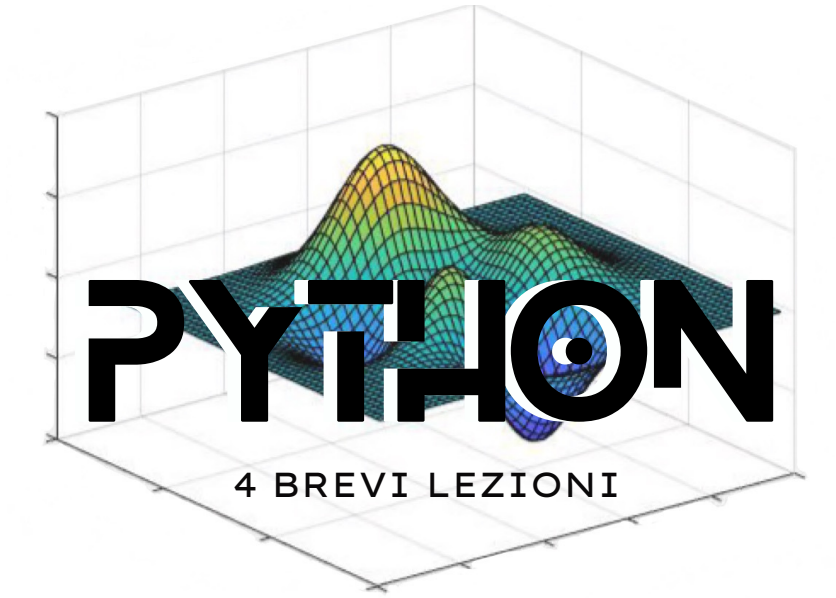
4 Brevi lezioni di Python

Francesco Zeno Costanzo

(Do you remember the) 21(st night of) September 2024



I think, it's time we blow this scene.
Get everybody and the stuff together.
Ok three, two, one, let's jam.
Seatbelts, Tank! (1999)

Indice

1 Introduzione

Uno dei problemi più frequenti che si pone durante la scrittura di programmi, è la questione *performance*: quando ci si approccia ad un problema che vogliamo risolvere non solo bisogna tenere bene a mente le limitazioni dell'hardware di cui disponiamo, ma anche di implementare un algoritmo che abbia un tempo di esecuzione il più piccolo possibile. Nella maggior parte dei casi è sufficiente implementare un algoritmo più efficiente per migliorare le prestazioni (per fare un esempio, per migliorare le performance nel calcolo della sequenza di Fibonacci è consigliato procedere con un algoritmo iterativo piuttosto che uno ricorsivo). Tuttavia, in alcuni casi, non è sufficiente: questo si rivela particolarmente vero nel caso in cui abbiamo una grande mole di dati e su ognuno di questi dobbiamo andare ad effettuare delle operazioni matematiche/logiche, che solitamente è un processo molto lento in termini di esecuzione. Un esempio da fisici può essere la simulazione numerica delle equazioni differenziali, dove dobbiamo andare a discretizzare lo spazio e il tempo ed effettuare su ognuno delle cellette una serie di operazioni matematiche.
Il motivo del lungo tempo di esecuzione nell'esempio riportato non è necessariamente da imputare ad un pessimo algoritmo: in realtà, per come sono nati e sono stati strutturati dei linguaggi come C++ e Python, il motivo deriva dal fatto che il microprocessore che esegue il programma itera singolarmente cella dopo cella. Quindi, nel caso di un gran numero di cellette, il tempo di esecuzioni ne risente particolarmente.
Il motivo è puramente storico: i computer, inizialmente, erano per lo più monoprocessore e, conseguentemente, i linguaggi di programmazione sono stati strutturati in modo tale da eseguire le operazioni in modo sequenziale. In Python tale blocco è dovuto alla presenza del cosiddetto **GIL** (global interpreter lock) che impedisce l'esecuzione di più thread (che per non scendere nel tecnico si può pensare come un processore) in parallelo[1]. Ad oggi, invece, i computer sono multiprocessore, ma la struttura *vanilla* dei linguaggi non è stata cambiata. E' possibile, però, attingere al vero potere delle risorse dei multiprocessori, e non solo, utilizzando delle librerie. Noi, tuttavia, vogliamo attaccare il problema della parallelizzazione introducendo uno strumento che sta prendendo sempre più piede in Python: i cosiddetti compilatori *jit* (just-in-time). Questi compilatori non permettono solo di parallelizzare il codice (siccome riescono a disabilitare il GIL) ma anche di ottimizzare il codice. Questi compilatori infatti, a differenza dell'interprete classico di Python, vanno a compilare, all'inizio dell'esecuzione del programma, le parti di codice che vengono "marcate" come da ottimizzare in un formato più vicino al linguaggio macchina (da qui il nome di compilatore just-in-time). Questo permette di migliorare le performance del codice per i seguenti motivi

1. il codice viene compilato in un linguaggio ottimizzato;
2. il codice compilato viene riutilizzato. Infatti, se indichiamo una funzione come da compilare, il compilatore, per ogni chiamata di quella funzione, inserirà un puntatore che punta all'indirizzo di memoria in cui sono contenute le istruzioni compilate della funzione;
3. (motivo tecnico): per come sono implementati questi compilatori, utilizzano delle politiche di compilazione che sono mirate per lo specifico processore della macchina su cui il codice viene eseguito. Questo consente su molti microprocessori, per esempio, di immagazzinare le istruzioni ottimizzate dalla compilazione direttamente in registri del microprocessore (detta cache), diminuendo così il tempo di lettura delle istruzioni e aumentando così le prestazioni.

La libreria che consente di effettuare tutte queste belle cose che abbiamo detto è la libreria `numba`. La documentazione di numba è molto vasta, quindi consiglio di leggerla per approfondire l'argomento, tuttavia spero che i seguenti esempi proposti siano comunque esaustivi per capire il funzionamento.

2 Equazione al laplaciano in 2 dimensioni

Uno dei problemi più complicati da risolvere in maniera analitica è il problema fondamentale dell'elettrostatica: l'equazione di Laplace. Il motivo per cui è abbastanza complicata è che spesso le condizioni al bordo di questa equazione fanno in modo che la soluzione non sia esprimibile sotto forma di serie e/o integrale. Tuttavia nella maggior parte dei casi è possibile dare una soluzione approssimata numericamente. Tale equazione è definita come

$$\nabla^2 V = 0,$$

e, ringraziando i nostri amici matematici, sappiamo che esiste sempre una soluzione a tale equazione. Mentre dal corso di Fisica 2 sappiamo che la soluzione è unica (permettendomi una perla del professore La Rocca, ai Fisici interessa solo l'unicità della soluzione: per verificare l'esistenza di una soluzione basta andare in laboratorio e verificare se effettivamente esiste). Possiamo procedere, come al solito, andando a discretizzare lo spazio e

[1] in realtà il GIL non impedisce completamente la parallelizzazione del codice, perché non impedisce la creazione di più processi tuttavia lato performance è più lenta rispetto alla parallelizzazione dei thread

approssimando il laplaciano con le derivate discrete, che assume la seguente forma

$$\frac{u(x+ih_x,y)-2u(x,y)+u(x-ih_x,y)}{h_x^2}+\frac{u(x,y+ih_y)-2u(x,y)+u(x,y-ih_y)}{h_y^2}=0,$$

dove h_x e h_y sono rispettivamente il passo fra le ascisse e il passo fra le ordinate. Esplicitando per $u(x,y)$, risulta che

$$u(x,y)=\frac{1}{4}(u(x+ih_x,y)+u(x-ih_x,y)+u(x,y+ih_y)-u(x,y-ih_y)). \tag{1}$$

In questo caso, siccome il laplaciano è un operatore che si comporta "abbastanza" bene, la nostra soluzione è sicuramente stabile per $h << 1$. La questione della stabilità non è banale e per questo si rimanda a delle trattazioni più accurate dell'argomento. Utilizzando `numba`, possiamo mettere delle condizioni al contorno più *impegnative*, ovvero possiamo mettere delle condizioni al bordo dove la nostra funzione è una sovrapposizioni di seni, coseni ed esponenziali, le quali sono decisamente più computazionalmente impattanti sul tempo di esecuzioni rispetto ai casi "soliti" che si vedono negli esercizi dove, per esempio, abbiamo un conduttore che è a potenziale costante. Prima di procedere con l'esempio, avevo "promesso" nelle scorse lezioni che ci saremmo soffermati un po' meglio per capirne come mai "funziona" (matematicamente) il metodo delle differenze finite (ovvero il procedimento adottato in cui discretizziamo lo spazio e approssimiamo le derivate a differenze), quindi cerchiamo di formalizzare questo processo un pochino: considerando l'operatore $L : L^2((0,1)^2) \to L^2((0,1)^2)$ definito come

$$u \overset{L}{\mapsto} \nabla^2 u = \frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2},$$

e indicando con $\bar{\omega}_h$ la rete di punti che approssima il nostro spazio continuo $(0,1)^2$, osserviamo che preso un punto (x^*,y^*) interno alla griglia, allora si ha che la nostra soluzione, se sufficientemente regolare, possiamo affermare che

$$u(x^*\pm h,y^*)=u(x^*,y^*)\pm\partial_x u(x^*,y^*)h+o(h).$$

Chiaramente la relazione qua sopra è ben posta se abbiamo che $u \in C^2$. Osserviamo che approssimando l'operatore differenziale $\mathrm{D} : L^2((0,1)) \to L^2((0,1))$ tale che $u \overset{\mathrm{D}}{\mapsto} \frac{d}{dx}u$ con $L_\pm u(x^*,y^*)=\frac{u(x^*\pm ih_x,y^*\pm ih_y)-u(x^*,y^*)}{|\vec{h}|}$ abbiamo che

$$L_\pm u(x^*,y^*)-u'(x^*,y^*)=O(h).$$

Se $\max_{\bar{\omega_h}} |L_\pm u(x^*,y^*)-u'(x^*,y^*)| \to 0$ per $h\to 0$ deduciamo, com'era lecito aspettarsi, che per $h\to 0$ si ha che $L_\pm v \to v'$.

Per approssimare la derivata qua abbiamo lavorato con un campione di due punti, ovvero abbiamo utilizzato solamente il punto successivo (nel caso di L_- abbiamo usato il punto precedente) e il punto nella nostra rete per approssimare la derivata. Nulla ci vieta di passare a un campione di tre punti: questo può essere particolarmente comodo nei sistemi meccanici per togliersi la dipendenza esplicita della velocità nel calcolo della posizione "aggiornata" di un punto materiale (integrazione di Verlet). Tuttavia è sconsigliato prendere un campione di troppi punti poiché rischieremmo di aumentare di troppo il numero di calcoli per delle correzioni misere.

Per le derivate seconde la cosa è leggermente più *tricky*, siccome per approssimare la derivata alla solita maniera è necessario (e questo si vede banalmente dai conti) che la nostra soluzione sia appartenente a C^4. Inoltre non è possibile approssimare la derivata seconda con un campione di due punti, quindi dobbiamo avere un campione almeno a tre punti: si ha dunque

$$u(x^*\pm ih,y^*)=u(x^*,y^*)\pm\partial_x u(x^*,y^*)h+\frac{h^2}{2}\partial^2_{xx}u(x^*,y^*)\pm\frac{h^3}{3}\partial^3_{xxx}u^{(3)}(x^*,y^*)+\frac{h^4}{24}\partial^4_{xxxx}u(x^*,y^*)+o(h^4).$$

Si ricava che, posto $\delta u(x^*,y^*)=\frac{u(x^*+ih,y^*+ih)-2u(x^*,y^*)+u(x^*-ih,y^*-ih)}{h}$, abbiamo che

$$\delta u(x^*,y^*)-\partial^2_{xx}u(x^*,y^*)=\frac{h^2}{12}\partial^4_{xxxx}u+o(h).$$

Dunque per $h\to 0$ questo errore diventa trascurabile.
Procediamo adesso a scrivere il codice:

```
import numpy as np
import numba
import matplotlib.pyplot as plt

"""
Immaginiamo di essere in una scatola (-1, 1)^2 e impostiamo le condizioni al bordo per la
  nostra scatola
```

```
"""
edge = np.linspace(-1, 1, 300)
bordo_supy = np.cos(np.pi * edge / 2)
bord_infy = edge**4
bordo_supx = 1/(np.e**(-1) - np.e) * (np.exp(edge)-np.e)
bordo_infx = 0.5 * (edge**2 - edge)

# Creiamo una meshgrid
xv, yv = np.meshgrid(edge, edge)
"""
La funzione meshgrid crea un array dove tutte le possibili combinazioni dei vettori edge, edge
vengono scissi in due matrici dove (xv[i, j], yv[i, j]) rappresenta un punto della griglia
"""



@numba.jit("f8[:,:](f8[:,:], i8)", nopython=True, nogil=True)
def sol_potential(potential, n_iter):
  """
  Compute the solution of the Laplace equation through finite difference method

  Params:
  -----
    potential: 2darray
      matrix of zeros that will be filled with the values of the potential in each point of the grid
    n_inter: int
      number of iterations for each point of the grid

  Return:
  -----
    potential: 2darray
      matrix cointaing the values of the potential evaluated in the each point of the grid
  """
  length = len(potential[0])
  for n in range(n_iter):
    for i in range(1, length-1):
      for j in range(1, length-1):
        potential[j][i] = 1/4 * (potential[j+1][i] + potential[j-1][i] + potential[j][i+1] + potential[j][i-1])
  return potential

# Settiamo le condizioni al bordo
potenziale = np.zeros((300, 300))
potenziale[0, :] = bordo_infy
potenziale[-1, :] = bordo_supy
potenziale[:, 0] = bordo_infx
potenziale[:, -1] = bordo_supx
potenziale = solve_potential(potenziale, n_iter=10000)

# Grafico potenziale
fig, ax = plt.subplots(1, 1, figsize=(8,6))
clr_plot = ax.contourf(xv, yv, potential, 30)
ax.set_xlabel('x/a', fontsize=20)
ax.set_ylabel('y/a', fontsize=20)
fig.colorbar(clr_plot, label='$V/V_0$', fontsize=20)
ax.set_title('Potential in square', fontsize=20)
plt.show()

```

Osserviamo come funziona la libreria `numba`: si nota, innanzitutto, che il wrapper `numba.jit` viene chiamato immediatamente prima della della funzione e osserviamo quali sono gli argomenti che gli sono stati passati:

- il primo è la *signature* della funzione, in cui sono indicati i tipi di dati che la funzione restituisce e prende in ingresso. Nel nostro caso noi vogliamo che la funzione restituisca un array 2-dimensionale e prenda in ingresso un array 2-dimensionale e un intero: si osservi che questi vanno indicati riportando prima i tipi dei dati restituiti e poi quelli che prende in input, riportando questi ultimi fra parentesi e separandoli fra virgole. Nell'esempio qua sopra, sono riportati tramite la forma abbreviata che `numba` consente di utilizzare: in questo caso `f8` sta per `float64`, dunque `f8[:, :]` rappresenta un array 2-dimensionale le cui entrate sono `float64`. Similmente, `i8` sta per `int64`;
- il parametro `nopython=True`, il quale comunica a numba che noi vorremmo (se riesce) compilare la funzione per aumentare le performance. Il motivo di tale nome deriva dal fatto che la funzione compilata sarà compilata in `C`, quindi non sarà più il formato `bytecode` che l'interprete legge ed esegue
- il parametro `nogil` che, da quanto detto prima, è abbastanza chiaro che cosa faccia.

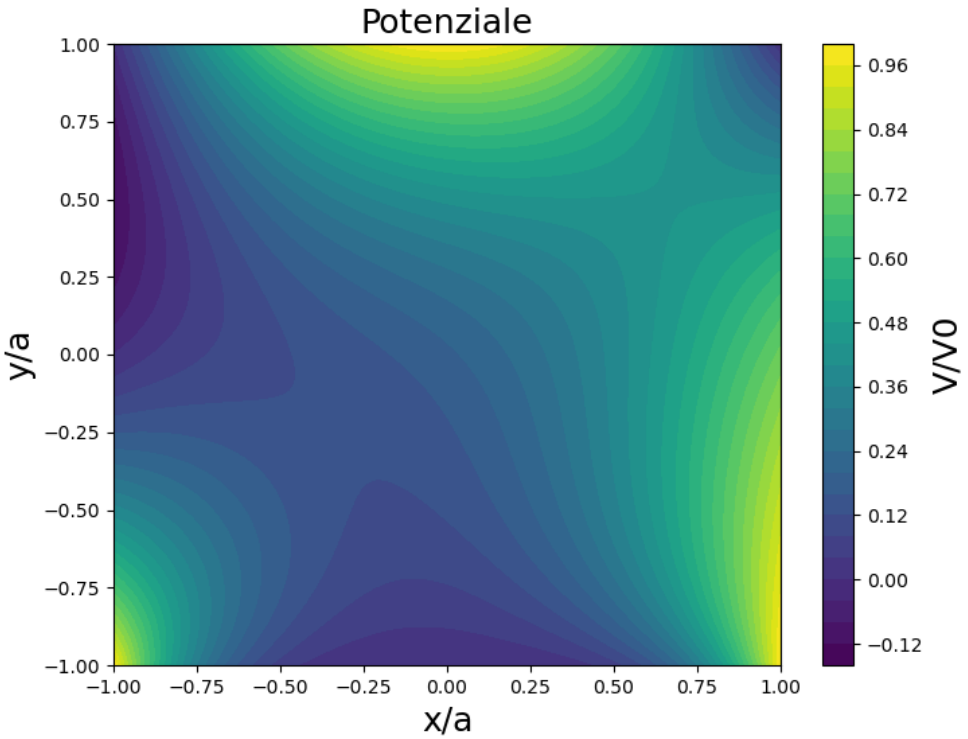


Figura 1: Soluzione numerica del potenziale nel quadrato. Il valore del potenziale è più alto per colori più chiari e più basso per colori più scuri

E' interessante, chiaramente, vedere se è migliorato e quanto è migliorato il tempo di esecuzione con `numba`: per fare ciò è sufficiente utilizzare la libreria `time`, limitandoci ad un numero di iterazioni pari a 1000.

```
numba_pot = np.copy(potenziale)
start = time.time()
numba_pot = sol_potential(potenziale, n_iter=1000)
end = time.time()

print(f"La soluzione con numba e': {end-start}")

# Funzione risolvente senza il wrapper di numba
def no_numbed_sol_potential(potential, n_iter):
  """
  Compute the solution of the potential but without numba

  Params:
  ----
    same as sol_potential

  Return:
  ----
    same as sol_potential

  """
  length = len(potential[0])
  for n in range(n_iter):
    for i in range(1, length-1):
      for j in range(1, length-1):
        potential[j][i] = 1/4 * (potential[j+1][i] + potential[j-1][i] + potential[j][i+1] +
 potential[j][i-1])
  return potential


start = time.time()
potenziale_notnumb = no_numbed_sol_potential(potenziale, n_iter=1000)
end = time.time()


print(f"La soluzione non numbata e': {end-start}")
[Output]
```

```
La soluzione con numba e': 0.593531608581543
La soluzione non numbata e': 181.39348196983337
```

Risulta quindi che `numba` è circa 307 volte più veloce! Pazzesco, no?
Le magie però non sono ancora finite: potremmo anche pensare di inserire delle condizioni al contorno ancora più *wild* e inserire, per esempio, un blocco di potenziale all'interno della nostra scatola in cui il potenziale è fisso: per fare ciò supponiamo che in tale regione a potenziale costante si ha che $V = 1$, dunque vi lascio al codice

```python
def blocco_potenziale(x, y):
  """
  Function that checks if (x, y) is a point of the grid which belongs to the area of
  constant potential
  """
  return np.select([(x>0.3)*(x<0.6)*(y > 0.3)*(y<0.6), (x <= 0.3)* (x>= 0.6)*(y<=0.3)*(y
  >=0.6)], [1., 0])

# Graphical settings for the graph of the block
plt.figure(figsize=(5,4))
plt.contourf(xv, yv, blocco_potenziale(xv, yv))
plt.colorbar()

# Creation of a grid to easily evaluate if a point belongs to the block
fixed = blocco_potenziale(xv, yv)
_bool = fixed!= 0


@numba.jit("f8[:, :](f8[:, :], b1[:, :], i8)",nopython=True, nogil=True)
def solve_potential_fixed(potential, fixed_bool, n_iter):
  """
  Compute the solution of the Laplace equation through finite difference method

  Params:
  -----
    potential: 2darray
      matrix of zeros that will be filled with the values of the potential in each point of
  the grid
    fixed_bool: 2darray
      matrix of boolean values to identify the points that are at constant potential
    n_inter: int
      number of iterations for each point of the grid

  Return:
  -----
    potential: 2darray
      matrix cointaing the values of the potential evaluated in the each point of the grid
  """
  length = len(potential[0])
    for n in range(n_iter):
        for i in range(1, length-1):
            for j in range(1, length-1):
                if not(fixed_bool[j][i]):
                    potential[i][j] = 1/4 * (potential[i+1][j] + potential[i-1][j] +
  potential[i][j+1] + potential[i][j-1])
  return potential

# Boundary conditions
potenziale = np.zeros((300, 300))
potenziale[0, :] = bordo_infy
potenziale[-1, :] = bordo_supy
potenziale[:, 0] = bordo_infx
potenziale[:, -1] = bordo_supx
potentiale[_bool] = fixed[_bool]

# Compute the solution
potenziale = solve_potential_fixed(potenziale, _bool, n_iter=10000)

# Graphical settings of the graph of the potential
fig, ax = plt.subplots(1, 1, figsize=(8,6))
clr_plot = ax.contourf(xv, yv, potential, 30)
ax.set_xlabel('x/a')
ax.set_ylabel('y/a')
fig.colorbar(clr_plot, label='$V/V_0$')
ax.set_title('Potential in square')
plt.show()
```

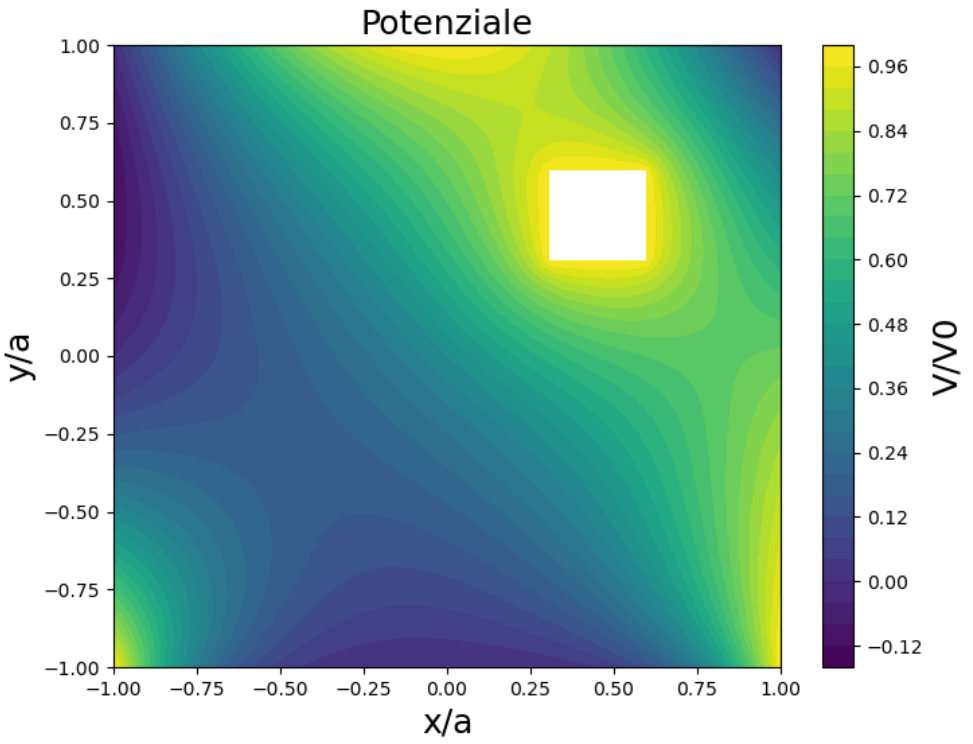


Figura 2: Soluzione del nostro potenziale con blocco a potenziale fissato.

Un problema che si pone negli esempi qua sopra proposti, tuttavia, è il fatto che il numero 10000 e tutti gli altri usati per valutare il tempo di esecuzione sono dei numeri arbitrari, conseguentemente vorremmo aggiungere un criterio con cui fermare la computazione della soluzione quando queste diventano abbastanza "vicine" fra un'iterazione e quella successiva. Chiaramente la "vicinanza" fra due soluzioni è una nozione ben posta solo definendo una norma: nel nostro esempio, possiamo pensare di prendere come norma la media integrale, ovvero l'integrale sull'area del potentiale che approssimiamo alla versione discretizzata tramite sommatoria

$$||\varphi|| = \frac{1}{m_2((-1,1)^2)} \int_{(-1,1)2} |\varphi(x,y)| dxdy \approx \frac{1}{(N_x N_y)} \sum_{i,j}^{N} |\varphi_i^j|,$$

dove $\varphi_i^j = \varphi(x_i, y_j)$ con (x_i, y_i) appartenenti alla griglia che discrettizza lo spazio. In questa maniera, possiamo passare alla funzione `solve_potential` un parametro τ che ferma l'iterazione quando

$$||\varphi^{(k+1)} - \varphi^{(k)}|| \approx \sum_{i,j}^{N} |\varphi_i^{j,(k+1)} - \varphi_i^{j,(k)}| < \tau,$$

dove $\varphi_i^{j,(k)}$ è la soluzione numerica valutata nel punto (x_i, y_i) della griglia all'iterazione $k-$esima. Piuttosto che valutare se $||\varphi^{(k+1)} - \varphi^{(k)}|| < \tau$, nell'algoritmo andremo a valutare se $\big|\, ||\varphi^{(k+1)}|| - ||\varphi^{(k)}|| \,\big| < \tau$, che chiaramente è una condizione più debole rispetto alla precedente siccome

$$\big|\, ||\varphi^{(k+1)}|| - ||\varphi^{(k)}|| \,\big| \leq ||\varphi^{(k+1)} - \varphi^{(k)}||,$$

ma per τ sufficientemente piccoli, ci si aspetta che la differenza fra i due termini sia "piccola" perché fra due iterazioni successive la distanza fra gli elementi diventa sempre più piccoli (siccome convergono verso la soluzione esatta). Fare quest'ultima operazione snellisce sensibilmente il programma, siccome ci permette di memorizzare solamente l'integrale sul volume della precedente iterazione e la matrice del nuovo potenziale, piuttosto che tenere in memoria sia la matrice del nuovo potenziale che della vecchia computazione. Oltre a ciò, possiamo pensare di utilizzare il metodo SOR (*Successive over Relaxation*) che è "imparentato" al metodo usato qua: tale metodo consiste nel calcolare la soluzione successiva usando

$$\varphi_{j,(k+1)}^i = \omega \bar{\varphi}_i^{j,(k)} + (1-\omega)\varphi_i^{j,(k)}.$$

Dove, nel nostro caso, $\bar{\varphi}_i^{j,(k)} = \frac{1}{4}(\varphi_{i+1}^{j,(k)} + \varphi_{i-1}^{j,(k)} + \varphi_i^{j+1,(k)} + \varphi_i^{j-1,(k)})$ e $\omega \in (0, 2)$. Scriviamo il codice

```
import numpy as np
import numba
import matplotlib.pyplot as plt
import time

@numba.jit("Tuple([f8[:, :], i8])(f8[:, :], f8, f8)", nopython=True, nogil=True, parallel=True)
def sol_potential(potential, tau, w):
    """
    Compute the solution of 2D Laplacian

    Params
    -----
    potential: matrix
        matrix representing the grid of the point in which the potential is valued
    tau:
        difference between two successive iteration
    w:
        relaxiation parameter for the SOR method
    Output:
    ------
    (potential, index): tuple
        tuple contained as first element a matrix in which the potential is valued and the number of iterations needed
    """
    length = len(potential[0])
    potential_0 = np.zeros((length, length), dtype="float64")
    index = 0
    integ0 = 0

    while True:
        integ = 0

    for i in numba.prange(1, length-1):
            for j in numba.prange(1, length-1):
                potential[j][i] = w * 0.25 * (potential[j+1][i] + potential[j-1][i] + potential[j][i+1] + potential[j][i-1]) + (1-w)*potential[j][i]
                integ += np.abs(potential[j][i])

    if  np.abs(integ - integ0)/((length)*(length)) > tau:
            integ0 = integ
            index += 1
        else:
            break

  return (potential, index)

# Boundary counditions
edge = np.linspace(-1, 1, 300)
bordo_supy = np.cos(np.pi * edge / 2)
bordo_infy = edge**4
bordo_supx = 1/(np.e**-1 - np.e) * (np.exp(edge)-np.e)
bordo_infx = 0.5 * (edge**2 - edge)

# Creation of the grid
potenziale = np.zeros((300, 300))
potenziale[0, :] = bordo_infy
potenziale[-1, :] = bordo_supy
potenziale[:, 0] = bordo_infx
potenziale[:, -1] = bordo_supx
xv, yv = np.meshgrid(edge, edge)

# Computing the solution and the time needed
start = time.time()
result = sol_potential(potenziale, tau=1e-8, w=1.99)
end = time.time()

# Graphical settings
fig, ax = plt.subplots(1, 1, figsize=(8,6))
clr_plot = ax.contourf(xv, yv, result[0], 30)
ax.set_xlabel('x/a', fontsize=18)
ax.set_ylabel('y/a', fontsize=18)
cbar = fig.colorbar(clr_plot)
cbar.set_label('V/V0', fontsize=18)
cbar.ax.tick_params(labelsize=10)
```

```
ax.set_title('Potenziale', fontsize=18)
plt.savefig("potenziale_senzablocco_GSA.png")
print(f"Sono state necessarie {result[1]} iterazioni e {end-start} secondi")
plt.show()

[Output]
Sono state necessarie 735 iterazioni e 0.14688754081726074 secondi

```

Allo stesso modo, si può implementare questo metodo anche nel caso della regione con un blocco a potenziale costante, il cui codice lo carico su GitHub (tanto basta modificare il programma leggermente come fatto qua sopra).

3 Equazione delle corde musicali

Un altro esempio in cui possiamo giovare della parallelizzazione è sicuramente l'equazione delle corde musicali

$$\partial^2_{xx}u - \frac{1}{c^2}\partial^2_{tt}u - \gamma\partial_t u - l^2\partial^4_{xxxx}u = 0,$$

dove γ è il coefficiente di *damping*, ovvero il coefficiente correlato alla dispersione di energia dell'onda, e l^2 il coefficiente di rigidità che ha le unità di misura di $\frac{m^4}{s^2}$ e quantifica la forza esercitata dal corpo dinanzi a delle sollecitazioni (nel caso della corda, la forza che ogni oppone dinanzi al pizzicamento che la fa muovere).
Ragionando come prima, approssimiamo l'equazione tramite derivate discrete, diventando così

$$\begin{aligned}\delta u = &\frac{y^m_{j+1} - 2y^m_j + y^m_{j-1}}{\Delta x^2} - \frac{1}{c^2}\frac{y^{m+1}_j - 2y^m_j + y^m_{j-1}}{\Delta t^2} - \gamma\frac{y^{m+1}_j - y^m_j}{\Delta t}\\ &- l^2\frac{y^m_{j-2} - 4y^m_{j-1} + 4y^m_j - 4y^m_{j+1} + y^m_{j+2}}{\Delta x^4} = 0.\end{aligned}$$

Esplicitando y^m_{i+1} otteniamo che

$$\begin{aligned}y^{m+1}_j = &\left[\frac{1}{c^2\Delta t^2} + \frac{\gamma}{2\Delta t}\right]^{-1}\left[\frac{1}{\Delta x^2}\left(y^m_{j+1} - 2y^m_j + y^m_{j-1}\right) - \frac{1}{c^2\Delta t^2}\left(y^{m-1}_j - 2y^m_j\right) + \right.\\ &\left. + \frac{\gamma}{2\Delta t}y^{m-1}_j - \frac{l^2}{\Delta x^4}\left(y^m_{j-2} - 4y^m_{j-1} + 6y^m_j - 4y^m_{j+1} + y^m_{j+2}\right)\right]\end{aligned}$$

Possiamo vedere come questa equazione sia computazionalmente molto più gravosa rispetto a quella del laplaciano, siccome il valore dell'iterazione successiva obbliga il computer ad accedere a y^m_j e le sue celle limitrofe più volte rispetto al caso sopra studiato. Senza ottimizzazioni derivanti dalla compilazione, che evitano riletture ripetute della memoria e altri fattori, il processo diventa molto lento. Questo è uno dei casi dove la compilazione `jit` (che evita la rilettura ripetuta delle celle), la parallelizzazione e altre, eventuali, ottimizzazioni diventano quasi un must per migliorare significativamente il tempo di esecuzione di questi algoritmi. Studiando l'operatore differenziale approssimato si ottiene che la condizione di stabilità si ha per $\frac{c\Delta t}{\Delta x} < 1$: la condizione più stringente per la stabilità, sebbene ci siano derivate di ordine quartico, si ottiene per $l^2 = \gamma = 0$, tornando ad essere la semplice equazione delle onde. Dunque possiamo discretizzare il nostro spazio con $N_x = d = 0.7$ m, $\Delta x = 0.07$ mm, $\Delta t = 5 \cdot 10^{-6}$ s. Pertanto

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from matplotlib.animation import PillowWriter
from scipy.io import wavfile
from IPython.display import Audio
import numba


# Values of the grid
Nx = 101
Nt = 500000
L = 0.7
dx = L/(Nx-1)
f = 440
c = 2*L*f
dt = 5e-6
l=5e-5
gamma=5e-5

```

```
# Boundary conditions
ya = np.linspace(0, 0.01, 70)
yb = np.linspace(0.01, 0, 31)
y0 = np.concatenate([ya, yb])


# Creation of the grid
sol = np.zeros((Nt, Nx))
sol[0] = y0
sol[1] = y0


@numba.jit("f8[:,:](f8[:,:], i8, i8, f8, f8, f8, f8)", nopython=True, nogil=True)
def compute_sol(d, times, length, dt, dx, l, gamma):
  """
  Compute the solution via finite difference method

  Params
  -----
    d: 2d array
      Matrix representing the space-time grid where the solution is evaluated
    times: int
      Height of the grid (number of points the time is discretized)
    length: int
      Basis of the grid (number of points the x is discretized)
    dt: float
      Difference between two consecutive 'time' points
    dx: float
      Difference between two consecutive 'x' points
    l: float
      Damping coefficient
    gamma: float
      Stifness term
  """
  for t in range(1, times-1):
    for i in range(2, length-2):
      outer_fact = (1/(c**2 * dt**2) + gamma/(2*dt))**(-1)
      p1 = 1/dx**2 * (d[t][i-1] - 2*d[t][i] + d[t][i+1])
      p2 = 1/(c**2 * dt**2) * (d[t-1][i] - 2*d[t][i])
      p3 = gamma/(2*dt) * d[t-1][i]
      p4 = l**2 / dx**4 * (d[t][i+2] - 4*d[t][i+1] + 6*d[t][i] - 4*d[t][i-1] + d[t][i-2])
      d[t+1][i] = outer_fact * (p1 - p2 + p3 - p4)
  return d

# Computing the solution and plotting some frames
sol = compute_sol(sol, Nt, Nx, dt, dx, l, gamma)
plt.plot(sol[500], label="Frame 500")
plt.plot(sol[1000], label="Frame 1000")
plt.legend()
plt.savefig("frame1000.png")
plt.show()

# Generating an animation
def animate(i):
    ax.clear()
    ax.plot(sol[i*10])
    ax.set_ylim(-0.01, 0.01)

fig, ax = plt.subplots(1,1)
ax.set_ylim(-0.01, 0.01)
ani = animation.FuncAnimation(fig, animate, frames=500, interval=50)
ani.save('string.gif',writer='pillow',fps=20)
```

Può essere interessante vedere il grafico di qualche soluzione a qualche istante:

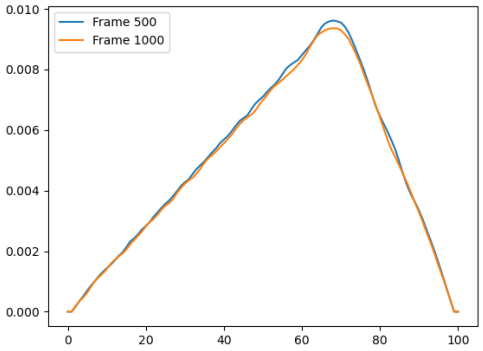


Figura 3: Grafico della soluzione al "frame" 500 e al "frame" 1000. Anche se non sembra, una corda musicale quando viene pizzicata assume queste "posizioni", tuttavia il nostro occhio non è il grado di vederle.

Possiamo fare molto di più: possiamo persino produrre dei file audio. Per fare questo utilizziamo il fatto che la nostra soluzione sarà una combinazione di seni, ma siccome la nostra soluzione sarà appartenente a $L^2(0,d)$ allora possiamo estrarre il coefficiente che moltiplica il termine $\sin(\frac{n\pi x}{L})$ usando la norma L^2

$$c_n(t) \propto \int_0^L y(x,t)\sin\left(\frac{n\pi x}{L}\right)dx \approx \sum_{i=1}^{N_x} y_i^j \sin\left(\frac{n\pi x}{L}\right).$$

Dove si è usata la convenzione (fatta anche prima) che $y_i^t = y(x_i,t_j)$ ovvero la soluzione all'equazione valutata nel punto della griglia di coordinate (x_i,t_j). Chiaramente il coefficiente sarà in funzione del tempo (siccome il segnale varia nel tempo), pertanto dovremmo fare il calcolo a t fissato. Oltre a questo, per produrre un file audio è necessario campionare il segnale e non è necessario prendere tutte le armoniche del segnale: ciò è dovuto al fatto che non siamo in grado di riprodurre perfettamente la "bontà" del segnale, anche se generato dal computer, per come vengono prodotti i suoni dalla scheda audio. Inoltre, non ha senso prendere tutte le armoniche che hanno un periodo $T <$ `dt` (perché chiaramente non vengono simulati correttamente). Quindi è sufficiente restringersi alle prime 10 armoniche che, solitamente, sono quelle maggiormente influenti e, per come abbiamo preso il `dt`, campionare il segnale ogni 10 punti 'temporali' (questo è necessario per generare un `.wav` con un sampling rate di 20 kHz).

```
def get_integral_fast(n):
  """
    Computing the coefficient of the n-th harmonic of the signal in all the point of 'time'

    Params
    -----
      n: int
        number of the harmonics we want to find the coefficient

    Return
    -----
      arr: 1darray
        array cointaing in every cell the value that signal in
  """
  sin_arr = np.sin(n*np.pi*np.linspace(0,1,101))
  return np.multiply(sol, sin_arr).sum(axis=1) # Sommiamo sulle x, ovvero sulle 'righe'

# Estraiamo solamente le prime 10 armoniche tramite questa list comprension
hms = [get_integral_fast(n) for n in range(10)]

# Campioniamo l'ampiezzia del segnale campionando temporalmente dei punti a distanza di 10
tot = sum(hms)[::10] # compute the instananeous value of the audio signal
tot = tot.astype(np.float32)
wavfile.write('la.wav',20000,tot)

```

Tramite il modulo `wavefile` della libreria `scipy.io` unire i suoni prodotti dalla sovrapposizione di due "note" da noi generate: runnando il programma mettendo come $f = 262$ Hz produrremo un DO4, con $f = 330$ Hz un MI4 e con $f = 392$ Hz produrremo un SOL4: con tali note è possibile creare l'accordo di DO maggiore e possiamo sovrapporre i segnali con

```
c = wavfile.read('do4.wav')[1]
e = wavfile.read('mi4.wav')[1]
```

```
g = wavfile.read('sol4.wav')[1]
wavfile.write('c_maj4.wav', 20000, 2 * c + 2 * e+ 2 * g)
c = c.astype(np.float32)
e = e.astype(np.float32)
g = g.astype(np.float32)
Audio('c_maj4.wav')

```