrid([j * h for j in range(N)], [k * h for k in range(N)])
    ax4.plot(y[:, 0], f[N//2, :])
    # ax6.contour(x, y, f, levels = 50)
    # fig6.savefig(nomi[i], dpi=300)
    # ax6.cla()

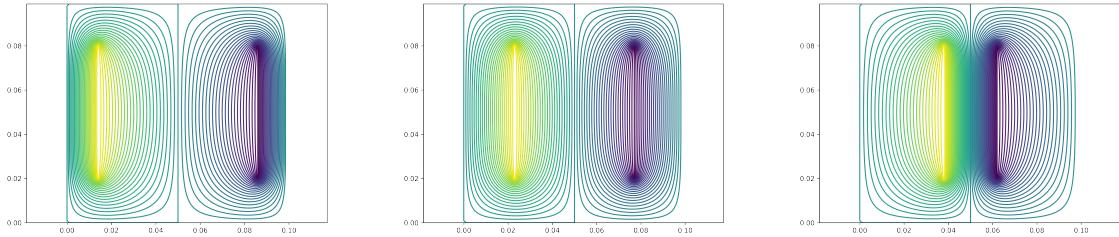
# fig4.savefig("PotCentroDist.png", dpi=300)

```



Si può notare che man mano che la distanza aumenta, il potenziale al centro inizia a perdere la linearità, lo stesso si nota all'esterno del condensatore man mano che si diminuisce la distanza tra le piastre.

Sono riportati i grafici delle linee di livello per le distanze 0.72, 0.54 e 0.24 metri:



Si è inoltre calcolato il numero di iterazioni per arrivare a convergenza al variare della distanza tra le piastre ottenendo il seguente grafico:

[]: # Grafico Numero di iterazioni al variare della distanza tra le piastre

```

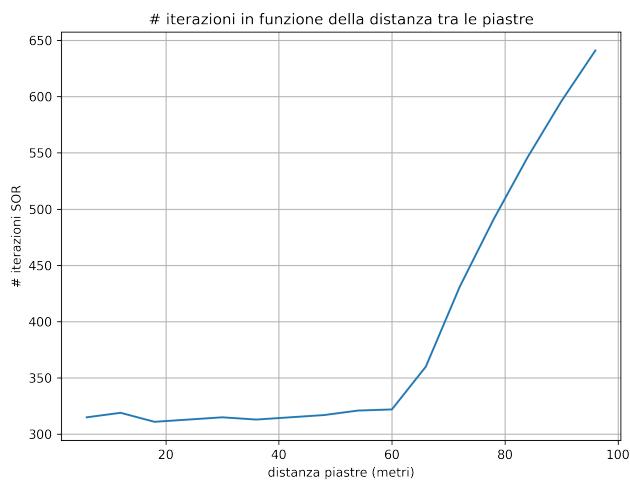
fig5, ax5 = plt.subplots()
ax5.set_ylabel("# iterazioni SOR")
ax5.set_xlabel("distanza piastre (metri)")
ax5.set_title("# iterazioni in funzione della distanza tra le piastre")

# array con le distanze tra le piastre
distanze = pos2 - pos1

ax5.plot(distanze, NStep_array)
ax5.grid(True)

fig5.set_figwidth(8)
fig5.set_figheight(6)
# fig5.savefig("itercondist.png", dpi=300)

```



Il numero di iterazioni aumenta considerevolmente dopo che si supera la distanza di 60cm tra le due piastre. Questo, si pensa derivi dal fatto che man mano che si aumenta la distanza la variazione del

potenziale tra due punti diminuisce e quindi, per come lavora il metodo SOR (cioè usare le soluzioni nuove dove sono disponibili, tirando così la soluzione verso quei valori), ci vogliono più iterazioni per arrivare a convergenza. Mentre per piccole distanze la variazione tra due punti è molto più grande rispetto al caso precedente e quindi la soluzione viene "tirata" di più verso i valori in cui c'è già la nuova soluzione.

Infine si è calcolato il peso ottimale al variare della grandezza della griglia, il peso varia linearmente con l'aumentare del numero di punti della griglia:

```
[ ]: # crea un array con i pesi da 1.7 a 1.99
w1_array = np.linspace(1.7, 1.9, 20)
w2_array = np.linspace(1.9, 2.0, 10)
w_array = np.concatenate((w1_array, w2_array))
w_array[-1] = 1.99
iter_w = np.size(w_array)

# array con le varie grandezze della griglia
N_array = np.arange(45, 100, 5)
iter_N = np.size(N_array)
pos1 = pos2 = np.zeros(iter_N)
# Calcola la posizione delle piastre al variare della griglia
pos1 = (1/5) * N_array
pos2 = N_array - pos1
# trasforma in interi dato che verranno usati come indici
pos1 = pos1.astype(int)
pos2 = pos2.astype(int)

NStep_array = np.zeros((iter_N, iter_w))

[ ]: # Grafico w_ottimale in funzione della grandezza della griglia

for i in range(iter_N):
    for j in range(iter_w):
        f = np.zeros((N_array[i], N_array[i]))
        f, NStep_array[i, j] = Run(f, rho, h, w_array[j], 1e-4, N_array[i] // 2, pos1[i], pos2[i])

[ ]: # Calcola il peso ottimale per ogni griglia e lo mette in un array
w_ottimale_array = np.zeros(iter_N)
for j in range(iter_N):
    min_steps = NStep_array[j, 0]
    for i in range(iter_w):
        if(NStep_array[j, i] < min_steps):
            min_steps = NStep_array[j, i]
    w_ottimale_array[j] = w_array[i]

[ ]: # Crea il grafico con il peso ottimale
# in funzione della grandezza della griglia
```

```

fig7, ax7 = plt.subplots()

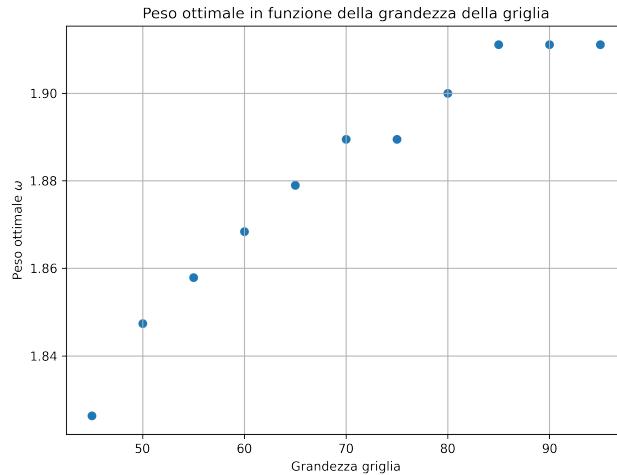
fig7.set_figwidth(8)
fig7.set_figheight(6)
ax7.grid(True)

ax7.set_ylabel("Peso ottimale $\omega$")
ax7.set_xlabel("Grandezza griglia")
ax7.set_title("Peso ottimale in funzione della grandezza della griglia")

w_ottimale_array = np.loadtxt("w_ottimale_array.txt")
ax7.scatter(N_array, w_ottimale_array)

# fig7.savefig("Pesoottegrandezzagrig.png", dpi=300)

```



Al variare della grandezza della griglia la simulazione diventa più accurata da come si può notare dal grafico successivo:

```

[ ]: # Crea quattro grafici del potenziale nella fetta centrale
      # Variando la grandezza della griglia: 25, 50, 75, 100

      # array con le varie grandezze della griglia
N_array = np.array([25, 50, 75, 100])
iter_N = np.size(N_array)
w = 1.87
pos1 = pos2 = np.zeros(iter_N)
      # Calcola la posizione delle piastre al variare della griglia
pos1 = (1/5) * N_array
pos2 = N_array - pos1

      # Spaziatura punti della griglia

```

```

h_array = L / N_array

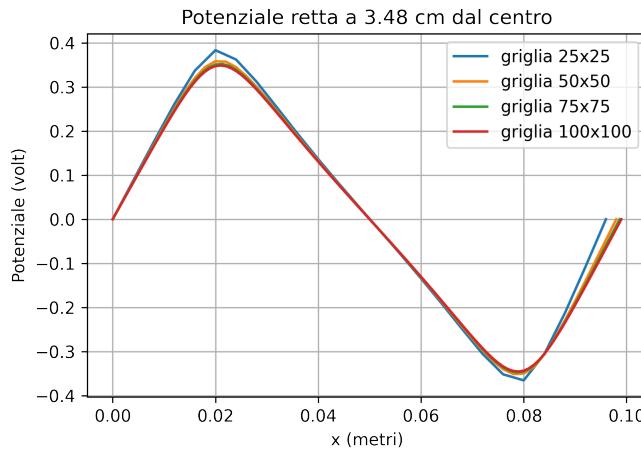
fig8, ax8 = plt.subplots()

ax8.set_ylabel("Potenziale (volt)")
ax8.set_xlabel("x (metri)")
ax8.set_title("Potenziale retta a 3.48 cm dal centro")

for i in range(iter_N):
    f = np.zeros((N_array[i], N_array[i]))
    f, _ = Run(f, rho, h_array[i], w, 1e-4, N_array[i] // 2, pos1[i], pos2[i])
    x, y = np.meshgrid([k * h_array[i] for k in range(N_array[i])], [j *_
    ↪h_array[i] for j in range(N_array[i])])
    ax8.plot(y[:, 0], f[N_array[i]//8, :], label=f"griglia_"
    ↪{N_array[i]}x{N_array[i]}")
    ax8.legend()

ax8.grid(True)
# fig8.savefig("Pot348.png", dpi=300)

```



Esso rappresenta il potenziale nella retta che dista 3.48cm dal centro.

Si è poi analizzato il massimo del potenziale nella retta a 3.48cm dal centro al variare della grandezza della griglia.

```

[ ]: # Calcola il potenziale in griglie di grandezza diversa

# array con le varie grandezze della griglia
N_array = np.arange(20, 205, 5)
iter_N = np.size(N_array)
w = 1.87
pos1 = pos2 = np.zeros(iter_N)
# Calcola la posizione delle piastre al variare della griglia

```

```

pos1 = (1/5) * N_array
pos2 = N_array - pos1

# Spaziatura punti della griglia
h_array = L / N_array

# Array che conterrà il massimo, per ogni griglia,
# della fetta di potenziale distante 3.48 cm dal centro
max_pot = np.zeros(iter_N)

for i in range(iter_N):
    f = np.zeros((N_array[i], N_array[i]))
    f, _ = Run(f, rho, h_array[i], w, 1e-4, N_array[i] // 2, pos1[i], pos2[i])
    max_pot[i] = np.max(f[N_array[i]//8, :])

```

```

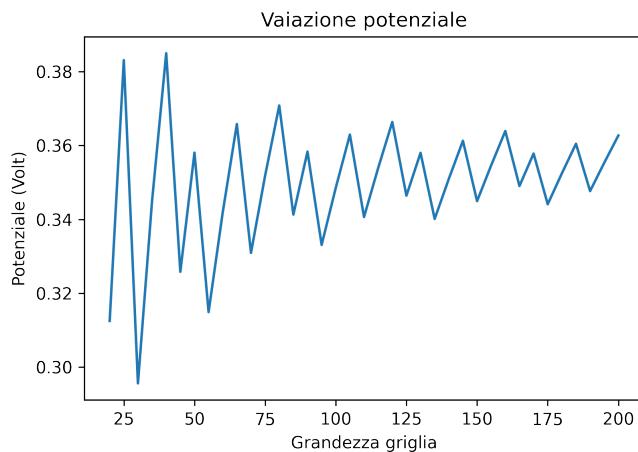
[ ]: max_pot = np.loadtxt("massimi.txt")
fig9, ax9 = plt.subplots()

ax9.set_title("Vaiazione potenziale")
ax9.set_ylabel("Potenziale (Volt)")
ax9.set_xlabel("Grandezza griglia")

ax9.plot(N_array, max_pot)

#fig9.savefig("max_potGraf.png", dpi=300)

```



Il valore del massimo oscilla al variare della grandezza della griglia, per dimensioni piccole le oscillazioni sono più ampie, mentre per dimensioni elevate le oscillazioni sono più piccole.

1.4 Conclusioni

Per concludere, il potenziale calcolato con il metodo SOR è risultato essere in accordo con ciò che ci si aspettava, ovvero il potenziale al centro del condensatore varia in modo lineare e man mano che ci si avvicina ai bordi esso varia con un'altra legge. Lo stesso vale per la variazione del peso ottimale in funzione della grandezza della griglia e la variazione del numero di iterazioni all'aumentare della distanza tra le piastre.

2 Millisecond Radio Pulsar

2.1 Introduzione

Le radio pulsar ordinarie sono delle stelle di neutroni con un campo magnetico di circa 10^{12} gauss e un periodo di rotazione che varia tra i 0.1 e 3 secondi. Nel caso seguente si è presa in considerazione una millisecond radio pulsar, le quali hanno un campo magnetico più piccolo di quelle ordinarie (circa 10^9 gauss) con periodo di rotazione dell'ordine dei millisecondi. In tutti e due i casi il fascio di onde radio emesso dalla stella è causato dall'azione combinata del campo magnetico e della rotazione. Esse si comportano come un faro, quando il fascio passa sulla Terra si rileva un aumento del numero di fotoni.

I dati ottenuti dall'esperimento rappresentano il numero di fotoni rilevati in un intervallo di tempo di 500 secondi. Sui dati è stata eseguita una trasformata di fourier veloce costruita per essere sensibile a variazioni rapide dei segnali.

2.2 Trasformata di Fourier

Un segnale che varia nel tempo può essere rappresentato come una sovrapposizione di onde di opportuna ampiezza, oscillanti nel tempo con specifica frequenza. La trasformata di Fourier è una operazione che trasforma una funzione della variabile di un dominio in una funzione della variabile del dominio duale. Nel caso seguente si è trasformata una funzione sul dominio temporale in una funzione sul dominio di frequenze. La definizione di trasformata di fourier è

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t} dt \quad (6)$$

mentre l'antitrasformata è

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{-i\omega t} d\omega \quad (7)$$

2.3 Codice

Vengono importate le librerie e funzioni utili per l'analisi dei dati. Oltre a matplotlib e numpy, usate rispettivamente per creare grafici e manipolare array, sono state utilizzate le funzioni rfft (real fast fourier transform) e find_peaks.

```
[115]: import matplotlib.pyplot as plt
import numpy as np
from scipy.fft import fft, fftfreq, rfft
```

```
from scipy.signal import find_peaks
```

Si inizia importando i dati con la funzione loadtxt di numpy, essa creerà un array (chiamato dati) con due colonne, nella prima sono salvati i tempi e nella seconda il numero di fotoni a quel dato istante di tempo. Poi le due colonne sono salvate separatamente su due variabili (x: tempo e y: numero di fotoni)

```
[116]: dati = np.loadtxt("1808_lc.dat")      # Carica i dati del file nella variabile dati  
x = dati[:,0]                                # Salva la prima colonna in x, ovvero il tempo  
y = dati[:,1]                                # Salva la seconda colonna in y, ovvero il  
    ↳ numero di fotoni
```

Di seguito vengono definite delle variabili come tau (distanza temporale tra un istante di tempo e il successivo), T (Periodo, in questo caso è il tempo totale che è circa 500 secondi) e N (numero di istanti di tempo). Infine viene definito l'array delle frequenze. Vengono create $N/2 + 1$ (la doppia sbarra indica che la divisione è intera) elementi perchè la funzione rfft scarta le frequenze negative.

```
[117]: tau = x[1] - x[0]                      # Tempo minimo misurato, time-step  
T = x[-1]                                     # Tempo dell'esperimento  
N = len(x)                                     # Lunghezza dell'array x  
freq = np.arange(N//2 + 1) / T                 # Calcola la frequenza per ogni istante di  
    ↳ tempo
```

Quindi si fa la trasformata di fourier discreta con l'algoritmo rfft e si calcola il valore assoluto.

```
[118]: yf = rfft(y) * tau                      # Trasformata solo della parte reale, scartate  
    ↳ le frequenze negative  
yf_abs = np.abs(yf)                            # Valore assoluto della trasformata
```

In questa cella vengono tagliate le frequenze più gradi di $400Hz$ perchè si è identificato un picco vicino a quella frequenza. Per verificare che ci sia effettivamente un picco si è usata la funzione find_peaks di scipy. Si è modificato il parametro prominence di find_peaks, esso misura l'importanza di un picco relativa all'altezza e la distanza dagli altri picchi.

```
[121]: y_final = yf_abs**2  
fft_pezzo = y_final[0:210000]  
freq_pezzo = freq[0:210000]                   # Taglia le frequenze alte perchè non sono  
    ↳ presenti picchi  
  
peaks, _ = find_peaks(fft_pezzo, prominence=2e7)
```

Si è scelto di normalizzare i dati secondo l'altezza del picco sui $400Hz$ in modo tale da visualizzarlo bene nel grafico.

```
[125]: norm = fft_pezzo[peaks[-1]]            # Prende il valore delle y in corrispondenza  
    ↳ della frequenza cercata
```

```

y_graf = y_final[50:300000] / norm # Normalizza i dati per ottenere un picco
                                    ↵visibile
freq_graf = freq[50:300000]          # Taglia le prime frequenze per eliminare il
                                    ↵picco al tempo zero

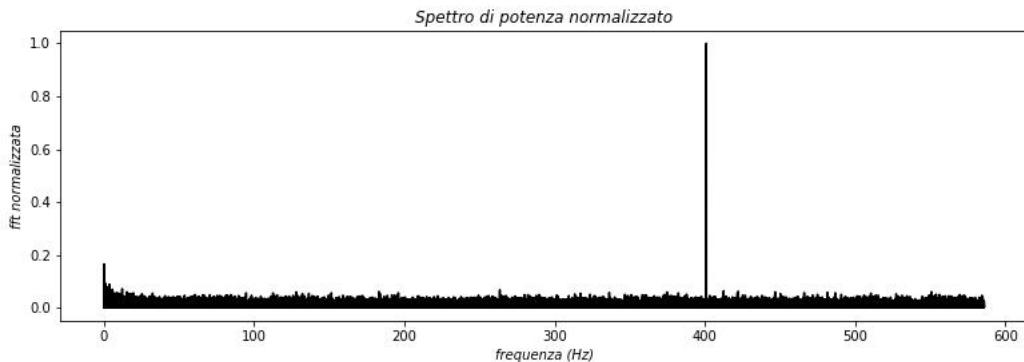
fig1, ax1 = plt.subplots()
fig1.set_figwidth(13)

ax1.plot(freq_graf, y_graf, color="black")

ax1.set_ylabel("fft normalizzata", style="italic")
ax1.set_xlabel("frequenza (Hz)", style="italic")
ax1.set_title("Spettro di potenza normalizzato", style="italic")

fig1.savefig("SpettroPotenza.jpg")

```



Il grafico mostra un picco vicino ai 400Hz , di seguito è riportato il valore preciso del picco e il periodo di spin della millisecond radio pulsar.

```
[124]: frequenza = freq[peaks[-1]]    # Hz
periodo = 1000 / frequenza        # ms
print("Frequenza di pulsazione =", frequenza, "Hz")
print("Periodo =", periodo, "ms")  # ms
```

Frequenza di pulsazione = 400.99032997853493 Hz
 Periodo = 2.493825724060553 ms

2.4 Conclusioni

L'analisi di fourier ha permesso di identificare una pulsazione dei raggi X emessa dalla millisecond radio pulsar vicino alla frequenza di 401Hz , ad essa è associato un periodo di spin di 2.5ms . Si conclude che la pulsar studiata può essere la *SAX J1808.4 – 3658* oppure *IGR J1749.8 – 2921*.