Corso: *Laboratorio di Calcolo Numerico*

Anno accademico 2019-20

Presentazione dei programmi in codice Python

Studente: *Montemurro Andrea*

Matricola: *681902*

E-mail: *a.montemurro@studenti.uniba.it*

Sommario

Repository e macchina di test

I codici Python dei programmi sono reperibili sulla piattaforma GitHub al seguente indirizzo: https://github.com/andmon97/CalcoloNumerico

Gli output mostrati sul documento sono ottenuti facendo girare i programmi su un calcolatore dotato di CPU Intel i5-6500 e 8 GB di RAM.

Per lo sviluppo è stato utilizzato l'IDE PyCharm 2020.1 con virtual environment basato su Python 3.8.

Condizionamento dei problemi

Condizionamento della somma

Codice Python

```
# Inserimento valori per esperimento
x = float(input('Inserire dato x: '))
y = float(input('Inserire dato y: '))
perturb = float(input('Inserire parturbazione : ')) #es. 1.0e-5

# Costruzione dati perturbati
from random import random
xt = x*(1 + perturb*(-1+2*random()))
yt = y*(1 + perturb*(-1+2*random()))
#nelle parentesi interna avremo un numero [-1.0, 1.0).


# Calcolo delle somme
S = x + y #somma corretta
St = xt + yt #somma perturbata


# Calcolo degli errori relativi
Ex = abs(x-xt)/abs(x)
Ey = abs(y-yt)/abs(y)
ES = abs(S-St)/abs(S)


# Visualizzazione dei risultati
print('Dati esatti x=%15.15f y=%15.15f' % (x,y))
print('Dati perturb x=%15.15f y=%15.15f' % (xt,yt))
print('Somma dati esatti S=%15.15f' % S)
print('Somma dati perturb S=%15.15f' % St)
print('Errori sui dati Ex=%e Ey=%e ' % (Ex,Ey))
print('Errore sul risultato Es=%e' % ES)
if ES>Ex and ES>Ey:
    print('L\'addizione rispetto ai dati inseriti risulta MALCONDIZIONATA')
else:
    print('L\'addizione rispetto ai dati inseriti risulta BENCONDIZIONATA')
```

Output

```
Inserire dato x: >? 2

Inserire dato y: >? 4

Inserire parturbazione : >? 1.0e-5

Dati esatti x=2.000000000000000 y=4.000000000000000

Dati perturb x=2.000013099051112 y=4.000025614475254

Somma dati esatti S=6.000000000000000

Somma dati perturb S=6.000038713526366
```

```
Errori sui dati Ex=6.549526e-06 Ey=6.403619e-06
Errore sul risultato Es=6.452254e-06
L'addizione rispetto ai dati inseriti risulta BENCONDIZIONATA
```

Output 2

```
Inserire dato x: >? 2.34563
Inserire dato y: >? -2.34562
Inserire parturbazione : >? 1.0E-5
Dati esatti x=2.345630000000000 y=-2.345620000000000
Dati perturb x=2.345621089516165 y=-2.345626305189255
Somma dati esatti S=0.000010000000000
Somma dati perturb S=-0.000005215673090
Errori sui dati Ex=3.798759e-06 Ey=2.688069e-06
Errore sul risultato Es=1.521567e+00
L'addizione rispetto ai dati inseriti risulta MALCONDIZIONATA
```

Considerazioni

Il codice è stato testato in due differenti situazioni, come evidenziato dai due output. Le due situazioni sono quelle che caratterizzano l'errore che si propaga con l'operazione di somma.

Dalla teoria, sappiamo che

$$e_r(s) \leq \frac{|x|}{|x+y|}\, e_r(x) + \frac{|y|}{|x+y|}\, e_r(y)$$

Ovvero, gli errori relativi su x e y vengono amplificati dai fattori $\frac{|x|}{|x+y|}$ e $\frac{|y|}{|x+y|}$. Possiamo affermare che qualora la somma che è al denominatore di questi due fattori è un numero abbastanza grande, allora il problema risulta ben condizionato poiché i due fattori saranno piccoli e di conseguenza gli errori non si propagano molto. Questo caso è infatti evidenziato dall' Output 1, dove è stato testato l'algoritmo sommando tra loro i numeri 2 e 4.

Nel caso dell'Output 2, sono stati sommati tra loro numeri x e y molto vicini tra loro e di segno opposto, così facendo la quantità x + y risulterà molto piccola, facendo diventare i due fattori che amplificano il risultato molto grandi. Ne consegue che in questo caso il problema risulta mal condizionato.

Condizionamento del prodotto

Codice Python

```
# Inserimento valori per esperimento
x = float(input('Inserire dato x: '))
y = float(input('Inserire dato y: '))
perturb = float(input('Inserire parturbazione : ')) #es. 1.0e-5


# Costruzione dati perturbati
from random import random
xt = x*(1 + perturb*(-1+2*random()))
yt = y*(1 + perturb*(-1+2*random()))
#nelle parentesi interna avremo un numero [-1.0, 1.0).


# Calcolo dei prodotti
P = x * y #prodotto corretto
Pt = xt * yt #prodotto perturbato


# Calcolo degli errori relativi
Ex = abs(x-xt)/abs(x)
Ey = abs(y-yt)/abs(y)
EP = abs(P-Pt)/abs(P) #EP sarà <= Ex + Ey + Ex*Ey (trascurabile)
#Se Ex e Ey sono piccoli, anche EP sarà piccolo, quindi
#il prodotto è sempre BENCONDIZIONATO


# Visualizzazione dei risultati
print('Dati esatti x=%15.12f y=%15.15f' % (x,y))
print('Dati perturb x=%15.12f y=%15.15f' % (xt,yt))
print('Prodotto dati esatti P=%15.15f' % P)
print('Prodotto dati perturb P=%15.15f' % Pt)
print('Errori sui dati Ex=%e Ey=%e ' % (Ex,Ey))
print('Errore sul risultato Ep=%e' % EP)
if EP< (Ex + Ey + Ex*Ey):
    print('Il prodotto è sempre BENCONDIZIONATO')
```

Output simulazione 1

```
Inserire dato x: >? 2

Inserire dato y: >? 3

Inserire parturbazione : >? 1.0e-5

Dati esatti x= 2.000000000000 y=3.000000000000000

Dati perturb x= 1.999990546860 y=3.000019073071534

Prodotto dati esatti P=6.000000000000000

Prodotto dati perturb P=6.000009786543335

Errori sui dati Ex=4.726570e-06 Ey=6.357691e-06

Errore sul risultato Ep=1.631091e-06

Il prodotto è sempre BENCONDIZIONATO
```

Considerazioni

Dalla teoria sappiamo che

$$e_r(p) <\approx e_r(x) + e_r(y)$$

Da questo deduciamo che l'errore relativo del prodotto sarà sempre abbastanza piccolo, assumendo che le due quantità di errore relativo sui dati sono piccoli (questo è assolutamente ragionevole).

Ne consegue, quindi, che il problema del prodotto è sempre ben condizionato, ovvero l'errore rimarrà sempre nell'ordine di grandezza dell'errore sui dati.

Condizionamento della funzione radice quadrata

Codice Python

```
Verifica condizionamento della funzione sqrt(x)

#Inserimento dati
x = float(input('Inserire x: '))
perturb = float(input('Inserire parturbazione : ')) #es. 1.0e-5

# Costruzione dati perturbati
from random import random
xt = x*(1 + perturb*(-1+2*random()))

#Calcolo della radice quadrata
import math
R = math.sqrt(x) #radice corretta
Rt = math.sqrt(xt) #radice perturbata

# Calcolo degli errori relativi
Ex = abs(x-xt)/abs(x)
ER = abs(R-Rt)/abs(R) #ER sarà approssimato a 1/2 * Ex
#Se Ex è piccolo, anche ER sarà piccolo, quindi
#la radice quadrata è sempre BENCONDIZIONATA


# Visualizzazione dei risultati
print('Dati esatti x=%15.15f' % x)
print('Dati perturb x=%15.15f' % xt)
print('Radice quadrata dato esatto R=%15.15f' % R)
print('Radice quadrata dato perturb Rt=%15.15f' % Rt)
print('Errori sui dati Ex=%e' % Ex)
print('Errore sul risultato Er=%e' % ER)
if (1/2)*Ex<=Ex:
    print('La radice quadrata è sempre BENCONDIZIONATA')
```

Output

```
Inserire x: >? 4
```

```
Inserire parturbazione : >? 1.0e-5
Dati esatti x=4.000000000000000
Dati perturb x=3.999974196163967
Radice quadrata dato esatto R=2.000000000000000
Radice quadrata dato perturb Rt=1.999993549030588
Errori sui dati Ex=6.450959e-06
Errore sul risultato Er=3.225485e-06
La radice quadrata è sempre BENCONDIZIONATA
```

Considerazioni

Per quanto riguarda le funzioni generiche, possiamo studiare il loro condizionamento attraverso un indice di condizionamento K, dato da

$$|x\ \frac{f'(x)}{f(x)}|$$

Per cui, essendo la nostra funzione la radice quadrata, esso sarà pari a

$$\left|x\ \frac{1}{2\sqrt{x}}\ \frac{1}{\sqrt{x}}\right| = \frac{1}{2}$$

Per cui, poiché

$$ER \leq K\ EX = \frac{1}{2}\ EX$$

L'errore sui dati sarà sempre minore o uguale alla metà dell'errore sui dati, quindi la radice quadrata è un problema sempre ben condizionato.

Condizionamento della funzione seno

Codice Python

```python
# Verifica condizionamento della funzione sin(x)
#Inserimento dati
x = float(input('Inserire x: '))
perturb = float(input('Inserire perturbazione : ')) #es. 1.0e-5

# Costruzione dati perturbati
from random import random
xt = x*(1 + perturb*(-1+2*random()))

#Calcolo del seno
import numpy as np
S = np.sin(x) #sin corretto
St = np.sin(xt) #sin perturbato
```

```
# Calcolo degli errori relativi
Ex = abs(x-xt)/abs(x)
ES = abs(S-St)/abs(S)

# Visualizzazione dei risultati
print('Dati esatti x=%15.15f' % x)
print('Dati perturb x=%15.15f' % xt)
print('Sin dato esatto S=%15.15f' % S)
print('Sin dato perturb S=%15.15f' % St)
print('Errore sui dati Ex=%e' % Ex)
print('Errore sul risultato Es=%e' % ES)
if abs((abs(x/np.tan(x))*Ex))<=Ex:
    print('Il sin della x scelta è BENCONDIZIONATO')
else :
    print('Il sin della x scelta è MALCONDIZIONATO')
```

Output

```
Inserire x: >? 1
Inserire perturbazione : >? 1.0E-5
Dati esatti x=1.000000000000000
Dati perturb x=0.999996696685901
Sin dato esatto S=0.841470984807897
Sin dato perturb S=0.841469200015081
Errore sui dati Ex=3.303314e-06
Errore sul risultato Es=2.121039e-06
Il sin della x scelta è BENCONDIZIONATO
```

Output 2

```
Inserire x: >? 3.123
Inserire perturbazione : >? 1.0e-5
Dati esatti x=3.123000000000000
Dati perturb x=3.122973885648083
Sin dato esatto S=0.018591582402588
Sin dato perturb S=0.018617692234600
Errore sui dati Ex=8.361944e-06
Errore sul risultato Es=1.404390e-03
Il sin della x scelta è MALCONDIZIONATO
```

Considerazioni

Valgono le considerazioni precedenti, in particolare in questo caso il numero di condizionamento è

$$\left|\frac{\cos(x)}{\sin(x)}\, x\right| = \left|\frac{x}{\tan(x)}\right|$$

Per cui, questo numero sarà grande quando il denominatore sarà piccolo, ossia per valori di periodo $k\pi \; \forall \; k \; \in N$.
Nelle simulazioni, infatti, questo è evidenziato, scegliendo x uguale a 1 il problema è ben condizionato, mentre con uno vicino a π nella simulazione 2 è mal condizionato.

Aritmetica del Calcolatore

Epsilon Machine

Codice Python

```
import numpy as np
u = 1.0
while (1+u)>1 :
    eps = u
    u = u/2;
print('Epsilon: %.20f' %eps)
p = 1 - np.log2(eps)
print('Cifre significative binario: %.5f' %p)
q = p * np.log10(2)
print('Cifre significative decimale: %.5f' %q)
```

Output

```
Epsilon: 0.00000000000000022204
Cifre significative binario: 53.00000
Cifre significative decimale: 15.95459
```

Considerazioni

L' epsilon machine è quel numero che sommato a 1 dà sempre 1 nell' aritmetica del calcolatore.

Esso può essere calcolato costruendo una successione, quindi con un ciclo nel codice nel quale si divide uno per la base di numerazione usata dal calcolatore in questione, quindi 2, ad ogni ripetizione del ciclo.

Conoscendo questo numero del calcolatore, si conosce anche la precisione e il numero di cifre significative nella rappresentazione dei numeri in base decimale e binaria.

Dall' output della precedente simulazione si evince che sul calcolatore utilizzato, i numeri in binario sono rappresentati con una precisione di 53 bit, 52 dei quali sono la mantissa e 1 è il bit nascosto.

Differenza tra aritmetica reale ed aritmetica dei calcolatori

Codice Python

```
print("Esempio 1")
a = 3.0e-16
b = 1.0
r1 = -1+(b+a)
r2 = (-1+b)+a
r3 = (-1 + (b + a)) / a
print("Operazione 1: -1 + (b + a) = %52.51f" % r1)
print("Operazione 1: (-1 + b) + a = %52.51f" % r2)
print("Operazione 1: (-1 + (b + a)) / a = %52.51f" % r3)
print("Esempio 2")
a = 2e-51
r1 = -1 + (b + a)
r2 = (-1 + b) + a
r3 = (-1 + (b + a)) / a
print("Operazione 1: -1 + (b + a) = %52.51f" % r1)
print("Operazione 1: (-1 + b) + a = %52.51f" % r2)
print("Operazione 1: (-1 + (b + a)) / a = %52.51f" % r3)
```

Output

```
Esempio 1
Operazione 1: -1 + (b + a) =
0.000000000000000222044604925031308084726333618164062
Operazione 1: (-1 + b) + a =
0.000000000000000299999999999999993729336017210381069
Operazione 1: (-1 + (b + a)) / a =
0.740148683083437730623188599565764889121055603027344
Esempio 2
Operazione 1: -1 + (b + a) =
0.000000000000000000000000000000000000000000000000000
Operazione 1: (-1 + b) + a =
0.000000000000000000000000000000000000000000000000002
Operazione 1: (-1 + (b + a)) / a =
0.000000000000000000000000000000000000000000000000000
```

Considerazioni

Posto a = 3.0×10−16 e b = 1.0, si verifica, eseguendo la semplice istruzione, che al calcolatore il valore di −1 + (b + a) risulta differente dal valore atteso a e che, in questo caso, solo una cifra decimale risulti corretta. Diversamente, se invece di calcolare −1 + (b + a), avessimo organizzato il calcolo come (−1 +b) + a non ci sarebbero stati problemi. Analogamente si può verificare che l'operazione (−1 + (b + a)) / a non restituisce il risultato 1 che invece ci attenderemmo eseguendo il calcolo in aritmetica reale (vedi output "esempio 1"). Questi risultati sono una ulteriore verifica di quanto detto riguardo il condizionamento dell'operazione somma.

Ponendo invece a = 2×10-51, possiamo notare un diverso comportamento del calcolatore. Il valore di −1 + (b + a) risulta nuovamente differente da quello atteso, ma in questo caso non vi sono cifre decimali corrette. Infatti, l'esecuzione dell'operazione b + a necessita l'esecuzione di una operazione di floating che produce la perdita di informazioni.

Test Matrici

Matrice con valori casuali

Codice Python

```
import numpy as np
from random import random
import math
n = 9
# Costruzione matrice test (composta da numeri casuali tra 1 e 10)
A = np.array([[float(math.ceil(random()*10)) for j in range(n)] for i in range(n)])
# Costruzione soluzione del problema test (vettore formato da numeri che vanno da 1 a
n)
xsol = np.ones((n,1))
# Costruzione vettore termini noti
b = np.dot(A,xsol)
print('\nSistema di equazioni lineari Ax = b: \n')
for i in range (n):
    for j in range (n):
        print('%2d*x%d ' %(A[i,j],j+1), end = " ")
        if j != (n-1):
            print('+', end = " ")
        else:
            print(' = %d\n' %b[i])
# Calcolo soluzione del sistema di equalzioni lineari
xt = np.linalg.solve(A,b)
print("\nSoluzione del sistema ipotizzata:")
for i in range (n):
    print("%.15f" %xsol[i])
print("\nSoluzione del sistema calcolata:")
for i in range (n):
    print("%.15f" %xt[i])
# Calcolo numero di condizione di A (ottenuto dal teorema del condizionamento)
K = np.linalg.cond(A) #K(A) = ||A||*||A^(-1)|| sempre >= 1
#Calcolo dell'errore sulla soluzione del sistema
Ex = np.linalg.norm(xsol-xt)/np.linalg.norm(xsol)
print('\nNumero di condizione di A : %e' % K)
print('Errore soluzione sistema : %e' % Ex)
```

Output

```
Sistema di equazioni lineari Ax = b:
 3*x1  +  8*x2  +  4*x3  +  3*x4  +  8*x5  + 10*x6  +  7*x7  +  1*x8  +  2*x9
= 46
 4*x1  +  8*x2  +  5*x3  +  5*x4  +  4*x5  +  1*x6  +  3*x7  +  5*x8  +  6*x9
= 41
```

```
 1*x1  +  1*x2  + 10*x3  + 10*x4  +  4*x5  +  9*x6  + 10*x7  +  6*x8  +  5*x9
= 56
 6*x1  +  1*x2  +  7*x3  +  5*x4  +  1*x5  +  9*x6  +  3*x7  +  8*x8  +  1*x9
= 41
 1*x1  +  5*x2  +  6*x3  + 10*x4  + 10*x5  +  1*x6  +  8*x7  +  5*x8  +  8*x9
= 54
 4*x1  + 10*x2  +  1*x3  +  6*x4  +  8*x5  +  8*x6  +  3*x7  +  7*x8  +  5*x9
= 52
 4*x1  +  6*x2  +  8*x3  +  2*x4  +  9*x5  +  4*x6  +  1*x7  +  5*x8  +  3*x9
= 42
 2*x1  +  2*x2  +  8*x3  +  6*x4  +  5*x5  +  9*x6  + 10*x7  +  6*x8  +  4*x9
= 52
 2*x1  +  3*x2  +  3*x3  +  7*x4  +  5*x5  +  1*x6  +  9*x7  +  8*x8  +  5*x9
= 43
Soluzione del sistema ipotizzata:
1.000000000000000
1.000000000000000
1.000000000000000
1.000000000000000
1.000000000000000
1.000000000000000
1.000000000000000
1.000000000000000
1.000000000000000
Soluzione del sistema calcolata:
1.000000000000002
0.999999999999999
0.999999999999999
0.999999999999997
1.000000000000000
1.000000000000001
1.000000000000000
1.000000000000000
1.000000000000003
Numero di condizione di A : 6.627618e+01
Errore soluzione sistema : 1.695490e-15
```

Considerazioni

Osserviamo che il numero di condizione K(A) non è molto grande, dunque il sistema con una matrice formata da valori random risulta ben condizionato e quindi anche l'errore sui risultati risultano essere molto piccoli.

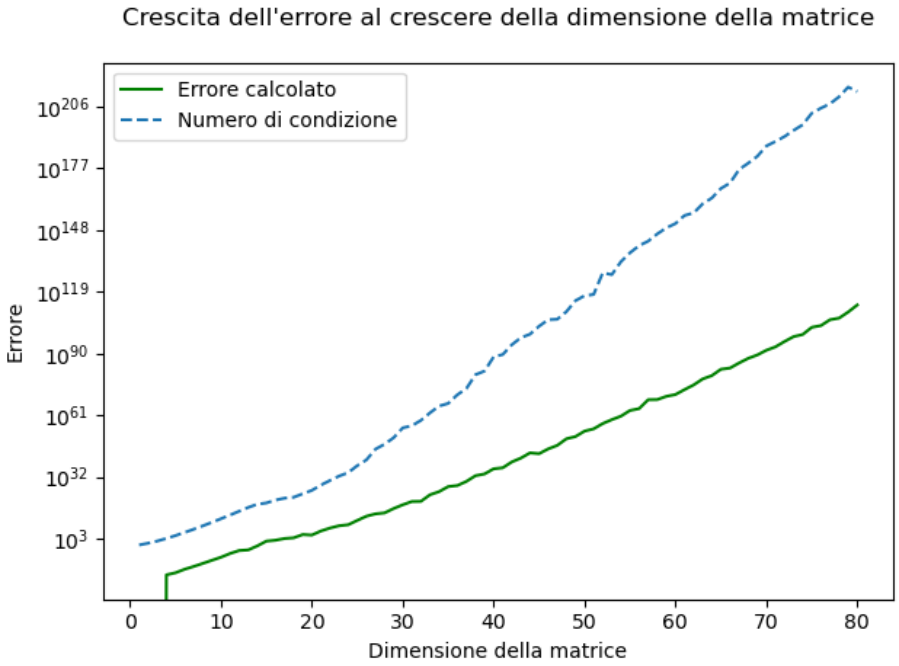
Crescere dell'errore al crescere della dimensione della matrice di Vandermonde

Codice Python

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

n = 80 # Dimensione massima della matrice
xdim = np.arange(1, n + 1)
y = np.zeros(n) # Array contenente gli errori di ogni matrice
z = np.zeros(n) # Array contenente il numero di condizione di ogni matrice

# Calcolo dei risultati
# Ogni iterazione corrisponde alla risoluzione di un sistema di equazioni #lineari di
```

```
ordine d
for dim in range(1, n + 1):
    # Costruzione matrice di Vandermonde
    c = np.array([i+1 for i in range(dim)])
    A = np.array([[float(c[j]) ** i for j in range(dim)] for i in range(dim)])
    # Costruzione soluzione
    xsol = np.ones((dim,1))
    # Costruzione vettore termini noti
    b = np.dot(A, xsol)
    # Calcolo soluzione numerica
    x = np.linalg.solve(A,b)
    # Calcolo numero di condizione di A e dell'errore
    z[dim - 1] = np.linalg.cond(A)
    y[dim - 1] = np.linalg.norm(xsol - x) / np.linalg.norm(xsol)

# Costruzione del grafico
plt.title("Crescita dell'errore al crescere della dimensione della matrice\n")
plt.xlabel('Dimensione della matrice')
plt.ylabel('Errore')
plt.yscale('log')
plt.plot(xdim, y, '-g', label = 'Errore calcolato')
plt.plot(xdim, z, '--', label = 'Numero di condizione')
plt.legend()
plt.show()
```

Output



Considerazioni

Osserviamo che al crescere della dimensione della matrice l'errore cresce molto a causa del mal condizionamento di questo tipo di matrice di test.

Inoltre, l'errore cresce quasi proporzionalmente al valore di K(A)* ε -machine.

Metodo di eliminazione di GAUSS

Eliminazione di Gauss senza pivoting

Codice Python

```
import numpy as np

def EliminazGauss(A,b):
    A = np.copy(A)
    b = np.copy(b)
    n = len(A)
    for j in range(n-1):
        for i in range(j+1,n):
            m = A[i,j]/A[j,j]
            A[i,j] = 0
            for k in range(j+1,n):
                A[i,k] = A[i,k] - m*A[j,k]
            b[i] = b[i] - m*b[j]
    return A, b


def SostIndietro(A,b):
    n = len(A)
    x = np.zeros(n)
    #controlla se il determinante è != 0
    if abs(np.prod(np.diag(A))) > 1.0e-14:
        for i in range(n-1,-1,-1):
            S = 0
            for j in range(i+1,n):
                S = S + A[i,j]*x[j]
            x[i] = (b[i] - S)/A[i,i]
    else:
        print('Errore: matrice singolare o mal condizionata')
        return [] #restituisce un vettore vuoto per segnalare che det != 0
    return x

import numpy as np
from random import random
import math

# Costruzione matrice (composta da numeri casuali tra 1 e 10)
n = 5
A = np.array([[float(math.ceil(random()*10)) for j in range(n)] for i in range(n)])
xsol = np.ones((n,1))
b = np.dot(A,xsol)
# Stampa del sistema
print('\nSistema di equazioni lineari Ax = b: \n')
for i in range (n):
    for j in range (n):
        print('%5.2f*x%d ' %(A[i,j],j+1), end = " ")
        if j != (n-1):
            print('+', end = " ")
        else:
            print(' = %2.2f\n' %b[i])

# Risoluzione del sistema
```

```
U, c = EliminazGauss(A,b)

# Stampa del sistema equivalente
print('\nSistema equivalente ottenuto con l\'algoritmo di Gauss senza pivoting: \n')
for i in range (n):
   for j in range (n):
        print('%6.2f*x%d ' %(U[i,j],j+1), end = " ")
        if j != (n-1):
           print('+', end = " ")
        else:
           print(' = %3.2f\n' %c[i])
   xt = SostIndietro(U,c)
if len(xt) > 0:
   print("\nSoluzione del sistema ipotizzata:")
   for i in range (n):
      print("%.20f" %xsol[i])
   print("\nSoluzione del sistema calcolata:")
   for i in range (n):
      print("%.20f" %xt[i])
   err = np.linalg.norm(xt-xsol)/np.linalg.norm(xsol)
   print('\nErrore in soluzione sistema: % e' % err)
```

Output

```
Sistema di equazioni lineari Ax = b:

 6.00*x1  +  4.00*x2  +  8.00*x3  +  5.00*x4  +  1.00*x5   = 24.00

10.00*x1  +  5.00*x2  +  8.00*x3  +  1.00*x4  +  9.00*x5   = 33.00

 6.00*x1  +  3.00*x2  +  7.00*x3  +  6.00*x4  + 10.00*x5   = 32.00

 9.00*x1  +  5.00*x2  +  9.00*x3  + 10.00*x4  +  1.00*x5   = 34.00

 4.00*x1  +  1.00*x2  +  1.00*x3  +  8.00*x4  +  1.00*x5   = 15.00


Sistema equivalente ottenuto con l'algoritmo di Gauss senza pivoting:

  6.00*x1  +   4.00*x2  +   8.00*x3  +   5.00*x4  +   1.00*x5   = 24.00

  0.00*x1  +  -1.67*x2  +  -5.33*x3  +  -7.33*x4  +   7.33*x5   = -7.00

  0.00*x1  +   0.00*x2  +   2.20*x3  +   5.40*x4  +   4.60*x5   = 12.20

  0.00*x1  +   0.00*x2  +   0.00*x3  +   6.41*x4  +  -5.32*x5   = 1.09

  0.00*x1  +   0.00*x2  +   0.00*x3  +   0.00*x4  +  -1.17*x5   = -1.17


Soluzione del sistema ipotizzata:
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000

Soluzione del sistema calcolata:
0.99999999999999766853
1.00000000000001021405
```

```
0.9999999999999655831
1.0000000000000044409
1.0000000000000066613

Errore in soluzione sistema:  1.105662e-14
```

Eliminazione di Gauss con Pivoting

Codice Python

```python
import numpy as np

def EliminazGaussConPivoting(A,b):
    A = np.copy(A) ; b = np.copy(b)
    n = len(A)

    for j in range(n-1):
        #individuazione elemento pivot
        amax = abs(A[j,j])
        imax = j
        for i in range(j+1,n):
            if abs(A[i,j]) > amax:
                amax = abs(A[i,j])
                imax = i

        #eventuale scambio di riga
        if imax > j:
            for k in range (j,n):
                A[j,k], A[imax,k] = A[imax,k], A[j,k] #scambio
            b[j], b[imax] = b[imax], b[j]

         #eliminazione di Gauss sulla colonna j
        for i in range(j+1,n):
            m = A[i,j]/A[j,j]
            A[i,j] = 0
            for k in range(j+1,n):
                A[i,k] = A[i,k] - m*A[j,k]
            b[i] = b[i] - m*b[j]

    return A, b

def EliminazGaussSenzaPivoting(A,b):
    A = np.copy(A) ; b = np.copy(b)
    n = len(A)

    for j in range(n-1):
        for i in range(j+1,n):
            m = A[i,j]/A[j,j]
            A[i,j] = 0
            for k in range(j+1,n):
                A[i,k] = A[i,k] - m*A[j,k]
            b[i] = b[i] - m*b[j]
    return A, b


def SostIndietro(A,b):
    n = len(A)
    x = np.zeros(n)
```

```
    #controlla se il determinante è != 0
    if abs(np.prod(np.diag(A))) > 1.0e-14:
        for i in range(n-1,-1,-1):
            S = 0
            for j in range(i+1,n):
                S = S + A[i,j]*x[j]
            x[i] = (b[i] - S)/A[i,i]
    else:
        print('Errore: matrice singolare o mal condizionata')
        return [] #restituisce un vettore vuoto per segnalare che det != 0
    return x

import numpy as np
from random import random
import math

# Costruzione matrice (composta da numeri casuali tra 1 e 10)
n = 5
A = np.array([[float(math.ceil(random()*10)) for j in range(n)] for i in range(n)])
xsol = np.ones((n,1))
b = np.dot(A,xsol)

# Stampa del sistema
print('\nSistema di equazioni lineari Ax = b: \n')
for i in range (n):
    for j in range (n):
        print('%5.2f*x%d ' %(A[i,j],j+1), end = " ")
        if j != (n-1):
            print('+', end = " ")
        else:
            print(' = %2.2f\n' %b[i])

# Risoluzione del sistema SENZA pivoting
U, c = EliminazGaussSenzaPivoting(A,b)

# Stampa del sistema
print('\nSistema equivalente ottenuto con l\'algoritmo di Gauss SENZA pivoting: \n')
for i in range (n):
    for j in range (n):
        print('%6.2f*x%d ' %(U[i,j],j+1), end = " ")
        if j != (n-1):
            print('+', end = " ")
        else:
            print(' = %3.2f\n' %c[i])

# Risoluzione del sistema con pivoting
U, c = EliminazGaussConPivoting(A,b)

# Stampa del sistema
print('\nSistema equivalente ottenuto con l\'algoritmo di Gauss CON pivoting: \n')
for i in range (n):
    for j in range (n):
        print('%6.2f*x%d ' %(U[i,j],j+1), end = " ")
        if j != (n-1):
            print('+', end = " ")
        else:
            print(' = %3.2f\n' %c[i])

xt = SostIndietro(U,c)
if len(xt) > 0:
```

```
    print("\nSoluzione del sistema ipotizzata:")
    for i in range (n):
        print("%.20f" %xsol[i])

    print("\nSoluzione del sistema calcolata:")
    for i in range (n):
        print("%.20f" %xt[i])
    err = np.linalg.norm(xt-xsol)/np.linalg.norm(xsol)
    print('\nErrore in soluzione sistema: % e' % err)
```

Output simulazione 1

```
Sistema di equazioni lineari Ax = b:

10.00*x1  +  2.00*x2  +  7.00*x3  +  3.00*x4  +  9.00*x5   = 31.00

10.00*x1  +  1.00*x2  +  6.00*x3  +  9.00*x4  +  6.00*x5   = 32.00

10.00*x1  + 10.00*x2  +  3.00*x3  +  4.00*x4  +  9.00*x5   = 36.00

 5.00*x1  +  9.00*x2  +  8.00*x3  +  5.00*x4  + 10.00*x5   = 37.00

 2.00*x1  +  9.00*x2  +  5.00*x3  +  1.00*x4  +  3.00*x5   = 20.00


Sistema equivalente ottenuto con l'algoritmo di Gauss SENZA pivoting:

 10.00*x1  +   2.00*x2  +   7.00*x3  +   3.00*x4  +   9.00*x5   = 31.00

  0.00*x1  +  -1.00*x2  +  -1.00*x3  +   6.00*x4  +  -3.00*x5   = 1.00

  0.00*x1  +   0.00*x2  + -12.00*x3  +  49.00*x4  + -24.00*x5   = 13.00

  0.00*x1  +   0.00*x2  +   0.00*x3  +  37.21*x4  + -11.50*x5   = 25.71

  0.00*x1  +   0.00*x2  +   0.00*x3  +   0.00*x4  +  -4.84*x5   = -4.84


Sistema equivalente ottenuto con l'algoritmo di Gauss CON pivoting:

 10.00*x1  +   2.00*x2  +   7.00*x3  +   3.00*x4  +   9.00*x5   = 31.00

  0.00*x1  +   8.60*x2  +   3.60*x3  +   0.40*x4  +   1.20*x5   = 13.80

  0.00*x1  +   0.00*x2  +  -7.35*x3  +   0.63*x4  +  -1.12*x5   = -7.84

  0.00*x1  +   0.00*x2  +   0.00*x3  +   6.00*x4  +  -2.77*x5   = 16.02

  0.00*x1  +   0.00*x2  +   0.00*x3  +   0.00*x4  +   5.70*x5   = 7.40


Soluzione del sistema ipotizzata:
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000

Soluzione del sistema calcolata:
-0.01290993932206632201
0.79032176413650545577
```

```
1.14878951741238743978
3.27258395655750566888
1.29879748593204746854

Errore in soluzione sistema:  2.519128e+00
```

Considerazioni

Osserviamo che il sistema equivalente calcolato con MEG senza Pivoting ha coefficienti più grandi rispetto a quello con Pivoting, questo perché i due algoritmi a parità di costo di calcolo non hanno la stessa stabilità.

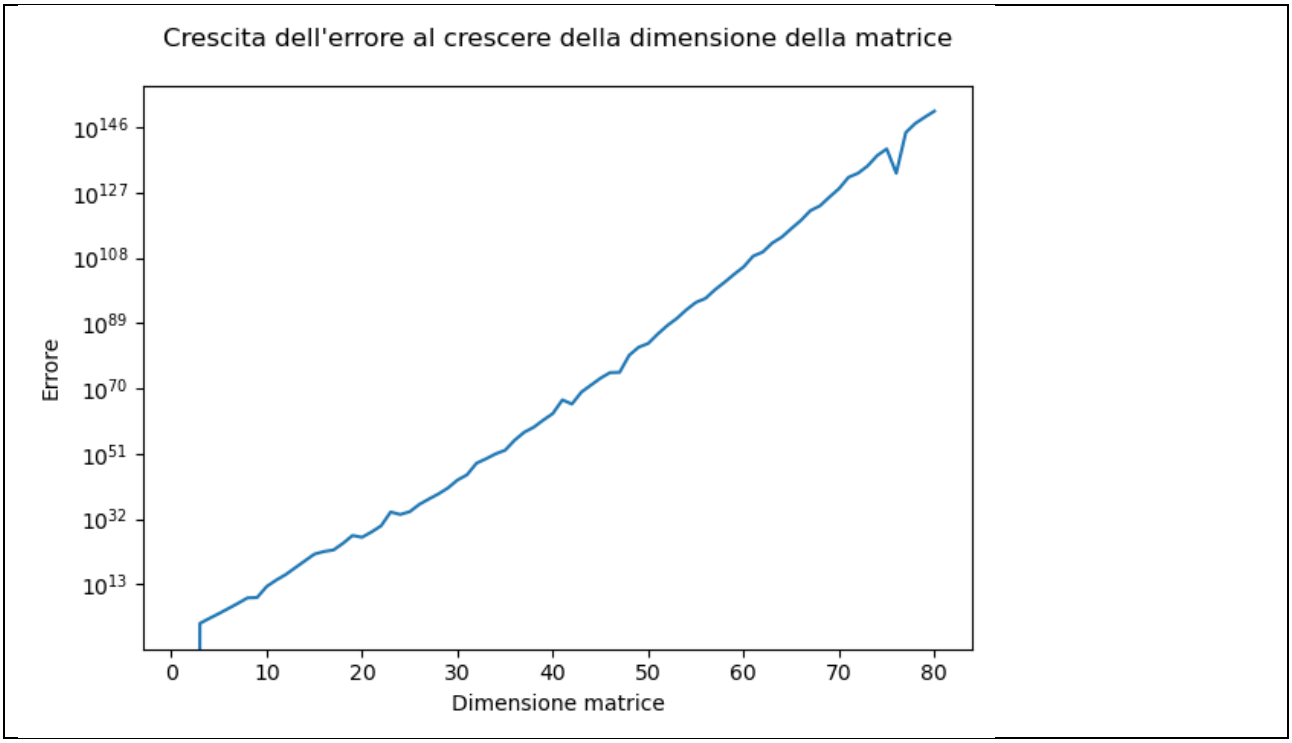
Crescita dell'errore al crescere della dimensione matrice di Vandermonde

Codice Python

```python
import numpy as np

def EliminazGaussConPivoting(A,b):
    A = np.copy(A) ; b = np.copy(b)
    n = len(A)

    for j in range(n-1):
        #individuazione elemento pivot
        amax = abs(A[j,j])
        imax = j
        for i in range(j+1,n):
            if abs(A[i,j]) > amax:
                amax = abs(A[i,j])
                imax = i

        #eventuale scambio di riga
        if imax > j:
            for k in range (j,n):
                A[j,k], A[imax,k] = A[imax,k], A[j,k] #scambio
            b[j], b[imax] = b[imax], b[j]

         #eliminazione di Gauss sulla colonna j
        for i in range(j+1,n):
            m = A[i,j]/A[j,j]
            A[i,j] = 0
            for k in range(j+1,n):
                A[i,k] = A[i,k] - m*A[j,k]
            b[i] = b[i] - m*b[j]

    return A, b


def SostIndietro(A,b):
    n = len(A)
    x = np.zeros(n)
    #controlla se il determinante è != 0
    if abs(np.prod(np.diag(A))) > 1.0e-14:
        for i in range(n-1,-1,-1):
            S = 0
            for j in range(i+1,n):
```

```
            S = S + A[i,j]*x[j]
        x[i] = (b[i] - S)/A[i,i]
    else:
        print('Errore: matrice singolare o mal condizionata')
        return [] #restituisce un vettore vuoto per segnalare che det != 0
    return x


#funzione che riceve come parametro il vettore [1,n] di grandezze di matrici e calcola
# l'errore del sistema di eq. lin.
def erroreSistemaEqLin(x):
    Ex = np.zeros(x.size) #inizializzo un vettore di errori di n elementi
    #effettuiamo il calcolo dell'errore per le matrici di dimensione da 1 a x.size
    for n in range(1,x.size+1):
        # Costruzione vettore c per la matrice di Vandermonde in [1,n]
        c = np.array([i+1 for i in range(n)] )

        # Costruzione matrice di Vandermonde (c[j]^i)
        A = np.array([[ float (c[j])**i for j in range(n)] for i in range(n)])

        # Costruzione soluzione del problema test (vettore formato da tutti 1)
        xsol = np.ones((n,1))

        # Costruzione vettore termini noti
        b = np.dot(A,xsol)

        # Risoluzione del sistema
        U, c = EliminazGaussConPivoting(A,b)
        xt = SostIndietro(U,c)

        #Calcolo dell'errore sulla soluzione del sistema
        Ex[n-1] = np.linalg.norm(xsol-xt)/np.linalg.norm(xsol)
    return Ex

import numpy as np
import matplotlib.pyplot as plt
n = 100
x = np.arange(1, n)
y = erroreSistemaEqLin(x)
plt.title("Crescita dell'errore al crescere della dimensione della matrice\n")

plt.semilogy(x, y)
plt.xlabel('Dimensione matrice')
plt.ylabel('Errore')
plt.show()
```

Output



Fattorizzazione A = L*U

Fattorizzazione senza Pivoting

Codice Python

```
def stampaMatrice(A):
    n = len(A)
    for i in range (n):
        for j in range (n):
            print('%5.2f ' %(A[i,j]), end = " ")
            if j == (n-1):
                print('\n')

import numpy as np

def FattLU(A):
    A = np.copy(A)
    n = len(A)
    L = np.zeros((n,n))
    for j in range(n-1):
        L[j,j] = 1
        for i in range(j+1,n):
            m = A[i,j]/A[j,j]
            A[i,j] = 0
            for k in range(j+1,n):
                A[i,k] = A[i,k] - m*A[j,k]
            L[i,j] = m
```

```
    L[n-1,n-1] = 1
    return L, A #restutuisce le matrici L e U


def SostIndietro(A,b):
    n = len(A)
    x = np.zeros(n)
    #controlla se il determinante è != 0
    if abs(np.prod(np.diag(A))) > 1.0e-14:
        for i in range(n-1,-1,-1):
            S = 0
            for j in range(i+1,n):
                S = S + A[i,j]*x[j]
            x[i] = (b[i] - S)/A[i,i]
    else:
        print('Errore: matrice singolare o mal condizionata')
        return [] #restituisce un vettore vuoto per segnalare che det != 0
    return x

def SostAvanti(A,b):
    n = len(A)
    y = np.zeros(n)
    #controlla se il determinante è != 0
    if abs(np.prod(np.diag(A))) > 1.0e-14:
        for i in range(n):
            S = 0
            for j in range(i):
                S = S + A[i,j]*y[j]
            y[i] = (b[i] - S)/A[i,i]
    else:
        print('Errore: matrice singolare o mal condizionata')
        return [] #restituisce un vettore vuoto per segnalare che det != 0
    return y

from random import random
import math
# Costruzione matrice (composta da numeri casuali tra 1 e 10)
n = 5
A = np.array([[float(math.ceil(random()*10)) for j in range(n)] for i in range(n)])
xsol = np.ones((n,1))
b = np.dot(A,xsol)

# Stampa del sistema
print('\nSistema di equazioni lineari Ax = b: \n')
for i in range (n):
    for j in range (n):
        print('%5.2f*x%d ' %(A[i,j],j+1), end = " ")
        if j != (n-1):
            print('+', end = " ")
        else:
            print(' = %2.2f\n' %b[i])

# Fattorizzazione A=L*U
L, U = FattLU(A)

# Stampa delle matrici L e U
print('\nMatrice L: \n')
stampaMatrice(L)

print('\nMatrice U: \n')
```

```
stampaMatrice(U)

yt = SostAvanti(L,b)
xt = SostIndietro(U,yt)

if len(xt) > 0:
   print("\nSoluzione del sistema ipotizzata:")
   for i in range (n):
      print("%.20f" %xsol[i])

   print("\nSoluzione del sistema calcolata:")
   for i in range (n):
      print("%.20f" %xt[i])

   err = np.linalg.norm(xt-xsol)/np.linalg.norm(xsol)
   print('\nErrore in soluzione sistema: % e' % err)

   test = np.linalg.norm(np.dot(L,U)-A)
   print('\nTest differenza ||L*U-A|| = %e' % test)
```

Output

```
Sistema di equazioni lineari Ax = b:

 8.00*x1  + 10.00*x2  +  8.00*x3  +  6.00*x4  +  8.00*x5   = 40.00

 7.00*x1  +  6.00*x2  +  9.00*x3  +  4.00*x4  +  3.00*x5   = 29.00

 1.00*x1  +  9.00*x2  + 10.00*x3  +  2.00*x4  +  3.00*x5   = 25.00

 8.00*x1  +  2.00*x2  +  8.00*x3  +  3.00*x4  +  7.00*x5   = 28.00

 4.00*x1  +  6.00*x2  +  8.00*x3  +  7.00*x4  +  3.00*x5   = 28.00


Matrice L:

 1.00   0.00   0.00   0.00   0.00

 0.88   1.00   0.00   0.00   0.00

 0.12  -2.82   1.00   0.00   0.00

 1.00   2.91  -0.40   1.00   0.00

 0.50  -0.36   0.32  -16.02   1.00


Matrice U:

 8.00  10.00   8.00   6.00   8.00

 0.00  -2.75   2.00  -1.25  -4.00

 0.00   0.00  14.64  -2.27  -9.27

 0.00   0.00   0.00  -0.27   6.95

 0.00   0.00   0.00   0.00  111.91
```

```
Soluzione del sistema ipotizzata:
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000
1.00000000000000000000

Soluzione del sistema calcolata:
1.00000000000000088818
1.00000000000000088818
0.99999999999999955591
0.99999999999999789058
0.99999999999999988898

Errore in soluzione sistema:  2.497385e-15

Test differenza ||L*U-A|| = 4.965068e-16
```

Considerazioni

Si osserva come, oltre all'errore nella soluzione del sistema, con la fattorizzazione si introduce anche un piccolo errore dovuto al calcolo delle matrici L e U.

Calcolo della derivata

Considerazioni

Di seguito saranno riportati tre esempi di calcolo di derivata di tre diverse funzioni, mediante il calcolo del rapporto incrementale variando il valore dell'incremento h.

Dai risultati che abbiamo ottenuto si può constatare che si può stimare il valore minimo di h per cui conviene approssimare la derivata mediante il rapporto incrementale. Dai grafici ottenuti si potrà notare che l'errore varia al cambiare di h, in particolare, in tutti i casi, dapprima l'errore diminuisce all'aumentare di h per poi aumentare.

Per cui il valore minimo dell'errore corrisponde il valore con cui conviene approssimare la derivata con il rapporto incrementale.

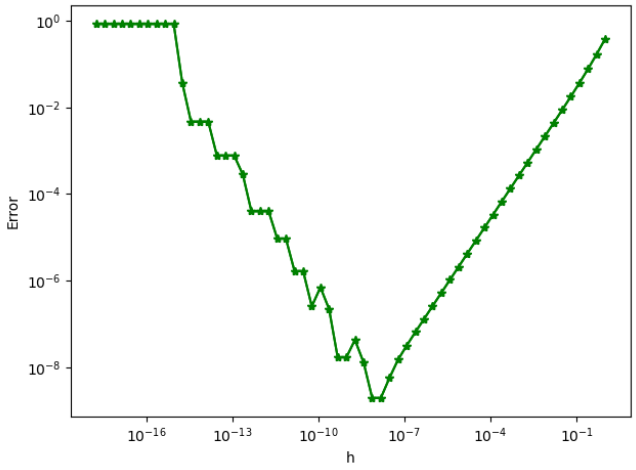
Funzione sin(x)

Codice Python

```
# Errore approssimazione derivata prima mediante rapporti incrementali
import matplotlib.pyplot as plt
import numpy as np

# Funzione di cui vogliamo calcolare la derivata
def f( x ):
   fx = np.sin(x)
   return fx
```

```
# Derivata prima della funzione
def f1( x ):
   f1x = np.cos(x)
   return f1x

# Punto in cui vogliamo calcolare la derivata
x = 10

# Sequenza di passi h in scala logaritmica
#h = np.logspace(-16,-0,100)
h = 2.0**(-np.array(range(60)))
# Calcolo della derivata in maniera esatta
df = f1(x)

errass = np.zeros(len(h))
print(' Derivata Approssimaz. Errore')
for i in range(len(h)):
   # Approssimazione derivata mediante rapporto incrementale
   dfh = (f(x+h[i])-f(x))/h[i]
   # Calcolo dell'errore assoluto
   errass[i] = np.abs(df-dfh)
   print('%10.8f %10.8f %e' % (df,dfh,errass[i]))

# Grafico in scala logaritmica
plt.figure(1)
plt.plot(h,errass,'*-g')
plt.loglog(h,errass,'*-g')
plt.xlabel('h')
plt.ylabel('Error')
plt.show()
```

Output

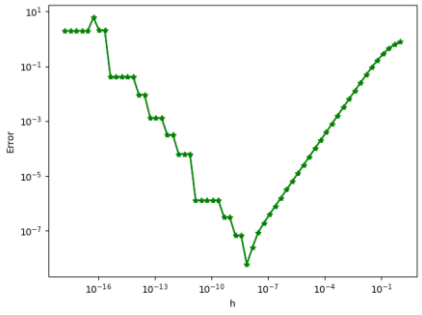


Funzione arccos(x)

Codice Python

```
# Errore approssimazione derivata prima mediante rapporti incrementali
import matplotlib.pyplot as plt
import numpy as np

# Funzione di cui vogliamo calcolare la derivata
def f( x ):
   fx = np.arccos(x)
   return fx

# Derivata prima della funzione
def f1( x ):
   f1x = -1/(np.sqrt(1-x**2))
   return f1x

# Punto in cui vogliamo calcolare la derivata
x = -0.86

# Sequenza di passi h in scala logaritmica
#h = np.logspace(-16,-0,100)
h = 2.0**(-np.array(range(60)))
# Calcolo della derivata in maniera esatta
df = f1(x)

errass = np.zeros(len(h))
print(' Derivata Approssimaz. Errore')
for i in range(len(h)):
   # Approssimazione derivata mediante rapporto incrementale
   dfh = (f(x+h[i])-f(x))/h[i]
   # Calcolo dell'errore assoluto
   errass[i] = np.abs(df-dfh)
   print('%10.8f %10.8f %e' % (df,dfh,errass[i]))

# Grafico in scala logaritmica
plt.figure(1)
plt.plot(h,errass,'*-g')
plt.loglog(h,errass,'*-g')
plt.xlabel('h')
plt.ylabel('Error')
plt.show()
```

Output

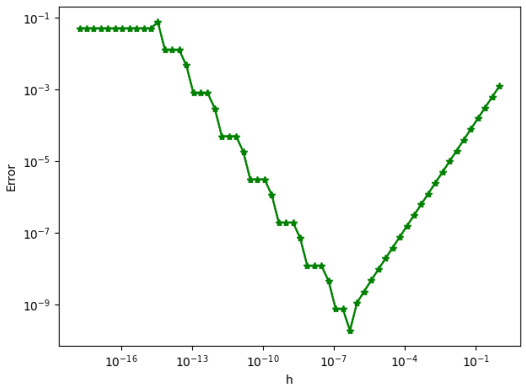


Funzione log(x)

Codice Python

```
# Errore approssimazione derivata prima mediante rapporti incrementali
import matplotlib.pyplot as plt
import numpy as np

# Funzione di cui vogliamo calcolare la derivata
def f( x ):
   fx = np.log(abs(x))
   return fx

# Derivata prima della funzione
def f1( x ):
   f1x = 1/x
   return f1x

# Punto in cui vogliamo calcolare la derivata
x = 20

# Sequenza di passi h in scala logaritmica
#h = np.logspace(-16,-0,100)
h = 2.0**(-np.array(range(60)))
# Calcolo della derivata in maniera esatta
df = f1(x)

errass = np.zeros(len(h))
print(' Derivata Approssimaz. Errore')
for i in range(len(h)):
   # Approssimazione derivata mediante rapporto incrementale
   dfh = (f(x+h[i])-f(x))/h[i]
   # Calcolo dell'errore assoluto
   errass[i] = np.abs(df-dfh)
   print('%10.8f %10.8f %e' % (df,dfh,errass[i]))

# Grafico in scala logaritmica
plt.figure(1)
plt.plot(h,errass,'*-g')
plt.loglog(h,errass,'*-g')
plt.xlabel('h')
plt.ylabel('Error')
plt.show()
```

Output



Metodi Iterativi

Metodo di Jacobi

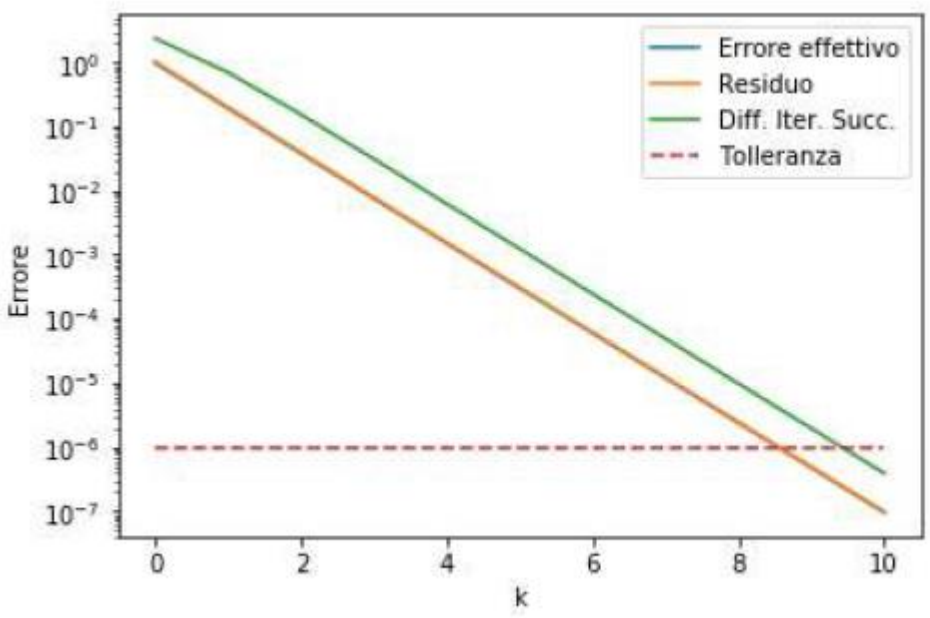
Codice Python

```
import numpy as np
import numpy.linalg as la
from random import random
import math

def Jacobi(A,b,x0,tol,Kmax,xsol):
    # Dimensione del problma
    n = len(A)

    # Inizializzare il ciclo di calcolo
    k = 0 ; stop = False
    x1 = np.zeros(n)

    err_rel_vett = np.zeros(Kmax)
    residuo = np.zeros(Kmax)
    diff_it_succ = np.zeros(Kmax)

    print('|-----+--------------+--------------+--------------+--------------|')
    print('| k   | Errore Rel   | Residuo Rel  | Diff It Succ | Tol          |')
    print('|-----+--------------+--------------+--------------+--------------|')
    while not (stop) and k < Kmax:
        for i in range(n):
            S = 0
            for j in range(0, i):
                S = S + A[i, j] * x0[j]
            for j in range(i + 1, n):
                S = S + A[i, j] * x0[j]
            x1[i] = (b[i] - S) / A[i, i]

        # Controllo della convergenza
        res_rel = la.norm(b - np.dot(A, x1)) / la.norm(b)  # residuo
        diff_rel = la.norm(x1 - x0) / la.norm(x1)  # confronto tra iter. succ.

        stop = (res_rel < tol) and (diff_rel < tol)

        err_rel = la.norm(x1 - xsol) / la.norm(xsol)  # errore effettivo al passo k
        err_rel_vett[k] = err_rel
        residuo[k] = res_rel;
        diff_it_succ[k] = diff_rel;

        k = k + 1

        print('| %3d | %e | %e | %e | %e | ' % (k, err_rel, res_rel, diff_rel, tol))
        x0 = np.copy(x1);

    print('|-----+--------------+--------------+--------------+--------------|')

    if not (stop):
        print('Processo non converge in %d iterazioni' % Kmax)

    return x1, k, err_rel_vett, residuo, diff_it_succ
```

```
# Costurzione del problema test con matrice a predominanza diagonale
n = 1200;
c = 10;  # se c è grande la convergenza è piu veloce
e1 = np.ones(n - 1)
A = np.diag(e1, -1) - c * np.eye(n) + np.diag(e1, 1)
xsol = np.ones(n)
b = np.dot(A, xsol)


# Parametri per il metodo: vettore di soluzione random con valori tra 1 e 10
x0 = np.array([float(math.ceil(random() * 10)) for i in range(n)])
Kmax = 100;
tol = 1.0e-6;

x, k, err_rel_vett, residuo, diff_it_succ = Jacobi(A, b, x0, tol, Kmax, xsol)

import matplotlib.pyplot as plt
plt.figure(1)

plt.semilogy(range(k), err_rel_vett[0:k], label='Errore effettivo')
plt.semilogy(range(k), residuo[0:k], label='Residuo')
plt.semilogy(range(k), diff_it_succ[0:k], label='Diff. Iter. Succ.')
plt.semilogy(range(k), np.ones(k) * tol, '--', label='Tolleranza')
plt.legend()
plt.xlabel('k')
plt.ylabel('Errore')
```

Output Simulazione

```
|-----+--------------+--------------+--------------+--------------|
| k   | Errore Rel   | Residuo Rel  | Diff It Succ | Tol          |
|-----+--------------+--------------+--------------+--------------|
|   1 | 9.799192e-01 | 1.027176e+00 | 2.372023e+00 | 1.000000e-06 |
|   2 | 1.918211e-01 | 1.991094e-01 | 6.956970e-01 | 1.000000e-06 |
|   3 | 3.794495e-02 | 3.917469e-02 | 1.537925e-01 | 1.000000e-06 |
|   4 | 7.536231e-03 | 7.752693e-03 | 3.112332e-02 | 1.000000e-06 |
|   5 | 1.499787e-03 | 1.538831e-03 | 6.194641e-03 | 1.000000e-06 |
|   6 | 2.988247e-04 | 3.059791e-04 | 1.230984e-03 | 1.000000e-06 |
|   7 | 5.958441e-05 | 6.091002e-05 | 2.448234e-04 | 1.000000e-06 |
|   8 | 1.188712e-05 | 1.213476e-05 | 4.873824e-05 | 1.000000e-06 |
|   9 | 2.372391e-06 | 2.418951e-06 | 9.709936e-06 | 1.000000e-06 |
|  10 | 4.736106e-07 | 4.824093e-07 | 1.935588e-06 | 1.000000e-06 |
|  11 | 9.457020e-08 | 9.624005e-08 | 3.860128e-07 | 1.000000e-06 |
|-----+--------------+--------------+--------------+--------------|
```



Considerazioni

Il test del metodo di Jacobi è stato effettuato utilizzando una matrice a diagonale predominanza in senso stretto: ciò ci assicura la convergenza del metodo.

La scelta del valore sulla diagonale principale, in questo caso c=10, influenza la velocità di convergenza e un valore maggiore provoca una convergenza più rapida.

Dal grafico si evince che il metodo viene interrotto quando entrambi i criteri di arresto vengono soddisfatti.

Confronto metodi iterativi – diretti

Codice Python

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import time


def GaussSeidel(A,b,x0,tol,Kmax):
    # Dimensione del problma
    n = len(A)
    # Inizializzare il ciclo di calcolo
    k = 0;
    stop = False
    x1 = np.zeros(n)
    while not (stop) and k < Kmax:
        for i in range(n):
            S = 0
            for j in range(0, i):
                S = S + A[i, j] * x1[j]
            for j in range(i + 1, n):
```

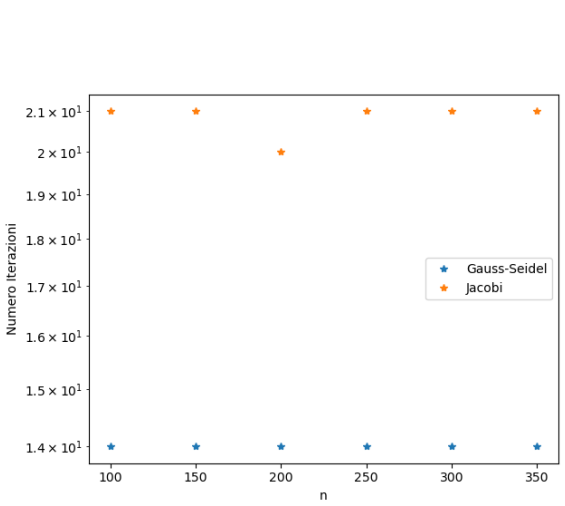
```
            S = S + A[i, j] * x0[j]
        x1[i] = (b[i] - S) / A[i, i]

    # Controllo della convergenza
    res_rel = la.norm(b - np.dot(A, x1)) / la.norm(b)
    diff_rel = la.norm(x1 - x0) / la.norm(x1)
    stop = (res_rel < tol) and (diff_rel < tol)
    k = k + 1
    x0 = np.copy(x1);

  if not (stop):
    print('Processo non converge in %d iterazioni' % Kmax)
  return x1, k

def Jacobi(A, b, x0, tol, Kmax):
  # Dimensione del problma
  n = len(A)
  # Inizializzare il ciclo di calcolo
  k = 0
  stop = False
  x1 = np.zeros(n)

  while not (stop) and k < Kmax:
    # calcolo di una nuova approssimazione
    for i in range(n):
      S = 0
      for j in range(0, n):
        if j == i: continue
        S = S + A[i, j] * x0[j]
      x1[i] = (b[i] - S) / A[i, i]
    # Controllo della convergenza
    res_rel = la.norm(b - np.dot(A, x1)) / la.norm(b)
    diff_rel = la.norm(x1 - x0) / la.norm(x1)

    stop = (res_rel < tol) and (diff_rel < tol)
    k = k + 1

    x0 = np.copy(x1)

  if not (stop):
    print('Processo non converge in %d iterazioni' % Kmax)
  return x1, k


def EGP(A,b):
  A = np.copy(A) ; b = np.copy(b)
  n = len(A)

  # Pivoting
  for j in range(n - 1):
    # Individuazioone del pivot
    amax = abs(A[j][j])
    imax = j

    for i in range(j + 1, n):
      if abs(A[i][j]) > amax:
        amax = abs(A[i][j])
        imax = i

    # Controllo sull'individuazione del pivot
```

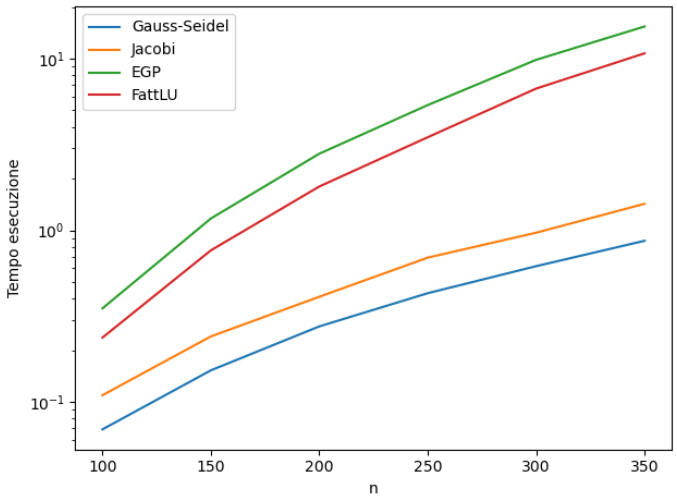
```
        if amax < 1.0e-15 :
            print("Matrice singolare o mal condizionata")
            return

        # Scambio di riga
        if imax > j :
            for k in range(j, n):
                A[j][k], A[imax][k] = A[imax][k], A[j][k]
            b[j][0], b[imax][0] = b[imax][0], b[j][0]

        # Eliminazione di Gauss sulla colonna j
        for i in range(j+1, n):
            m = A[i][j] / A[j][j]
            A[i][j] = 0
            for k in range(j + 1, n):
                A[i][k] = A[i][k] - m * A[j][k]
            b[i][0] = b[i][0] - m * b[j][0]

    x = SostIndietro(A, b)
    return x

def SostIndietro(A,b):
    n = len(A)
    x = np.zeros((n,1))
    if abs(np.prod(np.diag(A))) > 1.0e-14:
        for i in range(n-1,-1,-1):
            S = 0
            for j in range(i+1,n):
                S = S + A[i][j]*x[j][0]
            x[i][0] = (b[i][0] - S)/A[i][i]
    else:
        print('Errore: matrice singolare o mal condizionata')
    return x


def FattLU(A,b):
    A = np.copy(A)
    n = len(A)
    L = np.zeros((n,n))
    for j in range(n-1):
        L[j,j] = 1
        for i in range(j+1,n):
            m = A[i,j]/A[j,j]
            A[i,j] = 0
            for k in range(j+1,n):
                A[i,k] = A[i,k] - m*A[j,k]
            L[i,j] = m
    L[n-1,n-1] = 1

    yt = SostAvanti(L,b)
    xt = SostIndietro(A,yt)
    return xt


def SostAvanti(A, b):
    n = len(A) # Ordine del sistema
    x = np.zeros((n,1)) # Soluzione del sistema
    if abs(np.prod(np.diag(A))) > 1.0e-14:
        for i in range(n):
            S = 0
```

```
            for j in range(0, i):
                S += A[i][j] * x[j][0]
            x[i][0] = (b[i][0] - S) / A[i][i]
    else:
        print('Errore: matrice singolare o mal condizionata')
    return x



# Parametri per il metodo
Kmax = 100 ; tol = 1.0e-6 ;
n_vett = range(100,400,50)
#Statistiche Gauss Seidel
Iter_GS = np.zeros(len(n_vett))
Tempo_GS = np.zeros(len(n_vett))
#Statistiche Jacobi
Iter_J = np.zeros(len(n_vett))
Tempo_J = np.zeros(len(n_vett))
#Statistiche Eliminazione Gauss con Pivoting
Tempo_EGP = np.zeros(len(n_vett))
#Statistiche Fattorizzazione LU con Pivoting
Tempo_FattLUP = np.zeros(len(n_vett))

i = 0 ;

for n in n_vett:

    # Costurzione problema test di dimensione n
    c = 4
    e1 = np.ones(n-1)
    A = np.diag(e1,-1) - c*np.eye(n) + np.diag(e1,1)
    xsol = np.ones(n)
    b = np.dot(A,xsol)
    # Approssimazione iniziale
    x0 = 2*np.random.rand(n) - 1
    Ripet = 2
    # GaussSeidel
    inizio = time.time()
    for r in range(Ripet):
        x, k = GaussSeidel(A, b, x0, tol, Kmax)
    fine = time.time()
    tempo = (fine - inizio) / Ripet
    Iter_GS[i] = k
    Tempo_GS[i] = tempo

    # Jacobi
    inizio = time.time()
    for r in range(Ripet):
        x, k = Jacobi(A, b, x0, tol, Kmax)
    fine = time.time()
    tempo = (fine - inizio) / Ripet
    Iter_J[i] = k
    Tempo_J[i] = tempo

    b = np.reshape(b, (n, 1))

    # Eliminazione di Gauss con Pivoting
    inizio = time.time()
    for r in range(Ripet):
        x = EGP(A, b)
```

```
    fine = time.time()
    tempo = (fine - inizio) / Ripet
    Tempo_EGP[i] = tempo

    # Fattorizzazione LU senza Piv
    inizio = time.time()
    for r in range(Ripet):
        x = FattLU(A, b)
    fine = time.time()
    tempo = (fine - inizio) / Ripet
    Tempo_FattLUP[i] = tempo

    i = i + 1

plt.figure(1)
plt.semilogy(n_vett,Iter_GS,'*', label = 'Gauss-Seidel')
plt.semilogy(n_vett,Iter_J,'*', label='Jacobi')
plt.xlabel('n') ; plt.ylabel('Numero Iterazioni')
plt.legend()
plt.show()
plt.figure(2)
plt.semilogy(n_vett,Tempo_GS, label='Gauss-Seidel')
plt.semilogy(n_vett,Tempo_J, label='Jacobi')
plt.semilogy(n_vett,Tempo_EGP, label='EGP')
plt.semilogy(n_vett,Tempo_FattLUP, label='FattLU')
plt.xlabel('n') ; plt.ylabel('Tempo esecuzione')
plt.legend()
plt.show()
```

Output Simulazione





Considerazioni

Dai grafici sono evidenti le differenze tra i vari metodi.

Dal primo grafico si può notare come, a parità di tolleranza, il metodo di Gauss-Seidel abbia bisogno di un numero di iterazioni minore rispetto al metodo di Jacobi. Questo perché, come sappiamo dalla teoria, il metodo di Gauss-Seidel utilizza valori aggiornati e arriva prima alla approssimazione della soluzione. Bisogna ricordare che, però, il metodo di Jacobi può essere parallelizzato nel caso venga eseguito su un'architettura multiprocessore.

Dal secondo grafico si evince invece la differenza di velocità tra i metodi diretti (Fatt. LU e eliminazione di Gauss) rispetto ai metodi iterativi (Gauss-Seidel e Jacobi), anche se questi raggiungono solo un'approssimazione della soluzione del problema.

Interpolazione polinomiale

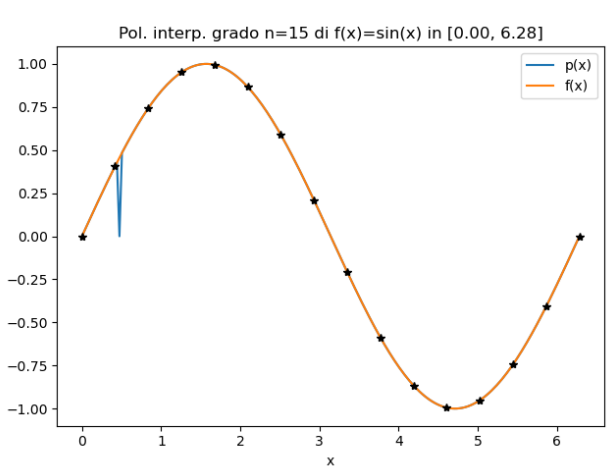
Prima formula baricentrica di Lagrange

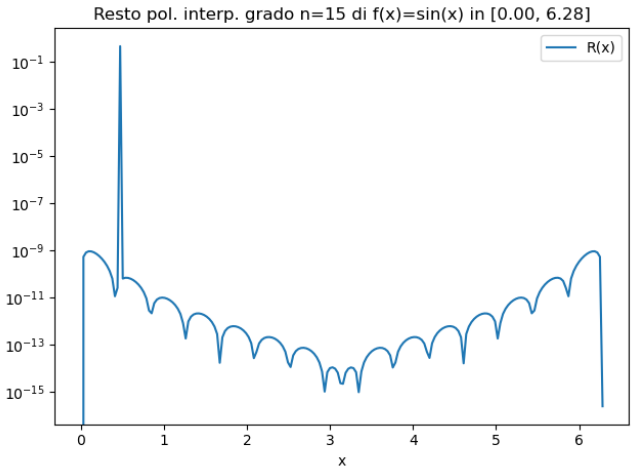
Codice Python

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

def z_coeff(x_nodi,y_nodi):
    n = len(x_nodi)
    x = np.ones((n,n))
    for i in range(n):
        for j in range(n):
            if j > i:
                x[i,j] = x_nodi[i] - x_nodi[j]
            elif j < i:
                x[i,j] = -x[j,i]
    z = np.zeros(n)
    for j in range(n):
        z[j] = y_nodi[j]/np.prod(x[j,:])
    return z

def calc_Lagrange(x_nodi,z,x):
    n = len(x_nodi)
    pin = np.prod(x-x_nodi)
    S = 0
    for j in range(n):
        S = S + z[j]/(x-x_nodi[j])
    p = pin * S
    return p

def Lagrange(x_nodi,y_nodi,x):
    z = z_coeff(x_nodi,y_nodi)
    m = len(x)
    p = np.zeros(m)
    for i in range(m):
        xx = x[i]
        check_nodi = abs(xx - x_nodi) < 1.0e-14
        if True in check_nodi:
            temp = np.where(check_nodi == True)
            i_nodo = temp[0][0]
            p[i_nodo] = y_nodi[i_nodo]
        else:
            p[i] = calc_Lagrange(x_nodi,z,xx)
    return p

# Funzione da interpolare
def funz(x):
```

```
    y = np.sin(x)
    return y

# Grado del polinomio di interpolazione
n = 15
# Calcolo dei nodi e dei valori associati
a = 0 ; b = 2*np.pi
x_nodi = np.linspace(a,b,n+1)
y_nodi = funz(x_nodi)
# Punti in cui calcolare il polinomio
x = np.linspace(a,b,200)

# Calcolo del polinomio e della funzione
p = Lagrange(x_nodi,y_nodi,x)
f = funz(x)

# Grafico del polinomio di interpolazione
plt.figure(0)
plt.plot(x,p,label='p(x)')
plt.plot(x,f,label='f(x)')
plt.plot(x_nodi,y_nodi,'k*')
plt.legend()
plt.xlabel('x')
plt.title('Pol. interp. grado n=%d di f(x)=sin(x) in [%4.2f, %4.2f]' % (n,a,b))
plt.show(block=False)

# Grafico del resto del polinomio di interpolazione
plt.figure(1)
plt.semilogy(x,abs(p-f),label='R(x)')
plt.legend()
plt.xlabel('x')
plt.title('Resto pol. interp. grado n=%d di f(x)=sin(x) in [%4.2f, %4.2f]' % (n,a,b))
plt.show(block=False)
```

Output





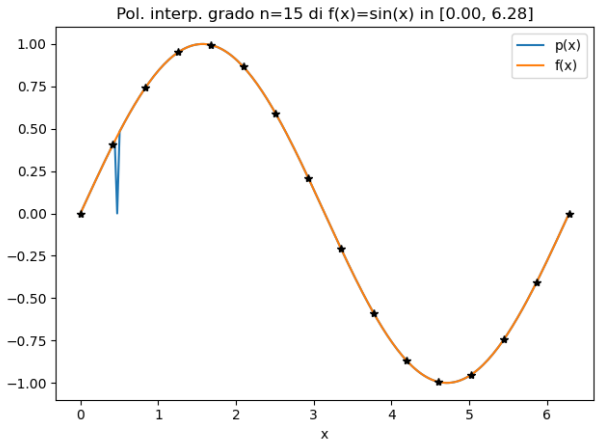
Formula di Newton alle differenze divise

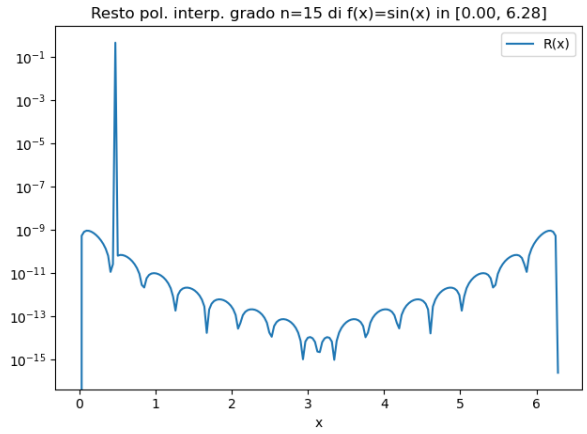
Codice Python

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

def Newton(x_nodi, y_nodi, x):
    n = len(x)
    p = np.zeros(n)
    d = diffDiv(x_nodi, y_nodi)
    for i in range(n):
        xx = x[i]
        # Se un punto in cui si calcola il polinomio corrisponde
        #ad un nodo: si imposta il valore corrispondente noto
        check_nodi = abs(xx - x_nodi) < 1.0e-14
        if True in check_nodi:
            temp = np.where(check_nodi == True)
            i_nodo = temp[0]
            p[i] = y_nodi[i_nodo]
        else:
            p[i] = calc_Newton(x_nodi, d, xx)
    return p

def diffDiv(x_nodi, y_nodi):
    n = len(x_nodi)
    d = np.copy(y_nodi)
    for j in range (1, n):
        for i in range(n-1, j-1, -1):
            d[i] = (d[i] - d[i-1]) / (x_nodi[i] - x_nodi[i-j])
    return d

def calc_Newton(x_nodi, d, x):
    n = len(x_nodi)
    p = d[-1]
    for i in range (n-2, -1, -1):
        p = p * (x - x_nodi[i]) + d[i]
    return p

# Funzione da interpolare
def funz(x):
    y = np.sin(x)
    return y

# Grado del polinomio di interpolazione
n = 15
# Calcolo dei nodi e dei valori associati
a = 0 ; b = 2*np.pi
x_nodi = np.linspace(a,b,n+1)
#k = np.array(range(n,-1,-1))
#x_nodi = (a+b)/2 +(b-a)/2*np.cos((2*k+1)*np.pi/2/(n+1))
y_nodi = funz(x_nodi)

# Punti in cui calcolare il polinomio
x = np.linspace(a,b,200)
# Calcolo del polinomio e della funzione
p = Newton(x_nodi, y_nodi, x)
f = funz(x)
# Grafico del polinomio di interpolazione plt.figure(0)
```

```
plt.plot(x,p,label='p(x)')
plt.plot(x,f,label='f(x)')
plt.plot(x_nodi,y_nodi,'k*')
plt.legend()
plt.xlabel('x')
plt.title('Pol. interp. grado n=%d di f(x)=sin(x) in [%4.2f %4.2f]' % (n,a,b))
plt.show(block=False)
# Grafico del resto del polinomio di interpolazione
plt.figure(1)
plt.semilogy(x,abs(p-f),label='R(x)')
plt.legend()
plt.xlabel('x')
plt.title('Resto pol. interp. grado n=%d di f(x)=sin(x) in [%4.2f %4.2f]' % (n,a,b))
plt.show(block=False)
```

Output

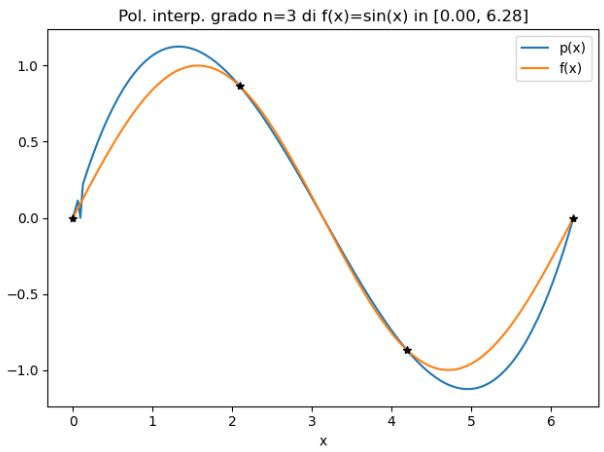


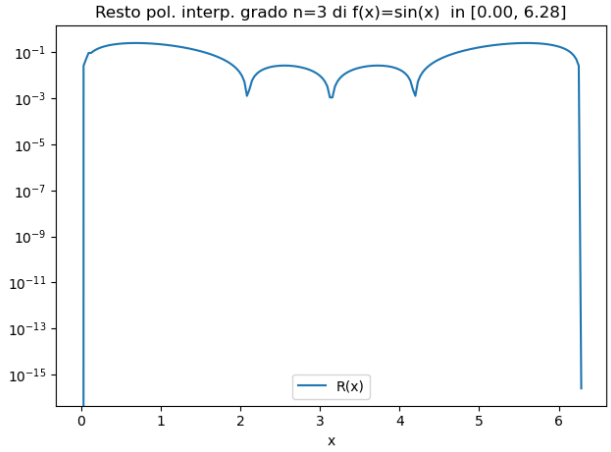


Considerazioni

Osserviamo che il polinomio di interpolazione di grado 10 ottenuto con la formula di newton è lo stesso di quello ottenuto con la formula di Lagrange, e questo perché secondo il teorema fondamentale dell'interpolazione esiste un solo polinomio di grado n che soddisfa i vincoli di interpolazione.

Se invece proviamo a diminuire il grado e il numero di nodi otteniamo che l'interpolazione è meno precisa.

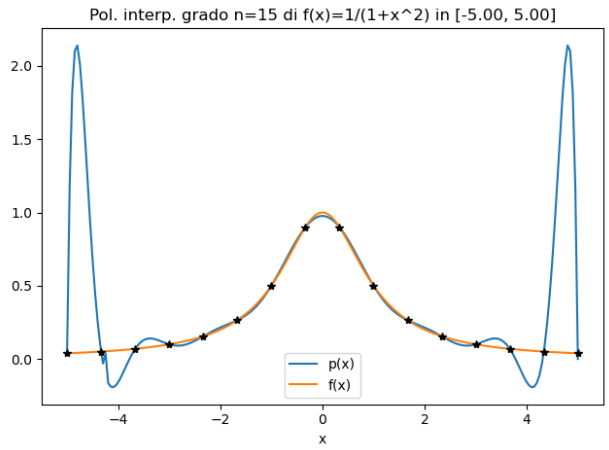


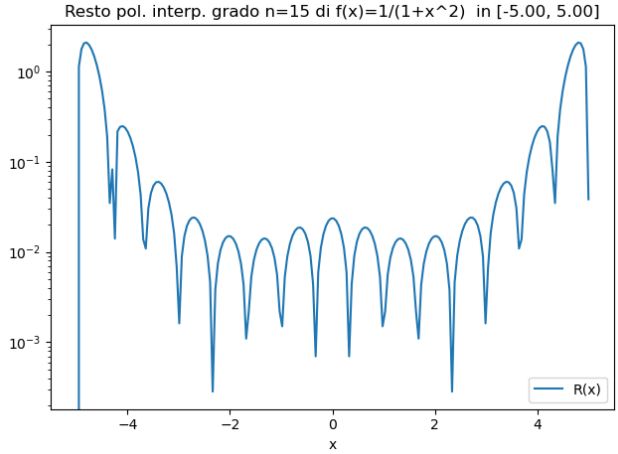


Un risultato di questo tipo potrebbe portarci a pensare che la scelta di un numero elevato di nodi e di un grado elevato per il polinomio di interpolazione siano la scelta corretta per ottenere un'approssimazione migliore.

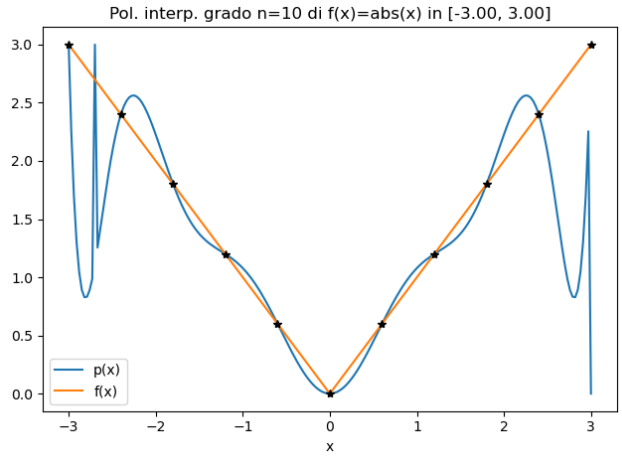
Tuttavia, il problema della convergenza del polinomio di interpolazione alla funzione interpolanda è molto più complesso da affrontare.

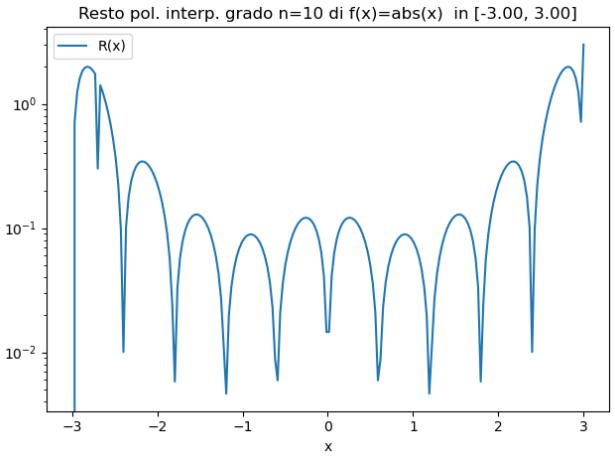
Lo studio della funzione di Runge ci permette di individuare come la soluzione individuata precedentemente non sia appropriata:





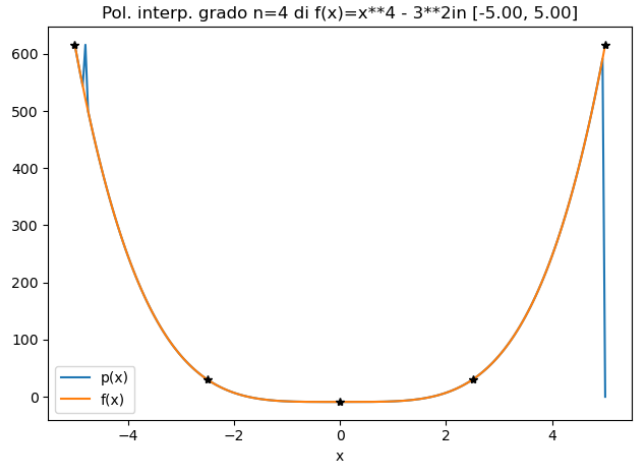
Ora proviamo a calcolare il polinomio con altre funzioni e intervalli.

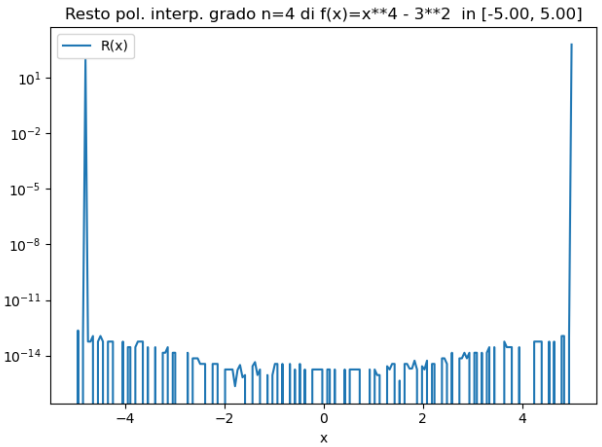




Si evince che nonostante il grado del polinomio ed il numero di nodi utilizzato sia elevato, agli estremi dell'intervallo in cui avviene l'interpolazione sono presenti perturbazioni significative.

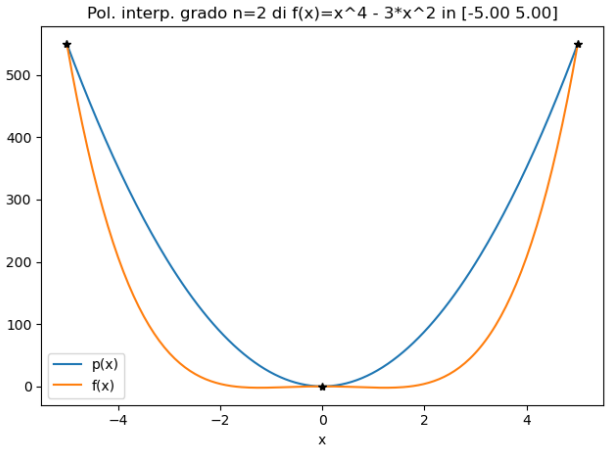
Il prossimo esperimento riguarda invece la trattazione di una funzione polinomiale.

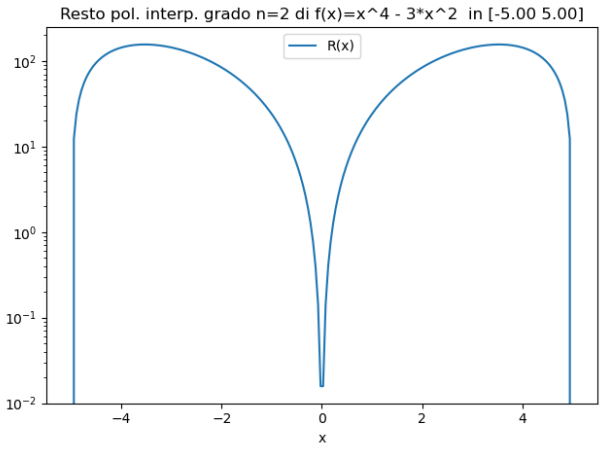




L'esperimento effettuato evidenzia come la scelta di un grado pari al polinomio interpolando si traduca in un resto del polinomio di interpolazione molto vicino allo 0. Inoltre, si può notare che, rispetto alle interpolazioni precedenti, l'approssimazione sia decisamente migliore. Tale risultato è ottenuto proprio perché si effettua una interpolazione polinomiale di una funzione che è anch'essa di classe polinomiale.

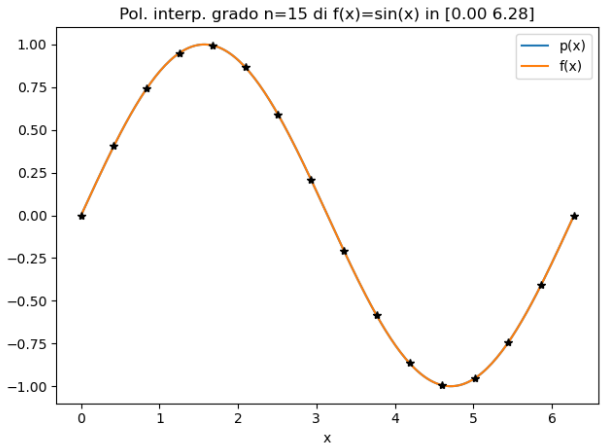
Scegliere invece un grado inferiore produce i seguenti risultati:

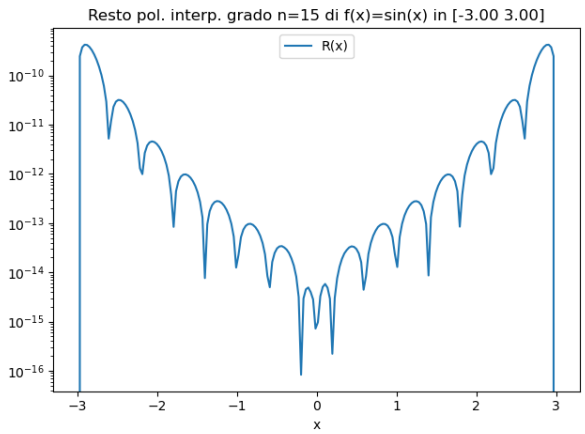


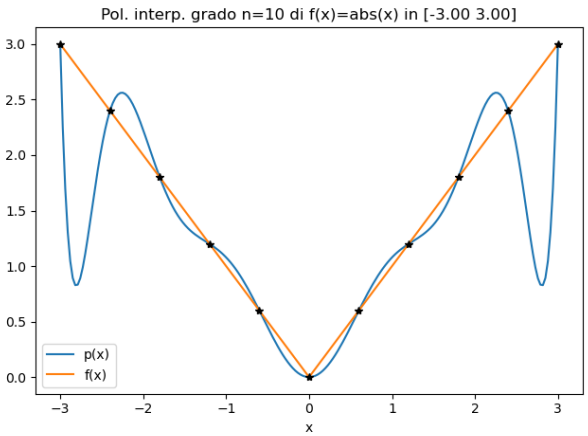


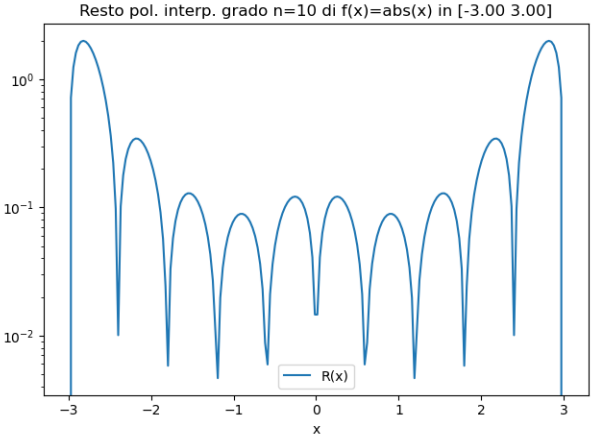
Proviamo ora a calcolare l'interpolazione di alcune funzioni usando i nodi di **Chebyshev**. Li settiamo in questa maniera:

```
k = np.array(range(n,-1,-1))
x_nodi = (a+b)/2 +(b-a)/2*np.cos((2*k+1)*np.pi/2/(n+1))
```









Nella teoria abbiamo individuato che nel resto del polinomio di interpolazione interviene un fattore legato alle derivate di ordine n+1 della funzione interpolanda e fattori legati esclusivamente alla distribuzione dei nodi. Lo studio di quest'ultimo fattore ci ha permesso di analizzare come la scelta dei nodi Chebyshev invece che di nodi equidistanti consenta di migliorare l'approssimazione della funzione.

Approssimazione ai minimi quadrati

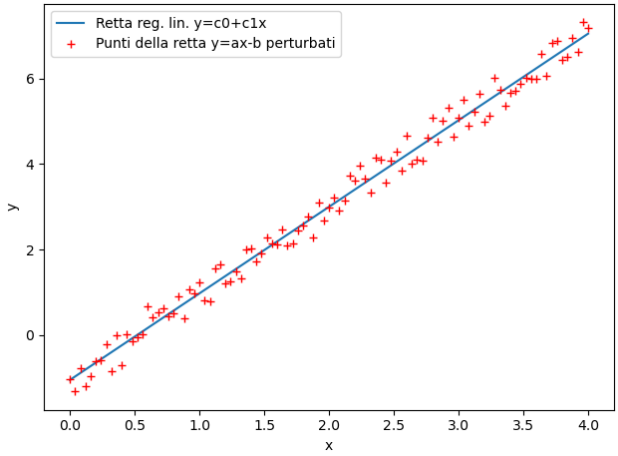
Retta di Regressione Lineare

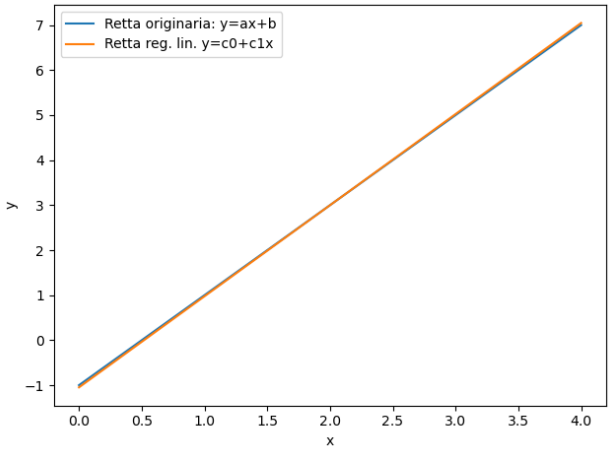
Codice Python

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

# Dati del problema
x0 = 0 ; xm = 4 ; # Intervallo dei valori
m = 100 # Numero di punti

# Nodi
x = np.linspace(x0,xm,m+1)

# Retta di partenza y=ax+b
b = -1 ; a = 2

# Perturbazione dei punti sulla retta
y = (b + a*x) + (np.random.rand(m+1)-0.5)

# Calcolo della retta di regressione lineare
A = np.zeros((2,2))
d = np.zeros(2)
A[0,0] = m+1
A[0,1] = np.sum(x) ; A[1,0] = A[0,1]
A[1,1] = np.sum(x**2)
d[0] = np.sum(y) ; d[1] = np.sum(x*y)
c = np.linalg.solve(A,d)

# Grafico della retta ricostruita
xg = np.linspace(x0,xm,200)
yg = c[0] + c[1]*xg

# Grafico
plt.figure(0)
plt.plot(xg,yg,label='Retta reg. lin. y=c0+c1x')
plt.plot(x,y,'r+', label='Punti della retta y=ax-b perturbati')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc='upper left')
plt.show(block=False)

# Grafico
plt.figure(1)
plt.plot(xg,a*xg+b,label='Retta originaria: y=ax+b')
plt.plot(xg,yg,label='Retta reg. lin. y=c0+c1x')
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.legend(loc='upper left')
plt.show(block=False)
```

Output





Conclusioni

Osserviamo che la retta di regressione lineare ottenuta ricalca quasi la funzione originaria per la quale sono stati perturbati i punti.

Ricerca degli zeri

Metodo di Newton

Codice Python

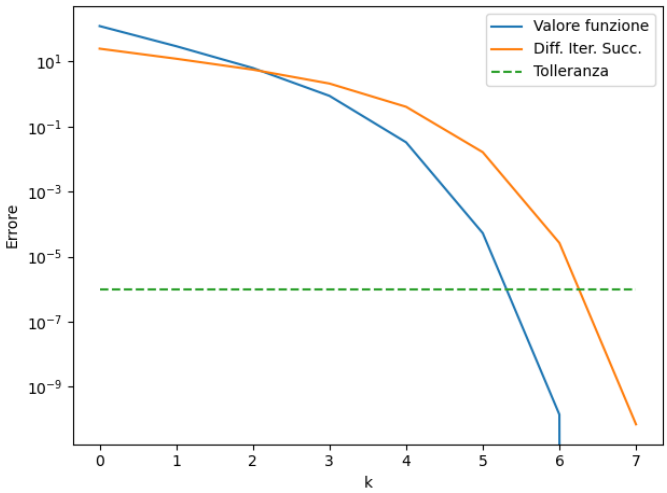
```
import numpy as np
import matplotlib.pyplot as plt

def Newton(x0, tol, kmax):
    #inizializzazione processo calcolo
    fx0 = fun(x0);
    dfx0 = dfun(x0)
    k = 0;
    stop = False
    valore_f = np.zeros(kmax)
    diff_it_succ = np.zeros(kmax)
    print('|-----+--------------+--------------+--------------+--------------+--------------| ')
    print('|  k  |     x0       |    fun(x0)   | Val Funz     | Diff It Succ |     Tol      | ')
    print('|-----+--------------+--------------+--------------+--------------+--------------| ')

    # ciclo di calcolo
    while not (stop) and k < kmax:
```

```
        x1 = x0 - fx0 / dfx0
        fx1 = fun(x1)
        val_fun = abs(fx1) * 5
        iter_succ = abs(x1 - x0) * 5
        stop = (val_fun + iter_succ) < tol
        print('| %3d | %3.10f | %3.10f | %e | %e | %e |'
            % (k, x0, fx0, val_fun, iter_succ, tol))

        valore_f[k] = val_fun
        diff_it_succ[k] = iter_succ

        k = k + 1
        if not (stop):
           x0 = x1;
           fx0 = fx1;
           dfx0 = dfun(x0)

        #verifica convergenza
    if not(stop):
        print('Processo iterativo non converge in %d iterazioni' %kmax)

    return x1, k, valore_f, diff_it_succ


def fun(x):
    y = x**2-1
    return y

def dfun(x):
    y = 2*x
    return y


x0 = 10;
kmax = 100 ; tol = 1.0e-6 ;
xsol, k, val_fun, it_succ = Newton(x0, tol, kmax)
print('La radice ottenuta col metodo di Newton affinche x^2-1 = 0 è x=%f (%f)'
    %(xsol, np.pi/2))

plt.figure(1)
plt.semilogy(range(k),val_fun[0:k],label='Valore funzione')
plt.semilogy(range(k),it_succ[0:k],label='Diff. Iter. Succ.')
plt.semilogy(range(k),np.ones(k)*tol,'--', label='Tolleranza')
plt.legend()
plt.xlabel('k')
plt.ylabel('Errore')
plt.show()
```

Output simulazione 1

```
|-----+--------------+--------------+--------------+--------------+--------------|
|  k  |      x0      |    fun(x0)   | Val Funz     | Diff It Succ |      Tol     |
|-----+--------------+--------------+--------------+--------------+--------------|
|   0 | 10.0000000000 | 99.0000000000 | 1.225125e+02 | 2.475000e+01 | 1.000000e-06 |
|   1 | 5.0500000000 | 24.5025000000 | 2.942714e+01 | 1.212995e+01 | 1.000000e-06 |
|   2 | 2.6240099010 | 5.8854279605 | 6.288328e+00 | 5.607284e+00 | 1.000000e-06 |
|   3 | 1.5025530120 | 1.2576655539 | 8.757512e-01 | 2.092548e+00 | 1.000000e-06 |
|   4 | 1.0840434673 | 0.1751502390 | 3.263158e-02 | 4.039281e-01 | 1.000000e-06 |
|   5 | 1.0032578511 | 0.0065263158 | 5.289578e-05 | 1.626281e-02 | 1.000000e-06 |
|   6 | 1.0000052896 | 0.0000105792 | 1.398970e-10 | 2.644775e-05 | 1.000000e-06 |
```

```
|   7 | 1.0000000000 | 0.0000000000 | 0.000000e+00 | 6.994849e-11 | 1.000000e-06 |
La radice ottenuta col metodo di Newton affinche x^2-1 = 0 è x=1.000000 (1.570796)
```



Conclusioni simulazione 1

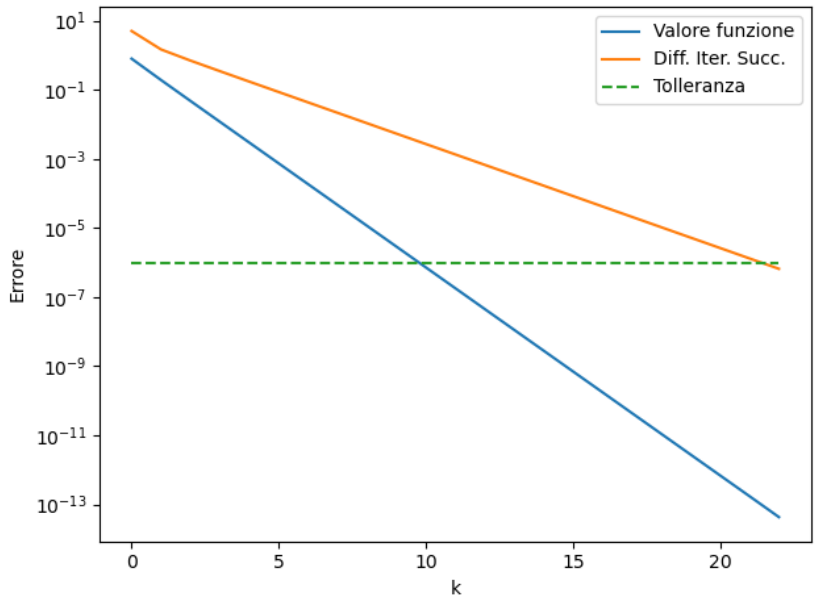
Osserviamo che il metodo iterativo converge in 7 iterazioni, cioè solo quando entrambe le stime dell'errore sono minori della tolleranza (che abbiamo deciso di settare con 1.0e-6).

Output simulazione 2

Mentre se applichiamo il metodo di Newton ad un'altra funzione f(x) = sin(x) + 1 con x0 = π, avremo il seguente output:

Output

```
|-----+--------------+--------------+--------------+--------------+--------------|
|  k  |      x0      |    fun(x0)   | Val Funz     | Diff It Succ |      Tol     |
|-----+--------------+--------------+--------------+--------------+--------------|
|   0 | 3.1400000000 | 1.0015926529 | 7.926416e-01 | 5.007970e+00 | 1.000000e-06 |
|   1 | 4.1415939232 | 0.1585283292 | 1.911297e-01 | 1.467037e+00 | 1.000000e-06 |
|   2 | 4.4350012268 | 0.0382259325 | 4.739543e-02 | 6.979504e-01 | 1.000000e-06 |
|   3 | 4.5745913047 | 0.0094790862 | 1.182536e-02 | 3.450403e-01 | 1.000000e-06 |
|   4 | 4.6435993720 | 0.0023650723 | 2.954882e-03 | 1.720419e-01 | 1.000000e-06 |
|   5 | 4.6780077457 | 0.0005909764 | 7.386296e-04 | 8.596155e-02 | 1.000000e-06 |
|   6 | 4.6952000566 | 0.0001477259 | 1.846517e-04 | 4.297337e-02 | 1.000000e-06 |
|   7 | 4.7037947301 | 0.0000369303 | 4.616257e-05 | 2.148576e-02 | 1.000000e-06 |
|   8 | 4.7080918817 | 0.0000092325 | 1.154062e-05 | 1.074276e-02 | 1.000000e-06 |
|   9 | 4.7102404343 | 0.0000023081 | 2.885154e-06 | 5.371367e-03 | 1.000000e-06 |
|  10 | 4.7113147078 | 0.0000005770 | 7.212884e-07 | 2.685682e-03 | 1.000000e-06 |
|  11 | 4.7118518441 | 0.0000001443 | 1.803221e-07 | 1.342841e-03 | 1.000000e-06 |
|  12 | 4.7121204123 | 0.0000000361 | 4.508052e-08 | 6.714203e-04 | 1.000000e-06 |
|  13 | 4.7122546963 | 0.0000000090 | 1.127013e-08 | 3.357101e-04 | 1.000000e-06 |
|  14 | 4.7123218384 | 0.0000000023 | 2.817532e-09 | 1.678551e-04 | 1.000000e-06 |
|  15 | 4.7123554094 | 0.0000000006 | 7.043832e-10 | 8.392753e-05 | 1.000000e-06 |
|  16 | 4.7123721949 | 0.0000000001 | 1.760958e-10 | 4.196377e-05 | 1.000000e-06 |
|  17 | 4.7123805876 | 0.0000000000 | 4.402367e-11 | 2.098189e-05 | 1.000000e-06 |
|  18 | 4.7123847840 | 0.0000000000 | 1.100620e-11 | 1.049088e-05 | 1.000000e-06 |
|  19 | 4.7123868822 | 0.0000000000 | 2.751688e-12 | 5.245538e-06 | 1.000000e-06 |
|  20 | 4.7123879313 | 0.0000000000 | 6.877832e-13 | 2.622918e-06 | 1.000000e-06 |
|  21 | 4.7123884559 | 0.0000000000 | 1.720846e-13 | 1.311286e-06 | 1.000000e-06 |
|  22 | 4.7123887181 | 0.0000000000 | 4.274359e-14 | 6.561765e-07 | 1.000000e-06 |
La radice ottenuta col metodo di Newton affinche sin(x) + 1 = 0 è x=4.712389 (1.570796)
```



Calcolo del reciproco con il metodo di Newton

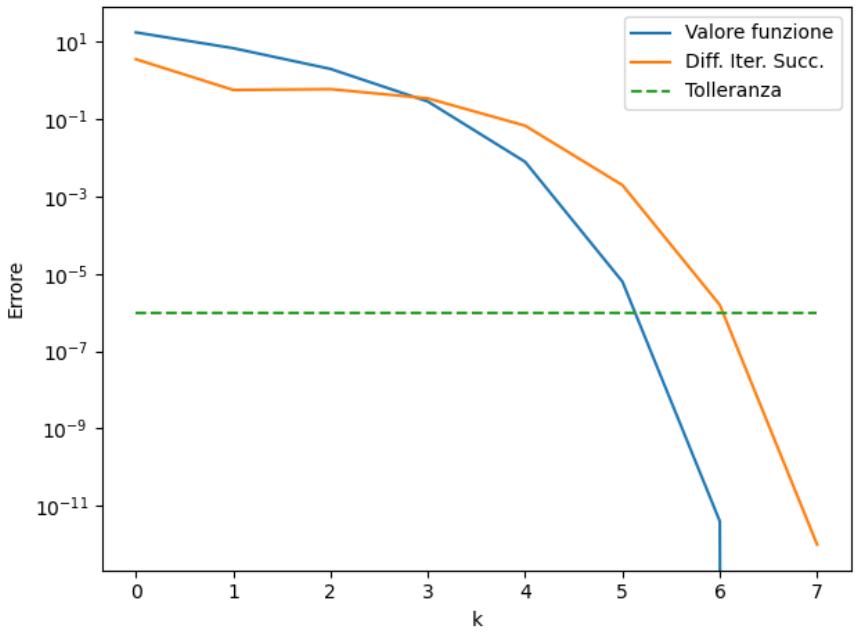
Codice Python

```
def fun(x):
   y = 2 - 1/x
   return y

def dfun(x):
   y = 1/(x**2)
   return y


x0 = 0.9;
kmax = 100 ; tol = 1.0e-6 ;
xsol, k, val_fun, it_succ = Newton(x0, tol, kmax)
print('La radice ottenuta col metodo di Newton partendo da x0=%.2f affinche 2-1x=0 (1/2)è x=%.5f'
%(x0,xsol))
plt.figure(1)
plt.semilogy(range(k),val_fun[0:k],label='Valore funzione')
plt.semilogy(range(k),it_succ[0:k],label='Diff. Iter. Succ.')
plt.semilogy(range(k),np.ones(k)*tol,'--', label='Tolleranza')
plt.legend()
plt.xlabel('k')
plt.ylabel('Errore')
plt.show()
```

Output

```
|-----+--------------+--------------+--------------+--------------+--------------|
|  k  |      x0      |    fun(x0)   | Val Funz     | Diff It Succ |      Tol     |
|-----+--------------+--------------+--------------+--------------+--------------|
|   0 | 0.9000000000 | 0.8888888889 | 1.777778e+01 | 3.600000e+00 | 1.000000e-06 |
|   1 | 0.1800000000 | -3.5555555556 | 6.937669e+00 | 5.760000e-01 | 1.000000e-06 |
|   2 | 0.2952000000 | -1.3875338753 | 2.015940e+00 | 6.045696e-01 | 1.000000e-06 |
|   3 | 0.4161139200 | -0.4031880500 | 2.896273e-01 | 3.490617e-01 | 1.000000e-06 |
```

```
|    4 | 0.4859262512 | -0.0579254519 | 7.929098e-03 | 6.838804e-02 | 1.000000e-06 |
|    5 | 0.4996038592 | -0.0015858197 | 6.277106e-06 | 1.979135e-03 | 1.000000e-06 |
|    6 | 0.4999996861 | -0.0000012554 | 3.941292e-12 | 1.569274e-06 | 1.000000e-06 |
|    7 | 0.5000000000 | -0.0000000000 | 0.000000e+00 | 9.850454e-13 | 1.000000e-06 |
La radice ottenuta col metodo di Newton partendo da x0=0.90 affinche 2-1x=0 (1/2)è
x=0.50000
```



Considerazioni

Questo codice permette di calcolare il valore di $\frac{1}{\alpha}$ con $\alpha = 2$.

Dunque si applica il metodo a $f(x) = \alpha - \frac{1}{x}$ con x0 = 0.9.

Sappiamo che si ha convergenza locale quando $|1 - \alpha\, x0| < 1$ e poiché $\alpha = 2$, avremo $0 < x0 < 1$.

Difatti se prendiamo x0 = 1.0, avremo come output:

```
ZeroDivisionError: float division by zero
```

Formule di quadratura

Formula del Trapezio

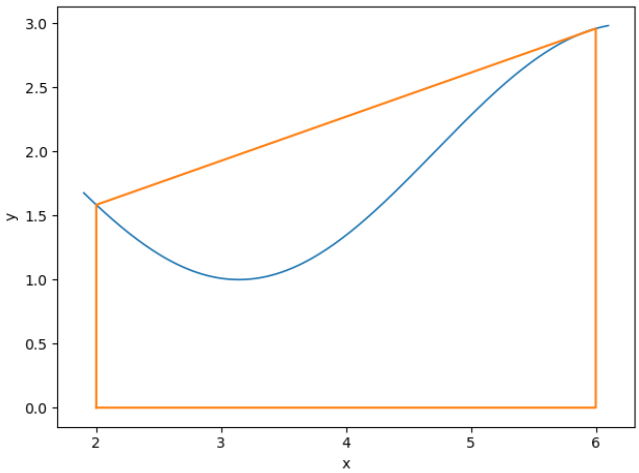
Codice Python

```
def f(x):
   y = np.cos(x) + 2
   return y
```

```
def F(x):
   y = np.sin(x) + 2*x
   return y

#Calcolo formula del Trapezio
def Trapezi(f,a,b):
   #formula di quadratura
   T = (f(a) + f(b))*(b-a)/2
   return T

# Intervallo di integrazione
a = 2.0 ; b = 6.0

# Calcolo valore esatto integrale
I = F(b) - F(a)

# Calcolo integrale ed errore
T = Trapezi(f,a,b)

E = abs(I - T)
print('Integrale esatto : %f' % I)
print('Formula del Trapezio : %f' % T)
print('Errore commesso : %e' % E)

# Rappresentazione grafica
x = np.linspace(a-0.1,b+0.1,200)
y = f(x)
plt.figure(1)
plt.plot(x,y,label='f(x)',linewidth=1.2)
xx = np.array([a,a,b,b,a])
yy = np.array([0,f(a),f(b),0,0])
plt.plot(xx,yy)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

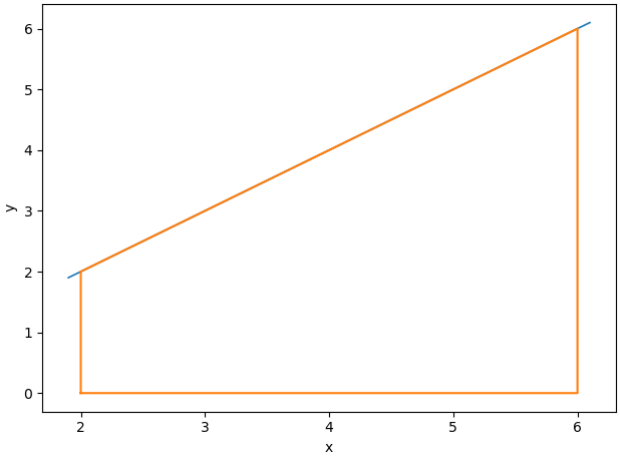
Output 1

```
Integrale esatto : 6.811287
Formula del Trapezio : 9.088047
Errore commesso : 2.276760e+00
```



Output 2

```
Integrale esatto : 16.000000
Formula del Trapezio : 16.000000
Errore commesso : 0.000000e+00
```



Considerazioni

Gli output delle simulazioni mostrano due situazioni differenti. Nella prima prova si è approssimata l'area di una funzione goniometrica, mentre nella seconda un polinomio di primo grado.

Sappiamo che la formula del trapezio approssima l'area di una funzione con un polinomio di primo grado: questo dimostra perché nella prima simulazione l'errore è grande, avendo approssimato una curva con una retta (polinomio di primo grado), mentre nel secondo caso l'errore è inesistente perché l'area calcolata è giusta in quanto anche la funzione è un polinomio di primo grado e, come si vede dal grafico, è esattamente la base superiore del trapezio che approssima l'area della nostra funzione.

Formula di Simpson

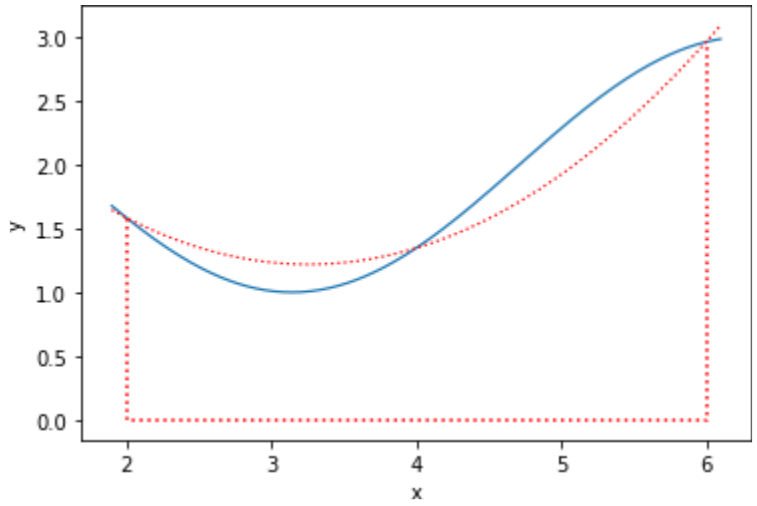
Codice Python

```
import numpy as np
import matplotlib.pyplot as plt
from interpolazione.lagrange_baricentrica import Lagrange

# Definizione funzione da integrare
def f(x):
   #y = -6*x**2 + 16*x - 4
   y = np.cos(x)+2
   return y

def F(x):
   #y = -2*x**3 + 8*x**2 - 4*x + 2
   y = np.sin(x)+2*x
   return y

# Calcolo formula semplice di Simpson
def FormulaSimpson(f,a,b):
```

```
    # formula di quadratura
    S = (f(a) + 4*f((a+b)/2) + f(b))*(b-a)/6
    return S

# Intervallo di integrazione
a = 2.0 ; b = 6

# Calcolo valore esatto integrale
I = F(b) - F(a)

# Calcolo integrale ed errore
S = FormulaSimpson(f,a,b)
E = abs(I - S)

print('\nFUNZIONE DA INTEGRARE: f(x) = cos(x)+2 nell intevallo [%.2f, %.2f]' %(a,b) )
print('Integrale esatto : %f' % I)
print('Formula di Simpson : %f' % S)
print('Errore commesso : %e' % E)

x = np.linspace(a-0.1,b+0.1,200)
y = f(x)

#Calcolo del polinomio di secondo grado sui nodi a,(a+b)/2, b usato da Simpson per
# approssimare la f(x) e calcolare l'integrale
x_nodi = [a,(a+b)/2,b]
y_nodi = [f(a),f((a+b)/2),f(b)]
p = Lagrange(x_nodi,y_nodi,x)

# Rappresentazione grafica
plt.figure(1)
plt.plot(x,y,label='f(x)',linewidth=1.8)
plt.plot(x,p, 'r--', label='Appross.\nSimpson')
xx = np.array([b,b,a,a])
yy = np.array([f(b),0,0,f(a)])
plt.plot(xx,yy, 'r--')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc='upper left')
plt.show()
```

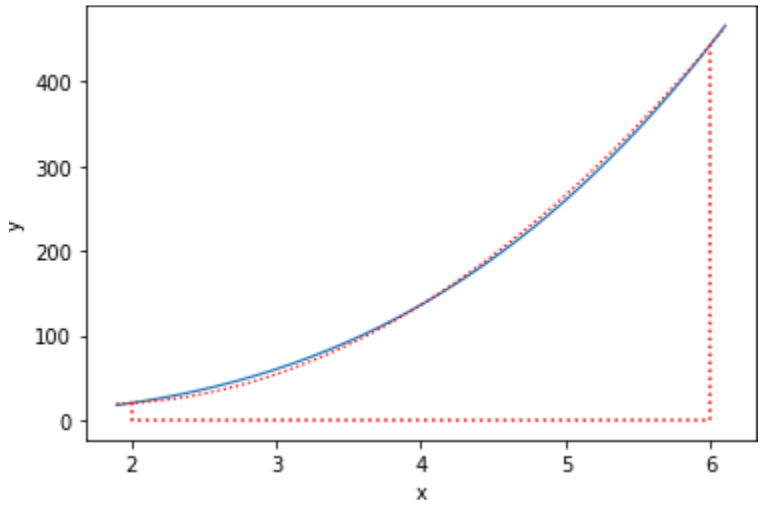
Output 1

```
Integrale esatto : 6.811287
Formula di Simpson : 6.619633
Errore commesso : 1.916544e-01
```



Output 2

```
Integrale esatto : 672.000000
Formula di Simpson : 672.000000
Errore commesso : 0.000000e+00
```



Considerazioni

L'utilizzo della formula di Simpson prevede l'utilizzo di un polinomio di secondo grado per approssimare la funzione. Possiamo notare come questo tipo di approssimazione si traduca in una differenza nettamente inferiore, e quindi in un errore di calcolo dell'integrale inferiore, rispetto all'utilizzo della formula del trapezio.

Il secondo output ottenuto è relativo al calcolo dell'integrale di un polinomio di terzo grado: il risultato ottenuto verifica che il grado di precisione della formula di Simpson sia tre, infatti l'errore ottenuto è nullo.

Formula composta del Trapezio

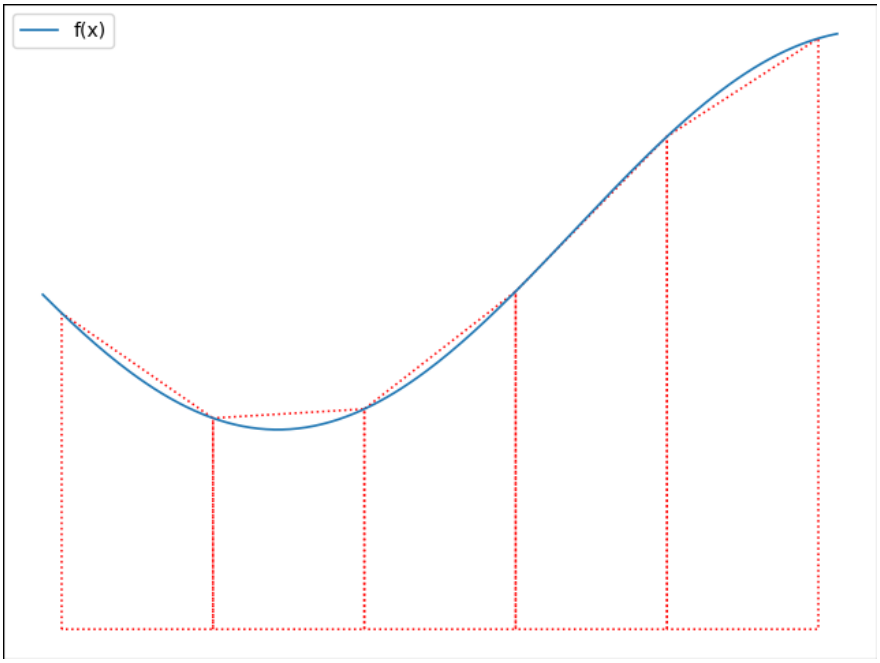
Codice Python

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

# Definizione funzione da integrare
def f(x):
   y = np.cos(x) + 2
   return y

def F(x):
   y = np.sin(x) + 2*x
   return y

# Calcolo formula composta Trapezi
def TrapeziComposti(f,a,b,N):
   # Calcolo estremi sottointervallo
   z = np.linspace(a,b,N+1)
   # Calcolo formula di quadratura
   fz = f(z)
   S = 0
   for i in range(1,N):
      S = S + fz[i]
   T = (fz[0] + 2*S + fz[N])*(b-a)/2/N

   # Rappresentazione grafica dei trapezi
   for i in range(0,N):
      x = np.array([z[i],z[i],z[i+1],z[i+1],z[i]])
      y = np.array([0,fz[i],fz[i+1],0,0])
      plt.plot(x,y,'r:',linewidth=1.1)

   return T

# Intervallo di integrazione
a = 2.0 ; b = 6.0

# Calcolo valore esatto integrale
I = F(b) - F(a)

# Inizializzazione grafica
fig = plt.figure(1)
ax = fig.add_axes([0,0,1,1])

# Calcolo integrale ed errore
N = 5
T = TrapeziComposti(f,a,b,N)
E = abs(I - T)

print('Integrale essatto : %f' % I)
print('Formula composta del Trapezio : %f' % T)
print('Errore commesso : %e' % E)
# Rappresentazione grafica
x = np.linspace(a-0.1,b+0.1,200)
y = f(x)
plt.plot(x,y,label='f(x)',linewidth=1.2)
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.legend()
plt.show()
```

Output

```
Integrale essatto : 6.811287
Formula composta del Trapezio : 6.875372
Errore commesso : 6.408474e-02
```



Formula composta di Simpson

Codice Python

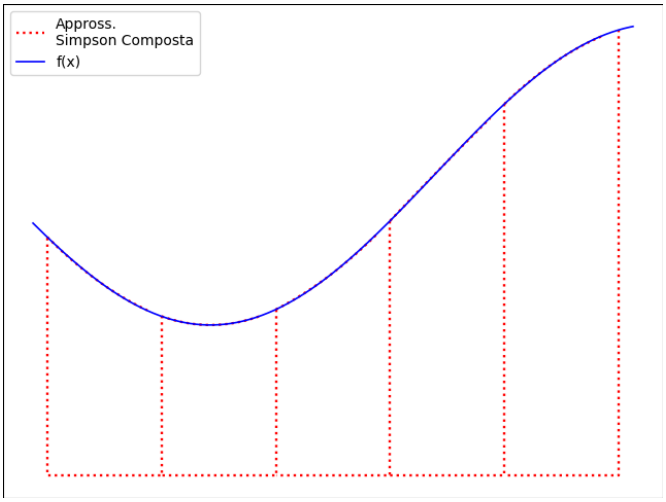
```
import numpy as np
import matplotlib.pyplot as plt
from interpolazione.lagrange_baricentrica import Lagrange

# Definizione funzione da integrare
def f(x):
    #y = -6*x**2 + 16*x - 4
    y = np.cos(x)+2
    return y

def F(x):
    #y = -2*x**3 + 8*x**2 - 4*x + 2
    y = np.sin(x)+2*x
    return y

# Calcolo formula composta Simpson
def SimpsonComposta(f,a,b,N):
    # Calcolo estremi sottointervallo
```

```
    z = np.linspace(a,b,N+1)
    # Calcolo formula di quadratura
    fz = f(z)
    S1 = 0
    for i in range(1,N):
        S1 = S1 + fz[i]

    S2 = 0
    for i in range(0, N):
        med = (z[i] + z[i + 1]) / 2
        S2 = S2 + f(med)

    S = (fz[0] + 2 * S1 + 4 * S2 + fz[N]) * (b - a) / 6 / N

    # Rappresentazione grafica di Simpson
    x_punti = np.linspace(z[0],z[1]-0.1,200)
    x_nodi = np.array([z[0], (z[0] + z[1]) / 2, z[1]])
    y_nodi = np.array([fz[0], f((z[0] + z[1]) / 2), fz[1]])
    p = Lagrange(x_nodi, y_nodi, x_punti)
    x = np.array([z[1], z[1], z[0], z[0]])
    y = np.array([fz[1], 0, 0, fz[0]])
    plt.plot(x_punti, p, 'r:', label='Appross.\nSimpson Composta', linewidth=1.7)
    plt.plot(x, y, 'r:', linewidth=1.7)

    for i in range(1, N):
        x_punti = np.linspace(z[i], z[i + 1] - 0.1, 200)
        x_nodi = [z[i], (z[i] + z[i + 1]) / 2, z[i + 1]]
        y_nodi = [fz[i], f((z[i] + z[i + 1]) / 2), fz[i + 1]]
        p = Lagrange(x_nodi, y_nodi, x_punti)
        x = np.array([z[i + 1], z[i + 1], z[i]])
        y = np.array([fz[i + 1], 0, 0])
        plt.plot(x_punti, p, 'r:', linewidth=1.7)
        plt.plot(x, y, 'r:', linewidth=1.7)

    return S


# Intervallo di integrazione
a = 2 ; b = 6

# Calcolo valore esatto integrale
I = F(b) - F(a)

# Inizializzazione grafica
fig = plt.figure(1)
ax = fig.add_axes([0,0,1,1])
# Calcolo integrale ed errore
N = 5
S = SimpsonComposta(f,a,b,N)
E = abs(I - S)
print('Integrale essatto : %f' % I)
print('Formula composta di Simpson : %f' % S)
print('Errore commesso : %e' % E)
# Rappresentazione grafica
x = np.linspace(a-0.1,b+0.1,200)
y = f(x)
plt.plot(x,y,'b-',label='f(x)',linewidth=1.2)
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.legend(loc='upper left')
plt.show()
```

Output

```
Integrale essatto : 6.811287
Formula composta di Simpson : 6.811115
Errore commesso : 1.723366e-04
```



Considerazioni

Le formule composte del Trapezio e di Simpson si dimostrano, negli esperimenti effettuati, più precise delle corrispondenti formule non composte. Inoltre possiamo osservare che anche nel caso delle formule composte, la formula di Simpson sia più precisa della formula del Trapezio.

Formula composta del Trapezio: variazione dell'errore al crescere dei sotto intervalli

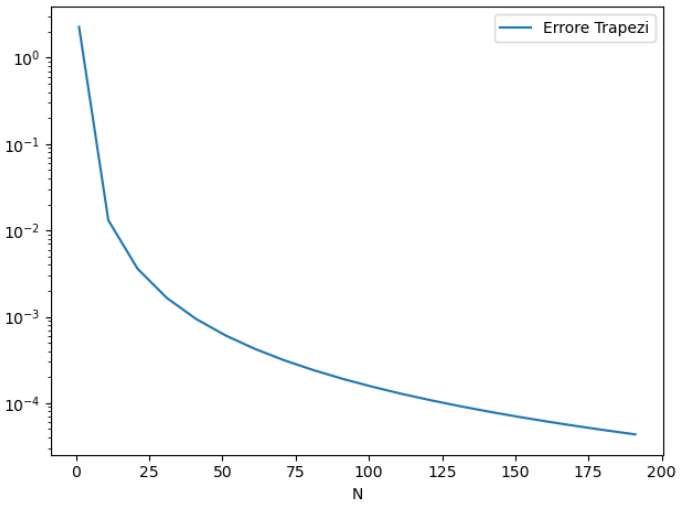
Codice Python

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

# Definizione funzione da integrare
def f(x):
   y = np.cos(x) + 2
   return y

def F(x):
   y = np.sin(x) + 2*x
   return y

# Calcolo formula composta Trapezi
def TrapeziComposti(f,a,b,N):
   #Calcolo estremi sottointervallo
   z = np.linspace(a,b,N+1)
```

```
    # Calcolo formula di quadratura
    fz = f(z)
    S = 0
    for i in range(1,N):
        S = S + fz[i]
    T= (fz[0] + 2*S + fz[N])*(b-a)/2/N
    return T

# Intervallo di integrazione
a = 2.0 ; b = 6.0
# Calcolo valore esatto integrale
I = F(b) - F(a)
# Calcolo integrale ed errore
print('Integrale esatto : %f' % I)
N_range = range(1,200,10)
Errore = np.zeros(len(N_range))
for i in range(len(N_range)):
    N = N_range[i]
    T = TrapeziComposti(f,a,b,N)
    Errore[i] = abs(I - T)
    print('N=%d Errore=%e' %(N,Errore[i]))
plt.figure(1)
plt.semilogy(N_range,Errore,label='Errore Trapezi')
plt.xlabel('N')
plt.legend()
plt.show()
```

Output

```
Integrale esatto : 6.811287
N=1 Errore=2.276760e+00
N=11 Errore=1.312772e-02
N=21 Errore=3.596167e-03
N=31 Errore=1.649730e-03
N=41 Errore=9.430113e-04
N=51 Errore=6.094245e-04
N=61 Errore=4.259780e-04
N=71 Errore=3.144286e-04
N=81 Errore=2.415813e-04
N=91 Errore=1.914022e-04
N=101 Errore=1.553761e-04
N=111 Errore=1.286409e-04
N=121 Errore=1.082562e-04
N=131 Errore=9.235914e-05
N=141 Errore=7.972294e-05
N=151 Errore=6.951314e-05
N=161 Errore=6.114605e-05
N=171 Errore=5.420352e-05
N=181 Errore=4.837958e-05
N=191 Errore=4.344624e-05
```



Formula composta di Simpson: variazione dell'errore al crescere dei sotto intervalli

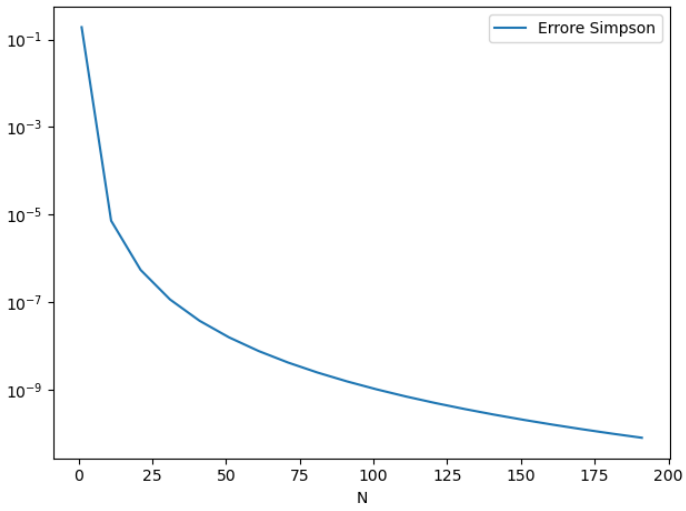
Codice Python

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

# Definizione funzione da integrare
def f(x):
   #y = -6*x**2 + 16*x - 4
   y = np.cos(x)+2
   return y

def F(x):
   #y = -2*x**3 + 8*x**2 - 4*x + 2
   y = np.sin(x)+2*x
   return y

# Calcolo formula composta Simpson
def SimpsonComposta(f,a,b,N):
   # Calcolo estremi sottointervallo
   z = np.linspace(a,b,N+1)
   # Calcolo formula di quadratura
   fz = f(z)
   S1 = 0
   for i in range(1,N):
      S1 = S1 + fz[i]

   S2 = 0
   for i in range(0, N):
      med = (z[i] + z[i + 1]) / 2
      S2 = S2 + f(med)

   S = (fz[0] + 2 * S1 + 4 * S2 + fz[N]) * (b - a) / 6 / N
   return S
```

```
# Intervallo di integrazione
a = 2.0 ; b = 6.0
# Calcolo valore esatto integrale
I = F(b) - F(a)
# Calcolo integrale ed errore
print('Integrale esatto : %f' % I)
N_range = range(1,200,10)
Errore = np.zeros(len(N_range))

for i in range(len(N_range)):
   N = N_range[i]
   T = SimpsonComposta(f,a,b,N)
   Errore[i] = abs(I - T)
   print('N=%d Errore=%e' %(N,Errore[i]))

plt.figure(1)
plt.semilogy(N_range,Errore,label='Errore Simpson')
plt.xlabel('N')
plt.legend()
plt.show()
```

Output

```
Integrale esatto : 6.811287
N=1 Errore=1.916544e-01
N=11 Errore=7.245451e-06
N=21 Errore=5.438967e-07
N=31 Errore=1.144703e-07
N=41 Errore=3.740349e-08
N=51 Errore=1.562152e-08
N=61 Errore=7.632393e-09
N=71 Errore=4.158453e-09
N=81 Errore=2.454798e-09
N=91 Errore=1.540932e-09
N=101 Errore=1.015453e-09
N=111 Errore=6.960645e-10
N=121 Errore=4.929435e-10
N=131 Errore=3.588010e-10
N=141 Errore=2.673382e-10
N=151 Errore=2.032472e-10
N=161 Errore=1.572653e-10
N=171 Errore=1.235794e-10
N=181 Errore=9.845191e-11
N=191 Errore=7.939605e-11
```


Errore Simpson
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
0
25
50
75
100
125
150
175
200
N