Università degli Studi di Bari “Aldo Moro”

Corso di Laurea in Informatica e Tecnologie per la Produzione del Software

A.A. 2021/2022

Elaborato delle Esercitazioni del Corso di Calcolo Numerico

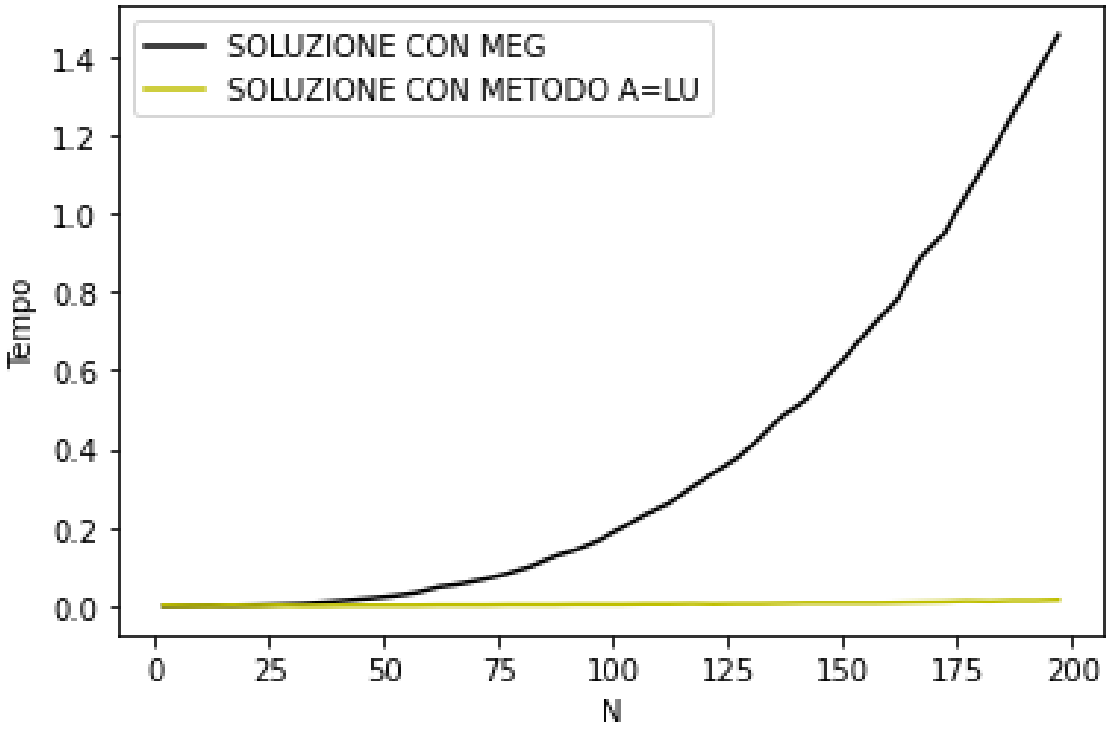
A cura di Giuseppe Tauro matricola 717164

Argomenti trattati:

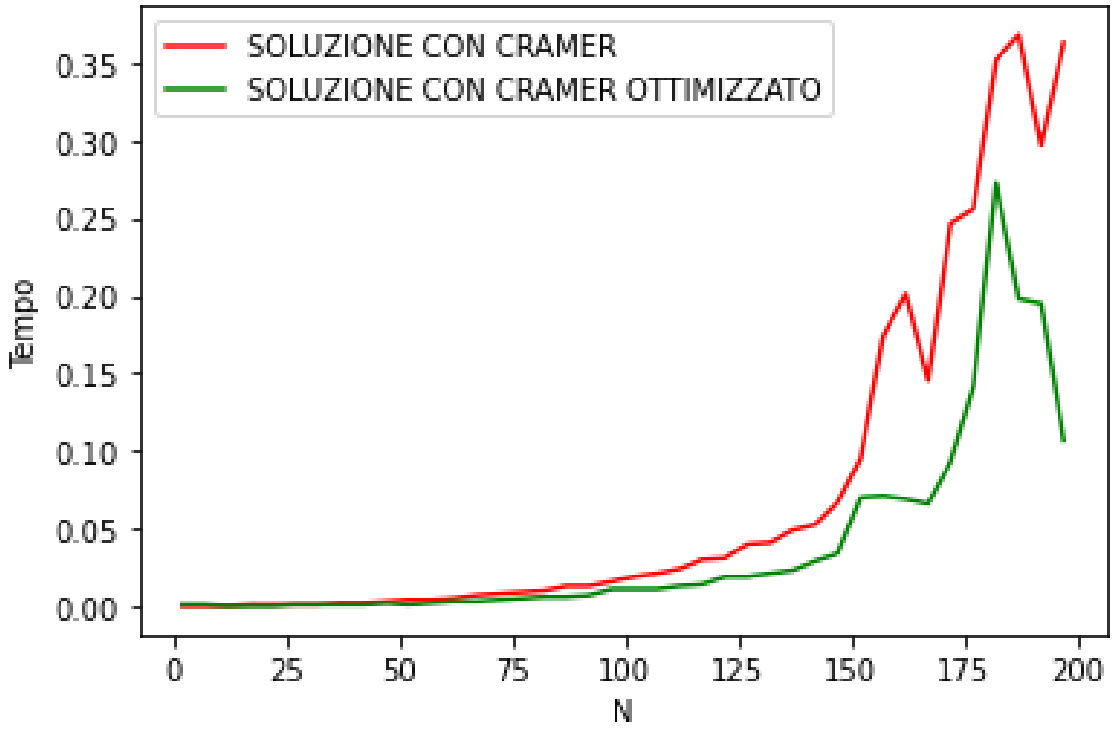
Confronto Algoritmi per la risoluzione di sistemi di equazione linerari Ax = b

Nel seguente programma sono stati implementati i seguenti algoritmi:

- Metodo Cramer e Cramer ottimizzato
- Metodo MEG + algoritmo di sostituzione all'indietro
- Metodo Decomposizione A = LU
- Metodo Built-in di numpy
- Metodo Matrice inversa



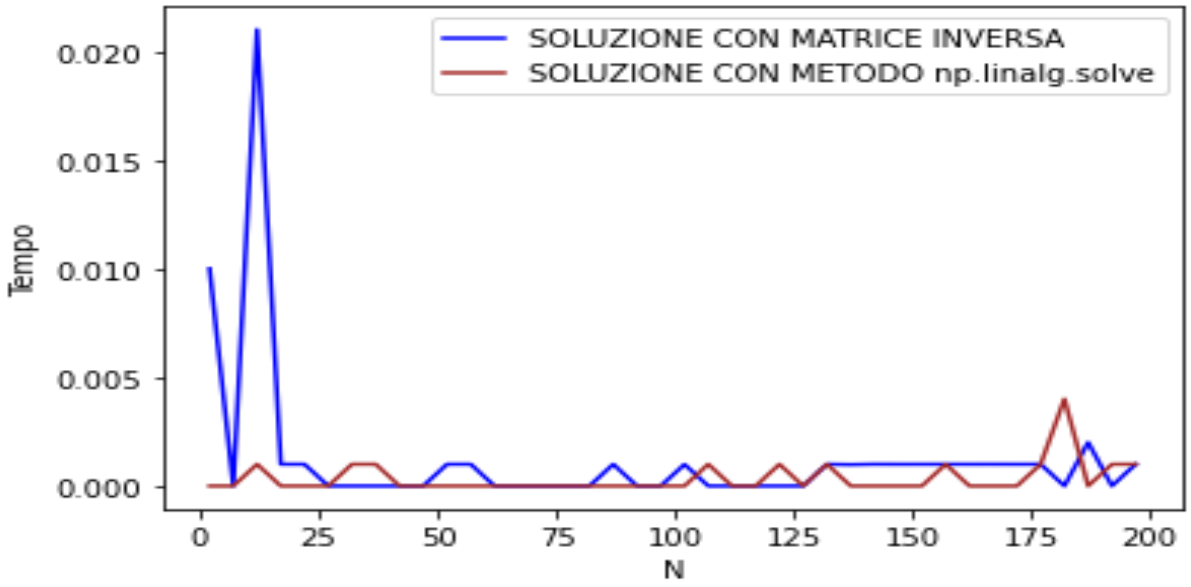
Osservando il grafico dove andiamo a paragonare SOLUZIONE CON MEG e SOLUZIONE CON METODO A=LU è possibile notare che l'adozione del MEG per rendere la matrice piena una triangolare U per poi risolvere il sistema di equazioni lineari con l'algoritmo di sostituzione all'indietro è un metodo più lento rispetto alla decomposizione A = LU



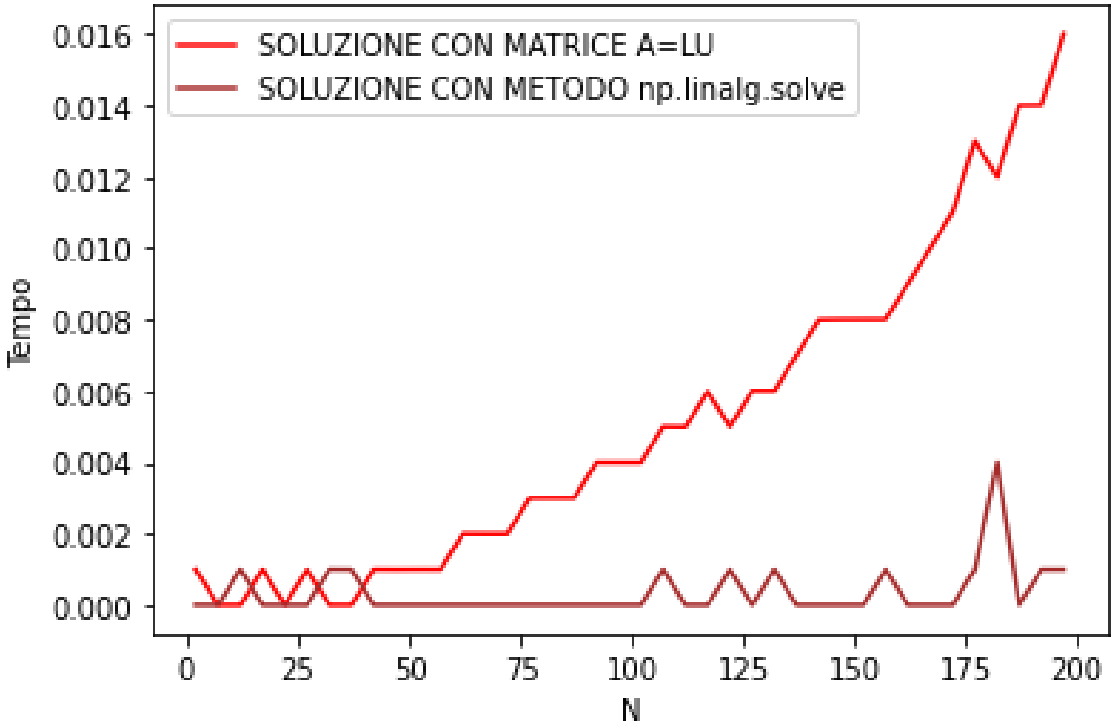
Possiamo notare che il metodo di Cramer che dovrebbe essere più lento rispetto all'eliminazione di Gauss, se implementato con il calcolo del determinante tramite funzioni built-in porta ad una velocità d'esecuzione maggiore.

Nel caso della versione ottimizzata notiamo che l'algoritmo è davvero molto veloce perchè calcoliamo il determinante di A una sola volta e non per ogni Xi.

Importante dire che il metodo di Cramer sia meno stabile del MEG con sostituzione all'indietro oppure rispetto alla decomposizione A = LU, infatti le irregolarità aumentano all'aumentare di N



La massima velocità di esecuzione nei test è data dai metodi nativi della libreria NUMPY, è osservabile che la risoluzione del sistema tramite la matrice inversa produca dei picchi di instabilità maggiori rispetto all'utilizzo del metodo nativo np.linalg.solve(A, b)



Per quanto riguarda il confronto tra np.linalg.solve(A, b) e la decomposizione A = LU possiamo osservare come l'algoritmo di built-in di NUMPY sia tendenzialmente più veloce però mostra picchi irregolari maggiori

Algoritmi implementati

```
def eliminazione_gauss(A, b):
    A = np.copy(A)
    b = np.copy(b)

    n = len(A)

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

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

    return A, b
```

```
def sostituzione_indietro(U, b):
    n = len(U)
    x = np.zeros(n)

    if abs(np.prod(np.diag(U))) < epsilon_machine():
        print("Attenzione! Questa matrice potrebbe non avere soluzioni")
    else:
        for i in range(n - 1, -1, -1):
            S = 0

            for j in range(i + 1, n):
                S = S + U[i, j] * x[j]

            x[i] = (b[i] - S) / U[i, i]

    return x
```

```python
def sostituzione_avanti(L, b):
    # Prendiamo il numero di righe
    n = L.shape[0]

    # Allochiamo spazio per il vettore soluzione
    y = np.zeros_like(b, dtype=np.double)

    # Applichiamo la sostituzione in avanti
    y[0] = b[0] / L[0, 0]

    # Cicliamo al contrario sulle righe(bottom up)
    # Iniziando dalla seconda all'ultima riga
    for i in range(1, n):
        y[i] = (b[i] - np.dot(L[i, :i], y[:i])) / L[i, i]

    return y
```

L'implementazione della sostituzione in avanti è un'implementazione presa da
https://johnfoster.pge.utexas.edu/numerical-methods-book/LinearAlgebra_LU.html

```python
def fattorizzazioneLU(A):
    n = len(A)

    U = A.copy()
    L = np.eye(n)

    for i in range(n):
        m = U[i + 1:, i] / U[i, i]
        L[i + 1:, i] = m
        U[i + 1:] = U[i + 1:] - m[:, np.newaxis] * U[i]

    return L, U
```

```
def fattorizzazioneLU_pivoting(A):
    A = np.copy(A)
    n = len(A)

    indice = np.array(range(n))

    for j in range(n - 1):
        max_A = abs(A[j, j])

        indice_pivot = j

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

        # possibile scambio di righe
        if indice_pivot > j:
            for k in range(n):
                A[indice_pivot, k], A[j, k] = A[j, k], A[indice_pivot, k]
            indice[indice_pivot], indice[j] = indice[j], indice[indice_pivot]

        # eleminazione della colonna j-esima

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

            for k in range(j + 1, n):
                A[i, k] = A[i, k] - A[i, j] * A[j, k]

    L = np.tril(A, - 1) + np.eye(n, n)
    U = np.tril(A)

    return L, U
```

```
def fattorizzazioneLU_pivoting_ottimizzato(A):
    A = np.copy(A)
    n = len(A)

    indice = np.array(range(n))

    for j in range(n - 1):
        # individuazione elemento pivot
        indice_pivot = np.argmax(abs(A[j: n, j])) + j

        # eventuale scambio di righe
        if indice_pivot > j:
            A[[indice_pivot, j]] = A[[j, indice_pivot], :]
            indice[[indice_pivot, j]] = indice[[j, indice_pivot]]

        # eliminazione della colonna j-esima
        for i in range(j + 1, n):
            A[i, j] = A[i, j] / A[j, j]
            A[i, j + 1: n] = A[i, j + 1: n] - A[i, j] * A[j, j + 1: n]

    L = np.tril(A, - 1) + np.eye(n, n)
    U = np.tril(A)

    return L, U
```

Per vedere l'implementazione della soluzione dei precedenti algoritmi:
[implementazione delle soluzioni]

Note:
Qui si trova la pagina di Jupiter Notebook dedicata all'implementazione
[fattorizzazione_LU.ipynb]

Confronto tra algoritmi di interpolazione polinomiale

Sono stati implementati i seguenti algoritmi:

- Interpolazione mediante formula baricentrica di Lagrange
- Interpolazione mediante formula di Newton alle differenze divise
- Formula di Chebyshev per il calcolo dei nodi

Confronto in base al tempo di esecuzione

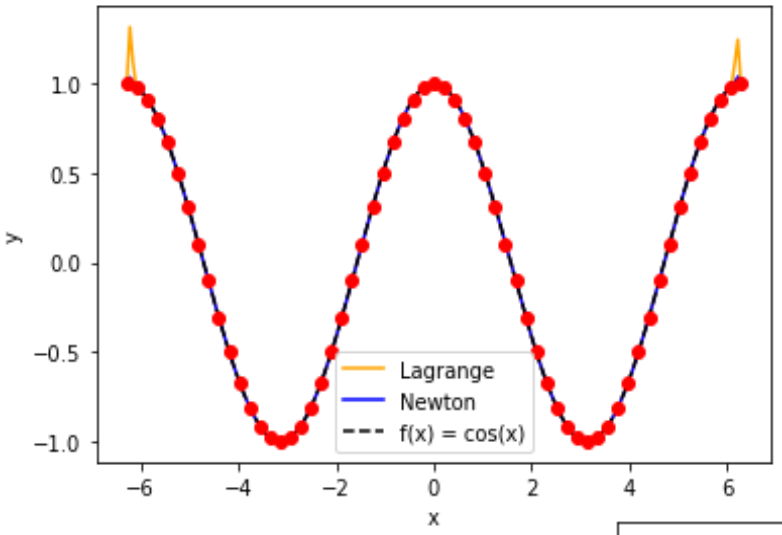
La funzione usata per il test dei metodi di interpolazione è **f(x) = cos(x)**, l'intervallo in cui viene calcolata è [-2pi, 2pi] e il grado del polinomio di interpolazione è 60.

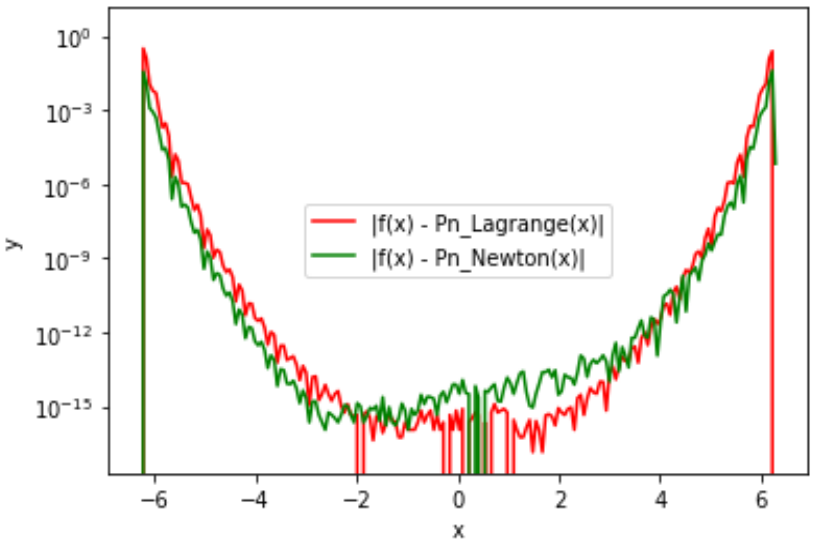
Il confronto viene fatto tra due polinomi:

1. I polinomi di grado 60, prodotto dalla Formula di Lagrange
2. Il polinomio di grado 60, prodotto dalla Formula di Newton

Da quanto osservabile si evince dal primo grafico l'interpolazione della funzione f(x) è praticamente la stessa per i due metodi E' possibile però nel secondo grafico un errore lievemente maggiore per il polinomio di Newton rispetto a quello di Lagrange

```
testInterpolazione(-2 * np.pi, 2 * np.pi, 60, 200, np.cos)
```





Confronto tra nodi equidistanti e nodi di Chebyshev

In questo caso è stata usata come funzione di test dei metodi di interpolazione **f(x) = cos(x)**. La funzione viene calcolata nell'intervallo [-2pi, 2pi] e il grado del polinomio di interpolazione è 60

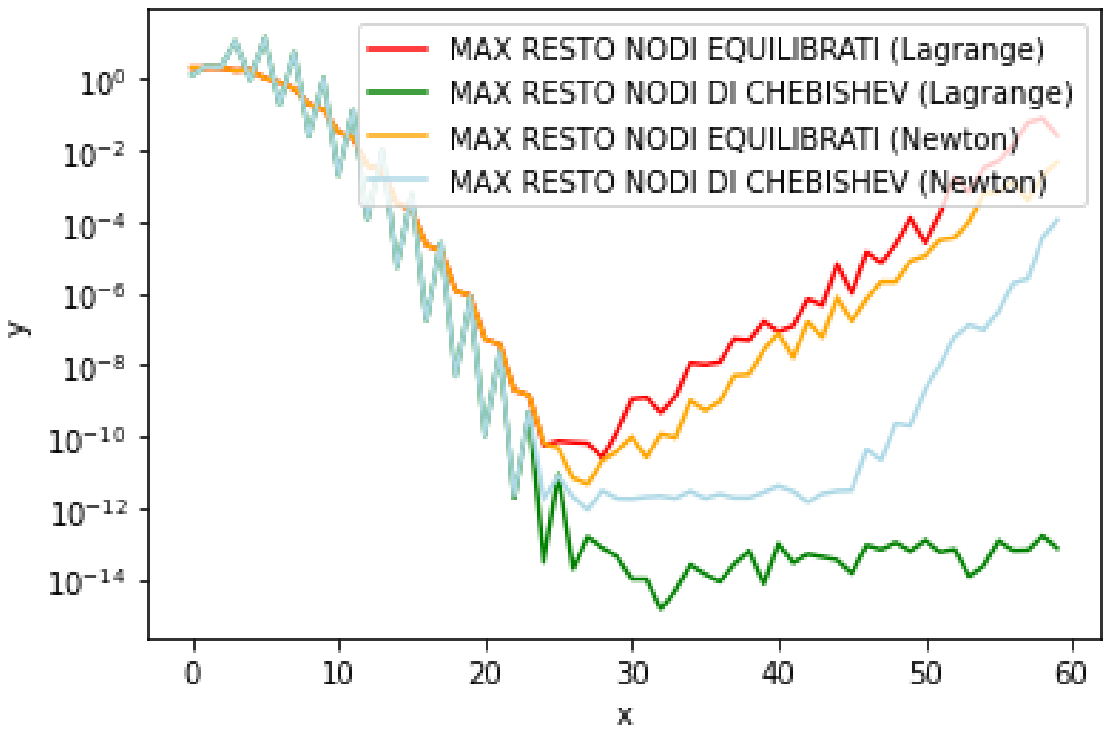
Il confronto viene fatto tra quattro polinomi, di grado 60, prodotti dalle seguenti formule:

- Formula baricentrica di Lagrange e nodi EQUIDISTANTI
- Formula baricentrica di Lagrange e nodi di CHEBYSHEV
- Formula di Newton e nodi EQUIDISTANTI
- Formula di Newton e nodi di CHEBYSHEV

Dai seguenti grafici possiamo notare come il resto del polinomio di interpolazione tende a crescere all'aumentare del grado. Un resto minore lo troviamo nei polinomi in cui i nodi sono stati prodotti dalla formula di Chebyshev con entrambe le formule.

In particolare, nei grafici notiamo che il resto minore lo ritroviamo nel polinomio prodotto dalla Formula di Lagrange con nodi di Chebyshev, tra l'altro i polinomi prodotti con la formula baricentrica di Lagrange tendono ad avere sempre un errore minore rispetto a quelli prodotti con la formula di Newton

```
test_nodi(-2 * np.pi, 2 * np.pi, 200, 60, np.cos)
```



Algoritmi implementati

```
def coefficienti_indeterminati(xn: np.ndarray, yn: np.ndarray, x):
    n = len(xn)
    p = np.zeros(len(x))

    # costruzione della matrice di Vandermonden
    A = np.zeros((n, n))

    for i in range(n):
        for j in range(n):
            A[i, j] = np.power(xn[i], j)

    # calcoliamo i coefficienti indeterminati
    c = soluzione_gauss(A, yn)

    for i in range(len(x)):
        for n in range(len(c)):
            p[i] += c[n] * (np.power(x[i], n))

    return p
```

```
def z_coefficiente(xn: np.ndarray, yn: np.ndarray) -> np.ndarray:
    n: int = len(xn)
    X: np.ndarray = np.eye(n)

    for i in range(n):
        for j in range(n):
            if j > i:
                X[i, j] = xn[i] - xn[j]
            elif j < i:
                X[i, j] = - X[j, i]
    zn: np.ndarray = np.zeros(n)
    for j in range(n):
        zn[j] = yn[j] / np.prod(X[j, :])

    return zn
```

```
def calcola_Lagrange(x: float, xn: np.ndarray, yn: np.ndarray, zn:
np.ndarray) -> float:
    trova_nodi = abs(x - xn) < epsilon_machine()

    if True in trova_nodi:
        temp = np.flatnonzero(trova_nodi == True)
```

```
        j = temp[0]
        pn = yn[j]
    else:
        n = len(xn)
        S = 0
        for j in range(n):
            S = S + zn[j] / (x - xn[j])
        pn = np.prod(x - xn) * S

    return pn
```

```
def metodo_Lagrange(xn: np.ndarray, yn: np.ndarray, x: np.ndarray) ->
np.ndarray:
    len_x: int = len(x)

    # Calcolo i coefficienti formula baricentrica
    zn: np.ndarray = z_coefficiente(xn, yn)

    # calcolo polinomio interpolazione nei punti di x
    p = np.zeros(len_x)
    for i in range(len_x):
        p[i] = calcola_Lagrange(x[i], xn, yn, zn)

    return p
```

```
def differenze_finite(xn: np.ndarray, yn: np.ndarray) -> np.ndarray:
    d: np.ndarray = np.copy(yn)
    n: int = len(yn)

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

    return d
```

```
def calcola_Newton(x: float, xn: np.ndarray, d: np.ndarray) -> float:
    n: int = len(xn) - 1
    p: float = d[n]

    for i in range(n - 1, -1, -1):
        p = d[i] + p * (x - xn[i])

    return p
```

```python
def metodo_Newton(xn: np.ndarray, yn: np.ndarray, x: np.ndarray) ->np.ndarray:
    len_x: int = len(x)
    d: np.ndarray = differenze_finite(xn, yn)
    p: np.ndarray = np.zeros(len_x)

    for i in range(len_x):
        p[i] = calcola_Newton(x[i], xn, d)

    return p
```

```python
def nodi_Chebyshev(a: float, b: float, n: int) -> np.ndarray:
    x = np.empty(n)
    for i in range(n):
        x[i] = a + (np.cos((2*i+1)/(2*n+2)*np.pi) + 1) * (b-a)/2
    return x
```

Per vedere l'implementazione della soluzione dei precedenti algoritmi:
[implementazione delle soluzioni]

Note:
Qui si trova la pagina di Jupiter Notebook dedicata all'implementazione
[Interpolazione.ipynb]

Confronto tra algoritmi per la ricerca delle radici di una funzione

Per effettuare il confronto sono stati implementati i seguenti algorirmi:

- Metodo delle Bisezioni Successive
- Metodo di Newton
- Metodo delle Secanti
- Metodo delle Corde

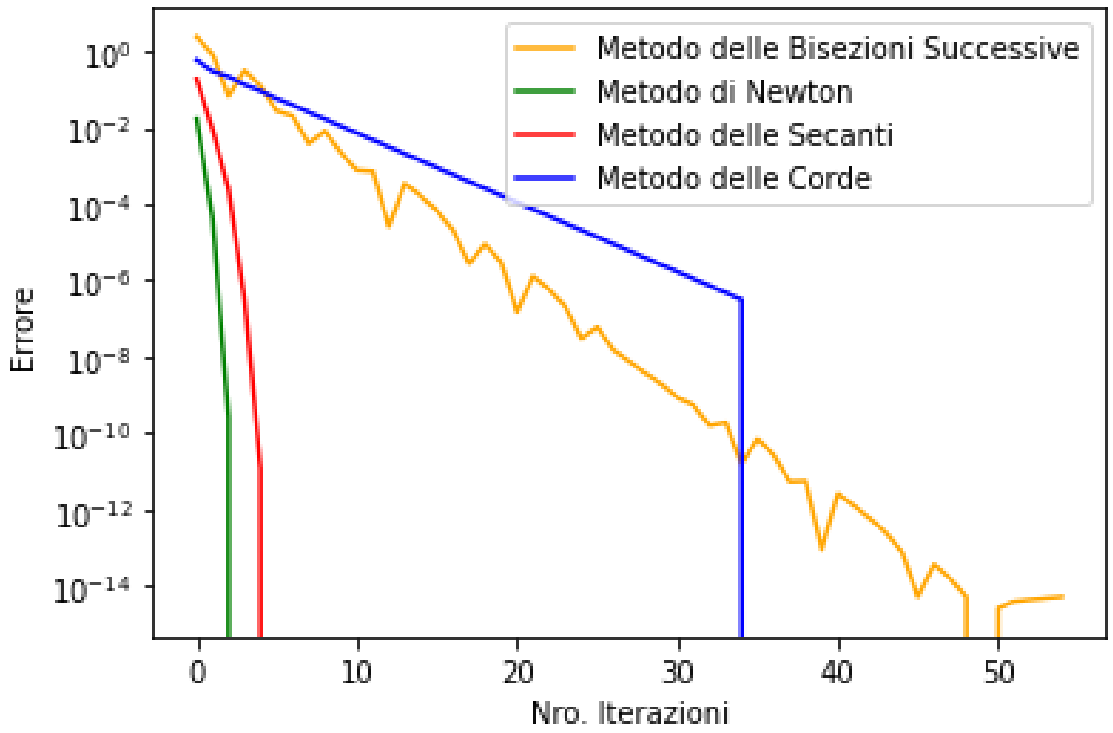
Questi ultimi due sono delle varianti al metodo di Newton. Considerata la funzione f(x) = x^2 - 11, vogliamo studiare come si comportano i diversi metodi al calcolatore. Fissata la tolleranza calcolata dal metodo epsilon_machine() e un numero di iterazioni massimo pari a Kmax = 35 per assicurarci che il calcolo non diverge

Confronto in base alla convergenza

Il grafico mostra la velocità di convergenza dei vari metodi per la ricerca delle radici di una funzione.

Come prima cosa notiamo dal grafico che il metodo che converge più lentamente è quello delle Bisezioni Successive rispetto al più rapido che è il metodo di Newton

L'osservazione principale che si può fare riguarda la convergenza più rapida di tutti gli altri metodi rispetto al metodo delle bisezioni successive, in effetti i metodi piùù rapidi hanno criteri più stringenti come ad esempio la conoscenza a priori della derivata delle funzione (metodo di Newton) o un punto iniziale vicino alla soluzione per gli altri metodi



Algoritmi implementati

```
def metodo_Bisezioni_Successive(a: float, b: float, tolleranza: float, x_reale:
float, fun):
    # Verifica presenza di radici
    fa = fun(a)
    fb = fun(b)

    c = None

    if fa * fb > 0:
        print('Errore: non garantita radice in [%f; %f]' % (a, b))
    else:
        n = math.ceil(math.log2((b - a) / tolleranza)) - 1
        err = np.zeros(n + 1)  # def dell'errore

        for k in range(n + 1):
            c = (a + b) / 2
            fc = fun(c)

            err[k] = abs(x_reale - c)
            if fa * fc < 0:
                b = c
            else:
                a = c
                fa = fc

    return c, err
```

```
def metodo_Newton(x0: float, tolleranza: float, kmax: int, x_reale: float, fun,
dfun):
    fx0 = fun(x0)
    dfx0 = dfun(x0)
    err = []

    iterazioni = 0
    stop = 0

    while not stop and iterazioni < kmax:
        x1 = x0 - fx0 / dfx0
        fx1 = fun(x1)

        stop = abs(fx1) + abs(x1 - x0) / abs(x1) < tolleranza / 5

        err.append(abs(x_reale - x1))

        iterazioni += 1
```

```
        if not stop:
            x0 = x1
            fx0 = fx1
            dfx0 = dfun(x0)

    if not stop:
        print("Accuratezza del metodo raggiunta in %d iterazioni" %
(int(iterazioni)))

    return x1, err
```

```
def metodo_Secanti(x0, x1, tolleranza, k_max, x_reale: float, fun):
    fx0 = fun(x0)
    fx1 = fun(x1)

    err = []

    iterazioni = 0
    stop = 0

    while not stop and iterazioni < k_max:
        x2 = x1 - ((fx1 * (x1 - x0)) / (fx1 - fx0))
        fx2 = fun(x2)

        stop = abs(fx2) + abs(x2 - x1) / abs(x2) < tolleranza / 5

        err.append(abs(x_reale - x2))

        iterazioni += 1

        if not stop:
            x0 = x1
            fx0 = fx1
            x1 = x2
            fx1 = fx2

    if not stop:
        print("Accuratezza del metodo raggiunta in %d iterazioni" %
(int(iterazioni)))

    return x1, err
```

```
def metodo_Corde(x0: float, m, tolleranza, kmax, x_reale: float, fun):
    fx0 = fun(x0)
    err = []

    iterazioni = 0
```

```
    stop = 0

    while not stop and iterazioni < kmax:
        x1 = x0 - fx0 / m
        fx1 = fun(x1)

        stop = abs(fx1) + abs(x1 - x0) / abs(x1) < tolleranza / 5
        err.append(abs(x_reale - x1))

        iterazioni += 1

        if not stop:
            x0 = x1
            fx0 = fx1

    if not stop:
        print("Accuratezza del metodo raggiunta in %d iterazioni" %
(int(iterazioni)))

        return x1, err
```

```
def vettore_standard(err, l_max):
    v = np.array([i - i for i in range(l_max)], dtype=float)

    for i in range(len(err)):
        v[i] = err[i]
    return np.array(v)
```

Per vedere l'implementazione della soluzione dei precedenti algoritmi:
[implementazione delle soluzioni]

Note:
Qui si trova la pagina di Jupiter Notebook dedicata all'implementazione
[Ricerca delle radici di una funzione.ipynb]

Confronto tra algoritmi per il calcolo integrale

Per effettuare il confronto sono stati implementati i seguenti algorirmi:

- Metodo della Formula Composta dei Trapezi
- Metodo della Formula Composta di Simpson
- Metodo della Formula Composta di Bool

Effettuando un test sull'errore assoluto commesso da questi algoritmi al crescere della suddivisione (N) dell'intervallo di integrazione

Il test è stato effettuato su diverse funzioni

Funzioni da confrontare:

- Grado 1: 8x - 4
- Grado 3: 4x^3 - 3x^2 + x - 7
- Grado 5: 9x^5 + 3x^4 + 2x^3 - 5x^2 + 11x - 32

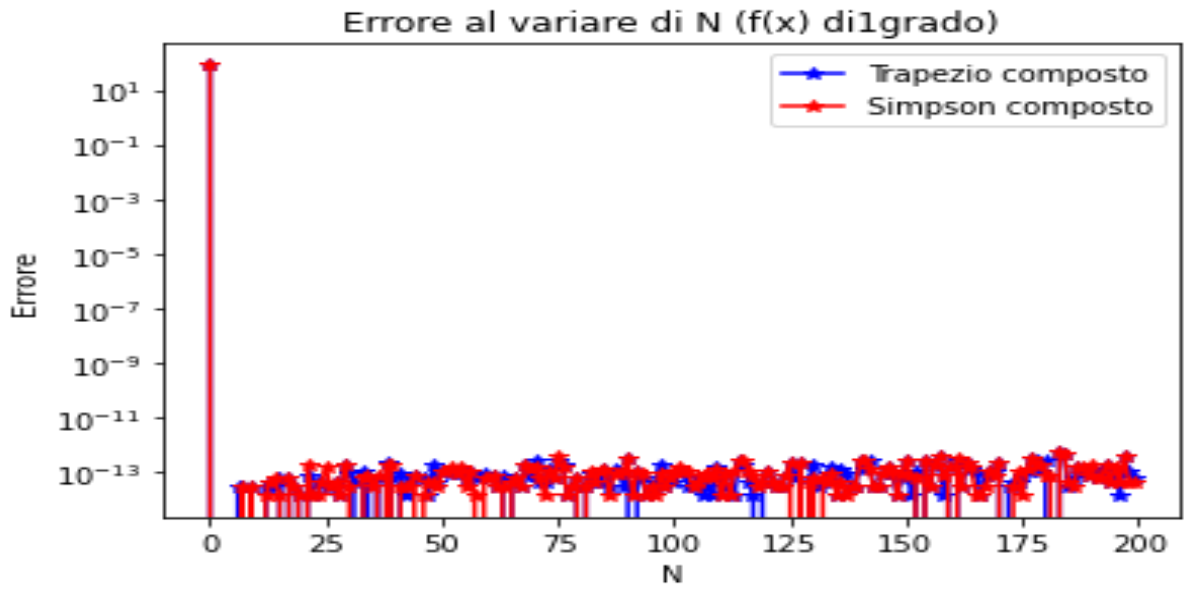
L'intervallo di integrazione è il medesimo per i 3 test, ovvero [-10, 10] Il numero massimo di divisioni dell'intervallo [-10, 10] è stato 200

```
def f1(x):
    return 8*x - 4

def F1(x):
    return 4*x**2 - 4*x

def f3(x):
    return 4*x**3 - 3*x**2 + x - 7

def F3(x):
    return x**4 - x**3 + x**2 / 2 - 7*x

def f5(x):
    return 9*x**5 + 3*x**4 + 2*x**3 - 5*x**2 + 11*x - 32

def F5(x):
    return (3 / 2)*x**6 + (3 / 5)*x**5 + x**4 / 2 - (5 / 3)*x** 3 + (11 / 2)*x**2
- 32*x

def I(F, a, b):
    return F(b) - F(a)


a = -10
b = 10
```

Osservazione dei Risultati

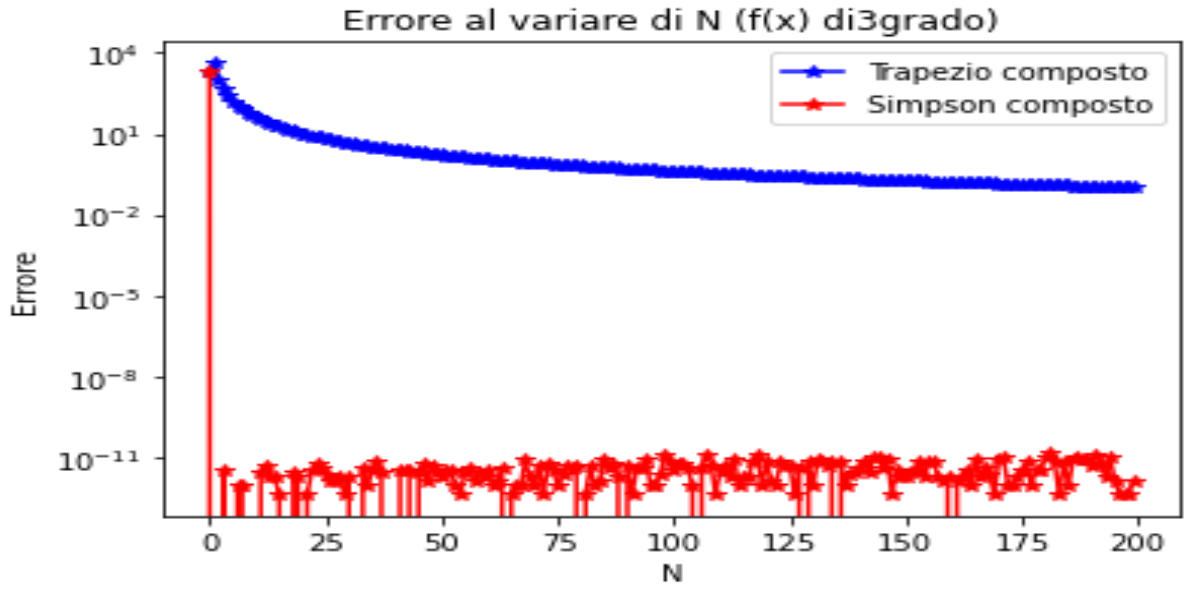
```
test_calcolo_integrale(f1, F1, a, b, 200, 1)
```

Osservando il grafico è notabile che nessuno dei tre metodi produce alti errori nel calcolo dell'integrale della funzione di primo grado



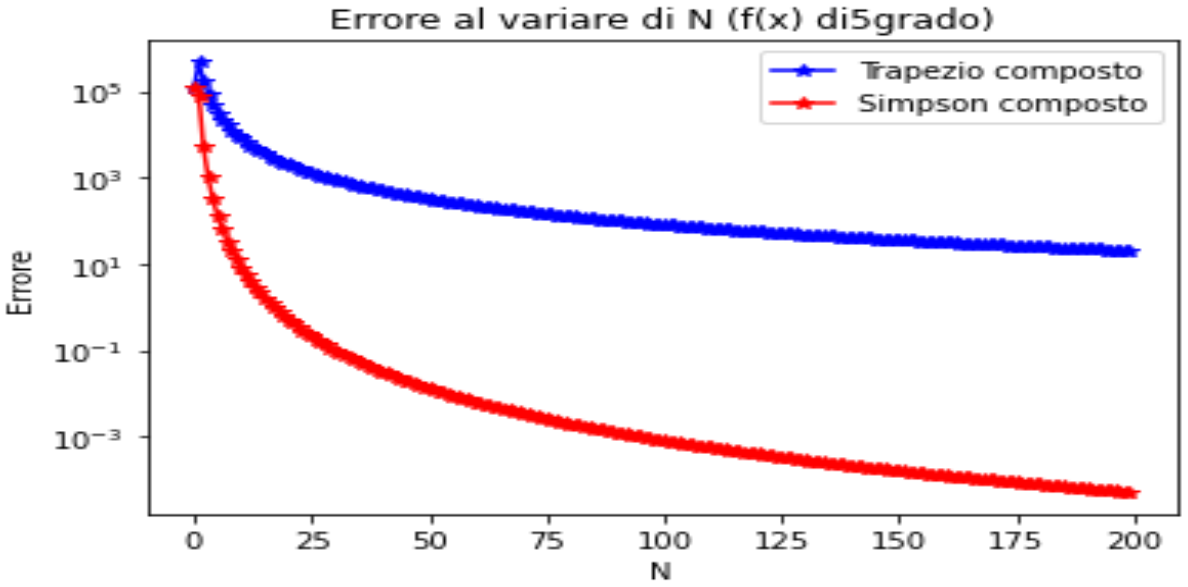

```
test_calcolo_integrale(f3, F3, a, b, 200, 3)
```

In questo grafico è osservabile come la formula composita del Trapezio produca un errore altamente maggiore rispetto a Simpson, questo accade perché la formula composita del trapezio è accurata con polinomi di grado minore o uguale ad 1



```
test_calcolo_integrale(f5, F5, a, b, 200, 5)
```

Medesima cosa accade in questo grafico dove possiamo notare che anche la formula composita di Simpson produce un errore elevato, comunque minore rispetto all'errore prodotto dal metodo del Trapezio composto.



Infine si nota come l'errore cresce all'aumentare del grado della funzione:

- Grado 1: Errore minimo vicino a 10^-13
- Grado 3: Errore minimo vicino a 10^-11
- Grado 5: Errore minimo vicino a 10^-9

Algoritmi implementati

```
def formulaTrapezio(f, a, b):
    return ((b-a)/2) * (f(a) + f(b))
```

```
def formulaSimpson(f, a, b):
    t = np.linspace(a, b, 3)
    return ((t[2] - t[0])/6) * (f(t[0]) + 4*f(t[1]) + f(t[2]))
```

```
def formulaComposta(f, formula, a, b, N):
    s = np.linspace(a, b, N + 1)
    S = []
    for i in range (N):
        S.append(float(formula(f, s[i], s[i + 1])))
    return sum(S)
```

Per vedere l'implementazione della soluzione dei precedenti algoritmi:
implementazione delle soluzioni

Note:
Qui si trova la pagina di Jupiter Notebook dedicata all'implementazione
Calcolo integrale.ipynb

Implementazione delle soluzioni rispetto alle implementazioni

Implementazione metodii per risolvere sistemi di equazione lineare Ax=b

```
def soluzione_gauss(A, b):
    A = np.copy(A)
    b = np.copy(b)

    U, b = eliminazione_gauss(A, b)
    x = sostituzione_indietro(U, b)

    return x
```

```
def soluzione_cramer(A, b):
    x = []
    n = len(A)
    detA = np.linalg.det(A)

    for i in range(n):
        Ai = np.copy(A)
        Ai[:, i] = b
        x.append(float(np.linalg.det(Ai) / np.linalg.det(A)))

    return np.copy(list(x))
```

```
def soluzione_cramer_ottimizzato(A, b):
    x = []
    n = len(A)
    detA = np.linalg.det(A)

    for i in range(n):
        Ai = np.copy(A)
        Ai[:, i] = b
        x.append(float(np.linalg.det(Ai) / detA))

    return np.copy(list(x))
```

```
def soluzione_matrice_inversa(A, b):
    return np.dot(np.linalg.inv(A), b)
```

```
def soluzioneLU(A, b):
    L, U = fattorizzazioneLU(A)

    y = sostituzione_avanti(L, b)
    x = sostituzione_indietro(U, y)

    return x
```

```
def soluzioneLU(A, b):
    L, U = fattorizzazioneLU(A)

    y = sostituzione_avanti(L, b)
    x = sostituzione_indietro(U, y)

    return x
```

```
def soluzioneBuildIn(A, b):
    return np.linalg.solve(A, b)
```

```
def test_fattorizzazione(N):
    t_Soluzione_Gauss = []
    t_soluzione_ALU_senza_Pivot = []
    t_soluzione_Cramer = []
    t_soluzione_Cramer_ottimizzato = []

    t_soluzione_matrice_inversa = []
    t_soluzione_doolittle = []
    t_soluzione_crout = []
    t_build_in = []


    for n in range(2, N + 1, 5):
        A = (2 * np.random.random((n, n)) - 1) * 10
        b = (2 * np.random.random(n) - 1) * 10

        # effettuo una verifica sul determinante osservando che
        # sia !=0 e quindi la matrice non sia singolare

        while np.linalg.det(A) < epsilon_machine():
            A = (2 * np.random.random((n, n)) - 1) * 10


        # eliminazione di gauss
        tempo_iniziale = time.time()
        x_gauss = soluzione_gauss(A, b)
        tempo_finale = time.time()

        t_Soluzione_Gauss.append(float(tempo_finale - tempo_iniziale))


        # cramer
        tempo_iniziale = time.time()
        x_cramer = soluzione_cramer(A, b)
        tempo_finale = time.time()
```

```
        t_soluzione_Cramer.append(float(tempo_finale - tempo_iniziale))

        # cramer ottimizzato
        tempo_iniziale = time.time()
        x_cramer_ott = soluzione_cramer_ottimizzato(A, b)
        tempo_finale = time.time()

        t_soluzione_Cramer_ottimizzato.append(float(tempo_finale -
tempo_iniziale))


        # matrice inversa
        tempo_iniziale = time.time()
        x_matrice_inversa = soluzione_matrice_inversa(A, b)
        tempo_finale = time.time()

        t_soluzione_matrice_inversa.append(float(tempo_finale - tempo_iniziale))


        # alu senza pivot
        tempo_iniziale = time.time()
        x_soluzione_ALU_senza_Pivot = soluzioneLU(A, b)
        tempo_finale = time.time()

        t_soluzione_ALU_senza_Pivot.append(float(tempo_finale - tempo_iniziale))


        # metodo numpy solve
        tempo_iniziale = time.time()
        x_soluzione_Build_In = soluzioneBuildIn(A, b)
        tempo_finale = time.time()

        t_build_in.append(float(tempo_finale - tempo_iniziale))

    plt.figure(1)

    plt.plot(range(2, N + 1, 5), t_Soluzione_Gauss, 'k-',
             label='SOLUZIONE CON MEG')

    plt.plot(range(2, N + 1, 5), t_soluzione_ALU_senza_Pivot, 'y-',
             label='SOLUZIONE CON METODO A=LU')

    plt.xlabel('N')
    plt.ylabel('Tempo')
    plt.legend()
    plt.show()

    plt.plot(range(2, N + 1, 5), t_soluzione_Cramer, 'r-',
             label='SOLUZIONE CON CRAMER')
```

```
    plt.plot(range(2, N + 1, 5), t_soluzione_Cramer_ottimizzato, 'g-',
             label='SOLUZIONE CON CRAMER OTTIMIZZATO')

    plt.xlabel('N')
    plt.ylabel('Tempo')
    plt.legend()
    plt.show()

    plt.plot(range(2, N + 1, 5), t_soluzione_matrice_inversa, 'b-',
             label='SOLUZIONE CON MATRICE INVERSA')

    plt.plot(range(2, N + 1, 5), t_build_in, 'brown',
             label='SOLUZIONE CON METODO np.linalg.solve')

    plt.xlabel('N')
    plt.ylabel('Tempo')
    plt.legend()
    plt.show()

    plt.plot(range(2, N + 1, 5), t_soluzione_ALU_senza_Pivot, 'red',
             label='SOLUZIONE CON MATRICE A=LU')

    plt.plot(range(2, N + 1, 5), t_build_in, 'brown',
             label='SOLUZIONE CON METODO np.linalg.solve')

    plt.xlabel('N')
    plt.ylabel('Tempo')
    plt.legend()
    plt.show()



test_fattorizzazione(200)
```

Implementazione metodi per calcolare l'interpolazione

```
def testInterpolazione(a: float, b: float, n: int, nx: int, fun:
type('function')):
    # Grado ed i nodi di interpolazione
    xn: np.ndarray = np.linspace(a, b, n + 1)
```

```
    yn: np.ndarray = fun(xn)

    # Creazione dell'insieme delle x su cui testare il polinomio di
interpolazione
    x: np.ndarray = np.linspace(a, b, nx)

    # Calcolo della funzione per ogni xi in x
    fx: np.ndarray = fun(x)

    # Calcolo delle x con i polinomi di interpolazione (Lagrange, Newton)
    px_lagrange: np.ndarray = metodo_Lagrange(xn, yn, x)
    px_newton: np.ndarray = metodo_Newton(xn, yn, x)

    plt.plot(1)
    plt.plot(x, px_lagrange, color='orange', label='Lagrange')
    plt.plot(x, px_newton, color='blue', label='Newton')
    plt.plot(x, fx, 'k--', label='f(x) = cos(x)')
    plt.plot(xn, yn, 'ro')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()

    plt.plot(2)
    plt.plot(x, abs(fx - px_lagrange), "red", label='|f(x) -
Pn_Lagrange(x)|')
    plt.plot(x, abs(fx - px_newton), "green", label='|f(x) -
Pn_Newton(x)|')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.yscale('log')
    plt.legend()
    plt.show()
```

```
def test_nodi(a: float, b: float, nx: int, nmax: int, fun:
type('function')):
    x = np.linspace(a, b, nx)
    fx = fun(x)

    resto_eq_lagrange = np.zeros(nmax)
    resto_ch_lagrange = np.zeros(nmax)

    resto_eq_newton = np.zeros(nmax)
    resto_ch_newton = np.zeros(nmax)
```

```
    for n in range(nmax):
        xn_eq = np.linspace(a, b, n + 1)
        yn_eq = fun(xn_eq)

        px_eq_lagrange = metodo_Lagrange(xn_eq, yn_eq, x)
        px_eq_newton = metodo_Newton(xn_eq, yn_eq, x)

        xn_ch = nodi_Chebyshev(a, b, n + 1)
        yn_ch = fun(xn_ch)

        px_ch_lagrange = metodo_Lagrange(xn_ch, yn_ch, x)
        px_ch_newton = metodo_Newton(xn_ch, yn_ch, x)

        resto_eq_lagrange[n] = max(abs(fx - px_eq_lagrange))
        resto_ch_lagrange[n] = max(abs(fx - px_ch_lagrange))

        resto_eq_newton[n] = max(abs(fx - px_eq_newton))
        resto_ch_newton[n] = max(abs(fx - px_ch_newton))

    plt.figure(3)
    plt.semilogy(range(nmax), resto_eq_lagrange, "red", label="MAX RESTO
NODI EQUILIBRATI (Lagrange)")
    plt.semilogy(range(nmax), resto_ch_lagrange, "green", label="MAX RESTO
NODI DI CHEBISHEV (Lagrange)")

    plt.semilogy(range(nmax), resto_eq_newton, "orange", label="MAX RESTO
NODI EQUILIBRATI (Newton)")
    plt.semilogy(range(nmax), resto_ch_newton, "lightblue", label="MAX
RESTO NODI DI CHEBISHEV (Newton)")

    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()
```

Implementazione metodi per trovare gli zeri in una funzione

```
def test_ricerca_radici():
    # soluzione reale del problema

    x_reale = math.sqrt(11)

    # numero massimo iterazioni
    k_max = 35

    # tolleranza per l'arresto del criterio

    tolleranza = epsilon_machine()

    # estremi dell'intervallo
    a = -2
    b = 4

    # ora calcolo la soluzione e il vettore degli errori per il metodo
delle Bisezioni Successive
    sol_BS, err_BS = metodo_Bisezioni_Successive(a, b, tolleranza, x_reale,
f)

    # punto iniziale del metodo di newton
    x0 = 3

    sol_New, err_New = metodo_Newton(x0, tolleranza, k_max, x_reale, f, df)

    # punti iniziali del metodo delle secanti

    x0 = 1
    x1 = 3

    sol_Sec, err_Sec = metodo_Secanti(x0, x1, tolleranza, k_max, x_reale,
f)

    # punti iniziali del metodo Corde
    x0 = 0
    m = 4

    sol_Corde, err_Corde = metodo_Corde(x0, m, tolleranza, k_max, x_reale,
f)

    len_max_err = max(len(err_BS), len(err_New), len(err_Sec),
len(err_Corde))
```

```
    err_BS = vettore_standard(err_BS, len_max_err)
    err_New = vettore_standard(err_New, len_max_err)
    err_Sec = vettore_standard(err_Sec, len_max_err)
    err_Corde = vettore_standard(err_Corde, len_max_err)

    plt.plot(1)
    plt.semilogy(range(len_max_err), err_BS, "orange", label="Metodo delle
Bisezioni Successive")
    plt.semilogy(range(len_max_err), err_New, "green", label="Metodo di
Newton")
    plt.semilogy(range(len_max_err), err_Sec, "red", label="Metodo delle
Secanti")
    plt.semilogy(range(len_max_err), err_Corde, "blue", label="Metodo delle
Corde")

    plt.legend()
    plt.xlabel("Nro. Iterazioni")
    plt.ylabel('Errore')
    plt.show()
```

Implementazione metodi per il calcolo integrale

```
def test_calcolo_integrale(f, F, a, b, N_INTERVALLI, grado):
    err_Trapezio = []
    err_Simpson = []

    I_REALE =  I(F, a, b)

    for N in range(N_INTERVALLI):
        IC_TRAPEZIO = formulaComposta(f,  formulaTrapezio, a, b, N)
        IC_SIMPSON = formulaComposta(f,  formulaSimpson, a, b, N)

        err_T = abs(I_REALE - IC_TRAPEZIO)
        err_S = abs(I_REALE - IC_SIMPSON)

        err_Trapezio.append(err_T)
        err_Simpson.append(err_S)

    plt.figure(1)
    plt.semilogy(range(N_INTERVALLI), err_Trapezio, 'b-*', label='Trapezio
composto')
    plt.semilogy(range(N_INTERVALLI), err_Simpson, 'r-*', label='Simpson
composto')

    plt.xlabel('N')
    plt.ylabel('Errore')
    plt.legend()
    plt.title('Errore al variare di N (f(x) di' + str(grado) + "grado)")
    plt.show()
```