Università degli Studi di Milano Bicocca

DIPARTIMENTO DI INFORMATICA

CORSO DI STUDIO MAGISTRALE IN
INFORMATICA

Metodi del Calcolo Scientifico - Progetto 2 Compressione di immagini tramite la DCT

Membro:
Samuele Toniazzo
Matricola: 918799

Membro:
Emilio Daverio
Matricola: 918799

Anno accademico:
2023/2024

Indice

Capitolo 1

Introduzione al Progetto

1.1 Componenti del gruppo:

Il team di lavoro è composto dalle seguenti persone:

- Emilio Daverio (matricola: 918799)
- Samuele Toniazzo (matricola: 918624)

1.2 Obbietivo del progeto:

Lo scopo di questo progetto è di utilizzare l'implementazione della DCT2 in un ambiente open source e di studiare gli effetti di un algoritmo di compressione tipo jpeg (senza utilizzare una matrice di quantizzazione) sulle immagini in toni di grigio. Comprende la implementazione di un codice e la scrittura di una relazione da consegnare al docente.

1.3 Ambiente di Sviluppo e Lingiuaggio:

Per la realizzazione di questo progetto (sia per la parte 1, sia per la parte 2), abbiamo scelto il linguaggio Python nell'ambiente di sviluppo IntelliJ IDEA. La scelta di Python è stata motivata da diversi fattori, tra cui i suoi notevoli vantaggi nell'ambito dell'elaborazione delle immagini e dello sviluppo di applicazioni desktop. In particolare, Python si distingue per la sua:

- Sintassi semplice e intuitiva, che facilita la scrittura, la lettura e la comprensione del codice, soprattutto quando si affrontano compiti complessi come l'elaborazione delle immagini.
- Natura intuitiva, che rende lo sviluppo del codice più efficiente e accessibile.
- Ampia gamma di librerie specializzate, dedicate all'elaborazione delle immagini e all'analisi dei dati. Queste librerie offrono strumenti potenti e già ottimizzati, evitando la necessità di svilupparli da zero.

Parte I

DCT2 - Discrete Cosine Transform

Capitolo 2

Introduzione al Progetto Parte 1

2.1 Introduzione Parte 1:

Il progetto descritto in questa relazione ha lo scopo di eseguire un'analisi comparativa delle prestazioni di un'implementazione manuale della Discrete Cosine Transform di secondo tipo (DCT2) e di una basata sulla libreria Fast Fourier Transform (FFT).

2.2 Aspettative:

Per raggiungere questo obiettivo, valuteremo come i tempi di esecuzione delle due implementazioni variano al crescere delle dimensioni della matrice fornita in input, includendo la nostra implementazione della DCT2 e quella di una libreria di riferimento. In questo capitolo esamineremo anche come le loro prestazioni si confrontano con le complessità teoriche attese.

Come previsto, è possibile anticipare che l'implementazione manuale della DCT2 avrà un tempo di esecuzione approssimativamente proporzionale a N^3, mentre l'implementazione basata sulla libreria FFT avrà un tempo di esecuzione proporzionale a $N^2 \log(N)$. Questa differenza è dovuta alle diverse strategie algoritmiche utilizzate nei due approcci.

L'analisi dei tempi di esecuzione dei due algoritmi contribuirà a valutarne l'efficienza in relazione alle dimensioni della matrice data in input.

2.2.1 DCT:

La Discrete Cosine Transform (DCT) è un concetto molto importante quando si parla di compressione di un segnale o di un immagine; in quanto è una trasformata matematica utilizzata principalmente, come accennato, per la compressione di segnali e immagini. Converte un segnale o un'immagine dal dominio del tempo (o spazio fisico) al dominio della frequenza. La **DCT** è strettamente correlata alla Trasformata di Fourier, ma utilizza solo coseni invece di seni e coseni. Questa proprietà la rende particolarmente utile per applicazioni in cui i dati sono di natura reale e non complessa, come le immagini. Quello che possiamo dire è che la DTC viene applicata per convertire un vettore di numeri reali (f) in base canonica in un vettore di frequenze (c) (base dei coseni).

$$D_{i,j} = a_l^N \cos\left(\frac{l\pi(2j+1)}{2N}\right) \qquad \text{(DCT Transformation Matrix)}$$

$$a_l^N = \begin{cases} \frac{1}{\sqrt{N}} & \text{se } l = 0 \\ \sqrt{\frac{2}{N}} & \text{altrimenti} \end{cases} \qquad \text{(Normalization Coefficient)}$$

In questo caso abbiamo che:

- $D_{i,j}$: è la matrice di trasformazione che è ottenuta dalla fromula "Normalization Coefficient" e che viene utilizzata per calcolare i coefficienti della DCT
- a_l^N: Fattori di normalizzazione che assicurano che la trasformazione sia ortogonale. Questi valori possono essere: $\frac{1}{\sqrt{N}}$ se $l = 0$ altrimenti $\sqrt{\frac{2}{N}}$ se l è diverso da zero.
- $\cos\left(\frac{l\pi(2j+1)}{2N}\right)$: Parte della formula che esegue la trasformazione cosinusoidale.

La matrice risultante dalla DCT rappresenta il segnale originale in termini di frequenze, permettendo di identificare e mantenere le componenti più significative mentre si scartano quelle meno rilevanti per la compressione dei dati.

Naturalmente il processo è reversibile. Attraverso l'operazione inversa, chiamata **IDCT**, possiamo ricostruire il segnale originale dallo spazio delle frequenze. Tuttavia, la ricostruzione non è perfetta a causa di un troncamento delle frequenze che avviene durante la trasformazione DCT.
In parole semplici, la DCT concentra l'energia del segnale nelle frequenze basse, trascurando quelle più alte. L'IDCT, pur utilizzando le informazioni preservate, non può ripristinare completamente le frequenze eliminate, generando una approssimazione del segnale originale.
Per ottenere una ricostruzione del segnale originale il più fedele possibile mediante l'IDCT, è fondamentale che i coefficienti DCT, indicati come a_i, mantengano valori elevati. Più alto è il valore di un coefficiente a_i, maggiore sarà la quantità di informazioni preservate per quella specifica frequenza e, di conseguenza, più simile sarà il segnale ricostruito da IDCT al segnale originale di partenza.

2.2.2 DCT2:

Le matrici, pur potendo essere interpretate come vettori, possiedono una struttura bidimensionale che le distingue dai vettori unidimensionali. La DCT2 sfrutta questo vantaggio per trasformare le matrici dallo spazio temporale a quello delle frequenze, rivelando informazioni utili per l'analisi e la compressione dei dati. La DCT2 opera su righe e colonne individualmente, preservando la struttura bidimensionale delle matrici. Il processo si articola

in due fasi:
Partendo dalla matrice A abbiamo che:

1. **DCT sulle colonne:** La DCT viene applicata a ciascuna colonna della matrice originale, generando una nuova matrice intermedia (matrice A').
2. **DCT sulle righe:** La DCT viene applicata a ciascuna riga della matrice intermedia (matrice A'), ottenendo la matrice finale nello spazio delle frequenze. La mtrice che otteniamo la chiamia: matrice α

L'idea alla base della matrice α è di eliminare le frequenze basse, conservando solo quelle alte. Questo permette di ridurre la dimensione dei dati da rappresentare, ottenendo una compressione della matrice (dell'immagine).
Anche in questo caso, come nel caso dei vettori, questa strategia è reversibile: applicando l'IDCT2 è possibile ricostruire una matrice (immagine) approssimativa dell'originale. Il risultato che si ottiene deve essere il più vicino possibile alla matrice di partenza.
È importante sottolineare che la tecnica descritta sopra, seppur efficace, non è quella utilizzata dal formato JPEG per la compressione delle immagini. Il motivo risiede in alcune operazioni critiche che possono introdurre distorsioni e artefatti nella ricostruzione finale. Il formato JPEG adotta soluzioni specifiche per ovviare a queste problematiche e ottenere una compressione efficiente pur mantenendo una buona qualità dell'immagine.

Capitolo 3

Descrizione Codice Parte 1

3.1 Repository del Codice:

Il codice del progetto che stato sviluppato è disponibile si gitHub al seguente link: Link del progetto. Il repository Git del progetto è strutturato in due cartelle principali: "Prima_parte" e "Seconda_parte". All'interno di ciascuna cartella si trovano le classi che contengono il codice sorgente sviluppato e funzionante. I codice che interessa a noi per questa parte di progetto si trova nella cartella Prima_parte

3.2 Descrizione del Codice:

Il nostro progetto per la parte 1 è stato sviluppato nelle seguenti classi:

- utilss.py
- test.py
- generate_graphs.py
- ___init___.py

3.3 Classe utilss.py:

Questa classe implementa diverse funzioni per la trasformazione discreta del coseno (DCT) e la sua variante bidimensionale (DCT2). L'obiettivo principale è confrontarle con le implementazioni equivalenti presenti nella libreria Scipy. Scipy offre funzionalità avanzate per la trasformazione discreta di Fourier (FFT) e altre trasformazioni correlate, tra cui una versione ottimizzata dell'algoritmo DCT2 con elevata efficienza computazionale. Il confronto tra le nostre implementazioni e quelle di Scipy mira a valutare le seguenti caratteristiche:

- tempo: necessario per eseguire la trasformata DCT2
- coplessità computazionale

I metodi implementati sono i seguenti:

3.3.1 Metodo per la creazione della DCT:

```
# Funzione per DCT creata
def dct_created(input_vector):
    # Creazione della matrice di trasformazione
    transformation_matrix =
    ↪  create_transformation_matrix(input_vector)

    # Calcolo del risultato (c) come la moltiplizazione tra la
    ↪  matrice di trasformazione con il vettore dato in input
    ↪  (f)
    dct_result = np.dot(transformation_matrix, input_vector)
    return dct_result
```

Questa funzione implementa la trasformazione discreta del coseno (DCT) sfruttando una matrice di trasformazione generata dalla funzione create_transformation_matrix. Quest'ultima si occupa di costruire la matrice necessaria per il calcolo della DCT. La funzione in oggetto riceve in ingresso un vettore monodimensionale input_vector e restituisce il risultato della trasformazione DCT applicata.

3.3.2 Metodo per la creazione della matrice di trasformazione:

```
def create_transformation_matrix(a):
# Calcolo della lunghezza dell'array
n = len(a)

# Creazione del vettore alfa lungo quanto il vettore passato
alpha = np.zeros(n)

# Calcolo dei valori in base alla posizione
alpha[0] = 1 / np.sqrt(n)
alpha[1:] = np.sqrt(2/n)

# Creazione della matrice di trasformazione
transformation_matrix = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        transformation_matrix[i, j] = alpha[i] * np.cos((i *
        ↪  math.pi * (2 * j + 1)) / (2 * n))
return transformation_matrix
```

In questo caso abbiamo che la funzione prende in input un array a di cui ne calcoliamo la lunghezza. In uotput quello che otterremo sarà una matrica di trasformazione DCT di dimensione n x n, dove n è la lunghezza dell'array a. Questa funzione calcola i valori della matrice di trasformazione DCT in base alla formula standard, ovvero ottenuta la dimensione del vetore a, e calcolati i coefficienti $alpha$,nel seguente modo:

$$alpha[0] = 1/sqrt(n) \tag{3.1}$$

$$alpha[1:] = sqrt(2/n) \tag{3.2}$$

si passa a riemopire la matrice di trasformazione con i valori di alpha calcolati seguendo i seguenti passaggi:

- Viene creata una matrice vuota di dimensioni n x n
- Ogni elemento della matrice viene calcolato seguendo la formula descritta precedentemente
- L'elemento appena calcolato viene aggiunto nella posizione i, j secondo duecicli for innestati tra loro
- L'ordine di riempimento prevede l'inserimento dei valori prima per le righe e successivamente per le colonne

In fine una volta terminati i cicli, viene restituita la matrice.

3.3.3 Metodo per la creazione della DCT2:

```
def dct2_created(input_matrix):
n, m = input_matrix.shape

# Crezione della matrice
dct2_result = np.copy(input_matrix.astype('float64'))

# DCT per ogni colonna
for j in range(m):
    dct2_result[:, j] = dct_created(dct2_result[:, j])

# DCT per ogni riga
for i in range(n):
    dct2_result[i, :] = dct_created(dct2_result[i, :])

return dct2_result
```

Questo metodo implementa la trasformazione discreta del coseno bidimensionale (DCT 2D) su una matrice di input di dimensioni arbitrarie. Restituisce una matrice contenente il risultato della trasformazione applicata alla matrice di input.

1. La funzione DCT viene richiamata per ogni colonna della matrice di input. Il risultato di ogni chiamata viene memorizzato nella corrispondente colonna della matrice di output.

2. La matrice di output ottenuta dal passaggio precedente viene trasposta (righe <–> colonne). Ovvero la funzione dct viene richiamata per ogni riga della matrice trasposta. Il risultato di ogni chiamata viene memorizzato nella corrispondente riga della matrice di output finale.

3.3.4 Metodo per l'utilizzo di DCT e DCT2 dalle librerie:

```
# Funzione DCT implementata da libreria esterna
def dct_library(f):
    return dct(f, norm='ortho')

# Funzione DCT2 implementata da libreria esterna
def dct2_library(f):
    return dct(dct(f.T, norm='ortho').T, norm='ortho')
```

Abbiamo utilizzato la libreria open-source SciPy per sfruttare le funzioni DCT e DCT2 implementate al suo interno. Importando il pacchetto FFT, è stato possibile utilizzare la funzione DCT della libreria per confrontare i tempi di esecuzione con quelli della nostra implementazione. La funzione per la DCT prende in input i seguenti parametri:

- Un array monodimensionale di cui deve essere calcolata la DCT.
- Il parametro T, che permette di calcolare il trasposto dell'array, replicando il calcolo della DCT da noi implementato che prevede l'applicazione della trasformata prima sulle colonne e successivamente sulle righe.
- Il parametro ortho, utile per indicare alla funzione DCT di considerare anche i valori alpha durante il calcolo della matrice di trasformazione.

La funzione DCT2 prende in input gli stessi parametri della funzione DCT, ma viene richiamata due volte per creare la matrice bidimensionale.

Concluso lo sviluppo di questa classe, abbiamo creato la classe test.py dove abbiamo implementato i test richiesti dalla traccia del progetto.

3.4 Classe test.py:

Il file `test.py` è progettato per eseguire una serie di test e misurazioni di performance delle trasformate discrete del coseno (DCT e DCT2) applicate su matrici di dati. Analizzando il codice, inizialmente vengono importate le librerie necessarie come `numpy`, utile per eseguire operazioni matematiche, e `timeit`, che ha il compito di misurare i tempi di esecuzione delle funzioni.

3.4.1 Prima Parte - Verifica dell'Accuratezza:

La funzione `run_test()` esegue i test di accuratezza delle implementazioni della DCT e DCT2 confrontandole con i risultati ottenuti utilizzando la libreria `scipy`. Questo codice è utile per verificare se le funzioni implementate restituiscono risultati corretti. Di seguito è riportata una descrizione dettagliata della funzione e il codice associato:

```
def run_test():
    # Test DCT monodimensionale
    a = np.array([231, 32, 233, 161, 24, 71, 140, 245])
    dct = utils.dct_created(a)
    print("\n--- TEST DCT HomeMade ---")
    print(["{:.2e}".format(val) for val in dct])

    # Matrice di test 8x8
    input_matrix = np.array([
        [231, 32, 233, 161, 24, 71, 140, 245],
        [247, 40, 248, 245, 124, 204, 36, 107],
        [234, 202, 245, 167, 9, 217, 239, 173],
        [193, 190, 100, 167, 43, 180, 8, 70],
        [11, 24, 210, 177, 81, 243, 8, 112],
        [97, 195, 203, 47, 125, 114, 165, 181],
        [193, 70, 174, 167, 41, 30, 127, 245],
        [87, 149, 57, 192, 65, 129, 178, 228]])

# DCT2 Homemade
dct2_result = utilss.dct2_created(input_matrix)
expected_dct2_result = np.array([
    [1.11e+03, 4.40e+01, 7.59e+01, -1.38e+02, 3.50e+00,
    ↪  1.22e+02, 1.95e+02, -1.01e+02],
    [7.71e+01, 1.14e+02, -2.18e+01, 4.13e+01, 8.77e+00,
    ↪  9.90e+01, 1.38e+02, 1.09e+01],
    [4.48e+01, -6.27e+01, 1.11e+02, -7.63e+01, 1.24e+02,
    ↪  9.55e+01, -3.98e+01, 5.85e+01],
    [-6.99e+01, -4.02e+01, -2.34e+01, -7.67e+01, 2.66e+01,
    ↪  -3.68e+01, 6.61e+01, 1.25e+02],
    [-1.09e+02, -4.33e+01, -5.55e+01, 8.17e+00, 3.02e+01,
    ↪  -2.86e+01, 2.44e+00, -9.41e+01],
    [-5.38e+00, 5.66e+01, 1.73e+02, -3.54e+01, 3.23e+01,
    ↪  3.34e+01, -5.81e+01, 1.90e+01],
    [7.88e+01, -6.45e+01, 1.18e+02, -1.50e+01, -1.37e+02,
    ↪  -3.06e+01, -1.05e+02, 3.98e+01],
    [1.97e+01, -7.81e+01, 9.72e-01, -7.23e+01, -2.15e+01,
    ↪  8.13e+01, 6.37e+01, 5.90e+00]
])
```

```
print("\n--- TEST DCT2 HomeMade ---")
print(dct2_result)

# Verifica con la libreria scipy
dct_lib = utils.dct_library(a)
print("\n--- TEST DCT Library ---")
print(["{:.2e}".format(val) for val in dct_lib])

dct2_lib_result = utils.dct2_library(input_matrix)
print("\n--- TEST DCT2 Library ---")
print(dct2_lib_result)
```

La funzione verifica se le implementazioni personalizzate della DCT e DCT2 restituiscono risultati corretti confrontandoli con quelli ottenuti dalla libreria `scipy`. La matrice di test 8x8 e i risultati attesi della DCT2 sono specificati nel progetto, e l'esecuzione del codice conferma che l'implementazione è corretta.

3.4.2 Seconda Parte - Misurazione delle Prestazioni:

La funzione `test_N()` esegue una serie di test di performance su matrici di dimensioni crescenti (NxN), misurando il tempo impiegato per la risoluzione di DCT e DCT2. Questo è cruciale per valutare l'efficienza degli algoritmi implementati rispetto a quelli della libreria `scipy`.

```
def test_N():
    matrix_dimensions = list(range(50, 1001, 50))
    times_scipy_dct = []
    times_my_dct = []

    for n in matrix_dimensions:
        np.random.seed(5)
        matrix = np.random.uniform(low=0.0, high=255.0,
        ↪  size=(n, n))

        time_scipy = timeit.timeit(lambda:
        ↪  utilss.dct2_library(matrix), number=1)
        times_scipy_dct.append(time_scipy)

        time_my_dct = timeit.timeit(lambda:
        ↪  utilss.dct2_created(matrix), number=1)
        times_my_dct.append(time_my_dct)

    return times_scipy_dct, times_my_dct, matrix_dimensions
```

Questa funzione genera matrici casuali di dimensioni crescenti, misura i tempi di esecuzione della DCT2 utilizzando sia l'implementazione personalizzata sia

la libreria `scipy`, e restituisce i tempi di esecuzione insieme alle dimensioni delle matrici testate.

3.4.3 Visualizzazione dei Risultati:

I tempi di esecuzione sono visualizzati graficamente utilizzando la funzione `generate_graphs.plot_dct_times()` per confrontare le prestazioni delle diverse implementazioni. Questo è fondamentale per analizzare come i tempi di esecuzione crescono con le dimensioni della matrice e per valutare l'efficienza relativa delle implementazioni.

3.4.4 Risultati:

Accuratezza:

I risultati ottenuti con l'implementazione personalizzata della DCT2 sono risultati coerenti con quelli attesi, dimostrando l'accuratezza dell'implementazione. La DCT applicata alla prima riga della matrice di test ha fornito risultati accurati rispetto a quelli attesi. I risultati ottenuti utilizzando la libreria `scipy` sono risultati coerenti con quelli attesi.

Prestazioni:

I tempi di esecuzione delle implementazioni della DCT2 sono stati misurati e registrati per diverse dimensioni di matrici. I risultati mostrano una differenza significativa nelle prestazioni tra l'implementazione personalizzata e quella della libreria `scipy`.

3.5 Classe generate_graphs.py:

La classe in questione è progettata per confrontare le prestazioni della Trasformata Discreta del Coseno (DCT) e della Trasformata Discreta del Coseno bidimensionale (DCT2) tra due implementazioni: una fornita dalla libreria SciPy e una sviluppata manualmente. La classe offre strumenti per calcolare i tempi di esecuzione di entrambe le implementazioni su array di diverse dimensioni e per visualizzare graficamente questi tempi di esecuzione, permettendo così un'analisi comparativa. Il metodo della classe che confronta i tempi delle due DCT è plot_dct_times(times_scipy_dct, times_my_dct, matrix_dimensions). Questo metodo genera un grafico che mostra i tempi di esecuzione. I valori attesi sono proporzionali a N^3 per la DCT2 implementata manualmente e a $N^2 * log(N)$ per l'implementazione del pacchetto FFT di SciPy. Per un confronto accurato, vengono utilizzate matrici N×N, con dimensioni che vanno da 50x50 a 1000x1000, incrementate di 50 unità ad ogni iterazione.

3.5.1 Metodo per la generazione del grafico:

Questo metodo genera un grafico che confronta i tempi di esecuzione della DCT2 utilizzando la libreria SciPy e la propria implementazione della DCT2. Il grafico include anche curve di riferimento per la complessità computazionale n^3 e $n^2 log(n)$. Quello che posiamo dire è che questo metodo prenderà:
In input gli vengono passati i seguenti parametri:

- **times_scipy_dct (array)**: I tempi di esecuzione della DCT2 utilizzando la libreria SciPy.
- **times_my_dct (array)**: tempi di esecuzione della propria implementazione della DCT2.
- **matrix_dimensions (array)**: Le dimensioni degli array su cui sono state eseguite le trasformate.

```
def plot_dct_times(times_scipy_dct, times_my_dct,
 ↪  matrix_dimensions):
# Calcolo delle curve di riferimento per n^3 e n^2 * log(n)
# Dividiamo per 10^6 e 10^8 rispettivamente per scalare i
 ↪  valori in modo che siano comparabili con i tempi di
 ↪  esecuzione
n3 = [n**3 /1e5 for n in matrix_dimensions]
n2_logn = [n**2 * np.log(n) / 1e8 for n in matrix_dimensions]

# Creazione della figura del grafico con dimensioni 10x6
 ↪  pollici
plt.figure(figsize=(10, 6))

# Aggiunta della curva dei tempi di esecuzione della DCT2
 ↪  utilizzando la libreria scipy
plt.semilogy(matrix_dimensions, times_scipy_dct, label='Library
 ↪  DCT2', color="tab:green")
# Aggiunta della curva di riferimento n^2 * log(n)
plt.semilogy(matrix_dimensions, n2_logn, label='n^2 * log(n)',
 ↪  color="tab:green", linestyle='dashed')

# Aggiunta della curva dei tempi di esecuzione della tua
 ↪  implementazione della DCT2
plt.semilogy(matrix_dimensions, times_my_dct, label='DCT2
 ↪  created', color="tab:blue")
# Aggiunta della curva di riferimento n^3
plt.semilogy(matrix_dimensions, n3, label='n^3',
 ↪  color="tab:blue", linestyle='dashed')
```

```
# Impostazione delle etichette degli assi e del titolo del
↪ grafico
plt.xlabel('Dimensione N')
plt.ylabel('Tempo di esecuzione in secondi')
plt.title('Tempi di esecuzione della DCT2 al variare della
↪ dimensione N')

# Aggiunta della legenda per identificare le diverse curve
plt.legend()

# Aggiunta di una griglia al grafico
plt.grid(True)

# Salvataggio dell'immagine del grafico in un file PNG
#plt.savefig('Parte1/untiled/grafico_dct_times.png')

# Visualizzazione del grafico
plt.show()
```

In output, otteniamo un grafico che confronta i tempi di esecuzione delle due DCT2. Questo grafico rappresenta sull'asse delle y il tempo di esecuzione necessario per eseguire le due DCT2 all'aumentare della dimensione N della matrice, mentre sull'asse delle x viene visualizzata la dimensione della matrice. I tempi di esecuzione sono riportati in scala logaritmica, mostrando quanto tempo la funzione ha impiegato per eseguire la trasformata.

Analizzando i risultati riportati nel grafico 3.1, si osserva che il tempo di esecuzione delle implementazioni della DCT segue, come previsto, un andamento proporzionale. In particolare, l'utilizzo del metodo basato su FFT (con Scipy.fft) consente un notevole miglioramento delle prestazioni per le matrici nella DCT2. Tuttavia, è importante sottolineare che il tempo di esecuzione nel caso della DCT2 non sempre presenta un andamento lineare. Questo comportamento è dovuto a diversi fattori interni alla libreria utilizzata, tra cui ottimizzazioni implementate, gerarchia di cache e parallelismo nei calcoli. Tali dinamiche possono generare variazioni non monotone nei tempi di esecuzione al variare della dimensione della matrice (N), determinando scostamenti rispetto a una semplice relazione lineare. Di conseguenza, sebbene ci si possa attendere un aumento costante del tempo di esecuzione all'aumentare di N, la presenza di questi fattori può causare fluttuazioni o, in alcuni casi, un apparente calo delle prestazioni.

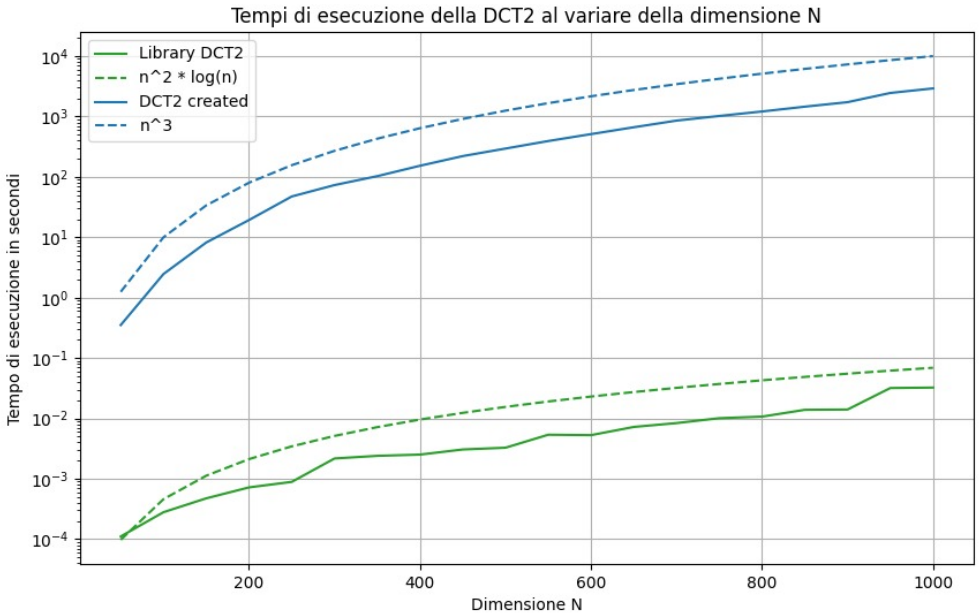

Figura 3.1: Tempi di esecuzione della DCT2 al variare della dimensione N

Per evitare equivoci, la linea intera di colore blu rappresenta l'andamento della DCT2 implementata da noi, mentre la linea intera di colore verde rappresenta l'andamento della funzione DCT2 resa disponibile dalla libreria SciPy. La linea tratteggiata blu rappresenta la complessità teorica N^3 della nostra implementazione, mentre la linea tratteggiata verde mostra la complessità teorica $N^2log(N)$ della versione ottimizzata della libreria. Come si può osservare, la nostra implementazione ha un tempo di esecuzione che cresce più rapidamente con l'aumentare della dimensione N, confermando la complessità teorica attesa. Al contrario, l'implementazione della libreria SciPy risulta significativamente più efficiente per dimensioni elevate, grazie alla complessità inferiore $N^2log(N)$

3.6 Problemi Riscontrati:

In questo progetto, abbiamo riscontrato un problema legato al fattore di scaling della Trasformata Coseno Discreta (DCT), il quale differiva da quello teorico. Questo fattore può variare a seconda della definizione e della libreria utilizzata. In particolare, nella libreria SciPy, la DCT è scalata in modo da ottenere una base ortonormalizzata. Questo significa che la trasformata inversa della DCT applicata ai coefficienti della DCT restituisce esattamente il segnale originale.
La formula generale per la DCT di tipo II, comunemente utilizzata, può essere scritta come:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \tag{3.3}$$

SciPy usa una forma scalata della DCT che rende la trasformata ortonormale. La DCT di tipo II ortonormalizzata è definita come:

$$X_k = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \tag{3.4}$$

Per il caso particolare di X_0 si fa:

$$X_0 = \sqrt{\frac{1}{N}} \sum_{n=0}^{N-1} x_n \tag{3.5}$$

Questo si traduce in un fattore di scaling che include una radice quadrata, normalizzando la trasformata in modo che le basi siano ortonormali. Per il calcolo della DCT ortonormalizzata utilizzando SciPy, le formule di riferimento cambiano leggermente rispetto a quelle teoriche. Quando si utilizza la normalizzazione ortonormalizzata (norm='ortho'), vengono aggiunti dei fattori di radice quadrata per garantire l'ortonormalità delle basi.
Per risolvere questo problema, abbiamo aggiunto una radice al denominatore nel calcolo dei valori a_i^n. Per garantire la correttezza, abbiamo consultato la documentazione disponibile al seguente link:https://docs.scipy.org/doc/scipy/reference/generated/scipy.fftpack.dct.html

3.7 Conclusione parte 1:

Le funzioni da noi create hanno soddisfatto le aspettative, rispettando i tempi di esecuzione previsti. Abbiamo compreso il motivo e l'importanza dell'uso della DCT2 ottimizzata tramite FFT, evidenziando la rilevanza degli algoritmi ottimizzati per operazioni computazionalmente intensive come la DCT2. Mentre l'implementazione manuale della DCT2 è utile per scopi didattici e per comprendere i principi di base, l'uso di librerie ottimizzate è essenziale per applicazioni pratiche su larga scala, dove l'efficienza computazionale è cruciale.

Parte II

Compressione JPEG

Capitolo 4

Introduzione al Progetto Parte 2

4.1 Introduzione Parte 2:

La seconda parte del progetto si focalizza sulla compressione di immagini in formato .bmp utilizzando la Trasformata Coseno Discreta di secondo tipo (DCT2), una tecnica ampiamente utilizzata per simulare la compressione JPEG. L'obiettivo principale è sviluppare un'applicazione che permetta all'utente di scegliere un'immagine e regolare il livello di compressione, visualizzando sia l'immagine originale che quella compressa.

4.2 Aspettative:

Ci aspettiamo che l'utente possa selezionare una foto in bianco e nero attraverso un'interfaccia user-friendly realizzata con la grafica dell'ambiente di sviluppo. L'utente deve poter specificare i parametri di compressione (f, d) seguendo criteri appropriati. Se tutto è configurato correttamente, l'utente potrà applicare la compressione e visualizzare il risultato ottenuto. Ovvero solo una volta inseriti entrambi i parametri, l'utente può avviare il processo di compressione.

4.2.1 Parametri di compressione:

I parametri di compressione, in particolare il Fattore di compressione (F) e il Parametro di quantizzazione (d), svolgono ruoli importanti nel processo di compressione delle immagini. Questi parametri permettono di bilanciare la qualità dell'immagine con la riduzione delle dimensioni del file. La loro regolazione consente di ottenere risultati ottimali per diverse applicazioni e requisiti di archiviazione.

Fattore di compressione (F):

Determina la dimensione dei blocchi in cui suddividere l'immagine. Ovvero Indica quanto l'immagine viene compressa rispetto alla sua dimensione originale. Un valore più alto indica una maggiore compressione e una dimensione file più piccola.

Parametro di quantizzazione (d):

Controlla il livello di compressione applicato a ciascun blocco. Ovvero controlla il grado di "approssimazione" applicato ai valori dei pixel durante la compressione. Valori più alti di d portano a una maggiore compressione ma anche a una potenziale perdita di qualità. Influenza direttamente il compromesso tra dimensione del file e qualità dell'immagine.

4.2.2 Algoritmo JPEG

L'algoritmo JPEG (Joint Photographic Experts Group) è un metodo standard di compressione delle immagini che riduce la quantità di dati necessari per rappresentare un'immagine senza una significativa perdita di qualità visiva. Questo algoritmo sfrutta le caratteristiche della percezione visiva umana e le proprietà matematiche delle immagini per ottenere una compressione efficace.

4.2.3 Cosa risolve l'algoritmo JPEG

L'algoritmo JPEG risolve diversi problemi legati alla compressione delle immagini, tra cui:

- **Costo Computazionale Elevato**: La procedura di compressione tramite la Discrete Cosine Transform (DCT) e la sua inversa (IDCT) ha un costo computazionale elevato, specialmente per immagini di grandi dimensioni. Il JPEG risolve questo problema applicando la DCT su blocchi di 8x8 pixel, riducendo significativamente la complessità computazionale complessiva. Suddividendo l'immagine in blocchi più piccoli, l'algoritmo riduce il numero di operazioni necessarie, migliorando l'efficienza.
- **Fenomeno di Gibbs**: Il fenomeno di Gibbs si manifesta quando una funzione discontinua viene approssimata mediante una serie di Fourier troncata, producendo oscillazioni vicino ai punti di discontinuità. Questo fenomeno è visibile come artefatti lungo i bordi netti delle immagini. Nel contesto della compressione JPEG, l'uso della DCT aiuta a ridurre l'impatto del fenomeno di Gibbs, poiché lavora su blocchi separati di 8x8 pixel. Tuttavia, possono ancora verificarsi artefatti dovuti alle discontinuità tra i blocchi. Per mitigare questo effetto, JPEG utilizza tecniche come:
  - **Sovrapposizione dei Blocchi**: Per ridurre le discontinuità tra blocchi adiacenti.
  - **Regolazione della Quantizzazione**: Bilanciare il livello di compressione per ridurre gli artefatti.
- **Gestione dei Numeri con la Virgola**: Durante la ricostruzione dell'immagine, i valori ottenuti dalla IDCT possono non essere interi e fuori

dall'intervallo [0, 255]. Per risolvere questo problema, si applicano operazioni di arrotondamento e clipping dei valori per garantire che rientrino nell'intervallo desiderato. Le operazioni sono le seguenti:

$$a_{ij} = \text{round}(a_{ij})$$

$$a_{ij} = \max(a_{ij}, 0)$$

$$a_{ij} = \min(a_{ij}, 255)$$

Questo assicura che i valori siano interpretabili come livelli di colore dal computer.

Queste tecniche aiutano a mantenere un compromesso accettabile tra la dimensione del file compresso e la qualità visiva dell'immagine.

Capitolo 5

Descrizione Codice Parte 2

5.1 Repository del Codice:

Il codice del progetto che stato sviluppato è disponibile si gitHub al seguente link: Link del progetto. Il repository Git è strutturato in due cartelle principali: "Prima_parte" e "Seconda_parte". All'interno di ciascuna cartella si trovano le classi che contengono il codice sorgente sviluppato e funzionante. Il codice rilevante per questa parte del progetto si trova nella cartella "Seconda_parte. Inoltre All'interno della cartella "Seconda_parte", oltre alle classi.py, è presente una sottocartella denominata "Immagini". Questa contiene le immagini utilizzate per verificare la correttezza del codice e il corretto funzionamento della compressione JPEG.

5.2 Descrizione del Codice:

Il nostro progetto per la parte 2 è stato sviluppato nelle seguenti classi:

- utilss.py
- main.py

5.3 Classe utilis.py

La classe DCTApp è un'applicazione GUI costruita con il framework tkinter, progettata per gestire la compressione di immagini in formato .bmp utilizzando la Trasformata Coseno Discreta 2D (DCT2). L'applicazione permette all'utente di selezionare un'immagine, inserire i parametri di compressione, e visualizzare sia l'immagine originale che quella compressa. La GUI è composta da vari widget per l'interazione dell'utente, inclusi bottoni, etichette e campi di input.
L'applicazione dividerà l'immagine in blocchi di dimensione FxF, applicherà la DCT2 a ciascun blocco e quindi quantizzerà i coefficienti DCT2 basandosi sul valore del parametro d. I blocchi compressi verranno poi ricomposti al fine di riottenere l'immagine compressa finale. Una volta terminata l'esecuzione della DCT2, l'applicazione mostrerà sia l'immagine originale che l'immagine compressa in due finestre separate, permettendo all'utente di valutare visivamente l'effetto ottenuto dalla compressione. In particolare abbiamo sfruttato

la libreria **PIL** (Python Imaging Library) libreria di grande rilevanza nel campo dell'elaborazione delle immagini che consente una serie di soluzioni pragmatiche e potenti per la gestione delle immagini. La libreria consente il caricamento e il salvataggio di immagini in formati come jpeg, png, bmp e molti altri. Il vantaggio offerto dalla libreria PIL è la facilità con cui è possibile eseguire operazioni basilari sulle immagini

```
from PIL import Image, ImageOps
```

L'interfaccia è stata sviluppata utilizzando la libreria **tkinter**, che offre una gamma di widget predefiniti come finestre, bottoni, caselle di testo, etichette e altro, permettendo di creare e personalizzare l'aspetto dell'applicazione.

```
from tkinter import filedialog, messagebox
```

Metodo create_widgets(self):

Questo metodo è responsabile della creazione e configurazione dei widget dell'interfaccia utente.

```
def create_widgets(self):
    global label_path, entry_variable_f, entry_variable_d

    label_path = tk.Label(self.master, text="Nessun file
    ↪  selezionato", wraplength=300, font=self.custom_font)
    label_path.pack(pady=10)

    style = ttk.Style()
    style.configure("Material.TButton", font=self.custom_font)
    browse_button = ttk.Button(self.master, text="Sfoglia",
    ↪  command=browse_file, style="Material.TButton")
    browse_button.pack(pady=5)

    label_variable_f = tk.Label(self.master, text="Inserisci la
    ↪  variabile F:", font=self.custom_font)
    label_variable_f.pack(pady=10)

    entry_variable_f = tk.Entry(self.master,
    ↪  font=self.custom_font)
    entry_variable_f.pack(pady=5)

    label_variable_d = tk.Label(self.master, text="Inserisci la
    ↪  variabile d (intero):", font=self.custom_font)
    label_variable_d.pack(pady=10)

    entry_variable_d = tk.Entry(self.master,
    ↪  font=self.custom_font)
```

```
    entry_variable_d.pack(pady=5)

    compress_button = ttk.Button(self.master, text="Compress",
    ↪  command=flow, style="Material.TButton")
    compress_button.pack(pady=5)

    self.master.protocol("WM_DELETE_WINDOW", self.on_closing)

def on_closing(self):
    self.master.destroy()  # Chiudi la finestra principale
    self.master.quit()  # Interrompi il ciclo principale di
    ↪  Tkinter
```

configura l'interfaccia utente con etichette, campi di input e pulsanti necessari per selezionare un file, inserire i parametri di compressione e avviare il processo di compressione. Inoltre Il metodo on_closing(self) gestisce la chiusura della finestra principale, assicurandosi che il ciclo principale di tkinter venga interrotto correttamente.

Metodo flow():

Il metodo flow() gestisce il flusso principale dell'applicazione, dalla verifica delle variabili alla compressione dell'immagine e alla visualizzazione del risultato. Coordina il processo principale di compressione.

```
def flow():
F, d, image_path = check_variables()
blocks = divide_image_into_blocks(image_path, F)
blocks_dct_quantized = apply_dct2(blocks, F, d)

for i, block_dct_quantized in
↪  enumerate(blocks_dct_quantized[:3]):
    print(f"Risultati della DCT2 quantizzata per blocco {i +
    ↪  1}:\n{block_dct_quantized}\n")

blocks_idct_rounded = apply_idct2(blocks_dct_quantized)
save_compressed_image(blocks_idct_rounded)

# Visualizza le immagini originale e compressa
show_images(Image.open(file_path).convert('L'),
↪  Image.open("compressed_image.bmp"))
```

Metodo create first interface():

Il metodo create_first_interface() permette di creare l'interfaccia utente dove l'utente può selezionare un'immagine e i parametri di compressione.Come viene riportato in figura: 5.1

```
def create_first_interface():
global label_path, entry_variable_f, entry_variable_d

root = tk.Tk()
app = DCTApp(root)
root.mainloop()
```

Il risultato che si ottiene eseguendo questa funzione è mostrato in figura:

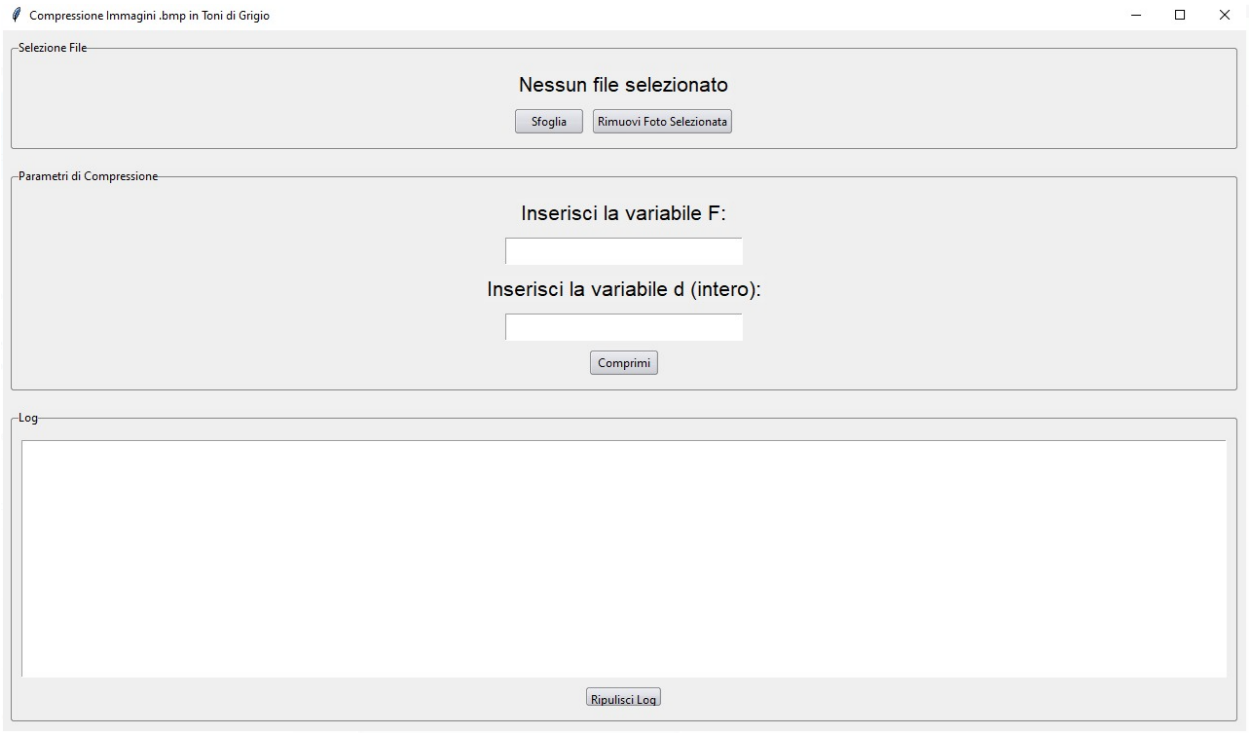

Figura 5.1: creazione prima GUI

All'interfaccia è stata aggiunta una label che indica la dimensione (in pixel) dell'immagine selezionata, poiché la scelta dei parametri F e d è strettamente legata alle misure dell'immagine stessa.

Metodo browse_file():

Il metodo browse_file() permette all'utente di selezionare un file immagine .bmp e aggiorna l'etichetta con il percorso e le dimensioni dell'immagine selezionata.

```
def browse_file():
global file_path
file_path = filedialog.askopenfilename(filetypes=[("Bitmap
 ↪  files", "*.bmp")])
if file_path:
    label_path.config(text="File selezionato: " + file_path)
```

```
        # Carichiamo l'immagine, la convertiamo in scala di grigi
        ↪ e calcoliamo le dimensioni
        img = Image.open(file_path).convert('L')
        img_width, img_height = img.size
        img.close()

        # Aggiorna la label con il testo che include il percorso
        ↪ dell'immagine e le dimensioni
        label_path.config(text=f"File selezionato:
        ↪ {file_path}\n\nDimensioni immagine:
        ↪ {img_width}x{img_height}")
    else:
        label_path.config(text="Nessun file selezionato")
```

L'utente può inserire i parametri F e d a sua scelta. È importante che l'utente tenga conto delle seguenti condizioni:

- La variabile F, che rappresenta la dimensione dei macro-blocchi in cui si effettuerà la DCT2, deve essere un numero intero e minore dell'altezza (o larghezza) dell'immagine selezionata.
- La variabile d, che rappresenta la soglia di taglio delle frequenze, deve essere un intero compreso tra 0 e 2F-2.

Naturalmente è stata implementata una funzione che verifica che tutte le condizioni richieste dalla traccia siano verificate prima di applicare la DCT2 e tutto questo viene svolto dalla seguente funzione:

Metodo per check_variables():

Il metodo check_variables() verifica che tutte le condizioni richieste siano rispettate prima di applicare la DCT2.

```
def check_variables():
    global entry_variable_f, entry_variable_d, file_path

    compressed_image_path = "compressed_image.bmp"

    try:
        # Verifica se esiste già un file "compressed_image.bmp"
        if os.path.exists(compressed_image_path):
            os.remove(compressed_image_path)  # Rimuovi il file
            ↪ esistente
            print("Rimossa l'immagine")
    except Exception as e:
        print("Errore durante la rimozione dell'immagine
        ↪ compressa:", str(e))
```

```
# Recupera i valori di F e d dalla label
F = entry_variable_f.get()
d = entry_variable_d.get()

# Verifica che le label e l'immagine non siano vuote. in caso
↪  contrario, viene mostrato un messaggio di errore
if not file_path:
    messagebox.showerror("Errore", "Seleziona un'immagine.")
    return

if not F:
    messagebox.showerror("Errore", "Inserisci il valore di F.")
    return

if not d:
    messagebox.showerror("Errore", "Inserisci il valore di d.")
    return

if not is_integer(F):
    messagebox.showerror("Errore", "F deve essere un numero
    ↪  intero")
    return

if not is_integer(d):
    messagebox.showerror("Errore", "d deve essere un numero
    ↪  intero")
    return

F = int(F)
d = int(d)

if F < 0:
    messagebox.showerror("Errore", "F deve essere maggiore o
    ↪  uguale a 0")
    return
if d < 0:
    messagebox.showerror("Errore", "d deve essere maggiore o
    ↪  uguale a 0")
    return

# Verifichiamo ora che il valore di F non sia superiore
↪  all'altezza/larghezza dell'immagine
try:
    with Image.open(file_path) as img:
        img_width, img_height = img.size
```

(a) Errore di F

(b) Errore di d

Figura 5.2: Possibili errori dei valori di compressione

```
        if F > img_width or F > img_height:
            messagebox.showerror("Errore", "Il valore di F non può
             ↪  superare le dimensioni dell'immagine.")
            return

        # Stampa debug delle dimensioni dell'immagine selezionata
        print("Image Width:", img_width)
        print("Image Height:", img_height)

    except Exception as e:
        messagebox.showerror("Errore", f"Errore durante il recupero
         ↪  delle dimensioni dell'immagine: {str(e)}")
        return

    # verifichiamo che il valore di d sia compreso tra 0 e 2F-2
    if d > 2 * F - 2 or d < 0:
        messagebox.showerror("Errore", "Il valore di d deve essere
         ↪  compreso tra 0 e 2F-2.")
        return

    # If all checks pass, proceed with compression
    print("All test are ok.")

    return F, d, file_path
```

Metodo divide_image_into_blocks(image_path, F):

Questo metodo ha il compito di suddividere l'immagine selezionata in blocchi quadrati di dimensione F x F, scartando eventuali pixel in eccesso dopo la conversione dell'immagine in scala di grigi.

```
def divide_image_into_blocks(image_path, F):
    try:
        with Image.open(image_path) as img:
            img_width, img_height = img.size

            # Converti l'immagine in scala di grigi
```

```
            img_gray = img.convert('L')

            # Calcoliamo il numero di blocchi in orizzontale e
            ↪  verticale
            num_blocks_horizontal = img_width // F
            num_blocks_vertical = img_height // F

            blocks = []

            # Iteriamo su tutti i blocchi
            for j in range(num_blocks_vertical):
                for i in range(num_blocks_horizontal):
                    # Calcoliamo le coordinate iniziali e finali
                    ↪  del blocco corrente
                    x0 = i * F
                    y0 = j * F
                    x1 = x0 + F
                    y1 = y0 + F

                    # Estraiamo il blocco corrente dall'immagine
                    block = img_gray.crop((x0, y0, x1, y1))
                    blocks.append(block)
            return blocks

    except Exception as e:
        print("Errore durante l'elaborazione dell'immagine:",
        ↪  str(e))
        return blocks
```

Inizialmente, la funzione apre l'immagine specificata attraverso il percorso image_path utilizzando la funzione Image.open(image_path) della libreria PIL (Python Imaging Library). Questo processo avviene all'interno di un blocco with, che garantisce la chiusura del file immagine una volta completate le operazioni. Dopo aver aperto l'immagine, vengono ottenute le dimensioni, ossia la larghezza e l'altezza, e l'immagine viene convertita in scala di grigi. Successivamente, viene calcolato il numero di blocchi orizzontali e verticali che possono essere estratti dall'immagine in base al parametro F. La funzione procede con un ciclo for nidificato che analizza tutti i possibili blocchi dell'immagine, determinati dal valore di F. È utile specificare che in ciascuna iterazione vengono calcolate le coordinate del blocco corrente, rappresentato dall'angolo superiore sinistro (x0, y0) e dall'angolo inferiore destro (x1, y1). Per estrarre il blocco corrente, viene utilizzato il metodo crop() della libreria PIL, che estrae la porzione definita dalle coordinate calcolate in precedenza. Ogni blocco estratto viene quindi aggiunto a una lista denominata blocks, che

viene restituita al termine dell'esecuzione della funzione.

Metodo apply_idct2():

Una volta raggiunta questa funzione, significa che la suddivisione dei blocchi è stata completata con successo e possiamo applicare la DCT2 tramite il seguente metodo.

```
def apply_idct2(blocks_dct_quantized):
# Lista per salvare i blocchi IDCT2 quantizzati e arrotondati
blocks_idct_rounded = []

# Iteriamo su tutti i blocchi quantizzati e applichiamo la
 ↪  IDCT2
for block_dct_quantized in blocks_dct_quantized:
    # Applichiamo la IDCT2 al blocco usando scipy
    block_idct = idct(idct(block_dct_quantized.T,
     ↪  norm='ortho').T, norm='ortho')

    # Arrotondiamo i valori della matrice IDCT2 al valore
     ↪  intero più vicino
    block_idct_rounded = np.round(block_idct)

    # Impostiamo a 0 i valori negativi
    block_idct_rounded[block_idct_rounded < 0] = 0

    # Impostiamo a 255 i valori maggiori di 255
    block_idct_rounded[block_idct_rounded > 255] = 255

    # Convertiamo la matrice risultante in un array di interi
     ↪  non segnati a 1 byte
    block_idct_rounded = block_idct_rounded.astype(np.uint8)

    # Aggiungiamo il blocco IDCT2 arrotondato e quantizzato
     ↪  alla lista
    blocks_idct_rounded.append(block_idct_rounded)

return blocks_idct_rounded
```

La funzione inizia creando una lista vuota chiamata blocks_dct_quantized, che servirà a contenere i blocchi d'immagine trasformati con DCT2 e successivamente quantizzati. Successivamente, si entra nel ciclo for che scorre ogni blocco presente nella lista blocks. In ciascuna iterazione, il blocco viene convertito in un array NumPy, facilitando così le operazioni matematiche sui dati dell'immagine. All'interno del ciclo, si applica la DCT2 al blocco, eseguendo due trasformazioni: prima lungo le colonne (dopo aver trasposto il blocco) e

poi lungo le righe. È importante sottolineare che l'argomento norm='ortho' garantisce l'uso di fattori di normalizzazione per mantenere l'ortogonalità, caratteristici della DCT.
Dopo aver calcolato la DCT2, si procede con la quantizzazione, moltiplicando l'array DCT2 per una maschera binaria definita dal parametro d. Questa maschera, creata con $np.abs(np.add.outer(range(F), range(F))) < d$ identifica gli elementi in cui la somma degli indici è inferiore a d, contribuendo così a ridurre le frequenze più alte e a comprimere i dati. Il blocco risultante dalla DCT2 e dalla quantizzazione viene aggiunto alla lista blocks_dct_quantized, che sarà restituita dalla funzione.
La funzione apply_idct2(), all'interno di flow(), ha il compito di ricostruire l'immagine compressa a partire dai blocchi DCT2 quantizzati. Per ogni blocco, viene applicata la IDCT2 tramite la libreria Scipy, trasformando ciascun blocco DCT2 in valori di pixel. I risultati della IDCT2 vengono arrotondati al numero intero più vicino utilizzando np.round(), un passaggio fondamentale per assicurare che le intensità dei pixel siano rappresentate come interi. Infine, i valori dei pixel vengono corretti per rimanere nel range appropriato: i valori negativi vengono fissati a 0 e quelli superiori a 255 vengono limitati a 255, mantenendo così l'intervallo consentito per le intensità.
La matrice risultante viene trasformata in un array di interi non segnati a 1 byte con astype, questo è un passaggio fondamentale per garantire una corretta rappresentazione dei valori dei pixel. Il blocco ricostruito, arrotondato e quantizzato, viene quindi aggiunto alla lista blocks_idct_rounded, che raccoglie tutti i blocchi dell'immagine ricostruita. Al termine del processo per tutti i blocchi, la funzione restituisce questa lista, che rappresenta l'immagine compressa originale.

Metodo save_compressed_image():

Una volta completati i passaggi descritti, è possibile ricostruire l'immagine compressa utilizzando la funzione save_compressed_image(), che, impiegando i blocchi IDCT2 arrotondati e quantizzati, consente di ricreare e salvare l'immagine in formato .bmp. Di seguito è fornito il codice di questo metodo.

```
def save_compressed_image(blocks_idct_rounded):
compressed_image_path = "compressed_image.bmp"
compressed_image = None

# Verifica se esiste già un file "compressed_image.bmp"
if os.path.exists(compressed_image_path):
    os.remove(compressed_image_path)  # Rimuovi il file
    ↪ esistente
    print("Rimossa l'immagine")

try:
```

```
        # Ricomponi l'immagine compressa utilizzando i blocchi
        ↪ compressi
        img_width, img_height = Image.open(file_path).size
        compressed_image = Image.new('L', (img_width, img_height))

        F = entry_variable_f.get()
        num_blocks_horizontal = img_width // int(F)

        for j in range(img_height // int(F)):
            for i in range(num_blocks_horizontal):
                x0 = i * int(F)
                y0 = j * int(F)

                block = blocks_idct_rounded.pop(0)

                compressed_image.paste(Image.fromarray(block), (x0,
                ↪ y0))

        # Salva l'immagine compressa nel formato .bmp
        compressed_image.save(compressed_image_path)
        compressed_image.close()

        print("Immagine compressa salvata con successo.")
    except Exception as e:
        print("Errore durante il salvataggio dell'immagine
        ↪ compressa:", str(e))
```

terminata l'esecuzione della funzione, l'immagine compressa viene salvata per essere successivamente visualizzata a schermo tramite l'utilizzo di una nuova finestra in cui sono mostrate contemporaneamente l'immagine originale e quella compressa.

5.4 Classe main:

Questa classe serve per far partire il programma e creare l'interfaccia utente.

5.5 Risulati Compressione JPEG:

Quello che osserviamo ora sono i possibili risultati ottenibili scegliendo diversi valori di F e d a partire da un'immagine .bmp. Come accennato nella sezione "Repository Codice" della parte due, le immagini sono disponibili nel repository indicato.

5.5.1 Casi Limite:

Per casi limite si intendono le compressioni di immagini in cui i valori di F e d sono accettabili al momento della compressione, ma non lo sono più se modificati oltre un certo limite. Di seguito, presentiamo:

1. La compressione di un'immagine con il valore di F pari alla dimensione dell'immagine (se non è quadrata, si considera il lato minore) e il valore di d esattamente pari a $2F - 2$.

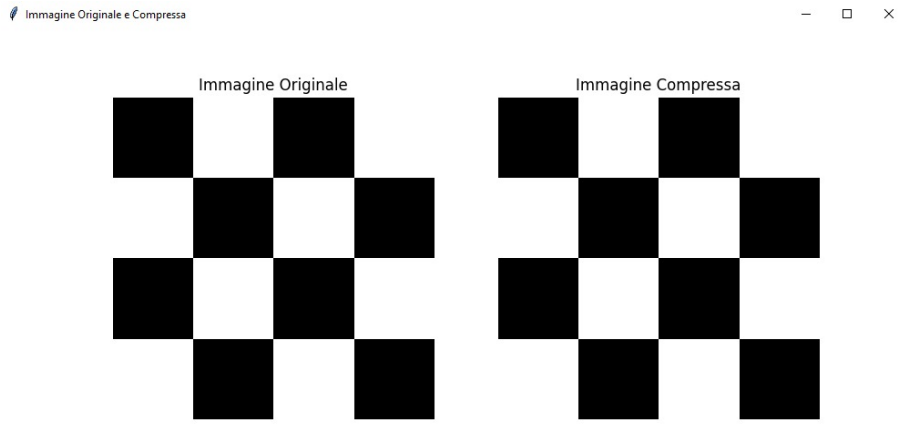

Figura 5.3: Caso Limite 1

da questa immagine abbiamo che i valori di F e d sono pari a

- F=40
- d=78

Da questa operazione notiamo che non ci sono evidenti differenze tra l'immagine inizialmente fornita e quella finale dato che la maggior parte delle frequenze, ottenute tramite DCT2, non vengono eliminate.

2. La compressione di un'immagine con il valore di F pari al massimo possibile per l'immagine, ma questa volta con il valore di d uguale a 0.

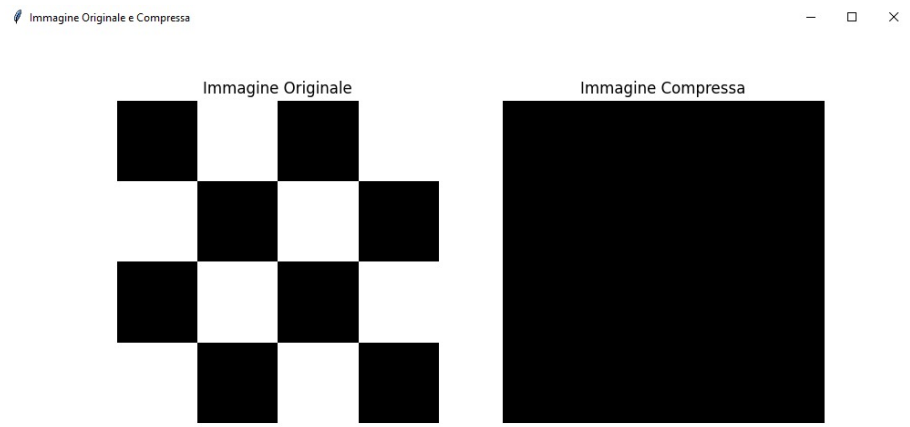

Figura 5.4: Caso Limite 2

da questa immagine abbiamo che i valori di F e d sono pari a

- F=40
- d=0

In questo caso avendo assegnato al valore d 0; allora la DCT2 assegnerà a tutte le frequenze dell'immagine originale il valore zero ottenendo un immagine completamente nera come risultato. Inoltre notiamo come, riducendo il valore di F nell'intervallo dei suoi valori accettabili e settando la variabile d sempre al valore zero, l'immagine ottenuta dalla compressione risulterà sempre totalmente nera.

5.5.2 Uteriori Casi:

A questo punto proviao a settare i valori di F e d e vedaimo come si compaorta la compressione: per questo test usimo l'immagine di una chiesa che ha una dimensione di 2000x3008 pixel. Quello che otteniamo è il seguente risulato:

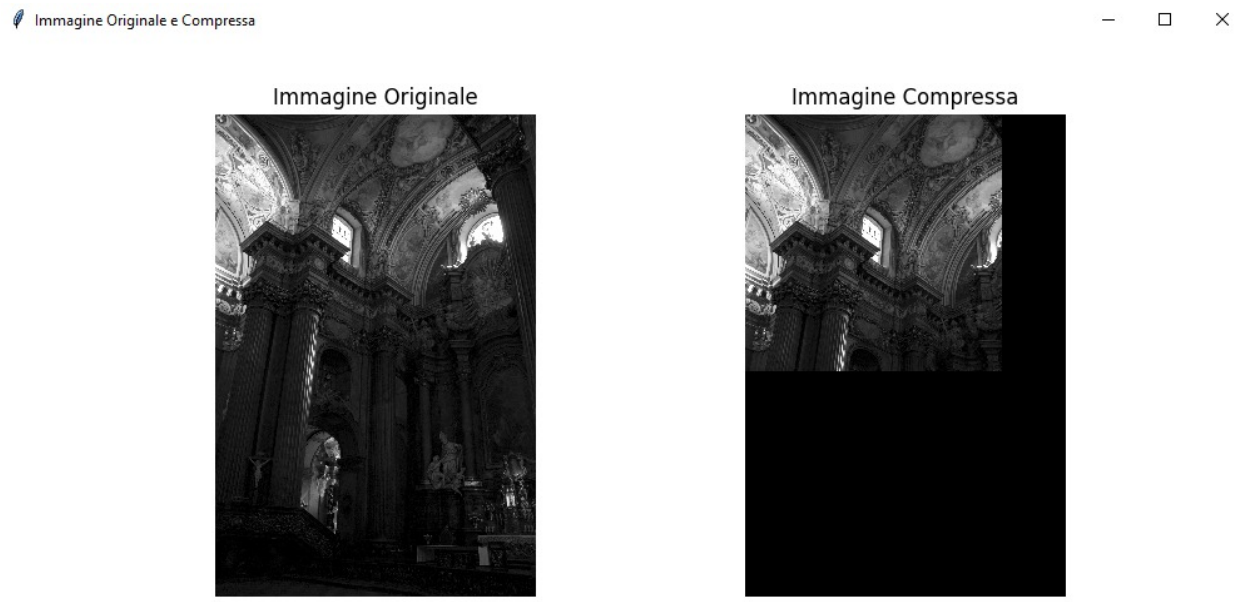

Figura 5.5: chiesa1

In questo esempio abbiamo dato ai due parametri i seguento valori:

- F=1600
- d=3198

Da questo esperimento si osserva una rilevante presenza di alcune aree completamente nere e due blocchi singoli di pixel compressi. Questo fenomeno è dovuto al fatto che il valore di F assegnato per la compressione non è un divisore perfetto della dimensione, sia orizzontale che verticale, dell'immagine originale. Di conseguenza, il programma non è in grado di creare blocchi di dimensione F x F che coprano interamente i pixel dell'immagine.

Se invece prendiamo un'immagine e assegnamo a F circa la metà della dimensione orizzontale, mentre a d assegniamo un valore relativamente basso, il risultato che otteniamo è il seguente:

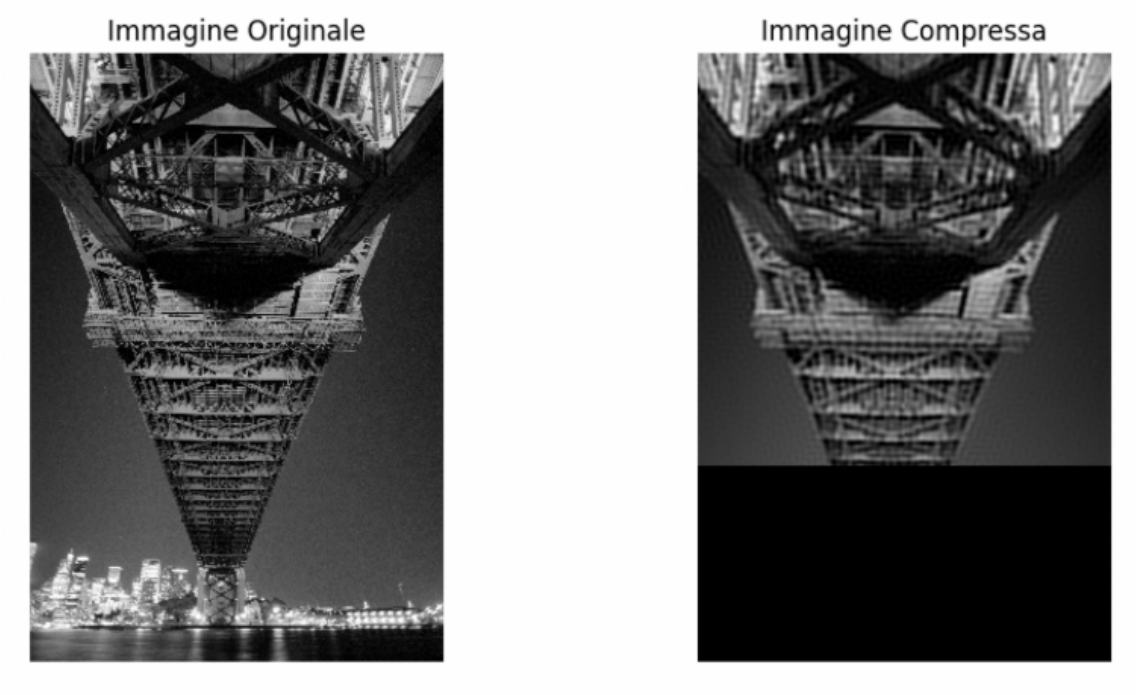

Figura 5.6: Caption

In questo esempio abbiamo dato ai due parametri i seguento valori:

- F=1372
- d=65

L'immagine ottenuta risulta visibilmente degradata, ma non in modo eccessivo, poiché la porzione rimanente dell'immagine presa in considerazione presenta un minor numero di discontinuità. Questo porta, di conseguenza, a un deterioramento meno marcato rispetto all'immagine originale.

Un ultimo caso che esaminiamo è quando assegniamo a F il valore massimo possibile, mentre a d attribuiamo un valore che, pur non essendo zero, è molto vicino ad esso.

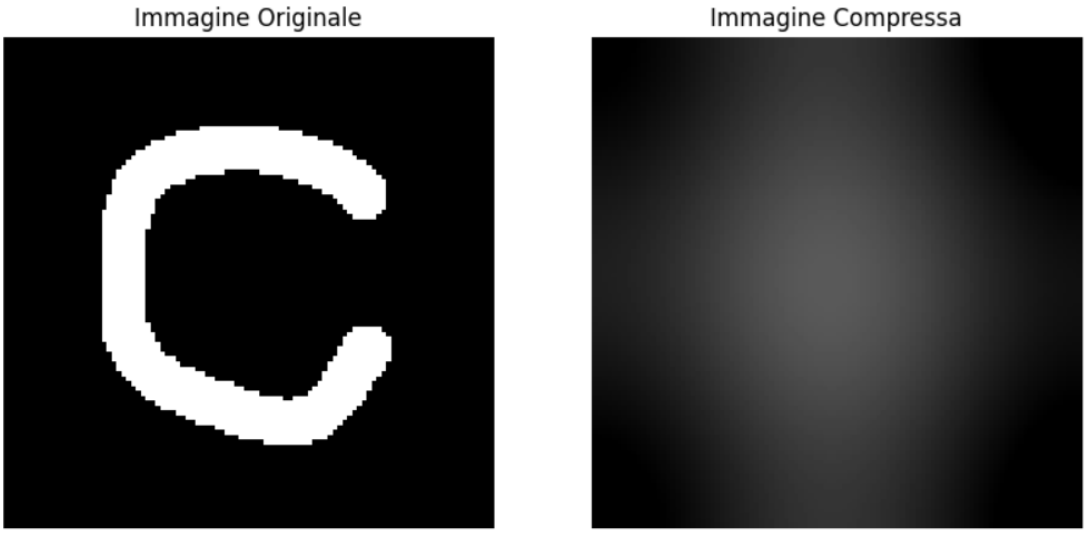

Figura 5.7: Caption

In questo esempio abbiamo dato ai due parametri i seguento valori:

- F=100
- d=3

L'immagine ottenuta non è completamente nera, come possiamo vedere nella figura 5.4, dove tutti i pixel risultavano essere di colore nero. In questo caso, le frequenze associate ai pixel dell'immagine, avendo un valore del parametro d leggermente superiore a zero, non vengono completamente tagliate, rappresentando così il colore bianco.

5.6 Problemi Riscontrati:

In questa seconda fase ci siamo scontrati con alcuni problemi che, nonostante tutto, siamo riusciti a risolvere in maniera efficiente.

- Gestione delle Dimensioni dei Blocchi:
  Il problema consisteva nell'assegnare correttamente la dimensione dei blocchi (parametro F) in modo che si adattassero perfettamente all'immagine. Abbiamo risolto questo problema verificando le dimensioni dell'immagine rispetto al parametro F per assicurarci che i blocchi si adattassero correttamente:

```
    if F > img_width or F > img_height:
messagebox.showerror("Errore", "Il valore di F non può
 ↪  superare le dimensioni dell'immagine.")
return None, None, None
```

- Arrotondamento e Clipping dei Pixel:
  Gli artefatti visivi o la perdita di dettaglio nell'immagine erano causati

dall'arrotondamento e dal clipping dei pixel. Abbiamo risolto questo problema arrotondando i valori dei pixel e limitandoli all'intervallo 0-255:

```
block_idct_rounded = np.round(block_idct)
block_idct_rounded[block_idct_rounded < 0] = 0
block_idct_rounded[block_idct_rounded > 255] = 255
block_idct_rounded = block_idct_rounded.astype(np.uint8)
```

- Ricostruzione dell'Immagine, Compatibilità dei Formati di Immagine, ... Questi sono solo alcuni dei problemi che abbiamo dovuto affrontare, ma che siamo riusciti a risolvere in maniera efficiente. Abbiamo implementato controlli accurati per la ricomposizione dei blocchi trasformati e abbiamo utilizzato librerie consolidate per garantire la compatibilità con vari formati di immagine.

5.7 Conclusione parte 2:

In questo progetto, abbiamo esplorato l'implementazione della Trasformata Coseno Discreta bidimensionale (DCT2) in un ambiente open source, analizzando l'efficacia di un algoritmo di compressione simile a JPEG applicato a immagini in scala di grigi. I risultati hanno mostrato come la scelta dei parametri di compressione influisce significativamente sulla qualità visiva delle immagini risultanti.

Parte III

Conclusioni Finali

Capitolo 6

Conclusione

Questo progetto ha permesso di approfondire la comprensione delle trasformate discrete e dei loro utilizzi pratici nella compressione delle immagini. I risultati ottenuti confermano la validità della DCT2 come strumento potente per la compressione delle immagini, evidenziando anche l'importanza della scelta accurata dei parametri per ottimizzare il bilancio tra compressione e qualità visiva.
Tutti gli obiettivi richiesti sono stati raggiunti, e questo progetto ci ha permesso di comprendere a fondo l'importanza degli algoritmi di ottimizzazione non solo per la compressione delle immagini, ma anche per la computazione di matrici di grandi dimensioni nell'ambito informatico. Ogni problema che abbiamo incontrato è stato affrontato con successo, ottenendo sempre una soluzione ottimale che soddisfacesse le specifiche del progetto.