UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

DIPARTIMENTO DI INFORMATICA

CORSO DI STUDIO IN
INFORMATICA

Utilizzo della DCT per compressione di immagini

Membro:
Lorenzo Erba
Matricola: 933012

Anno accademico:
2024/2025

Indice

1 Introduzione

1.1 Scopo del progetto

Lo scopo di questo progetto prevede di utilizzare l'implementazione della *DCT2* in un ambiente open source e di studiare gli effetti di un algoritmo di compressione tipo *JPEG* sulle immagini in toni di grigio.

1.2 Componenti del gruppo

- Lorenzo Erba, matricola: 933012

1.3 IDE e linguaggio di programmazione

I programmi richiesti sono stati sviluppati con il linguaggio Python, utilizzando come ambiente di sviluppo Visual Studio Code. Le motivazioni a sostegno di queste scelte risiedono in:

- **semplicità**. Python è un linguaggio che si contraddistingue per la sua sintassi intuitiva e un paradigma di programmazione adatto per la prototipazione rapida. Analogamente VS Code offre un'interfaccia utente e un ambiente di sviluppo intuitivo garantendo rapidità nella scrittura del codice, grazie anche ai numerosi plug-in che possono essere utilizzati.
- **open-source**. Sia Python che VS Code sono due artefatti software di utilizzo pubblico e gratuito, offrendo accesso a numerose librerie altamente testate e alla possibilità di effettuare troubleshooting in modo rapido ed efficiente. Per la realizzazione di questo progetto si è deciso di utilizzare NumPy, per effettuare operazioni aritmetiche, e SciPy, per l'utilizzo della DCT2 FFT.
- **conoscenza pregressa**. Python è un linguaggio che è stato ampiamente esplorato e utilizzato in numerosi ambiti, soprattutto laddove era richiesta una prototipazione rapida ed efficiente, come la realizzazione di modelli di Machine Learning.

2 Parte I - Confronto DCT

2.1 Specifiche del progetto

La prima parte del progetto prevede di fornire due implementazioni della *Discrete Cosine Transform* nella versione bidimensionale (DCT2) e di confrontarne i tempi d'esecuzione.
Più precisamente, la prima versione richiede di fornire una propria implementazione della DCT2, mentre la seconda versione prevede di eseguire la DCT2 in versione *FFT*, supportata dalla libreria scelta.
È richiesto di procurarsi array quadrati $N \times N$ con N crescente e rappresentare su un grafico, in scala semilogaritmica (solo sull'asse delle ordinate), il tempo impiegato per eseguire entrambe le versioni della DCT2 al variare di N.
I tempi dovrebbero essere proporzionali a N^3 per la DCT2 implementata manualmente, e a N^2 per la versione fast (più precisamente a $N^2 \log(N)$). I tempi ottenuti con la versione fast potrebbero mostrare un andamento irregolare, dovuto al tipo di algoritmo utilizzato.

2.2 DCT

La Trasformata Discreta del Coseno, nota come *DCT* (Discrete Cosine Transform), è una tecnica fondamentale in elaborazione dei segnali, in particolare nella compressione di immagini (ad esempio JPEG), audio (come MP3) e video (come MPEG).

L'obiettivo principale della DCT è quello di trasformare un segnale discreto in una somma di funzioni coseno a diverse frequenze. Questo consente di rappresentare l'informazione originale in termini di frequenze, permettendo una maggiore efficienza nella compressione:

- Le **frequenze basse** rappresentano variazioni lente, tipiche delle aree uniformi
- Le **frequenze alte** rappresentano variazioni rapide, come bordi e dettagli fini

Nella maggior parte dei segnali reali (soprattutto immagini), gran parte dell'energia è concentrata nelle frequenze più basse, per cui le frequenze alte possono essere scartate con perdite minime di qualità. Questa proprietà è alla base della compressione con perdita (*lossy*).

Sia $f : [0, 1] \to \mathbb{R}$ una funzione reale e continua, che intendiamo rappresentare come combinazione di funzioni armoniche. Supponiamo di campionare f in

N punti equidistanti all'interno dell'intervallo $[0, 1]$, precisamente nei punti medi:

$$x_j = \frac{2j+1}{2N}, \quad \text{per } j = 0, 1, \ldots, N-1$$

Otteniamo così un vettore di campionamenti:

$$\vec{f} = \left[f(x_0), f(x_1), \ldots, f(x_{N-1}) \right]$$

L'idea della **Discrete Cosine Transform** (DCT) è quella di rappresentare il vettore $\vec{f}$ come combinazione lineare di funzioni coseno di frequenza crescente. In particolare, il vettore $\vec{c}$ di frequenze ottenuto dall'esecuzione della DCT viene calcolato come:

$$\vec{c} = D\vec{f} \tag{2.1}$$

dove:

- D corrisponde alla matrice di trasformazione, con $D_{l,j} = \alpha_l \cos(l\pi \cdot \frac{2j+1}{2N})$
- $\tilde{\mathbf{f}}$ è il vettore dei campionamenti.
- $\alpha_{\mathbf{k}} = \begin{cases} \frac{1}{\sqrt{N}} & \text{se } k = 0 \\ \sqrt{\frac{2}{N}} & \text{se } k \geq 1 \end{cases}$ è il fattore di normalizzazione.

La DCT ammette anche l'operazione inversa, chiamata IDCT, che, partendo dal vettore delle frequenze, consente di ricostruire la funzione originale. Tuttavia, la funzione ricostruita presenta alcune differenze rispetto a quella originale, dovute alle approssimazioni introdotte nel calcolo delle frequenze tramite la DCT.

Infatti, la DCT concentra gran parte dell'energia del segnale nelle frequenze basse, riducendo la necessità di memorizzare le frequenze più alte, costituendo il principio fondamentale alla base della compressione.

Per garantire una ricostruzione fedele del segnale originale, senza perdita di informazione, è necessario memorizzare tutte le frequenze calcolate dalla DCT, poiché ciascuna di esse contribuisce alla qualità del segnale ricostruito, anche se le componenti con valori più elevati contengono la maggior parte delle informazioni intrinseche.

2.3 DCT2

La DCT nella sua versione unidimensionale, sebbene sia matematicamente valida e utile per segnali mono-dimensionali, risulta poco pratica nelle appli-

cazioni reali. Questo perché, nella maggior parte dei casi, i dati di ingresso non sono vettori ma matrici di campionamento, come ad esempio le immagini digitali, rappresentate da matrici bidimensionali di valori di intensità.

Nel caso di segnali bidimensionali di dimensioni $M \times N$, come le immagini, si utilizza la DCT bidimensionale o DCT2, calcolata applicando la DCT unidimensionale prima sulle righe e poi sulle colonne (o viceversa):

$$c_{k,l} = \underbrace{\alpha_k \sum_{i=0}^{N-1} \underbrace{\alpha_l \sum_{j=0}^{M-1} f_{i,j} \cos\left(\frac{(2j+1)l\pi}{2M}\right)}_{\text{DCT per righe}} \cos\left(\frac{(2i+1)k\pi}{2N}\right)}_{\text{DCT per colonne}}$$

e

$$\alpha_{k,l} = \alpha_k \alpha_l = \begin{cases} \sqrt{\frac{1}{N}} & \text{se } k = 0 \\ \sqrt{\frac{2}{N}} & \text{se } k > 0 \end{cases} \cdot \begin{cases} \sqrt{\frac{1}{M}} & \text{se } l = 0 \\ \sqrt{\frac{2}{M}} & \text{se } l > 0 \end{cases} \tag{2.2}$$

Analogamente al caso unidimensionale, anche nel contesto bidimensionale è possibile applicare l'operazione inversa, nota come IDCT2, che permette di ricostruire la matrice originale, a partire dai coefficienti della trasformata.

3 Implementazione

3.1 Repository della libreria

Lo sviluppo del programma è avvenuto tramite l'utilizzo della piattaforma di versioning di GitHub ed è disponibile al seguente link.

3.2 Struttura del programma

L'applicativo software richiesto nella prima parte è stato sviluppato in Python tramite il supporto delle librerie NumPy, per effettuare operazioni di calcolo numerico, e Scipy per l'utilizzo della DCT2 in versione *FFT*. Il programma si compone di tre classi principali:

- **Integrity_Test.py**. Si tratta di un modulo utilizzato per effettuare test di correttezza sulla DCT2 fornita da Scipy e la versione implementata nel nostro codice. Più precisamente, data una matrice di quantizzazione nota e il risultato atteso della sua applicazione alla DCT2, si eseguono entrambe le versioni ed effettuare un cross-check tra il risultato ottenuto e quello atteso.
- **Utils.py**. Il modulo *Utils.py* rappresenta il *core* dell'applicativo software. Al suo interno sono presenti i metodi che implementano la DCT2 personalizzata, richiamo alla DCT2 di Scipy e la funzione per generare i grafici di confronto.
- **Main.py**. Si tratta del modulo che contiene la versione del codice eseguibile e che prevede di generare array quadrati di dimensione $N \times N$, con N crescente, eseguire entrambe le versioni della DCT2 e confrontare i tempi di esecuzione. Per fornire un approccio di simulazione più realistico, le entrate della matrice di posseggono valori $\in [0, 255]$, simulando i valori della scala di grigi delle immagini.

3.3 Integrity_Test.py

Il modulo *Integrity_Test.py* implementa il metodo **test_mydct2** e **test_mydct1**, utilizzati per validare l'implementazione della DCT unidimensionale e bidimensionale.

```
def test_mydct2():
    example_sampled_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]
        ])

    # Verifica DCT2
    dct2_result = my_dct2(example_sampled_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]
        ])

    assert np.allclose(expected_dct2_result,dct2_result,
    ↪  atol=1e-2,rtol=1e-2)

def test_mydct1():
    example_sampled_vector = np.array([231, 32 ,233 ,161 ,24
    ↪  ,71 ,140, 245])
    expected_dct1_result = np.array([4.01e+02, 6.60e+00
    ↪  ,1.09e+02 ,-1.12e+02, 6.54e+01, 1.21e+02, 1.16e+02
    ↪  ,2.88e+01])
    my_dct1_result = my_dct1(example_sampled_vector)
    assert np.allclose(expected_dct1_result,my_dct1_result,
    ↪  atol=1e-2,rtol=1e-2)
```

Più precisamente il primo metodo definisce la seguente matrice:

$$D = \begin{bmatrix} 231 & 32 & 233 & 161 & 24 & 71 & 140 & 245 \\ 247 & 40 & 248 & 245 & 124 & 204 & 36 & 107 \\ 234 & 202 & 245 & 167 & 9 & 217 & 239 & 173 \\ 193 & 190 & 100 & 167 & 43 & 180 & 8 & 70 \\ 11 & 24 & 210 & 177 & 81 & 243 & 8 & 112 \\ 97 & 195 & 203 & 47 & 125 & 114 & 165 & 181 \\ 193 & 70 & 174 & 167 & 41 & 30 & 127 & 245 \\ 87 & 149 & 57 & 192 & 65 & 129 & 178 & 228 \end{bmatrix}$$

utilizzata come input alla DCT2, e definisce

$$\begin{bmatrix} 1.11e{+}3 & 4.40e{+}1 & 7.59e{+}1 & -1.38e{+}2 & 3.50e{+}0 & 1.22e{+}2 & 1.95e{+}2 & -1.01e{+}2 \\ 7.71e{+}1 & 1.14e{+}2 & -2.18e{+}1 & 4.13e{+}1 & 8.77e{+}0 & 9.90e{+}1 & 1.38e{+}2 & 1.09e{+}1 \\ 4.48e{+}1 & -6.27e{+}1 & 1.11e{+}2 & -7.63e{+}1 & 1.24e{+}2 & 9.55e{+}1 & -3.98e{+}1 & 5.85e{+}1 \\ -6.99e{+}1 & -4.02e{+}1 & -2.34e{+}1 & -7.67e{+}1 & 2.66e{+}1 & -3.68e{+}1 & 6.61e{+}1 & 1.25e{+}2 \\ -1.09e{+}2 & -4.33e{+}1 & -5.55e{+}1 & 8.17e{+}0 & 3.02e{+}1 & -2.86e{+}1 & 2.44e{+}0 & -9.41e{+}1 \\ -5.38e{+}0 & 5.66e{+}1 & 1.73e{+}2 & -3.54e{+}1 & 3.23e{+}1 & 3.34e{+}1 & -5.81e{+}1 & 1.90e{+}1 \\ 7.88e{+}1 & -6.45e{+}1 & 1.18e{+}2 & -1.50e{+}1 & -1.37e{+}2 & -3.06e{+}1 & -1.05e{+}2 & 3.98e{+}1 \\ 1.97e{+}1 & -7.81e{+}1 & 9.72e{+}{-}1{+} & -7.23e{+}1 & -2.15e{+}1 & 8.13e{+}1 & 6.37e{+}1 & 5.90e{+}0 \end{bmatrix}$$

come il risultato atteso dall'esecuzione di DCT2(D).
Successivamente viene eseguita la nostra versione della DCT2 e quella fornita da Scipy confrontando i risultati ottenuti con quello atteso. Se i risultati sono uguali, a meno di una tolleranza, allora il test viene considerato valido.

Il secondo metodo invece ha il compito di verificare la correttezza del metodo `my_dct1`, implementato nel modulo `Utils.py`.
Più precisamente, viene definito un vettore $\vec{e}$ di 8 elementi:

$$\vec{e} = \begin{bmatrix} 231 & 32 & 233 & 161 & 24 & 71 & 140 & 245 \end{bmatrix}$$

che coincide con la prima riga del blocco 8×8 definito precedentemente.
Successivamente viene definito il risultato atteso dall'esecuzione di DCT($\vec{e}$):

$$\begin{bmatrix} 4.01e{+}02, 6.60e{+}00, 1.09e{+}02, -1.12e{+}02, 6.54e{+}01, 1.21e{+}02, 1.16ee{+}02, 2.88e{+}01 \end{bmatrix}$$

Il metodo procede quindi a invocare la funzione `my_dct1` e confronta il risultato ottenuto con quello atteso. Se i risultati sono uguali, a meno di una tolleranza, allora il test viene considerato valido.

3.4 Utils.py

Il modulo *Utils.py* definisce i metodi principali e necessari a eseguire la DCT2 e a generare il grafico dei tempi di esecuzione.
Più precisamente implementa i seguenti metodi:

- **dct2_from_lib**

- **get_transformation_matrix**
- **my_dct1**
- **my_dct2**
- **generate_plot.**

3.4.1 Dct2_from_lib

Il metodo **dct2_from_lib** esegue al suo interno la DCT2 in versione FFT fornita dalla libreria Scipy.
Nello specifico il metodo:

```
def dct2_from_lib(f):
    return dctn(f,type=2,norm="ortho")
```

riceve in input la matrice contenente i valori campionati e restituisce il risultato della chiamata al metodo `dctn` che riceve in input tre parametri:

- **f**: la matrice dei campionamenti
- **type=2**: indica l'utilizzo della DCT 2D
- **norm="ortho"**: indica di utilizzare la normalizzazione ortonormale.

Di fatti, come indicato nella documentazione, il parametro `norm` supporta diversi valori, a seconda della normalizzazione desiderata. Nel caso venga assegnato il valore *ortho*, Scipy utilizza come fattori $\alpha_{k,l}$ gli stessi valori indicati nell'equazione (2.2)

3.4.2 Get_transformation_matrix

Il metodo **get_transformation_matrix** viene richiamato all'interno della funzione `my_dct1`, prendendo in input la dimensione `n` della matrice di trasformazione D, calcolata come indicato nell'equazione (2.1).

```
def get_transformation_matrix(n):
alpha = np.zeros(n)

alpha[0] = 1.0 / np.sqrt(n)
alpha[1:] = np.sqrt(2.0/n)

D = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        D[i, j] = alpha[i] * np.cos((i * math.pi * (2 * j +
        ↪  1)) / (2 * n))
return D
```

3.4.3 My_dct1

Il metodo **my_dct1** fornisce un'implementazione della DCT unidimensionale, utilizzata per poter implementare la DCT bidimensionale, calcolata prima per colonne e poi per righe. Più precisamente il metodo prende in input l'array `f` dei valori campionati, calcola la matrice di trasformazione `D` associata e restituisce il risultato del prodotto $D\vec{f}$ come indicato nell'equazione (2.1).

```
def my_dct1(sampled_vector):
    vector_dim = len(sampled_vector)
    D = get_transformation_matrix(vector_dim)
    return np.dot(D,sampled_vector)
```

3.4.4 My_dct2

Il metodo **my_dct2** fornisce un'implementazione della DCT bidimensionale, calcolata come l'esecuzione della DCT unidimensionale prima per colonne e poi per righe.
Nel dettaglio, data la matrice dei campionamenti `sampled_matrix`, viene eseguita prima la DCT unidimensionale per colonne, quindi lungo l'asse verticale, e poi per righe, lungo l'asse orizzontale.

```
def my_dct2(sampled_matrix):
    n,m = sampled_matrix.shape

    result = np.copy(sampled_matrix.astype('float64'))
    # DCT per ogni colonna
    for j in range(m):
        result[:, j] = my_dct1(result[:, j])

    # DCT per ogni riga
    for i in range(n):
        result[i, :] = my_dct1(result[i, :])

    return result
```

3.4.5 Generate_plot

L'ultimo metodo implementato in *Utils.py* è la funzione **generate_plot**, che ha lo scopo di generare il grafico di comparazione dei tempi di esecuzione di my_dct2 e `dct2_from_lib` applicate a matrici $N \times N$ con N crescente.
Nel dettaglio vengono generate quattro curve:

- una curva teorica rappresentante il tempo d'esecuzione della DCT2 FFT, proporzionale a $N^2 \log(N)$

- una curva teorica rappresentante il tempo d'esecuzione della DCT2 nella versione originale, non fast, proporzionale a N^3
- la curva del tempo d'esecuzione della funzione `my_dct2` che si presuppone simile alla curva N^3.
- la curva del tempo d'esecuzione della funzione `dct2_from_lib` che si presuppone simile alla curva $N^2 \log(N)$

Tutte le curve sono rappresentate in scala semilogaritmica, solo le ordinate, riportando il tempo di esecuzione in secondi sull'asse delle ordinate, e la dimensione N della matrice sull'asse delle ascisse.

```
def generate_plot(my_dct2_time,dct2_lib_time, N_list):
    plt.figure(figsize=(10,6))

    n3 = [n**3/ 1e5 for n in N_list]
    n2logn = [n**2 * np.log(n) / 1e8 for n in N_list]

    plt.semilogy(N_list,my_dct2_time,label="My DCT2",
    ↪  color="blue")

    plt.semilogy(N_list,dct2_lib_time,label="DCT2 from
    ↪  lib", color="red")

    plt.semilogy(N_list,n3,label="Theorical n3",
    ↪  color="blue",linestyle="dashed")

    plt.semilogy(N_list,n2logn,label="Theorical n2log(n)",
    ↪  color="red", linestyle="dashed")
    plt.xlabel('N')
    plt.ylabel('Tempo di esecuzione in secondi')
    plt.title('Confronto tempi di esecuzione DCT2 homemade
    ↪  e DCT2 libreria')
    plt.legend()
    plt.grid(True)
    plt.savefig("Plot_Di_Confronto.png")
    plt.show()
```

3.5 Main.py

All'interno del modulo *Main.py* vengono generate delle matrici $N \times N$ con $0 \leq N \leq 1000$, con un tasso d'incremento di N pari a 50. Ogni matrice viene popolata con un valore rappresentante il colore di un pixel in scala di grigi, e pertanto scelto casualmente dall'intervallo $[0.0, 255.0]$.
Successivamente viene computata la DCT2, in entrambe le versioni, sulla

matrice generata precedentemente e calcolati i rispettivi tempi d'esecuzione. Infine viene generato un grafico che mette a confronto i tempi di esecuzione dell'implementazione propria della DCT2 e la versione fornita da Scipy.

```
    def main():
        N_list = list(range(50,1001,50))
        my_dct2_time = []
        dct2_from_lib_time = []

        for n in N_list:
            print("Dimension: ",n)
            np.random.seed(10)
            samnpled_matrix = np.random.uniform(0.0, 255.0,
            ↪  size=(n,n))

            my_dct2_time.append(timeit.timeit(lambda:
            ↪  my_dct2(samnpled_matrix),number = 1))

            dct2_from_lib_time.append(timeit.timeit(lambda:
            ↪  dct2_from_lib(samnpled_matrix),number = 1))

    generate_plot(my_dct2_time,dct2_from_lib_time,N_list)

if(__name__ == "__main__"):
    main()
```

4 Analisi dei risultati

Prima di eseguire il modulo *Main.py* è stato eseguito *Integrity_test.py* che ha fornito un riscontro positivo sul test effettuato, garantendo correttezza e coerenza per entrambe le implementazioni della DCT2.

Successivamente è stato eseguito il metodo `main` all'interno del modulo *Main.py* che ha generato il seguente grafico:

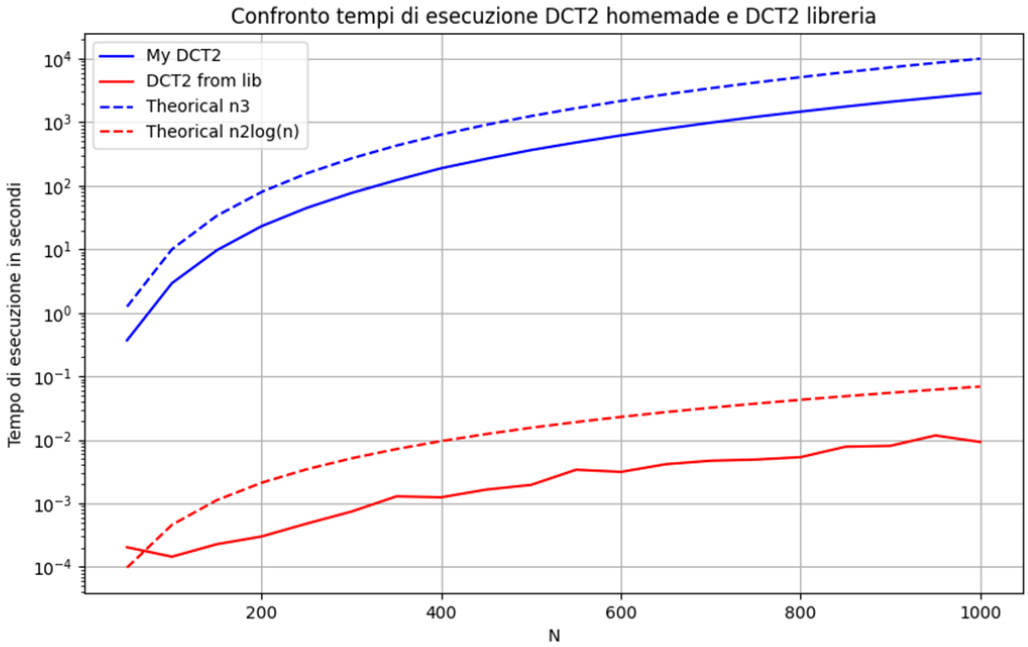

Figura 4.1: Tempi d'esecuzione della DCT2 al variare della dimensione N

Come indicato nella legenda in alto a sinistra, le curve teoriche sono tratteggiate mentre le curve che riportano i tempi di esecuzione al variare di N presentano un tratto continuo. Le curve rosse fanno riferimento alla propria implementazione della DCT2 mentre le curve blu rappresentano l'esecuzione della DCT2 fast.
E' possibile notare due aspetti fondamentali:

1. Entrambe le versioni della DCT2 posseggono tempi di esecuzione (linea continua) che coincidono con quanto previsto (linea tratteggiata). Di fatti entrambe i metodi riportano tempi di esecuzione leggermente inferiori rispetto al riferimento teorico, confermando le aspettative iniziali.

2. Come previsto, i tempi di esecuzione della DCT2 in versione FFT sono inferiori rispetto alla DCT2 implementata da zero. Questo comportamento

rispecchia le aspettative iniziali ed è causato dalla maggiore efficienza algoritmica dell'algoritmo basato sulla Fast Fourier Transform, che sfrutta proprietà di simmetria e decomposizione ricorsiva per ridurre la complessità computazionale da $O(N^3)$ a $O(N^2 \log(N))$. Al contrario, la nostra implementazione calcola la trasformata direttamente tramite definizione matriciale, con un costo computazionale più elevato.

5 Conclusione

Il progetto prevedeva la realizzazione di un programma che implementa una propria versione della trasformata discreta del coseno (DCT2), con l'obiettivo di confrontarne le prestazioni in termini di tempi di esecuzione, con quelle della versione ottimizzata basata sulla Fast Fourier Transform (FFT). Questo confronto si è rivelato fondamentale per mettere in luce le differenze computazionali tra un approccio diretto, basato su formule matematiche di base, e un metodo più sofisticato che sfrutta algoritmi di trasformazione rapida.

I risultati ottenuti sono stati pienamente coerenti con le aspettative teoriche. Infatti, la DCT2 implementata tramite FFT ha mostrato tempi di calcolo nettamente inferiori rispetto alla versione "classica" realizzata da zero. Questa maggiore efficienza è dovuta principalmente alla complessità computazionale dell'algoritmo FFT, che riduce drasticamente il numero di operazioni necessarie passando da una complessità di ordine N^3 della versione diretta a $N^2 \log(N)$ della versione ottimizzata.

Tale differenza si traduce in un notevole vantaggio pratico, soprattutto per matrici di grandi dimensioni, confermando l'importanza di utilizzare algoritmi *FFT-based* per l'applicazione della DCT2 in contesti reali, come la compressione di immagini o segnali.

In sintesi, il confronto sperimentale ha validato che l'adozione di tecniche basate sulla FFT non solo è teoricamente giustificata ma anche concretamente vantaggiosa in termini di prestazioni computazionali.

6 Parte II - Compressione d'immagine

6.1 Specifiche del progetto

La seconda parte del progetto prevede di realizzare un applicativo software che realizzi una compressione tipo JPEG.
Più precisamente, è richiesto di realizzare un programma con interfaccia grafica che chieda all'utente di:

- selezionare dal File System un'immagine in formato `.bmp` in scala di grigi
- inserire il parametro F che rappresenta l'ampiezza dei macro-blocchi in cui si applica la DCT2
- inserire il parametro $d \in [0, 2F-2]$ che rappresenta la soglia di taglio delle frequenze
- effettuare la compressione dell'immagine selezionata, che prevede di:
  - dividere l'immagine in blocchi di dimensione $F \times F$, scartando le eccedenze
  - per ogni blocco `f` applicare la DCT2 (versione FFT)
  - eliminare le frequenze $c_{k,l}$ con $k + l \geq d$. Pertanto, se $d = 0$ si eliminano tutte le frequenze, viceversa, se $d = 2F-2$ elimino l'ultima.
  - applicare l'IDCT2 a ogni array delle frequenze modificato al punto precedente, arrotondando ciascun valore all'intero più vicino e impostando a 0 i valori negativi e a 255 i valori maggiori di 255
  - ricostruire l'immagine mettendo i blocchi nell'ordine giusto
  - mostrare, in una seconda finestra, il confronto tra l'immagine originale e quella ricostruita.

6.2 Compressione JPEG

Il metodo di compressione da implementare possiede un comportamento tipo JPEG nonostante mantenga alcune differenze con quest'ultimo.

JPEG è uno standard di compressione delle immagini ampiamente utilizzato, basato sull'eliminazione delle frequenze meno percettibili all'occhio umano tramite la trasformata discreta del coseno (DCT). L'algoritmo JPEG opera tipicamente su blocchi 8×8, applica la DCT2 su ciascun blocco, quantizza i coef-

ficienti secondo una matrice predefinita, detta **matrice di quantizzazione**, per poter effettuare la compressione.

Formalmente, per ogni blocco `b` di dimensione 8×8, dopo aver traslato i valori nell'intervallo $[-128, 128]$, viene applicata la `DCT2(b)` ottenendo la matrice delle frequenze $c_{k,l}$. Successivamente, data la matrice di quantizzazione Q e il parametro di qualità $q \in \{1, \ldots, 100\}$, si deriva

$$q_f = \begin{cases} \frac{100-q}{50} & se\ q > 50 \\ \frac{50}{q} & se\ q \leq 50 \end{cases}$$

e si divide ogni elemento di $\vec{\vec{c}}$ per $max(1, q_f \cdot Q)$, approssimando il risultato all'intero più vicino.

Nel nostro caso, l'approccio implementato riprende solo la prima parte del processo JPEG, ovvero la trasformata DCT2 e la selezione delle frequenze significative, trascurando però le fase di quantizzazione.

La compressione è quindi "tipo JPEG" nel senso che si basa sulla soppressione selettiva delle alte frequenze, ma resta più semplice e controllabile didatticamente, in quanto evita la perdita ulteriore dovuta alla quantizzazione.

Analogamente la procedura di decompressione prevede di eseguire a ritroso le operazioni precedentemente descritte, applicando la trasformata discreta del coseno inversa (IDCT).

7 Implementazione

7.1 Repository della libreria

Lo sviluppo del programma è avvenuto tramite l'utilizzo della piattaforma di versioning di GitHub ed è disponibile al seguente link.

7.2 Struttura del programma

Il software richiesto nella seconda parte è stato sviluppato in Python tramite il supporto delle seguenti librerie:

- **NumPy**: fornisce supporto a operazioni di calcolo numerico
- **Scipy**: fornisce l'implementazione di DCT2 e IDCT2
- **PIL**: fornisce supporto per elaborazione di immagini
- **Tkinter**: permette la realizzazione di una GUI *user-friendly*
- **Matplotlib**: definisce la finestra di confronto tra immagine originale e compressa come grafico di comparazione.

Il programma realizzato si compone di due moduli:

- **Utils.py**: modulo di *core* che contiene la logica del programma
- **Main.py**: modulo che contiene l'eseguibile dell'applicativo

7.3 Utils.py

Il modulo `Utils.py` implementa la logica di business del software, decomponibile in due macro-gruppi:

1. Realizzazione della GUI per leggere l'input utente, applicare la compressione/decompressione e visualizzare il risultato.
2. Esecuzione delle operazioni back-end come compressione, taglio delle frequenze e ricostruzione dell'immagine.

7.3.1 Run_dialog

Il metodo `run_dialog` è responsabile della realizzazione dei widget dell'interfaccia utente.

```
def run_dialog():
    global entry_file_path, entry_f, entry_d, label_dim

    root = ThemedTk(theme="plastik")
    root.title("Selezione file e parametri")

    frame_file = ttk.Frame(root)
    frame_file.pack(padx=10, pady=10)

    btn_scegli = ttk.Button(frame_file, text="Scegli file",
    ↪  command=open_file_system,style="Material.TButton")
    btn_scegli.pack(side=tk.LEFT)

    entry_file_path = ttk.Entry(frame_file, width=50,
    ↪  state='readonly')
    entry_file_path.pack(side=tk.LEFT, padx=(5, 0))

    frame_dim = ttk.Frame(root)
    frame_dim.pack(padx=10, pady=5)
    label_dim = ttk.Label(frame_dim, text="Dimension: ")
    label_dim.pack(side=tk.LEFT)

    frame_f = ttk.Frame(root)
    frame_f.pack(padx=10, pady=5)
    label_f = ttk.Label(frame_f, text="F:")
    label_f.pack(side=tk.LEFT)

    entry_f = ttk.Entry(frame_f)
    entry_f.pack(side=tk.LEFT)

    frame_d = ttk.Frame(root)
    frame_d.pack(padx=10, pady=5)
    label_d = ttk.Label(frame_d, text="d:")
    label_d.pack(side=tk.LEFT)

    entry_d = ttk.Entry(frame_d)
    entry_d.pack(side=tk.LEFT)

    btn_avvia = ttk.Button(root, text="Avvia",
    ↪  command=start,style="Material.TButton")
    btn_avvia.pack(pady=10)

    root.mainloop()
```

Viene configurata l'interfaccia utente che prevede etichette, campi per l'inseri-

mento dei dati, pulsanti per selezionare il file `.bmp` e avviare la compressione.

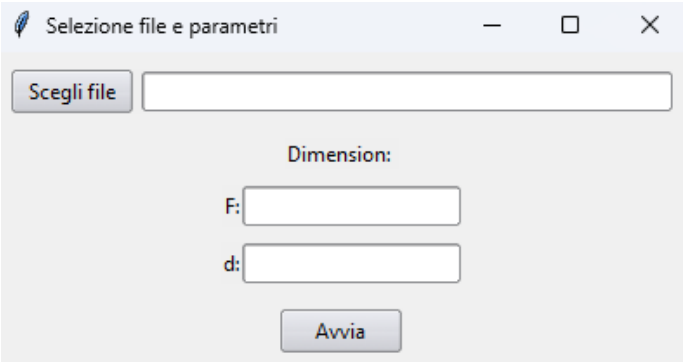

Figura 7.1: Interfaccia grafica

7.3.2 Open_file_system

Il metodo `open_file_system` si occupa di gestire la selezione del file `.bmp` dal File System e leggerne la risoluzione tramite il metodo `get_img_size` per popolare un'etichetta che comunica la dimensione in pixel dell'immagine scelta. Questa informazione è essenziale in quanto determina il limite massimo per il parametro F.

```
def open_file_system():
    global entry_file_path, path, label_dim
    path = filedialog.askopenfilename(
        filetypes=[("File BMP", "*.bmp")]
    )
    if path:
        entry_file_path.config(state='normal')
        entry_file_path.delete(0, tk.END)
        entry_file_path.insert(0, path)
        entry_file_path.config(state='readonly')
        size = get_img_size()
        if not size:
            return

        width, height = size
        label_dim.config(text=f"Dimension: {width}x{height}")
```

7.3.3 Start

Il metodo `start` viene invocato quando viene selezionato il bottone *Avvia*, dove al suo interno vengono invocati tutti i metodi necessari a effettuare la compressione/ricostruzione dell'immagine.

```
def start():
    result = check_input_values()
    if not result:
        return

    F, d = result
    blocks = divide_image_into_blocks(F)

    dct2_result = run_dct2_and_round(blocks,F,d)

    reconstructed = run_idct2(dct2_result)

    new_path = reconstruct_image(reconstructed,F)

    show_comparison(path,new_path)
```

Più precisamente invoca il metodo `check_input_values` per controllare la validità dei dati in input, `divide_image_into_blocks` che divide l'immagine in blocchi di dimensione $F \times F$, `run_dct2_and_round` che applica la DCT2 ed esegue la compressione, `run_idct2` che esegue la IDCT2 sulle frequenze approssimate, `reconstruct_image` che esegue la ricostruzione dell'immagine compressa e `show_comparison` che visualizza il confronto tra l'immagine originale e la versione ricostruita post-compressione.

7.3.4 Check_input_values

Il metodo `check_input_values` prevede di validare i dati inseriti dall'utente in input. Nel dettaglio vengono prelevati i valori dei parametri F, d e il percorso assoluto del file `.bmp` selezionato. Successivamente si controlla che non vi siano valori nulli e che i dati numerici siano di tipo intero. Più precisamente F dev'essere maggiore di 0 e al massimo pari alla larghezza/altezza dell'immagine selezionata mentre $d \in [0, 2F-2]$. Nel caso in cui uno di questi controlli fallisce, il metodo genera una finestra d'errore che informa l'utente del vincolo d'integrità violato.

```
def check_input_values():
    global entry_f, entry_d, path

    F = entry_f.get()
    d = entry_d.get()

    if not path:
        messagebox.showerror("Errore", "Selezionare
        ↪  un'immagine")
        return False
```

```
    if not F:
        messagebox.showerror("Errore", "Inserire un valore nel
        ↪  campo F")
        return False

    if not check_if_int(F):
        messagebox.showerror("Errore", "Il campo F dev'essere
        ↪  intero")
        return False

    F = int(F)

    if F <= 0:
        messagebox.showerror("Errore", "Il campo F dev'essere >
        ↪  0")
        return False

    size = get_img_size()
    if not size:
        return False

    width, height = size
    if F > width or F > height:
        messagebox.showerror("Errore", "F non può essere
        ↪  maggiore delle dimensioni dell'immagine")
        return False

    if not d:
        messagebox.showerror("Errore", "Inserire un valore nel
        ↪  campo d")
        return False

    if not check_if_int(d):
        messagebox.showerror("Errore", "Il campo d dev'essere
        ↪  intero")
        return False

    d = int(d)

    if d < 0:
        messagebox.showerror("Errore", "Il campo d dev'essere
        ↪  >= 0")
        return False

    if d > 2 * F - 2:
```

```
            messagebox.showerror("Errore", "d non può essere
            ↪  maggiore di 2F - 2")
            return False

    return F, d
```

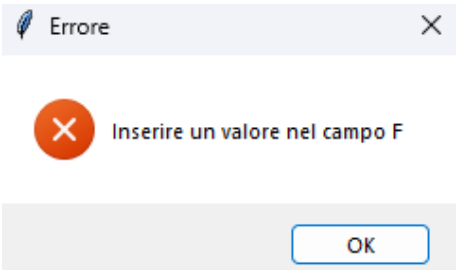

Figura 7.2: Schermata d'errore per campo F vuoto

7.3.5 Get_img_size

Il metodo `get_img_size` restituisce la dimensione, altezza e larghezza, dell'immagine selezionata. Se non viene selezionato nessun file, viene generata una finestra d'errore.

```
def get_img_size():
    try:
        img = Image.open(path)
        width, height = img.size
        img.close()
        return width, height
    except Exception as e:
        img.close()
        messagebox.showerror("Errore", str(e))
        return False
```

7.3.6 Check_if_int

Il metodo `check_if_int` viene utilizzato per determinare se il valore presente nel campo F e d è un numero intero. Se il valore inserito non è di tipo `int` viene intercettata l'eccezione `ValueError` e restituito `False`.

```
def check_if_int(value):
    try:
        int(value)
        return True
    except ValueError:
        return False
```

7.3.7 Divide_image_into_blocks

Il metodo `divide_image_into_blocks` è responsabile della suddivisione dell'immagine in blocchi di dimensione $F \times F$, su cui applicare successivamente la DCT2.

```
    def divide_image_into_blocks(F):
    blocks = []
    size = get_img_size()
    if(not size):
        return False

    width,height = size
    try:
        img_converted = Image.open(path).convert('L')
        max_width = width // F
        max_height = height // F

        for by in range(0, max_height):
            for bx in range(0, max_width):
                x = bx * F
                y = by * F
                block = img_converted.crop((x, y, x + F, y +
                ↪  F))  # (left, upper, right, lower)
                blocks.append(block)

        return blocks
    except Exception as e:
        messagebox.showerror("Errore",str(e))
        return False
```

Più precisamente si divide l'altezza e la larghezza per F, scartando le eccedenze (l'operatore // effettua la divisione intera), in modo da ottenere il numero massimo di blocchi presenti orrizzontalmente e verticalmente.

7.3.8 Run_dct2_and_round

Il metodo `run_dct2_and_round` applica la DCT2 a ogni blocco generato tramite il metodo precedente e successivamente procede a scartare le frequenze indicate dal parametro d, invocando il metodo `delete_frequencies`, restituendo la matrice di frequenze aggiornate.

```
def run_dct2_and_round(blocks,F,d):
    dct2_result = []
    for block in blocks:
        array_from_block = np.array(block)
        result = dctn(array_from_block,type=2,norm="ortho")
```

```
            result = delete_frequencies(result,F,d)
            dct2_result.append(result)

    return dct2_result
```

7.3.9 Delete_frequencies

Il metodo `delete_frequencies` prende in input una matrice di frequenze, ottenuta eseguendo la DCT2 su un blocco estratto dall'immagine, ed elimina le frequenze $c_{k,l}$ con $k + l \geq d$ azzerandole.
Più precisamente il metodo `np.add.outer` genera una matrice di dimensione $F \times F$, con indici appartenenti a $[0, F-1]$, per cui l'elemento $i, j = |i + j|$. Successivamente i valori $< d$ vengono sostituiti con il valore booleano `True`, interpretato come 1 nelle operazioni matematiche, mentre i valori $\geq d$ impostati a `False`, interpretato come 0. In questo modo si ottiene una matrice $F \times F$ che indica quali frequenze k, l sono $< d$. Infine, ogni elemento della matrice delle frequenze viene moltiplicato per il corrispettivo elemento nella matrice appena calcolata, in modo da mantenere inalterate le frequenze per cui $k + l < d$ e impostando a 0 quelle con $k + l \geq d$.

```
def delete_frequencies(dct_block, F, d):
    rounded_freq = dct_block * (np.abs(np.add.outer(range(F),
    ↪  range(F))) < d)
    return rounded_freq
```

7.3.10 Run_idct2

Il metodo `run_idct2` si occupa di eseguire l'IDCT2 sui blocchi di frequenze tagliate per poter ottenere i blocchi in scala di grigi dell'immagine compressa. Nel dettaglio, per ogni blocco di frequenze, viene eseguita la IDCT2 ottenendo un blocco di pixel in scala di grigi. Successivamente il valore di ogni pixel viene arrotondato all'intero più vicino e tramite il metodo `np.clip` vengono impostati a 0 i valori negativi e a 255 i valori maggiori di 255.

```
def run_idct2(frequencies):
    blocks_reconstructed = []

    for f in frequencies:
        idct2_result = idctn(f,type=2,norm="ortho")
        idct2_result = np.round(idct2_result)
        idct2_result = np.clip(idct2_result, 0,
        ↪  255).astype(np.uint8)
        blocks_reconstructed.append(idct2_result)

    return blocks_reconstructed
```

7.3.11 Reconstruct_image

Il metodo `reconstruct_image` prende in input l'insieme dei blocchi ottenuti dall'esecuzione dell'IDCT2 e restituisce l'immagine compressa ricostruita, disponendo nel giusto ordine i blocchi compressi. Importante notare che in questo caso la divisione intera causa uno scarto delle eccedenze, che durante la ricostruzione genera delle porzioni totalmente nere dell'immagine.

```
def reconstruct_image(blocks_reconstructed,F):
    size = get_img_size()
    if(not size):
        return False
    width,height = size
    new_image = Image.new('L',(width,height))
    new_image_path = "immagini/my_image_compressed.bmp"
    max_width = width // F
    max_height = height // F

    for j in range(max_height):
        for i in range(max_width):
            block = blocks_reconstructed.pop(0)
            x = i * F
            y = j * F
            new_image.paste(Image.fromarray(block), (x, y))


    new_image.save(new_image_path)
    new_image.close()

    return new_image_path
```

7.3.12 Show_comparison

Il metodo `show_comparison` si occupa di generare una schermata dove viene messa a confronto l'immagine originale con l'immagine ricostruita dopo la compressione.

```
def show_comparison(original_image, compressed_image):
    def on_closing():
        original_image.close()
        compressed_image.close()

        root.destroy()
        root.quit()

    original_image = Image.open(original_image).convert('L')
```

```
compressed_image =
↪  Image.open(compressed_image).convert('L')
root = tk.Tk()
root.title("Confronto Originale e Compressa")

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 7))

ax1.imshow(original_image, cmap="gray",
↪  interpolation="nearest")
ax1.set_title("Immagine Originale")
ax1.axis("off")

ax2.imshow(compressed_image, cmap="gray",
↪  interpolation="nearest", vmin=0, vmax=255)
ax2.set_title("Immagine Ricostruita")
ax2.axis("off")

canvas = FigureCanvasTkAgg(fig, master=root)
canvas.draw()
canvas.get_tk_widget().pack()

root.protocol("WM_DELETE_WINDOW", on_closing)

root.mainloop()
```

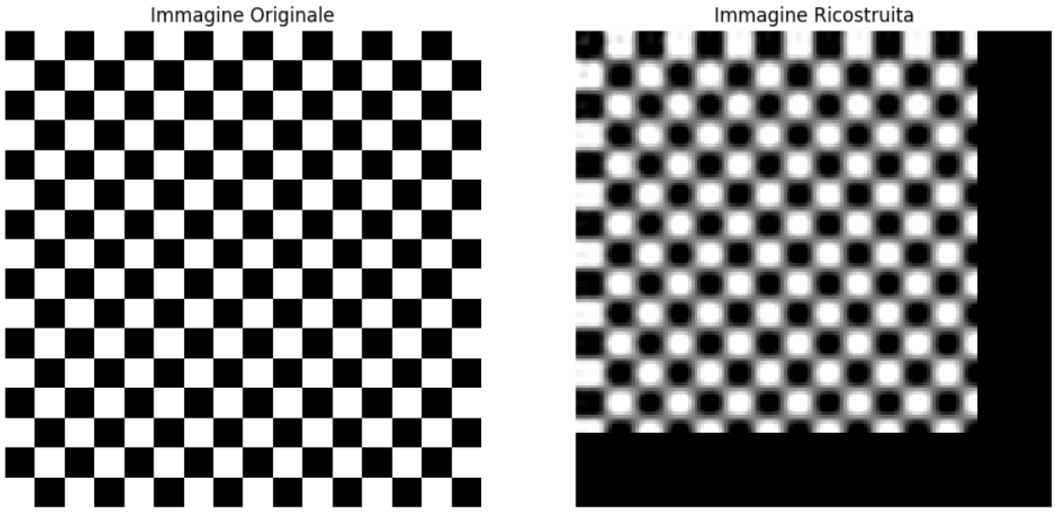

Figura 7.3: Confronto fra immagine originale e immagine ricostruita

8 Analisi dei risultati

All'interno di questo capitolo vengono descritti i risultati ottenuti dopo aver eseguito diversi test su immagini in formato `.bmp`, presenti nella cartella "immagini" del repository del progetto.

8.1 Casi particolari

Il primo macro-gruppo di test effettuati, riguardano l'utilizzo di valori estremi al proprio intervallo per entrambi i parametri F, d.

1. Il primo caso prevede di scegliere F uguale alla dimensione dell'immagine (nel caso di immagini non quadrate, si assume il lato minore) e $d = 2F-2$. In questo modo si generano blocchi $F \times F$ che tendono a coprire l'intera immagine in un unico blocco e la soglia di taglio delle frequenze d stabilisce che venga cancellata solo l'ultima frequenza. Di conseguenza ci si aspetta un'immagine ricostruita con caratteristiche molto simili all'immagine originale poichè vengono mantenute tutte le frequenze meno una, nonostante il valore F elevato che causa una perdita dei dettagli locali e l'introduzione di artefatti visivi.

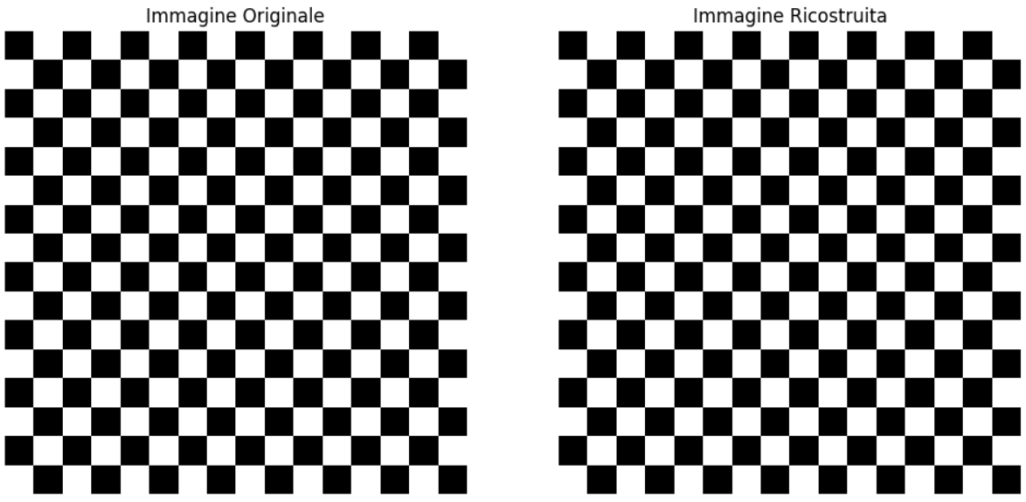

Figura 8.1: Confronto immagine 160x160 con F=160, d=318

Di seguito possiamo notare l'esito della compressione su un'immagine 160×160 con $F = 160$ e $d = 318$. L'immagine ricostruita non pos-

siede evidenti differenze con l'immagine originale poichè qausi tutte le frequenze ottenute dall'applicazione della DCT2 non vengono eliminate.

2. Il secondo caso prevede di scegliere F uguale alla dimensione dell'immagine (nel caso di immagini non quadrate, si assume il lato minore) e $d = 0$. In questo modo si generano blocchi $F \times F$ che tendono a coprire l'intera immagine in un unico blocco e la soglia di taglio delle frequenze d stabilisce che vengano cancellate tutte le frequenze. Di conseguenza ci si aspetta una totale compressione, a causa dell'azzeramento di tutte le frequenze, producendo un'immagine totalmente nera.

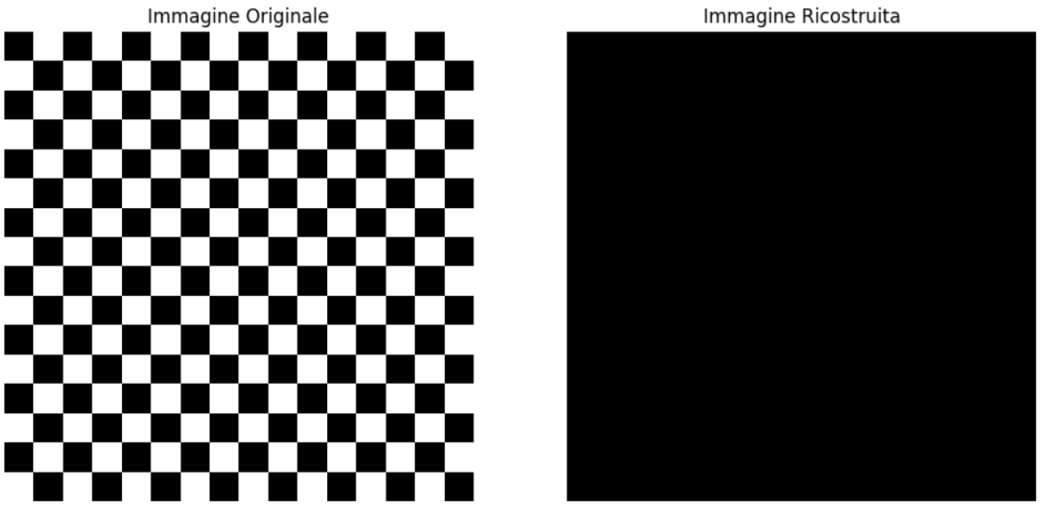

Figura 8.2: Confronto immagine 160x160 con F=160, d=0

Dopo aver eseguito la compressione è possibile notare immediatamente che l'immagine ricostruita possiede tutti i pixel impostati a zero. Questo è causato dalla scelta di eliminare tutte le frequenze, $d = 0$, che vengono azzerate, e che durante l'esecuzione della IDCT2 genera pixel di colore nero.

8.2 Casi generici

Il secondo gruppo di test si concentra sull'assegnazione di valori differenti per F e d, senza però considerare il comportamento della compressione nei casi limite.

Figura 8.3: Confronto immagine Bongo 1011x661 con F=280, d=120

In questo test è stata applicata la compressione di un'immagine che raffigura un animale in dimensione 1011×661, con $F = 280$ e $d = 120$.
Preliminarmente è possibile osservare che F possiede un valore elevato, che comporta l'applicazione della trasformata DCT2 su blocchi di grandi dimensioni. Questo ha due principali conseguenze: da un lato, si riduce la localizzazione spaziale dell'informazione, rendendo meno efficace la cattura di dettagli fini o variazioni locali; dall'altro, l'aumento della dimensione del blocco comporta un incremento del costo computazionale, poiché la complessità della trasformata cresce con la dimensione del blocco. Inoltre è possibile notare che il valore di F non è multiplo di 661, dimensione più piccola dell'immagine, causando un taglio dei bordi sia lateralmente che verticalmente.

Il valore di soglia $d = 120$ implica che, per ciascun blocco, vengono mantenuti solo i primi 120 coefficienti DCT, mentre i restanti vengono annullati. Considerando che un blocco $F \times F = 280 \times 280$ contiene 78400 coefficienti, la soglia scelta corrisponde a conservare soltanto una piccolissima frazione dello spettro ($\approx 0.15\%$).

Ciò suggerisce un approccio fortemente compressivo, che dovrebbe comportare una perdita significativa di dettaglio visivo, soprattutto nelle aree ad alta frequenza dell'immagine. Tuttavia, l'effetto percettivo dipenderà anche dal contenuto dell'immagine: aree ampie e uniformi (come sfondi o superfici lisce) potrebbero risultare visivamente accettabili anche con una soglia così bassa, mentre le zone con texture o bordi netti tenderanno a degradarsi più visibilmente. Di fatti è possibile notare come lo sfondo ricostruito possiede una lieve sfumatura che ne aumenta il contrasto mentre nell'immagine originale lo sfondo è quasi del tutto nero. Inoltre nei bordi che delimitano l'animale, come il muso, è possibile notare una perdita nel dettaglio che causa una sfocatura dell'immagine causata dall'azzeramento di alcune frequenze.

Nonostante la soglia di taglio relativamente bassa, l'immagine ricostruita mostra alcune differenze rispetto all'originale, tuttavia tali variazioni risultano generalmente impercettibili all'occhio umano. L'unico effetto negativo signi-

ficativo è legato alla scelta di F non multiplo delle dimensioni dell'immagine, che provoca un ritaglio parziale dei bordi, con conseguente perdita di informazioni lungo i margini.

Figura 8.4: Confronto immagine cattedrale 2000x3008 con F=8, d=4

In questo test è stata applicata la compressione di un'immagine che raffigura una cattedrale di dimensione 2000×3008, con $F = 8$ e $d = 4$. Questo test è stato eseguito per simulare una compressione tipo JPEG utilizzando blocchi 8×8 che consentono quindi di catturare un elevato livello di dettaglio eseguendo campionamento e compressione su gruppi di 64 pixel.

Il valore di soglia $d = 4$ implica che, per ciascun blocco, vengono mantenute solo le prime 4 frequenze mentre i coefficienti restanti vengono azzerati. Considerando che in un blocco sono presenti 64 coefficienti, la soglia di scelta corrisponde a conservare soltanto il 6.5%.

Questo indica un approccio più aggressivo dal punto di vista della compressione rispetto al caso precedente, ma che, dal punto di vista percettivo, non introduce differenze visivamente rilevanti. Ciò è dovuto al fatto che i blocchi di dimensione 8×8 utilizzati per la trasformata risultano molto piccoli rispetto alle dimensioni complessive dell'immagine, permettendo una rappresentazione locale più efficace.

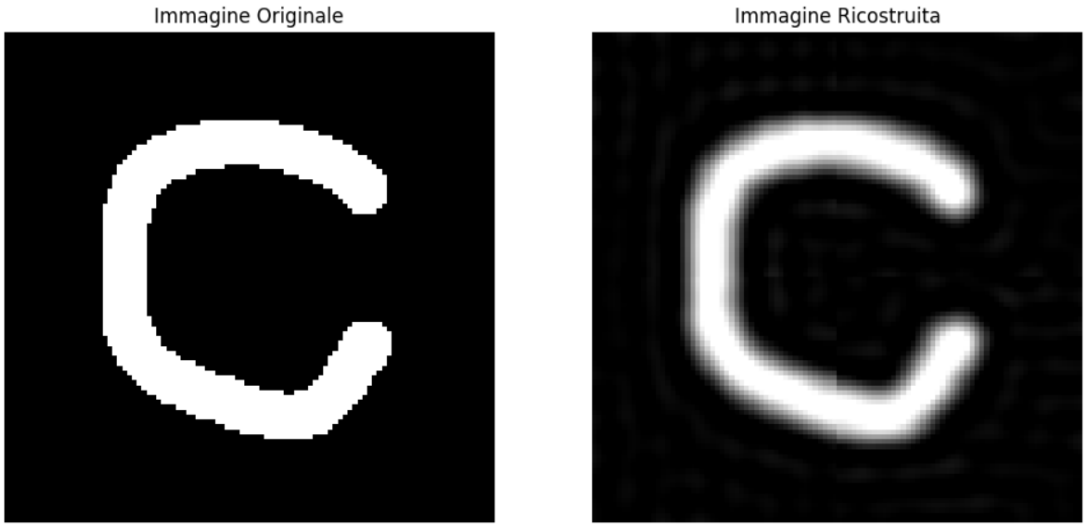

Figura 8.5: Confronto immagine cattedrale 2000x3008 con F=8, d=4

Il terzo test è stato eseguito su un'immagine che raffigura una lettera C di colore bianco su sfondo nero, di dimensione 100×100, con $F = 50$ e $d = 15$.

Il valore $F = 50$, essendo un divisore esatto delle dimensioni dell'immagine, garantisce che nella ricostruzione non si verifichi perdita di informazione ai margini, e che l'immagine venga suddivisa esattamente in blocchi di 50×50, ossia 2500 pixel ciascuno.

La soglia di taglio $d = 15$ implica che, per ciascun blocco, vengono mantenute solo le prime 15 frequenze, corrispondenti a circa lo 0,6% del totale dei coefficienti, mentre i restanti vengono azzerati.

Ci si aspetta quindi una compressione molto marcata, dovuta alla combinazione tra la grande dimensione dei blocchi e una soglia di conservazione molto bassa.

Analizzando il risultato, si osserva che la lettera viene ricostruita con una discreta precisione nella forma generale, ma con un livello di dettaglio visivamente ridotto. In particolare, si nota una sfocatura nei toni bianchi, causata dall'eliminazione delle alte frequenze. Tuttavia, lo sfondo nero, essendo uniforme e privo di variazioni, risulta correttamente ricostruito e non risente della compressione applicata.

9 Conclusione

Il progetto prevedeva di analizzare gli effetti della compressione d'immagine, esplorando diverse combinazioni di parametri applicate a differenti tipologie di immagini, caratterizzate da un diverso livello di dettaglio e complessità visiva. Attraverso l'utilizzo della trasformata discreta del coseno bidimensionale (DCT2), sono stati applicati vari scenari di compressione basati sulla dimensione dei blocchi $F \times F$ e sulla soglia di taglio d, che definisce il numero di coefficienti DCT mantenuti in ciascun blocco. Le immagini considerate includevano soggetti realistici ad alta risoluzione, elementi con dettagli strutturali pronunciati e immagini sintetiche con forme semplici e sfondi uniformi.

Dai test effettuati si sono potute osservare alcune considerazioni generali:

- Blocchi di piccole dimensioni ($F = 8$, tipici dello standard JPEG) garantiscono una maggiore capacità di catturare variazioni locali, con buona qualità percettiva anche a soglie di taglio basse. Tuttavia, aumentano il numero di trasformazioni da eseguire e quindi il costo computazionale.
- Blocchi molto grandi (F vicino o uguale alla dimensione dell'immagine) riducono drasticamente la localizzazione spaziale e rendono l'immagine sensibile alla perdita di dettaglio, specialmente nelle zone ad alta frequenza. In questi casi, la compressione tende a introdurre artefatti visivi più evidenti.
- Il parametro d ha un impatto diretto sul livello di qualità e compressione: soglie basse ($d \ll F^2$) determinano una maggiore perdita di informazione ma possono mantenere la struttura globale dell'immagine se questa presenta ampie zone omogenee.
- Le immagini con sfondi uniformi o geometrie semplici risultano più tolleranti alla compressione, mentre quelle con texture complesse, bordi netti o variazioni rapide sono più sensibili alla perdita di frequenze alte.
- La scelta di F deve essere effettuata tenendo conto anche della compatibilità con la dimensione dell'immagine, poiché valori non multipli possono causare la perdita di porzioni dell'immagine ai bordi.

Nel complesso, l'esperimento ha dimostrato come l'efficienza della compressione dipenda fortemente dalla natura del contenuto dell'immagine e dalla configurazione dei parametri utilizzati. Un corretto bilanciamento tra qualità visiva e grado di compressione richiede un'analisi mirata del tipo di immagine e degli obiettivi di compressione. I risultati ottenuti forniscono un buon punto di partenza per sviluppare compressori adattivi in grado di ottimizzare le risorse in base al contenuto.