



# Implementazione e test di programmi di Calcolo Numerico

---



# Propagazione degli errori nel Calcolo Numerico

---

# Condizionamento della somma - implementazione

```
from random import random

#inserimento dati esatti
x = float(input("Inserire dato x: "))
y = float(input("Inserire dato y: "))

#costruzione dati perturbati
x_pert = x + x*random()*1.0e-8
y_pert = y + y*random()*1.0e-8

#calcolo la somma esatta e somma perturbata
s = x + y
s_pert = x_pert + y_pert
print("\nSomma sui dati esatti:\n%.20f + %.20f = %.20f" % (x, y, s))
print("Somma sui dati perturbati:\n%.20f + %.20f = %.20f" % (x_pert, y_pert, s_pert))

#calcolo degli errori relativi
err_x = abs(x-x_pert)/abs(x)
err_y = abs(y-y_pert)/abs(y)
err_s = abs(s-s_pert)/abs(s)

#calcolo i fattori di amplificazione della somma
ampl_x = abs(x)/abs(x+y)
ampl_y = abs(y)/abs(x+y)

#stampo i risultati
print("\n-----Errori in input-----")
print("Errore su x = %e" % err_x)
print("Errore su y = %e" % err_y)
print("\n-----Errori in output-----")
print("Fattore di amplificazione |x|/|x+y|: %e" % ampl_x)
print("Fattore di amplificazione |y|/|x+y|: %e" % ampl_y)
print("Errore sulla somma = %e" % err_s)
```

# Condizionamento della somma - test

## Somma tra numeri non troppo:

La somma è ben condizionata quando si effettua tra numeri abbastanza distanti, infatti si può notare che i fattori di amplificazione sono piccoli e quindi l'errore sul risultato non viene amplificato molto.

Differenza tra numeri vicini molto:

```
Inserire dato x: 12.76352
Inserire dato y: -12.76351
Somma sui dati esatti:
12.76351999999999975444 + -12.76351000000000013301 = 0.00000999999999962142
Somma sui dati perturbati:
12.76352006004300321251 + -12.76351002627324149330 = 0.00001003376976171921
-----Errori in input-----
Errore su x = 4.704267e-09
Errore su y = 2.058465e-09

-----Errori in output-----
Fattore di amplificazione |x|/|x+y|: 1.276352e+06
Fattore di amplificazione |y|/|x+y|: 1.276351e+06
Errore sulla somma = 3.376976e-03
```

In questo caso i fattori di amplificazione sono più alti perché la somma è mal condizionata quando si esegue una differenza tra numeri vicini.

Infatti si può notare che l'errore sul risultato è più elevato rispetto al precedente test.

# Condizionamento del prodotto - implementazione

```
from random import random

#inserimento dati esatti
x = float(input("Inserire dato x: "))
y = float(input("Inserire dato y: "))

#costruzione dati perturbati
x_pert = x + x*random()*1.0e-4
y_pert = y + y*random()*1.0e-4

#calcolo prodotto esatto e prodotto perturbato
s = x * y
s_pert = x_pert * y_pert
print("\nProdotto sui dati esatti: %f + %f = %f" % (x, y, s))
print("Prodotto sui dati perturbati: %f + %f = %f" % (x_pert, y_pert, s_pert))

#calcolo degli errori
err_x = abs(x-x_pert)/abs(x)
err_y = abs(y-y_pert)/abs(y)
err_s = abs(s-s_pert)/abs(s)

#stampo i risultati
print("\n-----Errori in input-----")
print("Errore su x = %f" % err_x)
print("Errore su y = %f" % err_y)
print("\n-----Errori in output-----")
print("Errore sul prodotto = %f" % err_s)
```

# Condizionamento della prodotto - test

Prodotto tra numeri una perturbazione dell'ordine di 1.0e-4:

```
Inserire dato x: 4  
Inserire dato y: 3  
  
Prodotto sui dati esatti: 4.000000 + 3.000000 = 12.000000  
Prodotto sui dati perturbati: 4.000083 + 3.000265 = 12.001311  
  
-----Errori in input-----  
Errore su x = 2.087357e-05  
Errore su x = 8.840634e-05  
  
-----Errori in output-----  
Errore sul prodotto = 1.092818e-04
```

Prodotto tra numeri una perturbazione dell'ordine di 1.0e-12:

```
Inserire dato x: 4  
Inserire dato y: 3  
  
Prodotto sui dati esatti: 4.000000 + 3.000000 = 12.000000  
Prodotto sui dati perturbati:  
4.0000000000295941049 + 3.0000000000288347124 = 12.0000000002041211644  
  
-----Errori in input-----  
Errore su x = 7.398526e-13  
Errore su x = 9.611571e-13  
  
-----Errori in output-----  
Errore sul prodotto = 1.701010e-12
```

Il prodotto è una operazione ben condizionata infatti l'errore sul risultato risulta essere circa la somma degli errori sui dati di input.

# Condizionamento del calcolo di funzione - implementazione

Primo test:

```
np.log(x)
```

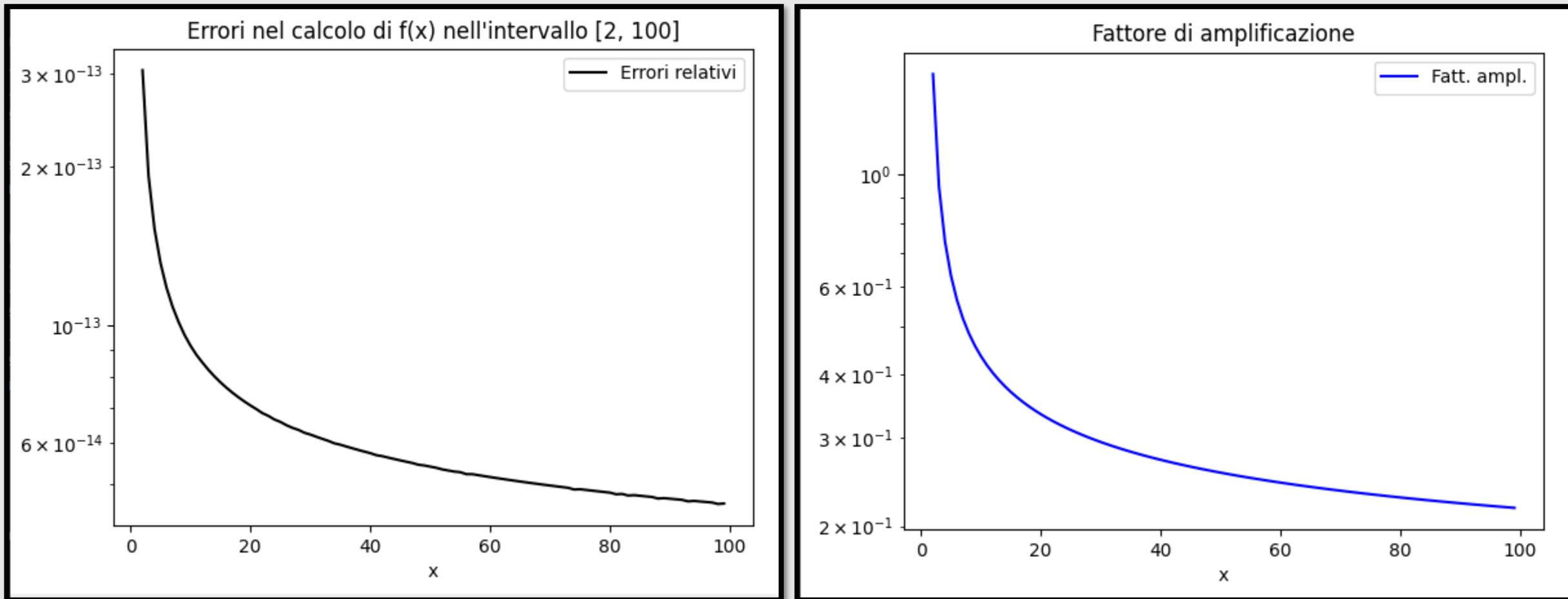
```
1  from random import random
2  from scipy.misc import derivative
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  def fun(x):
7      y = ←
8      return y
9
10 #intervallo di punti in cui calcolare la funzione
11 a = 2; b = 100
12 x = np.array(range(a, b))
13
14 #applico una perturbazione ai dati
15 x_pert = np.full(len(x), x + x*random()*1.0e-12)
16
17 #calcolo della funzione
18 fx = fun(x)
19 fx_pert = fun(x_pert)
20
21 #calcolo fattore di amplificazione dell'errore
22 dx = derivative(fun, x)
23 fatt_ampl = abs(x*dx)/abs(fx)
24
25 #calcolo degli errori
26 err_rel = abs(fx-fx_pert)/abs(fx)
27
28 #visualizzazione grafica dei risultati
29 plt.figure(1)
30 plt.title("Errori nel calcolo di f(x) nell'intervallo [%d, %d]" % (a, b))
31 plt.semilogy(x, err_rel, "k-", label="Errori relativi")
32 plt.xlabel("x")
33 plt.legend()
34 plt.show()
35
36 plt.figure(2)
37 plt.title("Fattore di amplificazione")
38 plt.semilogy(x, fatt_ampl, "b-", label="Fatt. ampl.")
39 plt.xlabel("x")
40 plt.legend()
41 plt.show()
```

Secondo test:

```
np.sqrt(x)
```

# Condizionamento del calcolo di funzione - test

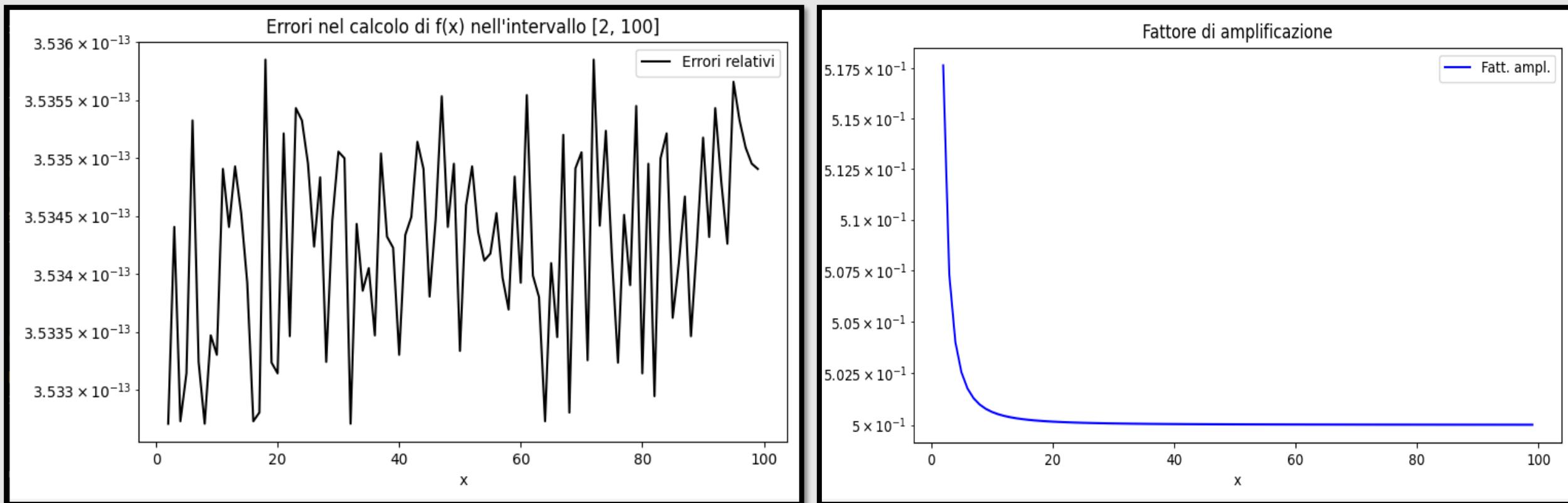
Test logaritmo naturale:



Il calcolo della funzione  $\ln(x)$  è B.C. quando  $\ln(x) > 1$  e M.C. quando  $\ln(x) \leq 1$  infatti si nota che il fattore di amplificazione è più alto per valori di  $x \leq 1$ , quindi in corrispondenza di questi valori l'errore è più grande.

# Condizionamento del calcolo di funzione - test

Test radice quadrata:



Il calcolo della radice quadrata è sempre B.C. infatti si può notare che il fattore di amplificazione dell'errore su tutti i valori di  $x$  è sempre molto simile e l'errore oscilla sempre tra valori dello stesso ordine di grandezza e molto vicini tra loro.

# Aritmetica degli elaboratori

---

# Calcolo epsilon machine

Implementazione:

```
from math import log2, log10

#determino l'epsilon machine
u = 1
while 1+u > 1:
    u_temp = u
    u = u/2

epsilon_machine = u_temp
print("L'epsilon machine vale: %e" % epsilon_machine)

#calcolo la precisione del calcolatore
p = -log2(epsilon_machine)
print("La precisione del calcolatore vale %f senza il bit nascosto." % p)

#calcolo il numero di cifre significative in base 10
q = p*log10(2)
print("Il numero di cifre significative in base 10 è: %f" % q)
```

Il programma calcola l'epsilon machine del calcolatore su cui viene eseguito, poi a partire da questo risultato calcola la precisione del calcolatore (senza contare il bit nascosto) la quale risulterà essere pari al numero di cifre della mantissa previsto dallo standard IEEE 754 per l'architettura del calcolatore su cui è eseguito il programma.

Test:

```
L'epsilon machine vale: 2.220446e-16
La precisione del calcolatore vale 52.000000 senza il bit nascosto.
Il numero di cifre significative in base 10 è: 15.653560
```

# Algebra lineare numerica

---

# Algoritmo di sostituzione avanti - implementazione

```
algoritmi_sostituzione.py × algebra_lineare.py ×

1 import numpy as np
2 import algebra_lineare as al
3
4 # ===== COSTRUZIONE DEL PROBLEMA TEST =====
5 n = 50
6
7 #genero una matrice n*n con valori random tra -10 e 10
8 A = (2*np.random.random((n,n))-1)*10
9 A = np.tril(A) #rendo la matrice triangolare inferiore
10
11 x = np.ones(n) #vettore delle soluzioni
12 b = np.dot(A, x) #vettore dei termini noti
13
14 if al.isSingolare(A):
15     print("Errore: La matrice dei coefficienti potrebbe essere singolare.")
16 else:
17     #calcolo la soluzione con l'algoritmo di sostituzione
18     x_pert = al.sostituzione_avanti(A, b)
19
20     #calcolo l'errore in output
21     err_ass = np.linalg.norm(x-x_pert)
22     err_rel = err_ass/np.linalg.norm(x)
23
24     #stampo i risultati
25     print("Soluzione attesa x =\n", x)
26     print("\nSoluzione calcolata x_pert =\n", x_pert)
27     print('\nAlgoritmo sostituzione avanti con matrice %d x %d' %(n,n))
28     print("Errore assoluto = %e" % err_ass)
29     print("Errore relativo = %e" % err_rel)
```

```
algoritmi_sostituzione.py × algebra_lineare.py ×

1 import numpy as np
2
3 """
4 metodo che verifica se una matrice triangolare o
5 diagonale è potenzialmente singolare
6 """
7 def isSingolare(A):
8     return abs(np.prod(np.diag(A))) <= 1.0e-14
9
10 def sostituzione_avanti(A, b):
11     A = np.copy(A)
12     b = np.copy(b)
13
14     #dimensione della matrice dei coefficienti
15     n = len(A)
16
17     #inizializzo il vettore delle soluzioni
18     x = np.zeros(n)
19
20     #algoritmo di sostituzione in avanti
21     for i in range(n):
22         somma = 0
23         for j in range(i):
24             somma = somma + A[i,j]*x[j]
25         x[i] = (b[i] - somma)/A[i,i]
26
27     return x
```

# Algoritmo di sostituzione avanti - test

Primo test:

```
Soluzione attesa x =
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
Soluzione calcolata x_pert =
[1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
Algoritmo sostituzione avanti con matrice 10 x 10
```

```
Errore assoluto = 1.098343e-11
```

```
Errore relativo = 3.473265e-12
```

Secondo test:

```
Soluzione attesa x =
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
Soluzione calcolata x_pert =
[1.          1.          1.          1.          1.          1.
 1.          1.          1.          1.          1.          1.
 1.          1.          1.          1.          1.          1.
 1.          1.          1.          1.          1.          1.
 1.          1.          1.          1.          1.          1.
 1.          1.          1.00000001 1.          1.00000001 0.99999999
 0.99999999 1.00000002 1.          0.99999999 0.99999998 1.00000002
 0.99999994 0.99999999 1.00000004 1.00000006 0.99999994 0.99999996
 0.99999998 1.00000055]
```

```
Algoritmo sostituzione avanti con matrice 50 x 50
```

```
Errore assoluto = 5.655086e-07
```

```
Errore relativo = 7.997499e-08
```

# Algoritmo di sostituzione indietro - implementazione

algoritmi\_sostituzione.py × algebra\_lineare.py ×

```
1 import numpy as np
2 import algebra_lineare as al
3
4 # ===== COSTRUZIONE DEL PROBLEMA TEST =====
5 n = 15
6
7 #genero una matrice n*n con valori random tra -10 e 10
8 A = (2*np.random.random((n,n))-1)*10
9 A = np.triu(A) #rendo la matrice triangolare superiore
10
11 x = np.ones(n) #vettore delle soluzioni
12 b = np.dot(A, x) #vettore dei termini noti
13
14 if al.isSingolare(A):
15     print("Errore: La matrice dei coefficienti potrebbe essere singolare.")
16 else:
17     #calcolo la soluzione con l'algoritmo di sostituzione
18     x_pert = al.sostituzione_indietro(A, b)
19
20     #calcolo l'errore in output
21     err_ass = np.linalg.norm(x-x_pert)
22     err_rel = err_ass/np.linalg.norm(x)
23
24     #stampo i risultati
25     print("Soluzione attesa x =\n", x)
26     print("\nSoluzione calcolata x_pert =\n", x_pert)
27     print('\nAlgoritmo sostituzione indietro con matrice %d x %d' %(n,n))
28     print("Errore assoluto = %e" % err_ass)
29     print("Errore relativo = %e" % err_rel)
```

algoritmi\_sostituzione.py × algebra\_lineare.py ×

```
28 def sostituzione_indietro(A, b):
29     A = np.copy(A)
30     b = np.copy(b)
31
32     #dimensione della matrice dei coefficienti
33     n = len(A)
34
35     #inizializzo il vettore delle soluzioni
36     x = np.zeros(n)
37
38     #algoritmo di sostituzione all'indietro
39     for i in range(n-1, -1, -1):
40         somma = 0
41         for j in range(i+1, n):
42             somma = somma + A[i,j]*x[j]
43         x[i] = (b[i] - somma)/A[i,i]
44
45     return x
```

# Algoritmo di sostituzione avanti - test

Primo test:

```
Soluzione attesa x =
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
Soluzione calcolata x_pert =
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
Algoritmo sostituzione indietro con matrice 15 x 15
```

```
Errore assoluto = 9.367372e-13
```

```
Errore relativo = 2.418645e-13
```

Secondo test:

```
Soluzione attesa x =
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
Soluzione calcolata x_pert =
[0.99999996 0.99999993 1.00000042 1.00000062 1.00000002 1.00000003
 0.99999977 0.99999995 0.99999989 0.99999998 0.99999993 1.00000002
 1.          1.          1.00000002 1.          1.          1.
 1.          1.          1.          1.          1.          1.          1.
 1.          1.          1.          1.          1.          1.          1.
 1.          1.          1.          1.          1.          1.          1.
 1.          1.          1.          1.          1.          1.          ]]
```

```
Algoritmo sostituzione indietro con matrice 40 x 40
```

```
Errore assoluto = 8.034486e-07
```

```
Errore relativo = 1.270364e-07
```

# Metodo di Eliminazione di Gauss - implementazione 1

```
algebra_lineare.py x metodo_eliminazione_gauss.py x
46 #routine che implementa il M.E.G.
47 def eliminazione_Gauss(A,b):
48     U = np.copy(A)
49     c = np.copy(b)
50
51     n = len(U)
52     for j in range(n-1):
53         for i in range(j+1, n):
54             m = U[i,j]/U[j,j]
55             U[i,j] = 0
56             for k in range(j+1, n):
57                 U[i,k] = U[i,k] - m*U[j,k]
58             c[i] = c[i] - m*c[j]
59     return U, c
60
```

Versione senza pivoting

```
algebra_lineare.py x metodo_eliminazione_gauss.py x
60
61 def eliminazione_Gauss_pivoting(A,b):
62     U = np.copy(A)
63     c = np.copy(b)
64
65     n = len(U)
66     for j in range(n-1):
67         #individuazione elemento pivot
68         pivot = abs(U[j,j])
69         i_pivot = j
70         for i in range(j+1, n):
71             if abs(U[i, j]) > pivot:
72                 pivot = abs(U[i, j])
73                 i_pivot = i
74
75         #eventuale scambio di riga
76         if i_pivot > j:
77             for k in range(j, n):
78                 U_temp = U[j, k]
79                 U[j, k] = U[i_pivot, k]
80                 U[i_pivot, k] = U_temp
81             c_temp = c[j]
82             c[j] = c[i_pivot]
83             c[i_pivot] = c_temp
84
85         #M.E.G.
86         for i in range(j+1, n):
87             m = U[i,j]/U[j,j]
88             U[i,j] = 0
89             for k in range(j+1, n):
90                 U[i,k] = U[i,k] - m*U[j,k]
91             c[i] = c[i] - m*c[j]
92     return U, c
```

Versione  
con  
pivoting

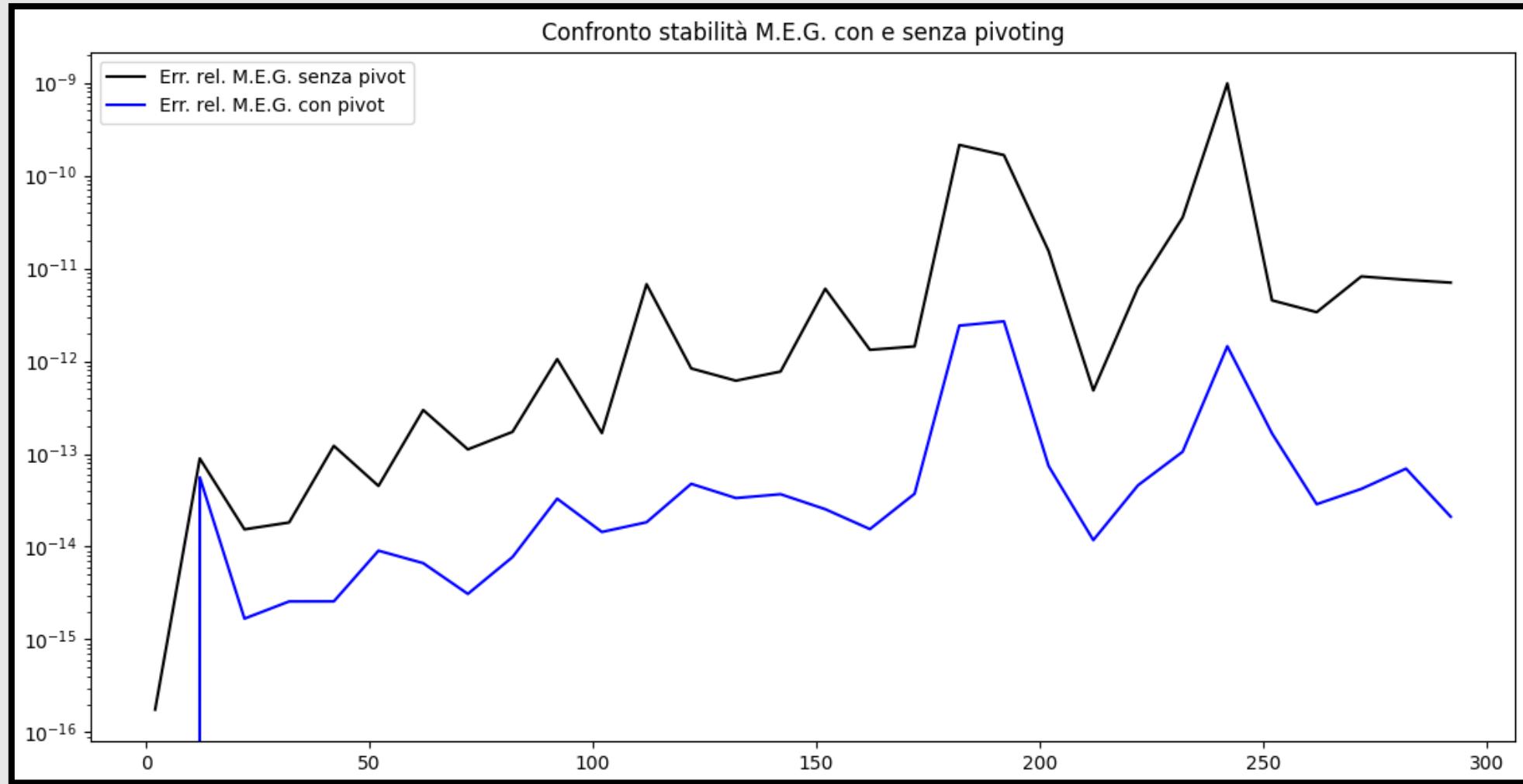
# Metodo di Eliminazione di Gauss - implementazione 2

```
algebra_lineare.py x metodo_eliminazione_gauss.py x

1 import numpy as np
2 import algebra_lineare as al
3 import matplotlib.pyplot as plt
4
5 #costruzione del problema test
6 nmax = 300
7 n_range = range(2, nmax, 10)
8 err_con_pivot = np.zeros(len(n_range))
9 err_senza_pivot = np.zeros(len(n_range))
10 k = 0
11 for n in n_range:
12     #genero una matrice n*n con valori random tra -2 e 2
13     A = (2*np.random.random((n,n)) - 1)*2
14
15     x = np.ones(n) #vettore delle soluzioni attese
16     b = np.dot(A, x) #vettore dei termini noti
17
18     #applicazione M.E.G. con e senza pivoting
19     U,c = al.eliminazione_Gauss(A,b)
20     U2, c2 = al.eliminazione_Gauss_pivoting(A, b)
21
22     """
23     Lo scopo del programma è confrontare gli errori delle due versioni
24     del M.E.G. quindi se una tra le matrici U e U2 è potenzialmente
25     singolare non risolvo nessuno dei due sistemi
26     """
27     if al.isSingolare(U) or al.isSingolare(U2):
28         print("Errore: La matrice dei coefficienti potrebbe essere singolare.")
29     else:
30         #risolvo i sistemi
31         x_pert = al.sostituzione_indietro(U, c)
32         x_pert2 = al.sostituzione_indietro(U2, c2)
33
34         #calcolo errori
35         err_senza_pivot[k] = np.linalg.norm(x-x_pert)/np.linalg.norm(x)
36         err_con_pivot[k] = np.linalg.norm(x-x_pert2)/np.linalg.norm(x)
37         k = k+1
38
39     #visualizzazione grafica dei risultati
40     plt.figure(1)
41     plt.title("Confronto stabilità M.E.G. con e senza pivoting")
42     plt.semilogy(n_range, err_senza_pivot, "k-", label="Err. rel. M.E.G. senza pivot")
43     plt.semilogy(n_range, err_con_pivot, "b-", label="Err. rel. M.E.G. con pivot")
44     plt.xlabel("dim(A)")
45     plt.legend()
46     plt.show()
```

Il programma confronta la stabilità degli algoritmi che implementano il M.E.G. con e senza pivoting, risolvendo una serie di sistemi di equazioni lineari di dimensione crescente.

# Metodo di Eliminazione di Gauss - test



Si noti come l'applicazione del M.E.G. con pivoting comporti notevoli benefici in termini di propagazione degli errori al crescere della dimensione dei sistemi.

# Metodo di Eliminazione di Gauss - implementazione confronto tempi 1

```
algebra_lineare.py gauss_confronto_tempi.py

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import time
4 import algebra_lineare as al
5
6 nmax = 300
7 n_range = range(2, nmax, 10)
8
9 tempo_ottim = np.zeros(len(n_range))
10 tempo_non_ottim = np.zeros(len(n_range))
11 k = 0
12 for n in n_range:
13     A = (2*np.random.random((n, n)) - 1)*2
14     x = np.ones(n)
15     b = np.dot(A, x)
16
17     inizio = time.time()
18     U, c = al.eliminazione_Gauss_pivoting_ottim(A, b)
19     fine = time.time()
20     tempo_ottim[k] = fine - inizio
21
22     inizio = time.time()
23     U2, c2 = al.eliminazione_Gauss_pivoting(A, b)
24     fine = time.time()
25     tempo_non_ottim[k] = fine - inizio
26     k = k+1
27
28 #visualizzazione grafica dei risultati
29 plt.figure(1)
30 plt.title("Confronto tempo di esecuzione M.E.G. ottimizzato e non")
31 plt.plot(n_range, tempo_ottim, "k-", label="Tempi M.E.G. ottim.")
32 plt.plot(n_range, tempo_non_ottim, "b-", label="Tempi M.E.G. non ottim.")
33 plt.xlabel("dim(A)")
34 plt.ylabel("Tempo di esecuzione")
35 plt.legend()
36 plt.show()
```

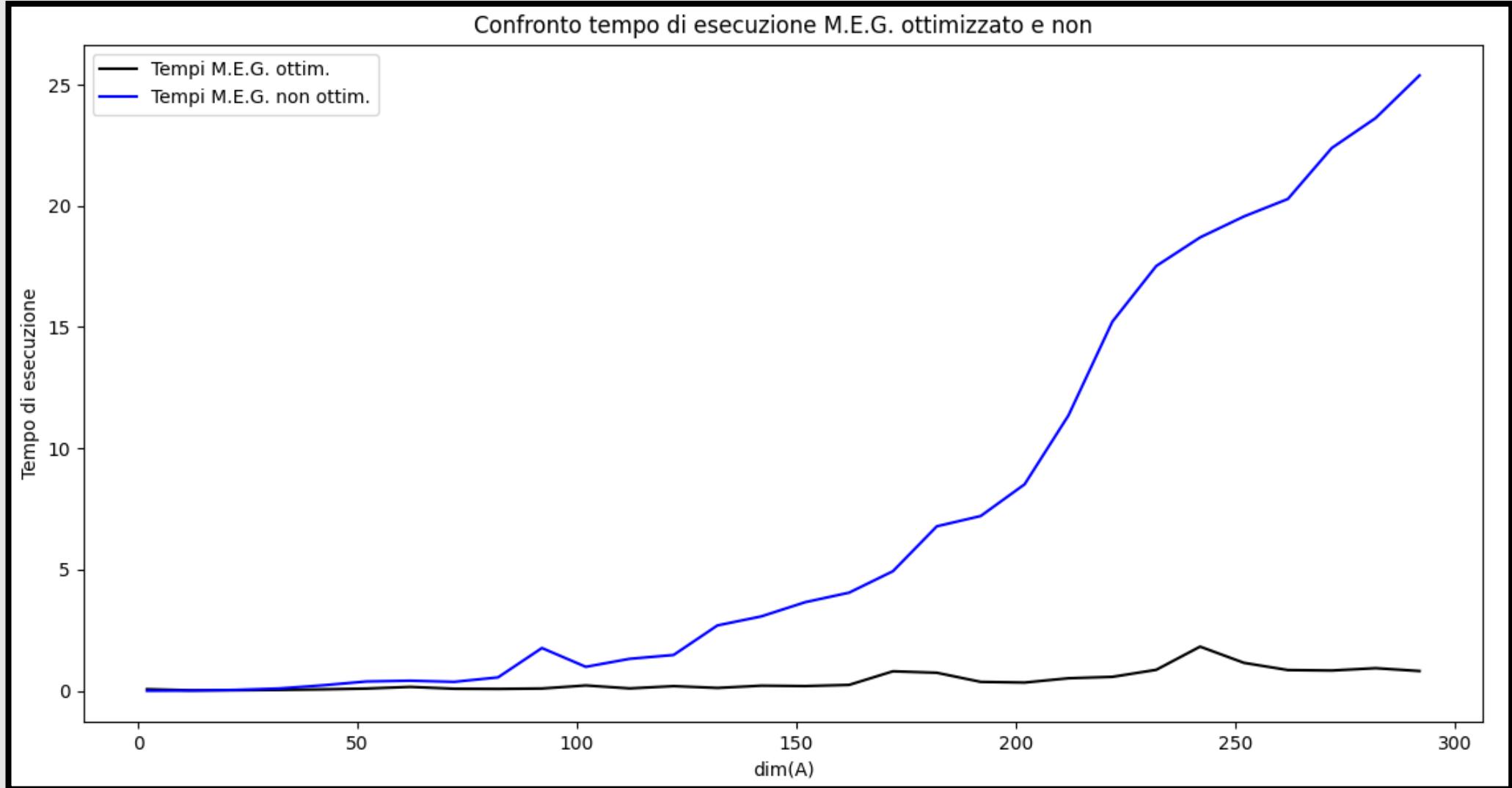
Il programma confronta il tempo di esecuzione del M.E.G. in versione ottimizzata e non ottimizzata nella trasformazione di sistemi di equazioni lineari di dimensione crescente.

# Metodo di Eliminazione di Gauss - implementazione confronto tempi 2

La versione del M.E.G. ottimizzato utilizzata nel programma precedente è la seguente:

```
94     def eliminazione_Gauss_pivoting_ottim(A,b):
95         U = np.copy(A)
96         c = np.copy(b)
97
98         n = len(U)
99         for j in range(n-1):
100             #individuazione elemento pivot
101             i_pivot = np.argmax(abs(U[j:n, j])) + j
102
103             #eventuale scambio di riga
104             if i_pivot > j:
105                 U[[i_pivot, j], :] = U[[j, i_pivot], :]
106                 c[[i_pivot, j]] = c[[j, i_pivot]]
107
108             #M.E.G.
109             for i in range(j+1, n):
110                 m = U[i,j]/U[j,j]
111                 U[i,j] = 0
112                 U[i, j+1:n] = U[i, j+1:n]-m*U[j, j+1:n]
113                 c[i] = c[i] - m*c[j]
114
115         return U, c
```

# Metodo di Eliminazione di Gauss - test confronto tempi



Si noti come la versione ottimizzata del M.E.G. ha prestazioni migliori in termini di tempo al crescere delle dimensioni del sistema di equazioni lineari.

# Fattorizzazione A = LU - implementazione 1

Versione senza pivoting:

```
algebra_lineare.py × fattorizzazione_A_LU.py ×  
116     #implementa la fattorizzazione A = LU senza pivoting  
117     def fattorizzazione_LU(A):  
118         A = np.copy(A)  
119  
120         n = len(A)  
121         for j in range(n-1):  
122             for i in range(j+1, n):  
123                 A[i, j] = A[i, j]/A[j, j]  
124                 for k in range(j+1, n):  
125                     A[i, k] = A[i, k] - A[i, j]*A[j, k]  
126         L = np.tril(A, -1) + np.eye(n)  
127         U = np.triu(A)  
128         return L, U
```

Versione con pivoting:

```
algebra_lineare.py × fattorizzazione_A_LU.py ×  
143     def fattorizzazione_LU_pivoting(A):  
144         A = np.copy(A)  
145  
146         n = len(A)  
147         indice = np.array(range(n))  
148         for j in range(n-1):  
149             #individuazione elemento pivot  
150             pivot = abs(A[j, j])  
151             i_pivot = j  
152             for i in range(j+1, n):  
153                 if abs(A[i, j]) > pivot:  
154                     pivot = abs(A[i, j])  
155                     i_pivot = i  
156  
157             #eventuale scambio di riga  
158             if i_pivot > j:  
159                 for k in range(n):  
160                     A_temp = A[j, k]  
161                     A[j, k] = A[i_pivot, k]  
162                     A[i_pivot, k] = A_temp  
163                     i_temp = indice[j]  
164                     indice[j] = indice[i_pivot]  
165                     indice[i_pivot] = i_temp  
166  
167             #fattorizzazione LU  
168             for i in range(j+1, n):  
169                 A[i, j] = A[i, j]/A[j, j]  
170                 for k in range(j+1, n):  
171                     A[i, k] = A[i, k] - A[i, j]*A[j, k]  
172         L = np.tril(A, -1) + np.eye(n)  
173         U = np.triu(A)  
174         return L, U, indice
```

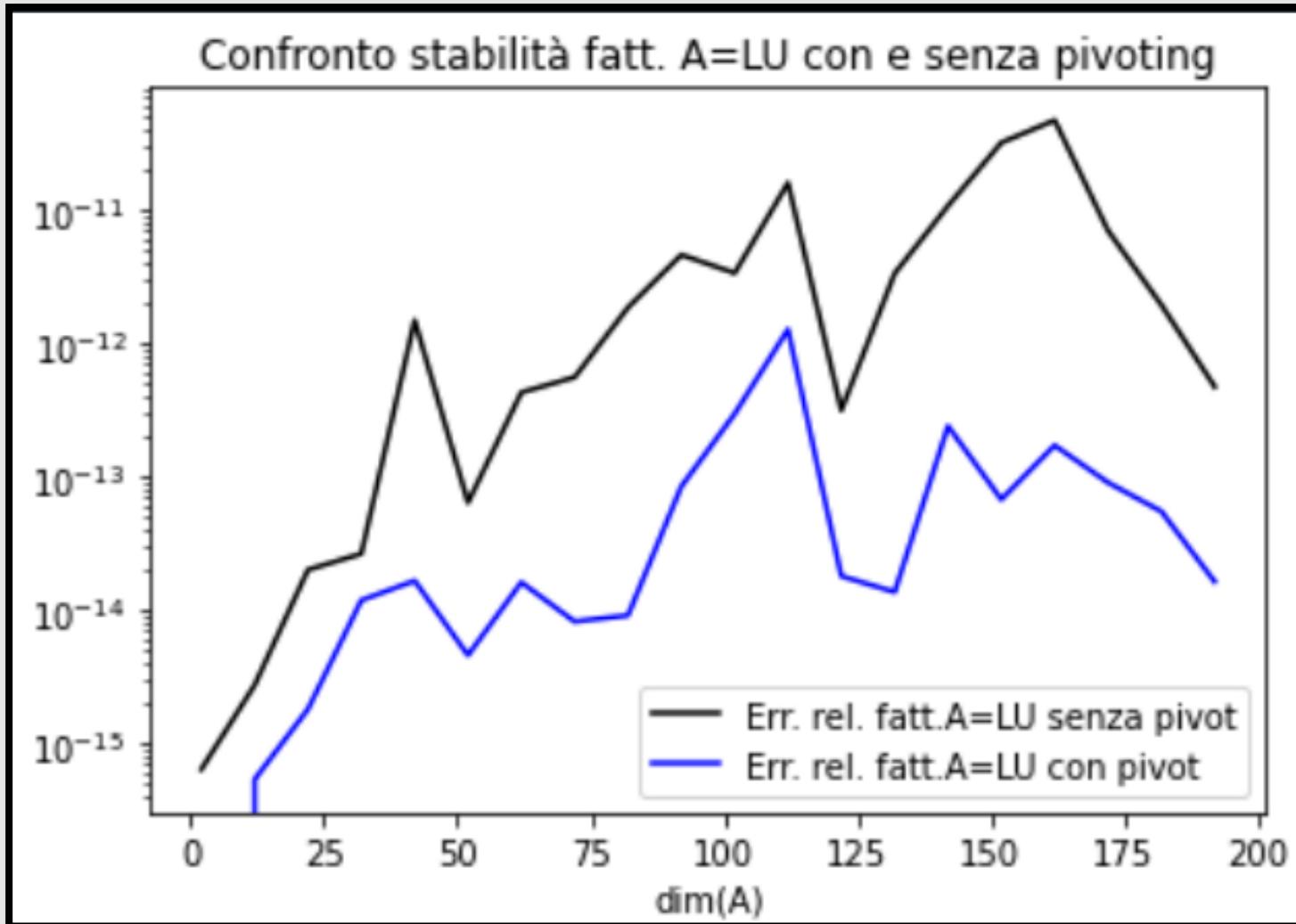
# Fattorizzazione A = LU - implementazione 2

```
algebra_lineare.py x fattorizzazione_A_LU.py x

1 import numpy as np
2 import algebra_lineare as al
3 import matplotlib.pyplot as plt
4
5 #costruzione del problema test
6 nmax = 200
7 n_range = range(2, nmax, 10)
8 err_con_pivot = np.zeros(len(n_range))
9 err_senza_pivot = np.zeros(len(n_range))
10 k = 0
11 for n in n_range:
12     #genero una matrice n*n con valori random tra -2 e 2
13     A = (2*np.random.random((n,n)) - 1)*2
14
15     x = np.ones(n) #vettore delle soluzioni attese
16     b = np.dot(A, x) #vettore dei termini noti
17
18     #applicazione M.E.G. con e senza pivoting
19     L, U = al.fattorizzazione_LU(A)
20     L2, U2, indice = al.fattorizzazione_LU_pivoting(A)
21
22     if al.isSingolare(U) or al.isSingolare(U2):
23         print("Errore: La matrice dei coefficienti potrebbe essere singolare.")
24     else:
25         #risolvo il primo sistema
26         y = al.sostituzione_avanti(L, b)
27         x_pert = al.sostituzione_indietro(U, y)
28
29         #risolvo il secondo sistema
30         b = b[indice]
31         y2 = al.sostituzione_avanti(L2, b)
32         x_pert2 = al.sostituzione_indietro(U2, y2)
33
34         #calcolo errori
35         err_senza_pivot[k] = np.linalg.norm(x-x_pert)/np.linalg.norm(x)
36         err_con_pivot[k] = np.linalg.norm(x-x_pert2)/np.linalg.norm(x)
37         k = k+1
38
39 #visualizzazione grafica dei risultati
40 plt.figure(1)
41 plt.title("Confronto stabilità fatt. A=LU con e senza pivoting")
42 plt.semilogy(n_range, err_senza_pivot, "k-", label="Err. rel. fatt.A=LU senza pivot")
43 plt.semilogy(n_range, err_con_pivot, "b-", label="Err. rel. fatt.A=LU con pivot")
44 plt.xlabel("dim(A)")
45 plt.legend()
46 plt.show()
```

Il programma confronta la stabilità degli algoritmi che implementano la fattorizzazione A = LU con e senza pivoting, risolvendo una serie di sistemi di equazioni lineari di dimensione crescente.

# Fattorizzazione $A = LU$ - test



Si noti come l'applicazione della fattorizzazione  $A=LU$  con pivoting comporti notevoli benefici in termini di propagazione degli errori al crescere della dimensione dei sistemi.

# Fattorizzazione A = LU - implementazione confronto tempi 1

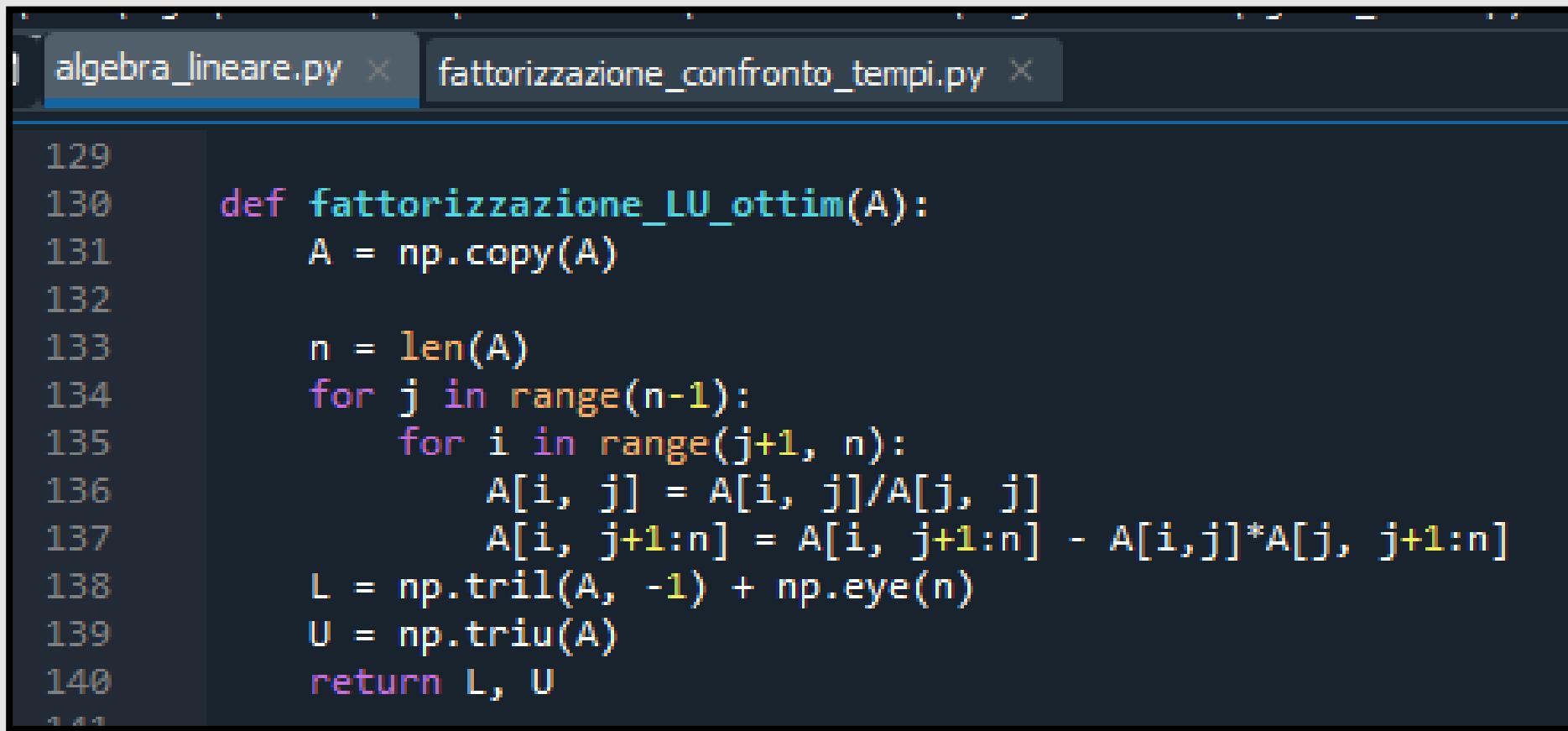
```
algebra_lineare.py × fattorizzazione_confronto_tempi.py ×

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import time
4 import algebra_lineare as al
5
6 nmax = 300
7 n_range = range(2, nmax, 10)
8
9 tempo_ottim = np.zeros(len(n_range))
10 tempo_non_ottim = np.zeros(len(n_range))
11 k = 0
12 for n in n_range:
13     A = (2*np.random.random((n, n)) - 1)*2
14     x = np.ones(n)
15     b = np.dot(A, x)
16
17     inizio = time.time()
18     L, U = al.fattorizzazione_LU_ottim(A)
19     fine = time.time()
20     tempo_ottim[k] = fine - inizio
21
22     inizio = time.time()
23     L2, U2 = al.fattorizzazione_LU(A)
24     fine = time.time()
25     tempo_non_ottim[k] = fine - inizio
26     k = k+1
27
28 #visualizzazione grafica dei risultati
29 plt.figure(1)
30 plt.title("Confronto tempo di esecuzione fatt. A=LU ottimizzato e non")
31 plt.plot(n_range, tempo_ottim, "k-", label="Tempi fatt. A=LU ottim.")
32 plt.plot(n_range, tempo_non_ottim, "b-", label="Tempi fatt. A=LU non ottim.")
33 plt.xlabel("dim(A)")
34 plt.ylabel("Tempo di esecuzione")
35 plt.legend()
36 plt.show()
```

Il programma confronta il tempo di esecuzione della fattorizzazione A = LU ottimizzata e non ottimizzata nella trasformazione di sistemi di equazioni lineari di dimensione crescente.

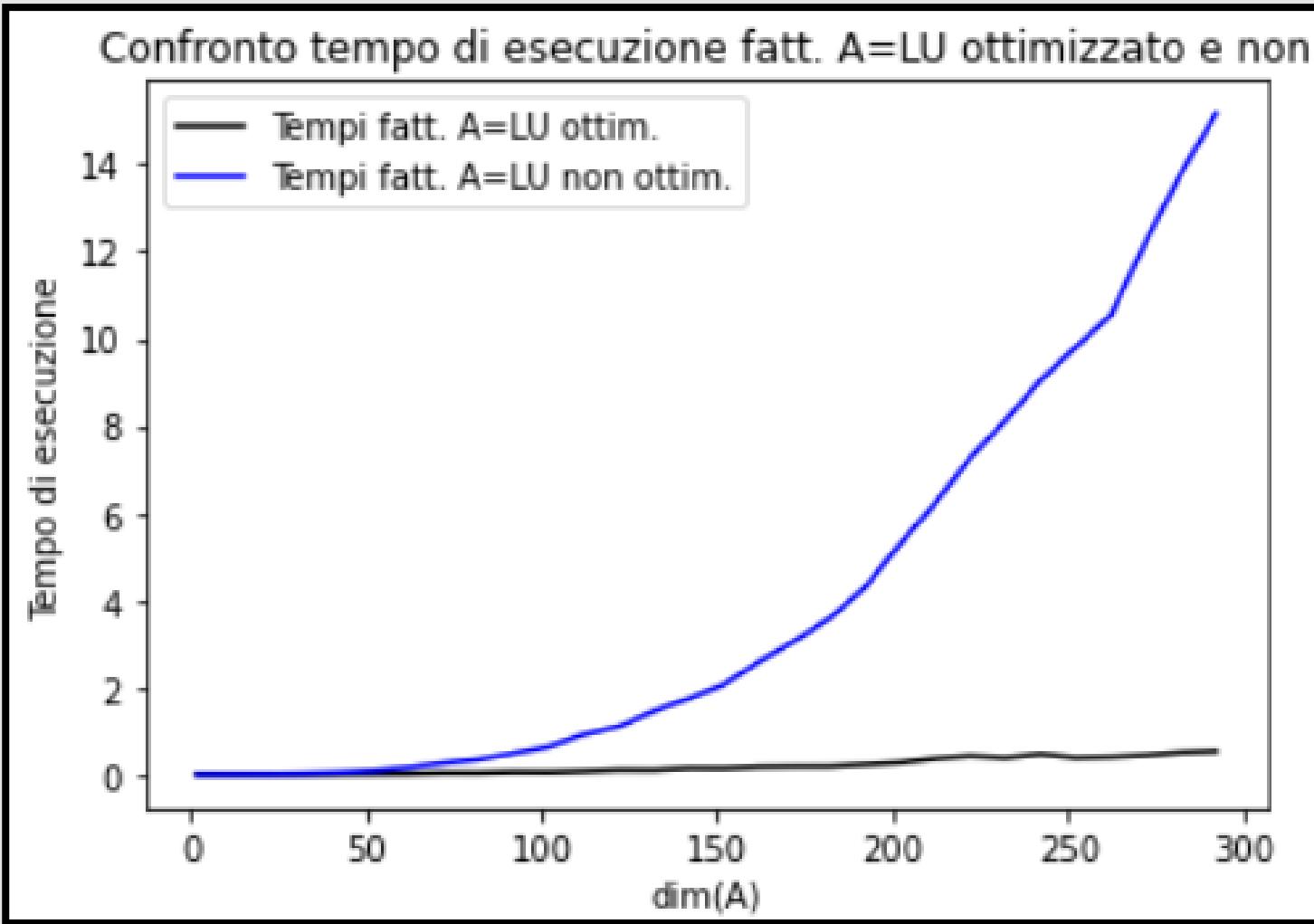
# Fattorizzazione A = LU - implementazione confronto tempi 2

L'algoritmo della fattorizzazione  $A = LU$  ottimizzato utilizzato nel programma precedente è il seguente:



```
129
130     def fattorizzazione_LU_ottim(A):
131         A = np.copy(A)
132
133         n = len(A)
134         for j in range(n-1):
135             for i in range(j+1, n):
136                 A[i, j] = A[i, j]/A[j, j]
137                 A[i, j+1:n] = A[i, j+1:n] - A[i,j]*A[j, j+1:n]
138         L = np.tril(A, -1) + np.eye(n)
139         U = np.triu(A)
140
141
```

# Fattorizzazione $A = LU$ - test confronto tempi



Si noti come la versione ottimizzata della fattorizzazione  $A = LU$  ha prestazioni migliori in termini di tempo al crescere delle dimensioni del sistema di equazioni lineari.

# Confronto tra Fattorizzazione A = LU & M.E.G. - implementazione 1

```
algebra_lineare.py x confronto_gauss_fattorizzazione.py x

1 import numpy as np
2 import algebra_lineare as al
3 import time
4
5 # dimensione della matrice
6 n = 100
7
8 #creo una matrice con valori random tra -10 e 10
9 A = (2*np.random.random((n, n)) - 1)*10
10
11 #creo la matrice identità
12 I = np.eye(n)
13
14 #inizializzo le matrici inverse
15 X = np.zeros((n,n)) #sarà calcolata con la fatt. A=LU e gli alg. di sost.
16 X2 = np.zeros((n,n)) #sarà calcolata con Gauss e l'alg. di sost. indietro
17
18 inizio1 = time.time()
19 #Fattorizzo la matrice A nelle matrici L e U che userò per risolvere i vari sistemi.
20 L, U, indice = al.fattorizzazione_LU_pivoting(A)
21 fine1 = time.time()
22
23 if al.isSingolare(L) or al.isSingolare(U):
24     print("Errore: la matrice è probabilmente singolare")
25 else:
26
27     # ====== CALCOLO INVERSA USANDO LE TRASFORMAZIONI L e U ======
28     inizio2 = time.time()
29     for j in range(n):
30         b = I[:, j]
31         b = b[indice]
32         y = al.sostituzione_avanti(L, b)
33         X[:, j] = al.sostituzione_indietro(U, y)
34     fine2 = time.time()
35
36     tempo_fatt = (fine1 - inizio1) + (fine2 - inizio2)
37     # ======
```

Il programma calcola la matrice inversa di una matrice A tramite la risoluzione di sistemi di equazioni lineari.

La matrice inversa viene calcolata in due modi:

- Usando la fattorizzazione A=LU per trasformare una sola volta la matrice A e quindi calcolando le soluzioni dei sistemi tramite gli algoritmi di sostituzione.
- Trasformando la matrice A, con il M.E.G., tante volte quanti sono i sistemi da risolvere per calcolare la matrice inversa.

Inoltre, viene calcolato il tempo impiegato in entrambi i casi al fine di evidenziarne la differenza e vengono confrontati gli errori commessi.

# Confronto tra Fattorizzazione A = LU & M.E.G. - implementazione 2

```
38
39      # ====== CALCOLO INVERSA USANDO GAUSS ======
40      inizio = time.time()
41      for j in range(n):
42          U, c = al.eliminazione_Gauss_pivoting_ottim(A, I[:, j])
43          X2[:, j] = al.sostituzione_indietro(U, c)
44      fine = time.time()
45
46      tempo_gauss = (fine - inizio)
47      # ======
48
49      #visualizzazione risultati
50      print("Test eseguito su una matrice %d x %d" %(n,n))
51      print("Tempo impiegato con fatt. e alg. di sost.: %f" %tempo_fatt)
52      print("Tempo impiegato con Gauss e alg. di sost. indietro: %f" %tempo_gauss)
```

## Confronto tra Fattorizzazione A = LU & M.E.G. - test

Test eseguito su una matrice 10 x 10

Tempo impiegato con fatt. e alg. di sost.: 0.009993

Tempo impiegato con Gauss e alg. di sost. indietro: 0.057965

Test eseguito su una matrice 50 x 50

Tempo impiegato con fatt. e alg. di sost.: 0.447723

Tempo impiegato con Gauss e alg. di sost. indietro: 2.840250

Test eseguito su una matrice 100 x 100

Tempo impiegato con fatt. e alg. di sost.: 3.843436

Tempo impiegato con Gauss e alg. di sost. indietro: 16.509180

Si noti la maggiore efficienza in termini di tempo della fattorizzazione A=LU, la quale viene applicata una sola volta, piuttosto che l'utilizzo del M.E.G. per ogni sistema.

# Metodo Iterativi: Jacobi e Gauss-Seidel - implementazione 1

Metodo di Jacobi:

```
algebra_lineare.py x metodi_iterativi.py x

214     #metodo iterativo di Jacobi per risoluzione di sistemi di equazioni lineari
215     def Jacobi(A, b, x0, tol, kmax):
216         A = np.copy(A)
217         b = np.copy(b)
218         n = len(A)
219
220         #inizializzo vettore delle soluzioni
221         x1 = np.zeros(n)
222         #inizializzo vettore dei termini noti
223         c = np.ones(n)
224
225         k = 0
226         stop = False
227         while not(stop) and k <= kmax:
228             #risoluzione del sistema Mx = c
229             for i in range(n):
230                 c[i] = b[i]
231                 for j in range(n):
232                     if j != i:
233                         c[i] = c[i] - A[i, j]*x0[j]
234                 x1[i] = (c[i]/A[i,i])
235
236             #applico i criteri di arresto
237             r1 = b - np.dot(A, x1)
238             arresto_r1 = np.linalg.norm(r1)/np.linalg.norm(b)
239             diff_succ = np.linalg.norm(x1-x0)/np.linalg.norm(x1)
240             stop = (arresto_r1 < tol) and (diff_succ < tol)
241
242             x0 = np.copy(x1)
243             k = k+1
244
245             if not(stop):
246                 print("Attenzione: accuratezza %e non raggiunta in %d operazioni." % (tol, kmax))
247             return x1
```

Metodo di Gauss-Seidel:

```
algebra_lineare.py x metodi_iterativi.py x

250     #metodo iterativo di Gauss-Seidel per risoluzione di sistemi di equazioni lineari
251     def Gauss_Seidel(A, b, x0, tol, kmax):
252         A = np.copy(A)
253         b = np.copy(b)
254         n = len(A)
255
256         #inizializzo vettore delle soluzioni
257         x1 = np.zeros(n)
258
259         k = 0
260         stop = False
261         while not(stop) and k <= kmax:
262             #calcolo il vettore dei termini noti del sistema Mx = c
263             for i in range(n):
264                 somma = 0
265                 for j in range(i):
266                     somma = somma + A[i,j]*x1[j]
267                 for j in range(i+1, n):
268                     somma = somma + A[i,j]*x0[j]
269                 x1[i] = (b[i] - somma)/A[i,i]
270
271             #applico i criteri di arresto
272             r1 = b - np.dot(A, x1)
273             arresto_r1 = np.linalg.norm(r1)/np.linalg.norm(b)
274             diff_succ = np.linalg.norm(x1-x0)/np.linalg.norm(x1)
275             stop = (arresto_r1 < tol) and (diff_succ < tol)
276
277             x0 = np.copy(x1)
278             k = k+1
279
280             if not(stop):
281                 print("Attenzione: accuratezza %e non raggiunta in %d operazioni." % (tol, kmax))
282             return x1
```

# Metodo Iterativi: Jacobi e Gauss-Seidel - implementazione 2

algebra\_lineare.py x metodi\_iterativi.py x

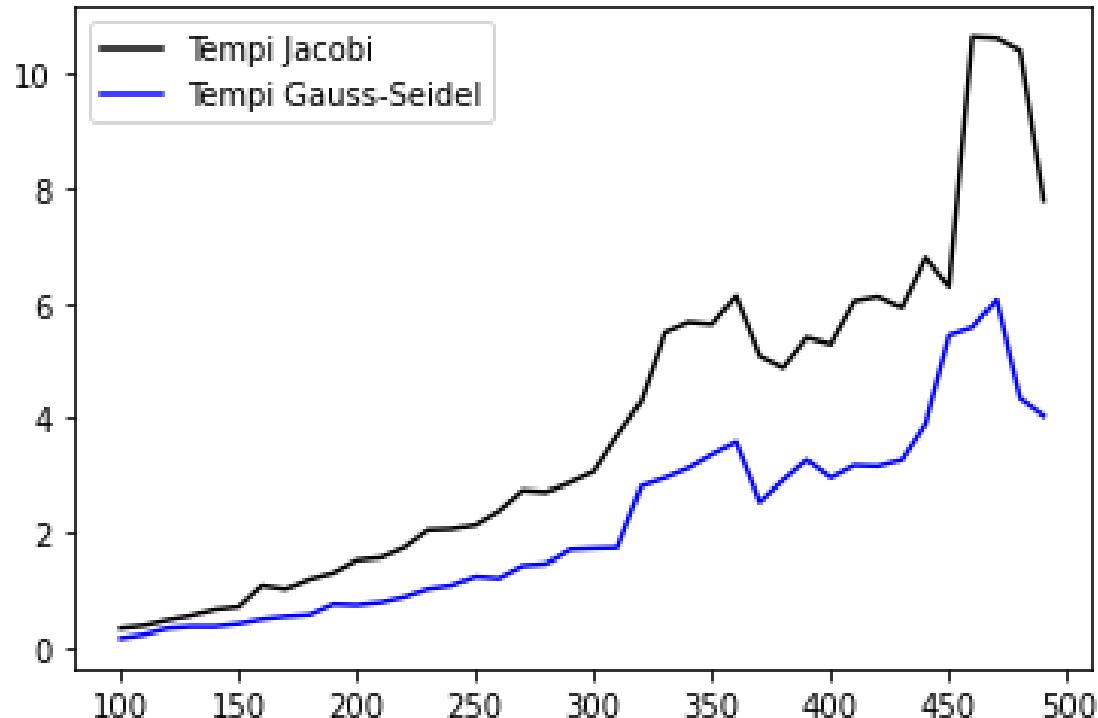
```
1 import numpy as np
2 import algebra_lineare as al
3 import time
4 import matplotlib.pyplot as plt
5
6 #dimensione della matrice dei coefficienti
7 nmax = 500
8 n_range = range(100, nmax, 10)
9 tempi_J = np.zeros(len(n_range))
10 errori_J = np.zeros(len(n_range))
11
12 tempi_GS = np.zeros(len(n_range))
13 errori_GS = np.zeros(len(n_range))
14 k = 0
15
16 for n in n_range:
17     #creo la matrice dei coefficienti
18     c = -10
19     e1 = np.ones(n-1)
20     A = np.diag(e1,-1) + c*np.eye(n) + np.diag(e1,1)
21
22     #inizializzo il vettore delle soluzioni
23     x = np.ones(n)
24     #calcolo il vettore dei termini noti
25     b = np.dot(A, x)
26
27     #parametri per i metodi iterativi
28     x0 = np.zeros(n)
29     tol = 1.0e-12
30     kmax = 500
```

algebra\_lineare.py x metodi\_iterativi.py x

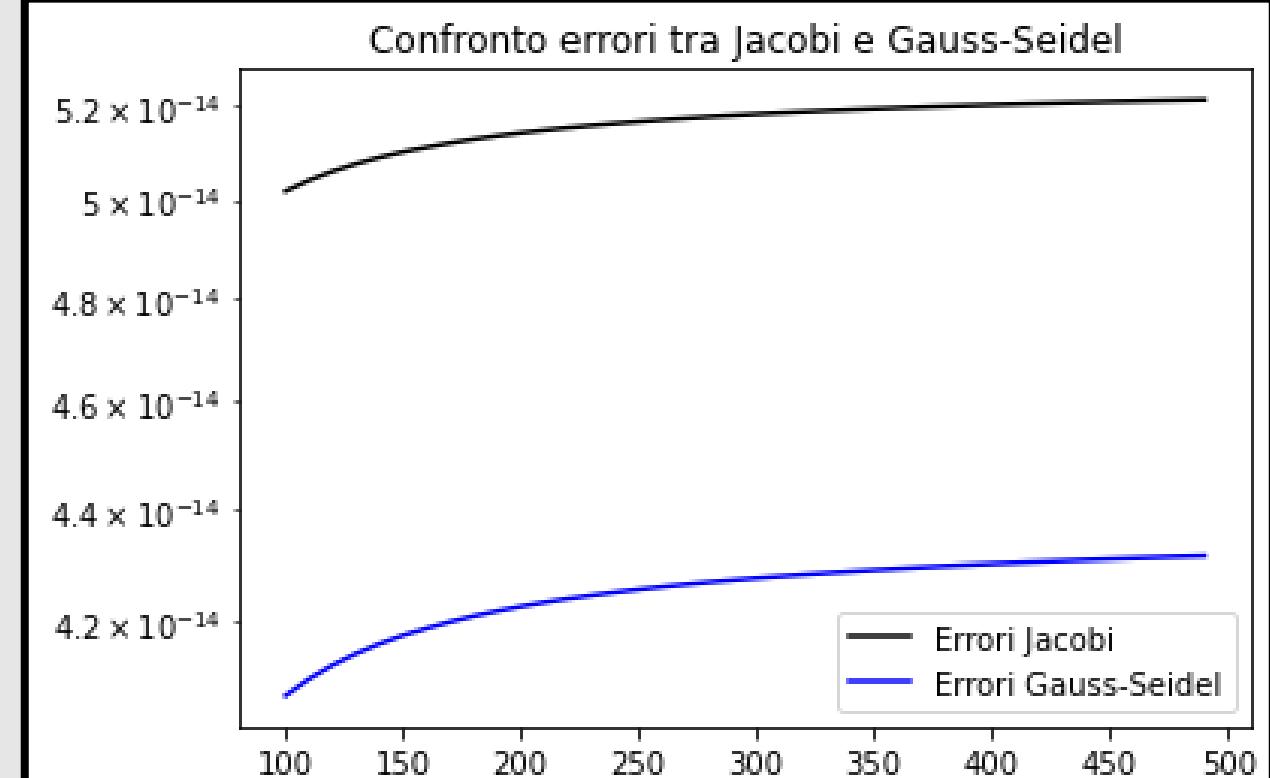
```
31
32     #calcolo soluzioni del sistema con i metodi iterativi
33     inizio = time.time()
34     x_pert_J = al.Jacobi(A, b, x0, tol, kmax)
35     fine = time.time()
36     tempi_J[k] = fine - inizio
37
38     inizio = time.time()
39     x_pert_GS = al.Gauss_Seidel(A, b, x0, tol, kmax)
40     fine = time.time()
41     tempi_GS[k] = fine - inizio
42
43     #calcolo errori
44     errori_J[k] = np.linalg.norm(x-x_pert_J)/ np.linalg.norm(x)
45     errori_GS[k] = np.linalg.norm(x-x_pert_GS)/ np.linalg.norm(x)
46     k = k + 1
47
48     #visualizzazione grafica dei risultati
49     plt.figure(1)
50     plt.title("Confronto tempi di esecuzione tra Jacobi e Gauss-Seidel")
51     plt.plot(n_range, tempi_J, "k-", label="Tempi Jacobi")
52     plt.plot(n_range, tempi_GS, "b-", label="Tempi Gauss-Seidel")
53     plt.legend()
54     plt.show()
55
56     plt.figure(2)
57     plt.title("Confronto errori tra Jacobi e Gauss-Seidel")
58     plt.semilogy(n_range, errori_J, "k-", label="Errori Jacobi")
59     plt.plot(n_range, errori_GS, "b-", label="Errori Gauss-Seidel")
60     plt.legend()
61     plt.show()
```

# Metodo Iterativi: Jacobi e Gauss-Seidel - test

Confronto tempi di esecuzione tra Jacobi e Gauss-Seidel



Confronto errori tra Jacobi e Gauss-Seidel



Dal test eseguito si nota come il metodo iterativo di Gauss-Seidel converge più velocemente alla soluzione rispetto al metodo di Jacobi ed è anche leggermente più accurato.

# Interpolazione

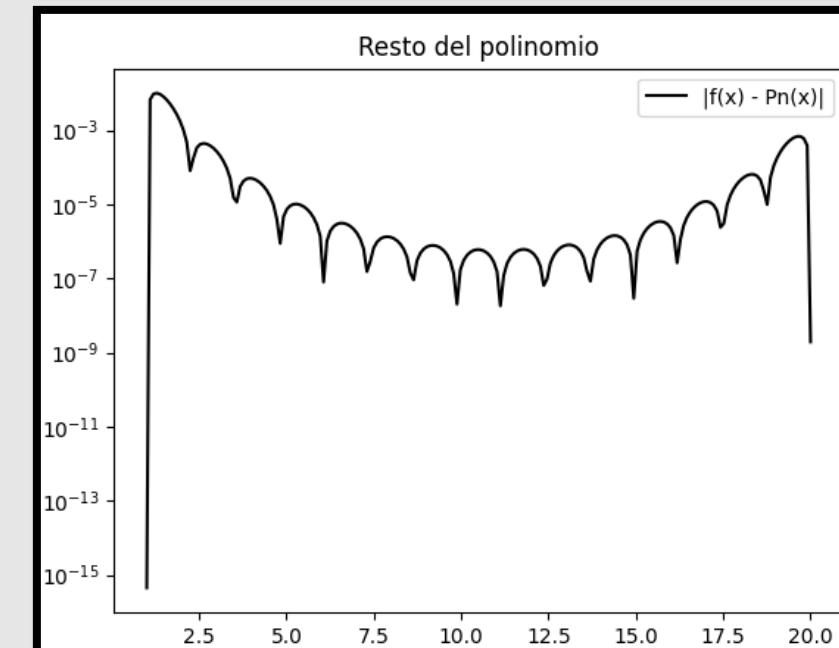
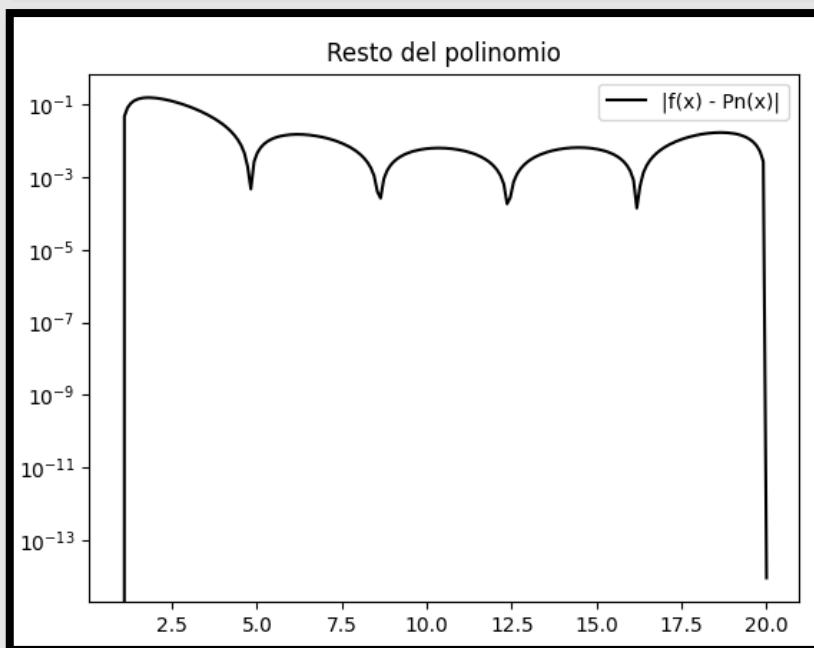
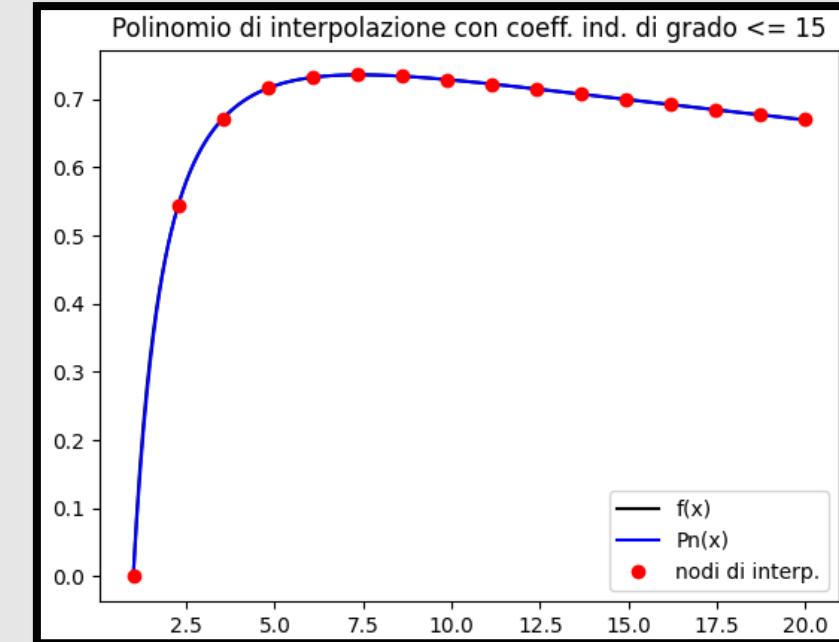
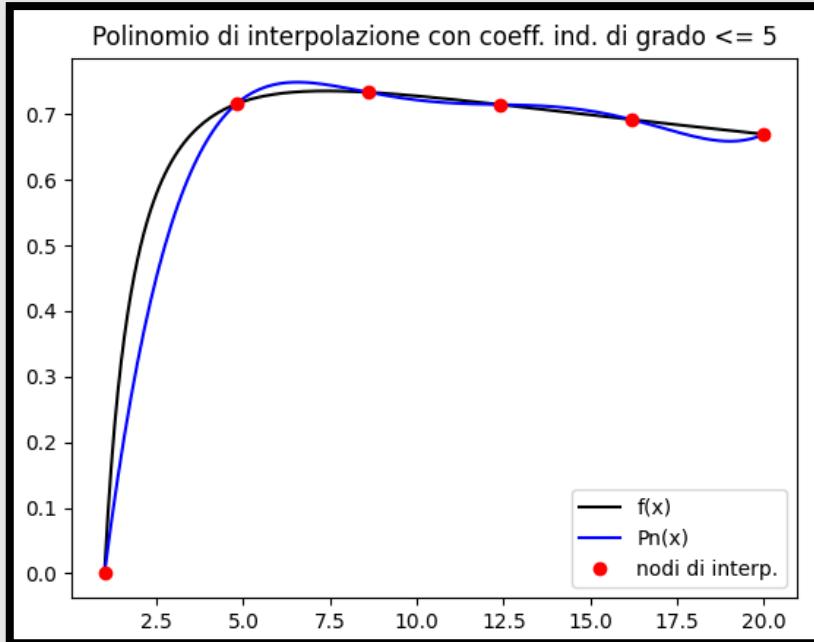
---

# Coefficienti indeterminati - implementazione

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import interpolazione.coefficienti_indeterminati as ci
4
5 def fun(x):
6     y = np.log(x)/np.sqrt(x)
7     return y
8
9 #intervallo di interpolazione
10 a = 1; b = 20
11
12 #grado del polinomio di interpolazione
13 n = 5
14
15 #nodi di interpolazione
16 xn = np.linspace(a, b, n+1)
17 #valori associati ai nodi
18 yn = fun(xn)
19
20 #punti in cui calcolare il polinomio
21 x = np.linspace(a, b, 200)
22 #calcolo la funzione nei punti generati
23 fx = fun(x)
24
25 #calcolo il polinomio di interpolazione
26 coeff = ci.coefficienti_indeterminati(xn, yn)
27 pn = np.polyval(np.flip(coeff), x)
28
29 #Visualizzazione grafica del polinomio
30 plt.figure(1)
31 plt.title("Polinomio di interpolazione con coeff. ind. di grado <= %d" % n)
32 plt.plot(x, fx, "k-", label="f(x)")
33 plt.plot(x, pn, "b-", label="Pn(x)")
34 plt.plot(xn, yn, "ro", label="nodi di interp.")
35 plt.legend()
36 plt.show()
37
38 #Visualizzazione grafica del resto del polinomio
39 plt.figure(2)
40 plt.title("Resto del polinomio")
41 plt.semilogy(x, abs(fx-pn), "k-", label="|f(x) - Pn(x)|")
42 plt.legend()
43 plt.show()
```

```
1 import numpy as np
2 import algebra_lineare as al
3
4 def Vandermonde(xn):
5     n = len(xn)
6     Vm = np.zeros((n, n))
7     for i in range(n):
8         for j in range(n):
9             Vm[i,j] = xn[i]**j
10    return Vm
11
12 def coefficienti_indeterminati(xn, yn):
13     Vm = Vandermonde(xn)
14     U, c = al.eliminazione_Gauss_pivoting_ottim(Vm, yn)
15     coeff = al.sostituzione_indietro(U, c)
16
17
```

# Coefficienti indeterminati - test



# Formula di Lagrange - implementazione

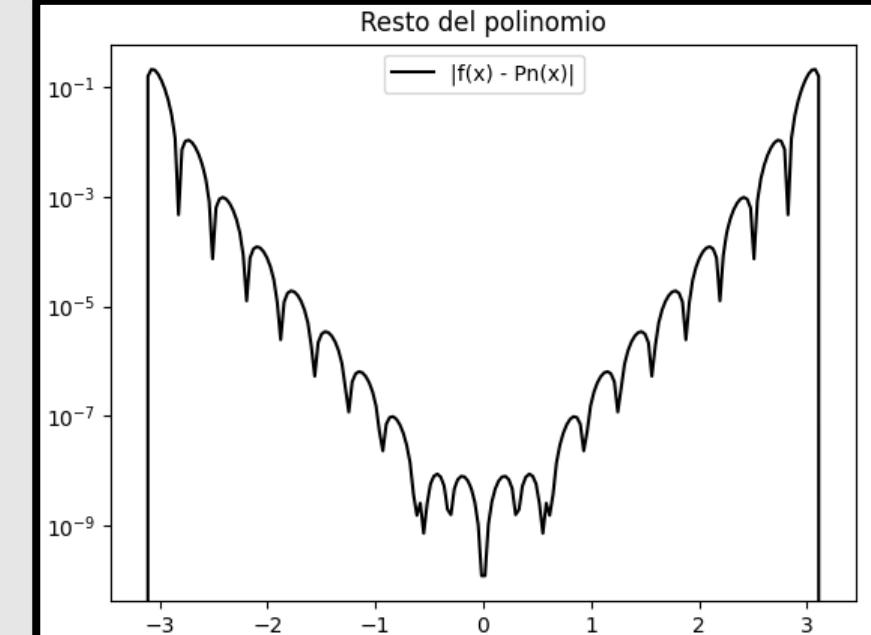
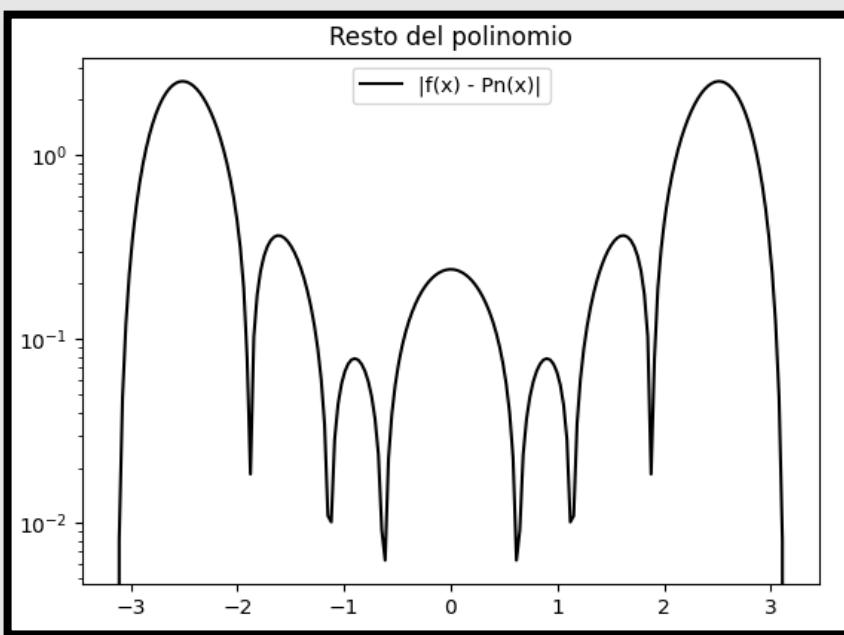
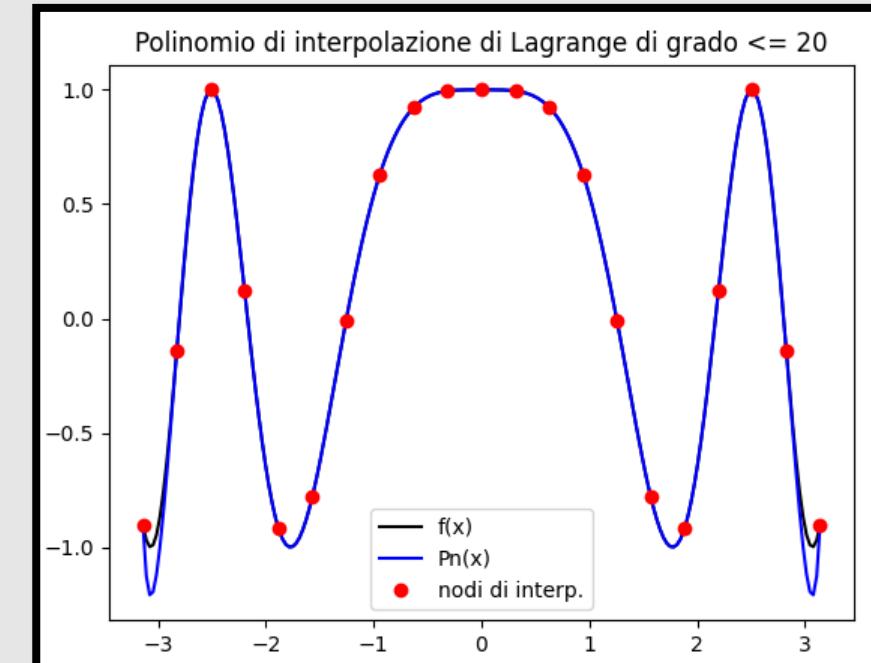
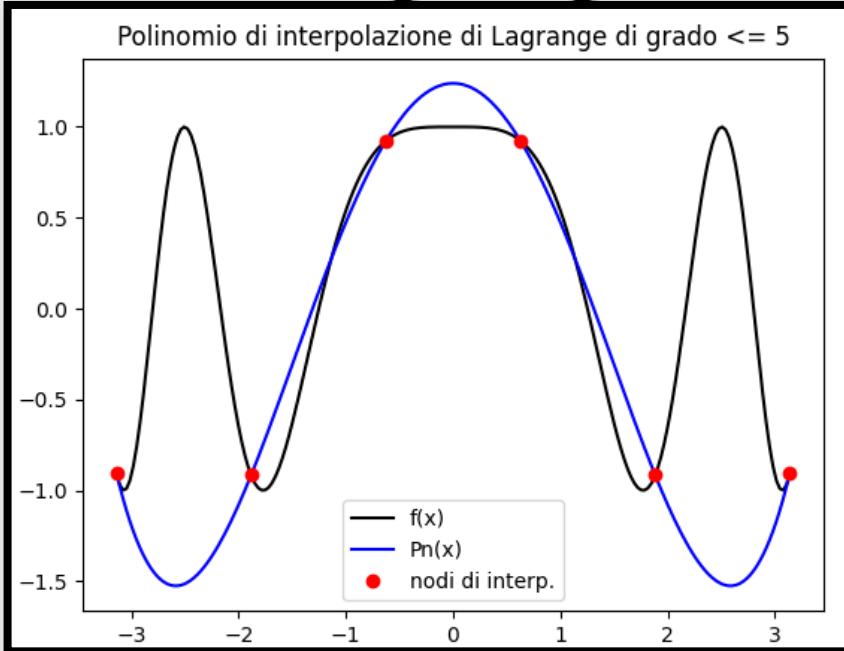
```
polinomio_lagrange.py x test_polinomio_lagrange.py x

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import interpolazione.polinomio_lagrange as pl
4
5 def fun(x):
6     y = np.cos(x**2)
7     return y
8
9 #intervallo di interpolazione
10 a = -np.pi; b = np.pi
11
12 #grado del polinomio di interpolazione
13 n = 10
14
15 #nodi di interplazione
16 xn = np.linspace(a, b, n+1)
17 #valori associati ai nodi
18 yn = fun(xn)
19
20 #punti in cui calcolare il polinomio
21 x = np.linspace(a, b, 200)
22 #calcolo la funzione nei punti generati
23 fx = fun(x)
24
25 #calcolo il polinomio di interpolazione
26 pn = pl.Lagrange(xn, yn, x)
27
28 #Visualizzazione grafica del polinomio
29 plt.figure(1)
30 plt.title("Polinomio di interpolazione di Lagrange di grado <= %d" % n)
31 plt.plot(x, fx, "k-", label="f(x)")
32 plt.plot(x, pn, "b-", label="Pn(x)")
33 plt.plot(xn, yn, "ro", label="nodi di interp.")
34 plt.legend()
35 plt.show()
36
37 #Visualizzazione grafica del resto del polinomio
38 plt.figure(2)
39 plt.title("Resto del polinomio")
40 plt.semilogy(x, abs(fx-pn), "k-", label="|f(x) - Pn(x)|")
41 plt.legend()
42 plt.show()
```

```
polinomio_lagrange.py x test_polinomio_lagrange.py x

1 import numpy as np
2
3 def Lagrange(xn, yn, x):
4     n = len(x)
5
6     pn = np.zeros(n)
7     for i in range(n):
8         pn[i] = calcolo_polinomio(xn, yn, x[i])
9     return pn
10
11 def calcolo_polinomio(xn, yn, xi):
12     n = len(xn)
13
14     Dj = D(xn, n)
15     pol = 0
16     for j in range(n):
17         Lj = N(xn, xi, j, n)/Dj[j]
18         pol = pol + yn[j]*Lj
19     return pol
20
21 #funzione che calcola la parte degli Lj(x) che varia per ogni x
22 def N(xn, xi, j, n):
23     Nj = 1
24     for k in range(n):
25         if k != j:
26             Nj = Nj * (xi - xn[k])
27     return Nj
28
29 #funzione che calcola la parte fissa degli Lj(x)
30 def D(xn, n):
31     #costruisco la matrice che memorizza le differenze
32     A = np.ones((n,n))
33     for j in range(n):
34         for k in range(n):
35             if k > j:
36                 A[j, k] = xn[j] - xn[k]
37             elif k < j:
38                 A[j, k] = -A[k, j]
39
40     #memorizzo il prodotto degli elementi di ogni riga in un vettore
41     Dj = np.ones(n)
42     for j in range(n):
43         Dj[j] = np.prod(A[j, :])
44     return Dj
```

# Formula di Lagrange - test



# Confronto dei tempi di calcolo tra le formule di Lagrange - implementazione

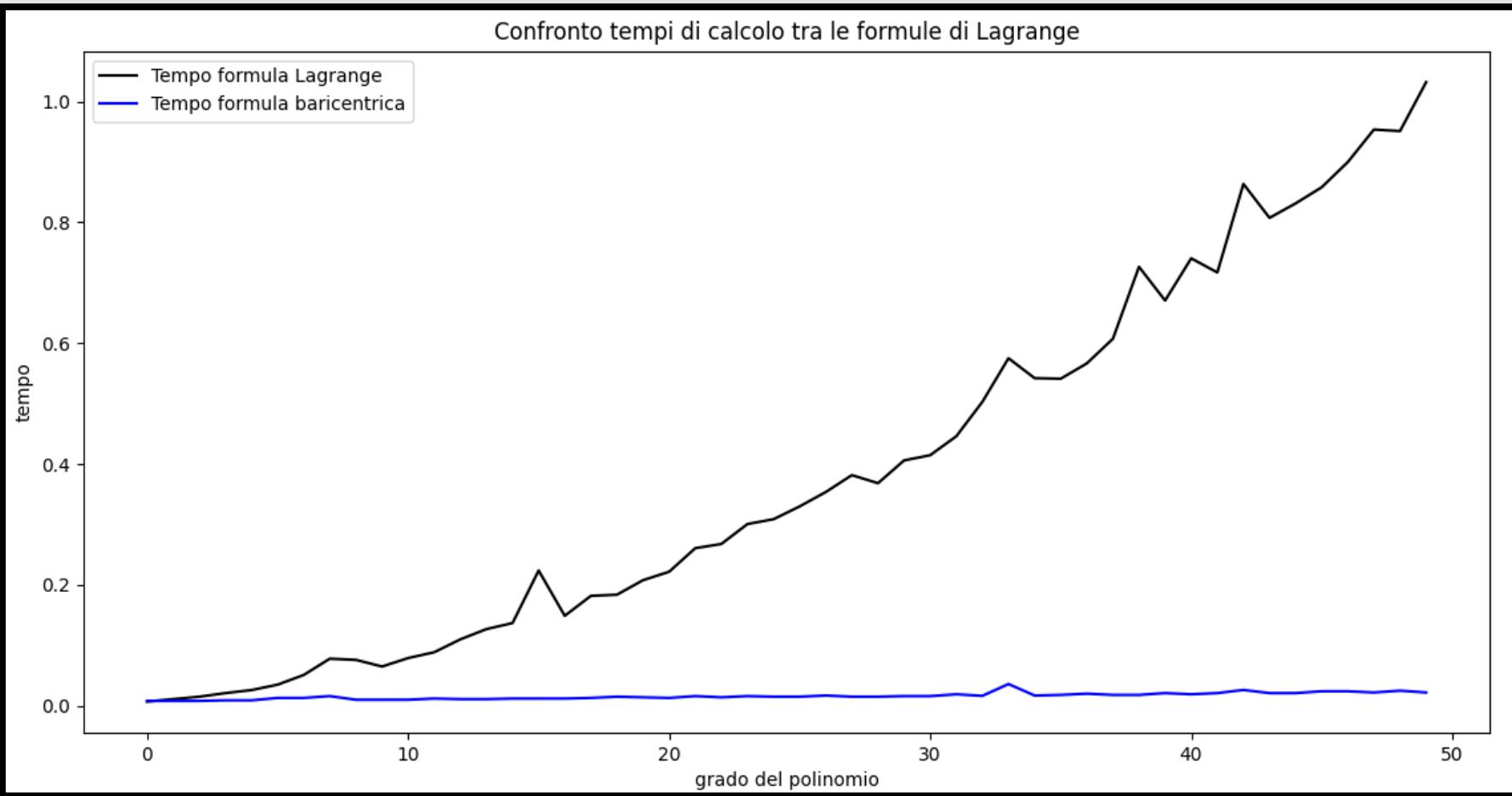
```
lagrange_baricentrica.py x confronto_formule_lagrange.py x

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import interpolazione.lagrange_baricentrica as lb
4 import interpolazione.polinomio_lagrange as pl
5 import time
6
7 def fun(x):
8     y = np.cos(x**2)
9     return y
10
11 #intervallo di interpolazione
12 a = -np.pi; b = np.pi
13
14 #punti in cui calcolare il polinomio
15 x = np.linspace(a, b, 200)
16
17 #massimo grado del polinomio di interpolazione
18 nmax = range(50)
19 tempo_lagrange = np.zeros(len(nmax))
20 tempo_baricentrica = np.zeros(len(nmax))
21 k = 0
22 for n in nmax:
23     #nodi di interplazione
24     xn = np.linspace(a, b, n+1)
25     #valori associati ai nodi
26     yn = fun(xn)
27
28     #calcolo il polinomio di Lagrange
29     inizio = time.time()
30     pn = pl.Lagrange(xn, yn, x)
31     fine = time.time()
32     tempo_lagrange[k] = fine - inizio
33
34     #calcolo il polinomio con formula baricentrica di lagrange
35     inizio = time.time()
36     pn2 = lb.Lagrange_baricentrica(xn, yn, x)
37     fine = time.time()
38     tempo_baricentrica[k] = fine - inizio
39
40     k = k + 1
41
42 #visualizzazione grafica dei risultati
43 plt.figure(1)
44 plt.title("Confronto tempi di calcolo tra le formule di Lagrange")
45 plt.plot(nmax, tempo_lagrange, "k-", label="Tempo formula Lagrange")
46 plt.plot(nmax, tempo_baricentrica, "b-", label="Tempo formula baricentrica")
47 plt.xlabel("grado del polinomio")
48 plt.ylabel("tempo")
49 plt.legend()
50 plt.show()
```

```
lagrange_baricentrica.py x confronto_formule_lagrange.py x

1 import numpy as np
2
3 def Lagrange_baricentrica(xn, yn, x):
4     n = len(x)
5     zn = coefficienti(xn, yn)
6
7     pn = np.zeros(n)
8     for i in range(n):
9         pn[i] = calcolo_polinomio(xn, zn, yn, x[i])
10    return pn
11
12 def coefficienti(xn, yn):
13     n = len(xn)
14     #inizializzo la matrice delle differenze
15     X = np.ones((n,n))
16     for j in range(n):
17         for k in range(n):
18             if k > j:
19                 X[j, k] = xn[j] - xn[k]
20             elif k < j:
21                 X[j, k] = -X[k, j]
22
23     zn = np.zeros(n)
24     for j in range(n):
25         zn[j] = yn[j]/np.prod(X[j, :])
26     return zn
27
28 #funzione che calcola il polinomio se xi non è uno dei nodi di interpolazione
29 #se invece restituisce il valore associato al nodo
30 def calcolo_polinomio(xn, zn, yn, xi):
31     check_nodi = abs(xi-xn) < 1.0e-14
32     if True in check_nodi:
33         #estratto tutti gli indici degli elementi che in check_nodi sono uguali a True
34         indice = np.flatnonzero(check_nodi == True)
35         p = yn[indice[0]]
36     else:
37         n = len(xn)
38         somma = 0
39         for j in range(n):
40             somma = somma + (zn[j]/(xi - xn[j]))
41         p = np.prod(xi-xn)*somma
42
43     return p
```

# Confronto dei tempi di calcolo tra le formule di Lagrange - test



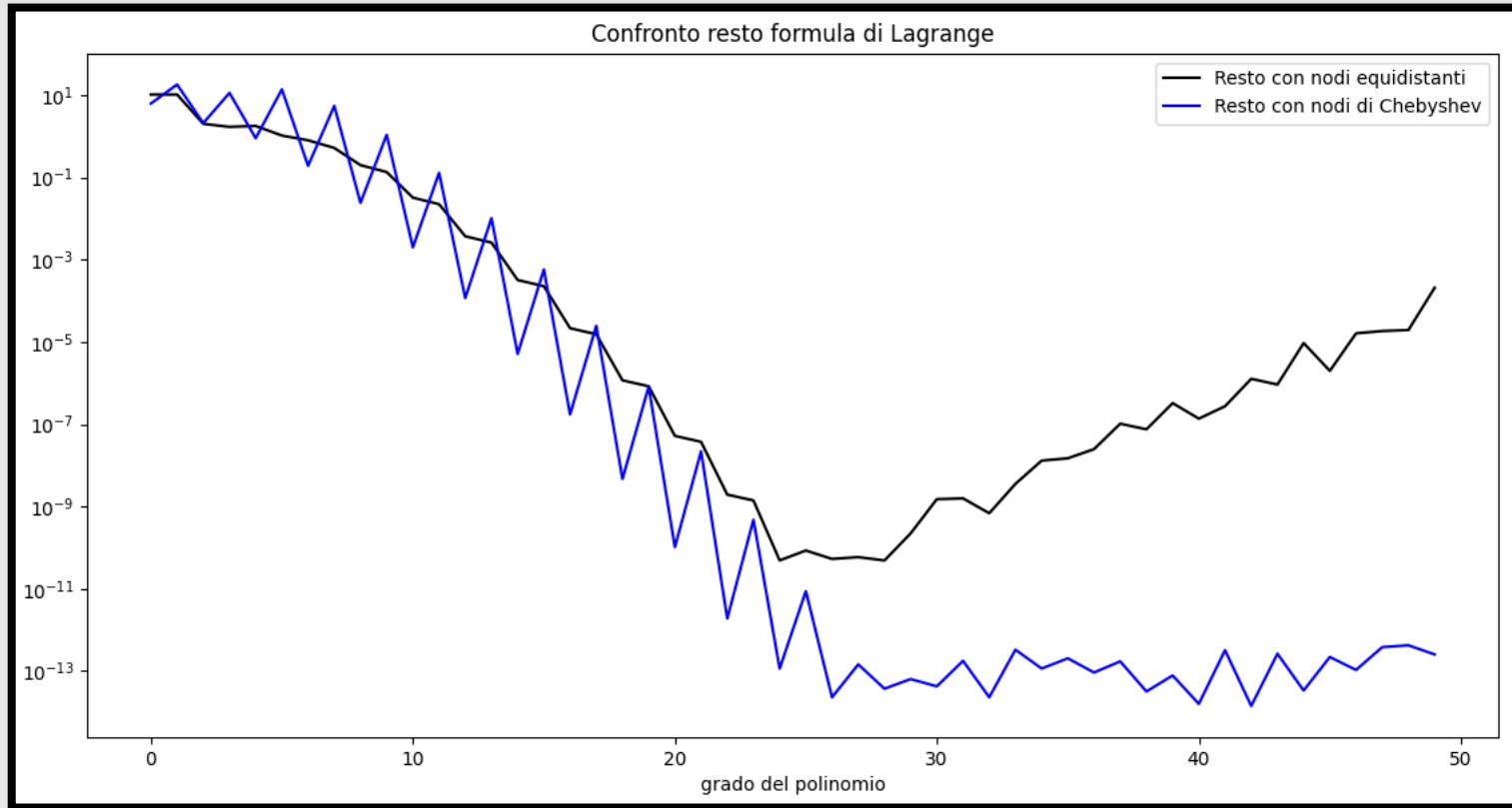
Si noti come al crescere del grado del polinomio l'utilizzo della prima formula baricentrica di Lagrange comporti notevoli vantaggi in termini di tempo rispetto alla formula normale.

# Stabilità del polinomio di Lagrange - implementazione

```
1 import numpy as np
2 import interpolazione.lagrange_baricentrica as lb
3 import matplotlib.pyplot as plt
4
5 #routine che calcola i nodi di Chebyshev
6 def Chebyshev(a, b, n):
7     xn = np.zeros(n)
8     for i in range(n):
9         cos = np.cos(((2*i + 1)/(2*n + 2))*np.pi)
10        xn[i] = a + (cos + 1)*((b-a)/2)
11    return xn
12
13 def fun(x):
14     y = x**2 + np.cos(2*x)
15     return y
16
17 #intervallo di interpolazione
18 a = -np.pi; b = np.pi
19
20 #punti in cui calcolare il polinomio
21 x = np.linspace(a, b, 400)
22 #calcolo la funzione nei punti generati
23 fx = fun(x)
24
25 #massimo grado del polinomio di interpolazione
26 nmax = range(50)
27 resto_equi = np.zeros(len(nmax))
28 resto_cheby = np.zeros(len(nmax))
29 k = 0
```

```
30 for n in nmax:
31     # ===== nodi equidistanti =====
32     xn_equi = np.linspace(a, b, n+1)
33     #valori associati ai nodi
34     yn_equi = fun(xn_equi)
35
36     #calcolo il polinomio di Lagrange su nodi equidistanti
37     pn_equi = lb.Lagrange_baricentrica(xn_equi, yn_equi, x)
38     resto_equi[k] = max(abs(fx - pn_equi))
39     # =====
40
41     # ===== nodi di Chebyshev =====
42     xn_cheby = Chebyshev(a, b, n+1)
43     #valori associati ai nodi
44     yn_cheby = fun(xn_cheby)
45
46     #calcolo il polinomio di Lagrange su nodi equidistanti
47     pn_cheby = lb.Lagrange_baricentrica(xn_cheby, yn_cheby, x)
48     resto_cheby[k] = max(abs(fx - pn_cheby))
49     # =====
50
51     k = k + 1
52
53 plt.figure(1)
54 plt.title("Confronto resto formula di Lagrange")
55 plt.semilogy(nmax, resto_equi, "k-", label="Resto con nodi equidistanti")
56 plt.semilogy(nmax, resto_cheby, "b-", label="Resto con nodi di Chebyshev")
57 plt.xlabel("grado del polinomio")
58 plt.legend()
59 plt.show()
```

# Stabilità del polinomio di Lagrange - test



Dal seguente test si possono osservare due fenomeni:

- Se si calcolano polinomi di grado elevato, il resto tende a decrescere fino ad un certo punto e poi a crescere, questo a causa degli errori introdotti dal calcolatore che aumentano all'aumentare delle operazioni effettuate;
- L'utilizzo dei nodi di Chebyshev permette di avere un migliore condizionamento del problema, infatti all'aumentare del grado del polinomio gli errori vengono propagati di meno rispetto all'uso di nodi equidistanti;

# Polinomio di Newton - implementazione

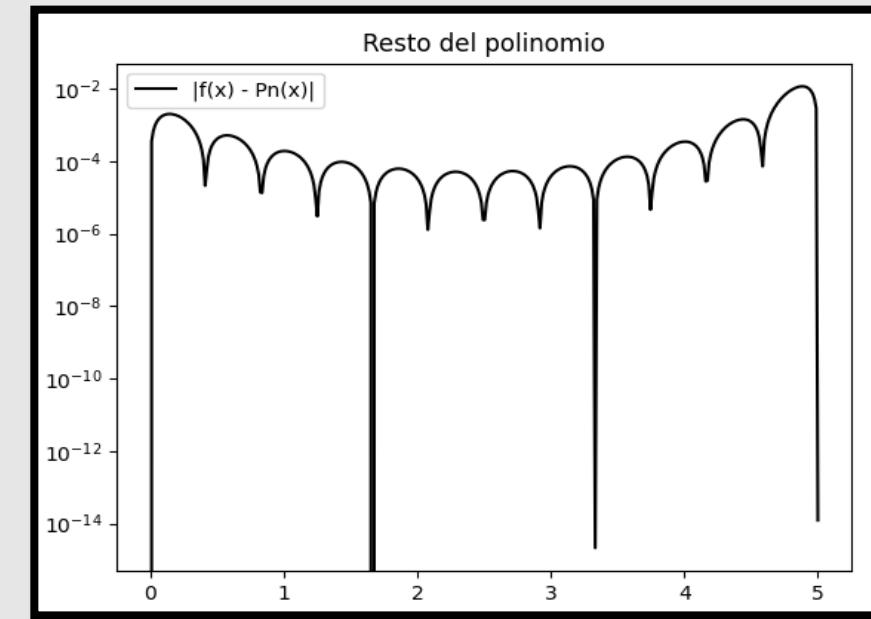
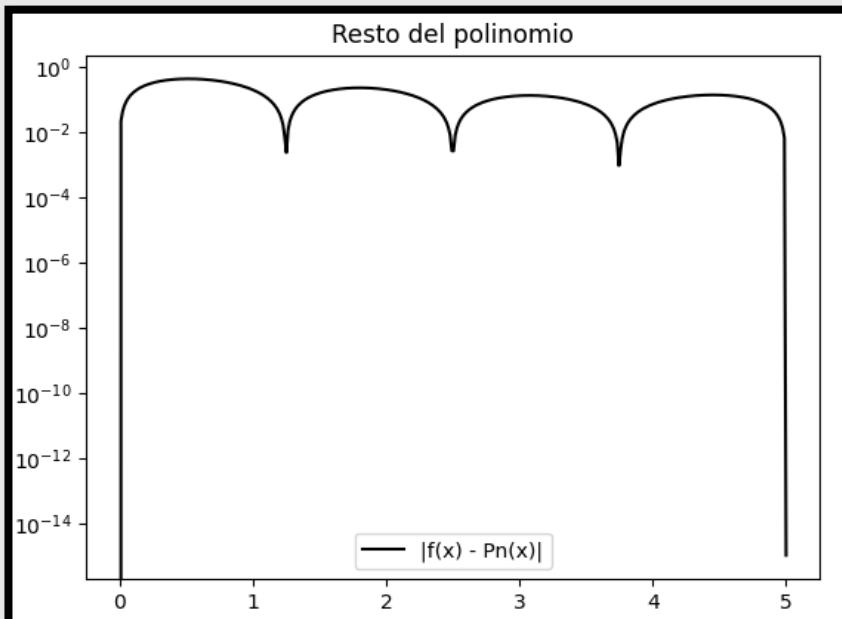
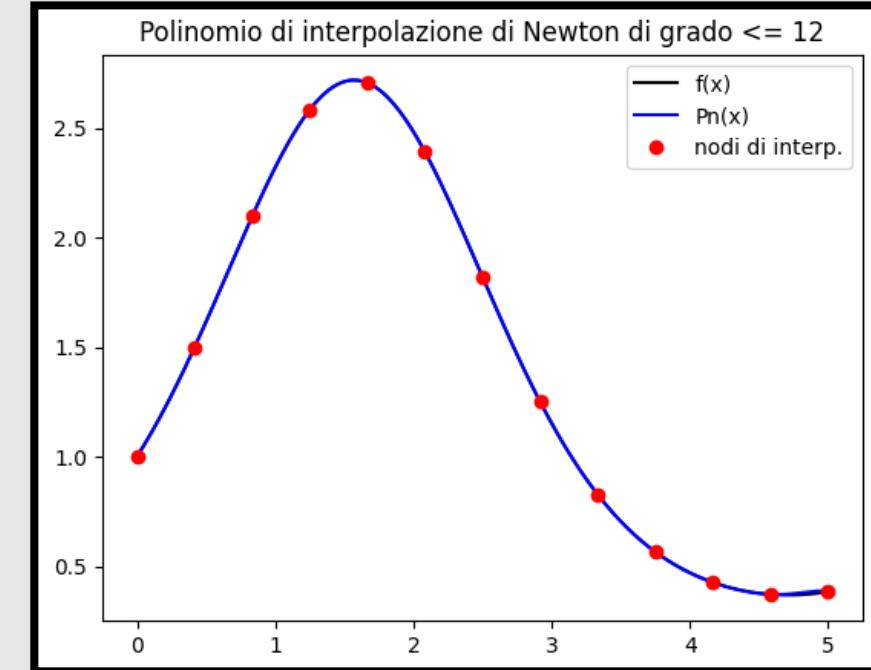
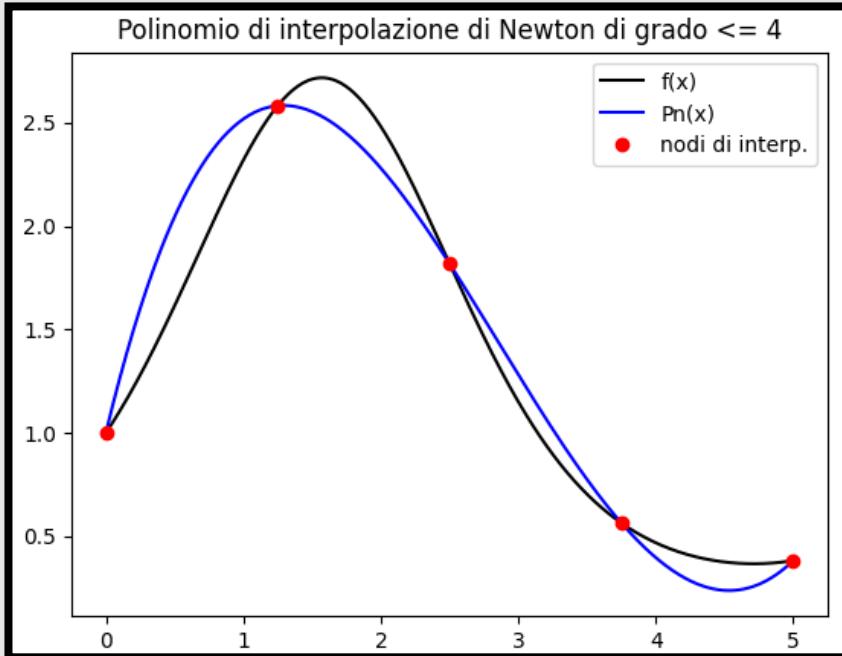
```
polinomio_newton.py × test_polinomio_newton.py ×

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import interpolazione.polinomio_newton as pn
4
5 def fun(x):
6     y = np.exp(np.sin(x))
7     return y
8
9 #intervallo di interpolazione
10 a = 0; b = 5
11
12 #grado del polinomio di interpolazione
13 n = 4
14
15 #nodi di interplazione
16 xn = np.linspace(a, b, n+1)
17 #valori associati ai nodi
18 yn = fun(xn)
19
20 #punti in cui calcolare il polinomio
21 x = np.linspace(a, b, 400)
22 #calcolo la funzione nei punti generati
23 fx = fun(x)
24
25 #calcolo il polinomio di interpolazione
26 pn = pn.Newton(xn, yn, x)
27
28 #Visualizzazione grafica del polinomio
29 plt.figure(1)
30 plt.title("Polinomio di interpolazione di Newton di grado <= %d" % n)
31 plt.plot(x, fx, "k-", label="f(x)")
32 plt.plot(x, pn, "b-", label="Pn(x)")
33 plt.plot(xn, yn, "ro", label="nodi di interp.")
34 plt.legend()
35 plt.show()
36
37 #Visualizzazione grafica del resto del polinomio
38 plt.figure(2)
39 plt.title("Resto del polinomio")
40 plt.semilogy(x, abs(fx-pn), "k-", label="|f(x) - Pn(x)|")
41 plt.legend()
42 plt.show()
```

```
polinomio_newton.py × test_polinomio_newton.py ×

1 import numpy as np
2
3 def Newton(xn, yn, x):
4     dn = differenze_divise(xn, yn)
5
6     n = len(x)
7     pn = np.zeros(n)
8     for i in range(n):
9         pn[i] = calcolo_polinomio(dn, xn, x[i])
10    return pn
11
12 #calcolo il polinomio nel punto xi
13 def calcolo_polinomio(dn, xn, xi):
14     n = len(xn)
15
16     p = dn[n-1]
17     for j in range(n-2, -1, -1):
18         p = dn[j] + p*(xi - xn[j])
19     return p
20
21 #calcolo differenze divise con la tecnica del vettore
22 def differenze_divise(xn, yn):
23     n = len(xn)
24     dn = np.copy(yn)
25     for j in range(1, n):
26         for i in range(n-1, j-1, -1):
27             dn[i] = (dn[i] - dn[i-1])/(xn[i] - xn[i-j])
28
29
```

# Polinomio di Newton - test



# Ricerca delle radici di funzione

---

# Bisezioni successive - implementazione

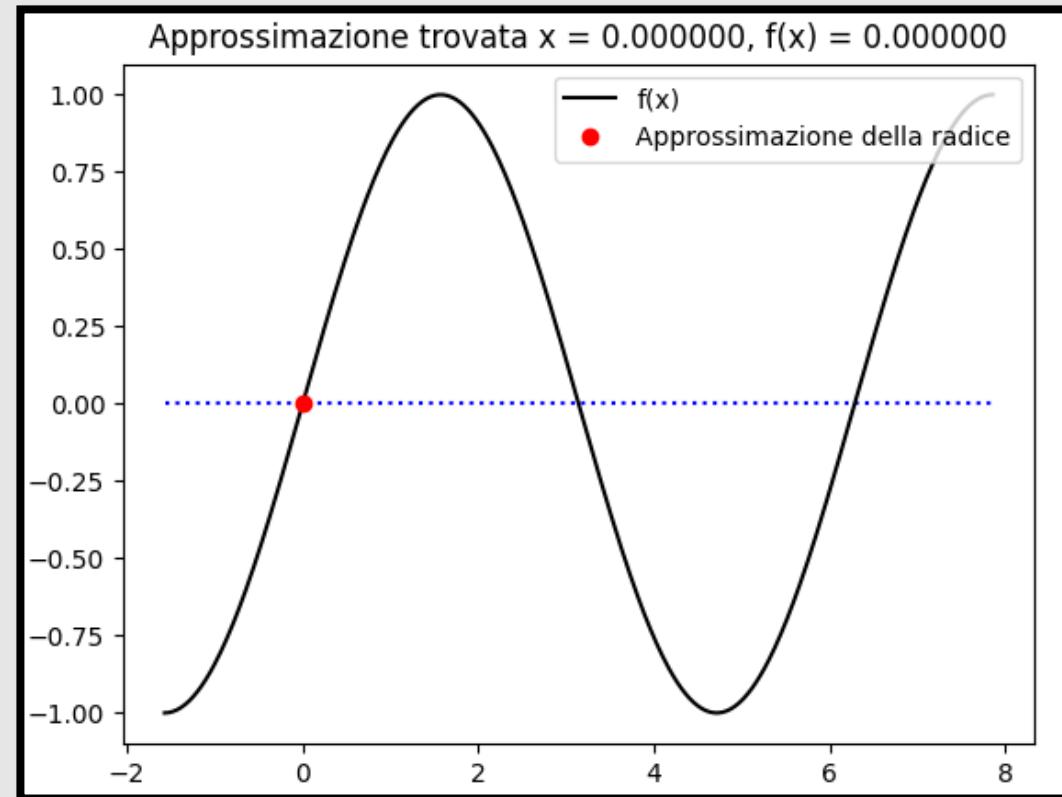
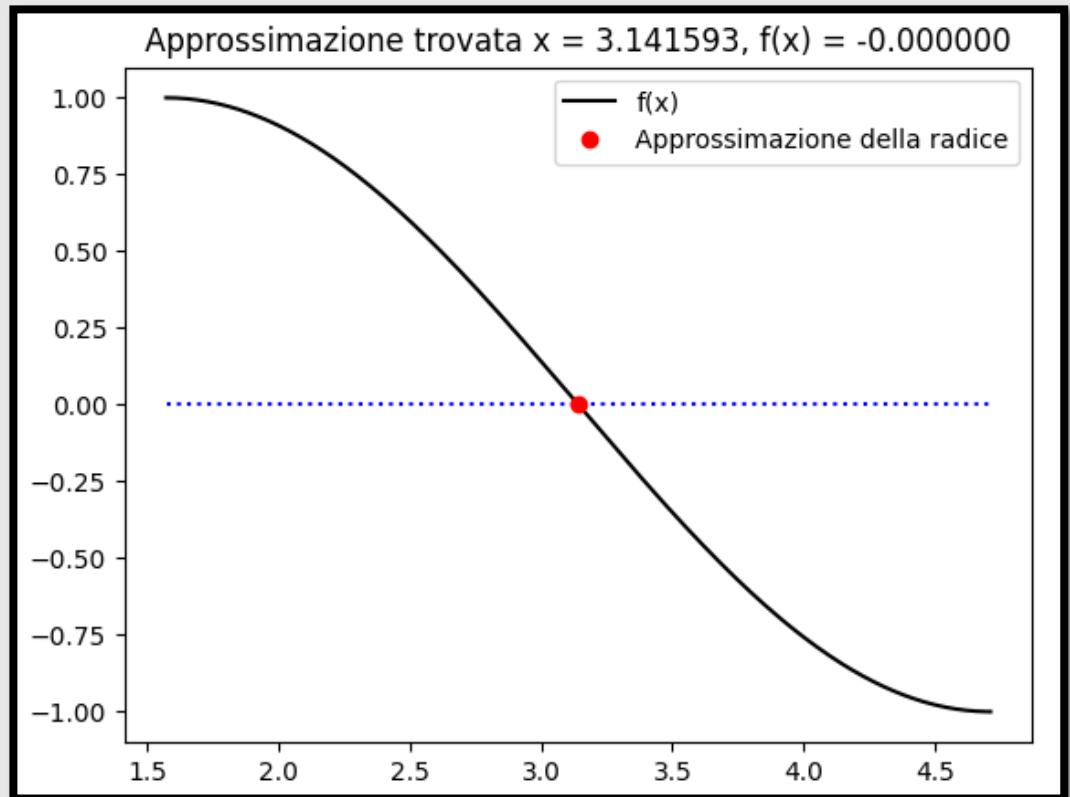
```
ricerca_radici.py x test_bisezioni_successive.py x

1 import ricerca_radici as rr
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 def fun(x):
6     y = np.sin(x)
7     return y
8
9 #intervallo in cui cercare la radice
10 a = np.pi/2; b = (3*np.pi)/2
11 tol = 1.0e-14
12
13 #cerco la radice
14 radice = rr.bisezioni_successive(fun, a, b, tol)
15 f_radice = fun(radice)
16
17 #calcolo la funzione
18 x = np.linspace(a, b, 400)
19 fx = fun(x)
20
21 plt.figure(1)
22 plt.title("Approssimazione trovata x = %f, f(x) = %f" % (radice, f_radice))
23 zero = np.zeros(len(x))
24 plt.plot(x, zero, "b:")
25 plt.plot(x, fx, "k-", label="f(x)")
26 plt.plot(radice, f_radice, "ro", label = "Approssimazione della radice")
27 plt.legend()
28 plt.show()
```

```
ricerca_radici.py x test_bisezioni_successive.py x

1 from math import ceil, log2
2 import sys
3
4 def bisezioni_successive(fun, a, b, tol):
5     fa = fun(a)
6     fb = fun(b)
7     if fa*fb > 0:
8         print("Errore: la presenza della radice non è garantita in [%f, %f]" % (a,b))
9         sys.exit(-1)
10    else:
11        n = ceil(log2(b-a) - log2(tol)) - 1
12        for i in range(n+1):
13            c = (a+b)/2
14            fc = fun(c)
15            if fa*fc <= 0:
16                b = c
17            else:
18                a = c
19                fa = fc
20
21
22
23
24
25
26
27
28
```

# Bisezioni successive - test



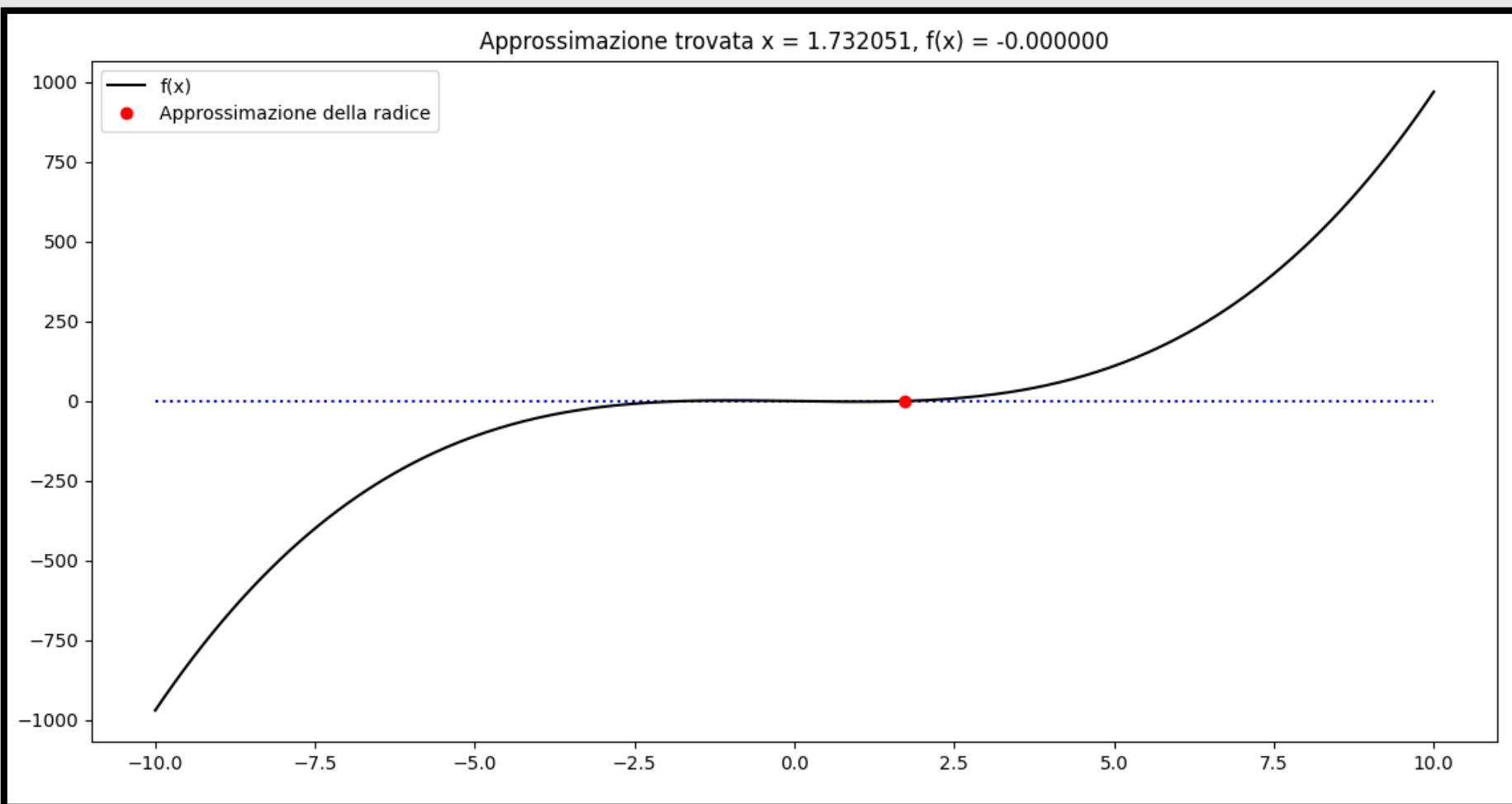
Questo test è stato eseguito nell'intervallo  $[-\pi/2, 5/2\pi]$  in cui sono presenti più radici, ma il metodo ne trova solo una.

# Metodo delle tangenti - implementazione

```
ricerca_radici.py x test_metodo_tangenti.py x
1 import ricerca_radici as rr
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 def fun(x):
6     y = x**3 - 3*x
7     return y
8
9 #derivata della funzione fun
10 def der(x):
11     y = 3*x**2 - 3
12     return y
13
14 #parametri del metodo delle tangenti
15 tol = 1.0e-14
16 x0 = 2
17 kmax = 200
18
19 #cerco la radice
20 radice = rr.metodo_tangenti(x0, fun, der, tol, kmax)
21 f_radice = fun(radice)
22
23 #costruisco il grafico delle funzione
24 #l'intervallo è stato preso in modo da contenere abbondantemente
25 #l'approssimazione iniziale x0
26 a = -10; b = 10
27 x = np.linspace(a, b, 400)
28 fx = fun(x)
29
30 plt.figure(1)
31 plt.title("Approssimazione trovata x = %f, f(x) = %f" % (radice, f_radice))
32 zero = np.zeros(len(x))
33 plt.plot(x, zero, "b:")
34 plt.plot(x, fx, "k-", label="f(x)")
35 plt.plot(radice, f_radice, "ro", label = "Approssimazione della radice")
36 plt.legend()
37 plt.show()
```

```
ricerca_radici.py x test_metodo_tangenti.py x
22 def metodo_tangenti(x0, fun, der, tol, kmax):
23     stop = False
24     k = 0
25
26     while not(stop) and k < kmax:
27         #calcolo gli elementi da applicare nella formula
28         y0 = fun(x0)
29         d0 = der(x0)
30
31         #calcolo la nuova approssimazione
32         x1 = x0 - (y0/d0)
33
34         #verifico se si è raggiunta l'accuratezza tramite i criteri di arresto
35         stop = (abs(fun(x1)) < tol) and (abs(x1-x0)/(1 + abs(x1)) < tol)
36         if not(stop):
37             x0 = x1
38             k +=1
39
40     if not(stop):
41         print("Il metodo non converge in %d iterazioni" % kmax)
42     return x1
```

# Metodo delle tangenti - test



# Confronto metodi delle tangenti, secanti e corde - implementazione

```
1  import ricerca_radici as rr
2
3  def fun(x):
4      y = x**3 - 3*x
5      return y
6
7  #derivata della funzione fun
8  def der(x):
9      y = 3*x**2 - 3
10     return y
11
12 #parametri dei metodi per la ricerca della radice
13 tol = 1.0e-14
14 x0 = 2
15 #seconda approssimazione per il metodo delle secanti
16 x1 = 3
17 #coefficiente angolare per il metodo delle corde
18 m = 6
19 kmax = 200
20
21 #ricerca con il metodo delle tangenti
22 r_tan, k_tan = rr.metodo_tangenti(x0, fun, der, tol, kmax)
23
24 #ricerca con il metodo delle corde
25 r_cor, k_cor = rr.metodo_corde(x0, fun, m, tol, kmax)
26
27 #ricerca con il metodo delle tangenti
28 r_sec, k_sec = rr.metodo_secanti(x0, x1, fun, tol, kmax)
29
30 #visualizzazione dei risultati
31 print("Risultati ottenuti da ciascun metodo:")
32 print("- Metodo delle tangenti:")
33 print("\tApprossimazione trovata: x = %.15f => f(x) = %.15f" % (r_tan, fun(r_tan)))
34 print("\tNumero di iterazioni effettuate: %d" % k_tan)
35
36 print("\n- Metodo delle secanti:")
37 print("\tApprossimazione trovata: x = %.15f => f(x) = %.15f" % (r_sec, fun(r_sec)))
38 print("\tNumero di iterazioni effettuate: %d" % k_sec)
39
40 print("\n- Metodo delle corde (con m = %d):" % m)
41 print("\tApprossimazione trovata: x = %.15f => f(x) = %.15f" % (r_cor, fun(r_cor)))
42 print("\tNumero di iterazioni effettuate: %d" % k_cor)
```

I seguenti metodi restituiscono anche il valore di k esclusivamente per questo programma.

```
44  def metodo_secanti(x0, x1, fun, tol, kmax):
45      stop = False
46      k = 0
47
48      while not(stop) and k < kmax:
49          #calcolo gli elementi da applicare nella formula
50          y1 = fun(x1)
51          s1 = (fun(x1)-fun(x0))/(x1-x0) #rapporto incrementale
52
53          #calcolo la nuova approssimazione
54          x2 = x1 - (y1/s1)
55
56          #verifico se si è raggiunta l'accuratezza tramite i criteri di arresto
57          stop = abs(fun(x2)) + (abs(x2-x1)/(1+abs(x2))) < tol/5
58          if not(stop):
59              x0 = x1
60              x1 = x2
61              k +=1
62
63          if not(stop):
64              print("Il metodo non converge in %d iterazioni" % kmax)
65      return x2, k
```

```
67  def metodo_corde(x0, fun, m, tol, kmax):
68      stop = False
69      k = 0
70
71      while not(stop) and k <= kmax:
72          f0 = fun(x0)
73          x1 = x0 - (f0/m)
74
75          stop = abs(fun(x1)) + (abs(x1-x0)/(1+abs(x1))) < tol
76
77          x0 = x1
78          k = k+1
79
80      return x1, k
```

# Confronto metodi delle tangenti, secanti e corde - test

Risultati ottenuti da ciascun metodo:

- Metodo delle tangenti:  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 6
- Metodo delle secanti:  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 9
- Metodo delle corde (con  $m = 10$ ):  
Approssimazione trovata:  $x = 1.732050807568878 \Rightarrow f(x) = 0.0000000000000005$   
Numero di iterazioni effettuate: 36

Risultati ottenuti da ciascun metodo:

- Metodo delle tangenti:  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 6
- Metodo delle secanti:  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 9
- Metodo delle corde (con  $m = 2$ ):  
Approssimazione trovata:  $x = 1.000000000000000 \Rightarrow f(x) = -2.000000000000000$   
Numero di iterazioni effettuate: 201

Risultati ottenuti da ciascun metodo:

- Metodo delle tangenti:  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 6
- Metodo delle secanti:  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 9
- Metodo delle corde (con  $m = 6$ ):  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 6

Risultati ottenuti da ciascun metodo:

- Metodo delle tangenti:  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 6
- Metodo delle secanti:  
Approssimazione trovata:  $x = 1.732050807568877 \Rightarrow f(x) = -0.0000000000000001$   
Numero di iterazioni effettuate: 9
- Metodo delle corde (con  $m = 15$ ):  
Approssimazione trovata:  $x = 1.732050807568879 \Rightarrow f(x) = 0.0000000000000007$   
Numero di iterazioni effettuate: 64

Dai test si evince che il metodo delle tangenti ha, in generale, una convergenza più veloce, anche se scegliendo opportunamente il valore di  $m$ , il metodo delle corde può raggiungere le stesse prestazioni.

# Formule di quadratura

---

# Formula del trapezio - implementazione

The screenshot shows a code editor with two tabs open: "formule\_quadratura.py" and "test\_formula\_trapezio.py". The "formule\_quadratura.py" tab is active, displaying the following code:

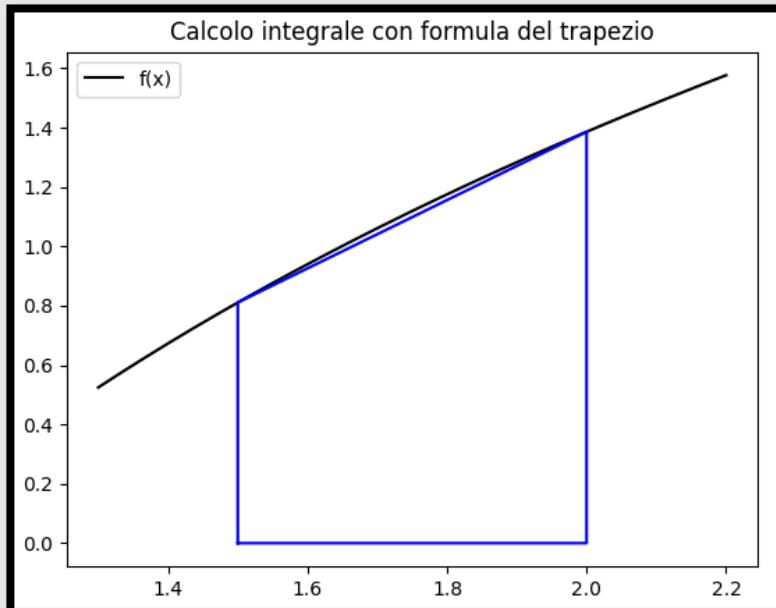
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import formule_quadratura as fq
4
5 #funzione di cui calcolare l'integrale
6 def f(x):
7     y = 2*np.log(x)
8     return y
9
10 #primitiva di fun
11 def F(x):
12     y = 2*x*np.log(x) - 2*x
13     return y
14
15 # intervallo di integrazione
16 a = 1.5; b = 2
17
18 #calcolo l'integrale tramite la primitiva
19 I = F(b) - F(a)
20
21 #calcolo l'integrale con la formula del trapezio
22 T = fq.formula_trapezio(a, b, f)
23
24 #calcolo errore
25 E = abs(I-T)/abs(I)
26
27 #visualizzazione risultati
28 print("Integrale esatto: %f" % I)
29 print("Formula del trapezio : %f" % T)
30 print("Errore commesso: %e" % E)
31
32 #rappresentazione grafica
33 x = np.linspace(a-0.2, b+0.2, 400)
34 fx = f(x)
35
36 plt.figure(1)
37 plt.title("Calcolo integrale con formula del trapezio")
38 plt.plot(x, fx, "k-", label="f(x)")
39 #costruzione trapezio
40 xx = np.array([a, a, b, b, a])
41 yy = np.array([0, f(a), f(b), 0, 0])
42 plt.plot(xx, yy, "b-")
43 plt.legend()
44 plt.show()
```

The screenshot shows the "formule\_quadratura.py" tab in the code editor, displaying the definition of the `formula_trapezio` function:

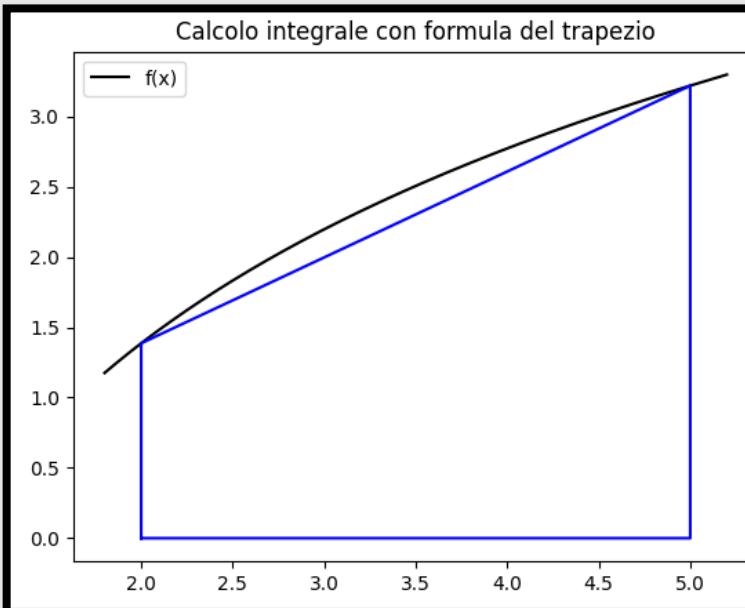
```
1 def formula_trapezio(a, b, fun):
2     T = (b-a)/2
3     T = T*(fun(a) + fun(b))
4     return T
```

# Formula del trapezio - test

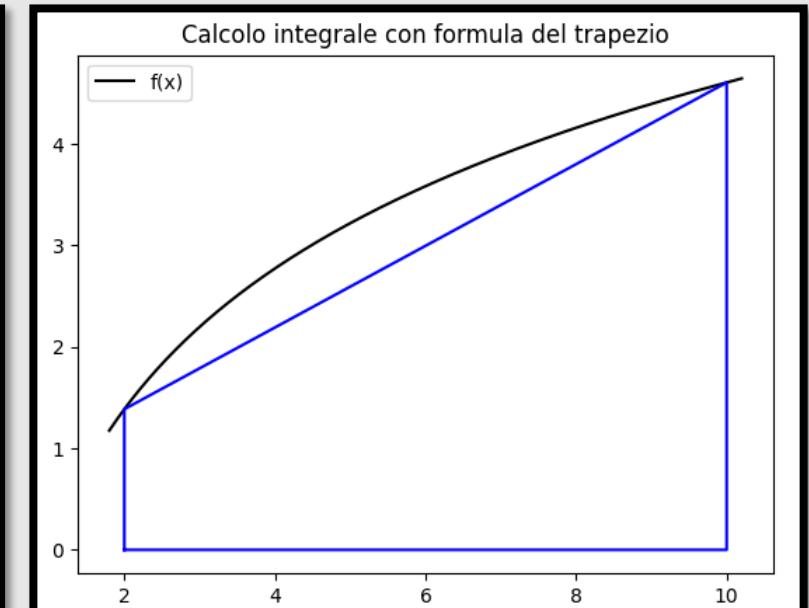
Test su intervallo [1.5, 2]:



Test su intervallo [2, 5]:



Test su intervallo [2, 10]:



Programmi esame )

Integrale esatto: 0.556193

Formula del trapezio : 0.549306

Errore commesso: 1.238284e-02

Integrale esatto: 7.321790

Formula del trapezio : 6.907755

Errore commesso: 5.654834e-02

Integrale esatto: 27.279113

Formula del trapezio : 23.965858

Errore commesso: 1.214576e-01

Si noti come all'aumentare dell'intervallo di integrazione aumenta anche l'errore sull'approssimazione dell'integrale.

# Formula composta del trapezio - implementazione

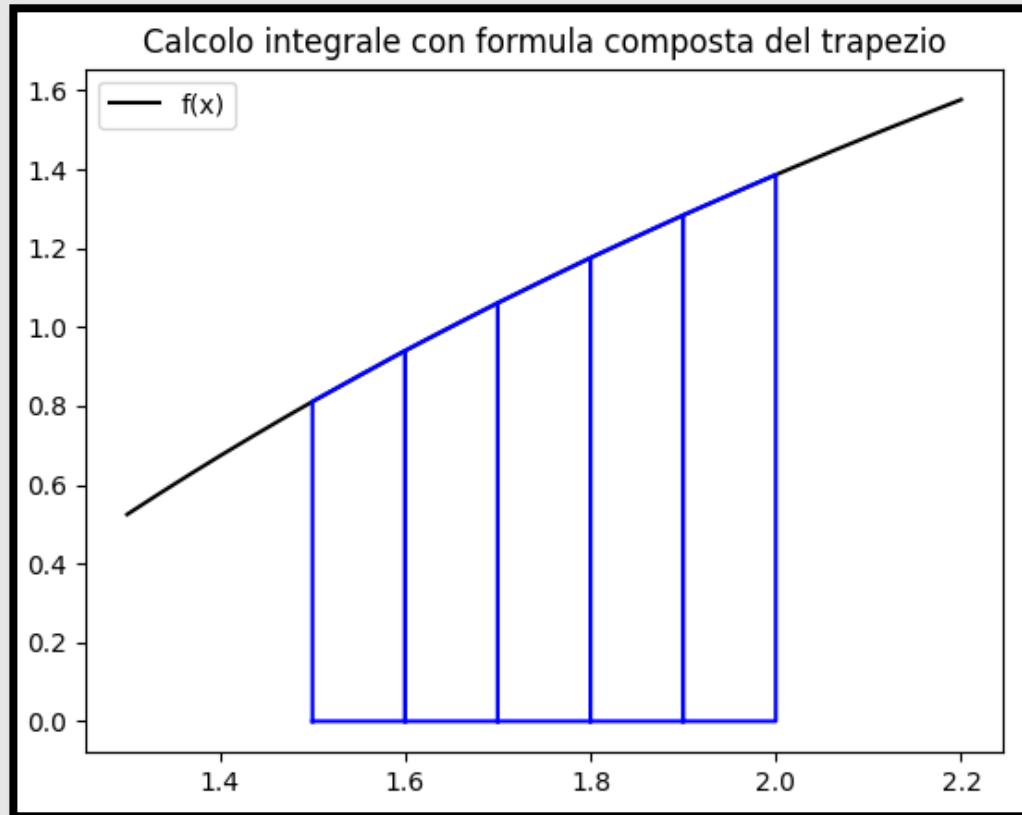
```
formule_quadratura.py x test_formula_composta_trapezio.py x

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import formule_quadratura as fq
4
5 #funzione di cui calcolare l'integrale
6 def f(x):
7     y = 2*np.log(x)
8     return y
9
10 #primitiva di fun
11 def F(x):
12     y = 2*x*np.log(x) - 2*x
13     return y
14
15 # intervallo di integrazione
16 a = 1.5; b = 2
17
18 #calcolo l'integrale tramite la primitiva
19 I = F(b) - F(a)
20
21 #calcolo l'integrale con la formula del trapezio
22 N = 40
23 TN = fq.formula_composta_trapezio(a, b, N, f)
24
25 #calcolo errore
26 E = abs(I-TN)/abs(I)
27
28 #visualizzazione risultati
29 print("Integrale esatto: %f" % I)
30 print("Formula del trapezio con N = %d : %f" % (N,TN))
31 print("Errore commesso: %e" % E)
32
33 #rappresentazione grafica
34 x = np.linspace(a-0.2, b+0.2, 400)
35 fx = f(x)
36
37 plt.figure(1)
38 plt.title("Calcolo integrale con formula composta del trapezio")
39 plt.plot(x, fx, "k-", label="f(x)")
40 #costruzione degli N trapezi
41 x = np.linspace(a, b, N+1)
42 for i in range(N):
43     xx = np.array([x[i], x[i], x[i+1], x[i+1], x[i]])
44     yy = np.array([0, f(x[i]), f(x[i+1]), 0, 0])
45     plt.plot(xx, yy, "b-")
46 plt.legend()
47 plt.show()
```

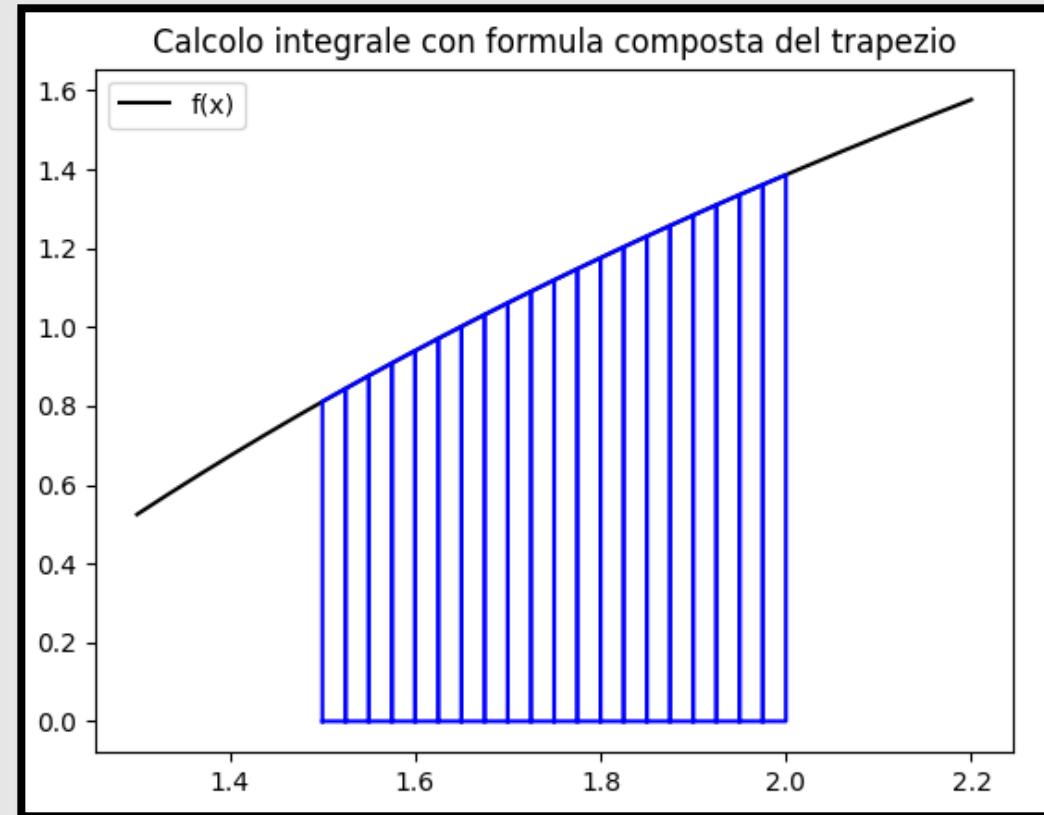
```
formule_quadratura.py x test_formula_composta_trapezio.py x

8 def formula_composta_trapezio(a, b, N, fun):
9     #genero N+1 nodi equidistanti
10    x = np.linspace(a, b, N+1)
11    fx = fun(x)
12
13    somma = 0
14    for i in range(1, N):
15        somma = somma + fx[i]
16    TN = (b-a)/(2*N)
17    TN = TN*(fx[0] + 2*somma + fx[N])
18    return TN
```

# Formula composta del trapezio- test

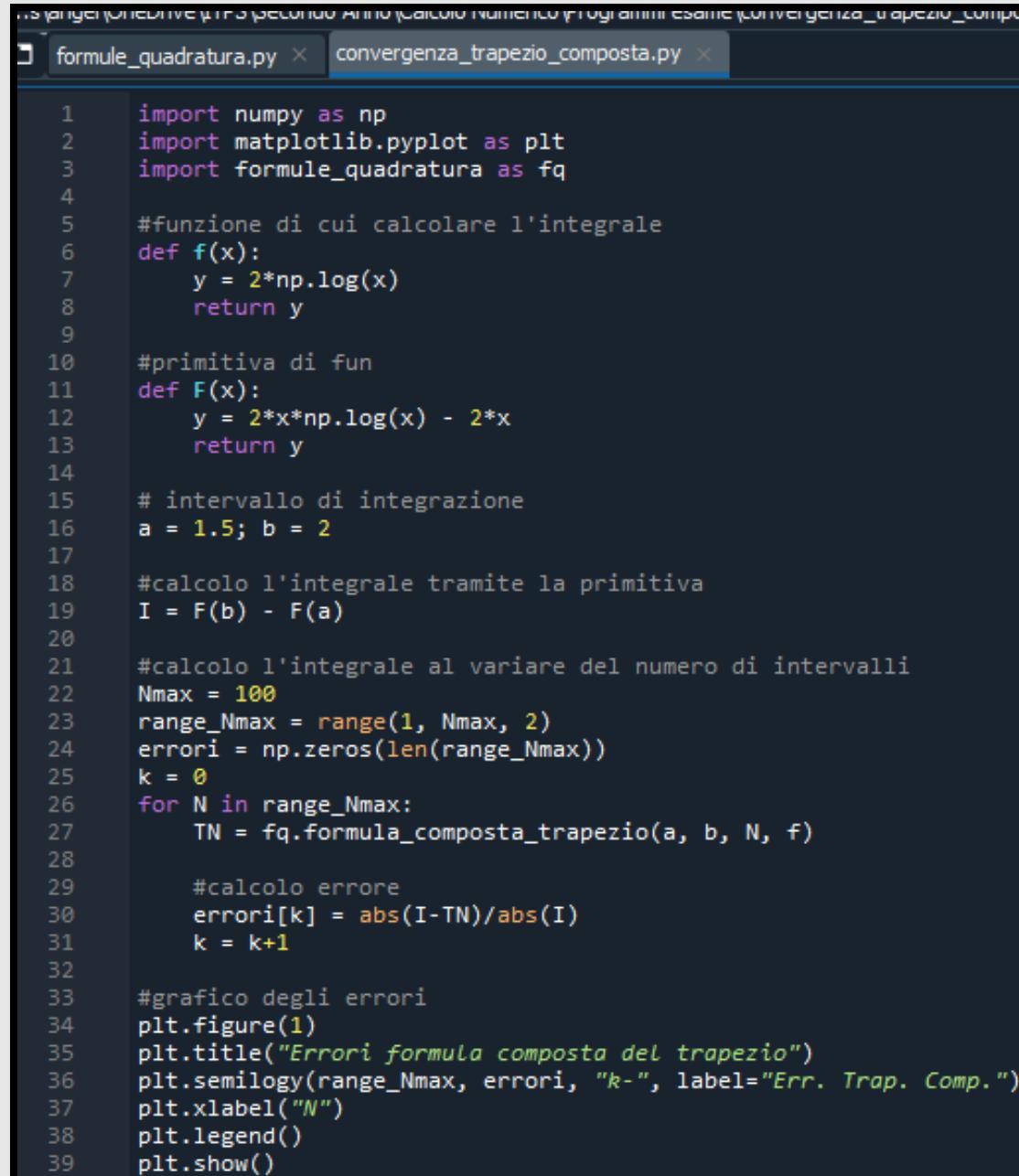


Integrale esatto: 0.556193  
Formula del trapezio con  $N = 5$  : 0.555916  
Errore commesso: 4.992558e-04



Integrale esatto: 0.556193  
Formula del trapezio con  $N = 20$  : 0.556176  
Errore commesso: 3.121349e-05

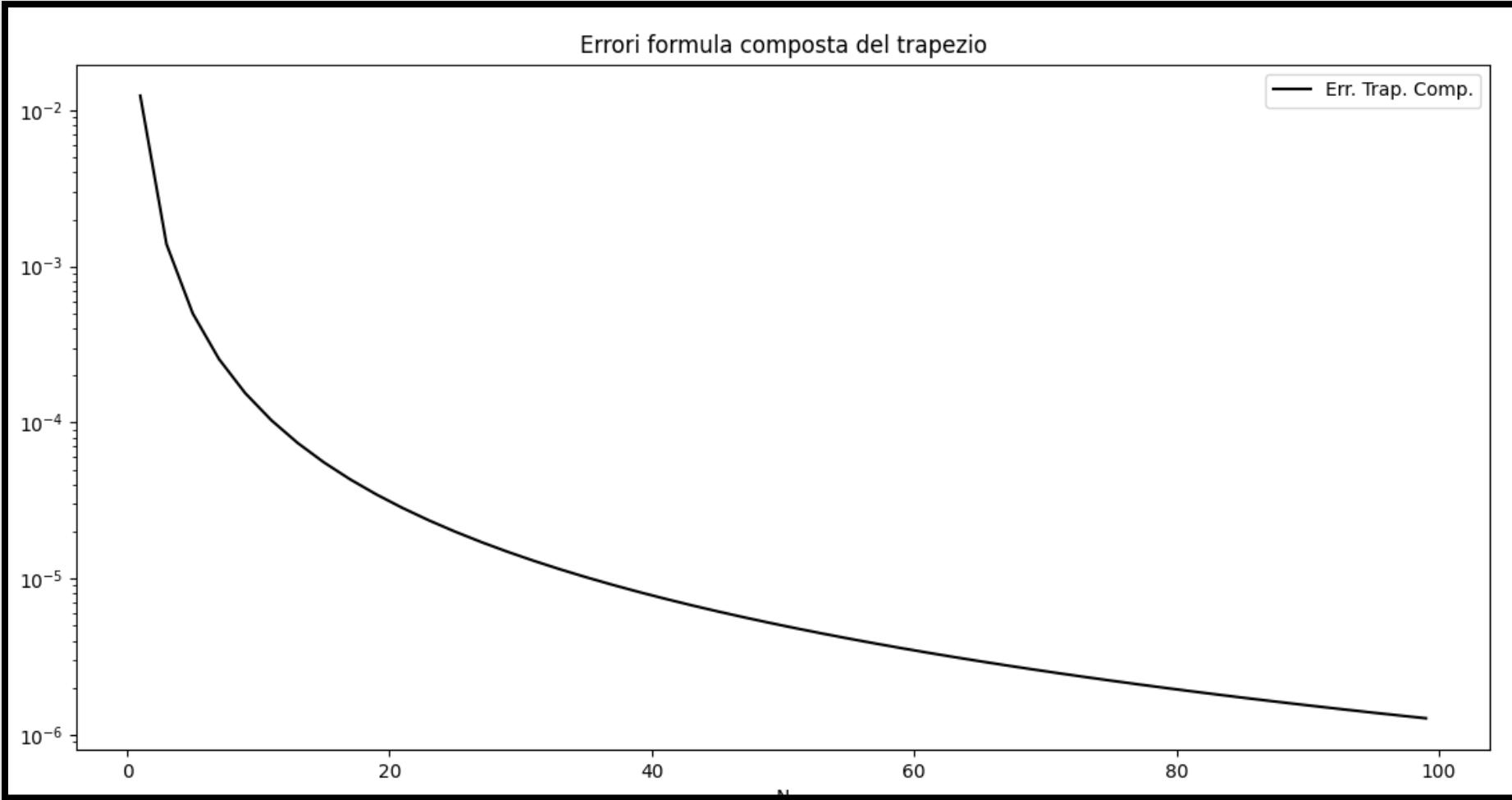
# Convergenza formula composta del trapezio - implementazione



The screenshot shows a code editor window with two tabs: "formule\_quadratura.py" and "convergenza\_trapezio\_composta.py". The "convergenza\_trapezio\_composta.py" tab is active, displaying the following Python code:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import formule_quadratura as fq
4
5 # funzione di cui calcolare l'integrale
6 def f(x):
7     y = 2*np.log(x)
8     return y
9
10 # primitiva di fun
11 def F(x):
12     y = 2*x*np.log(x) - 2*x
13     return y
14
15 # intervallo di integrazione
16 a = 1.5; b = 2
17
18 # calcolo l'integrale tramite la primitiva
19 I = F(b) - F(a)
20
21 # calcolo l'integrale al variare del numero di intervalli
22 Nmax = 100
23 range_Nmax = range(1, Nmax, 2)
24 errori = np.zeros(len(range_Nmax))
25 k = 0
26 for N in range_Nmax:
27     TN = fq.formula_composta_trapezio(a, b, N, f)
28
29     # calcolo errore
30     errori[k] = abs(I-TN)/abs(I)
31     k = k+1
32
33 # grafico degli errori
34 plt.figure(1)
35 plt.title("Errori formula composta del trapezio")
36 plt.semilogy(range_Nmax, errori, "k-", label="Err. Trap. Comp.")
37 plt.xlabel("N")
38 plt.legend()
39 plt.show()
```

# Convergenza formula composta del trapezio - test



Il test effettuato evidenzia che l'errore commesso dalla formula composta del trapezio nell'approssimare l'integrale, tende a 0 all'aumentare del numero di intervalli considerati.

# Confronto formula del Trapezio e di Simpson- implementazione

```
formule_quadratura.py x confronto_simpson_trapezio.py x

2     import formule_quadratura as fq
3     import matplotlib.pyplot as plt
4
5     #funzione da integrare
6     def f(x):
7         y = x**3 - 2*x
8         return y
9
10    #primitiva di f
11    def F(x):
12        y = (x**4)/4 - x**2
13        return y
14
15    #intervallo di integrazione
16    a = -1; b = -0.5
17
18    #calcolo l'integrale tramite la primitiva
19    I = F(b) - F(a)
20
21    #calcolo integrale tramite la formula del trapezio
22    T = fq.formula_trapezio(a, b, f)
23
24    #calcolo integrale tramite la formula di Simpson
25    S = fq.formula_Simpson(a, b, f)
26
27    #calcolo errori
28    Err_T = abs(I-T)/abs(I)
29    Err_S = abs(I-S)/abs(I)
30
31    #visualizzazione risultati
32    print("Confronto tra formula del trapezio e di Simpson:")
33    print("Integrale esatto: %f" % I)
34    print("Formula del trapezio:")
35    print("\t - Approssimazione : %f" % T)
36    print("\t - Errore : %e" % Err_T)
37    print("Formula di Simpson:")
38    print("\t - Approssimazione : %f" % S)
39    print("\t - Errore : %e" % Err_S)
40
```

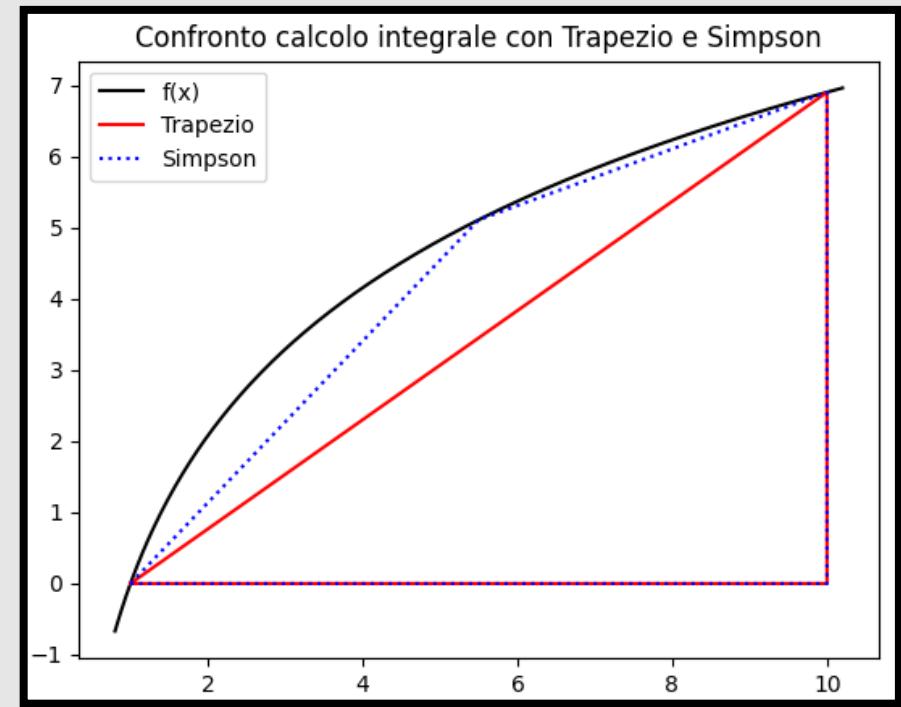
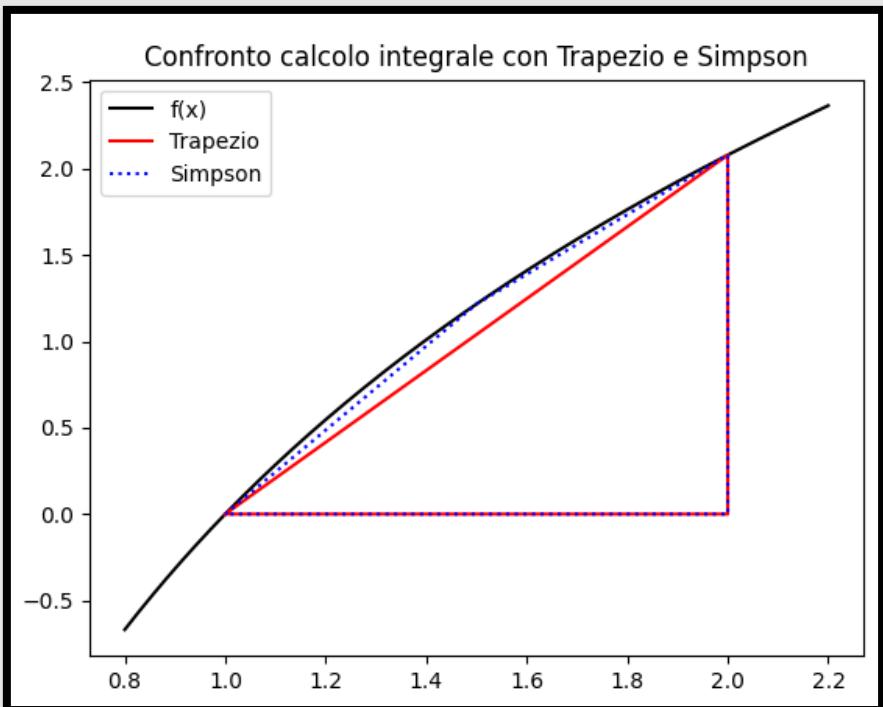
```
41     #rappresentazione grafica
42     x = np.linspace(a-0.2, b+0.2, 400)
43     fx = f(x)
44
45     plt.figure(1)
46     plt.title("Confronto calcolo integrale con Trapezio e Simpson")
47     plt.plot(x, fx, "k-", label="f(x)")
48     #costruzione del trapezio
49     xx_T = np.array([a, a, b, b, a])
50     yy_T = np.array([0, f(a), f(b), 0, 0])
51
52     #costruzione approssimazione di Simpson
53     c = (a+b)/2
54     xx_S = np.array([a, a, c, b, b, c, a])
55     yy_S = np.array([0, f(a), f(c), f(b), 0, 0, 0])
56     plt.plot(xx_T, yy_T, "r-", label="Trapezio")
57     plt.plot(xx_S, yy_S, "b:", label="Simpson")
58     plt.legend()
59     plt.show()
```

Formula di Simpson utilizzata:

```
formule_quadratura.py x confronto_simpson_trapezio.py x

19
20     def formula_Simpson(a, b, fun):
21         #punto medio dell'intervallo
22         c = (a+b)/2
23
24         S = (b-a)/6
25         S = S * (fun(a) + 4*fun(c) + fun(b))
26         return S
```

# Confronto formula del Trapezio e di Simpson- test



Confronto tra formula del trapezio e di Simpson:  
Integrale esatto: 1.158883  
Formula del trapezio:  
- Approssimazione : 1.039721  
- Errore : 1.028251e-01  
Formula di Simpson:  
- Approssimazione : 1.157504  
- Errore : 1.190178e-03

Confronto tra formula del trapezio e di Simpson:  
Integrale esatto: 42.077553  
Formula del trapezio:  
- Approssimazione : 31.084899  
- Errore : 2.612475e-01  
Formula di Simpson:  
- Approssimazione : 41.047099  
- Errore : 2.448940e-02

Dai test si può notare la maggiore precisione della formula di Simpson nel fornire un'approssimazione dell'integrale.

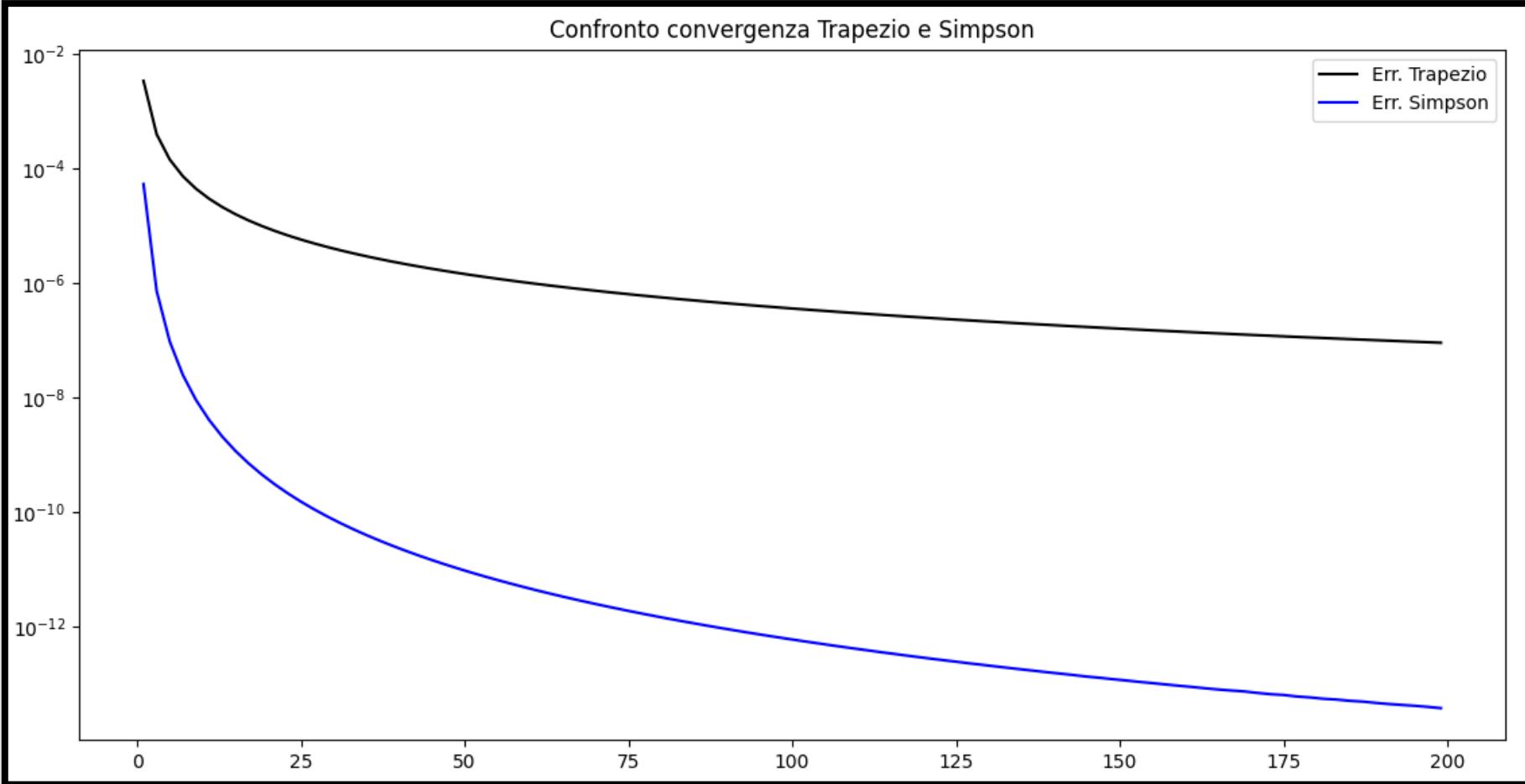
# Convergenza formula composta del trapezio e di Simpson - implementazione

```
| formule_quadratura.py x convergenza_simpson_trapezio.py x
1 import formule_quadratura as fq
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 #funzione da integrare
6 def f(x):
7     y = 2-np.exp(x)
8     return y
9
10 #primitiva di f
11 def F(x):
12     y = 2*x - np.exp(x)
13     return y
14
15 #intervallo di integrazione
16 a = -5; b = -3
17
18 #calcolo integrale tramite primitiva
19 I = F(b) - F(a)
20
21 #calcolo l'integrale usando le formule composte del trapezio
22 # e di Simpson all'aumentare degli intervalli
23 Nmax = 200
24 range_Nmax = range(1, Nmax, 2)
25 errori_T = np.zeros(len(range_Nmax))
26 errori_S = np.zeros(len(range_Nmax))
27 k = 0
28 for N in range_Nmax:
29     #calcolo integrali
30     TN = fq.formula_composta_trapezio(a, b, N, f)
31     SN = fq.formula_composta_Simpson(a, b, N, f)
32
33     #calcolo errori
34     errori_T[k] = abs(I-TN)/abs(I)
35     errori_S[k] = abs(I-SN)/abs(I)
36     k = k + 1
37
38 #grafico degli errori
39 plt.figure(1)
40 plt.title("Confronto convergenza Trapezio e Simpson")
41 plt.semilogy(range_Nmax, errori_T, "k-", label="Err. Trapezio")
42 plt.semilogy(range_Nmax, errori_S, "b-", label="Err. Simpson")
43 plt.legend()
44 plt.show()
```

Algoritmo della formula composta di Simpson usato nel programma:

```
| formule_quadratura.py x convergenza_simpson_trapezio.py x
27
28 def formula_composta_Simpson(a, b, N, fun):
29     #genero N+1 nodi equidistanti
30     x = np.linspace(a, b, N+1)
31     fx = fun(x)
32
33     sommal = 0
34     for i in range(N):
35         c = (x[i] + x[i+1])/2
36         sommal = sommal + fun(c)
37     somma2 = 0
38     for i in range(1, N):
39         somma2 = somma2 + fx[i]
40     S = (b-a)/(6*N)
41     S = S * (fx[0] + 4*sommal + 2*somma2 + fx[N])
42     return S
43
```

# Convergenza formula del trapezio e di Simpson composta - test



Dal seguente test si nota come la formula di composta di Simpson converge a zero, al crescere del numero di intervalli considerati, più velocemente rispetto alla formula composta del trapezio.