Università degli Studi di Bari “Aldo Moro”
Corso di Laurea in Informatica e Tecnologie per la Produzione del Software

Calcolo numerico

Implementazione degli algoritmi e studio di opportuni casi di test

Leonardo Saccotelli
14/04/2020

Indice

Aritmetica del calcolatore

Algebra lineare

Interpolazione polinomiale

Radici di funzione

Quadratura numerica

TEST SULLO STUDIO DEL CONDIZIONAMENTO DELLA SOMMA

Implementazione

```
"""
Created on Thu Oct  5 14:12:52 2017

@author: Leonardo Saccotelli
"""
print(' ------------------------------------------------------')
print('| STUDYING OF CONDITIONING OF THE ADDITIONAL OPERATION |')
print(' ------------------------------------------------------')
from random import random

#Primo numero inserito
first_number = 0
#Secondo numero inserito
second_number = 0
#Ordine di grandezza della perturbazione
orderPerturbation = 0

#-----------------------------------------------
#                ACQUISIZIONE DEI DATI IN INPUT
#-----------------------------------------------

first_number = float(input(' - Enter a first number: '))
second_number = float(input(' - Enter a second number: '))
orderPerturbation = float(input(' - Enter a order for perturbation: '))

#-----------------------------------------------
#                PERTURBAZIONE DEI DATI
#-----------------------------------------------
#Generazione della perturbazione sul primo input
deltaFirstNumber = ((random()-0.5)*2)*orderPerturbation
#Generazione della perturbazione sul secondo input
deltaSecondNumber = ((random()-0.5)*2)*orderPerturbation
#Variabile utilizzato per memorizzare il primo numero che è stato
perturbato
firstDisturbedNumber = first_number + deltaFirstNumber
#Variabile utilizzato per memorizzare il secondo numero che è stato
perturbato
secondDisturbedNumber = second_number + deltaSecondNumber

#-------------------------------------------------
#          CALCOLO DELL'OPERAZIONE SOMMA
#-------------------------------------------------

#Risultato della somma sui dati non perturbati
correct_sum = first_number + second_number
#Risultato della somma sui dati perturbati
sum_perturbed = firstDisturbedNumber + secondDisturbedNumber

#-------------------------------------------------
```

```
#   CALCOLO DELL'ERRORE RELATIVO
#-------------------------------------------------
#Errore relativo sul primo valore in input
error_firstNumber = abs(first_number - firstDisturbedNumber) /
abs(first_number)
#Errore relativo sul secondo valore in input
error_secondNumber = abs(second_number - secondDisturbedNumber) /
abs(second_number)
#Errore relativo che occorre nel calcolo della somma
error_sum = abs(correct_sum - sum_perturbed) / abs(correct_sum)
#-------------------------------------------------
#   STAMPA DEI RISULTATI
#-------------------------------------------------
print('  ------------------------------------------------------')
print('\n - CORRECT VALUE ENTERED BY USER\n ')
print('    First number = %34.16f \n    Second number= %34.16f' %
(first_number,second_number) )
print('  ------------------------------------------------------')
print('\n - PERTURBED VALUES\n ')
print('    First number = %34.16f \n    Second number= %34.16f' %
(firstDisturbedNumber,secondDisturbedNumber) )
print('  ------------------------------------------------------')
print('\n - CORRECT SUM\n ')
print('    First+Second = %34.16f' %correct_sum)
print('  ------------------------------------------------------')
print('\n - PERTURBED SUM\n ')
print('    First+Second = %34.16f' %sum_perturbed)
print('  ------------------------------------------------------')
print('\n - ERROR ANALYSIS\n ')
print('    Error on the first number : %15e' %error_firstNumber)
print('    Error on the second number: %15e' %error_secondNumber)
print('    Error on the sum: %25e' %error_sum)
```

- **ESEMPIO 1**: Problema BEN CONDIZIONATO, l'ordine di grandezza sui dati in input è lo stesso ordine di grandezza sui dati in output. Pertanto, il risultato non sarà molto perturbato.

```
 ------------------------------------------------------
| STUDYING OF CONDITIONING OF THE ADDITIONAL OPERATION |
 ------------------------------------------------------

- Enter a first number: 15.012

- Enter a second number: 17.12

- Enter a order for perturbation: 0.000000001
  ----------------------------------------------------

- CORRECT VALUE ENTERED BY USER

    First number =                 15.012000000000005
    Second number=                 17.120000000000010
  ----------------------------------------------------

- PERTURBED VALUES

    First number =                 15.011999997150657
    Second number=                 17.119999990937322
  ----------------------------------------------------

- CORRECT SUM

    First+Second =                 32.132000000000050
  ----------------------------------------------------

- PERTURBED SUM

    First+Second =                 32.131999988087943
  ----------------------------------------------------

- ERROR ANALYSIS

    Error on the first number :     1.898047e-11
    Error on the second number:     5.293626e-11
    Error on the sum:               3.707241e-11
```

- **ESEMPIO 2**: Problema MAL CONDIZIONATO, l'ordine di grandezza sui dati in input NON è lo stesso ordine di grandezza sui dati in output. Pertanto, il risultato sarà molto perturbato. Il mal condizionamento è dovuto al fatto che stiamo sottraendo quantità molto vicine, pertanto i coefficienti di amplificazione saranno molto grandi e quindi gli errori sugli input saranno molto perturbati.

```
 ---------------------------------------------------------
| STUDYING OF CONDITIONING OF THE ADDITIONAL OPERATION |
 ---------------------------------------------------------

 - Enter a first number: -23.45

 - Enter a second number: 23.55

 - Enter a order for perturbation: 0.00001
  ---------------------------------------------------------

 - CORRECT VALUE ENTERED BY USER

    First number =                -23.4499999999999993
    Second number=                 23.5500000000000007
  ---------------------------------------------------------

 - PERTURBED VALUES

    First number =                -23.4500072146540006
    Second number=                 23.5499913916869019
  ---------------------------------------------------------

 - CORRECT SUM

    First+Second =                  0.1000000000000014
  ---------------------------------------------------------

 - PERTURBED SUM

    First+Second =                  0.0999841770329013
  ---------------------------------------------------------

 - ERROR ANALYSIS

    Error on the first number :     3.076612e-07
    Error on the second number:     3.655335e-07
    Error on the sum:               1.582297e-04
```

TEST SULLO STUDIO DEL CONDIZIONAMENTO DEL PRODOTTO

Implementazione

```
"""
Created on Thu Oct  5 16:19:11 2017

@author: Leonardo Saccotelli
"""
from random import random

print(' ---------------------------------------------------------')
print(' ---------------------------------------------------------')
print('|   STUDYING OF CONDITIONING OF THE PRODUCT OPERATION   |')
print(' ---------------------------------------------------------')

#Primo numero inserito
first_number = 0

#Secondo numero inserito
second_number = 0

#Perturbazione inseita in input dall'utente
orderPerturbation = 0

#-----------------------------------------------
#   INSERIMENTO DEI DATI
#-----------------------------------------------

first_number = float(input(' - Enter a first number: '))

second_number = float(input(' - Enter a second number: '))

orderPerturbation = float(input(' - Enter a order for perturbation: '))
#-----------------------------------------------
#   PERTURBIAMO I DAII
#-----------------------------------------------

#Generazione della perturbazione sul primo input
deltaFirstNumber = ((random()-0.5)*2)*orderPerturbation

#Generazione della perturbazione sul secondo input
deltaSecondNumber = ((random()-0.5)*2)*orderPerturbation

#Variabile utilizzato per memorizzare il primo numero che è stato
perturbato
firstDisturbedNumber = first_number + deltaFirstNumber

#Variabile utilizzato per memorizzare il secondo numero che è stato
perturbato
secondDisturbedNumber = second_number + deltaSecondNumber
#-----------------------------------------------
#   ESEGUIAMO L'OPERAZIONE
```

```
#-------------------------------------------------

#Risultato del prodotto sui dati non perturbati
correct_product = first_number * second_number

#Risultato del prodotto sui dati perturbati
product_perturbed = firstDisturbedNumber * secondDisturbedNumber

#-------------------------------------------------
#          CALCOLO DELL'ERRORE RELATIVO
#-------------------------------------------------
#Errore relativo sul primo valore in input
error_firstNumber = abs(first_number - firstDisturbedNumber) /
abs(first_number)


#Errore relativo sul secondo valore in input
error_secondNumber = abs(second_number - secondDisturbedNumber) /
abs(second_number)

#Errore relativo sull'output
error_product = abs(correct_product - product_perturbed) /
abs(correct_product)

#-------------------------------------------------
#            PRINT THE RESULT
#-------------------------------------------------
print('  -----------------------------------------------------')
print('\n - CORRECT VALUE ENTEREDBY USER\n ')
print('    First number = %34.16f \n    Second number= %34.16f' %
(first_number,second_number) )
print('  -----------------------------------------------------')
print('\n - PERTURBED VALUES\n ')
print('    First number = %34.16f \n    Second number= %34.16f' %
(firstDisturbedNumber,secondDisturbedNumber) )
print('  -----------------------------------------------------')
print('\n - CORRECT PRODUCT\n ')
print('    First x Second = %34.16f' %correct_product)
print('  -----------------------------------------------------')
print('\n - PERTURBED PRODUCT\n ')
print('    First x Second = %34.16f' %product_perturbed)
print('  -----------------------------------------------------')
print('\n - ERROR ANALYSIS\n ')
print('    Error on the first number : %20e' %error_firstNumber)
print('    Error on the second number: %20e' %error_secondNumber)
print('    Relative error on the product: %17e' %error_product)
```

- **Esempio 1:** Problema BEN CONDIZIONATO, l'ordine di grandezza dell'errore relativo sui dati in input e sui dati in output è lo stesso.
  Ricordiamo che $Er(p) <\approx Er(x) + Er(y)$, quando $Er(x)\ e\ Er(y)$ sono piccoli.

```
-------------------------------------------------------
|   STUDYING OF CONDITIONING OF THE PRODUCT OPERATION   |
-------------------------------------------------------

- Enter a first number: 15

- Enter a second number: 12.52

- Enter a order for perturbation: 0.0000000000001
 -----------------------------------------------------

- CORRECT VALUE ENTEREDBY USER

   First number =                15.000000000000000
   Second number=                12.519999999999996
 -----------------------------------------------------

- PERTURBED VALUES

   First number =                15.000000000000817
   Second number=                12.520000000000440
 -----------------------------------------------------

- CORRECT PRODUCT

   First x Second =               187.79999999999829
 -----------------------------------------------------

- PERTURBED PRODUCT

   First x Second =               187.80000000000016882
 -----------------------------------------------------

- ERROR ANALYSIS

   Error on the first number :          5.447494e-15
   Error on the second number:          3.547038e-15
   Relative error on the product:       9.080418e-15
```

TEST SULLO STUDIO DEL CONDIZIONAMENTO NEL CALCOLO DELLA FUNZIONE

Siano:

- $f(x)$: una generica funzione
- x : input non perturbato
- $\mathbb{x}$: input perturbato

Vogliamo misurare l'errore relativo commesso nel calcolare $f(\mathbb{x})$ invece di $f(x)$:

$$Er\big(f(x)\big) \approx Er(x) * \left| x * \frac{f'(x)}{f(x)} \right|$$

La quantità $K\big(f(x)\big) = \left| x * \frac{f'(x)}{f(x)} \right|$ è detta indice di condizionamento del problema ed è la quantità che amplifica o meno l'errore relativo sull'input.

Negli esempi che seguiranno vogliamo studiare il condizionamento di alcune funzioni abbastanza regolari, cercando di capire se il problema è ben condizionato o meno.

In particolare, per ogni funzione, mostreremo:

- Il grafico di $k\big(f(x)\big)$, in modo tale da capire per quali valori di x l'indice è grande
- Una tabella che mette a confronto l'errore relativo sull'input con l'errore relativo sull'output
- Il grafico costruito a partire dall'errore relativo sull'input e sull'output.

Implementazione

```
"""
Created on Thu Oct  5 16:56:42 2017
@author: Leonardo Saccotelli
"""
import numpy as np
import math
import matplotlib.pyplot as plt

print(' ---------------------------------------------------------')
print('|  STUDYING OF CONDITIONING OF FUNCTION: F(x) = sin(x)   |')
print(' ---------------------------------------------------------')

perturbation = 0

perturbation = float(input('  Enter a perturbation: '))

#Selezioniamo 50 punti equidistanti nell'intervallo [Pi,4Pi]
x = np.linspace(np.pi, 4*np.pi,100)

#Calcoliamo il valore di sin(x) in ogni punto
y = np.sin(x)

errorX = np.zeros(len(x))

#Perturbo tutti i punti x nell'intervallo scelto
for i in range(len(y)):
    errorX[i] = x[i] + perturbation

#Calcolo il seno in ogni punto di x peturbato
errorY = np.sin(errorX)

relativeErrorX = np.zeros(len(errorX))

#Calcolo l'errore relativo commesso sul dato x
for i in range(len(y)):
    relativeErrorX[i] = abs(x[i]- errorX[i]) / abs(x[i])

relativeErrorY = np.zeros(len(relativeErrorX))

#Calcolo dell'errore relativo commesso nel calcolo del seno in punti di
x perturbati
for i in range(len(y)):
    relativeErrorY[i] = relativeErrorX[i] * abs(x[i]/math.tan(x[i]))

#Stampo a video una tabella per confrontare gli errori su input e output
print('\n ---------------------------------------------------------')

strErrorX = 'Error_X'
strErrorY = 'Error_Y'

print('%25s %25s' %(strErrorX,strErrorY))
print(' ---------------------------------------------------------')

for i in range(len(y)):
```

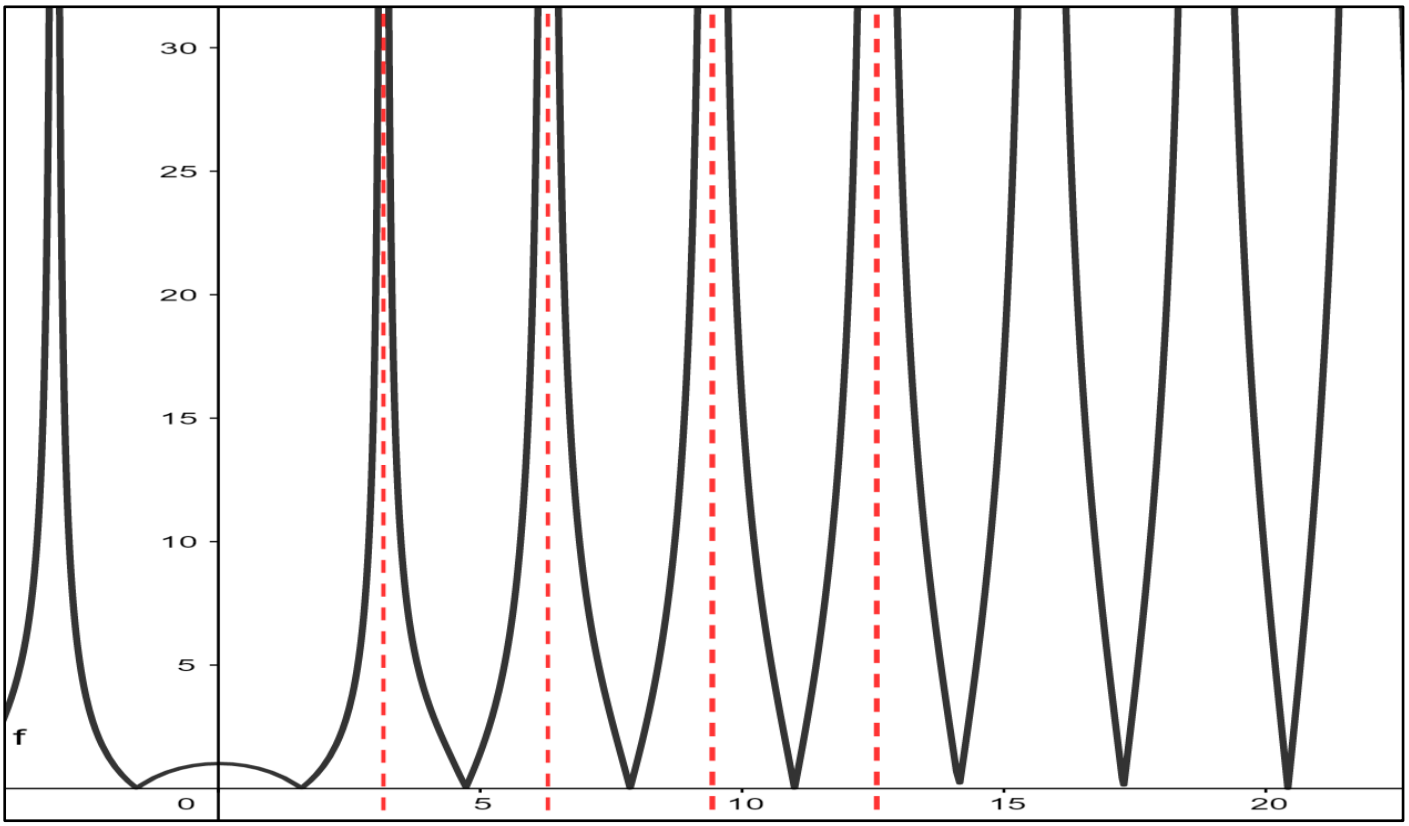
```
    print('%25e %25e' %(relativeErrorX[i],relativeErrorY[i]))

#stampa dei risultati e visualizzazione grafico
plt.loglog(relativeErrorX,relativeErrorY,':k')
plt.xlabel('error_X')
plt.ylabel('error_Sin')
plt.title('Relative error on the function f(x)=sin(x)')

plt.show()
```

Esempio 1: $f(x) = \sin x$

- Grafico di $k\big(f(x)\big) = \left|\frac{x}{\tan x}\right|$

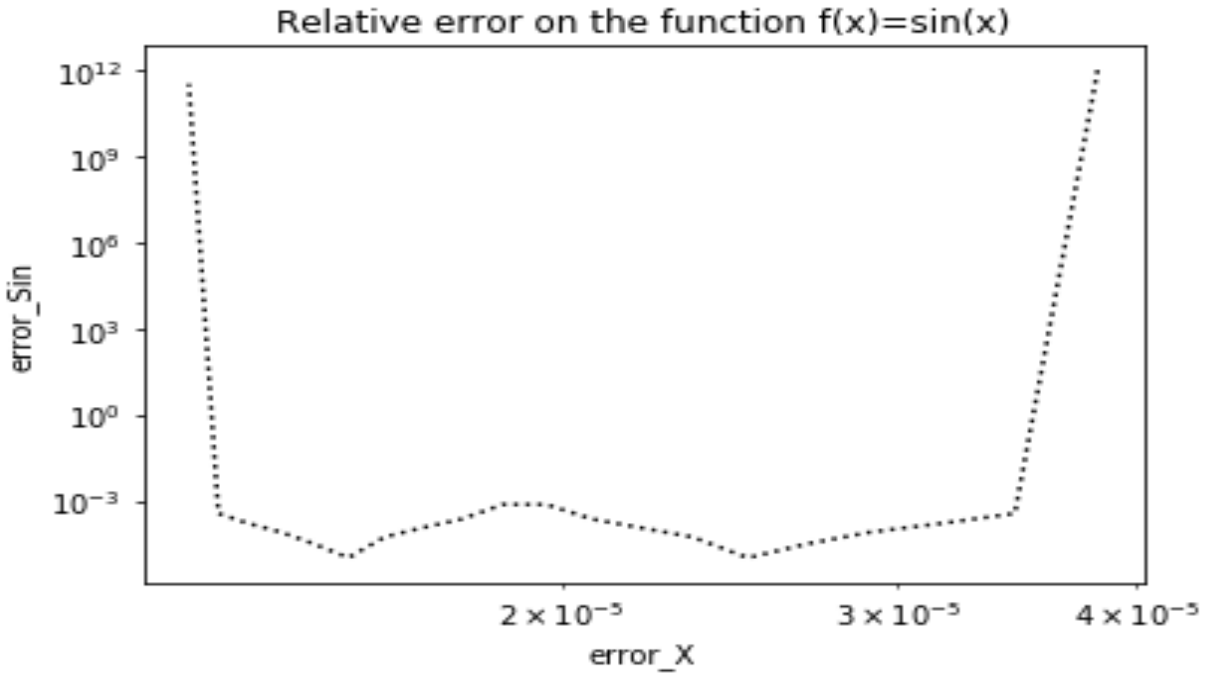
  - Analizzando il grafico dell'indice di condizionamento del problema è facile rendersi conto che nell'intorno di $\pi, 2\pi, \dots, k\big(f(x)\big)$ è molto grande; nell'intorno di $\frac{\pi}{2}, \frac{3\pi}{2}, \dots, k\big(f(x)\big)$ è piccolo.

  - Mi aspetto che il problema sarà:
    - BEN CONDIZIONATO nell'intorno di $x = \frac{k\pi}{2}, k \in \mathbb{Z}$
    - MAL CONDIZIONATO nell'intorno di $x = k\pi, k \in \mathbb{Z}$



- Calcolo della funzione $\sin x$ sull'intervallo $[\pi, 3\pi]$ e studio dell'errore commesso.

```
-----------------------------------------------------------
|  STUDYING OF CONDITIONING OF FUNCTION: F(x) = sin(x)   |
-----------------------------------------------------------

 Enter a perturbation: 0.00012

-----------------------------------------------------------
                  Error_X                    Error_Y
-----------------------------------------------------------
             3.819719e-05               9.798744e+11 ★
             3.455936e-05               3.495481e-04
             3.155420e-05               1.541760e-04
             2.902986e-05               7.839991e-05
             2.687950e-05               3.038813e-05
             2.502574e-05               9.943483e-06
             2.341118e-05               5.263688e-05
             2.199232e-05               1.104678e-04
             2.073562e-05               2.217406e-04
             1.961477e-05               7.191206e-04
             1.860889e-05               7.191206e-04
             1.770114e-05               2.217406e-04
             1.687783e-05               1.104678e-04
             1.612770e-05               5.263688e-05
             1.544142e-05               9.943483e-06
             1.481115e-05               3.038813e-05
             1.423032e-05               7.839991e-05
             1.369333e-05               1.541760e-04
             1.319539e-05               3.495481e-04
             1.273240e-05               3.266248e+11 ★
```

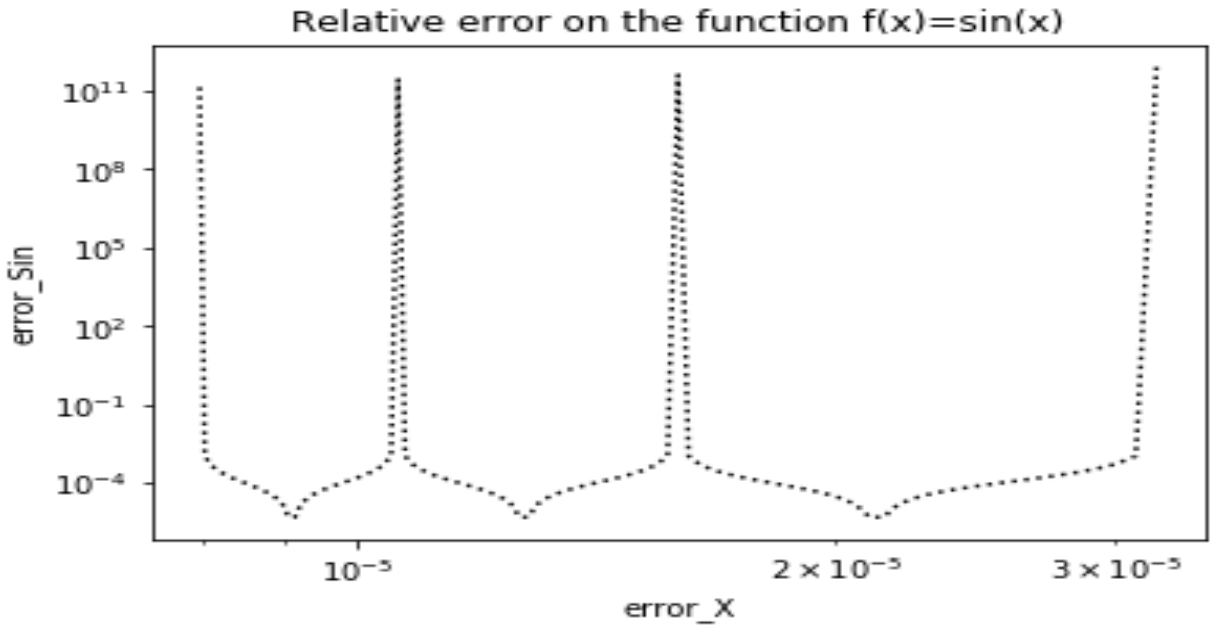


COMMENTO SUL GRAFICO

- Gli errori su input e output segnati con ★ fanno riferimento a punti x in cui il calcolo della funzione $\sin x$ risulta mal condizionata
- Nell'intervallo $[\pi, 3\pi]$ mi aspetto un' amplificazione dell'errore sull'output così elevata in prossimità di π e 3π.
- Negli altri punti dell'intervallo $[\pi, 3\pi]$ $Er\big(f(x)\big) \cong Er(x)$, quindi è ben condizionato.

In questo secondo test, calcoliamo la funzione $\sin x$ in cento diversi punti appartenenti all'intervallo $[\pi, 4\pi]$ e di nuovo troviamo in quali punti dell'intervallo il calcolo del seno risulta un problema mal condizionato.

Notiamo come ci sono punti nell'intervallo in cui ad un errore relativo sull'input dell'ordine di 10^{-5} corrisponde un errore relativo sull'output dell'ordine di 10^{11}.

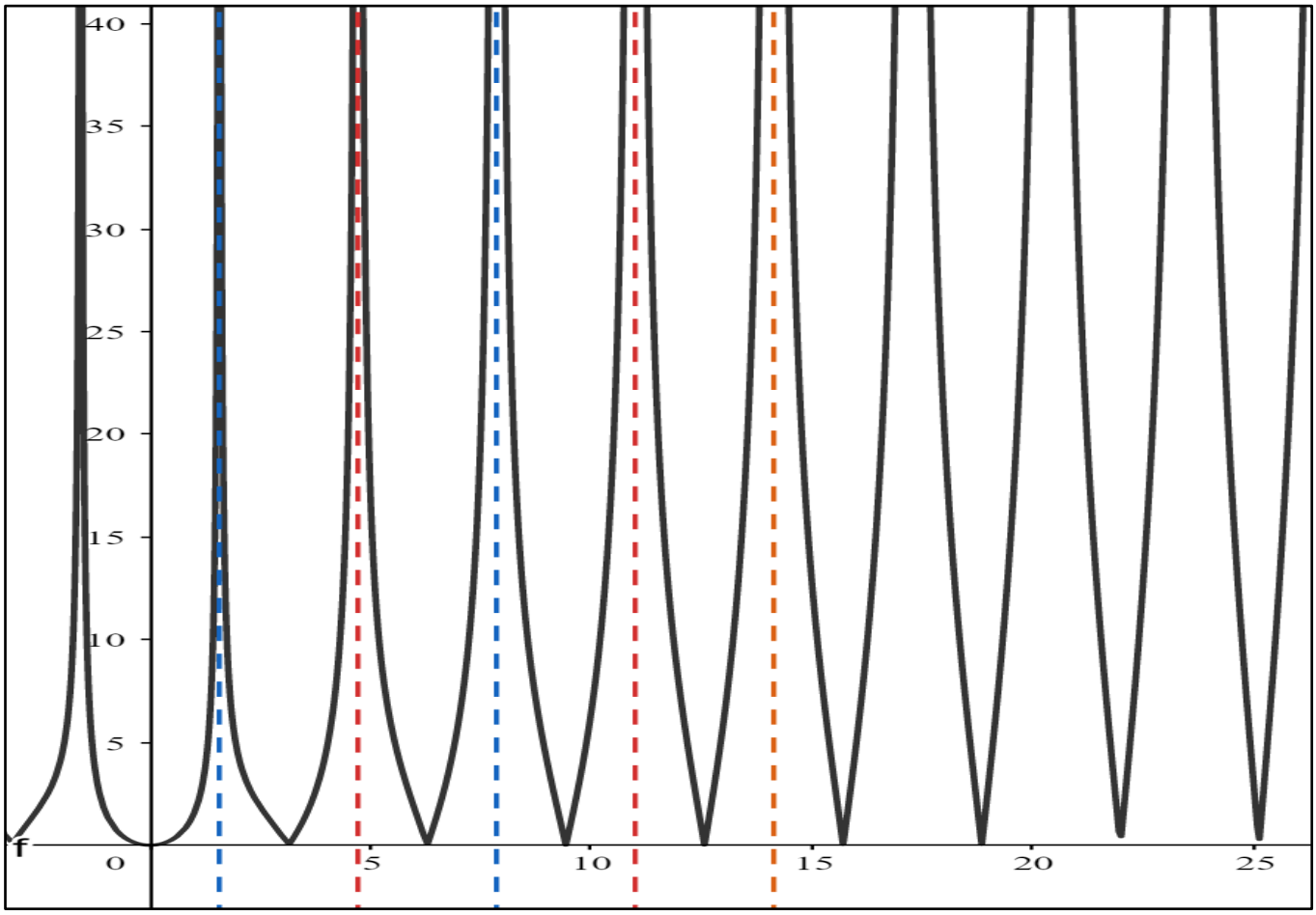


Esempio 2: $f(x) = \cos x$

- Grafico di $k\big(f(x)\big) = |-x \tan x|$

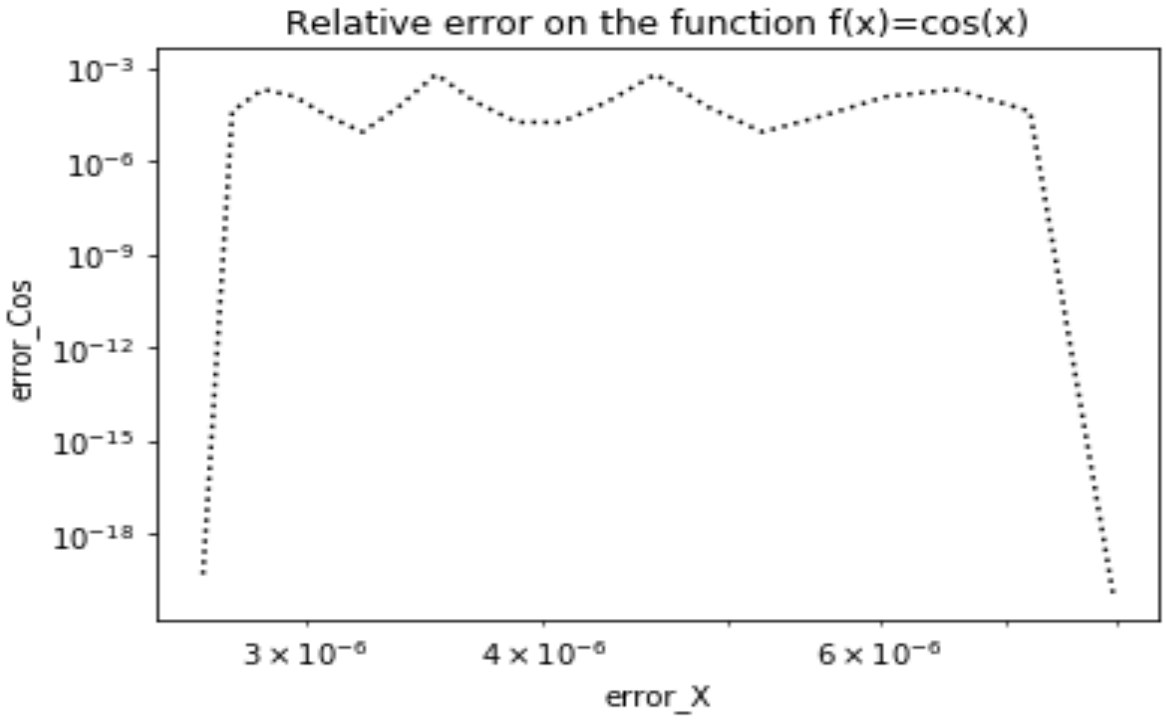
  - Analizzando il grafico dell'indice di condizionamento del problema è facile rendersi conto che nell'intorno di $\pi, 2\pi, \dots, k\big(f(x)\big)$ è piccolo; nell'intorno di $\frac{\pi}{2}, \frac{3\pi}{2}, \dots$ $k\big(f(x)\big)$ è molto grande.

  - Mi aspetto che il problema sarà:
    - BEN CONDIZIONATO nell'intorno di $x = k\pi, k \in \mathbb{Z}$
    - MAL CONDIZIONATO nell'intorno di $x = \frac{k\pi}{2}, k \in \mathbb{Z}$



- Calcolo della funzione $\cos x$ sull'intervallo $[2\pi, 6\pi]$ e studio dell'errore commesso.

```
-------------------------------------------------------
|  STUDYING OF CONDITIONING OF FUNCTION: F(x) = cos(x)  |
-------------------------------------------------------

 Enter a perturbation: 0.00005

-------------------------------------------------------
                  Error_X                   Error_Y
-------------------------------------------------------
             7.957747e-06              1.224647e-20
             7.199866e-06              3.891656e-05
             6.573791e-06              1.974455e-04
             6.047888e-06              1.139885e-04
             5.599896e-06              2.705865e-05
             5.213696e-06              8.343524e-06
             4.877329e-06              5.431448e-05
             4.581733e-06              6.034103e-04
             4.319920e-06              7.653070e-05
             4.086411e-06              1.716502e-05
             3.876851e-06              1.716502e-05
             3.687736e-06              7.653070e-05
             3.516214e-06              6.034103e-04
             3.359938e-06              5.431448e-05
             3.216962e-06              8.343524e-06
             3.085657e-06              2.705865e-05
             2.964651e-06              1.139885e-04
             2.852777e-06              1.974455e-04
             2.749040e-06              3.891656e-05
             2.652582e-06              3.673940e-20
```

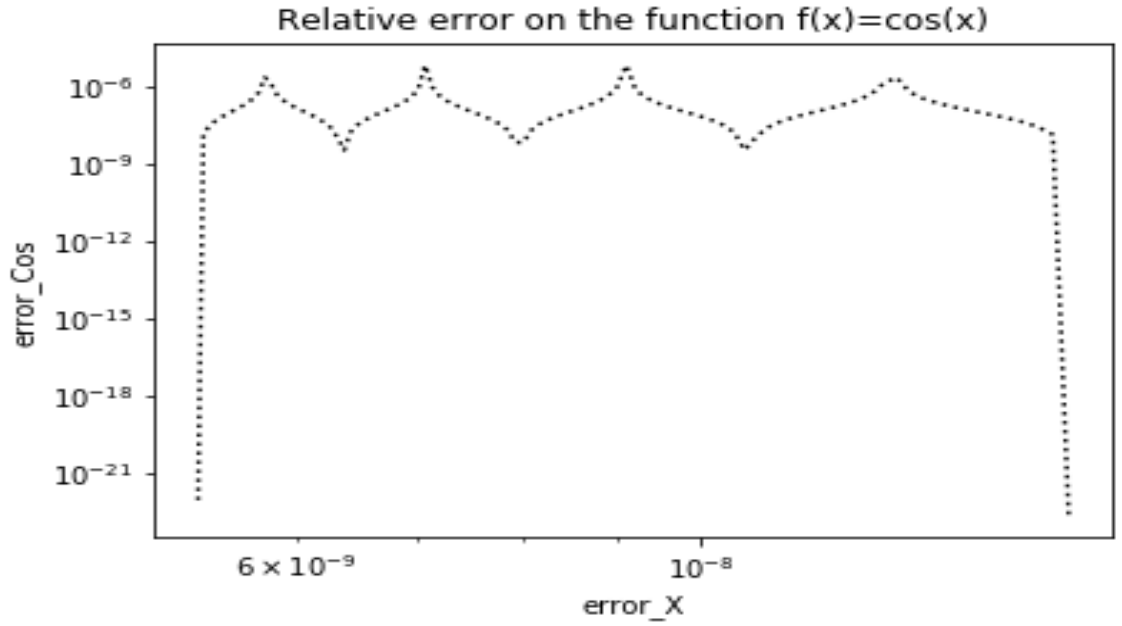


COMMENTO SUL GRAFICO

- Nei punti $x = 2\pi$ e $x = 6\pi$, $Er\big(f(x)\big) \ll Er(x)$. Infatti, si passa da un errore dell'ordine di 10^{-6} a un errore dell'ordine di 10^{-20}.
  Nell'intervallo $[2\pi, 6\pi]$ ci sono poi punti in cui $Er\big(f(x)\big) \cong Er(x)$, in particolare gli errori sono dell'ordine di 10^{-6}.
  In entrambi i casi il calcolo della funzione $\cos x$ risulta essere un problema ben condizionato.

- Ci sono punti in cui $Er\big(f(x)\big) > Er(x)$. Infatti, da un errore sull'input dell'ordine di 10^{-6} si passa ad un errore dell'ordine di 10^{-5} e di 10^{-4}.
  Proprio in quest'ultimo caso il problema risulta mal condizionato. Questo si verifica perché stiamo calcolando $\cos x$ nell'intorno di $x = \frac{k\pi}{2}, k \in \mathbb{Z}$.

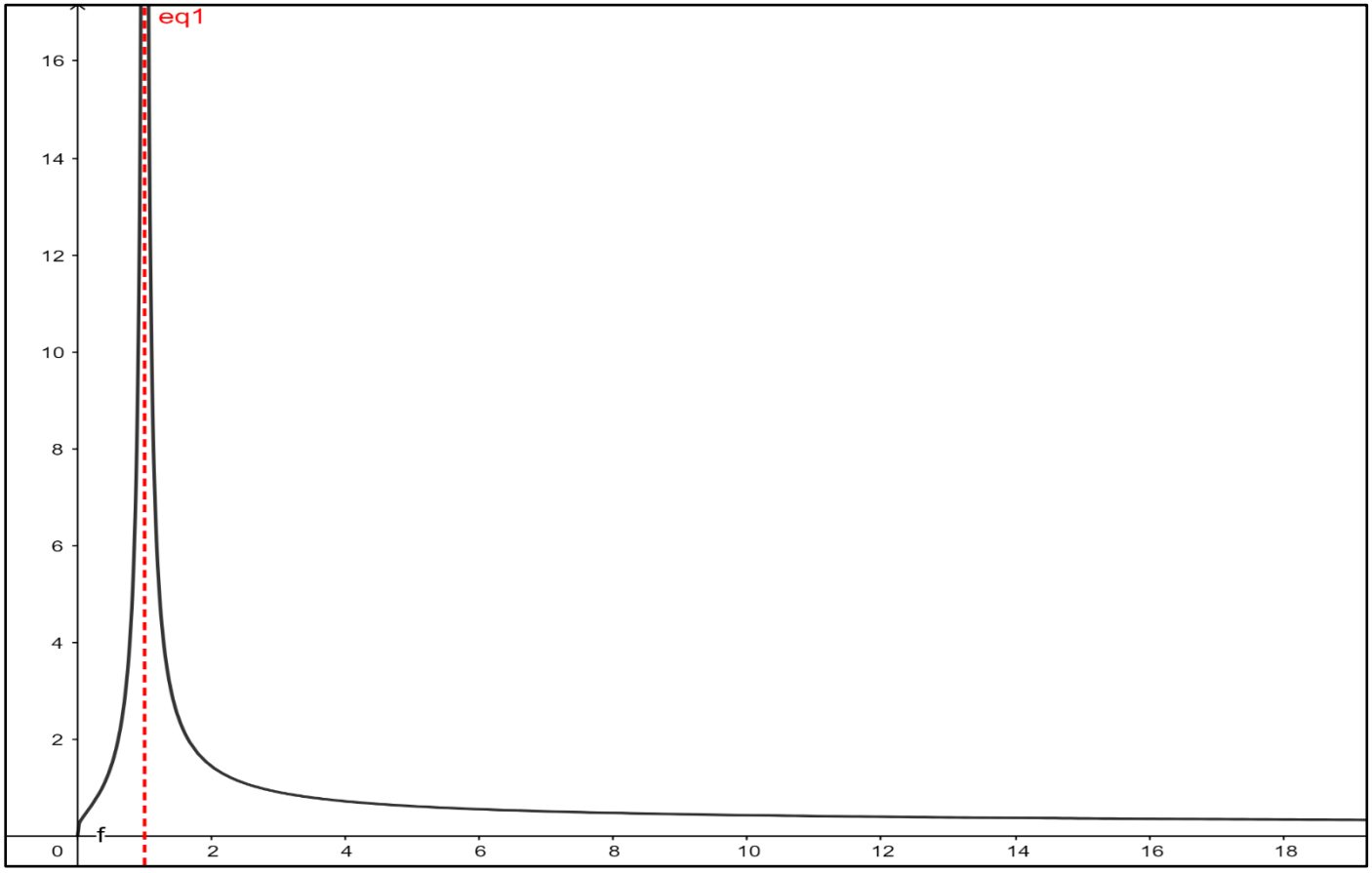
In questo secondo test, calcoliamo la funzione $\cos x$ in cento diversi punti appartenenti all'intervallo $[2\pi, 6\pi]$.



Esempio 3: $f(x) = \log x$

- Grafico di $k\big(f(x)\big) = \left|\frac{1}{\log x}\right|$
  - Analizzando il grafico dell'indice di condizionamento del problema è facile rendersi conto che nell'intorno di 1, $k\big(f(x)\big)$ è molto grande;
  - Il problema sarà:
    - BEN CONDIZIONATO $\forall x \in (0, +\infty), x \neq 1$

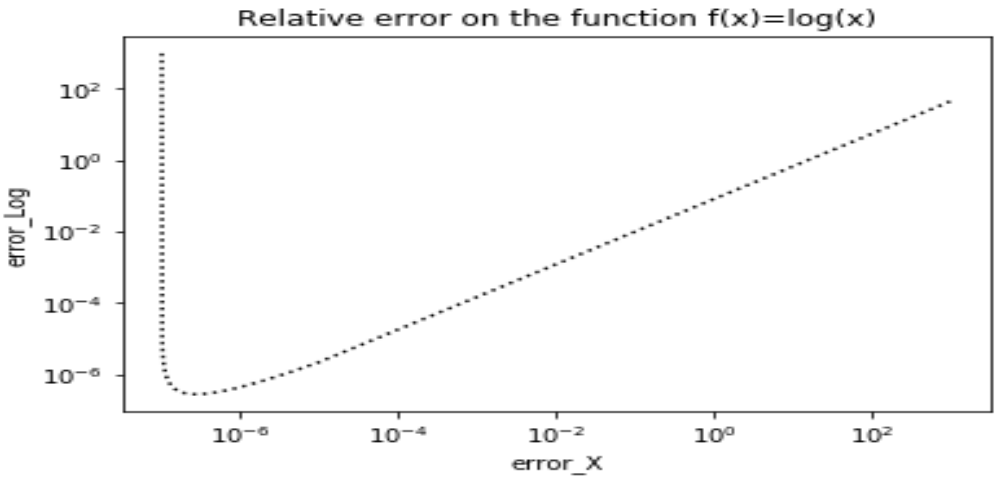
      Notiamo come $x = 1$ non è un punto del dominio, quindi il problema sarà ben condizionato ovunque purché lontano da $x = 1$
    - MAL CONDIZIONATO nell'intorno di $x = 1$



- Calcolo della funzione $\log x$ sull'intervallo e studio dell'errore commesso. Vogliamo mostrare che nell'intorno destro e sinistro di $x = 1$, il problema è fortemente mal condizionato. Altrove il problema è ben condizionato.

    - Prendo 100 punti nell'intorno sinistro di 1, mostriamo uno stralcio degli errori.

```
 ------------------------------------------------------
|  STUDYING OF CONDITIONING OF FUNCTION: F(x) = log(x)  |
 ------------------------------------------------------

 Enter a perturbation: 0.0000001

 ------------------------------------------------------
                 Error_X                  Error_Y
 ------------------------------------------------------
           1.000000e+03             4.342945e+01
           9.900000e-06             2.154460e-06
           4.950000e-06             1.268589e-06
           3.300000e-06             9.437989e-07
           2.475000e-06             7.713103e-07
           1.980000e-06             6.631651e-07
           1.650000e-06             5.885793e-07
           1.414286e-06             5.338519e-07
           1.237500e-06             4.919150e-07
           1.100000e-06             4.587356e-07
           1.164706e-07             7.638989e-07
           1.151163e-07             8.177466e-07
           1.137931e-07             8.806716e-07
           1.125000e-07             9.551460e-07
           1.112360e-07             1.044631e-06
           1.100000e-07             1.154126e-06
           1.087912e-07             1.291132e-06
           1.076087e-07             1.467433e-06
           1.064516e-07             1.702671e-06
           1.053191e-07             2.032205e-06
           1.042105e-07             2.526747e-06
           1.031250e-07             3.351298e-06
           1.020619e-07             5.000857e-06
           1.010204e-07             9.950425e-06
           1.000000e-07             9.999999e+02
```



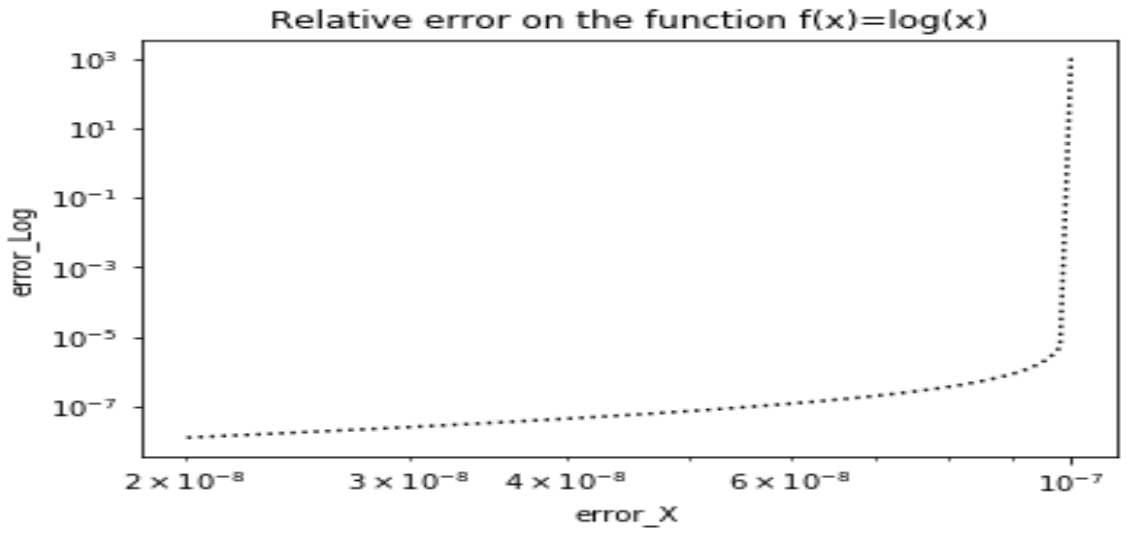
COMMENTO SUL GRAFICO

- L'intervallo dal quale abbiamo estratto i 100 punti è $[1 * 10^{-10}, k\,]$ con $k \approx 1$.

- Notiamo che $Er\big(f(x)\big) \cong Er(x)$ in quasi ogni punto dell'intervallo.
  Quindi come abbiamo già detto il problema risulta ben condizionato.

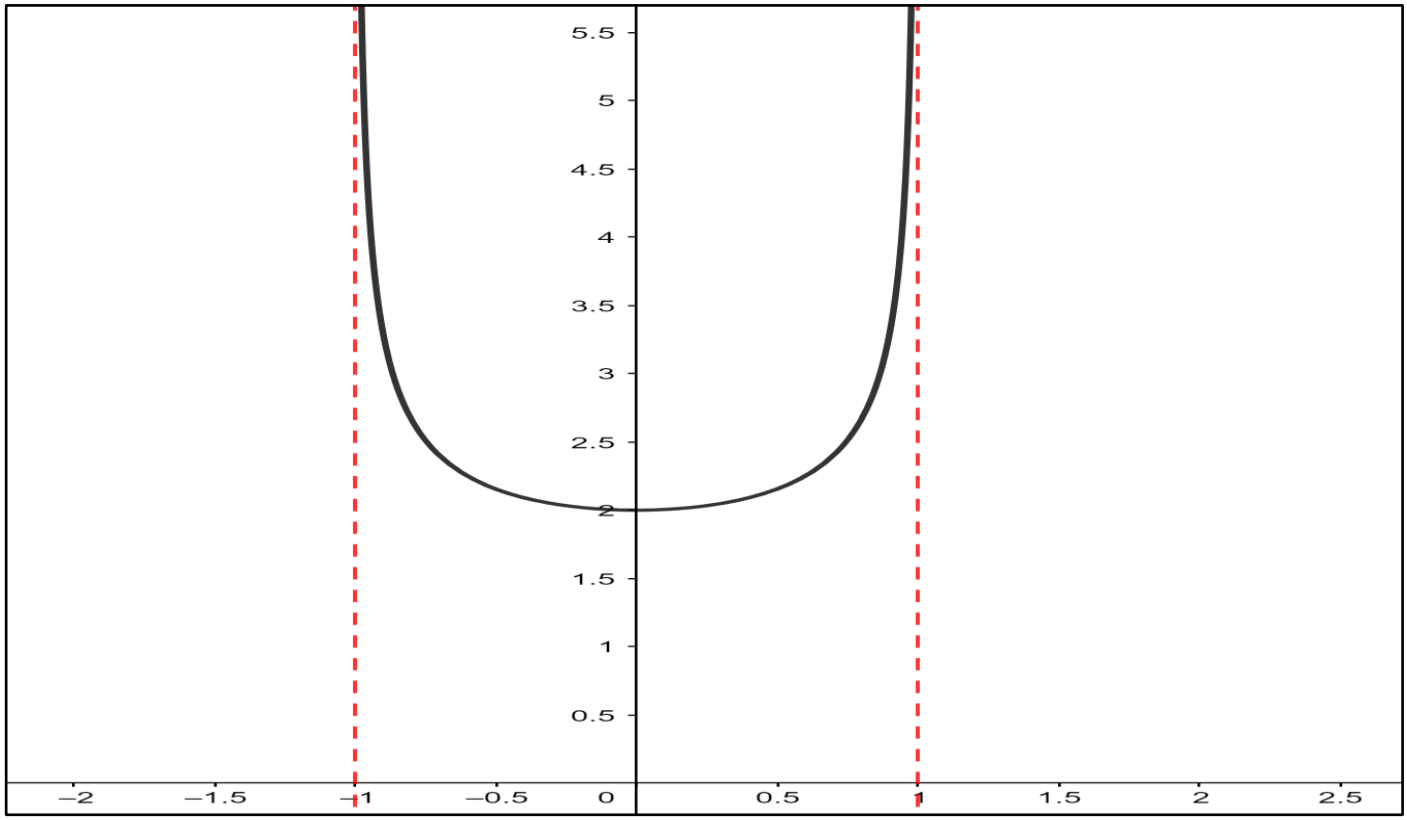
- Notiamo come $Er\big(f(x)\big) \gg Er(x)$ man mano che ci avviciniamo a $x = 1$;
  In particolare, quando calcoliamo $f(x)$, con $x \approx 1$ l'errore esplode passando da un errore sull'input dell'ordine di 10^{-7} ad un errore sull'output dell'ordine di 10^{2}.
  Il problema è evidentemente mal condizionato.

Presentiamo adesso l'intorno destro di $x = 1$, arrivando alla stessa conclusione. Il problema è evidentemente mal condizionato in prossimità di 1, mentre è ben condizionato altrove.



Esempio 4: $f(x) = 1 - \sqrt{1 - x^2}$

- Grafico di $k\big(f(x)\big) = \left|\frac{x^2}{\sqrt{1-x^2}-1+x^2}\right|$
  - Analizzando il grafico dell'indice di condizionamento del problema è facile rendersi conto che nell'intorno di 1 e -1, $k\big(f(x)\big)$ è molto grande
  - Il problema sarà:
    - MAL CONDIZIONATO nell'intorno sinistro di 1 e nell'intorno destro di -1
    - BEN CONDIZIONATO negli altri punti del dominio



- Calcolo della funzione $f(x) = 1 - \sqrt{1 - x^2}$ su un intervallo di estremi prossimi a $x \approx -1$ e $x \approx 1$ e studio dell'errore commesso. Vogliamo mostrare che nell'intorno destro di $x = -1$ e nell'intorno sinistro di $x = 1$, il problema è fortemente mal condizionato. Altrove il problema è ben condizionato in quanti gli errori sono della stessa grandezza.

```
------------------------------------------------------------
|  STUDYING OF CONDITIONING OF FUNCTION: F(x) = 1 - sqrt(1-x^2)  |
------------------------------------------------------------

Enter a perturbation: 0.000001

-----------------------------------------------------------
                Error_X                    Error_Y
-----------------------------------------------------------
            1.000000e-06               7.073896e+00  ★
            1.020619e-06               6.123972e-06
            1.042105e-06               4.745636e-06
            1.064516e-06               4.169513e-06
            1.087912e-06               3.850443e-06
            1.112360e-06               3.652171e-06
            1.137931e-06               3.522468e-06
            1.164706e-06               3.436550e-06
            1.192771e-06               3.381019e-06
            1.222222e-06               3.347976e-06
            1.253165e-06               3.332470e-06
            1.285714e-06               3.331273e-06
            1.477612e-06               3.484701e-06
            1.434783e-06               3.435593e-06
            1.394366e-06               3.395189e-06
            1.356164e-06               3.363854e-06
            1.320000e-06               3.342222e-06
            1.285714e-06               3.331273e-06
            1.253165e-06               3.332470e-06
            1.222222e-06               3.347976e-06
            1.192771e-06               3.381019e-06
            1.164706e-06               3.436550e-06
            1.137931e-06               3.522468e-06
            1.112360e-06               3.652171e-06
            1.087912e-06               3.850443e-06
            1.064516e-06               4.169513e-06
            1.042105e-06               4.745636e-06
            1.020619e-06               6.123972e-06
            1.000000e-06               7.073896e+00  ★
```

COMMENTO SUL GRAFICO

- L'intervallo dal quale abbiamo estratto i 100 punti è $[j, k\,]$ con

  $j = -(1 - 1 * 10^{-14})$

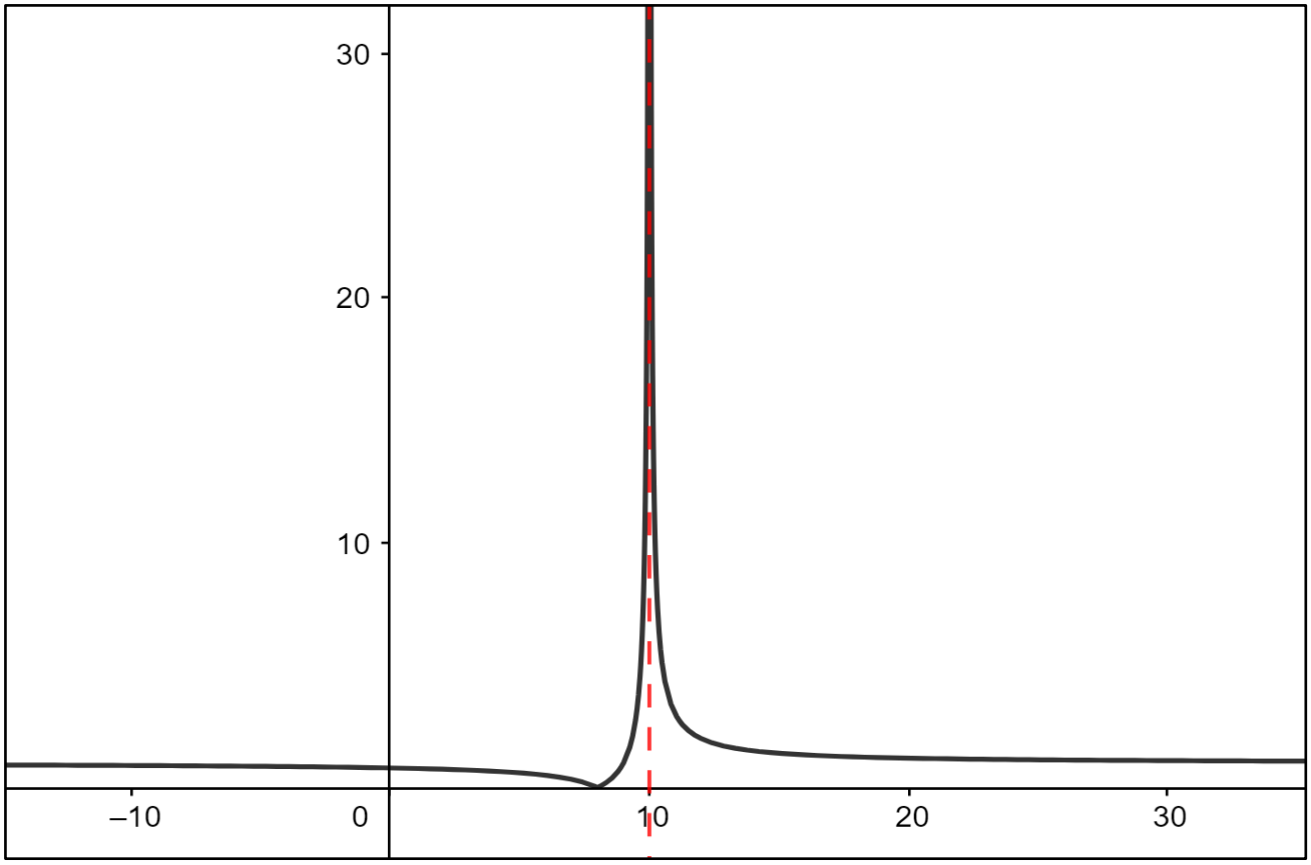
  $k = 1 - 1 * 10^{-14}$

- Notiamo che $Er\big(f(x)\big) \cong Er(x)$ in quasi ogni punto dell'intervallo.
  Quindi come abbiamo già detto il problema risulta ben condizionato.

- Nel calcolare la funzione in $x = j$, $x = k$ il problema è fortemente mal condizionato.
  $Er\big(f(x)\big) \gg Er(x)$ passando da un errore sull'input dell'ordine di 10^{-6} a un errore sull'output dell'ordine di 10^{1}.

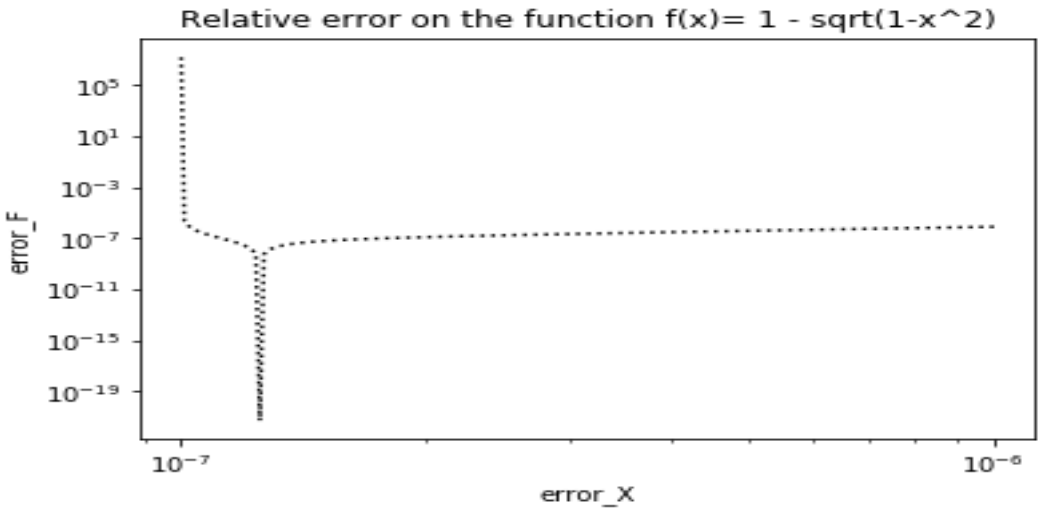
Esempio 5: $f(x) = \sqrt[5]{x^4}\sqrt[5]{10-x}$

- Grafico di $k\big(f(x)\big) = \left|\frac{8-x}{10-x}\right|$
  - Analizzando il grafico dell'indice di condizionamento del problema è facile rendersi conto che nell'intorno di $x = 10$ $k\big(f(x)\big)$ è molto grande
  - Il problema sarà:
    - MAL CONDIZIONATO nell'intorno di $x = 10$
    - BEN CONDIZIONATO negli altri punti del dominio



- Calcolo della funzione $f(x) = \sqrt[5]{x^4}\sqrt[5]{10-x}$ su un intervallo di estremi $[1,10)$ e studio dell'errore commesso. Vogliamo mostrare che man mano ci si avvicina a $x = 10$ l'errore cresce rendendo il problema è fortemente mal condizionato. Altrove il problema è ben condizionato in quanti gli errori sono della stessa grandezza. Mostriamo uno stralcio degli errori commessi nei cento punti in cui calcoliamo la funzione. In particolare, notiamo come nell'andare a calcolare la funzione in punto che è vicinissimo a $x = 10, Er\big(f(x)\big) \gg Er(x)$ passando da 10^{-10} a 10^{4}.

```
------------------------------------------------------------------
|  STUDYING OF CONDITIONING OF FUNCTION: F(x) = 1 - sqrt(1-x^2)  |
------------------------------------------------------------------
Enter a perturbation:  0.000000001

------------------------------------------------------
                Error_X                   Error_Y
------------------------------------------------------
           1.000000e-09              7.777778e-10
           9.166667e-10              7.108844e-10
           8.461539e-10              6.542427e-10
           7.857144e-10              6.056548e-10
           7.333334e-10              5.635088e-10
           6.875001e-10              5.265958e-10
           6.470589e-10              4.939912e-10
           6.111112e-10              4.649759e-10
           5.789474e-10              4.389821e-10
           5.500000e-10              4.155556e-10
           1.111111e-10              1.111111e-10
           1.100000e-10              1.320000e-10
           1.089109e-10              1.573157e-10
           1.078431e-10              1.887255e-10
           1.067961e-10              2.288488e-10
           1.057692e-10              2.820513e-10
           1.047619e-10              3.561905e-10
           1.037736e-10              4.669812e-10
           1.028037e-10              6.510904e-10
           1.018519e-10              1.018519e-09
           1.009174e-10              2.119266e-09
           1.000000e-10              1.876500e+04
```



TEST SULL' APPROSSIMAZIONE DELLA DERIVATA

Supponiamo di non poter valutare direttamente la derivata della $f(x)$ e di volerla approssimare facendo uno sviluppo in serie di Taylor:

$$f(xo+h) = f(xo) + hf'(xo) + \frac{h^2}{2}f''(xo) + \frac{h^3}{6}f'''(xo) + \cdots$$

da cui possiamo ottenere

$$f'(xo) = \frac{f(xo+h) - f(xo)}{h} - \left(\frac{h}{2}f''(xo) + \frac{h^2}{6}f'''(xo) + \cdots\right)$$

Approssimiamo, quindi, la $f'(xo)$ calcolando $\frac{f(xo+h)-f(xo)}{h}$.
L'errore di discretizzazione che si commette è

$$\left|f'(xo) - \frac{f(xo+h) - f(xo)}{h}\right| = \left|\frac{h}{2}f''(xo) + \frac{h^2}{6}f'''(xo) + \cdots\right|$$

Per piccoli valori di h possiamo dare una stima dell'errore di discretizzazione,

$$\left|f'(xo) - \frac{f(xo+h) - f(xo)}{h}\right| \approx \frac{h}{2}|f''(xo)|$$

Implementazione

```
"""
Created on Sat Mar 14 22:51:38 2020
@author: leonardo Saccotelli
"""
import numpy as np
import matplotlib.pyplot as plt

#Definizione della funzione sin(x) di cui si vuole calcolare la derivata
def f(x):
    function = np.sin(x)
    return function

#Definizione della derivata prima della funzione f(x) = sin(x)
def f1(x) :
    derivative = np.cos(x)
    return derivative

def createTable():
    str1 = 'h'
    str2 = 'Derivative'
    str3 = 'Approximation'
    str4 = 'Error'
    print(' ------------------------------------------------------------')
    print('|                  Derivative approximation                  |')
    print(' ------------------------------------------------------------')
    print('   %6s%21s %15s%13s' %(str1,str2,str3,str4))
    print(' ------------------------------------------------------------')

#Punto in cui effettuare l'esperimento e valore derivata
x = 1.2
#Calcolo la derivata prima nel punto x scelto
df = f1(x)

# Sequenza di passi di grandezza decrescente (Calcolo h via via più
piccolo)
h = 2.0**(-np.array(range(50)))

# Approssimazione della derivata
Err = np.zeros(len(h))

createTable()

for i in range(len(h)):

    #Calcolo dell'approssimazione della derivata
    dfh = (f(x+h[i]) - f(x))/h[i]

    #Calcolo dell'errore totale commesso dall'approssimazione
    Err[i] = np.abs(df - dfh)/np.abs(df)
    print(' %13e  %14.8f  %13.8f  %15e ' % (h[i],df,dfh,Err[i]))

#calcolo dell'errore di discretizzazione  dato da f''(x) = -sin(x)
discretizationError = abs(-f(x))*h/2
```

```
#Rappresentazione grafica, rappresentiamo l'errore totale e l'errore di
#discretizzazione
plt.figure(1,figsize=(14,7))
plt.loglog(h,discretizationError)
plt.loglog(h,Err,'*-r')
plt.xlabel('h')
plt.ylabel('Error')
plt.show()
```

Esempio 1: $f(x) = \sin x$

Sia $xo = 1.2$, $f'(x) = \cos x$, vogliamo stimare l'errore che si commette nell'andare ad approssimare la derivata prima come mostrato.

Prendiamo una sequenza di passi h via via più piccoli e calcoliamo l'approssimazione in ogni punto.

Ci aspettiamo che diminuendo il passo h l'errore che commettiamo diminuisca. Mostriamo uno stralcio dei dati ricavati dal test.

Come detto, l'errore decresce al decresce di h.

Derivative approximation			
h	Derivative	Approximation	Error
1.000000e+00	0.36235775	-0.12354268	1.340941e+00
5.000000e-01	0.36235775	0.11925145	6.709013e-01
2.500000e-01	0.36235775	0.24269562	3.302320e-01
1.250000e-01	0.36235775	0.30323822	1.631524e-01
6.250000e-02	0.36235775	0.33300515	8.100449e-02
3.125000e-02	0.36235775	0.34773685	4.034935e-02
1.562500e-02	0.36235775	0.35506160	2.013522e-02
7.812500e-03	0.36235775	0.35871331	1.005759e-02
3.906250e-03	0.36235775	0.36053645	5.026270e-03
1.953125e-03	0.36235775	0.36144733	2.512502e-03
9.765625e-04	0.36235775	0.36190260	1.256092e-03
4.882812e-04	0.36235775	0.36213019	6.280064e-04
2.441406e-04	0.36235775	0.36224398	3.139933e-04
1.220703e-04	0.36235775	0.36230087	1.569942e-04
6.103516e-05	0.36235775	0.36232931	7.849645e-05
3.051758e-05	0.36235775	0.36234353	3.924807e-05
1.525879e-05	0.36235775	0.36235064	1.962400e-05
7.629395e-06	0.36235775	0.36235420	9.811956e-06
3.814697e-06	0.36235775	0.36235598	4.905935e-06
1.907349e-06	0.36235775	0.36235687	2.452865e-06
9.536743e-07	0.36235775	0.36235731	1.226249e-06
4.768372e-07	0.36235775	0.36235753	6.129411e-07
2.384186e-07	0.36235775	0.36235764	3.064479e-07
1.192093e-07	0.36235775	0.36235770	1.522374e-07
5.960464e-08	0.36235775	0.36235773	7.256200e-08
2.980232e-08	0.36235775	0.36235774	3.143921e-08
1.490116e-08	0.36235775	0.36235775	1.087782e-08
7.450581e-09	0.36235775	0.36235775	1.087782e-08
3.725290e-09	0.36235775	0.36235777	3.024497e-08

Possiamo pensare di ottenere un'accuratezza grande quanto vogliamo a condizione di prendere valori di h sempre più piccoli. In realtà, per valori di h molto piccoli, gli errori iniziano ad aumentare!

L'errore che noi valutiamo è dato dalla **somma dell'errore di discretizzazione** e **dell'errore di arrotondamento**. Per valori di h grandi, l'errore di discretizzazione decresce al diminuire di h e domina l'errore di arrotondamento.

Ma quando l'errore di discretizzazione diventa molto piccolo, per valori di h minori di 10^{-8} , allora l'errore di arrotondamento inizia a dominare e ad aumentare sempre più al diminuire di h.

Nell'errore di arrotondamento, per h via via più piccoli, si verifica un errore di cancellazione: $f(xo + h)$ è praticamente uguale a $f(xo)$ per h molto piccoli!

Derivative approximation

h	Derivative	Approximation	Error
1.862645e-09	0.36235775	0.36235780	1.124905e-07
9.313226e-10	0.36235775	0.36235785	2.769817e-07
4.656613e-10	0.36235775	0.36235785	2.769817e-07
2.328306e-10	0.36235775	0.36235809	9.349463e-07
1.164153e-10	0.36235775	0.36235809	9.349463e-07
5.820766e-11	0.36235775	0.36235809	9.349463e-07
2.910383e-11	0.36235775	0.36235809	9.349463e-07
1.455192e-11	0.36235775	0.36235809	9.349463e-07
7.275958e-12	0.36235775	0.36236572	2.198981e-05
3.637979e-12	0.36235775	0.36236572	2.198981e-05
1.818989e-12	0.36235775	0.36236572	2.198981e-05
9.094947e-13	0.36235775	0.36242676	1.904288e-04
4.547474e-13	0.36235775	0.36254883	5.273066e-04
2.273737e-13	0.36235775	0.36279297	1.201062e-03
1.136868e-13	0.36235775	0.36230469	1.464491e-04
5.684342e-14	0.36235775	0.36328125	2.548574e-03
2.842171e-14	0.36235775	0.36328125	2.548574e-03
1.421085e-14	0.36235775	0.36718750	1.332867e-02
7.105427e-15	0.36235775	0.37500000	3.488885e-02
3.552714e-15	0.36235775	0.37500000	3.488885e-02
1.776357e-15	0.36235775	0.37500000	3.488885e-02

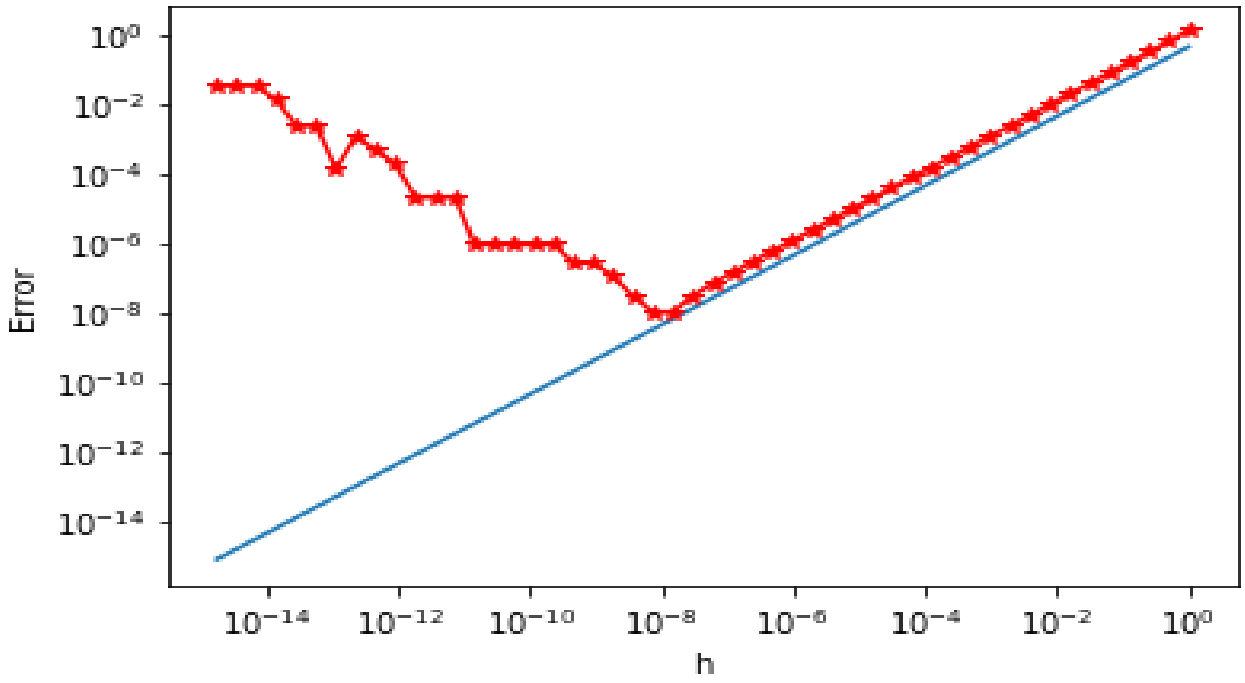
È facile verificare quanto detto direttamente dal grafico riportato di seguito.

In rosso riportiamo l'errore totale commesso nell'approssimare la derivata; in blu riportiamo l'errore di discretizzazione.

È evidente come l'errore di discretizzazione diventi molto piccolo al decrescere di h e l'errore totale decresce. Questo succede fintanto che $h \approx 10^{-8}$.

Per ordine di grandezza inferiori sebbene l'errore di discretizzazione continui a decrescere, l'errore di arrotondamento inizia a crescere e a prevalere portando di fatto l'errore totale a crescere.

Notiamo come in corrispondenza di $h \approx 10^{-14}$, l'errore totale è $\approx 10^{-2}$.



Esempio 2: $f(x) = \sqrt[3]{x}$

Riportiamo lo stralcio dell'esecuzione e il grafico dell'errore totale commesso nell'approssimazione della derivata.

Notiamo nuovamente che decrescendo h, l'errore totale decresce e si mantengono nello stesso ordine di grandezza.

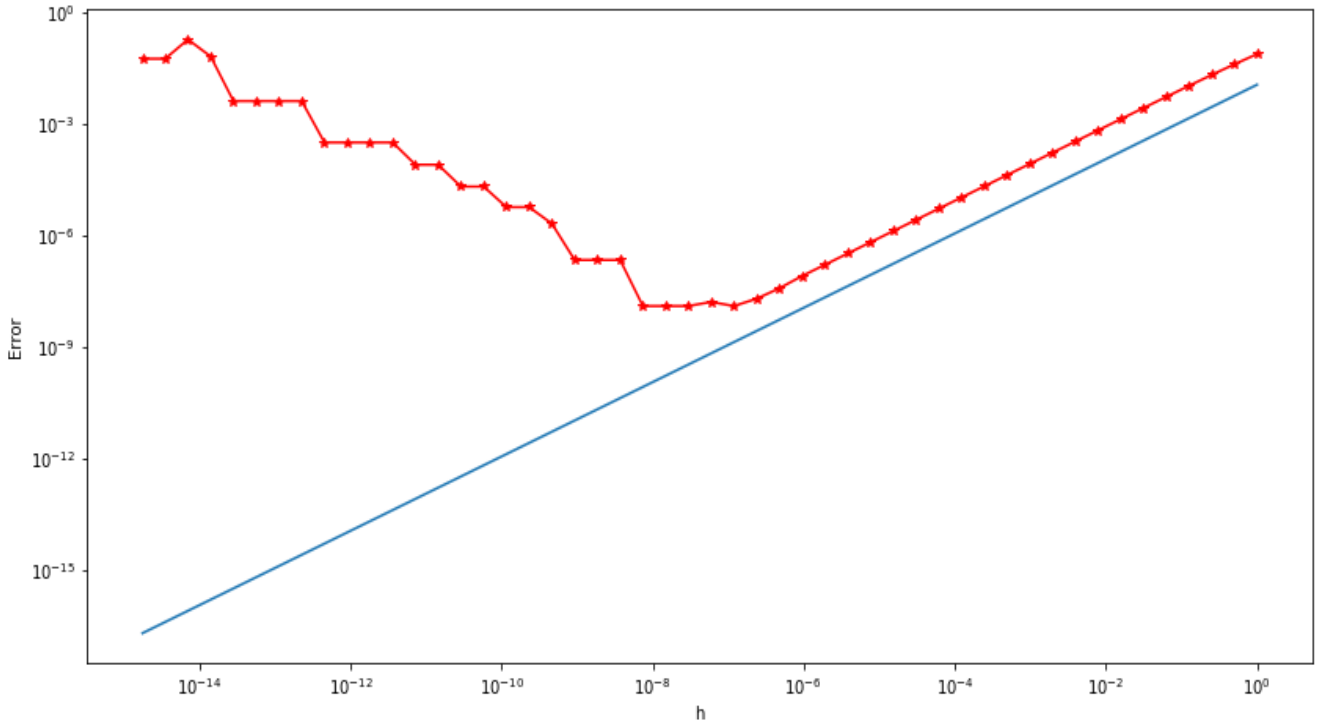
Raggiunto $h \approx 10^{-8}$, l'errore totale cresce con il decrescere di h.

Infatti, per $h \approx 10^{-15}$, $Etot \approx 10^{-2}$.

Derivative approximation			
h	Derivative	Approximation	Error
1.000000e+00	0.13228342	0.12257489	7.339186e-02
5.000000e-01	0.13228342	0.12712514	3.899412e-02
2.500000e-01	0.13228342	0.12961939	2.013878e-02
1.250000e-01	0.13228342	0.13092890	1.023951e-02
6.250000e-02	0.13228342	0.13160036	5.163588e-03
3.125000e-02	0.13228342	0.13194042	2.592922e-03
1.562500e-02	0.13228342	0.13211155	1.299265e-03
7.812500e-03	0.13228342	0.13219739	6.503362e-04
3.906250e-03	0.13228342	0.13224038	3.253443e-04
1.953125e-03	0.13228342	0.13226190	1.627163e-04
9.765625e-04	0.13228342	0.13227266	8.136917e-05
4.882812e-04	0.13228342	0.13227804	4.068734e-05
2.441406e-04	0.13228342	0.13228073	2.034436e-05
1.220703e-04	0.13228342	0.13228208	1.017234e-05
6.103516e-05	0.13228342	0.13228275	5.086222e-06
3.051758e-05	0.13228342	0.13228308	2.543086e-06
1.525879e-05	0.13228342	0.13228325	1.271531e-06
7.629395e-06	0.13228342	0.13228334	6.358087e-07
3.814697e-06	0.13228342	0.13228338	3.176726e-07
1.907349e-06	0.13228342	0.13228340	1.583845e-07
9.536743e-07	0.13228342	0.13228341	7.918047e-08
4.768372e-07	0.13228342	0.13228342	3.693832e-08
2.384186e-07	0.13228342	0.13228342	1.933743e-08
1.192093e-07	0.13228342	0.13228342	1.229707e-08
5.960464e-08	0.13228342	0.13228342	1.586436e-08
2.980232e-08	0.13228342	0.13228342	1.229707e-08
1.490116e-08	0.13228342	0.13228342	1.229707e-08
7.450581e-09	0.13228342	0.13228342	1.229707e-08
3.725290e-09	0.13228342	0.13228345	2.129944e-07
1.862645e-09	0.13228342	0.13228345	2.129944e-07
9.313226e-10	0.13228342	0.13228345	2.129944e-07
4.656613e-10	0.13228342	0.13228369	2.015326e-06

Derivative approximation

h	Derivative	Approximation	Error
2.328306e-10	0.13228342	0.13228416	5.619989e-06
1.164153e-10	0.13228342	0.13228416	5.619989e-06
5.820766e-11	0.13228342	0.13228607	2.003864e-05
2.910383e-11	0.13228342	0.13228607	2.003864e-05
1.455192e-11	0.13228342	0.13229370	7.771325e-05
7.275958e-12	0.13228342	0.13229370	7.771325e-05
3.637979e-12	0.13228342	0.13232422	3.084117e-04
1.818989e-12	0.13228342	0.13232422	3.084117e-04
9.094947e-13	0.13228342	0.13232422	3.084117e-04
4.547474e-13	0.13228342	0.13232422	3.084117e-04
2.273737e-13	0.13228342	0.13281250	3.999587e-03
1.136868e-13	0.13228342	0.13281250	3.999587e-03
5.684342e-14	0.13228342	0.13281250	3.999587e-03
2.842171e-14	0.13228342	0.13281250	3.999587e-03
1.421085e-14	0.13228342	0.14062500	6.305839e-02
7.105427e-15	0.13228342	0.15625000	1.811760e-01
3.552714e-15	0.13228342	0.12500000	5.505921e-02
1.776357e-15	0.13228342	0.12500000	5.505921e-02

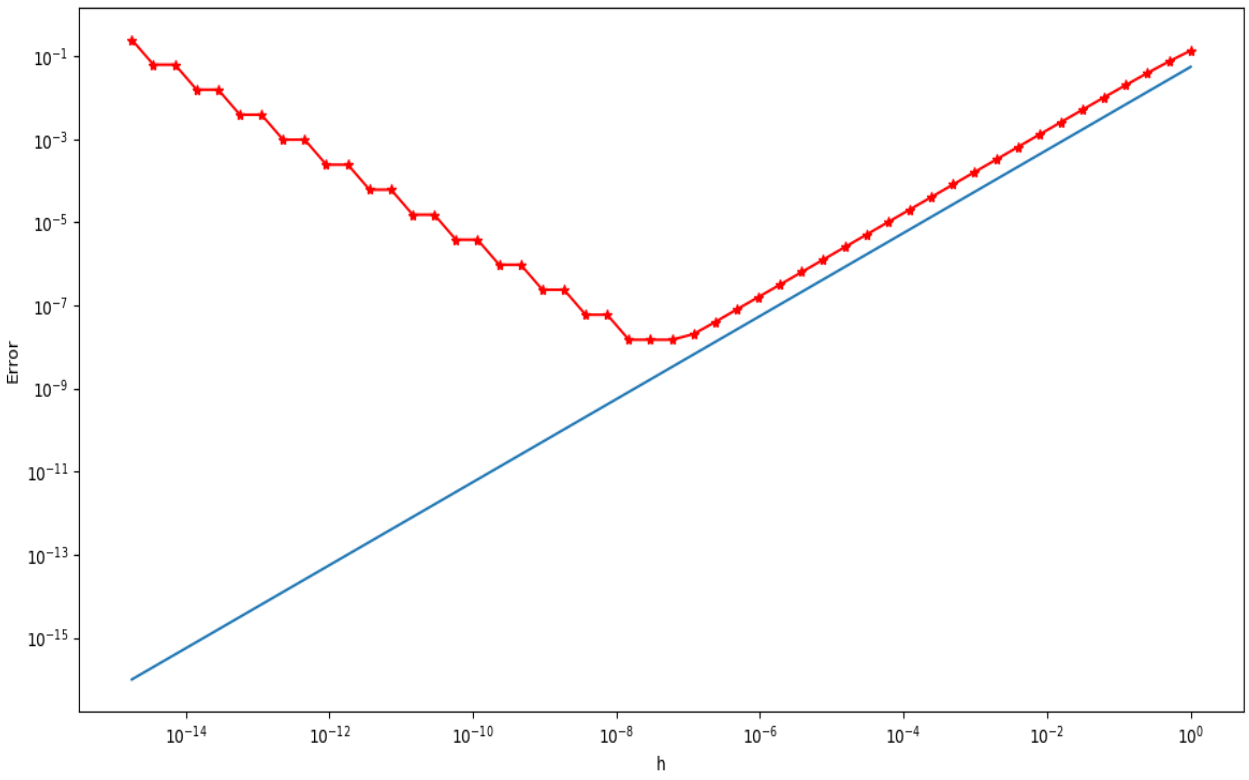


Esempio 3: $f(x) = \log x$

Forniamo un ulteriore esempio come ulteriore prova di quello che abbiamo già detto.

Derivative approximation			
h	Derivative	Approximation	Error
1.000000e+00	0.33333333	0.28768207	1.369538e-01
5.000000e-01	0.33333333	0.30830136	7.509592e-02
2.500000e-01	0.33333333	0.32017083	3.948751e-02
1.250000e-01	0.33333333	0.32657596	2.027213e-02
6.250000e-02	0.33333333	0.32990860	1.027421e-02
3.125000e-02	0.33333333	0.33160919	5.172445e-03
1.562500e-02	0.33333333	0.33246828	2.595160e-03
7.812500e-03	0.33333333	0.33290006	1.299827e-03
3.906250e-03	0.33333333	0.33311651	6.504771e-04
1.953125e-03	0.33333333	0.33322487	3.253796e-04
9.765625e-04	0.33333333	0.33327909	1.627251e-04
4.882812e-04	0.33333333	0.33330621	8.137138e-05
2.441406e-04	0.33333333	0.33331977	4.068790e-05
1.220703e-04	0.33333333	0.33332655	2.034450e-05
6.103516e-05	0.33333333	0.33332994	1.017239e-05
3.051758e-05	0.33333333	0.33333164	5.086236e-06
1.525879e-05	0.33333333	0.33333249	2.543136e-06
7.629395e-06	0.33333333	0.33333291	1.271575e-06
3.814697e-06	0.33333333	0.33333312	6.358605e-07
1.907349e-06	0.33333333	0.33333323	3.180467e-07
9.536743e-07	0.33333333	0.33333328	1.594890e-07
4.768372e-07	0.33333333	0.33333331	8.055940e-08
2.384186e-07	0.33333333	0.33333332	4.004687e-08
1.192093e-07	0.33333333	0.33333333	2.048910e-08
5.960464e-08	0.33333333	0.33333333	1.490116e-08
2.980232e-08	0.33333333	0.33333333	1.490116e-08
1.490116e-08	0.33333333	0.33333333	1.490116e-08
7.450581e-09	0.33333333	0.33333331	5.960464e-08
3.725290e-09	0.33333333	0.33333331	5.960464e-08
1.862645e-09	0.33333333	0.33333325	2.384186e-07
9.313226e-10	0.33333333	0.33333325	2.384186e-07
4.656613e-10	0.33333333	0.33333302	9.536743e-07
2.328306e-10	0.33333333	0.33333302	9.536743e-07
1.164153e-10	0.33333333	0.33333206	3.814697e-06

Derivative approximation			
h	Derivative	Approximation	Error
5.820766e-11	0.33333333	0.33333206	3.814697e-06
2.910383e-11	0.33333333	0.33332825	1.525879e-05
1.455192e-11	0.33333333	0.33332825	1.525879e-05
7.275958e-12	0.33333333	0.33331299	6.103516e-05
3.637979e-12	0.33333333	0.33331299	6.103516e-05
1.818989e-12	0.33333333	0.33325195	2.441406e-04
9.094947e-13	0.33333333	0.33325195	2.441406e-04
4.547474e-13	0.33333333	0.33300781	9.765625e-04
2.273737e-13	0.33333333	0.33300781	9.765625e-04
1.136868e-13	0.33333333	0.33203125	3.906250e-03
5.684342e-14	0.33333333	0.33203125	3.906250e-03
2.842171e-14	0.33333333	0.32812500	1.562500e-02
1.421085e-14	0.33333333	0.32812500	1.562500e-02
7.105427e-15	0.33333333	0.31250000	6.250000e-02
3.552714e-15	0.33333333	0.31250000	6.250000e-02
1.776357e-15	0.33333333	0.25000000	2.500000e-01



CONFRONTO TEMPI RICHIESTI NEL CALCOLO DEL DETERMINANTE DI UNA MATRICE

Nel seguente esperimento vogliamo confrontare i tempi richiesti dal calcolatore per calcolare il determinante di matrici di varie dimensioni.

Abbiamo implementato il metodo di Laplace per il calcolo del determinante, cronometrando i tempi richiesti dall'algoritmo per calcolare il determinante al variare della dimensione della matrice.

Eseguiamo lo stesso esperimento utilizzando il metodo della libreria numpy per il calcolo del determinante, metodo intuitivamente molto più efficiente del metodo di Laplace.

Nella prima tabella riportiamo i risultati ottenuti utilizzando l'algoritmo di Laplace. Nella seconda i risultati ottenuti utilizzando l'algoritmo della libreria numpy. È facile notare la differenza dei tempi richiesti da entrambi gli algoritmi nel calcolare il determinante di una matrice 10×10.

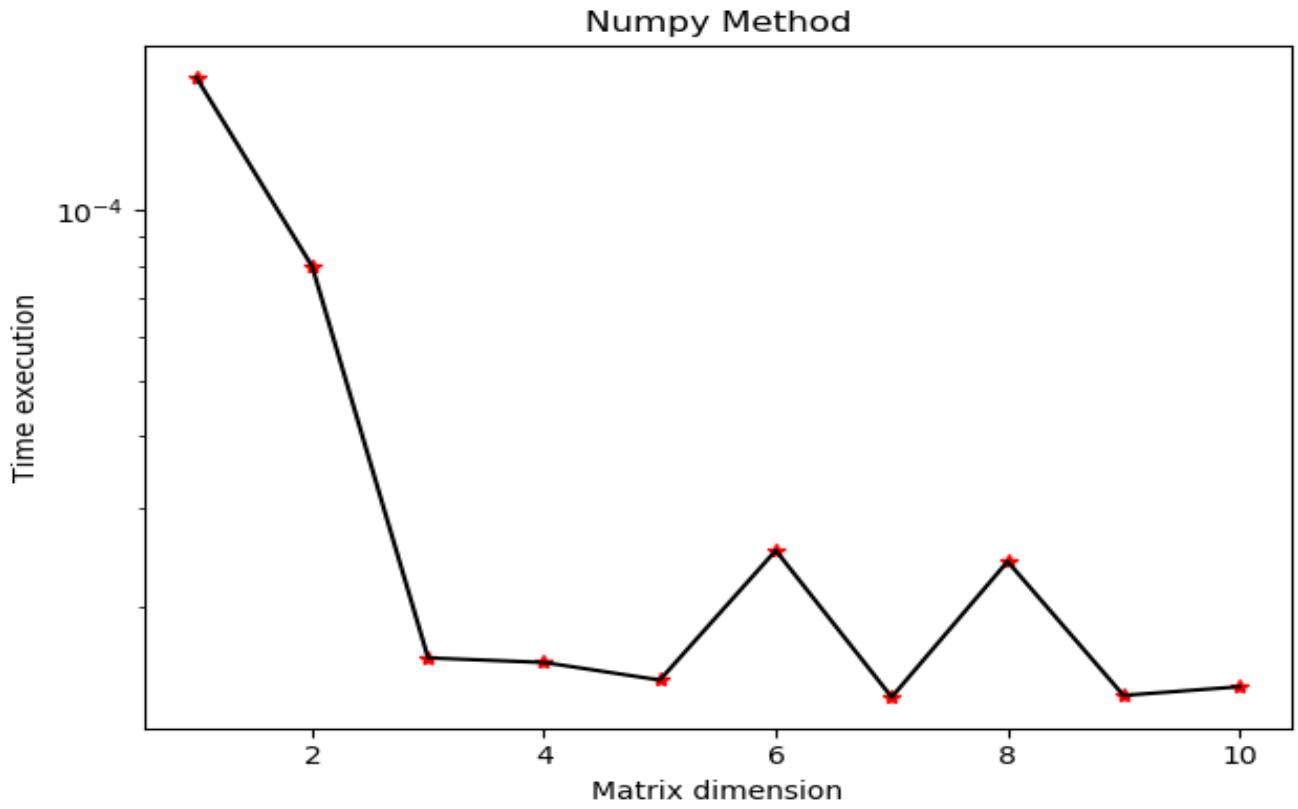
Implementazione

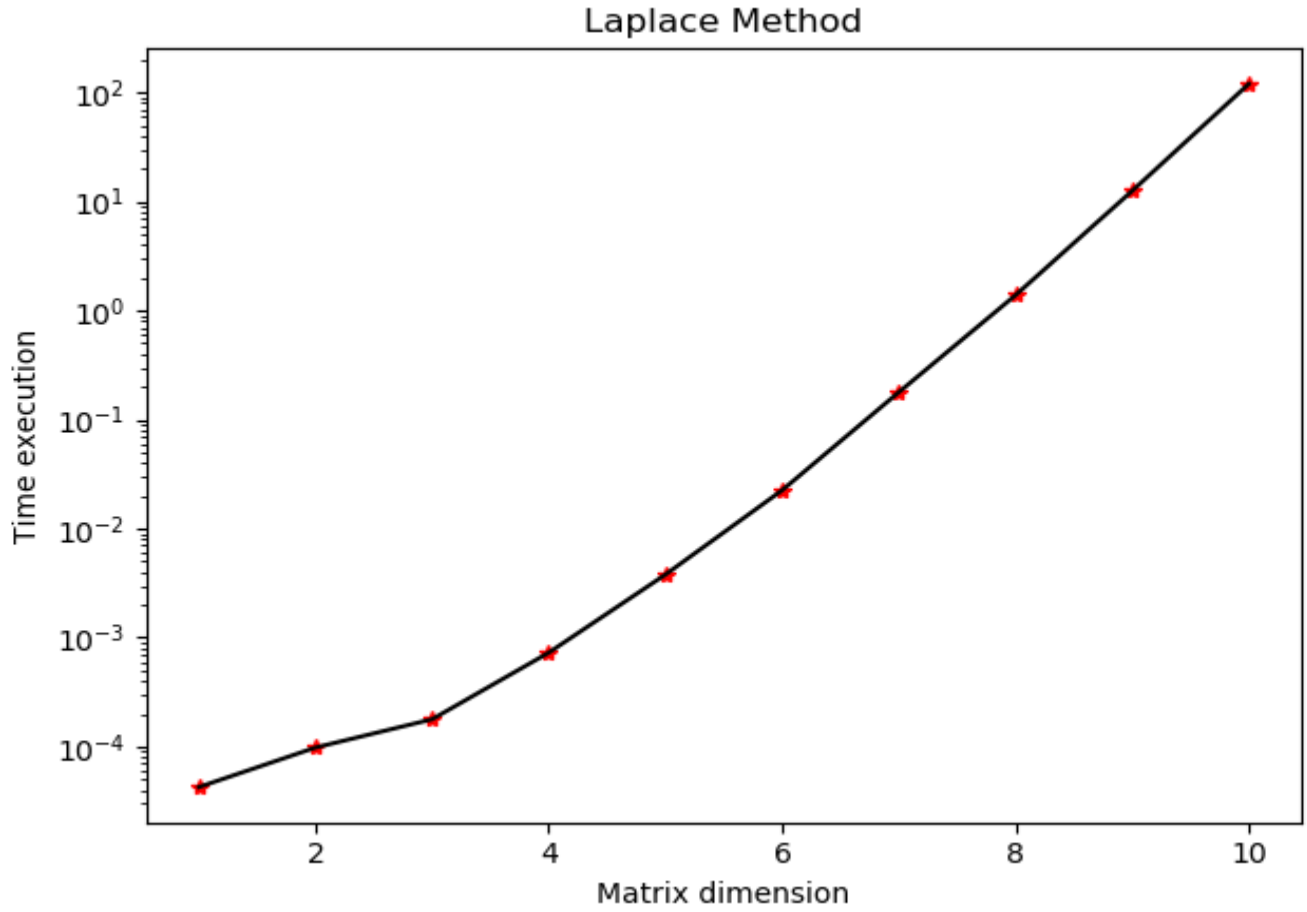
```
"""
Created on Tue Mar 17 23:17:42 2020

@author: Leonardo Saccotelli
"""
import numpy as np
import matplotlib.pyplot as plt
from time import clock

#Metodo di Laplace per il determinante
def determinant(matrix):
    #Ritorna le dimensioni nella lista (r,c) quindi in shape[0] ritorna
    #la lunghezza della riga
    n = matrix.shape[0]
    #Controllo se la matrice è una 1x1
    if n == 1:
        #il determinante coincide con l'unico elemento della matrice
        det = matrix[0, 0]
    else:
        det = 0
        for j in range(1, n + 1):
            #rimuovo la colonna j-1 della matrice, rimuovo la riga 0
            matrix1j = np.delete(np.delete(matrix, j - 1, 1), 0, 0)
            #calcolo il determinante
            det = det + (-1) ** (j + 1) * matrix[0, j - 1] *
determinant(matrix1j)
    return det


#Testiamo l'algoritmo su matrici di dimensioni comprese tra 1 e 10
print('         ------------------------------------------------')
str1 = 'Matrix dimension'
str2 = 'Execution time'
print('%30s %20s' %(str1,str2))
print('         ------------------------------------------------')

#Numero massimo per la dimensione della matrice
max_n = 10

#Vettore per memorizzare i tempi di esecuzione di ogni determinante
timeDet = np.zeros(max_n)

for n in range(1, max_n+1):
    #popolo la matrice con valori random
    matrix = np.random.random((n, n))
    #Registro l'istante in cui inizia l'esecuzione del determinante
    startDet = clock()
    d = al.determinant(matrix)
    #d = np.linalg.det(matrix)
    #Registro l'istante in cui termina l'esecuzione del determinante
    endDet = clock()
    timeDet[n-1] = (endDet - startDet)

    print('%22d%28e' % (n, timeDet[n-1]))
```

```
#Creo un grafico
dimMatrix = np.array([i for i in range(1,max_n+1)])
plt.figure(1)
plt.plot(dimMatrix,timeDet,'r*')
plt.semilogy(dimMatrix,timeDet, 'k')
plt.title('Laplace Method')
plt.xlabel('Matrix dimension')
plt.ylabel('Time execution')
plt.show()
```

Matrix dimension	Execution time
1	1.716000e-04
2	7.970000e-05
3	1.630000e-05
4	1.600000e-05
5	1.490000e-05
6	2.520000e-05
7	1.390000e-05
8	2.410000e-05
9	1.400000e-05
10	1.450000e-05

Matrix dimension	Execution time
1	4.260000e-05
2	9.820000e-05
3	1.785000e-04
4	7.289000e-04
5	3.789200e-03
6	2.247930e-02
7	1.779766e-01
8	1.379619e+00
9	1.248641e+01
10	1.195586e+02


Numpy Method
Time execution
10^{-4}
2
4
6
8
10
Matrix dimension


Laplace Method
Time execution
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
2
4
6
8
10
Matrix dimension

APPLICAZIONE DEL METODO DI ELIMINAZIONE DI GAUSS (CON E SENZA PIVOTING) ALLA MATRICE DI HILBERT

In questo esperimento vogliamo confrontare l'algoritmo di eliminazione di Gauss con e senza pivoting.

La matrice su cui testeremo i due algoritmi è la matrice di Hilbert,

una matrice fortemente mal condizionata.

Costruiamo quindi un sistema lineare che andremo a risolvere separatamente, fornendo i risultati dei diversi algoritmi.

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{n+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n-1} \end{bmatrix}$$

Poiché l'algoritmo con pivoting porta in posizione pivot quel valore sulla colonna in modulo massimo, questa operazione tende a stabilizzare l'algoritmo e a propagare meno gli errori.

Ci aspettiamo quindi un errore minore quando utilizziamo l'algoritmo con pivoting rispetto a quando utilizziamo quello senza pivoting.

Ripetiamo il test su una matrice di dimensione 10×10, 15×15 e 20×20.

Leggendo i risultati, non riscontriamo quello che abbiamo ipotizzato.

La matrice di Hilbert è fortemente mal condizionata, talmente mal condizionata che il miglioramento dovuto al pivoting viene sovrastato dagli errori legati al mal condizionamento.

Implementazione algoritmo di sostituzione all'indietro, metodo di eliminazione di Gauss senza pivoting, metodo di eliminazione di Gauss con pivoting

```
#Algoritmo di sostituzione all'indietro
def backwardSubstition(matrix, vettb):
    #estrapolo il numero di righe della matrice
    n = matrix.shape[0]

    #inizializzo il vettore delle soluzioni
    x = np.zeros(n)

    #Controllo se il determinante è uguale a zero
    #nel caso il sistema non è risolvibile
    if (abs(np.prod(np.diag(matrix))) < 1.0e-200):
        print(' Matrix could be singular.\n' +
              ' System without solution!')
    else:
        #Calcolo la soluzione immediata
        x[n-1] = vettb[n-1]/matrix[n-1][n-1]
        for i in range(n-2, -1, -1):
            s = 0
            for j in range(i+1,n):
                s = s + matrix[i][j]*x[j]
            x[i] = (vettb[i]-s)/matrix[i][i]
    #restituisco le soluzioni
    return x

#Algoritmo di eliminazione di Gauss
def GaussElimination(matrixA, bV):

    #Effettuo una copia dei valori contenuti nei parametri
    matrix = np.copy(matrixA)
    b = np.copy(bV)

    #Ricavo la dimensione della matrice
    n = matrix.shape[0]

    for j in range(0, n):
        for i in range(j+1, n):
            #Calcolo m = matrix[i][j]/matrix[j][j]
```

```
            m = matrix[i][j] / matrix[j][j]

            #Azzero l'elemento sotto la diagonale principale
            matrix[i][j] = 0
            #Aggiorno i valori dell'i-esima riga della matrice
            for k in range(j+1, n):
                matrix[i][k] = matrix[i][k] - m * matrix[j][k]
            #Aggiorno i-esimo elemento del vettore dei termini noti
            b[i] = b[i] - m * b[j]
    #Restituisco la matrice triangolare e il vettore dei termini noti
    return matrix, b


#Algoritmo di eliminazione di Gauss con Pivoting
def GaussEliminationPivoting(matrixA, bV):

    #Effettuo una copia dei valori contenuti nei parametri
    matrix = np.copy(matrixA)
    b = np.copy(bV)

    #Ricavo la dimensione della matrice
    n = matrix.shape[0]

    for j in range(0, n):
        #INDIVIDUAZIONE ELEMENTO PIVOT
        pivotMax = abs(matrix[j][j])
        indexPivotMax = j

        for i in range(j+1, n):
            if abs(matrix[i][j]) > pivotMax:
                pivotMax = abs(matrix[i][j])
                indexPivotMax = i

        #SCAMBIO RIGA j CON RIGA indexPivotMax
        if (indexPivotMax > j):
            #Scambio le righe della matrice
            for k in range (j,n):
                Atemp = np.copy(matrix[j][k])
                matrix[j][k] = np.copy(matrix[indexPivotMax][k])
                matrix[indexPivotMax][k] = np.copy(Atemp)
            #Scambio l'ordine del vettore dei termini noti
            btemp = np.copy(b[j])
            b[j] = np.copy(b[indexPivotMax])
            b[indexPivotMax] = np.copy(btemp)

        #ELIMINAZIONE DI GAUSS
        for i in range(j+1, n):
            #Calcolo m = matrix[i][j]/matrix[j][j]
            m = matrix[i][j] / matrix[j][j]

            #Azzero l'elemento sotto la diagonale principale
            matrix[i][j] = 0

            #Aggiorno i valori dell'i-esima riga della matrice
            for k in range(j+1, n):
                matrix[i][k] = matrix[i][k] - m * matrix[j][k]
```

```
            #Aggiorno i-esimo elemento del vettore dei termini noti
            b[i] = b[i] - m * b[j]
    #Restituisco la matrice triangolare e il vettore dei termini noti
    return matrix, b


#Funzione per la creazione della matrice di Hilbert
def hilbertMatrix(n):
    #Creo la matrice
    matrix = np.zeros((n,n))

    for i in range(1,n+1):
        for j in range(1,n+1):
            matrix[i-1][j-1]= 1/(i+j-1)

    return matrix
```

Test matrice di Hilbert senza pivoting

```
"""
Created on Fri Mar 20 18:51:00 2020

@author: Leonardo Saccotelli

"""
import numpy as np
import AlgoritmiAlgebraLineare as al

#Funzione per perturbare il vettore dei termini noti
def perturbation(bvett):
    b = np.copy(bvett)
    n = len(b)


 # Creo un vettore di perturbazioni
    bDelta = np.array([i for i in range(1,n+1)])*10**(-8)

    #Perturbo il vettore dei termini noti
    for i in range(0, n):
        b[i] = b[i] + bDelta[i]
    return b

#------------------- TEST ELIMINAZIONE DI GAUSS SENZA PIVOTING

#Dimensione della matrice
n = 10

#Creo la matrice di Hilbert
matrix = al.hilbertMatrix(n).astype(float)

#Vettore delle soluzioni
xSol = np.array([i for i in range(1, n+1)])

#Vettore dei termini noti, a cui applico una perturbazione
b = perturbation(np.dot(matrix, xSol))
```

```
#Calcolo dell'indice di condizionamento del problema
conditionNumber = np.linalg.cond(matrix,1 )

# ------ APPLICO GAUSS SENZA PIVOTING

#Creo la matrice triangolare superiore
matrix, b = al.GaussElimination(matrix, b)

#Calcolo le soluzioni tramite la backwardSubstition
xFind = al.backwardSubstition(matrix, b)

#Calcolo l'errore relativo sulla struttura
#applicando la norma 2
xError = np.linalg.norm((xSol - xFind), 2)

# ------- PRINT RESULT GAUSS ELIMANTION WITHOUT PIVOTING

print('\n Gaussian elimination without Pivoting: Test on Hilbert
matrix')
print(' ------------------------------------------------------------
----')
for i in range(n):
    print('   xFind[%2d] = %24.16f    xSol[%2d] = %5.3f' % (i, xFind[i],
i, xSol[i]))

print(' ------------------------------------------------------------
----')
print('   Difference ||xFind-xSol|| = %e' %xError)
print('   Matrix condition number   = %e' %conditionNumber )
```

Test matrice di Hilbert con pivoting

```
"""
Created on Fri Mar 20 18:51:00 2020
@author: Leonardo Saccotelli
"""
import numpy as np
import AlgoritmiAlgebraLineare as al

#Funzione per perturbare il vettore dei termini noti
def perturbation(bvett):

    b = np.copy(bvett)
    n = len(b)
    # Creo un vettore di perturbazioni
    bDelta = np.array([i for i in range(1,n+1)])*10**(-8)

    #bDelta = 0
    #Perturbo il vettore dei termini noti
    for i in range(0, n):
        b[i] = b[i] + bDelta[i]
    return b

#------------------- TEST ELIMINAZIONE DI GAUSS PIVOTING
```

```
#Dimensione della matrice
n = 10
#Creo la matrice di Hilbert
matrix = al.hilbertMatrix(n).astype(float)

#Vettore delle soluzioni
xSol = np.array([i for i in range(1, n+1)])

#Vettore dei termini noti, a cui applico una perturbazione
b = perturbation(np.dot(matrix, xSol))

#Calcolo dell'indice di condizionamento del problema
conditionNumber = np.linalg.cond(matrix,1 )
# ------ APPLICO GAUSS CON PIVOTING

#Creo la matrice triangolare superiore
matrix, b = al.GaussEliminationPivoting(matrix, b)

#Calcolo le soluzioni tramite la backwardSubstition
xFind = al.backwardSubstition(matrix, b)

#Calcolo l'errore relativo sulla struttura
#applicando la norma 2
xError = np.linalg.norm((xSol - xFind), 2)


# ------- PRINT RESULT GAUSS ELIMANTION WITH PIVOTING

print('\n Gaussian elimination with Pivoting: Test on Hilbert matrix')
print(' ------------------------------------------------------------------
-')
for i in range(n):
    print('   xFind[%2d] = %24.16f    xSol[%2d] = %5.3f' % (i, xFind[i],
i, xSol[i]))

print(' ------------------------------------------------------------------
-')
print('   Difference ||xFind-xSol|| = %e' %xError)
print('   Matrix condition number   = %e' %conditionNumber )
```

```
Gaussian elimination without Pivoting: Test on Hilbert matrix
-------------------------------------------------------------------
  xFind[ 0] =       1.0000000072187927    xSol[ 0] = 1.000
  xFind[ 1] =       1.9999993666726295    xSol[ 1] = 2.000
  xFind[ 2] =       3.0000136353971389    xSol[ 2] = 3.000
  xFind[ 3] =       3.9998750466346400    xSol[ 3] = 4.000
  xFind[ 4] =       5.0005996874629668    xSol[ 4] = 5.000
  xFind[ 5] =       5.9983433771741508    xSol[ 5] = 6.000
  xFind[ 6] =       7.0027288335004068    xSol[ 6] = 7.000
  xFind[ 7] =       7.9973542151948154    xSol[ 7] = 8.000
  xFind[ 8] =       9.0013928566728882    xSol[ 8] = 9.000
  xFind[ 9] =       9.9996929685647213    xSol[ 9] = 10.000
-------------------------------------------------------------------
  Difference ||xFind-xSol|| = 4.427285e-03
  Matrix condition number   = 3.535295e+13


 Gaussian elimination with Pivoting: Test on Hilbert matrix
 -----------------------------------------------------------------
   xFind[ 0] =       1.0000000064432690    xSol[ 0] = 1.000
   xFind[ 1] =       1.9999994314341103    xSol[ 1] = 2.000
   xFind[ 2] =       3.0000122975066690    xSol[ 2] = 3.000
   xFind[ 3] =       3.9998868853921348    xSol[ 3] = 4.000
   xFind[ 4] =       5.0005445483722131    xSol[ 4] = 5.000
   xFind[ 5] =       5.9984917922028478    xSol[ 5] = 6.000
   xFind[ 6] =       7.0024898493458627    xSol[ 6] = 7.000
   xFind[ 7] =       7.9975813337195865    xSol[ 7] = 8.000
   xFind[ 8] =       9.0012753984419476    xSol[ 8] = 9.000
   xFind[ 9] =       9.9997184519786391    xSol[ 9] = 10.000
 -----------------------------------------------------------------
   Difference ||xFind-xSol|| = 4.042200e-03
   Matrix condition number   = 3.535295e+13
```

```
Gaussian elimination without Pivoting: Test on Hilbert matrix
-------------------------------------------------------------------
  xFind[ 0] =        0.9999993524751538    xSol[ 0] = 1.000
  xFind[ 1] =        2.0000952056336527    xSol[ 1] = 2.000
  xFind[ 2] =        2.9965764562218182    xSol[ 2] = 3.000
  xFind[ 3] =        4.0523215459504325    xSol[ 3] = 4.000
  xFind[ 4] =        4.5835242592867962    xSol[ 4] = 5.000
  xFind[ 5] =        7.8579553898138892    xSol[ 5] = 6.000
  xFind[ 6] =        2.6038338462881541    xSol[ 6] = 7.000
  xFind[ 7] =       10.4805607529166078    xSol[ 7] = 8.000
  xFind[ 8] =       25.7499763815651939    xSol[ 8] = 9.000
  xFind[ 9] =      -47.4169493956056982    xSol[ 9] = 10.000
  xFind[10] =      105.5003279186352643    xSol[10] = 11.000
  xFind[11] =      -81.4619240428928464    xSol[11] = 12.000
  xFind[12] =       69.4998461593527992    xSol[12] = 13.000
  xFind[13] =       -5.3174358606881889    xSol[13] = 14.000
  xFind[14] =       17.8712918147674280    xSol[14] = 15.000
-------------------------------------------------------------------
  Difference ||xFind-xSol|| = 1.576244e+02
  Matrix condition number   = 1.093605e+18


  Gaussian elimination with Pivoting: Test on Hilbert matrix
  -----------------------------------------------------------------
    xFind[ 0] =        0.9999991918077900    xSol[ 0] = 1.000
    xFind[ 1] =        2.0001249277493112    xSol[ 1] = 2.000
    xFind[ 2] =        2.9952484484944351    xSol[ 2] = 3.000
    xFind[ 3] =        4.0777268494653791    xSol[ 3] = 4.000
    xFind[ 4] =        4.3226197000758599    xSol[ 4] = 5.000
    xFind[ 5] =        9.4719300104842734    xSol[ 5] = 6.000
    xFind[ 6] =       -3.8007386526300087    xSol[ 6] = 7.000
    xFind[ 7] =       27.2887273081592667    xSol[ 7] = 8.000
    xFind[ 8] =       -3.5010962560664991    xSol[ 8] = 9.000
    xFind[ 9] =      -14.8476602753374092    xSol[ 9] = 10.000
    xFind[10] =       85.2304542983732745    xSol[10] = 11.000
    xFind[11] =      -78.8725258752507301    xSol[11] = 12.000
    xFind[12] =       75.1074313583280286    xSol[12] = 13.000
    xFind[13] =       -9.1625650579965416    xSol[13] = 14.000
    xFind[14] =       18.6903239834583417    xSol[14] = 15.000
  -----------------------------------------------------------------
    Difference ||xFind-xSol|| = 1.394645e+02
    Matrix condition number   = 1.093605e+18
```

```
 Gaussian elimination without Pivoting: Test on Hilbert matrix
 ----------------------------------------------------------------------
   xFind[ 0] =        1.0000010317094130    xSol[ 0] = 1.000
   xFind[ 1] =        1.9998887556526594    xSol[ 1] = 2.000
   xFind[ 2] =        3.0021009928829971    xSol[ 2] = 3.000
   xFind[ 3] =        4.0085988425267018    xSol[ 3] = 4.000
   xFind[ 4] =        4.4424997691237120    xSol[ 4] = 5.000
   xFind[ 5] =       12.2060355616599576    xSol[ 5] = 6.000
   xFind[ 6] =      -27.7161537285829205    xSol[ 6] = 7.000
   xFind[ 7] =      120.9522210550006207    xSol[ 7] = 8.000
   xFind[ 8] =     -208.7249621938958910    xSol[ 8] = 9.000
   xFind[ 9] =      239.5028969409272293    xSol[ 9] = 10.000
   xFind[10] =      -87.3620203978163943    xSol[10] = 11.000
   xFind[11] =       42.1651704198484083    xSol[11] = 12.000
   xFind[12] =     -139.2653692537922154    xSol[12] = 13.000
   xFind[13] =      190.7593118458560184    xSol[13] = 14.000
   xFind[14] =       17.4328989978599367    xSol[14] = 15.000
   xFind[15] =       59.5995476500012842    xSol[15] = 16.000
   xFind[16] =     -362.6584226944192437    xSol[16] = 17.000
   xFind[17] =      506.3974078266589913    xSol[17] = 18.000
   xFind[18] =     -239.0082781946647401    xSol[18] = 19.000
   xFind[19] =       71.2666275687167996    xSol[19] = 20.000
 ----------------------------------------------------------------------
   Difference ||xFind-xSol|| = 7.955297e+02
   Matrix condition number   = 7.921173e+18

Gaussian elimination with Pivoting: Test on Hilbert matrix
----------------------------------------------------------------------
   xFind[ 0] =        0.9999949194017077    xSol[ 0] = 1.000
   xFind[ 1] =        2.0008148016344971    xSol[ 1] = 2.000
   xFind[ 2] =        2.9677813798872674    xSol[ 2] = 3.000
   xFind[ 3] =        4.5465927308668830    xSol[ 3] = 4.000
   xFind[ 4] =        0.0890165677681846    xSol[ 4] = 5.000
   xFind[ 5] =       31.7360360261054701    xSol[ 5] = 6.000
   xFind[ 6] =      -74.3358484820292773    xSol[ 6] = 7.000
   xFind[ 7] =      158.5602479989587721    xSol[ 7] = 8.000
   xFind[ 8] =     -129.5619416428225463    xSol[ 8] = 9.000
   xFind[ 9] =       28.8530263278084043    xSol[ 9] = 10.000
   xFind[10] =       16.3696440881973899    xSol[10] = 11.000
   xFind[11] =      204.3007690928917270    xSol[11] = 12.000
   xFind[12] =     -306.3515629976349715    xSol[12] = 13.000
   xFind[13] =      192.5612425805877024    xSol[13] = 14.000
   xFind[14] =      -43.6160882438704931    xSol[14] = 15.000
   xFind[15] =       79.5303912783751059    xSol[15] = 16.000
   xFind[16] =       56.8649425004252009    xSol[16] = 17.000
   xFind[17] =     -154.3401287982326551    xSol[17] = 18.000
   xFind[18] =      151.8477000958695555    xSol[18] = 19.000
   xFind[19] =      -13.0226335365210151    xSol[19] = 20.000
----------------------------------------------------------------------
   Difference ||xFind-xSol|| = 5.271667e+02
   Matrix condition number   = 7.921173e+18
```

METODI ITERATIVI: IMPLEMENTAZIONE

Implementazione algoritmo di Gauss – Seidel

```
#Metodo iterativo di Gauss-Seidel
def GaussSeidel (matrix, b, x0, eps, k_max):
    #dimensione della matrice
    n = matrix.shape[0]

    #Variabile usata per interrompere il metodo
    stop = False

    #Variabile usata per contare il numero di iterazioni
    k = 0

    #Variabile per memorizzare la soluzione dell'iterazione corrente
    x1 = np.zeros(n)

    while not(stop) and k < k_max:
        for i in range(0, n):
            sum = 0

            for j in range(0, i):
                sum = sum + matrix[i][j] * x0[j]

            for j in range(i+1, n):
                sum = sum + matrix[i][j] * x1[j]

            x1[i] = (b[i] - sum) / matrix[i][i]

        #Controllo sui criteri di arresto

        #Calcolo del residuo
        res =  np.linalg.norm(b - np.dot(matrix, x1)) /
np.linalg.norm(b)

        #Calcolo della differenza tra due iterazioni successive
        diff_iter = np.linalg.norm(x1-x0)/np.linalg.norm(x1)

        stop = (res < eps ) and (diff_iter < eps)

        #Incremento il numero di iterazioni eseguite
        k = k + 1

        x0 = np.copy(x1)

    if not(stop):
        print('Process does not converge in %d iterations!' %k)

    return x1, k
```

Implementazione metodo di Jacobi

```
#Metodo iterativo di Jacobi
def Jacobi(matrix, b, x0, eps, k_max):
    #dimensione della matrice
    n = matrix.shape[0]

    #Variabile usata per interrompere il metodo
    stop = False

    #Variabile usata per contare il numero di iterazioni
    k = 0

    #Variabile per memorizzare la soluzione dell'iterazione corrente
    x1 = np.zeros(n)

    while not(stop) and k < k_max:
        for i in range(0, n):
            sum = 0

            for j in range(0, n):
                if j != i:
                    sum = sum + matrix[i][j] * x0[j]

            x1[i] = (b[i] - sum) / matrix[i][i]

        #Controllo sui criteri di arresto

        #Calcolo del residuo
        res =  np.linalg.norm(b - np.dot(matrix, x1)) /
np.linalg.norm(b)

        #Calcolo della differenza tra due iterazioni successive
        diff_iter = np.linalg.norm(x1-x0)/np.linalg.norm(x1)

        stop = (res < eps ) and (diff_iter < eps)

        #Incremento il numero di iterazioni eseguite
        k = k + 1

        x0 = np.copy(x1)

    if not(stop):
        print('Process does not converge in %d iterations!' %k)

    return x1, k
```

METODO DI GAUSS – SEIDEL: VERIFICA DELLA VARIAZIONE DEL NUMERO DI ITERAZIONI

Nel seguente esperimento vogliamo cronometrare i tempi di esecuzioni e il numero di iterazioni richieste al variare della dimensione della matrice.

La matrice che andiamo a studiare è

$$\begin{bmatrix} -C & 1 & 0 & \cdots & 0 & 0 \\ 1 & -C & 1 & \dots & 0 & 0 \\ 0 & 1 & -C & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -C & 1 \\ 0 & 0 & 0 & \cdots & 1 & -C \end{bmatrix}$$

In particolare, mostreremo come più è grande la costante posta sulla diagonale principale, più il numero di iterazioni richieste diminuirà.

In modo analogo, il numero di iterazioni aumenterà al diminuire della grandezza della costante usata.

Vogliamo inoltre mostrare come varia il tempo di esecuzione al variare della dimensione della matrice.

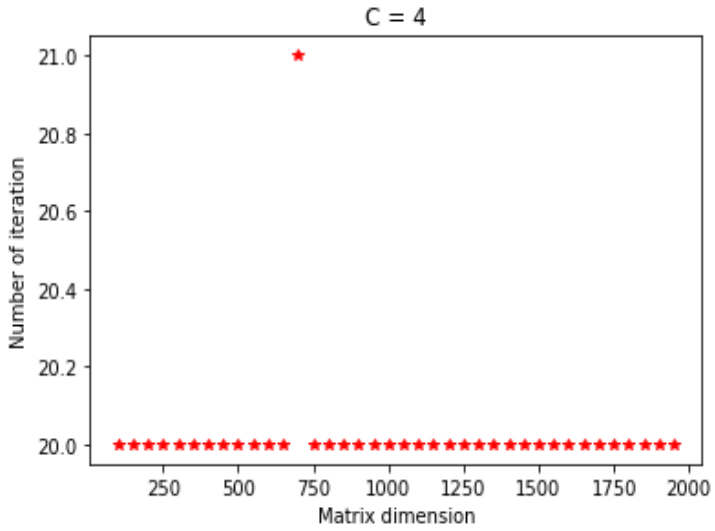
Sono forniti i seguenti dati del problema:

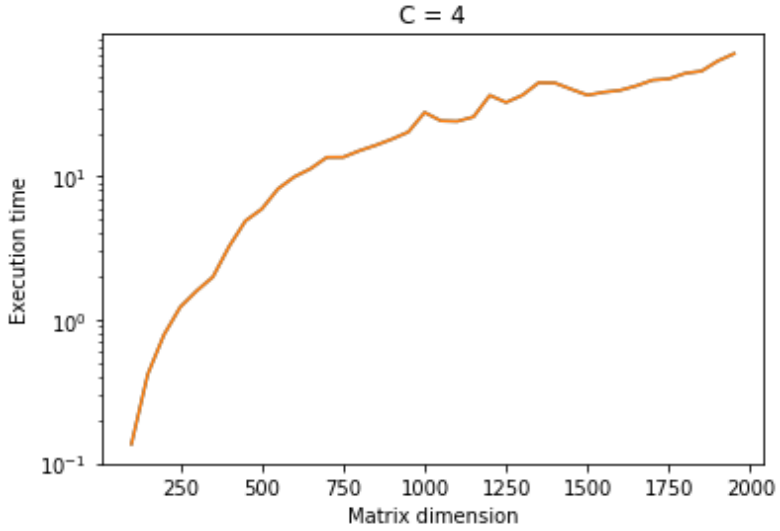
- Tolleranza: $\approx 10^{-6}$
- Numero massimo di iterazioni: 100
- Dimensione matrice : $100 - \ 2000$ con un incremento della dimensione pari a 50

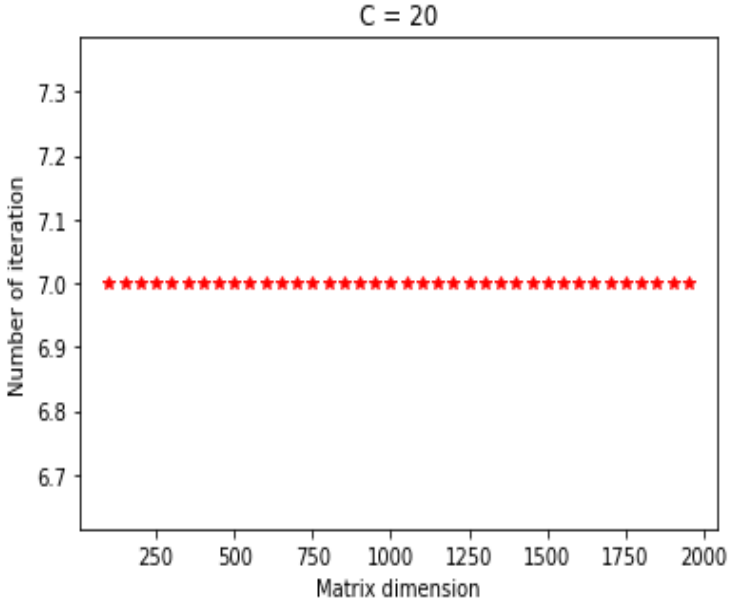
Implementazione Test

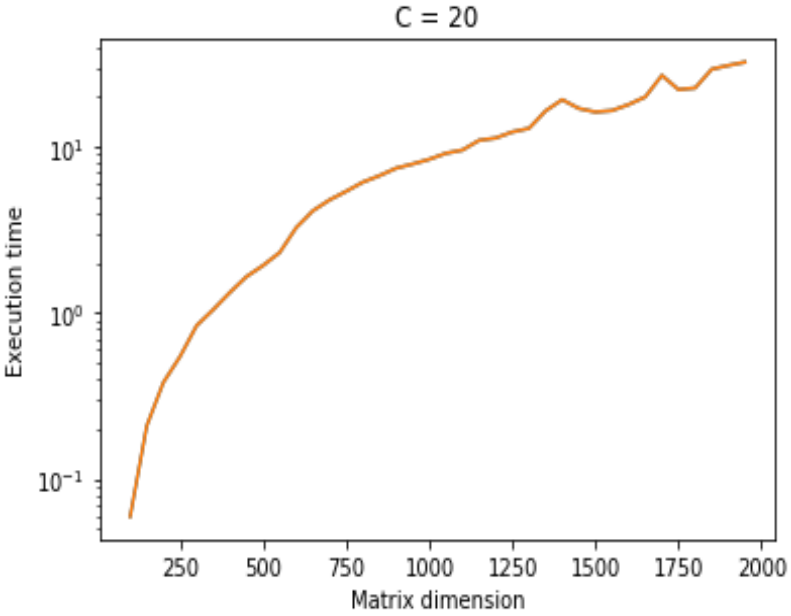
```
"""
Created on Tue Mar 24 22:27:17 2020

@author: Leonardo Saccotelli

"""
import numpy as np
import matplotlib.pyplot as plt
import time

#Numero massimo di iterazioni
k_max = 100
#Tolleranza dell'errore ammessa
eps = 1.0e-6

#Fisso un range della dimensione della matrice, in particolare
#partiamo da una matrice di 100 fino ad una matrice di 1000 al passo di
50
n_matrix = range(100, 2000, 50)

#Vettore utilizzato per memorizzare il numero di iterazioni eseguite al
#variare della dimensione della matrice
count_GS= np.zeros(len(n_matrix))

#Vettore utilizzato per memorizzare il tempo di esecuzione al
#variare della dimensione della matrice
executionTime = np.zeros(len(n_matrix))

#indice usato per la gestione dei vettore count_GS, executionTime
i = 0

for n in n_matrix:
    # ----- COSTRUZIONE DELLA MATRICE TEST
    #Costante da utilizzare sulla diagonale della matrice
    c = 4

    e1 = np.ones(n - 1)
    matrix = np.diag(e1, -1) - c * np.eye(n) + np.diag(e1, 1)

    #Fisso il vettore delle soluzioni
    xsol = np.ones(n)

    #Calcolo il vettore dei termini noti
    b = np.dot(matrix, xsol)

    #Fisso la prima approssimazione da utilizzare da Gauss - Seidel
    x0 = 2 * np.random.rand(n) - 1

    #Avvio il cronometro
    startGS = time.time()

    #Eseguo Gauss Seidel
    x_find, k = AL.GaussSeidel(matrix, b, x0, eps, k_max)
```

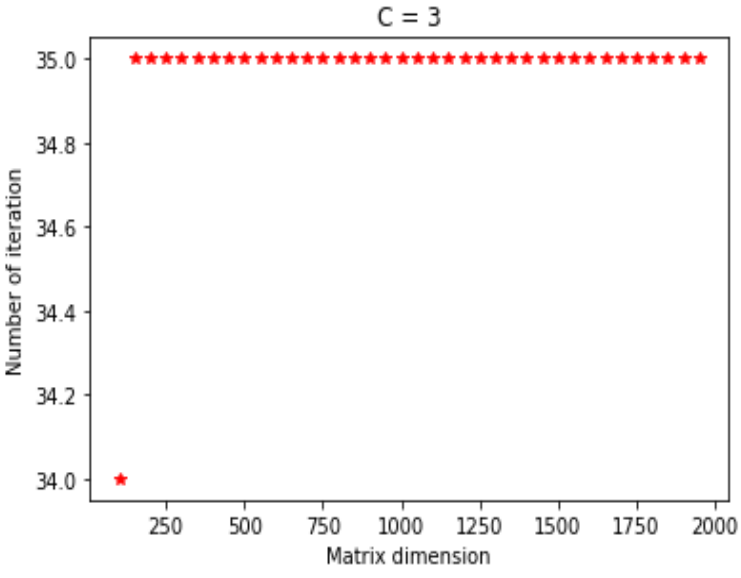
```
    #Fermo il cronometro
    stopGS = time.time()

    #Calcolo i tempi di esecuzione
    timeGS = (stopGS - startGS)

    #Salvo le informazioni
    count_GS[i] = k
    executionTime[i] = timeGS

    i = i + 1

#------ CREO IL GRAFICO DEL NUMERO DI ITERAZIONI
plt.figure(1)
plt.plot(n_matrix, count_GS, 'r*')
plt.xlabel('Matrix dimension');
plt.ylabel('Number of iteration')
plt.title('C = 4')

#----- CREO IL GRAFICO DEI TEMPI DI ESECUZIONE
plt.figure(2)
plt.plot(n_matrix, executionTime)
plt.semilogy(n_matrix, executionTime)
plt.xlabel('Matrix dimension');
plt.ylabel('Execution time')
plt.title('C = 4')
plt.show()
```

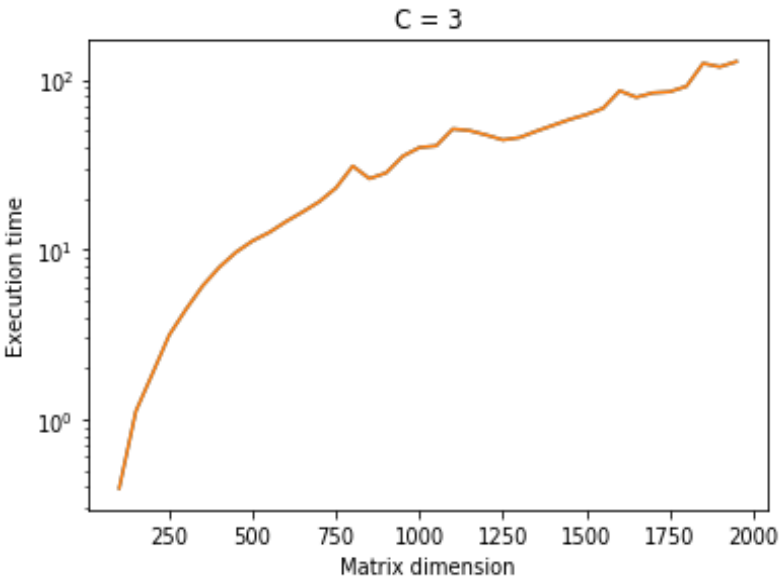

C = 4
Number of iteration
21.0
20.8
20.6
20.4
20.2
20.0
250
500
750
1000
1250
1500
1750
2000
Matrix dimension


C = 4
Execution time
10^1
10^0
10^{-1}
250
500
750
1000
1250
1500
1750
2000
Matrix dimension


C = 20
Number of iteration
7.3
7.2
7.1
7.0
6.9
6.8
6.7
250
500
750
1000
1250
1500
1750
2000
Matrix dimension


C = 20
Execution time
10^1
10^0
10^{-1}
250
500
750
1000
1250
1500
1750
2000
Matrix dimension


C = 3
Number of iteration
35.0
34.8
34.6
34.4
34.2
34.0
250
500
750
1000
1250
1500
1750
2000
Matrix dimension


C = 3
Execution time
10^2
10^1
10^0
250
500
750
1000
1250
1500
1750
2000
Matrix dimension

METODO DI JACOBI: VERIFICA DELLA VARIAZIONE DEL NUMERO DI ITERAZIONI

Nel seguente esperimento vogliamo cronometrare i tempi di esecuzioni e il numero di iterazioni richieste al variare della dimensione della matrice.

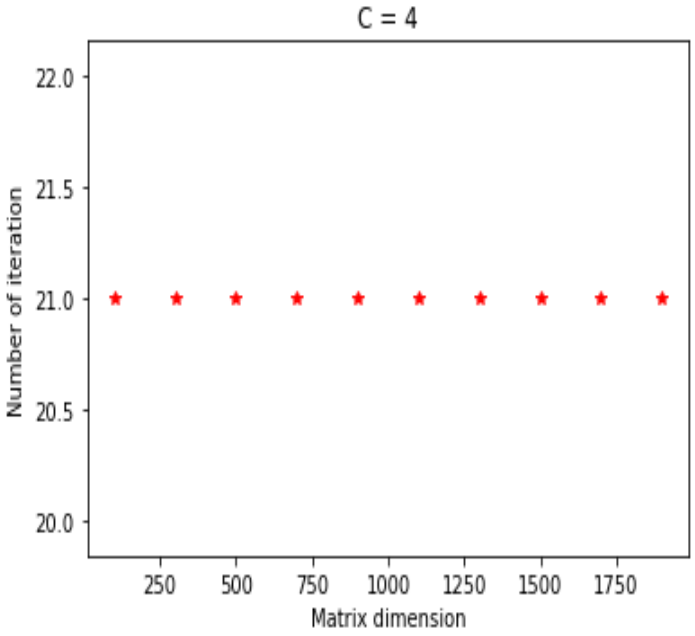
In modo analogo, il numero di iterazioni aumenterà al diminuire della grandezza della costante usata.

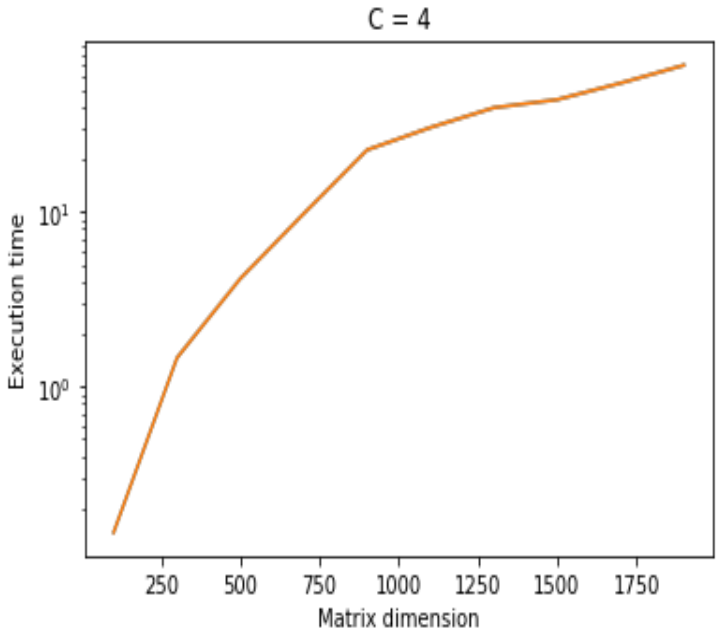
Vogliamo inoltre mostrare come varia il tempo di esecuzione al variare della dimensione della matrice.

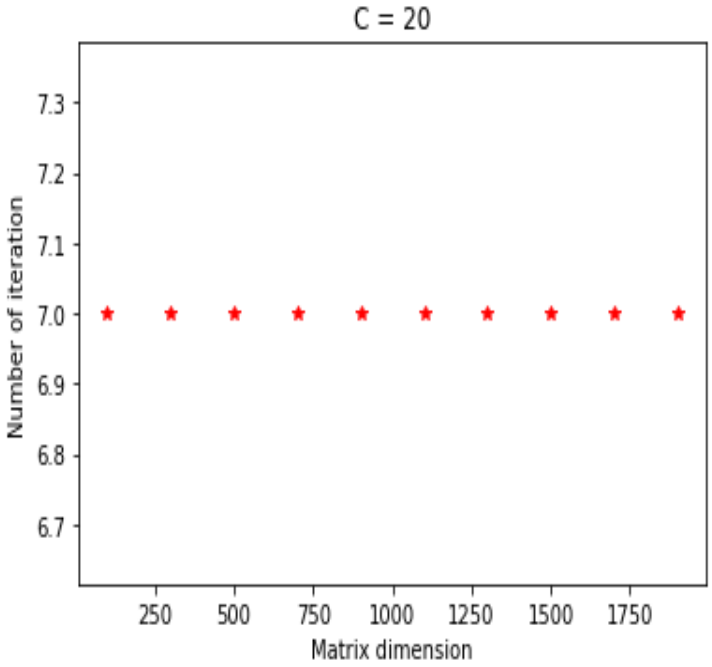
Sono forniti i seguenti dati del problema:

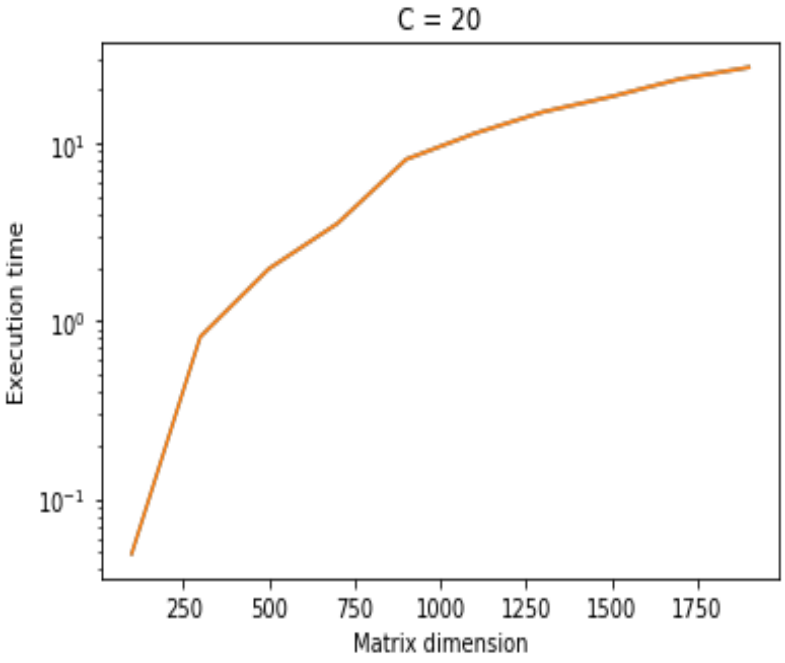
- Tolleranza: $\approx 10^{-6}$
- Numero massimo di iterazioni: 100
- Dimensione matrice: $100 - 2000$ con un incremento della dimensione pari a 200

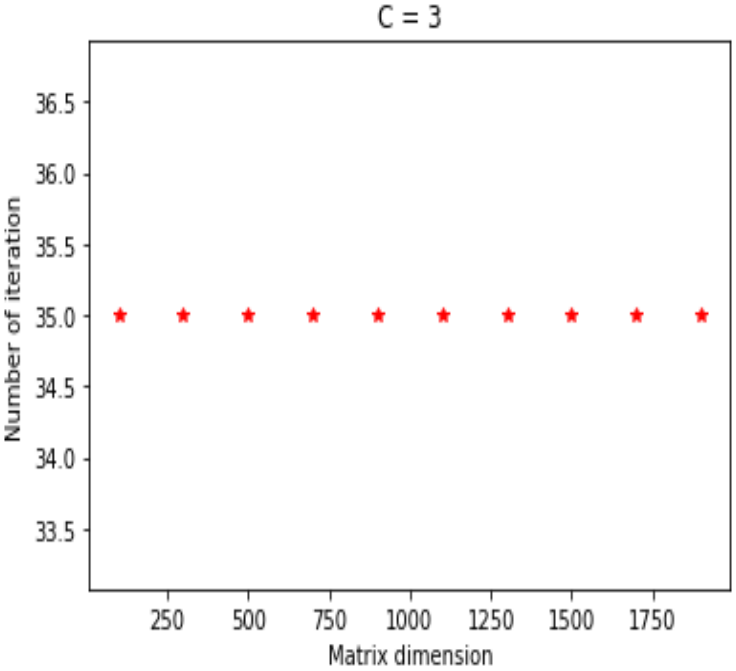
L'implementazione del test del metodo di Jacobi è sostanzialmente lo stesso del test del metodo di Gauss – Seidel.

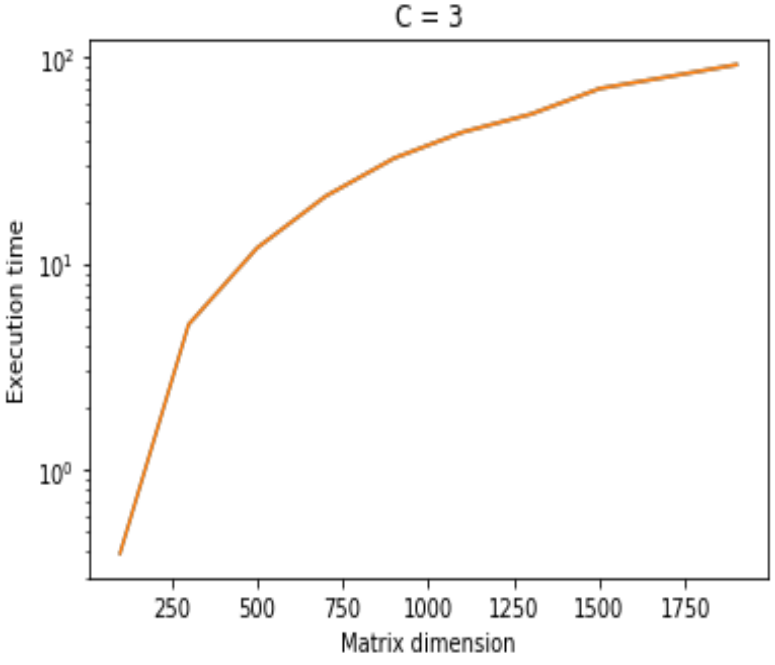





C = 20
Number of iteration
7.3
7.2
7.1
7.0
6.9
6.8
6.7
250
500
750
1000
1250
1500
1750
Matrix dimension


C = 20
Execution time
10^1
10^0
10^{-1}
250
500
750
1000
1250
1500
1750
Matrix dimension


C = 3
Number of iteration
36.5
36.0
35.5
35.0
34.5
34.0
33.5
250
500
750
1000
1250
1500
1750
Matrix dimension


C = 3
Execution time
10^2
10^1
10^0
250
500
750
1000
1250
1500
1750
Matrix dimension

CONFRONTO DEI TEMPI DI ESECUZIONE FRA METODI ITERATIVI

In questo esperimento vogliamo confrontare i tempi di esecuzione richiesti dagli algoritmi iterativi per risolvere un sistema lineare.

Il sistema lineare che andremo a risolvere è il seguente:

$$A = \begin{bmatrix} -50 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -50 & 1 & \ldots & 0 & 0 \\ 0 & 1 & -50 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & -50 & 1 \\ 0 & 0 & 0 & \cdots & 1 & -50 \end{bmatrix}$$

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ n \end{bmatrix}$$

Dopo aver calcolato b tramite il metodo della libreria numpy, utilizzeremo la matrice A e il vettore b per calcolare il vettore delle soluzioni x.

Nel calcolare il vettore delle soluzioni x utilizziamo i seguenti algoritmi:

- Metodo di Gauss – Seidel
- Metodo di Jacobi

Per ciascuno di questi metodi cronometriamo i tempi richiesti per risolvere il sistema lineare.

Ripetiamo l'esperimento facendo variare la dimensione della matrice. In particolare, consideriamo l' intervallo $[1000, 10000]$ incrementando di volta in volta la dimensione della matrice di 200. Mostriamo i seguenti risultati.

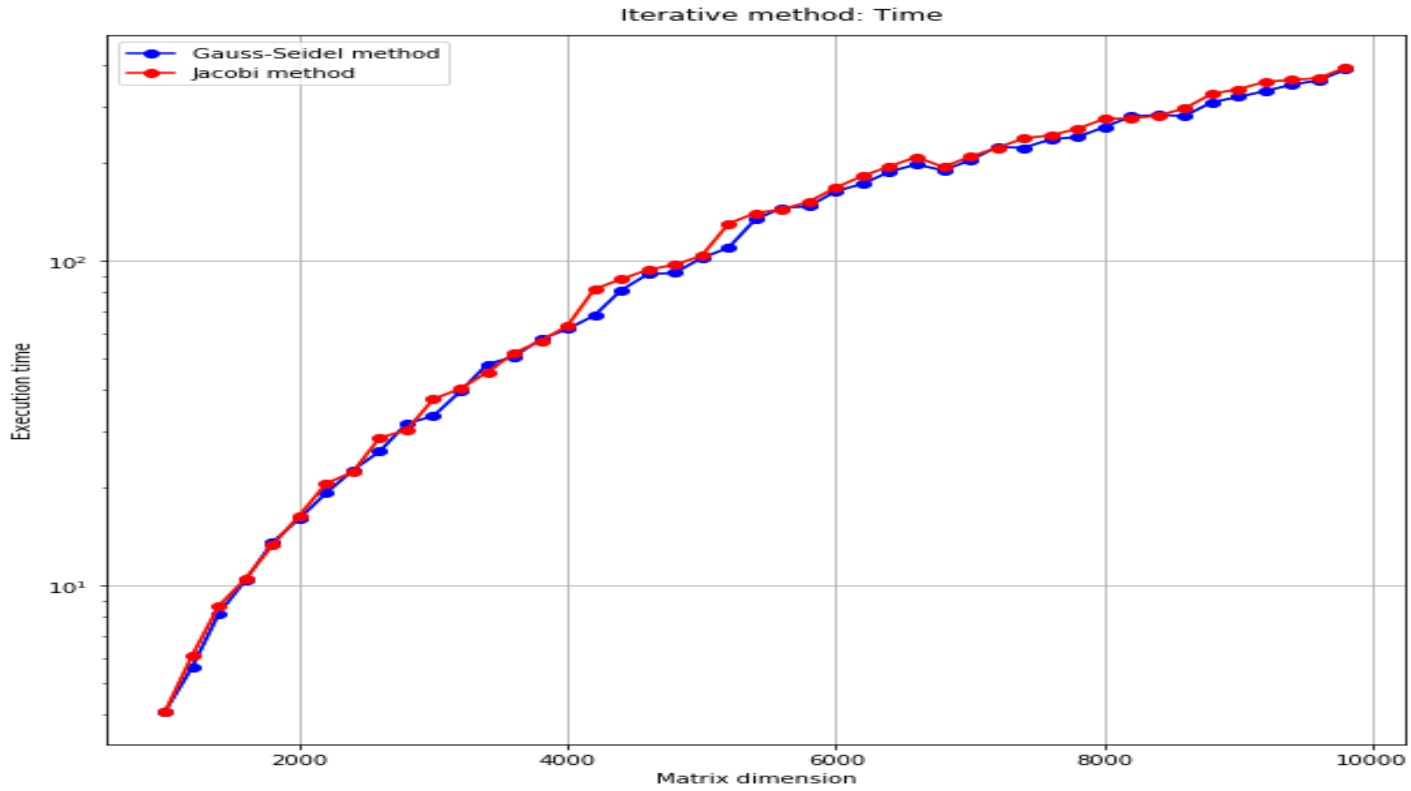
Implementazione del test.

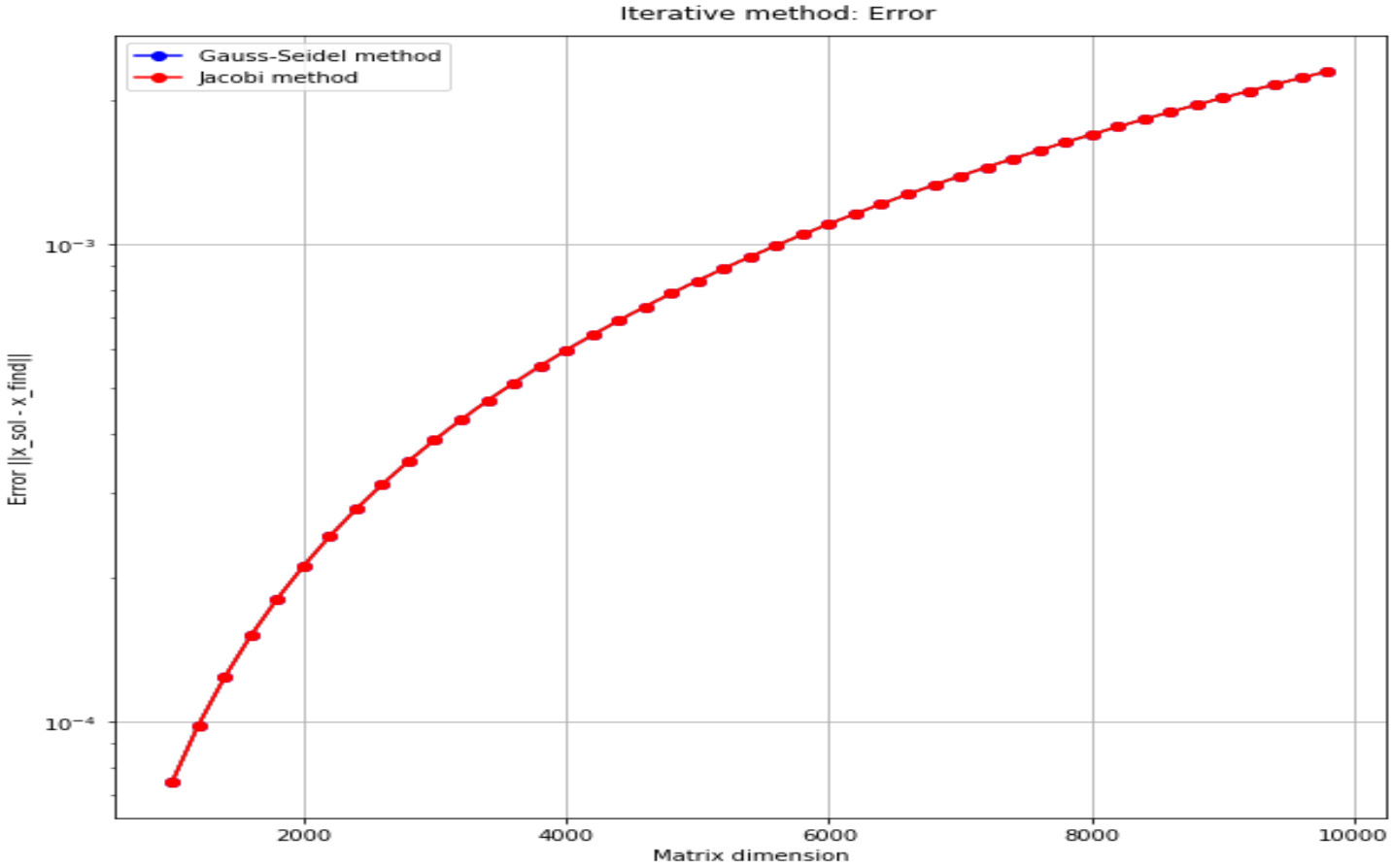
```
"""
Created on Wed Mar 25 22:12:46 2020

@author: Leonardo Saccotelli
"""

import numpy as np
import matplotlib.pyplot as plt
import time
import AlgoritmiAlgebraLineare as AL

def createMatrix(n):
    #Costante da utilizzare sulla diagonale della matrice
    c = 50
    e1 = np.ones(n - 1)
    matrix = np.diag(e1, -1) - c * np.eye(n) + np.diag(e1, 1)

    return matrix

#Funzione per cronometrare i tempi per l'esecuzione dell'algoritmo di
GaussSeidel
def timeGaussSeidel(matrix, b1, x0, eps, k_max, xSol):

    A = np.copy(matrix)

    b = np.copy(b1)

    #Avvio il cronometro
    start = time.time()

    xFind, k = AL.GaussSeidel(A, b, x0, eps, k_max)

    #Fermo il cronomtro
    end = time.time()

    #Calcolo il tempo di esecuzione
    timeDiff = end - start

    #Calcolo l'errore commesso
    errX = np.linalg.norm(xSol - xFind)

    return timeDiff, errX

#Funzione per cronometrare i tempi per l'esecuzione dell'algoritmo di
GaussSeidel
def timeJacobi(matrix, b1, x0, eps, k_max, xSol):
```

```
    A = np.copy(matrix)

    b = np.copy(b1)

    #Avvio il cronometro
    start = time.time()

    xFind, k = AL.Jacobi(A, b, x0, eps, k_max)

    #Fermo il cronomtro
    end = time.time()

    #Calcolo il tempo di esecuzione
    timeDiff = end - start

    #Calcolo l'errore commesso
    errX = np.linalg.norm(xSol - xFind)

    return timeDiff, errX

#Numero massimo di iterazioni
k_max = 100

#Tolleranza dell'errore ammessa
eps = 1.0e-6

#Fisso un range della dimensione della matrice, in particolare
#partiamo da una matrice di 1000 fino ad una matrice di 10000
#incrementando di 250 la dimensione
n_matrix = range(1000, 10000, 200 )

#Vettore utilizzato per memorizzare il tempo di esecuzione
#del metodo di Gauss-Seidel al variare della dimensione della matrice
executionGaussSeidelTime = np.zeros(len(n_matrix))

#Vettore utilizzato per memorizzare il tempo di esecuzione
#del metodo di Jacobi al variare della dimensione della matrice
executionJacobiTime = np.zeros(len(n_matrix))

#Vettore utilizzato per memorizzare l'errore commesso
#con l'algortimo di Gauss-Seidel
errorGaussSeidel = np.zeros(len(n_matrix))

#Vettore utilizzato per memorizzare l'errore commesso
#con l'algortimo di Jacobi
errorJacobi = np.zeros(len(n_matrix))


#indice usato per la gestione dei vettore dei tempi e degli errori
i = 0
for n in n_matrix:

    #Creo la matrice
    matrix = createMatrix(n)
```

```
    #Fisso la soluzione del sistema
    x_sol = np.array([i for i in range(1,n+1)])

    #Calcolo il vettore dei termini noti
    b = np.dot(matrix, x_sol)

    #Fisso la prima approssimazione da utilizzare
    #nei metodi iterativi
    x0 = 2 * np.random.rand(n) - 1

    #Registro il tempo di Gauss-Seidel
    executionGaussSeidelTime[i], errorGaussSeidel[i] =
timeGaussSeidel(matrix, b, x0, eps, k_max, x_sol)

    #Registro il tempo di Jacobi
    executionJacobiTime[i], errorJacobi[i] = timeJacobi(matrix, b, x0,
eps, k_max, x_sol)

    #Incremento l'indice dei cronometri
    i = i + 1


# ------- CREO I GRAFICI DEI CRONOMETRI
plt.figure(1, figsize = (10, 10))
plt.title('Direct method vs Iterative method: Time')

plt.grid(True)

plt.plot(n_matrix, executionGaussSeidelTime)
plt.semilogy(n_matrix, executionGaussSeidelTime,'bo-', label = 'Gauss-
Seidel method')

plt.plot(n_matrix, executionJacobiTime)
plt.semilogy(n_matrix, executionJacobiTime,'ro-', label = 'Jacobi
method')

plt.xlabel('Matrix dimension');
plt.ylabel('Execution time')
plt.legend(loc = 'best')


# ------------- CREO IL GRAFICO DELL'ERRORE
plt.figure(2, figsize=(10,10))
plt.title('Direct method vs Iterative method: Error')

#plt.plot(n_matrix, errorGauss)
#plt.loglog(n_matrix, errorGauss,'b', label = 'Gauss elimination')

#plt.plot(n_matrix, errorGaussPivoting)
#plt.loglog(n_matrix, errorGaussPivoting,'y', label = 'Gauss elimination
pivoting')

plt.grid(True)

plt.plot(n_matrix, errorGaussSeidel)
```

```
plt.semilogy(n_matrix, errorGaussSeidel, 'bo-', label = 'Gauss-Seidel
method')

plt.plot(n_matrix, errorJacobi)
plt.semilogy(n_matrix, errorJacobi,'ro-', label = 'Jacobi method'  )

plt.xlabel('Matrix dimension');
plt.ylabel('Error ||x_sol - x_find||')
plt.legend(loc = 'best')

plt.show()
```


Iterative method: Time
Gauss-Seidel method
Jacobi method
Execution time
10^2
10^1
2000
4000
6000
8000
10000
Matrix dimension


Iterative method: Error
Gauss-Seidel method
Jacobi method
Error ||x_sol - x_find||
10^{-3}
10^{-4}
2000
4000
6000
8000
10000
Matrix dimension

IMPLEMENTAZIONE METODI DI INTERPOLAZIONE

Implementazione metodo dei coefficienti indeterminati

```
"""-----------------------------------------------
    METODO DEI COEFFICIENTI INDETERMINATI
   -----------------------------------------------"""
"""
Created on Sat Mar 28 22:19:13 2020

@author: Leonardo Saccotelli
"""
import numpy as np
import matplotlib.pyplot as plt

#Funzione per la creazione della matrice di Vandermonde dei termini noti
def createVandermonde(value):

    #Calcolo la dimensione della matrice
    n = len(value)

    #Creo la matrice
    vandermondeMatrix = np.zeros((n,n))

    #Inizializzo la matrice
    for i in range(n):
        for j in range(n):
            vandermondeMatrix[i][j] = value[i]**j

    return vandermondeMatrix

#Funzione per il calcolo del polinomio di intepolazione con il metodo
#dei coefficienti indeterminati
def indeterminateCoefficientFormula(xNodes, yNodes, xPoints):
    #Creo la matrice di Vandermonde partendo da xNodes
    vandermondeMatrix = createVandermonde(xNodes).astype(float)

    #Risolvo il sistema lineare dato dalla matrice di Vandermonde e il
vettore dei
    #termini noti yNodes per trovare i coefficienti
    coefficient  = np.linalg.solve(vandermondeMatrix, yNodes)

    #Calcolo il polinomio di interpolazione sui coefficienti non
perturbati
    #nei punti xPoints
    interpolationPolynomial = np.polyval(np.flipud(coefficient),
xPoints)

    return interpolationPolynomial
```

Implementazione prima formula baricentrica di Lagrange

```
#Funzione per calcolare il polinomio di Lagrange (formula 1) in un
insieme di punti x
def lagrangeFirstBarycentricFormula(xNodes, yNodes, xPoints):
    # Cacolo dei coefficienti formula baricentrica
    z = Z_Coeff(xNodes, yNodes)

    # Calcolo poliniomio di interpolazione nei punti di x
    p = np.zeros(len(xPoints))

    for i in range(len(xPoints)):
        p[i] = ComputeLagrange_1(xPoints[i], xNodes, z, yNodes)

    return p


# Routine per il cacolo coefficienti Zj formula baricentrica
def Z_Coeff(xNodes, yNodes):
    #Determino il numero di nodi di interpolazione
    n = len(xNodes)

    #Creo la matrice identità di dimensione n x n
    X = np.eye(n)

    #Calcolo PROD[(Xj - Xk)] PER K = 0 .... N, CON K != J
    for i in range(n):
        for j in range(n):
            #Calcolo le differenze (Xj - Xk) con j != k
            if j > i:
                X[i, j] = xNodes[i] - xNodes[j]
            elif j < i:
                X[i, j] = -X[j, i]
    #Creo il vettore Z per memorizzare i coeff Zj
    z = np.zeros(n)
    for j in range(n):
        #Calcolo i coefficienti Zj = Yj / PROD[(Xj - Xk)] PER K = 0 ....
N, CON K != J
        z[j] = yNodes[j] / np.prod(X[j, :])
    return z

#Calcolo del polinomio in un singolo punto x
def ComputeLagrange_1(xPoints, xNodes, z, yNodes):
    #Verifico che la distanza fra il punto x in cui voglio calcolare il
polinomio e i
    #valori dei nodi usati sia minore di 10^-14.
    #Se questo è vero, vuol dire che stiamo calcolando il polinomio in
uno dei nodi
    #di interpolazione.
    #In questo caso è noto il valore assunto dal polinomio nei nodi di
interpolazione.
    check_nodes = abs(xPoints - xNodes) < 1.0e-14
    if True in check_nodes:
        #Salvo il punto x che coincide con il nodo
        temp = np.where(check_nodes == True)
        #Salvo il suo valore in una variabile i
```

```
        i = temp[0][0]
        #Uso i come indice per ricavare il valore associato al xNodes
        #in cui vogliamo calcolare il polinomio, valore che è già noto.
        p = yNodes[i]
    else:
        #Ricavo il numero di nodi
        n = len(xNodes)
        S = 0
        #Calcolo la sommatoria per j = 0...n di (Zj/(x-xNodes[j])
        for j in range(n):
            S = S + z[j] / (xPoints - xNodes[j])
        #Moltiplico la sommatoria per PROD(x - xNodes) k = 0 ... n
        p = np.prod(xPoints - xNodes) * S
    return p
```

Implementazione seconda formula baricentrica di Lagrange

```
#Funzione per calcolare il polinomio di Lagrange (formula 2) in un
insieme di punti x
def lagrangeSecondBarycentricFormula(xNodes, yNodes, xPoints):
    # Cacolo dei coefficienti formula baricentrica
    lambdaCoeff = Lambda_Coeff(xNodes)

    # Calcolo poliniomio di interpolazione nei punti di x
    p = np.zeros(len(xPoints))

    for i in range(len(xPoints)):
        p[i] = ComputeLagrange_2(xPoints[i], xNodes, lambdaCoeff,
yNodes)

    return p

#Routine per il cacolo coefficienti LambdaJ formula baricentrica
def Lambda_Coeff(xNodes):
    #Determino il numero di nodi di interpolazione
    n = len(xNodes)

    #Creo la matrice identità di dimensione n x n
    X = np.eye(n)

    #Calcolo PROD[(Xj - Xk)] PER K = 0 .... N, CON K != J
    for i in range(n):
        for j in range(n):
            #Calcolo le differenze (Xj - Xk) con j != k
            if j > i:
                X[i, j] = xNodes[i] - xNodes[j]
            elif j < i:
                X[i, j] = -X[j, i]
    #Creo il vettore lambda per memorizzare i coeff Lambdaj
    Lambda_j = np.zeros(n)
    for j in range(n):
        #Calcolo i coefficienti Zj = Yj / PROD[(Xj - Xk)] PER K = 0 ....
N, CON K != J
        Lambda_j[j] = 1 / np.prod(X[j, :])
    return Lambda_j
```

```
#Calcolo del polinomio in un singolo punto x
def ComputeLagrange_2(xPoints, xNodes, lam, yNodes):
    #Verifico che la distanza fra il punto x in cui voglio calcolare il
polinomio e i valori dei nodi usati
    #sia minore di 10^-14. Se questo è vero, vuol dire che stiamo
calcolando il polinomio in uno dei nodi
    #di interpolazione. In questo caso è noto il valore assunto dal
polinomio nei nodi di interpolazione.
    check_nodes = abs(xPoints - xNodes) < 1.0e-14
    if True in check_nodes:
        #Salvo il punto x che coincide con il nodo
        temp = np.where(check_nodes == True)
        #Salvo il suo valore in una variabile i
        i = temp[0][0]
        #Uso i come indice per ricavare il valore associato al xNodes
        #in cui vogliamo calcolare il polinomio, valore che è già noto.
        p = yNodes[i]
    else:
        #Ricavo il numero di nodi
        n = len(xNodes)
        S_Num = 0
        S_Den = 0
        #Calcolo la sommatoria per j = 0...n di (Zj/(x-xNodes[j])
        for j in range(n):
            S_Num = S_Num + ((lam[j] / (xPoints - xNodes[j]))*yNodes[j])
            S_Den = S_Den + (lam[j] / (xPoints - xNodes[j]))
        p = S_Num/S_Den
    return p
```

Implementazione formula di Newton alle differenze divise

```
#Algorimto per il calcolo delle differenze divise di f sui nodi x
def differenceDivision (x, y):
    #Creo una copia dei nodi di interpolazione
    #e dei valori associati ai nodi di interpolazione
    xNodes = np.copy(x)
    diagonalCoeff = np.copy(y)

    #Determino il numero dei nodi
    n = len(xNodes)
    #Parto dalla prima colonna sino all'ultima colonna
    for j in range(0, n):
        #Parto dall'ultima riga sino all'elemento sulla diagonale
        for i in range(n-1, j, -1):
            #Calcolo la differenza divisa
            diagonalCoeff[i] = (diagonalCoeff[i]- diagonalCoeff[i-
1])/(xNodes[i]-xNodes[i-j-1])
    return diagonalCoeff


#Funzione per il calcolo del Polinomio di interpolazione di Newton
#alle differenze divise in un insieme di punti x
def NewtonPolynomialInterpolation(xNodes, yNodes, xPoints):
```

```
    #Determino il numero di punti in cui calcolare la funzione
    n = len(xPoints)

    #Determino i coeffiecienti del polinomio di interpolazione
    #attraverso le differenze divise
    coeff = differenceDivision(xNodes, yNodes)

    #Creo il vettore che conterrà il
    #valore assunto dal polinomio nei punti xPoints
    p = np.zeros(n)

    #Calcolo il valore assunto dal polinomio in ogni punto
    for i in range(len(xPoints)):
        p[i] = ComputeNewton(xNodes, coeff, xPoints[i])

    return p

#Funzione per calcolare il polinomio in un singolo punto x
def ComputeNewton(xNodes, coeff, xPoint):
    #Numero di nodi
    n = len(xNodes)
    #Salvo l'ultima differenza divisa
    p = coeff[n-1]
    #Calcolo il polinomio nel punto x
   for i in range(n-2, -1, -1):
        p = p *(xPoint - xNodes[i]) + coeff[i]
    return p
```

INTERPOLAZIONE POLINOMIALE: METODO DEI COEFFICIENTI INDETERMINATI

Obiettivo del seguente esperimento è quello di costruire il polinomio di interpolazione $\Phi(x)$ che approssimi la funzione $f(x) = e^x$.

Nella costruzione del polinomio utilizzeremo il metodo dei coefficienti indeterminati. Fissati $n+1$ nodi d'interpolazione equidistanti, estrapolati dall'intervallo $[0, 10]$,

$$\Phi(x) = \sum_{j=0}^{n} a_j x^j$$

Siano inoltre dati i seguenti dati:

- $n = 30$
- $x_{0,} x_1, \dots, x_n$ nodi di interpolazione;
- $y_0, y_1, \dots, y_n$ valori associati ai nodi di interpolazione;
- $\gamma_0, \gamma_1, \dots, \gamma_n$ valori perturbati associati ai nodi di interpolazione.

Risolvendo i sistemi lineari

$$\Phi(x_i) = \sum_{j=0}^{n} a_j {x_i}^j = y_i \quad i = 0, 1, \dots, n$$

$$\mathbb{P}(x_i) = \sum_{j=0}^{n} a_j {x_i}^j = \gamma_i \quad i = 0, 1, \dots, n$$

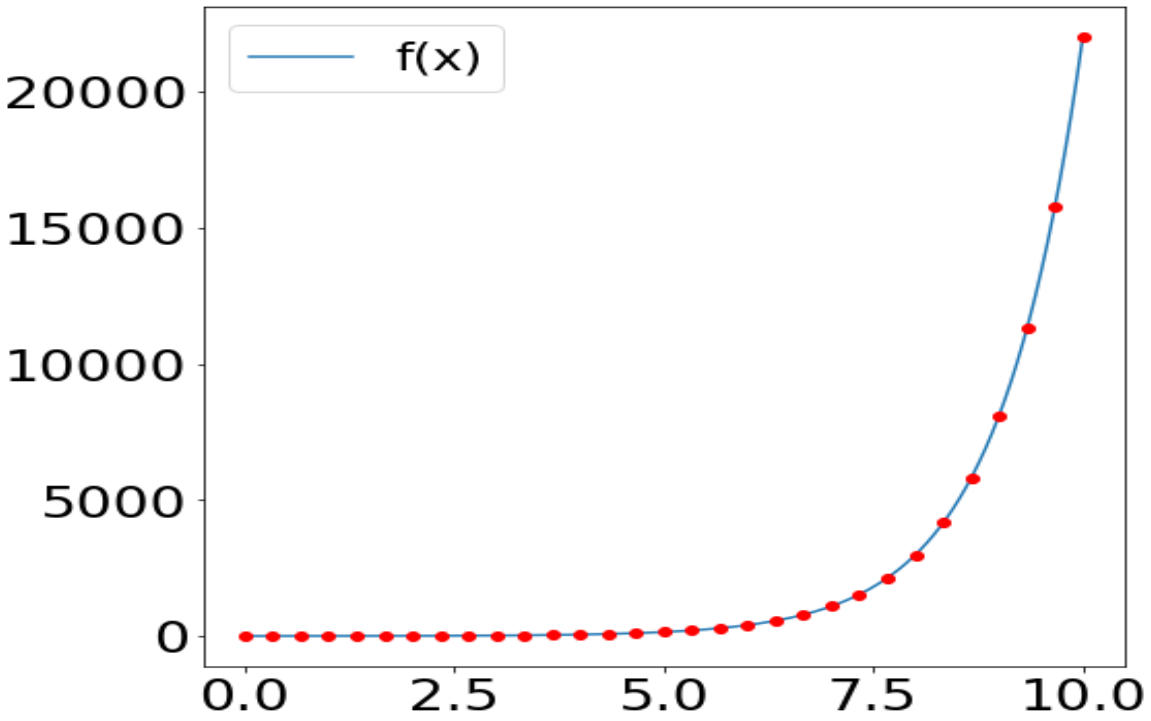
troveremo rispettivamente due vettori di coefficienti, esatti e perturbati, per la costruzione del polinomio di interpolazione.

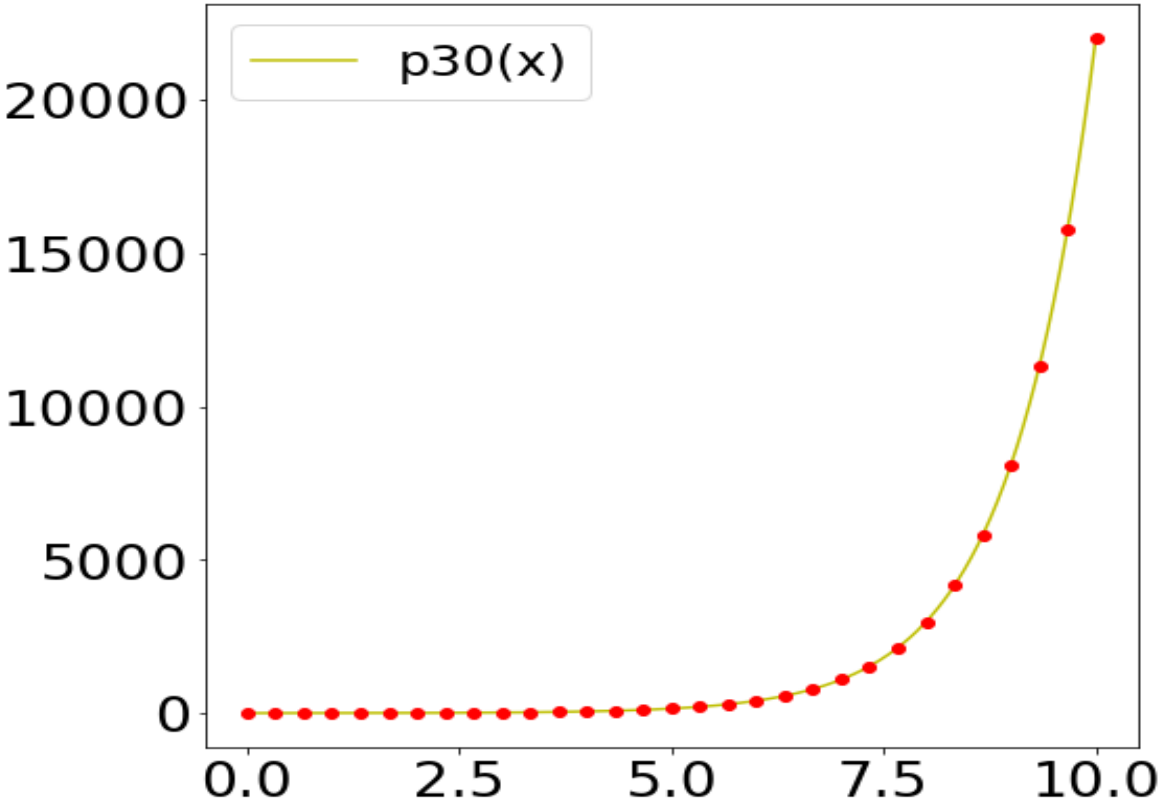
A questo punto sarà sufficiente prendere un insieme di punti in cui calcolare i polinomi di interpolazione. Confronteremo pertanto i risultati con la funzione $f(x) = e^x$ che vogliamo approssimare.

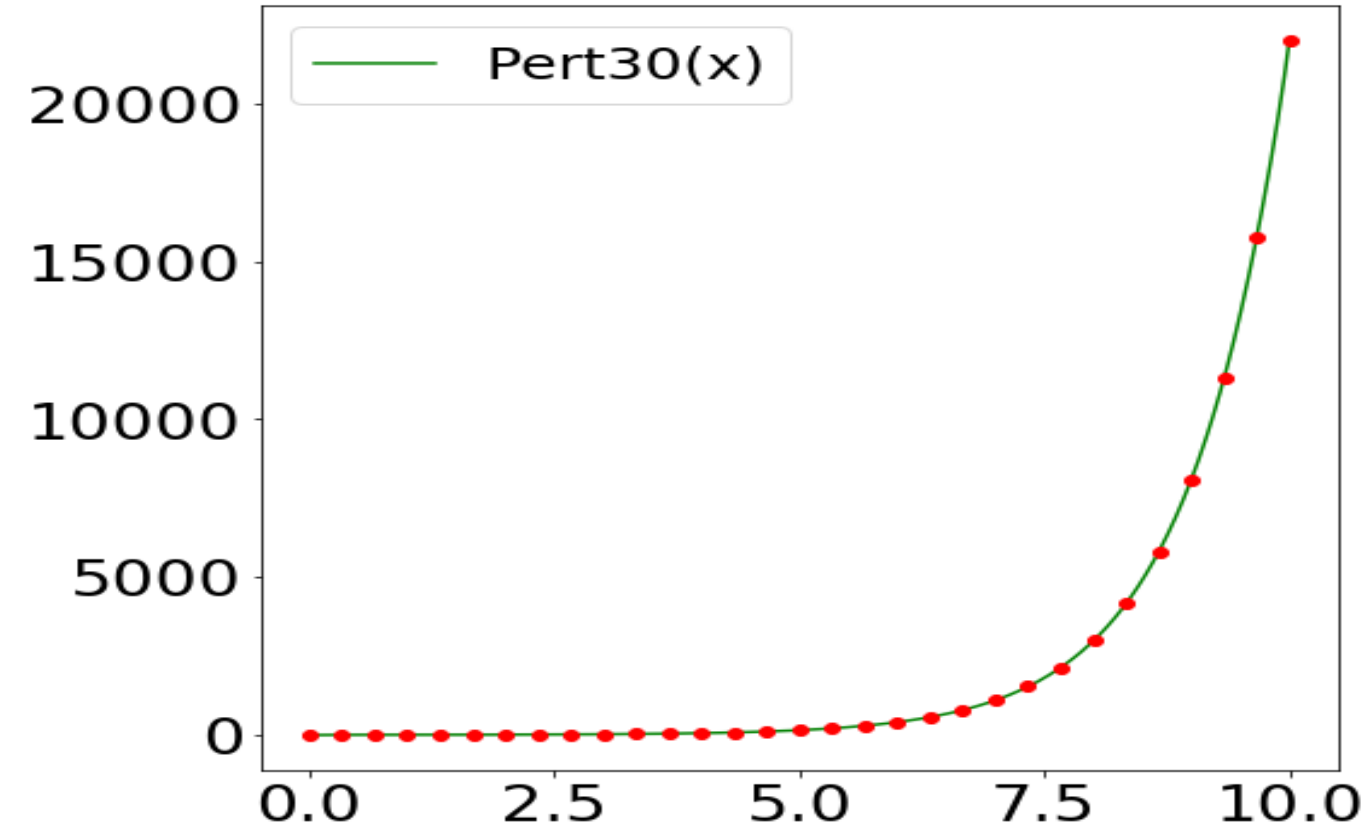
Implementazione test

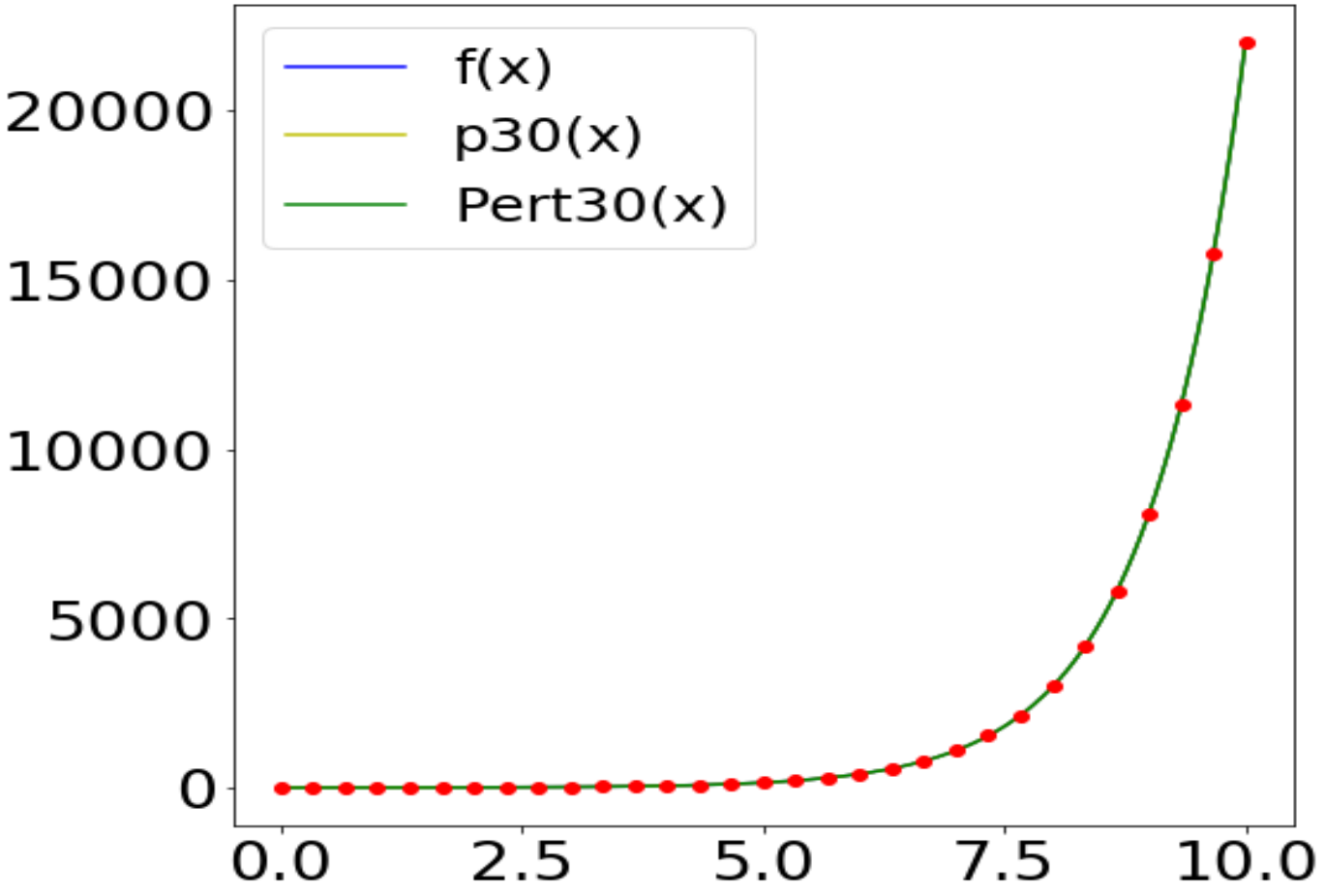
```
#Funzione per il calcolo di f(x) = cos(x)
def createExp(x):
    y = np.exp(x)
    return y


#-------------------------------------------     DATI DI
INTERPOLAZIONE
#Estremo inferiore dell'intervallo
a = 0
#Estremo superiore dell'intervallo
b = 6
#Numero di nodi da estrapolare dall'intervallo [a,b]
n = 50

#Estrapolo n+1 nodi equidistanti dall'intervallo [a,b]
xNodes = np.linspace(a,b,n+1)

#Calolo la funzione f(x)=cos(x) in tutti i nodi xNodes
yNodes = createExp(xNodes)

#Ordine di grandezza della perturbazione
perturbationGrades = 1.0e-3

#Sommo le perturbazioni ai valori di yNodes (Perturbo le ordinate)
perturbationYNodes = yNodes + (np.random.rand(n+1)*2-
1)*perturbationGrades

#-------------------------------------------------------------------------
#------------------------------- METODO DEI COEFFICIENTI INDETERMINATI
#------------------------------- COSTRUZIONE E CALCOLO DEL POLINOMIO


#Estrapoliamo 10*(n+1) + 1 punti equidistanti in cui calcolare il
polinomio
xPoints = np.linspace(a, b, 50*len(xNodes) + 1)

#Calcolo il polinomio di interpolazione sui coefficienti non perturbati
nei punti xPoints
interpolationPol = IP.indeterminateCoefficientFormula(xNodes, yNodes,
xPoints)

#Calcolo il polinomio di interpolazione sui coefficienti perturbati nei
punti xPoints
pertInterpolationPol = IP.indeterminateCoefficientFormula(xNodes,
perturbationYNodes, xPoints)
```

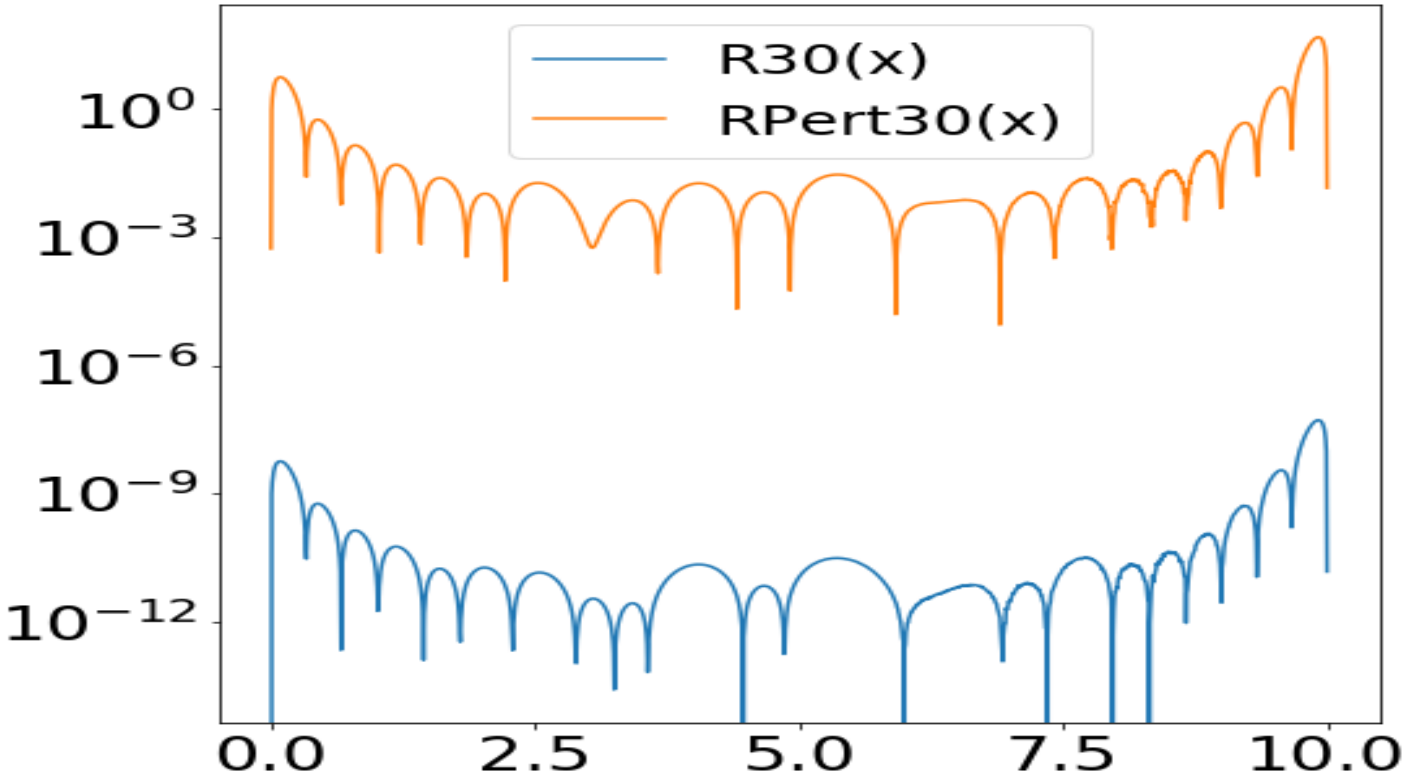
```
#Calcoliamo f(x) = cos(x) nei punti xPoints
f_xPoints = createExp(xPoints)

#---------------------------------Rappresentazione grafica dei risultati
plt.figure(0,figsize=(8,8))
#Stampo il grafico di f(x) = cos(x)
plt.plot(xPoints, f_xPoints, label = 'f(x)')


p_lab = 'p%d(x)' % n
#Stampo il grafico del polinomio di interpolazione con coeff non
perturbati
plt.plot(xPoints, interpolationPol, label = p_lab)

p_lab = 'Pert%d(x)' % n
#Stampo il grafico del polinomio di interpolazione con coeff perturbati
plt.plot(xPoints, pertInterpolationPol, label=p_lab)

#Stampo i nodi di interpolazione
plt.plot(xNodes, yNodes,'ro')
plt.legend(prop={'size': 25})


#-------------------------------------Rappresentazione grafica del
resto
plt.figure(1,figsize=(8,8))
r_lab = 'R%d(x)' % n
#Stampo l'errore commesso nell'approssimare F con il polinomio con coeff
non perturbati
plt.semilogy(xPoints, abs(f_xPoints - interpolationPol), label=r_lab)


r_lab = 'RPert%d(x)' % n
#Stampo l'errore commesso nell'approssimare F con il polinomio con coeff
perturbati
plt.semilogy(xPoints, abs(f_xPoints - pertInterpolationPol),label=r_lab)

plt.legend(prop={'size': 25})
plt.rcParams.update({'font.size': 28})
plt.show()
```


f(x)
20000
15000
10000
5000
0
0.0
2.5
5.0
7.5
10.0


p30(x)
20000
15000
10000
5000
0
0.0
2.5
5.0
7.5
10.0


Pert30(x)
20000
15000
10000
5000
0
0.0
2.5
5.0
7.5
10.0


f(x)
p30(x)
Pert30(x)
20000
15000
10000
5000
0
0.0
2.5
5.0
7.5
10.0



Alcune considerazioni sui risultati ottenuti. Notiamo come il polinomio $\Phi(x)$ dia una buona approssimazione della funzione $f(x) = e^x$.

Dal grafico possiamo notare come l'errore $f(x) - \Phi(x)$ ha un ordine di grandezza compreso tra 10^{-16} e 10^{-7}.

Notiamo invece come $\mathbb{P}(x)$ dia un'approssimazione peggiore di $f(x)$.

Le ragioni sono fondamentalmente due:

- la perturbazione sui valori associati ai nodi di interpolazione
- il forte mal condizionamento della matrice di Vandermonde, matrice dei coefficienti, che amplifica notevolmente gli errori sugli input.

L'errore $f(x) - \mathbb{P}(x) \gg f(x) - \Phi(x)$.

INTERPOLAZIONE POLINOMIALE: CONFRONTO DEI METODI STUDIATI E ANALISI DELL'ERRORE

Obiettivo del seguente esperimento è quello di confrontare i seguenti metodi per determinare il polinomio di interpolazione:

- Metodo dei coefficienti indeterminati
- Polinomio di interpolazione di Lagrange (prima formulazione baricentrica)
- Polinomio di interpolazione di Lagrange (seconda formulazione baricentrica)
- Polinomio di interpolazione di Newton alle differenze divise.

Studieremo la funzione $f(x) = \sqrt{x}$ nell'intervallo $[0, 1]$.

Per la costruzione del polinomio di interpolazione useremo un numero crescente di nodi equidistanti.

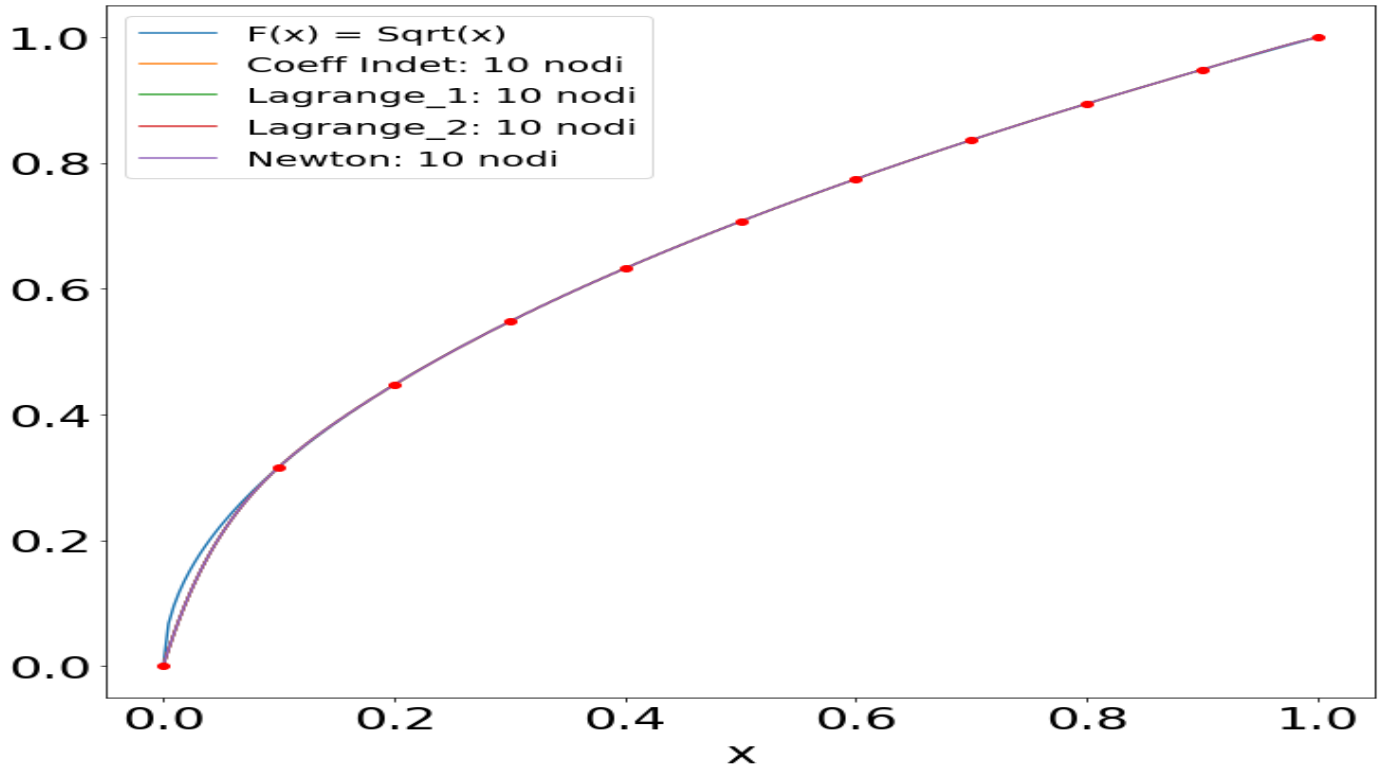
Mostreremo come il resto di interpolazione varia aumentando i nodi di interpolazione.

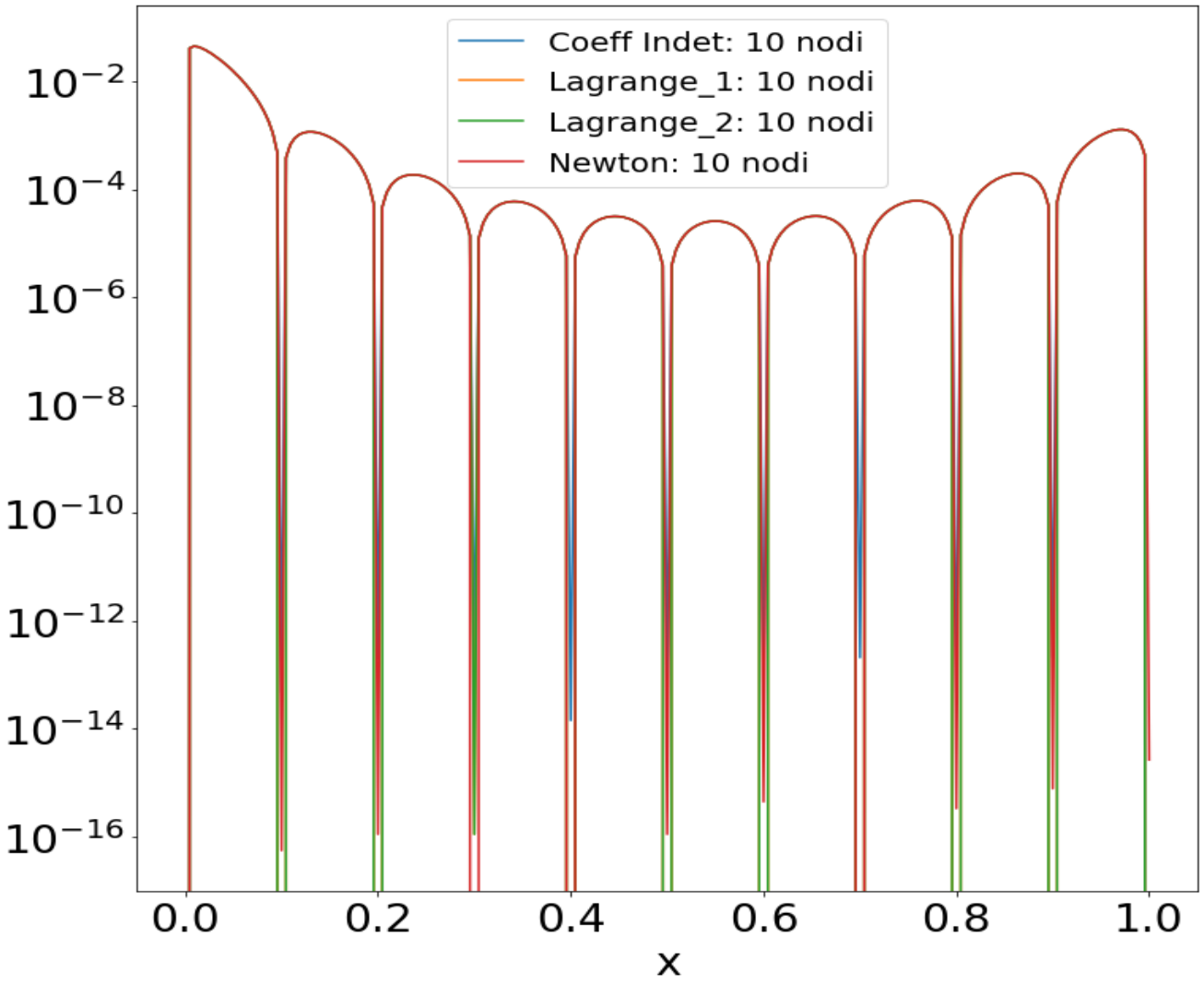
Implementazione test

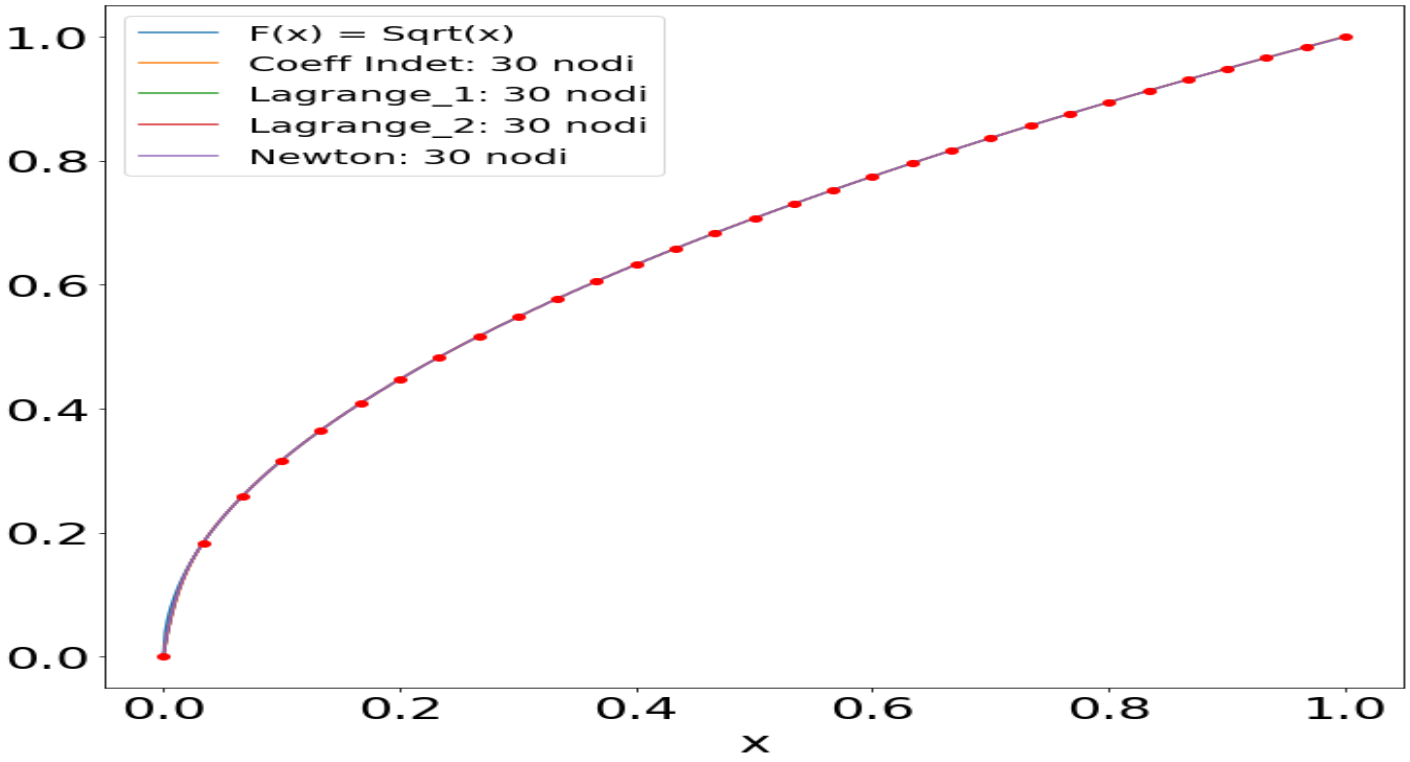
```
import numpy as np
import matplotlib.pyplot as plt
import InterpolazionePolinomiale_Algoritmi as IP

def createMyFunction(x):
    y = np.sqrt(x)
    return y

#----------------  DEFINIZIONI DATI DI INTERPOLAZIONE
#Estremo inferiore dell'intervallo da cui estrapolo i nodi
a = 0
#Estremo superiore dell'intervallo da cui estrapolo i nodi
b = 1

#Numero di nodi da estrapolare dall'intervallo [a,b]
n = 50
#Estrapolo n+1 nodi equidistanti dall'intervallo [a,b]
xNodes = np.linspace(a, b, n + 1)
#Calcolo la funzione f(x) = sin(x) negli n+1 nodi
yNodes = createMyFunction(xNodes)
```

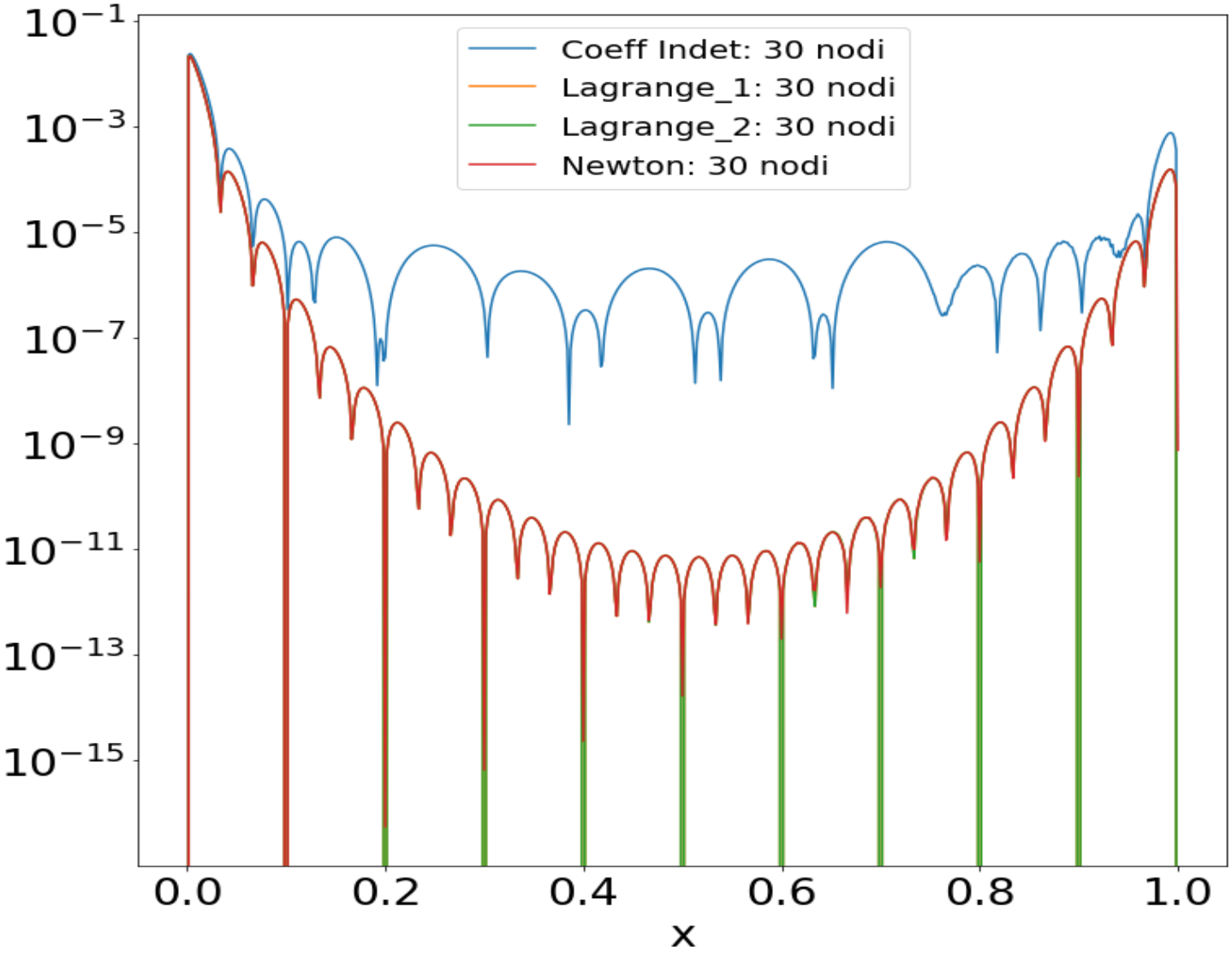
```
#-------------- COSTRUZIONE POLINOMIO DI INTERPOLAZIONE
#Estrapolo i punti in cui calcolare il polinomio
xPoints = np.linspace(a, b, 20 * len(xNodes) + 1)

#Creo e calcolo il polinomio con il metodo dei coefficienti
indeterminati
pol_coeff_indet = IP.indeterminateCoefficientFormula(xNodes, yNodes,
xPoints)

#Creo e calcolo il polinomio con la prima formula baricentrica di
Lagrange in ogni punto xPoint
pol_1_Lagrange = IP.lagrangeFirstBarycentricFormula(xNodes, yNodes,
xPoints)

#Creo e calcolo il polinomio con la seconda formula baricentrica di
Lagrange in ogni punto xPoint
pol_2_Lagrange = IP.lagrangeSecondBarycentricFormula(xNodes, yNodes,
xPoints)

#Creo e calcolo il polinomio con la formula di Newton in ogni punto
xPoint
pol_Newton = IP.NewtonPolynomialInterpolation(xNodes, yNodes, xPoints)

#Calcolo la funzione in ogni punto xPoints
f_xPoints = createMyFunction(xPoints)

#------------- Grafico del polinomio di interpolazione
plt.figure(1, figsize=(12, 12))
plt.plot(xPoints, f_xPoints, label='F(x) = Sqrt(x)')

p_lab = 'Coeff Indet: %d nodi' % n
plt.plot(xPoints, pol_coeff_indet, label=p_lab)

p_lab = 'Lagrange_1: %d nodi' % n
plt.plot(xPoints, pol_1_Lagrange, label=p_lab)

p_lab = 'Lagrange_2: %d nodi' % n
plt.plot(xPoints, pol_2_Lagrange, label=p_lab)

p_lab = 'Newton: %d nodi' % n
plt.plot(xPoints, pol_Newton, label=p_lab)

plt.plot(xNodes, yNodes, 'ro')
plt.xlabel('x')
plt.legend(prop={'size': 20}, loc ='best')

#----------------Grafico del resto del polinomio di interpolazione
plt.figure(2, figsize=(12, 12))

r_lab = 'Coeff Indet: %d nodi' % n
plt.semilogy(xPoints, abs(f_xPoints - pol_coeff_indet), label=r_lab)

r_lab = 'Lagrange_1: %d nodi' % n
plt.semilogy(xPoints, abs(f_xPoints - pol_1_Lagrange), label=r_lab)
```

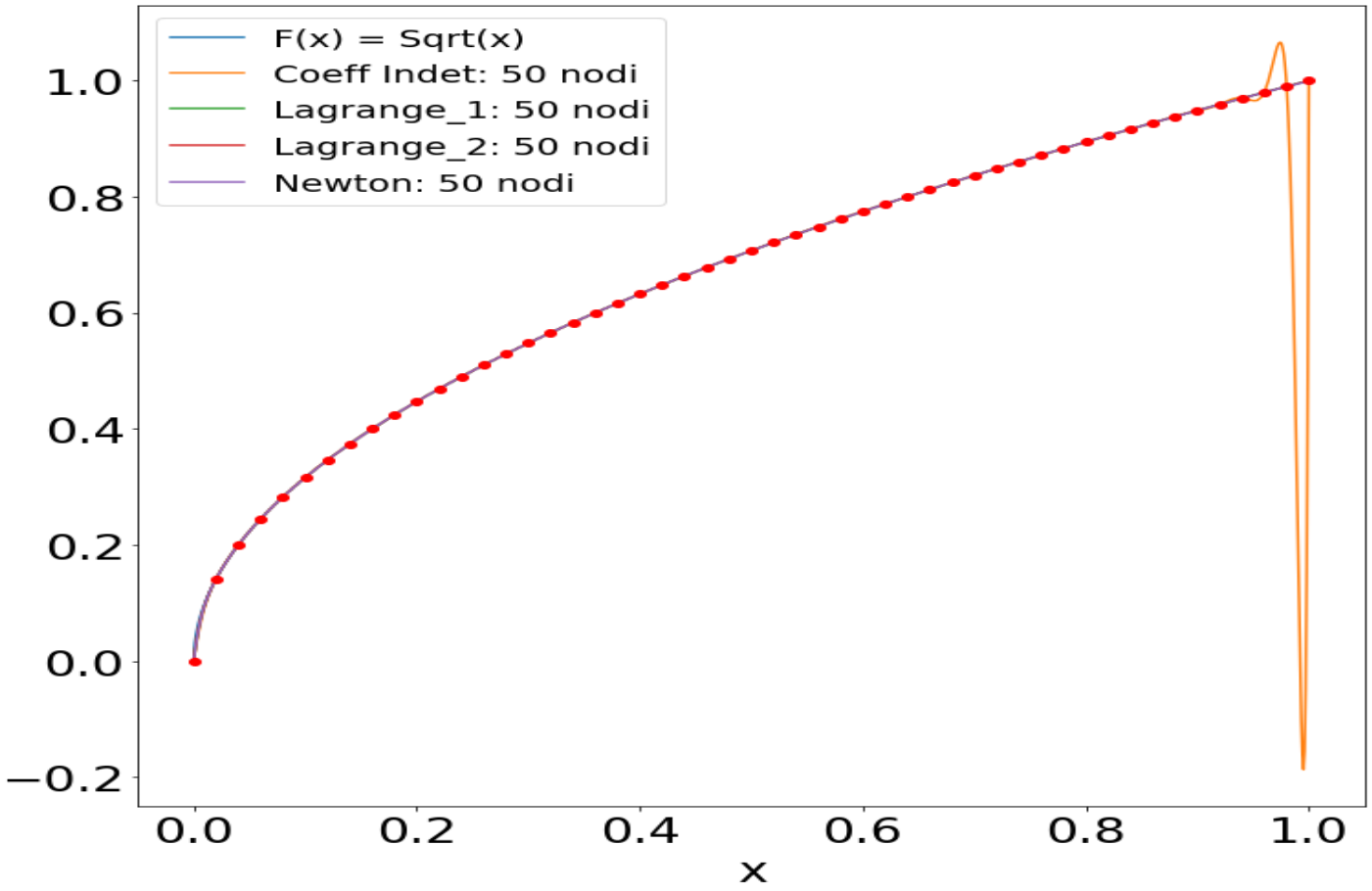
```
r_lab = 'Lagrange_2: %d nodi' % n
plt.semilogy(xPoints, abs(f_xPoints - pol_2_Lagrange), label=r_lab)

r_lab = 'Newton: %d nodi' % n
plt.semilogy(xPoints, abs(f_xPoints - pol_Newton), label=r_lab)

plt.xlabel('x')

plt.legend(prop={'size': 20}, loc ='best')

plt.show()
```

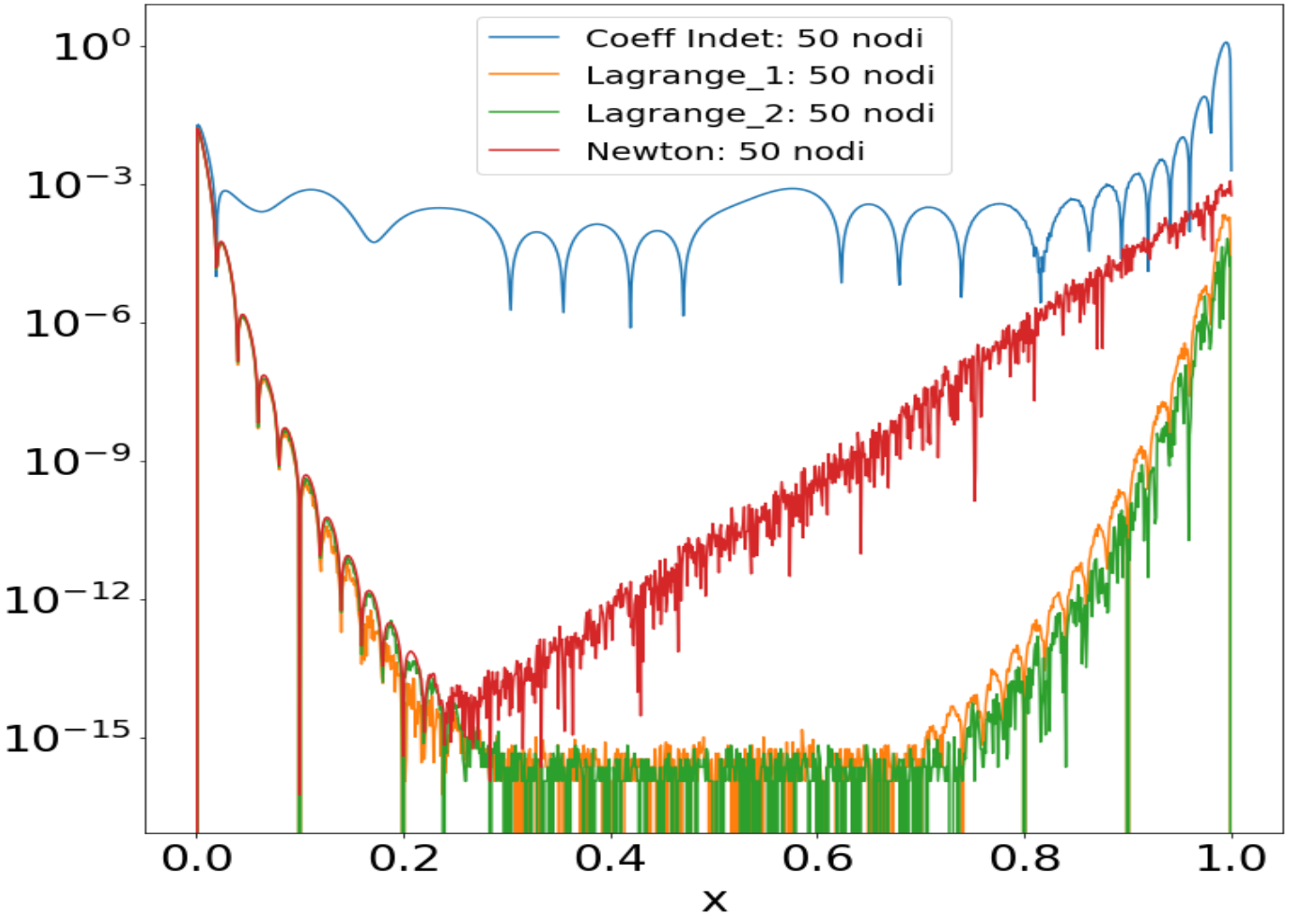



Coeff Indet: 10 nodi
Lagrange_1: 10 nodi
Lagrange_2: 10 nodi
Newton: 10 nodi
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-16}
0.0
0.2
0.4
0.6
0.8
1.0
x


F(x) = Sqrt(x)
Coeff Indet: 30 nodi
Lagrange_1: 30 nodi
Lagrange_2: 30 nodi
Newton: 30 nodi
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
x


Coeff Indet: 30 nodi
Lagrange_1: 30 nodi
Lagrange_2: 30 nodi
Newton: 30 nodi
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
0.0
0.2
0.4
0.6
0.8
1.0
x


F(x) = Sqrt(x)
Coeff Indet: 50 nodi
Lagrange_1: 50 nodi
Lagrange_2: 50 nodi
Newton: 50 nodi
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
x


Coeff Indet: 50 nodi
Lagrange_1: 50 nodi
Lagrange_2: 50 nodi
Newton: 50 nodi
10^{0}
10^{-3}
10^{-6}
10^{-9}
10^{-12}
10^{-15}
0.0
0.2
0.4
0.6
0.8
1.0
x

METODO DI NEWTON ALLE DIFFERENZE DIVISE: CONFRONTO TRA I NODI EQUIDISTANTI E I NODI DI CHEBYSHEV

Nel seguente esperimento vogliamo studiare il fenomeno di Runge.

Data la funzione $f(x) = \frac{1}{1+x^2}$ in $[-1, 1]$ costruiremo il polinomio di interpolazione attraverso il metodo di Newton alle differenze divise. Nella costruzione utilizzeremo prima i nodi equidistanti, successivamente i nodi di Chebyshev.

Nel corso dell'esperimento aumenteremo il numero di nodi di interpolazione utilizzati, mostrando graficamente come questo si ripercuote sull'approssimazione che si vuole dare di $f(x)$.

Analizzando il grafico dell'errore di interpolazione notiamo come questo sia di un ordine di grandezza maggiore agli estremi dell'intervallo, mentre tende ad attenuarsi nella parte interna dell'intervallo quando utilizziamo i nodi equidistanti. Notiamo anche come aumentando il numero dei nodi, l'ordine di grandezza dell'errore cresca notevolmente agli estremi.

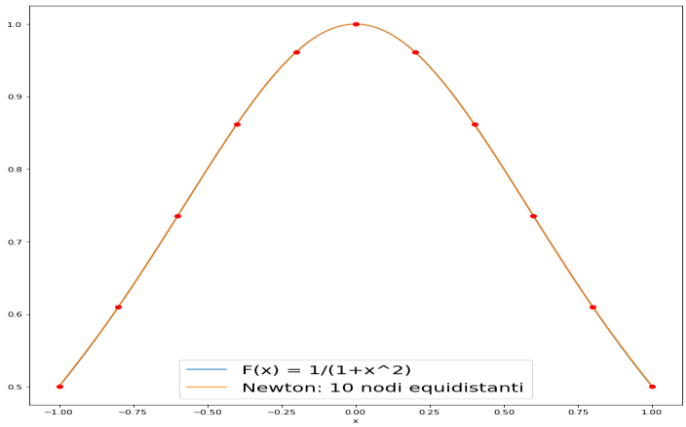
Nell'utilizzo dei nodi di Chebyshev invece riscontriamo una maggiore uniformità nella distribuzione dell'errore di interpolazione su tutto l'intervallo; inoltre aumentando i nodi questa uniformità tende a conservarsi.

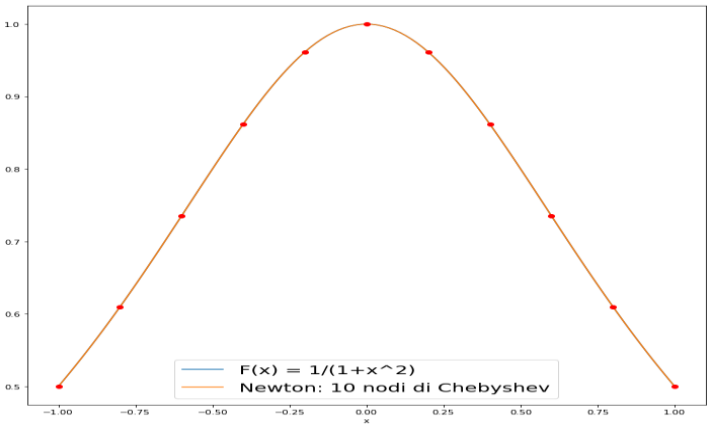
Implementazione del test

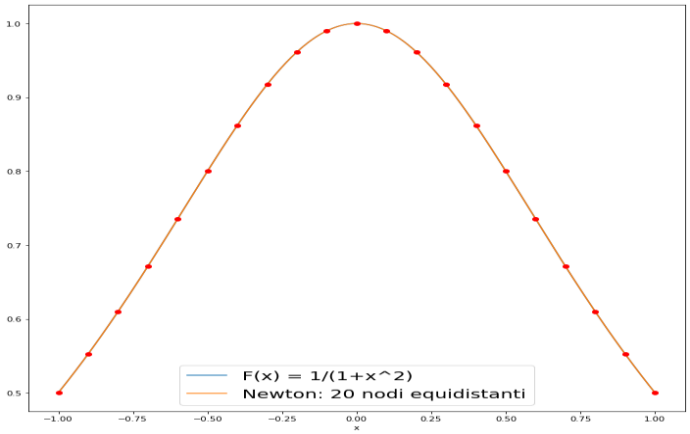
```
"""
Created on Wed Apr  1 22:30:12 2020

@author: Leonardo Saccotelli
"""
import numpy as np
import matplotlib.pyplot as plt
```

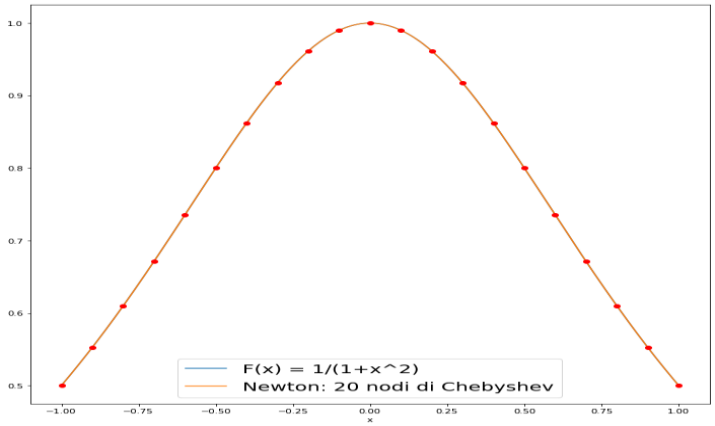
```
import InterpolazionePolinomiale_Algoritmi as IP

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

#----------------  DEFINIZIONI DATI DI INTERPOLAZIONE
#Estremo inferiore dell'intervallo da cui estrapolo i nodi
a = -1
#Estremo superiore dell'intervallo da cui estrapolo i nodi
b = 1

#Numero di nodi da estrapolare dall'intervallo [a,b]
n = 10

#Estrapolo n+1 nodi equidistanti dall'intervallo [a,b]
xEquidistantNodes = np.linspace(a, b, n + 1)

#Calcolo la funzione f(x) negli n+1 nodi equidistanti
yEquidistantNodes = createMyFunction(xEquidistantNodes)

#Estrapolo n+1 nodi di Chebyshev dall'intervallo [a,b]
xChebyshevNodes = IP.createChebyshevNodes(a, b, n)

#Calcolo la funzione f(x) negli n+1 nodi di Chebyshev
yChebyshevNodes = createMyFunction(xChebyshevNodes)

#Estrapolo i punti in cui calcolare il polinomio
xPoints = np.linspace(a, b, 10 * len(xEquidistantNodes) + 1)

#Creo e calcolo il polinomio con la formula di Newton in ogni punto
xPoint
pol_Equi_Newton = IP.NewtonPolynomialInterpolation(xEquidistantNodes,
yEquidistantNodes, xPoints)

#Creo e calcolo il polinomio con la formula di Newton in ogni punto
xPoint
pol_Chebyshev_Newton = IP.NewtonPolynomialInterpolation(xChebyshevNodes,
yChebyshevNodes, xPoints)

#Calcolo la funzione in ogni punto xPoints
f_xPoints = createMyFunction(xPoints)


#------------- Grafico del polinomio di interpolazione con nodi
equidistanti
plt.figure(1, figsize=(12, 12))
plt.plot(xPoints, f_xPoints, label='F(x) = 1/(1+x^2)')

p_lab = 'Newton: %d nodi equidistanti' % n
plt.plot(xPoints, pol_Equi_Newton, label=p_lab)

plt.plot(xEquidistantNodes, yEquidistantNodes, 'ro')
plt.xlabel('x')
plt.legend(prop={'size': 20}, loc ='best')
```

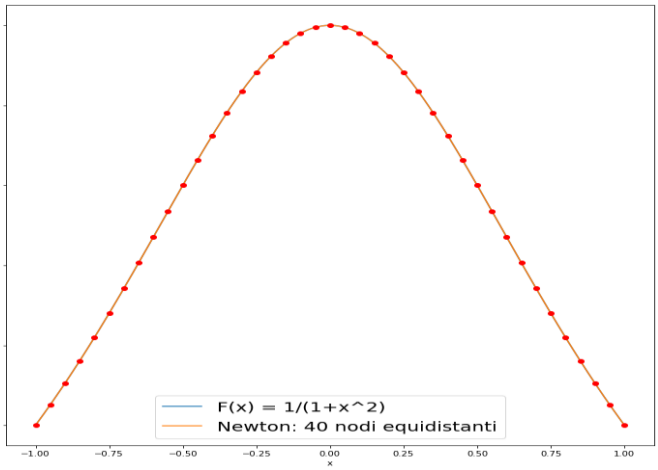
```
#------------- Grafico del polinomio di interpolazione con nodi di
Chebyshev
plt.figure(2, figsize=(12, 12))
plt.plot(xPoints, f_xPoints, label='F(x) = 1/(1+x^2)')

p_lab = 'Newton: %d nodi di Chebyshev' % n
plt.plot(xPoints, pol_Chebyshev_Newton, label=p_lab)

plt.plot(xEquidistantNodes, yEquidistantNodes, 'ro')
plt.xlabel('x')
plt.legend(prop={'size': 20}, loc ='best')


#----------------Grafico del resto del polinomio di interpolazione
plt.figure(3, figsize=(12, 12))
r_lab = 'Nodi Equidistanti: %d nodi' % n
plt.semilogy(xPoints, abs(f_xPoints - pol_Equi_Newton), label=r_lab)

r_lab = 'Nodi Chebyshev: %d nodi' % n
plt.semilogy(xPoints, abs(f_xPoints - pol_Chebyshev_Newton),
label=r_lab)

plt.xlabel('x')
plt.legend(prop={'size': 20}, loc ='best')
plt.show()
```

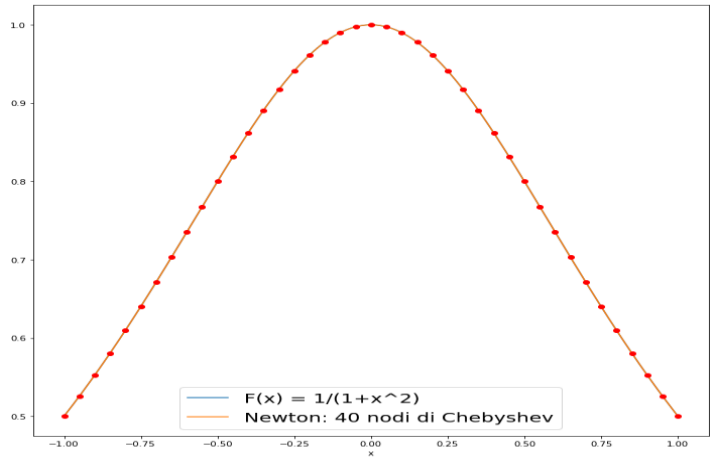

F(x) = 1/(1+x^2)
Newton: 10 nodi equidistanti

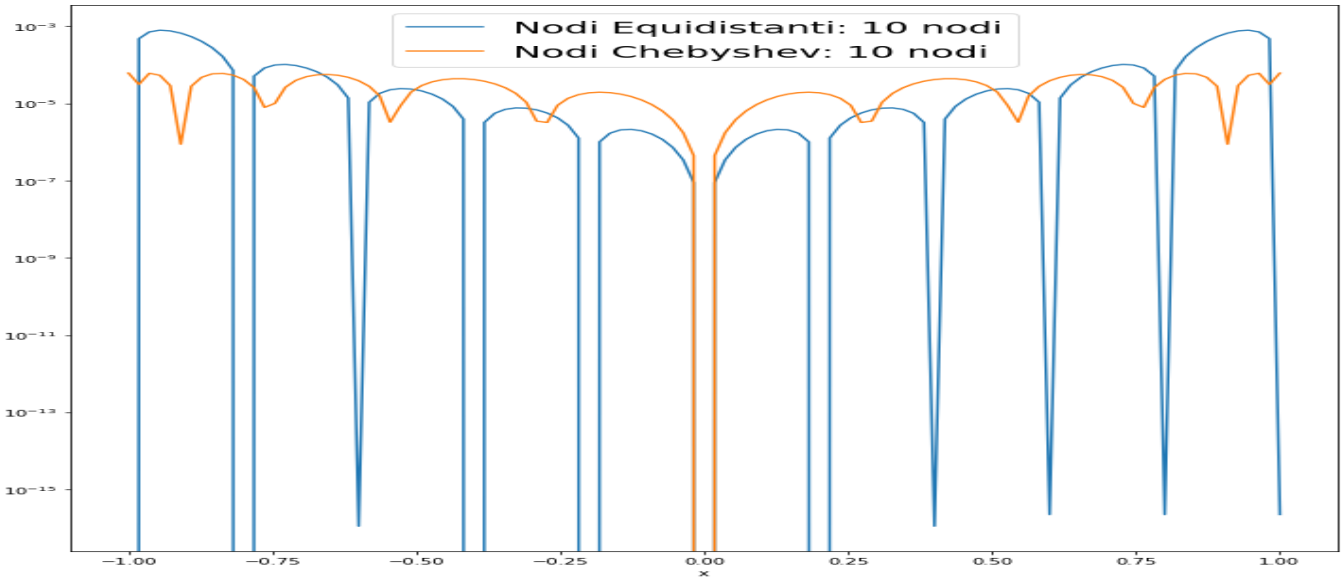

F(x) = 1/(1+x^2)
Newton: 10 nodi di Chebyshev

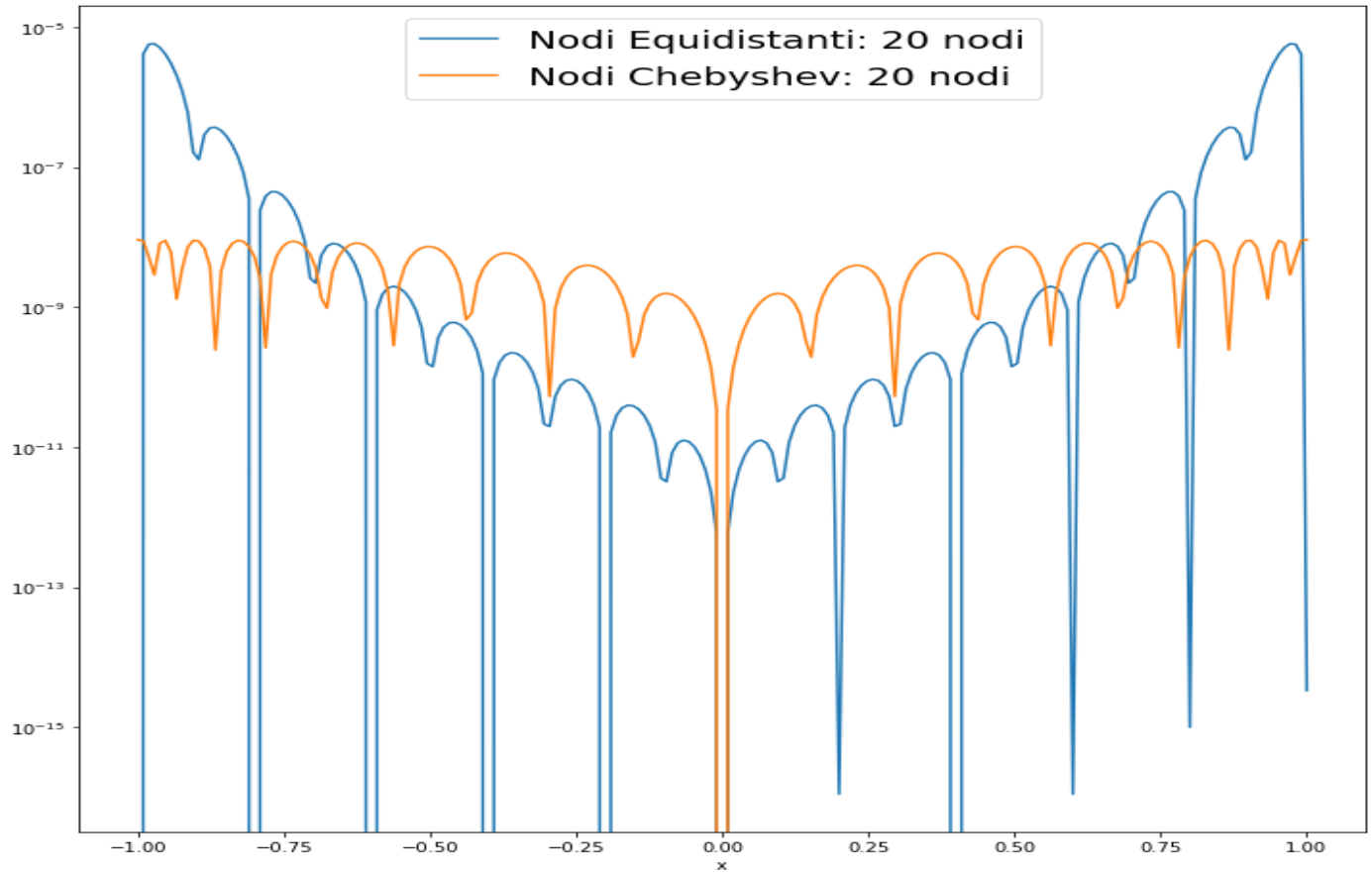

F(x) = 1/(1+x^2)
Newton: 20 nodi equidistanti

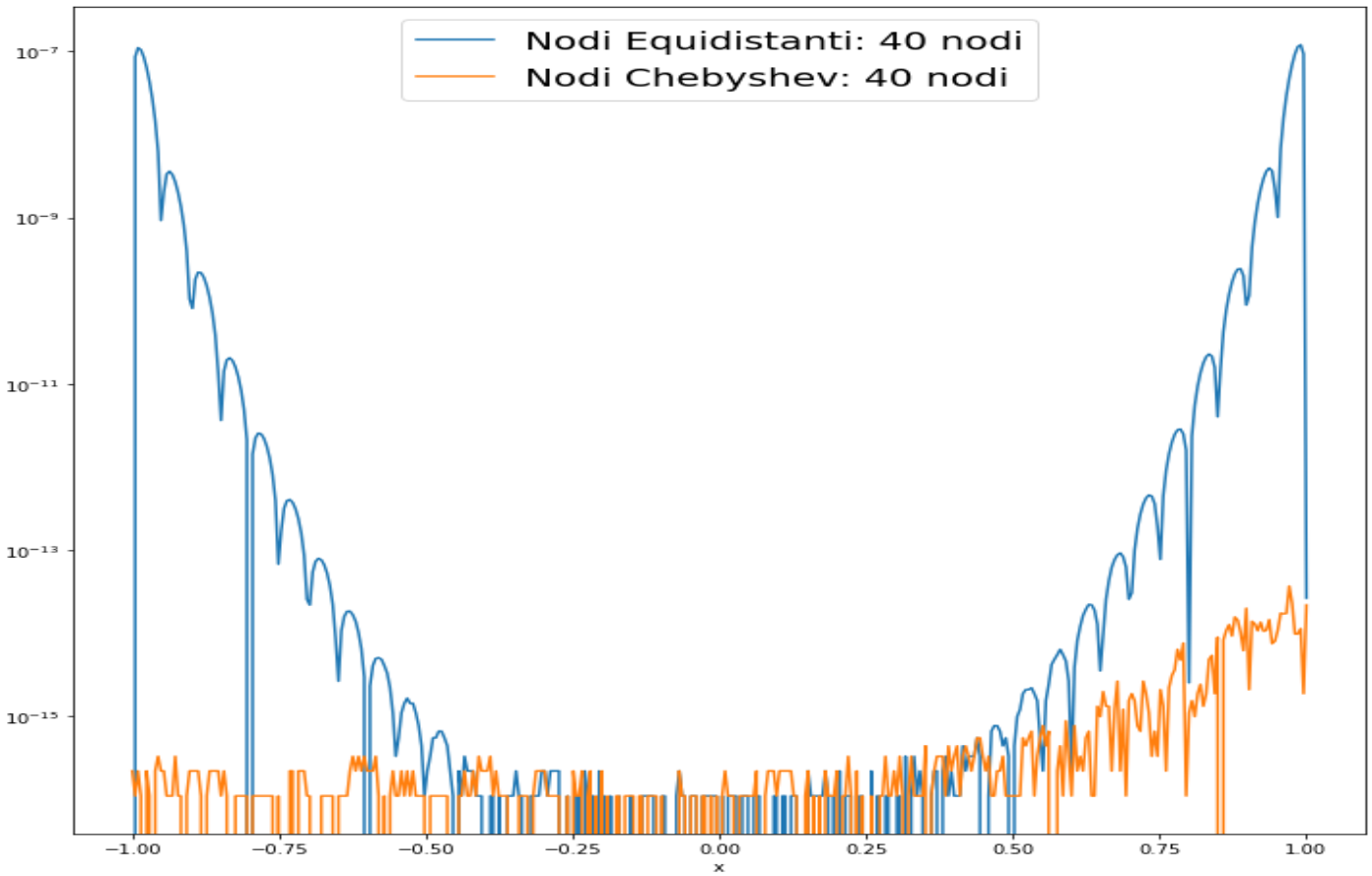

F(x) = 1/(1+x^2)
Newton: 20 nodi di Chebyshev


F(x) = 1/(1+x^2)
Newton: 40 nodi equidistanti


F(x) = 1/(1+x^2)
Newton: 40 nodi di Chebyshev


Nodi Equidistanti: 10 nodi
Nodi Chebyshev: 10 nodi


Nodi Equidistanti: 20 nodi
Nodi Chebyshev: 20 nodi
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x


Nodi Equidistanti: 40 nodi
Nodi Chebyshev: 40 nodi
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

APPLICAZIONE DEL METODO DI NEWTON PER IL CALCOLO DELLA RADICE N-ESIMA

Nel seguente esperimento vogliamo applicare il metodo di Newton per la ricerca della radice n-esima di un numero reale.

Sia $a \in \mathcal{R}$, vogliamo conoscere la $\sqrt[n]{a}$ considerando la funzione

$$f(x) = x^n - a$$

la cui radice è data proprio da $x^* = \sqrt[n]{x}$.

Essendo $f'(x) = n * x^{n-1}$, il metodo di Newton diventa

$$x_{k+1} = x_k - \frac{x_k^n - a}{n * x_k^{n-1}}$$

Implementazione

```
"""
Created on Sun Apr  5 13:15:29 2020

@author: Leonardo Saccotelli
"""

def NewtonMethod ( f_x , df_x ,n ,i , x0 , tolerance , maxIteration ) :

    #Calcolo f in xo
    f_x0 = f_x(x0, n, i)

    #Calcolo f' in xo
    df_x0 = df_x(x0, i)

    #Contatore di iterazioni
    iteration = 0
    #Flag sul termine del processo
    stop = False

    while not (stop) and iteration < maxIteration :
        #Calcolo la nuova sol x1 = x0 - f(x0)/f'(x0)
        x1 = x0 - f_x0 / df_x0

        #Calcolo f(x) in x1
        f_x1 = f_x(x1, n, i)

        #Verifica del criterio di arresto
        stop = abs(f_x1) +abs(x1-x0)/abs(x1) < tolerance/5
```

```
        #Incremento il contatore del numero di iterazioni
        iteration = iteration+1

        #Se non ho raggiunto lo zero, aggiorno i dati del problema
        #per la prossima iterazione
        if not(stop):
            x0 = x1
            f_x0 = f_x1
            df_x0 = df_x(x0, i)

    #Controllo sull'avvenuta convergenza
    if not(stop):
        print (' The method doesn\'t converge in %d iterations'
%iteration)

    return x1, iteration

def rootSquareProblem(x, number, i):
    y = x**i-number
    return y

def df(x, i):
    return i*(x**(i-1))

def df_2(x,i):
    return i*(i-1)*(x**(i-2))

def check_X0(x, i, a, b):
    if rootSquareProblem(a, x, i) * df_2(a, i) > 0:
        X0 = a
    elif rootSquareProblem(a, x, i) * df_2(a, i) < 0:
        X0 = b
    return X0

index = int(input('- Enter the index root: '))
number = float(input('- Enter a positive number: '))
print('-----------------------------------------------')


#Definisco l'intervallo in cui cercare la radice
a = 1
b = 6
#Definisco la tollerenza accettata nell'approssimare la soluzione esatta
del problema
tolerance = 1.0e-10

#Verifico xo per innescare il metodo
x0 = check_X0(number, index, a, b)

#Stampo i risultati
x, max_iter = NewtonMethod(rootSquareProblem, df, number, index, x0,
tolerance, 50)

print('The root is %f' %x)
```

```
- Enter the index root: >? 2
- Enter a positive number: >? 2
  ----------------------------------
  The root is 1.414214


· Enter the index root: >? 3
· Enter a positive number: >? 27
  -------------------------------
  The root is 3.000000


  Enter the index root: >? 5
  Enter a positive number: >? 25
  ---------------------------------
  The root is 1.903654
```

CONFRONTO TRA IL NUMERO DI ITERAZIONI NECESSARIE A RAGGIUNGERE UNA CERTA ACCURATEZZA DA PARTE DIVERSI METODI

Nel seguente esperimento vogliamo mettere a confronto le prestazioni dei metodi studiati per la ricerca degli zeri. In particolare, utilizzeremo i seguenti metodi:

- Bisezioni successive
- Newton (metodo delle tangenti)
- Secanti
- Corde

Effettueremo il test sulla funzione $f(x) = 3x^3 + 5x^2 - 6x - 5$, verificando la presenza di eventuali zeri nell'intervallo $[1, 5]$.

Confronteremo infine il numero di iterazioni necessarie ad ogni metodo per convergere alla soluzione del problema entro una certa tolleranza.

Implementazione

```
"""
Created on Fri Apr  3 16:07:20 2020

@author: Leonardo Saccotelli
"""
from math import *
from fractions import Fraction

"""
METODO DI BISEZIONE
 Al metodo viene passata:
  - la funzione di cui cercare uno zero
  - l'estremo inferiore dell'intervallo
  - l'estemo superiore dell'intervallo
  - la tollerenza dell'errore'
 Il metodo restituisce:
  - Lo zero di funzione
  - il numero di iterazioni eseguite
"""

def bisectionMethod(f_x, a, b, tolerance ):
```

```
    #Controllo la presenza di almeno uno zero nell'intervallo [a, b]
    f_a = f_x(a)
    f_b = f_x(b)

    if (f_a * f_b)>0:
        print('Error: root in [%f , %f] not guaranteed' % (a,b))
    #Assicurata la presenza di almeno uno zero in [a,b]
    else:
    #Determino il numero di iterazione necessario per convergere
    #allo zero di funzione entro l'errore stabilito
        n = int(ceil((log(b-a)-log(tolerance))/log(2)))

        for k in range(n+1):
            #Calcolo il punto medio dell'intervallo [a,b]
            c = a + (b-a)/2

            #Calcolo il valore di in c
            f_c = f_x(c)


          #Verifico che lo zero trovato coincida con lo zero del
          problema
            if abs(f_c) < 1.0e-12:
                #c è lo zero di funzione
                break

            #Controllo se lo zero si trova nell'intervallo [a, c]
            if (f_a * f_c) < 0:
                b = c
                f_b = f_c
            else:
            #Lo zero si trova nell'intervallo [c, b]
                a = c
                f_a = f_c

    #Restituisco i risultati del problema
    return c, k+1


"""
METODO DI NEWTON(DELLE TANGENTI)
 Al metodo viene passata:
  - la funzione di cui cercare uno zero
  - la derivata prima della funzione
  - la prima approssimazione da cui avviare il metodo
  - la tollerenza dell'errore'
  - il numero massimo di iterazioni
 Il metodo restituisce:
  - Lo zero di funzione
  - il numero di iterazioni eseguite
"""
def NewtonMethod ( f_x , df_x , x0 , tolerance , maxIteration ) :

    #Calcolo f in xo
    f_x0 = f_x(x0)
```

```
    #Calcolo f' in xo
    df_x0 = df_x(x0)

    #Contatore di iterazioni
    iteration = 0
    #Flag sul termine del processo
    stop = False

    while not (stop) and iteration < maxIteration :
        #Calcolo la nuova sol x1 = x0 - f(x0)/f'(x0)
        x1 = x0 - f_x0 / df_x0

        #Calcolo f(x) in x1
        f_x1 = f_x(x1)

        #Verifica del criterio di arresto
        stop = abs(f_x1) +abs(x1-x0)/abs(x1) < tolerance/5

        #Incremento il contatore del numero di iterazioni
        iteration = iteration+1

        #Se non ho raggiunto lo zero, aggiorno i dati del problema
        #per la prossima iterazione
        if not(stop):
            x0 = x1
            f_x0 = f_x1
            df_x0 = df_x(x0)

    #Controllo sull'avvenuta convergenza
    if not(stop):
        print (' The method doesn\'t converge in %d iterations'
%iteration)

    return x1, iteration


"""
METODO DELLE SECANTI
 Al metodo viene passata:
  - la funzione di cui cercare uno zero
  - la prima approssimazione da cui avviare il metodo
  - la seconda approssimazione da cui avviare il metodo
  - la tollerenza dell'errore'
  - il numero massimo di iterazioni
 Il metodo restituisce:
  - Lo zero di funzione
  - il numero di iterazioni eseguite
"""
def SecantMethod ( f_x, x0, x1, tolerance , maxIteration ) :

    #Calcolo f in xo
    f_x0 = f_x(x0)

    #Calcolo f in x1
    f_x1 = f_x(x1)
```

```
    #Contatore di iterazioni
    iteration = 0
    #Flag sul termine del processo
    stop = False

    while not (stop) and iteration < maxIteration :
        #Calcolo la nuova sol x2
        x2 = x1 - ( f_x1 / ((f_x0 - f_x1)/(x0 - x1)))

        #Calcolo f(x) in x2
        f_x2 = f_x(x2)

        #Verifica del criterio di arresto
        stop = abs(f_x2) +abs(x2-x1)/abs(x2) < tolerance/5

        #Incremento il contatore del numero di iterazioni
        iteration = iteration+1

        #Se non ho raggiunto lo zero, aggiorno i dati del problema
        #per la prossima iterazione
        if not(stop):
            x0 = x1
            f_x0 = f_x1

            x1 = x2
            f_x1 = f_x2

    #Controllo sull'avvenuta convergenza
    if not(stop):
        print (' The method doesn\'t converge in %d iterations'
%iteration)

    return x2, iteration


"""
METODO DELLE CORDE
 Al metodo viene passata:
  - la funzione di cui cercare uno zero
  - la derivata prima della funzione
  - la prima approssimazione da cui avviare il metodo
  - la tollerenza dell'errore'
  - il numero massimo di iterazioni
 Il metodo restituisce:
  - Lo zero di funzione
  - il numero di iterazioni eseguite
"""
def stringRootMethod ( f_x , a, x0 , tolerance , maxIteration ) :

    #Calcolo f in xo
    f_x0 = f_x(x0)

    #Contatore di iterazioni
    iteration = 0
    #Flag sul termine del processo
    stop = False
```

```
    while not (stop) and iteration < maxIteration :

        m = (f_x(a) - f_x0)/(a - x0)

        if (abs(m) < 1.0e-14):
            print('\'m\'= 0.')
            break

        #Calcolo la nuova sol x1 = x0 - f(x0)/m
        x1 = x0 - f_x0 / m

        #Calcolo f(x) in x1
        f_x1 = f_x(x1)

        #Verifica del criterio di arresto
        stop = abs(f_x1) +abs(x1-x0)/abs(x1) < tolerance/5

        #Incremento il contatore del numero di iterazioni
        iteration = iteration+1

        #Se non ho raggiunto lo zero, aggiorno i dati del problema
        #per la prossima iterazione
        if not(stop):
            x0 = x1
            f_x0 = f_x1

    #Controllo sull'avvenuta convergenza
    if not(stop):
        print ('The method doesn\'t converge in %d iterations'
%iteration)

    return x1, iteration


"""
Created on Fri Apr  3 22:09:08 2020

@author: Leonardo Saccotelli
"""
import numpy as np
from Zeri_di_funzione_Algoritmi import *

#Definisco la funzione di cui voglio conoscere lo zero
def f_x(x):
    y = 3*x**3 + 5*x**2 - 6*x - 5
    return y

#Definisco la derivata prima della funzione di cui
#voglio conoscere lo zero
def df_x(x):
    y = 9*x**2 + 10*x - 6
    return y

#Definisco l'intervallo in cui cercare lo zero
a = 1
```

```
b = 5

#Definisco la tollerenza accettata nell'approssimare la soluzione esatta
del problema
tolerance = 1.0e-5

#Fisso il numero massimo di iterazioni
maxIteration = 100

#Calcolo dello zero attraverso il metodo di bisezione
xBis, iterBis = bisectionMethod(f_x, a, b, tolerance)

#Calcolo dello zero attraverso il metodo di Newton
xNewton, iterNewton = NewtonMethod(f_x, df_x, b, tolerance,
maxIteration)

#Calcolo dello zero attraverso il metodo delle secanti
xSec, iterSec = SecantMethod(f_x, a, b, tolerance, maxIteration)

#Calcolo dello zero attraverso il metodo delle corde
xCor, iterCor = stringRootMethod(f_x, a, b, tolerance, maxIteration)

strBis = 'Bisezioni successive'
strNew = 'Newton (tangenti)'
strSec = 'Secanti'
strCor = 'Corde'

print('|----------------------+----------------------+----------------
--|')
print('|   Metodo utilizzato   |  Iterazioni eseguite  |     Soluzione
|')
print('|----------------------+----------------------+----------------
--|')
print('| %20s  |  %10d           |     %f     |' %(strBis, iterBis,
xBis))
print('|----------------------+----------------------+----------------
--|')
print('| %19s   |  %10d           |     %f     |' %(strNew, iterNewton,
xNewton))
print('|----------------------+----------------------+----------------
--|')
print('| %14s        |  %10d           |     %f     |' %(strSec,
iterSec, xSec))
print('|----------------------+----------------------+----------------
--|')
print('| %13s         |  %10d           |     %f     |' %(strCor,
iterCor, xCor))
print('|----------------------+----------------------+----------------
--|')
```

Metodo utilizzato	Iterazioni eseguite	Soluzione
Bisezioni successive	20	1.190212
Newton (tangenti)	8	1.190216
Secanti	7	1.190216
Corde	10	1.190216

IMPLEMENTAZIONE METODI DI QUADRATURA

Implementazione metodi di Simpson e Trapezi composti

```
import numpy as np
"""
FORMULA DEI TRAPEZI COMPOSTI
 Al metodo vengono passati:
     - la funzione integranda
     - l'estremo inferiore di integrazione
     - l'estremo superiore di integrazione
     - il numero di intervallini
"""
def CompositeTrapezoid(f_x, a, b, N):
    #Estrpolo N+1 intervalli equidistanti da [a,b]
    z = np.linspace(a,b,N+1)

    #Calcolo f_x() in ogni punto di z
    fz = f_x(z)

    S = 0
    #Calcolo del trapezio composto
    for i in range(1,N):
        S = S + fz[i]
    TC = (fz[0] + 2*S + fz[N])*(b-a)/2/N
    return TC

"""
FORMULA DI SIMPSON COMPOSTA
 Al metodo vengono passati:
     - la funzione integranda
     - l'estremo inferiore di integrazione
     - l'estremo superiore di integrazione
     - il numero di intervalli
"""
def CompositeSimpson(f, a, b, N):
    #Genero n+1 intervallini in [a,b]
    z = np.linspace(a,b,N+1)
    #Calcolo f negli intervalli z
    fz = f(z)

    #Definisco le somme dispari e le somme pari
    S_d = 0; S_p = 0
    #Definisco l'ampiezza dei singoli intervalli
    h = (b-a)/N

    #Calcolo le somme dispari
    for i in range(1,N,2):
        S_d = S_d + fz[i]
    #Calcolo le somme pari
    for i in range(2,N-1,2):
        S_p = S_p + fz[i]
    Tsc = (fz[0] + 4*S_d + 2*S_p + fz[N])*h/3

    return Tsc
```

Implementazione metodi per la realizzazione dei grafici di Simpson e Trapezi; per la creazione del grafico relativo al metodo di Simpson è richiesto il metodo per la creazione del polinomio di interpolazione di Lagrange nella prima formulazione Baricentrica.

```
import numpy as np
import matplotlib.pyplot as plt
from InterpolazionePolinomiale_Algoritmi import
lagrangeFirstBarycentricFormula

"""
METODO PER LA RAPPRESENTAZIONE DEL METODO DEL TRAPEZIO COMPOSTO
 Il metodo riceve:
     - la funzione integranda
     - l'estremo inferiore di integrazione
     - l'estremo superiore di integrazione
"""
def DrawCompositeTrapezoid(f_x, a, b, N):
    #Estrapolo 200 punti equidistanti sull'intervallo [a-0.2, b+0.2]
    x = np.linspace(a-0.2, b+0.2, 200)

    #Calcolo la funzione f nei 200 punti equidistanti
    fx = f_x(x)

    #Estrpolo N+1 intervalli equidistanti da [a,b]
    z = np.linspace(a,b,N+1)

    #Calcolo f_x() in ogni punto di z
    fz = f_x(z)

    plt.figure(1,figsize=(10, 10))

    #Creo il grafico di f
    plt.plot(x,fx,'k-',label='f(x)')

    #Creo i trapezi
    for i in range(0,N):
        xx = np.array([z[i], z[i], z[i+1], z[i+1], z[i]])
        yy = np.array([0, fz[i], fz[i+1], 0, 0])
        plt.plot(xx,yy,'b-')
        plt.fill(xx, yy, color ='lavender')


    plt.xlabel('x')
    plt.legend(prop={'size':26})
    plt.show()


"""
METODO PER LA RAPPRESENTAZIONE DEL METODO DEL TRAPEZIO
 Il metodo riceve:
     - la funzione integranda
     - l'estremo inferiore di integrazione
     - l'estremo superiore di integrazione
```

```
      - il numero di intervalli
"""
def DrawCompositeSimpson(f, a, b, N):

    x = np.linspace(a-0.2,b+0.2,200)
    fx = f(x)

    #Estrpolo N+1 intervalli equidistanti da [a,b]
    z = np.linspace(a,b,N+1)

    #Calcolo f() in ogni punto di z
    fz = f(z)

    plt.figure(1,figsize=(10,8))
    plt.plot(x,fx,'k-',label='f(x)')

    for i in range(0,N):

        #------- Costruzione del polinomio di interpolazione
        x_Pol = np.array([z[i], (z[i+1] + z[i])/2, z[i+1]])
        y_Pol = np.array([fz[i], f((z[i]+z[i+1])/2), fz[i+1]])
        xPoints = np.linspace(z[i], z[i+1], 10)
        P_x = lagrangeFirstBarycentricFormula(x_Pol, y_Pol, xPoints)
        #plt.plot(x_Pol, y_Pol, 'ro')

        #------- Costruzione del  grafico del polinomio di
interpolazione
        xx = np.array([z[i],z[i]])
        xx = np.append(np.append(xx, xPoints), np.array([z[i+1], z[i+1],
z[i]]))

        yy = np.array([0,fz[i]])
        yy = np.append(np.append(yy, P_x), np.array([fz[i+1], 0, 0]))

        plt.plot(xx,yy,'b')
        plt.fill(xx, yy, color ='lavender')

    plt.xlabel('x')
    plt.legend(prop={'size':26})
    plt.show()
```

METODO DI SIMPSON E METODO DEI TRAPEZI COMPOSTI

Nel seguente esperimento vogliamo confrontare i metodi di quadratura composta mostrando quale delle due fornisca una migliore approssimazione del valore dell'integrale

$$\int_1^6 \sqrt[3]{x}\, dx$$

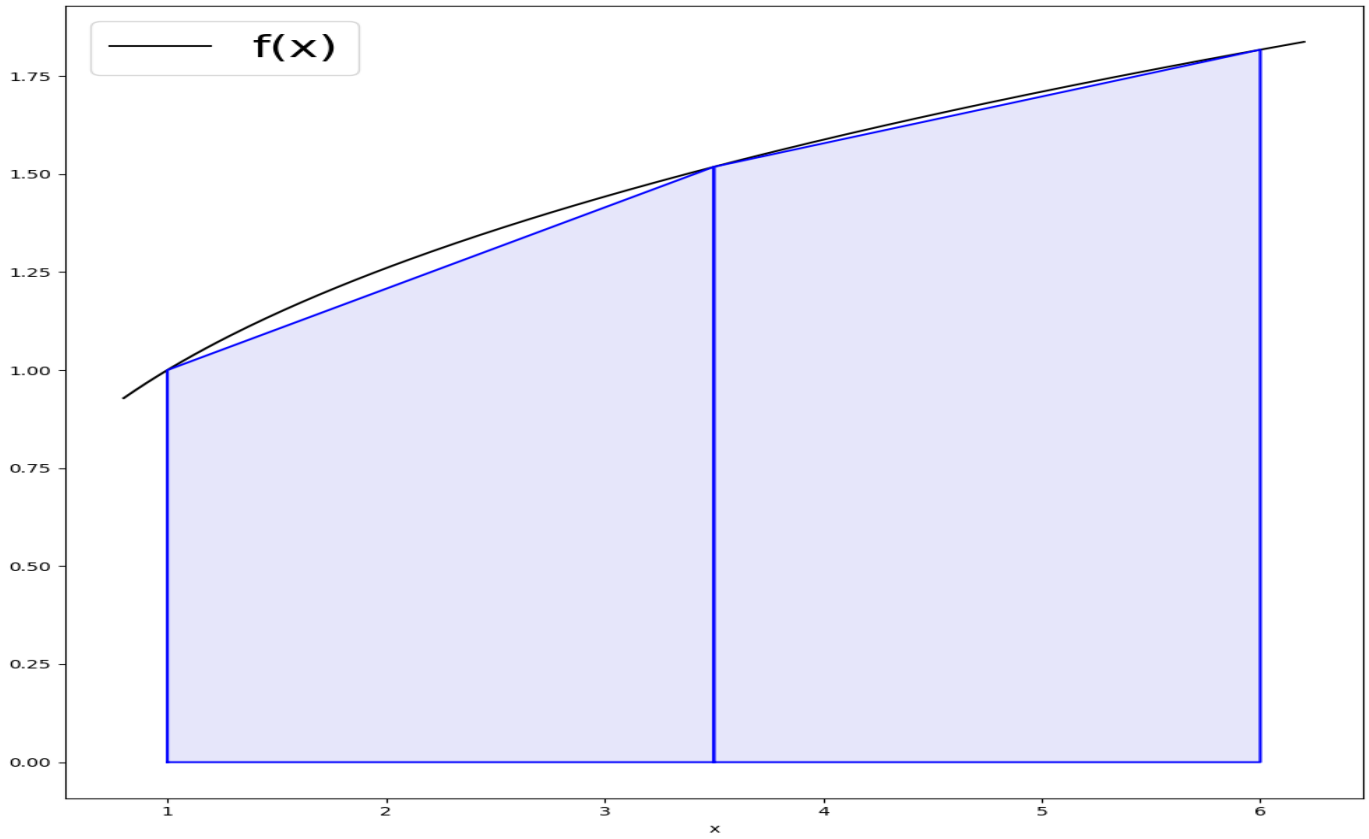
Varieremo il numero di sotto intervalli in cui suddividiamo $[1, 6]$ misurando di volta in volta l'errore commesso.

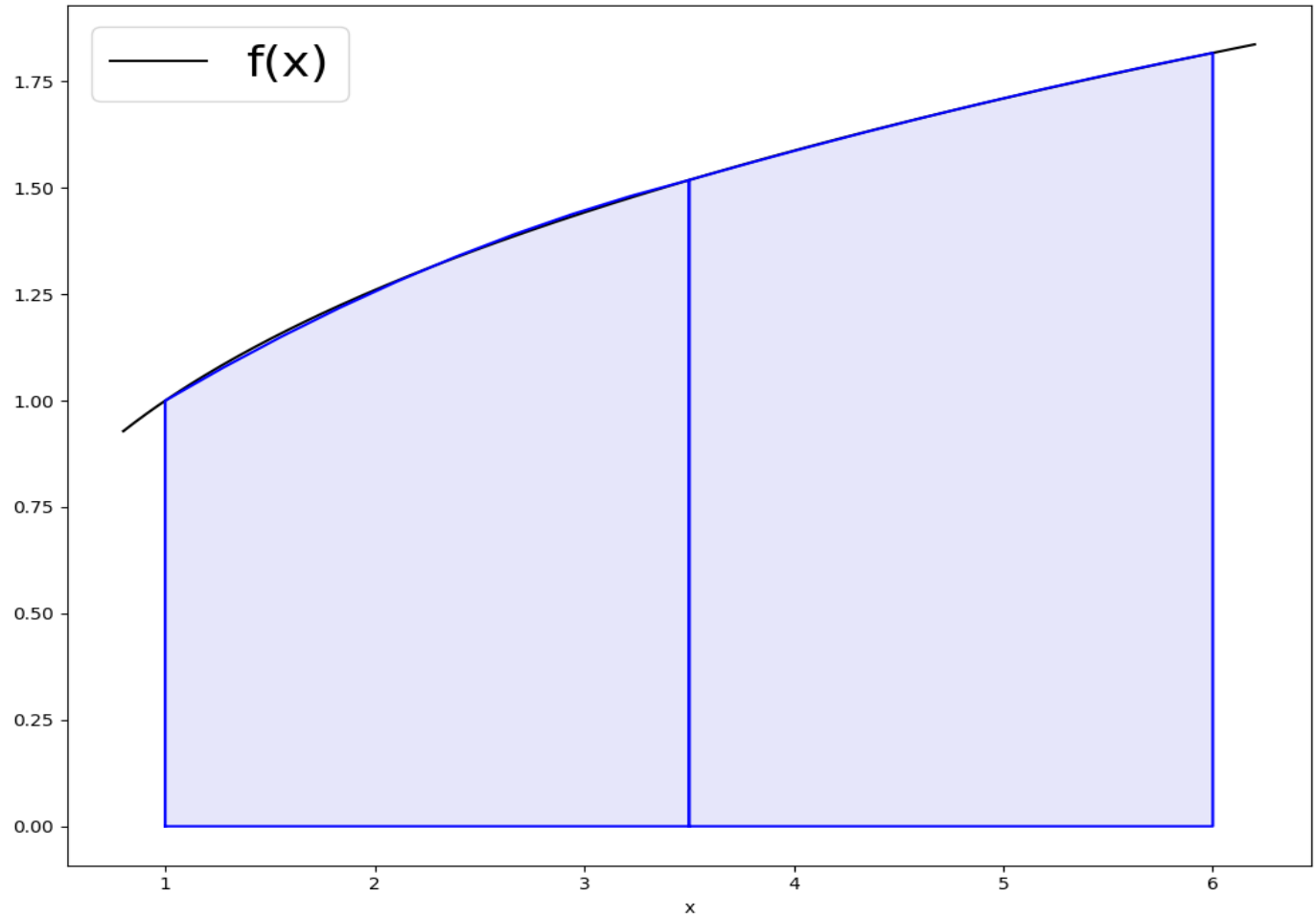
Implementazione test

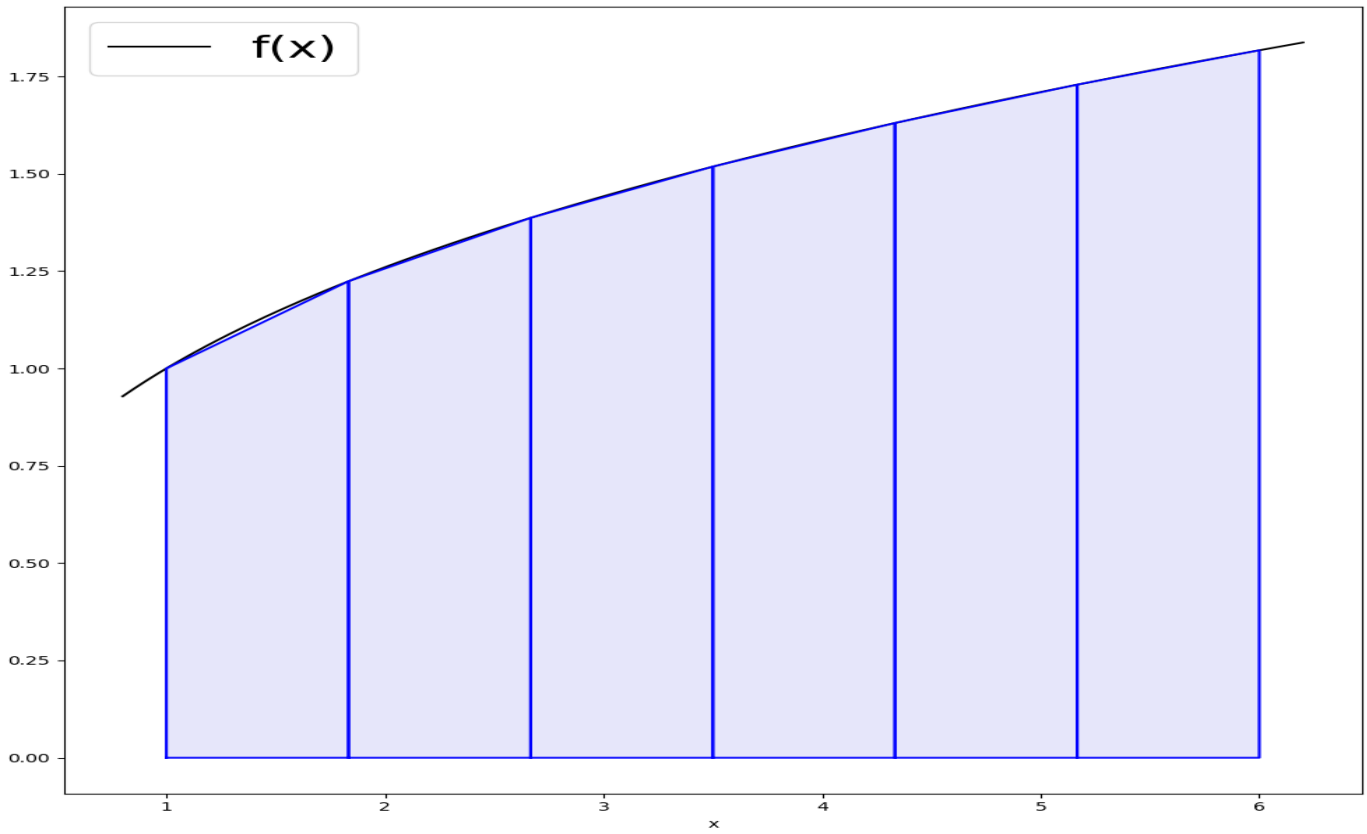
```
import numpy as np
from Algoritmi_Grafico_Integrali import *
from Algoritmi_Quadratura import *

#Definizione della funzione da integrare
def f(x):
    y = x**(1/3)
    return y

#Definizione della primitiva di f
def F(x):
    y = 3/4 * x * x**(1/3)
    return y

#Definizione dell'intervallo di integrazione
a = 1.0 ; b = 6.0

#Definizone del numero di intervalli in [a,b]
N = 2
#Calcolo dell'integrale di f(x) tramite trapezi composti
TC = CompositeTrapezoid(f, a, b, N)

#Calcolo dell'integrale di f(x) tramite Simpson composti
SC = CompositeSimpson(f, a, b, N)

#Calcolo dell'integrale di f(x) tramite la primitiva
I = F(b) - F(a)

#Calcolo dell'errore commesso tramite trapezi composti
E_tc = abs(I - TC)

#Calcolo dell'errore commesso tramite simpson composti
E_sc = abs(I - SC)

# Visualizzazione risultati relativi al metodo di trapezi
print('\n  Integrale esatto:              %f ' % I)
```

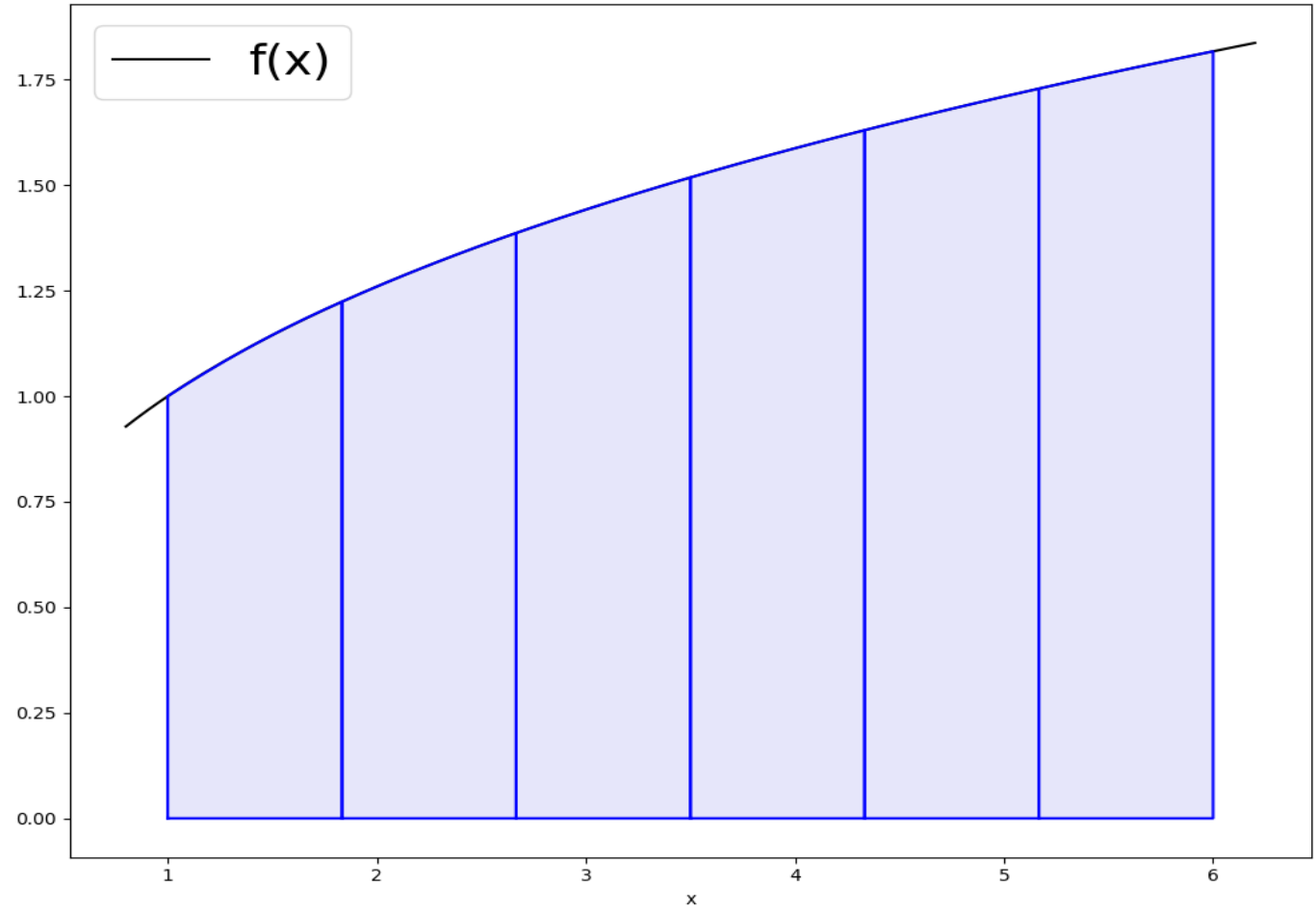
```
print('  Formula del trapezio composta: %f ' % TC)
print('  Errore commesso:               %e ' % E_tc)
print('  ----------------------------------------')
# Visualizzazione risultati relativi al metodo di Simpson
print('  Integrale esatto:              %f ' % I)
print('  Formula di Simpson composta:   %f ' % SC)
print('  Errore commesso:               %e ' % E_sc)

#Visualizzazione del grafico
DrawCompositeTrapezoid(f, a, b, N)
DrawCompositeSimpson(f, a, b, N)
```

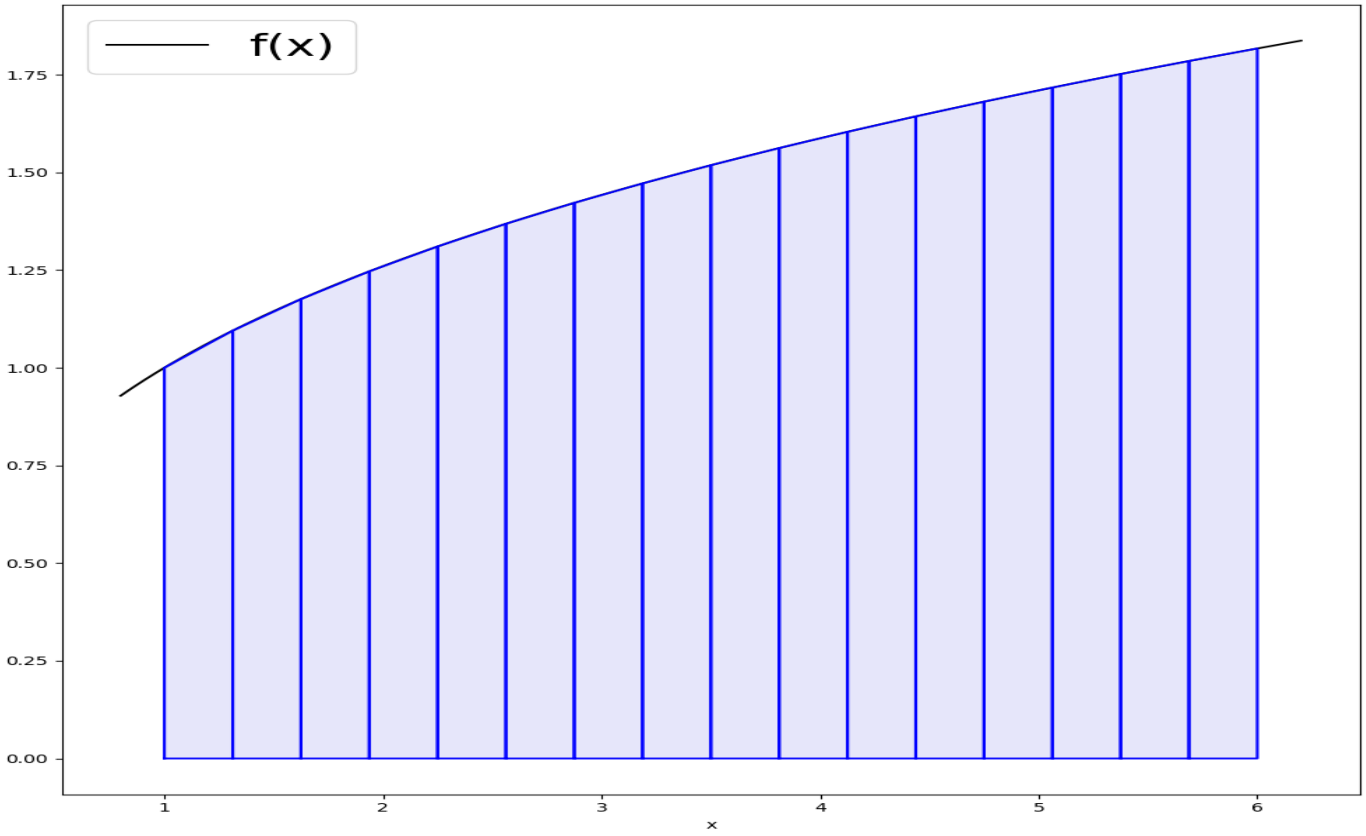
```
Numero di nodi di quadratura:    2
-----------------------------------------------
Integrale esatto:                7.427043
Formula del trapezio composta:   7.317137
Errore commesso:                 1.099057e-01
-----------------------------------------------
Integrale esatto:                7.427043
Formula di Simpson composta:     7.408582
Errore commesso:                 1.846055e-02


Numero di nodi di quadratura:    6
-----------------------------------------------
Integrale esatto:                7.427043
Formula del trapezio composta:   7.413811
Errore commesso:                 1.323137e-02

Numero di nodi di quadratura:    16
-----------------------------------------------
Integrale esatto:                7.427043
Formula del trapezio composta:   7.425156
Errore commesso:                 1.886372e-03
-----------------------------------------------
Integrale esatto:                7.427043
Formula di Simpson composta:     7.427025
Errore commesso:                 1.765049e-05
```

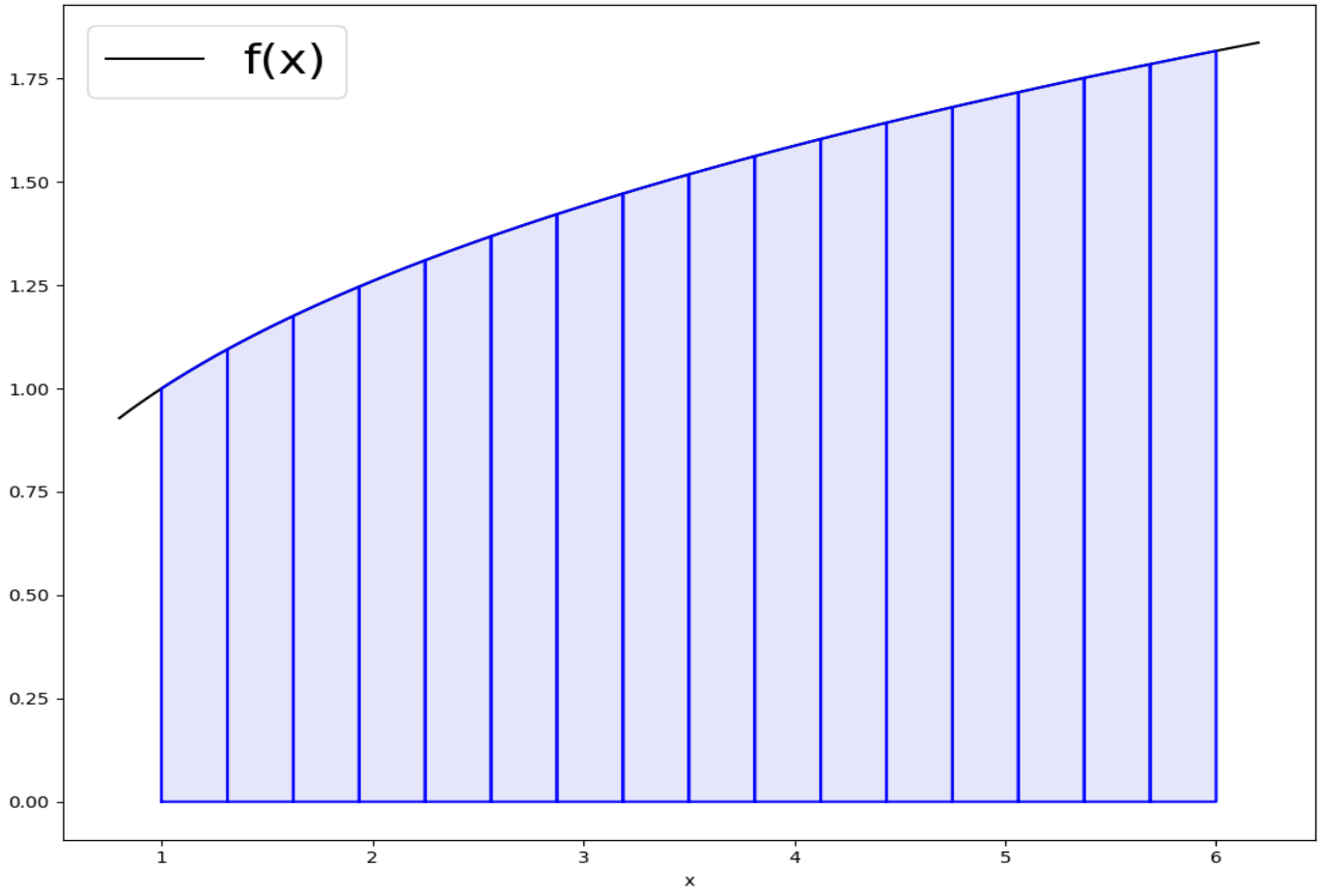

f(x)
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1
2
3
4
5
6
x


f(x)
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1
2
3
4
5
6
x


f(x)
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1
2
3
4
5
6
x


f(x)
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1
2
3
4
5
6
x


f(x)
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1
2
3
4
5
6
x


f(x)
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1
2
3
4
5
6
x

CONVERGENZA DEI METODI DI QUADRATURA NUMERICA

Nel seguente esperimento vogliamo misurare l'errore commesso nell'approssimare il valore dell'integrale definito

$$\int_1^{10} \frac{1}{x} + \frac{1}{x^2}\, dx$$

attraverso le formule di quadratura.

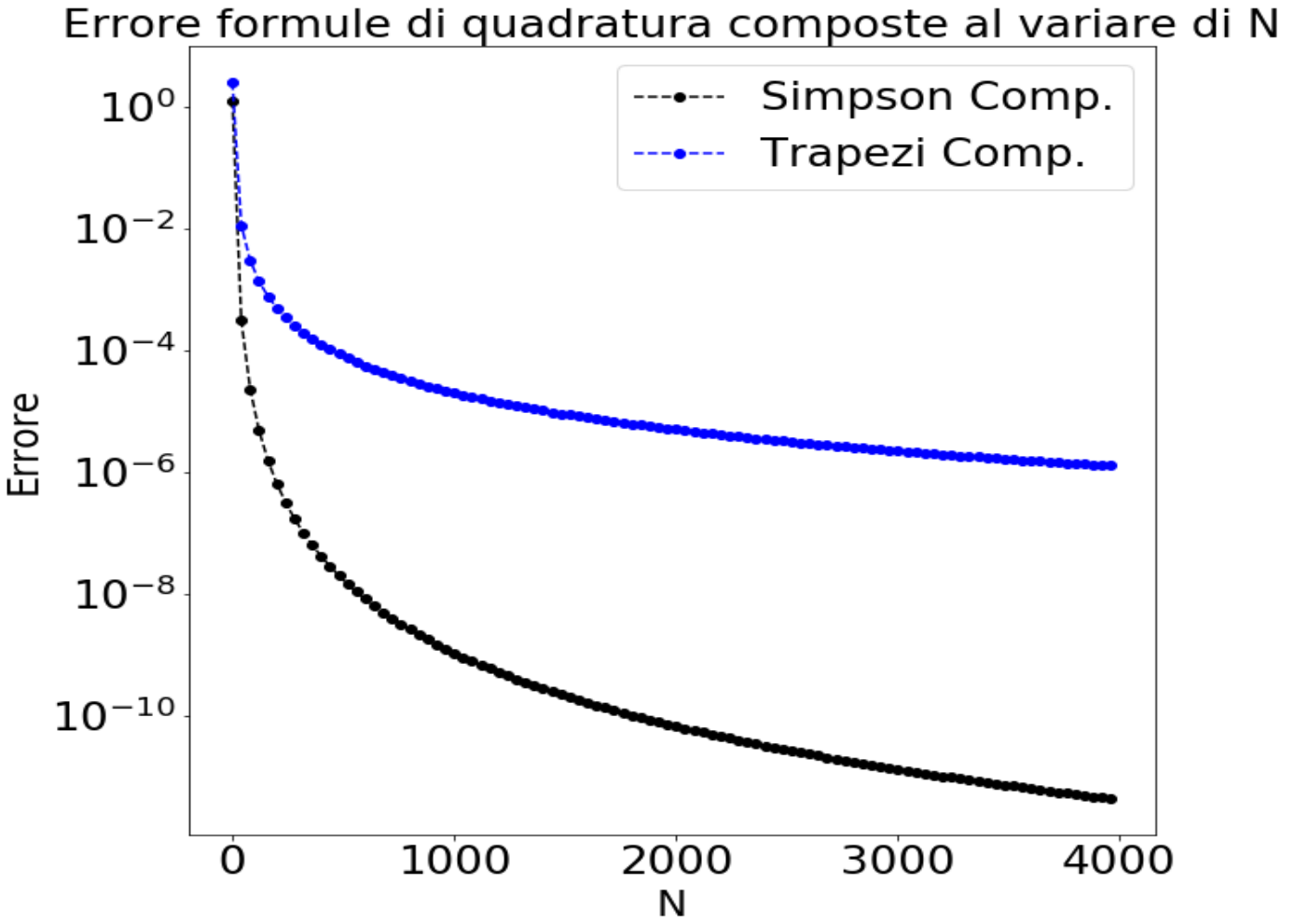
Misureremo l'errore commesso al variare del numero di sotto intervalli in cui suddividiamo l'intervallo di integrazione.

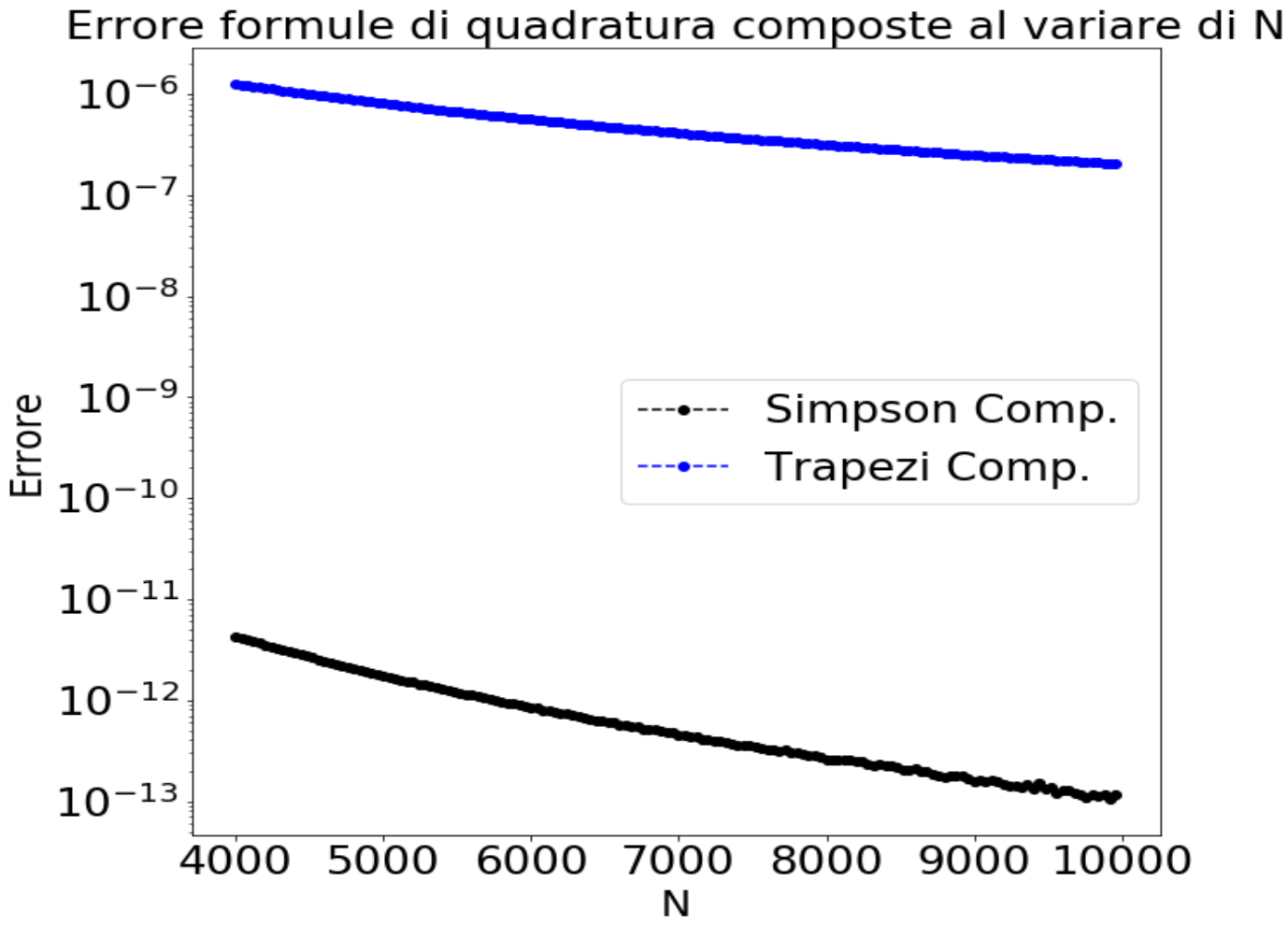
Confronteremo pertanto i risultati ottenuti attraverso il metodo di Simpson con i risultati forniti dal metodo dei trapezi.

Implementazione test.

```
import numpy as np
import matplotlib.pyplot as plt
from Algoritmi_Quadratura import *
from matplotlib import rc

#Definizione della funzione da integrare
def f(x):
    y = 1/x + 1/(x**2)
    return y

#Definizione della primitiva di f
def F(x):
    y = -1/x + np.log(abs(x))
    return y

#Definizione dell'intervallo di integrazione
a = 1.0 ; b = 10.0

#Calcolo l'integrale di f attraverso la sua primitiva
I = F(b) - F(a)

# Formula Simpson e trapezi composti al variare di N
#Numero massimo di intervalli su cui integrare
N_max = 4000
N_range = range(2,N_max,40)
```

```
#Creo i vettori che conterranno l'errore di integrazione al variare di N
Err_Simpson = np.zeros(len(N_range))
Err_Trapezi = np.zeros(len(N_range))

#Indice per i vettori degli errori
k = 0

for N in N_range:
    #Calcolo l'integrale tramite Simpson
    CS = CompositeSimpson(f,a,b,N)
    #Calcolo l'integrale tramite Trapezi
    TC = CompositeTrapezoid(f, a, b, N)
    #Caloclo l'errore con Simpson
    Err_Simpson[k] = abs(I-CS)
    #Calcolo l'errore con Trapezi
    Err_Trapezi[k] = abs(I-TC)

    k = k + 1

# Grafico errori al variare di N
plt.figure(1, figsize=(10,10))
plt.semilogy(N_range,Err_Simpson,'k--o',label='Simpson Comp.')
plt.semilogy(N_range,Err_Trapezi,'b--o',label='Trapezi Comp.')
plt.xlabel('N',fontsize=24)
plt.ylabel('Errore',fontsize=24)
plt.legend(prop={'size':26})
plt.title('Errore formule di quadratura composte al variare di
N',fontsize=26)
rc('xtick',labelsize=26)
rc('ytick',labelsize=26)
```


Errore formule di quadratura composte al variare di N
Simpson Comp.
Trapezi Comp.
Errore
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
0
1000
2000
3000
4000
N


Errore formule di quadratura composte al variare di N
Simpson Comp.
Trapezi Comp.
Errore
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
10^{-11}
10^{-12}
10^{-13}
4000
5000
6000
7000
8000
9000
10000
N