



Università degli studi Milano Bicocca
Facoltà di informatica magistrale
Milano, Padiglione U24

Anno accademico 23/24

Francesco Cavallini

Matricola: 920835
f.cavallini8@campus.unimib.it
Corso di studi di Informatica Magistrale

DCT, DCT2 e compressione di immagini Metodi del Calcolo Scientifico

Relazione Progetto 2

Consegna:

"""""

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.

"""""

Riferimenti :

Repository git-hub: | https://github.com/VR3ED/Scientific_Calculus_2

Sommario

0. Introduzione.....	3
0.1. Obbiettivi.....	3
0.2. Scelte implementative.....	3
1. Parte-1: Sviluppo DCT & DCT2.....	4
1.1. Requisiti tecnici:.....	4
1.2. Le basi per lo sviluppo: cosa ci aspettiamo:	4
1.3. Le basi per lo sviluppo: come è strutturato il codice:.....	5
1.4. Utilss.py: Implementazione DCT:	7
1.5. Utilss.py: Implementazione DCT2:	9
1.6. Utilss.py: Metodi di libreria per DCT e DCT2:.....	11
1.7. Analisi dei risultati e conclusioni:	12
2. Parte-2: Software con interfaccia grafica	14
2.1. Requisiti tecnici:.....	14
2.2. Le basi per lo sviluppo: cosa ci aspettiamo:	15
2.3. Le basi per lo sviluppo: come è strutturato il codice:.....	16
2.4. ProcessImages.py: generate_blocks:.....	20
2.5. ProcessImages.py: apply_dct2:.....	21
2.6. ProcessImages.py: apply_idct2:.....	22
2.7. ProcessImages.py: save_compressed_image:.....	24
2.8. Analisi dei risultati prodotti dall'app:.....	25
2.9. Problemi riscontrati con l'applicazione:.....	29
2.10. Conclusioni:	29

0. Introduzione

0.1. Obiettivi

Come definito nella pagina iniziale, l'obiettivo del progetto è quello di creare un implementazione custom della DCT2. In particolare il lavoro è suddiviso in 2 parti in cui abbiamo:

- **Parte 1:** Implementare la DCT2 come spiegata a lezione in un ambiente open source a vostra scelta e confrontare i tempi di esecuzione con la DCT2 ottenuta usando la libreria dell'ambiente utilizzato (che si presuppone essere nella versione FFT)
- **Parte 2:** Implementare compressione di immagini (nel formato .bmp), sfruttando la DCT2: una metodologia comunemente impiegata per emulare la compressione JPEG. L'obiettivo principale consiste nello sviluppo di un'applicazione che consenta all'utente di selezionare un'immagine e regolare il livello di compressione, offrendo la possibilità di visualizzare sia l'immagine originale che quella compressa.

0.2. Scelte implementative

Si sceglie di utilizzare il linguaggio python per lo sviluppo di entrambe le parti del progetto, utilizzando visual studio code come ambiente di sviluppo. Le motivazioni dietro a questa scelta sono:

- Vasta disponibilità di librerie open source per i paragoni necessari ad altre librerie afici
- Vasta gamma di librerie per lo sviluppo di grafici
- Sintassi semplice ed intuitiva, renderà le fasi di analisi del codice più scorrevoli, e, allo stesso tempo, diversamente da C# (il linguaggio utilizzato per lo sviluppo del progetto 1) questo linguaggio è più adatto alla stesura di codice in ambito di calcolo scientifico.

1. Parte-1: Sviluppo DCT & DCT2

1.1. Requisiti tecnici:

I requisiti necessari per lo sviluppo della prima parte sono quelli di prestare molta attenzione a come viene scalata la DCT2 (o la DCT). Come caso test è necessario verificare che il seguente blocchetto 8×8 :

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

venga trasformato in questo modo dalla DCT2:

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

È inoltre necessario controllare per la DCT monodimensionale che la prima riga del blocchetto 8×8 precedentemente mostrato venga trasformata in:

4.01e+02	6.60e+00	1.09e+02	-1.12e+02	6.54e+01	1.21e+02	1.16e+02	2.88e+01
----------	----------	----------	-----------	----------	----------	----------	----------

1.2. Le basi per lo sviluppo: cosa ci aspettiamo:

Per questo progetto, analizzeremo le prestazioni della nostra implementazione della DCT2 rispetto a quella fornita da una libreria di riferimento, concentrandoci sui tempi di esecuzione e sulle complessità teoriche dei due approcci. In particolare, studieremo come i tempi di calcolo variano al crescere delle dimensioni della matrice fornita in input.

Ci si aspetta che la nostra implementazione manuale della DCT2 dovrebbe presentare una complessità temporale proporzionale a $O(N^3)$, mentre la versione basata su una libreria ottimizzata, tipicamente costruita utilizzando FFT, dovrebbe raggiungere una complessità $O(N^2 \log(N))$. Questa differenza nasce dalle diverse strategie algoritmiche impiegate:

- la nostra implementazione si basa su una definizione diretta della DCT2,
- le librerie ottimizzate sfruttano algoritmi avanzati per ridurre il numero di operazioni necessarie.

Nel corso dell'analisi, confronteremo i tempi di esecuzione ottenuti per matrici di dimensioni crescenti, valutando in che misura i risultati sperimentali confermano le complessità teoriche attese.

1.3. Le basi per lo sviluppo: come è strutturato il codice:

La parte 1 è stata sviluppata suddividendo il codice nei seguenti file:

- generate_graphs.py
- main.py
- test.py
- utilss.py

Dove abbiamo che:

test.py: Implementa i requisiti di controllo sull'accuratezza precedentemente accennati:

```
def run_test():
    # Matrice di test 8x8
    test_matrix = np.array([
        [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
        [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
        [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0],
        [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
    ])
    print(test_matrix)

    #risultati aspettati:
    expected_dct2_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]
    ])
    expected_dct_first_row = 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]
    ])

    with open("test_results.txt", "w") as file:
        try:
            # Verifica DCT calcolata manualmente sulla prima riga
            a = test_matrix[0, :] # Prima riga della matrice di test
            dct = utilss.dct_created(a) # DCT calcolata manualmente solo sulla prima riga
            formatted_dct = ["{:.2e}".format(val) for val in dct]
            print("\n-----TEST DCT HomeMade-----")
            print(formatted_dct)
            # Verifica che il risultato della DCT calcolata manualmente sulla prima riga sia simile a quello fornito
            assert np.allclose(dct, expected_dct_first_row, rtol=1e-2), "DCT HomeMade test failed!"
            file.write("DCT HomeMade test passed!\n")
        except AssertionError as e:
            file.write(str(e) + "\n")

            # Verifica DCT2 calcolata manualmente
            try:
                dct2_result = utilss.dct2_created(test_matrix) # DCT2 calcolata manualmente
                print("\n-----TEST DCT2 HomeMade-----")
                print(dct2_result)
                # Verifica che il risultato della DCT2 calcolata manualmente sia simile a quello fornito
                assert np.allclose(dct2_result, expected_dct2_result, rtol=1e-2), "DCT2 HomeMade test failed!"
                file.write("DCT2 HomeMade test passed!\n")
            except AssertionError as e:
                file.write(str(e) + "\n")

            try:
                # Verifica DCT con libreria sulla prima riga
                dct_lib = utilss.dct_library(a) # DCT calcolata con libreria solo su prima riga
                formatted_dct_lib = ["{:.2e}".format(val) for val in dct_lib]
                print("\n-----TEST DCT Library-----")
                print(formatted_dct_lib)
                # Verifica che il risultato della DCT calcolata con la libreria sia simile a quello fornito
                assert np.allclose(dct_lib, expected_dct_first_row, rtol=1e-2), "DCT Library test failed!"
                file.write("DCT library test passed!\n")
            except AssertionError as e:
                file.write(str(e) + "\n")

            try:
                # Verifica DCT2 con libreria sulla matrice di test
                dct2_result_lib = utilss.dct2_library(test_matrix) #
                print("\n-----TEST DCT2 Library-----")
                print(dct2_result_lib)
                # Verifica che il risultato della DCT2 calcolata con la libreria sia simile a quello fornito
                assert np.allclose(dct2_result_lib, expected_dct2_result, rtol=1e-2), "DCT2 Library test failed!"
                file.write("DCT2 library test passed!\n")
            except AssertionError as e:
                file.write(str(e) + "\n")

        print("\n-----TEST COMPLETED-----")
```

I commenti nel codice mostrato dovrebbero fornire tutte le informazioni necessarie per individuare tutti i test che vengono svolti per rispettare i requisiti tecnici imposti dalla consegna. Inoltre si può visualizzare che ogni test è inserito all'interno di un try-catch se il test concluderà con successo allora verrà scritto in un file txt che il test è stato completato correttamente, altrimenti, verrà scritto il messaggio di errore.

Sempre nel file tests.py viene poi implementato un altro metodo per misurare il tempo necessario per eseguire computazioni di DCT e DCT2 su matrici di grandezze progressivamente sempre più grandi (da $N = 50$ ad $N = 1000$):

```
def test_N():
    # Testo le prestazioni della DCT2 su matrici di dimensioni crescenti fino a 1024x1024.
    # da 50x50 a 1000x1000, con incrementi di 50.
    matrix_dimensions = list(range(50, 1001, 50))
    # nota: per diminuire i tempi basta diminuire il range da 1001 a 501

    # Creo una lista vuota per memorizzare i tempi di esecuzione della DCT2 utilizzando la libreria scipy.
    times_scipy_dct = []

    # Creo una lista vuota per memorizzare i tempi di esecuzione della mia implementazione della DCT2.
    times_my_dct = []

    for n in matrix_dimensions:
        print("Dimension: ", n)
        # Imposto un seed per il generatore di numeri casuali per garantire la riproducibilità dei test.
        np.random.seed(5)

        # Genero una matrice NxN con valori casuali compresi tra 0 e 255.
        matrix = np.random.uniform(low=0.0, high=255.0, size=(n, n))

        # Misuro il tempo di esecuzione della DCT2 utilizzando la libreria scipy.
        time_scipy = timeit.timeit(lambda: utilss.dct2_library(matrix), number=1)
        times_scipy_dct.append(time_scipy)

        # Misuro il tempo di esecuzione della mia implementazione della DCT2.
        time_my_dct = timeit.timeit(lambda: utilss.dct2_created(matrix), number=1)
        times_my_dct.append(time_my_dct)

    # Restituisco i tempi di esecuzione e le dimensioni delle matrici testate.
    return times_scipy_dct, times_my_dct, matrix_dimensions
```

Anche qui abbiamo che i commenti sono abbastanza autoesplicativi, tutto quello che succede è l'esecuzione ricorsiva di metodi di DCT2 (creato a mano e di libreria) cambiando ad ogni iterazione la grandezza di N . Ad ogni iterazione viene anche misurato il tempo di esecuzione di modo da poterlo graficare in seguito.

main.py: Avvia semplicemente i test definiti nel file test.py

generate_graphs.py: Contiene una sola funzione che colleziona i timestamps di esecuzione forniti dai metodi in test.py e plotta un grafico per confrontare le prestazioni temporali 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. Nota che di questo file non verrà mostrato il codice in quanto non rilevante ai fini della relazione, ma verranno mostrati i grafici da essa prodotti.

utilss.py: Cuore del progetto, contiene infatti i due metodi creati manualmente per l'implementazione di DCT e DCT2. Per ciascuna di queste verrà dedicato un capitolo a parte che ne spiega l'implementazione. Questo file contiene inoltre altri 2 metodi wrapper che servono a chiamare la versione da libreria (Scipy) delle implementazioni di DCT e DCT2.

1.4. Utilss.py: Implementazione DCT:

L'idea di base per la realizzazione di DCT è creare una trasformata matematica utilizzata per rappresentare un segnale discreto (dal dominio del tempo) come una somma di funzioni coseno (conversione al dominio della frequenza) con pesi opportuni. Si può usare poi questo principio per scartare le parti con minor quantitativo informativo per creare compressioni di segnali ed immagini. Inoltre DCT è un'operazione reversibile tramite l'operazione IDCT, tuttavia, la ricostruzione non è perfetta a causa di un troncamento delle frequenze che avviene durante la trasformazione DCT. Più nello specifico abbiamo che la DCT funziona secondo questo principio:

1. **Calcolo vettore α :** Il calcolo di α è semplicemente una fase di normalizzazione necessaria per costruire la matrice di trasformazione della DCT seguendo la definizione teorica. Questo vettore contiene infatti i fattori di normalizzazione che dipendono dalla posizione (indice i) nella matrice di trasformazione:

- Per $i = 0$: abbiamo che $\alpha[0] = \frac{1}{\sqrt{n}}$
- Per $i > 0$: abbiamo che $\alpha[i] = \sqrt{\frac{2}{n}}$

2. **Calcolo DCT tramite matrice di trasformazione (T):** Abbiamo che DCT (X) utilizza funzioni coseno con frequenze diverse per rappresentare il segnale, ossia abbiamo per ogni vettore X_i che:

$$X_i = \alpha_i \cdot \sum_{j=0}^{N-1} x_j \cdot \cos\left(\frac{i \cdot \pi \cdot (2j+1)}{2N}\right)$$

Dove x_j Rappresenta il campione del segnale originale (nel dominio spazio/tempo)

Per semplicità implementativa possiamo però dividere questa trasformazione in:

1. **Calcolare la matrice di trasformazione T :** Ossia una matrice unicamente composta da vettori riga ($T[i,:]$) che rappresentano funzioni coseno a diverse frequenze; ogni posizione singola di questa matrice è calcolata in questo modo:

$$T[i,j] = \alpha[i] \cdot \cos\left(\frac{i \cdot \pi \cdot (2j+1)}{2n}\right)$$

Nota che la parte della formula $\cos(...)$ applica la trasformazione cosinuoidale, utilizzando il vettore α assicuriamo che la trasformazione sia ortogonale; ossia:

- I vettori riga saranno mutuamente ortogonali: Il prodotto scalare tra due vettori riga $T[i,:]$ e $T[j,:]$ della matrice deve essere zero per $i \neq j$. Questo garantisce che le frequenze diverse (associate a i e j) siano indipendenti l'una dall'altra.
- I vettori riga saranno normalizzati: Il prodotto scalare di ciascun vettore con sé stesso deve essere $\neq 0$

2. **Applicare la trasformazione al segnale originale:** ossia moltiplicare il segnale originale per il vettore di trasformazione:

$$X = x \cdot T$$

Dove X contiene i coefficienti della DCT, che rappresentano le ampiezze delle diverse frequenze.

Se andiamo a visualizzare quindi l'implementazione del metodo di creazione manuale della DCT all'interno del file `utilss.py` possiamo osservare che verranno applicati gli stessi principi teorici appena descritti:

```

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

    # Calcolo del risultato (X = T * input_vector)
    dct_result = np.dot(transformation_matrix, input_vector)
    return dct_result

```

Si decide di esportare il calcolo di α e di T in un altro metodo a parte

Applica la matrice di trasformazione al segnale originale: $X = T \cdot x$

Come possibile osservare il calcolo di α e di T viene spostato in un metodo diverso (questo semplicemente per migliorare la leggibilità del codice) , questo metodo è il seguente:

```

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

```

Calcolo di α con:

- $\alpha[0] = \frac{1}{\sqrt{n}}$
- $\alpha[i] = \sqrt{\frac{2}{n}}$ (per $i > 0$)

Calcolo matrice T cella per cella: $T[i,j] = \alpha[i] \cdot \cos\left(\frac{i \cdot \pi \cdot (2j+1)}{2n}\right)$

Svolgendo i calcoli nell'ordine di esecuzione abbiamo quindi:

1. Calcolo di α
2. Calcolo di T
3. Applicazione di T al segnale originale

Questa serie di operazioni corrisponde quindi perfettamente con la teoria precedentemente descritta.

1.5. Utilss.py: Implementazione DCT2:

La DCT2 bi-dimensionale (2D) è un'estensione della DCT al dominio delle matrici. Il principio fondamentale della DCT2 è che concentra la maggior parte dell'informazione utile nei coefficienti a bassa frequenza (tipicamente in prossimità dell'angolo in alto a sinistra della matrice trasformata), rendendo possibile una significativa riduzione della dimensionalità del dato e una compressione efficace. Si ha infatti che la matrice risultante da dalla trasformazione trasfoallrisultatrasDCT2 ($X_{i,j}$) viene calcolata applicando separatamente DCT lungo le righe e le colonne alla matrice originale, ossia:

$$X_{i,j} = \alpha_i \cdot \alpha_j \cdot \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x,y) \cdot \cos\left(\frac{i \cdot \pi \cdot (2x+1)}{2N}\right) \cdot \cos\left(\frac{j \cdot \pi \cdot (2y+1)}{2M}\right)$$

```
graph TD; X_ij["X_{i,j} = α_i · α_j · ∑_{x=0}^{N-1} ∑_{y=0}^{M-1} f(x,y) · cos((i · π · (2x+1)) / 2N) · cos((j · π · (2y+1)) / 2M)"]; X_ij --- Col["Applicazione DCT sulle colonne"]; X_ij --- Row["Applicazione DCT sulle righe"];
```

Dove:

- $f(x,y)$ è il valore dell'elemento della matrice di input.
- N e M sono le dimensioni della matrice di input.
- α_i Insieme a α_j sono i 2 fattori di normalizzazione per le rispettive DCT.

Essendo questa trasformazione direttamente derivata da DCT è facile capire che anche in questo caso è reversibile (mantenendo sempre il problema di ricostruzione perfetta a causa di un troncamento delle frequenze che avviene durante la trasformazione DCT).

NOTA: DCT2 nel formato JPEG

È fondamentale evidenziare che la tecnica appena descritta, pur essendo efficace, non corrisponde a quella impiegata dal formato JPEG per comprimere le immagini. Questo perché alcune operazioni critiche potrebbero generare distorsioni e artefatti nella ricostruzione finale. Il formato JPEG, invece, utilizza strategie specifiche per superare tali problematiche, garantendo una compressione efficiente senza compromettere eccessivamente la qualità dell'immagine.

Avendo quindi già spiegato il funzionamento della funzione DCT nel capitolo precedente diventa molto facile spiegare il codice generato per l'implementazione di DCT2

Se andiamo quindi a visualizzare quindi l'implementazione del metodo di creazione manuale della DCT2 (sempre all'interno del file `utilss.py`) possiamo osservare che verranno applicati gli stessi principi teorici appena descritti:

```
# Funzione per DCT2 creata
def dct2_created(input_matrix):
    n, m = input_matrix.shape

    # Creazione 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
```

La DCT2 implementata nel codice antecedente segue una strategia "a separazione di variabili". Ossia viene sfruttata la linearità della trasformazione consentendo di ridurre il problema bidimensionale in **due trasformazioni monodimensionali** consecutive:

1. Prima si calcola la DCT lungo una dimensione (colonne).
2. Poi si calcola la DCT lungo l'altra dimensione (righe).

Questo approccio è computazionalmente efficiente perché riduce la complessità computazionale rispetto al calcolo diretto della DCT2, specialmente per matrici di grandi dimensioni. Abbiamo infatti che la definizione diretta della DCT2 per una matrice $N \times M$ è:

$$X_{i,j} = \alpha_i \cdot \alpha_j \cdot \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x,y) \cdot \cos\left(\frac{i \cdot \pi \cdot (2x+1)}{2N}\right) \cdot \cos\left(\frac{j \cdot \pi \cdot (2y+1)}{2M}\right)$$

Questo implica di avere due somme nidificate ($\Sigma\Sigma$) su tutte le $N \times M$ celle della matrice. Dove ogni somma richiede $O(N \times M)$ moltiplicazioni e somme per calcolare un singolo elemento $X_{i,j}$. Per calcolare tutti i $N \times M$ coefficienti della DCT2, la complessità totale diventa:

$$O((N \cdot M)^2) = O(N^2 \cdot M^2) \cong O(N^4)$$

Invece, utilizzando il principio della scomposizione di due operazioni lineari abbiamo che:

- Per la trasformazione lungo le colonne: per una colonna sola richiede $O(n^2)$ operazioni, quindi per tutte le colonne: $O(M \cdot N^2)$
- Per la trasformazione lungo le righe: analogamente per una riga sola richiede $O(m^2)$ operazioni, quindi per tutte le colonne: $O(N \cdot M^2)$

Avendo quindi che la complessità totale utilizzando questo metodo è:

$$O(M \cdot N^2) + O(N \cdot M^2) \cong O(M^3)$$

Avendo quindi che questa strategia è computazionalmente più efficiente (e rispecchia anche la complessità di implementazione che ci aspettavamo all'inizio)

NOTA: matrici quadrate

Nota che per entrambi i risultati finali se operiamo con una matrice quadrata allora i " \cong " diventano " $=$ "

1.6. Utilss.py: Metodi di libreria per DCT e DCT2:

Come già anticipato si usano i metodi della libreria SciPy per avere un paragone con altre implementazioni di DCT e DCT2, di seguito vengono mostrati i metodi wrapper che richiamano le funzioni di libreria:

```
# 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')
```

La funzione DCT2 prende in input gli stessi parametri della funzione DCT, ma viene richiamata due volte per creare la matrice bidimensionale sul trasposto della matrice fornita in input. Si da in pasto al metodo DCT2 la matrice trasposta appunto per creare un'analogia con il codice scritto manualmente: qui, la trasposizione fa sì che la prima trasformazione venga applicata lungo le righe (tramite T), e la seconda lungo le colonne (invertendo con un altro T). Se non si usasse la trasposizione T nella funzione `dct2_library`, la trasformazione sarebbe applicata solo lungo una dimensione della matrice (ad esempio, solo lungo le righe o solo lungo le colonne), anziché su entrambe le dimensioni.

NOTA: Coefficienti di normalizzazione

Si vuole far notare però che questa libreria implementa il fattore di scaling α in maniera analoga da come lo abbiamo appena visto dalle descrizioni teoriche che abbiamo dato nelle sezione [1.4](#) ma in maniera diversa da come abbiamo visto l'implementazione teorica della DCT.

Abbiamo infatti visto durante le lezioni teoriche che la formula per la DCT sarebbe:

$$X_i = \frac{\sum_{j=0}^{N-1} x_j \cdot \cos\left(\frac{i \cdot \pi \cdot (2j+1)}{2N}\right)}{w_i \cdot w_i}$$

dove: (con w_i vettore base della DCT)

- se $i = 0$ allora $w_i \cdot w_i = N$ (che possiamo trasformare quindi in moltiplicazione per $a_{i=0} = \frac{1}{N}$)
- se $i > 0$ allora $w_i \cdot w_i = \frac{N}{2}$ (che possiamo trasformare quindi in moltiplicazione per $a_{i>0} = \frac{2}{N}$)

A questi si sceglie di aggiungere la radice quadrata per fare normalizzazione, senza normalizzazione il prodotto scalare di un vettore base w_i con sé stesso sarebbe proporzionale a N . La radice quadrata viene utilizzata per compensare questa dipendenza.

Ma se andiamo ad esplorare la [documentazione di SciPy](#) (sotto `dct-type2` e `norm=ortho`) possiamo vedere che questa implementa il calcolo dei coefficienti di normalizzazione in maniera leggermente diversa, ma molto simile, a quella che abbiamo deciso di applicare:

- se $k = 0$ allora $w_i \cdot w_i = \sqrt{4N}$ (trasformabile quindi in moltiplicazione per $a_{i=0} = \frac{1}{\sqrt{4N}}$)
- se $k > 0$ allora $w_i \cdot w_i = \sqrt{\frac{N}{2}}$ (trasformabile quindi in moltiplicazione per $a_{i>0} = \sqrt{\frac{2}{N}}$)

1.7. Analisi dei risultati e conclusioni parte-1:

Una volta definiti tutti i metodi utilizzati per l'implementazione di DCT2 possiamo dunque eseguirli.

1.7.1. Esecuzione test per i requisiti tecnici forniti:

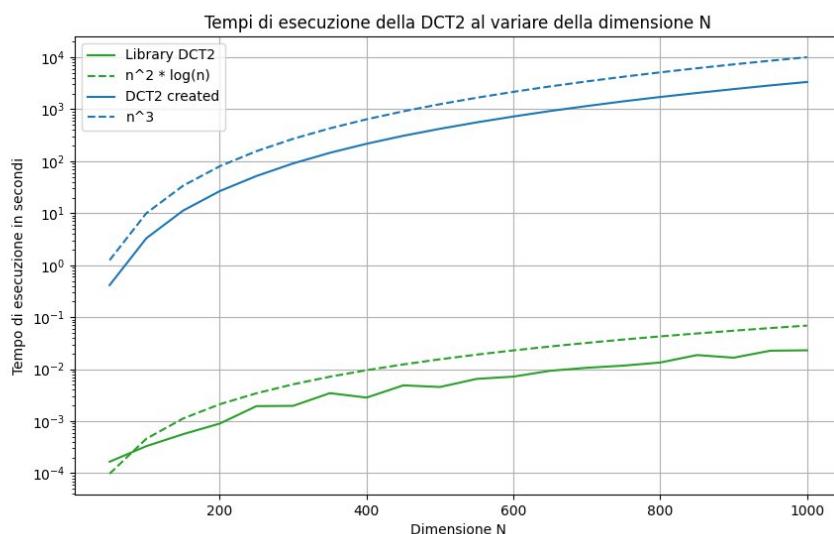
Abbiamo precedentemente mostrato nella sezione [1.3](#) la funzione `run_test()` (all'interno del file `test.py`). Questa funzione è incaricata di controllare il rispetto dei requisiti tecnici forniti dalla consegna. Abbiamo infatti che per ogni test eseguito viene scritta una riga di codice all'interno del file “`test_results.txt`”, di seguito ne vengono mostrati i risultati:

```
test_results.txt
1 DCT HomeMade test passed!
2 DCT2 HomeMade test passed!
3 DCT library test passed!
4 DCT2 library test passed!
5 TESTS COMPLETED
```

Come possibile visualizzare tutti i test eseguiti vengono eseguiti correttamente (nonostante la piccola differenza di fattore di scala precedentemente notata). Di conseguenza possiamo affermare con certezza che i risultati prodotti rispettino i requisiti di accuratezza richiesti da consegna.

1.7.2. Esecuzione test per verificare i tempi di esecuzione:

Sempre nel capitolo [1.3](#) abbiamo precedentemente mostrato il funzionamento della funzione `test_N` che permette di misurare il tempo necessario per eseguire computazioni di DCT2 (sia la versione scritta manualmente che quella di libreria) su matrici di grandezze progressivamente più grandi (da $N = 50$ ad $N = 1000$). Avevamo anche accennato poi alla presenza di un metodo presente nel file `generate_graphs.py` che permette di prendere i risultati dei tempi di computazione calcolati con il metodo precedente e mostrarli in un grafico. In seguito all'esecuzione di entrambi i metodi questi sono i risultati riportati:



Analizzando i risultati riportati, si osserva che il tempo di esecuzione delle implementazioni della DCT segue, segue un andamento molto simile a quello di $O(N^3)$; in maniera allineata a ciò che avevamo previsto nei capitoli precedenti. In particolare, l'utilizzo del metodo di libreria consente un notevole miglioramento delle prestazioni per le matrici nella DCT2. Infatti, come anticipato, questo metodo segue un andamento di $O(N^2 \cdot \log N)$.

Possiamo quindi concludere che per matrici di piccole dimensioni ($N \leq 200$), i tempi di esecuzione delle due implementazioni sono simili: essendo le y in scala logaritmica abbiamo che per entrambi i metodi è necessario meno di un secondo (circa) per calcolare entrambi i metodi. Ovviamente però diventa necessario utilizzare il metodo di libreria per calcolare matrici di dimensioni $N > 200$ in quanto la differenza tra i due metodi inizia a diventare molto più accennata.

Siamo dunque soddisfatti dei risultati ottenuti sia sui test precedenti che su quelli misurazione delle tempistiche di esecuzione, in quanto, in entrambi i casi siamo riusciti ad arrivare alle nostre aspettative iniziali.

2. Parte-2: Software con interfaccia grafica

2.1. Requisiti tecnici:

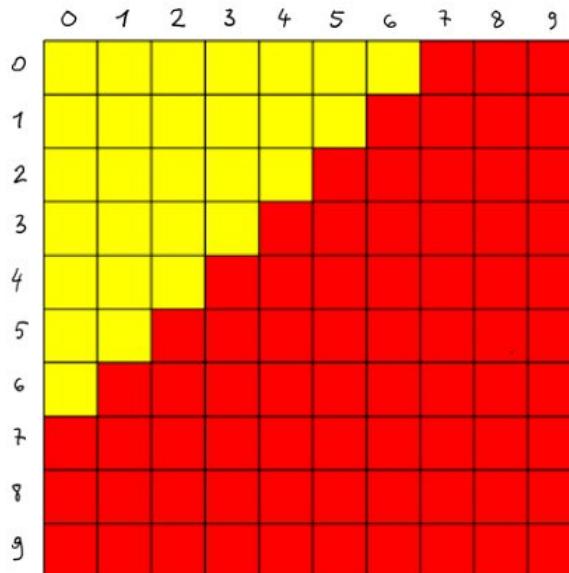
2.1.1: Come deve funzionare l'applicazione:

Creare un software con interfaccia grafica che permetta di suddividere l'immagine in blocchi quadrati f di pixel di dimensioni $F \times F$ partendo in alto a sinistra, scartando gli avanzi; Ossia, per ogni blocco f eseguire le seguenti operazioni:

- applicare la DCT2 (della libreria): $c = DCT2(f)$;
- eliminare le frequenze $c_{k\ell}$ con $k + \ell \geq d$: assumendo che le frequenze partano da 0: se $d = 0$ le elimino tutte, se $d = (2F - 2)$ elimino solo la più alta, cioè quella con:

$$k = F - 1, \ell = F - 1$$

In sostanza bisogna eliminare i coefficienti in frequenza a destra della diagonale individuata dall'intero d , come esemplificato qui sotto:



Blocco 10×10 con $d = 7$

Qui abbiamo $F = 10$ e $d = 7$. I coefficienti da eliminare sono indicati in rosso.

- applicare la DCT2 inversa all'array c così modificato: $ff = IDCT2(c)$
- arrotondare ff all'intero più vicino, mettere a zero i valori negativi a 0 e si troncano i valori maggiori di 255 (limitandoli a 255) in modo da avere dei valori ammissibili (1 byte);

Una volta fatto questo per ogni blocco f si rimette insieme l'immagine mettendo assieme i blocchi ff ottenuti, mettendoli nell'ordine giusto e paragonare l'immagine ottenuta con l'originale di partenza.

2.1.2: Requisiti sui parametri in input:

Una parte fondamentale per lo sviluppo di questa applicazione è la gestione dell'input dell'utente. Più nello specifico è fondamentale garantire che:

- L'utente possa prendere in input solo immagini in formato .bmp
- Il parametro F sia un numero intero tale che $0 < F < \text{"Base/Altezza_immagine caricata"}$
- Il parametro d sia un numero intero tale che $0 \leq d \leq 2 \cdot F - 2$

2.2. Le basi per lo sviluppo: cosa ci aspettiamo:

2.2.1: Aspettative a livello qualitativo dell'immagine:

I parametri F e d sono fondamentali per fare delle previsioni sui risultati di compressione della nostra applicazione, abbiamo infatti che:

- Per F : all'aumentare il valore di F più è possibile che l'immagine venga distorta, questo perché (indicando F la dimensione dei blocchetti dell'immagine che verranno compressi), quindi (potenzialmente) se si sceglie una regione $F \times F$ molto grande e si eliminano molti coefficienti in relazione alla dimensione $F \times F$ si rischia di rovinare la visibilità dell'immagine e di creare grandi artifact di diverso tipo:
 - Artifact di forma quadrata all'interno dell'immagine compressa risultante (che rappresentano i bordi della finestra $F \times F$), oppure anche creare altri disturbi in cui compare una serie di "discontinuità" nell'immagine ai bordi delle finestre $F \times F$.
 - 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.
- Per d : si può facilmente intuire che questo valore è inversamente proporzionale alla qualità dell'immagine risultante. Al diminuire del valore di d più ci si può aspettare che la qualità dell'immagine peggiori, in quanto:
 - utilizzando un alto valore di d ci si aspetta di mantenere la maggior parte dei coefficienti di tutti i blocchetti $F \times F$, il che vuol dire che la compressione rimuove solo una piccola parte di informazione dell'immagine
 - Utilizzando un basso valore di d ci si aspetta, invece, di eliminare la maggior parte dei coefficienti presenti nel blocchetto $F \times F$ portando ad avere un'immagine molto compressa, ma anche con una qualità visiva peggiore (in quanto una gran quantità del valore informativo dell'immagine è stato rimosso)

La chiave per una buona compressione sarà dunque trovare un giusto compromesso per il valore di soglia d , in modo da non ridurre eccessivamente la qualità dell'immagine, riuscendo contemporaneamente a utilizzare meno memoria.

2.2.2: Aspettative a livello di tempi di computazione:

Come abbiamo precedentemente mostrato la procedura di applicazione della DCT2 (e di conseguenza anche quella di IDCT2) è un operazione $O(N^3)$. Se consideriamo N come la dimensione di un'immagine (che solitamente è dell'ordine del migliaia di pixel), avremo che il nostro metodo di compressione "scala male", ossia potrebbe arrivare ad avere tempi di computazione molto alti. Anche qualora utilizzassimo la funzione di libreria che fa' utilizzo di FFT (cosa che faremo), il costo rimane elevato: $O(N^2 \cdot \log(N))$.

Possiamo dunque aspettarci che i tempi di computazione di immagini molto grandi fornite al nostro programma possano diventare altrettanto molto alti.

NOTA TEORICA: La compressione Jpeg

Abbiamo appena elencato tutta una serie di problemi che ci possiamo aspettare di avere quando implementiamo il nostro algoritmo di compressione delle immagini. Abbiamo però che, nonostante il metodo di compressione Jpeg sia molto simile a quello descritto nella sezione di [requisiti tecnici](#), si ha che l'algoritmo di compressione Jpeg riesce comunque a migliorare la compressione delle immagini applicando delle migliorie ad alcuni step:

- Per risolvere il problema di avere computazioni che richiedono alto tempo di computazione usa finestre di dimensione fissa 8×8 , la loro piccola dimensione le rende più facili da calcolare, questo aiuta anche a contrastare il fenomeno di Gibbs.
- Per mitigare la creazione di discontinuità lungo i bordi delle finestre 8×8 Jpeg usa finestre con sovrapposizione. Ossia alcuni pixel contenuti nella finestra 8×8 precedente sono contenuti anche nella finestra 8×8 successiva. In questo modo, dopo che si è fatta l'elaborazione di entrambi i blocchi si possono sovrapporre.

2.3. Le basi per lo sviluppo: come è strutturato il codice:

Il codice di quest'applicazione è strutturato in 2 file:

- MainUI.py
- ProcessImages.py

Dove abbiamo che:

MainUI.py: Classe che contiene le funzioni di GUI (frontend) per gestire la grafica del programma ed il loro collegamento con il backend. In particolare l'interfaccia è stata sviluppata utilizzando la libreria tkinter, che offre una gamma di widjet predefiniti come finestre, bottoni, caselle di testo, etichette e altro, permettendo di creare e personalizzare l'aspetto dell'applicazione. Più nello specifico abbiamo che in questo file è presente una classe "App" al quale al suo interno è descritto il metodo **CreateWidgets** dedicato alla creazione della GUI dell'applicazione.

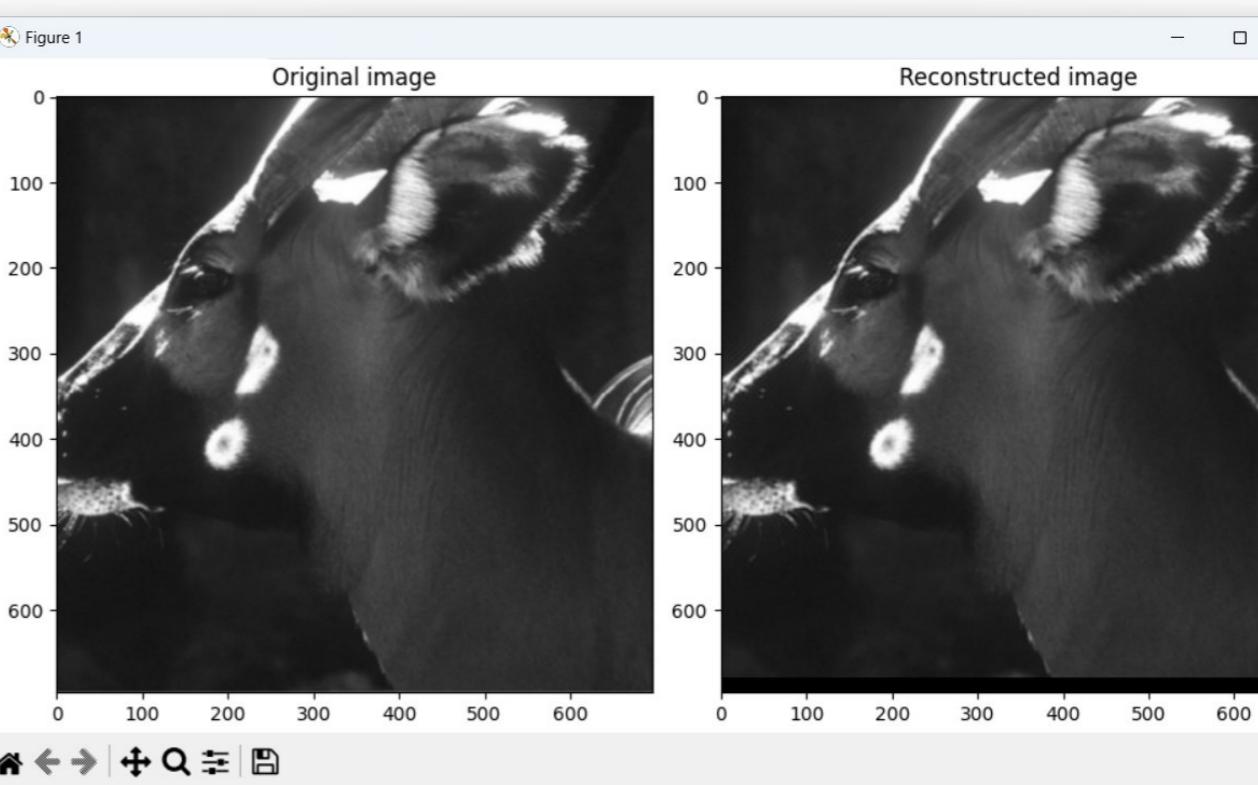
Abbiamo poi che sempre all'interno di questo file è presente il metodo main. All'avvio dell'applicazione questo metodo crea un'istanza della classe App, il quale costruttore avvia il metodo **CreateWidgets** permettendo così di visualizzare la grafica.

A pagina seguente è presentata una breve overview del codice appena descritto e la corrispondente grafica generata tramite applicazione:

```

9  class App:
10     def __init__(self, root):
11         self.root = root
12         self.root.title("COMPRESSEIONE DI IMMAGINI TRAMITE DCT2")
13         self.root.geometry("800x780") # Dimensione della finestra
14
15         self.image_path = None # Variabile per memorizzare il percorso dell'immagine
16         self.max_F = None # Variabile per memorizzare la grandezza massima di F
17
18         self.create_widgets()
19
20     >     def create_widgets(self): ...
21
22         # apre file dialog per selezionare un'immagine .bmp
23         def choose_image(self): ...
24
25         # Calcola la grandezza massima di F data immagine selezionata
26         def calculate_max_F(self, image_path): ...
27
28         # Mostra l'immagine caricata nell'interfaccia utente
29         def display_image(self, image_path, label): ...
30
31         # Verifica gli input "F" e "d" ed avvia la compressione
32         def process_inputs(self): ...
33
34             # Esegui la compressione dell'immagine
35             def run_compression(self, image_path, F, soglia): ...
36
37             # Visualizza la matrice di compressione
38             def visualize_matrix(self): ...
39
40             # Print di una stringa sul LOG
41             def log_message(self,message): ...
42
43
44     >     if __name__ == "__main__":
45         root = tk.Tk()
46         app = App(root)
47         root.mainloop()

```



Nota che nel file `MainUI.py` abbiamo anche i controlli necessari per verificare le corrette dimensioni di d ed F . Abbiamo infatti che, al momento di caricamento di un'immagine col il metodo `choose_image` (oltre all'avvio del metodo `display_image` necessario per mostrare nella UI l'immagine appena selezionata da file explorer) viene avviato anche il metodo `calculate_max_F`; questo metodo, serve per salvare la dimensione minima tra la base e l'altezza dell'immagine appena caricata (che corrisponde al valore massimo che potrà assumere il valore F). Abbiamo poi che quando viene premuto il bottone “Esegui compressione” viene avviato il metodo `process_inputs` descritto di seguito:

```
# Verifica gli input "F" e "d" ed avvia la compressione
def process_inputs(self):
    try:
        # Recupero la grandezza delle finestrelle
        F = int(self.entry_F.get())
        # Controllo che la grandezza delle finestrelle sia valida
        if not self.max_F:
            raise ValueError("Per favore seleziona un'immagine prima di procedere")
        # Controllo che un'immagine sia stata selezionata
        if not self.image_path:
            raise ValueError("Per favore seleziona un'immagine .bmp prima di procedere") } 1

        # Controllo che la grandezza delle finestrelle sia compresa tra 1 e max_F
        if F <= 0 or F > self.max_F:
            raise ValueError(f"La grandezza delle finestrelle deve essere compresa tra 1 e {self.max_F}") } 2

        # Recupero la soglia di compressione (d)
        d = int(self.entry_threshold.get())
        # Controllo che la soglia sia compresa tra 1 e 2F-2
        if not (0 < d <= (2 * F - 2)):
            raise ValueError(f"La soglia di compressione deve essere compresa tra 1 e {2 * F - 2}") } 3

        # Visualizza la matrice di compressione } 4
        self.visualize_matrix()

        # Esegui la compressione in un thread separato
        threading.Thread(target=self.run_compression, args=(self.image_path, F, d)).start() } 5
    except ValueError as e:
        messagebox.showerror("Errore di input", str(e))
```

Abbiamo dunque che nel codice appena mostrato vengono fatti rispettare i vincoli imposti sui parametri (segnalati al capitolo [2.1.2](#)):

1. Si verifica che sia stata caricata un'immagine a priori.
2. Si verifica che il parametro F sia un numero intero tale che:

$$0 < F < \text{Base}/\text{Altezza_immagine_caricata}$$
3. Si verifica che il parametro d sia un numero intero tale che:

$$0 \leq d \leq 2 \cdot F - 2$$

Solo dopo aver fatto rispettare i vincoli sui parametri appena descritti allora:

4. Viene fatta visualizzare la matrice di compressione dinamicamente (le dimensioni e la diagonale selezionata cambiano sempre in base ai parametri di F e d inseriti) grazie al metodo `visualize_matrix`.
5. Avvia il metodo `run_compression` in un thread separato (in questo modo non viene bloccata la grafica dell'applicazione), questo servirà finalmente ad avviare la compressione dell'immagine.

ProcessImages.py: All'interno di questo file sono presenti tutti i metodi che permettono di realizzare la compressione dell'immagine. Abbiamo infatti appena visualizzato che il primo metodo avviato per iniziare il calcolo della compressione è `run_compression`, la definizione di questo metodo è infatti presente in questo file e, come vedremo di seguito, questo permette di applicare la compressione esattamente come descritta al capitolo [2.1.1](#):

```
def run_compression(image_path, block_size, d_threshold, logger):
    try:
        blocks = generate_blocks(image_path, block_size) } 1
        total_blocks = len(blocks)

        blocks_processed_dct = apply_dct2(blocks, d_threshold, block_size, logger, total_blocks) } 2
        blocks_idct_rounded = apply_idct2(blocks_processed_dct, logger, total_blocks) }

        save_compressed_image(blocks_idct_rounded, image_path, block_size) } 3
    except Exception as e:
        print("Error during compression:", str(e)) } 4
```

Abbiamo infatti che:

1. Tramite il metodo `generate_blocks` si suddivide l'immagine in blocchi quadrati (f) di dimensione $F \times F$
2. Tramite il metodo `apply_dct2` si applica la dct2 ad ogni blocchetto f e si eliminano le frequenze al di sotto della diagonale d
3. Tramite il metodo `apply_idct2` si applica la trasformazione inversa dei blocchetti f
4. Infine il metodo `save_compressed_image` rimette insieme i blocchetti f nella loro posizione originale (in modo da ricostruire l'immagine), salva l'immagine in memoria e mostra il risultato di compressione affiancato all'immagine originale

Nelle sezioni seguenti approfondiremo il codice dietro ciascuno dei metodi appena citati per dimostrare che la compressione avvenga correttamente.

2.4. ProcessImages.py: generate_blocks:

Funzione dedicata alla suddivisione dell'immagine in blocchi quadrati (f) di dimensione $F \times F$, di seguito se ne riporta il codice:

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

            # Converti in bianco e nero se necessario
            img_gray = img.convert('L') if img.mode != 'L' else img

            # calcola il numero di blocchi orizzontali e verticali
            num_blocks_horizontal = img_width // block_size
            num_blocks_vertical = img_height // block_size

            blocks = []

            # estrai blocchi dall'immagine
            for j in range(num_blocks_vertical):
                for i in range(num_blocks_horizontal):
                    # calcola le coordinate del blocco:
                    # x0 e y0 alto a sx - x1 e y1 basso a dx
                    x0 = i * block_size
                    y0 = j * block_size
                    x1 = x0 + block_size
                    y1 = y0 + block_size

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

    return blocks

except Exception as e:
    print("Error during image processing:", str(e))
    return []
```

Calcolo numero di blocchi per ogni riga e per ogni colonna dell'immagine originale

Calcolo delle coordinate di ogni blocco di dimensione $F \times F$ da estrarre dall'immagine. Subseguentemente si preleva (crop) dall'immagine quell'esatto blocchetto f . Infine si aggiunge il blocchetto ad un array (che conterrà alla fine del ciclo tutti i blocchi), da ritornare in output

Una volta ritornato la lista di “blocks” riprende l'esecuzione del metodo `run_compression` che, come prossima istruzione applicherà la trasformazione DCT2 tramite il metodo `apply_dct`

NOTA: Crop dell'immagine

Siccome la divisione che effettuiamo per andare calcolare il numero di blocchi per riga e per colonna dall'immagine è una divisione senza resto ci si può aspettare che se non si seleziona una dimensione di F che sia un sottomultiplo del numero di pixel della base/altezza dell'immagine allora questa verrà tagliata alla base/altezza.

2.5. ProcessImages.py: apply_dct2:

In seguito all'aver ottenuto tutti i blocchetti f che compongono l'immagine si applica la DCT2 a ciascun blocchetto f , inoltre, per ogni blocchetto si eliminaranno le frequenze al di sotto della diagonale d come riportato di seguito:

```
def apply_dct2(blocks, d_threshold, F_size, logger, total_blocks):
    blocks_processed_threshold = []
    for idx, block in enumerate(blocks):
        block_array = np.array(block)
        block_dct = dct(dct(block_array.T, norm='ortho').T, norm='ortho')

        # Maschera di quantizzazione: crea matrice di booleani
        # con valore true solo nelle posizioni < del parametro di threshold
        mask = np.abs(np.add.outer(range(F_size), range(F_size))) < d_threshold

        # Operazione che permette di mantenere solo i valori che corrispondono
        # agli indici degli elementi della matrice
        # di quantizzazione impostati a true
        block_processed_threshold = block_dct * mask

        blocks_processed_threshold.append(block_processed_threshold)

        log_message(logger, "✓ DCT2 " + str(idx + 1) + "/" + str(total_blocks) + " processed")

    return blocks_processed_threshold
```

Abbiamo infatti che, nel codice riportato, per ogni blocco f ricavato dallo step precedente:

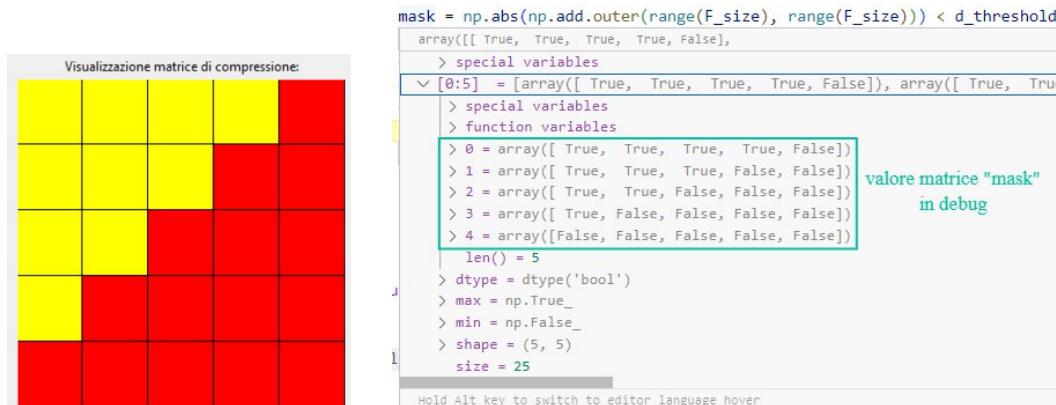
1. si calcola la DCT2 usando la libreria Scipy (esattamente come fatto nel [progetto-1](#)). In questo modo otteniamo:

$$c = DCT2(f)$$

2. Una volta calcolata c (nel codice rappresentata dalla variabile "block_dct"), si troncano le frequenze al di sopra della diagonale d , ossia si eliminano le frequenze $c_{k\ell}$ con $k + \ell \geq d$, per fare questo si usa la riga di codice:

```
mask = np.abs(np.add.outer(range(F_size), range(F_size))) < d_threshold
```

Questa permette di creare una matrice di booleani la quale posizione corrisponde perfettamente con le posizioni $c_{k\ell}$ con $k + \ell \geq d$, di seguito un esempio con $F = 5$ e $d = 4$:



In python il valore False equivale a 0 ed il valore True equivale ad 1, il che vuol dire che moltiplicare questa matrice "Mask" per il blocchetto DCT2 (c) otteniamo il troncamento delle frequenze desiderato.

2.6. ProcessImages.py: apply_idct2:

NOTA: paginazione della relazione

Prima di passare alla visualizzazione del metodo che implementa IDCT2 se ne da una breve descrizione teorica, in quanto, a differenza del metodo di DCT2, non abbiamo ancora fornito una spiegazione teorica di questo metodo all'interno di questa relazione.

Come abbiamo già accennato IDCT2 è l'operazione inversa di DCT2, questa permette infatti di prendere un segnale 2 dimensionale nel dominio delle frequenze e riconvertirlo al dominio del tempo/spazio. Di conseguenza abbiamo che IDCT (1D) è l'operazione inversa di DCT. Si ha infatti che IDCT viene definita dalla seguente formula:

$$x_i = \sum_{j=0}^{N-1} X_j \cdot \alpha_j \cdot \cos\left(\frac{i \cdot \pi \cdot (2j + 1)}{2N}\right)$$

dove:

- X_j è il coefficiente DCT corrispondente alla frequenza j ,
- α_j è lo stesso identico fattore di normalizzazione già usato nella DCT,
- $\cos(...)$ è la funzione che combina le componenti di frequenza per ricostruire il segnale.
- x_i è il valore originale ricostruito nel dominio del tempo/spaziale

Applicare quindi l'IDCT a un segnale trasformato con la DCT restituisce il segnale originale:

$$\text{IDCT}(\text{DCT}(x)) = x$$

Abbiamo dunque che (analogamente a DCT2) IDCT2 bidimensionale è semplicemente l'applicazione sequenziale della IDCT prima sulle colonne e poi sulle righe della matrice dei coefficienti calcolati con DCT2. La formula generale per l'IDCT2 è:

$$f(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \alpha_i \cdot \alpha_j \cdot X_{i,j} \cdot \cos\left(\frac{i \cdot \pi \cdot (2x + 1)}{2N}\right) \cdot \cos\left(\frac{j \cdot \pi \cdot (2y + 1)}{2M}\right)$$

dove:

- $X_{i,j}$ è il coefficiente DCT2 corrispondente alla frequenza i,j
- α_i insieme ad α_j sono i 2 fattori di normalizzazione per le rispettive DCT.
- N e M sono le dimensioni della matrice di input.
- $f(x,y)$ è il valore dell'elemento della matrice di input.

Abbiamo dunque che, analogamente all'uso di IDCT su DCT, anche applicare IDCT2 sui coefficienti calcolati con DCT2 permette di ritornare al segnale originale:

$$\text{IDCT2}(\text{DCT2}(f(x,y))) = f(x,y)$$

Questo principio verrà utilizzato nel codice che mostreremo a pagina seguente.

Una volta descritti i principi teorici dietro l'applicazione di IDCT2 possiamo continuare con la descrizione del funzionamento del codice dell'applicazione, arrivando ora a descrivere il seguente metodo il quale scopo è quello di applicare IDCT2 a tutti i blocchetti c precedentemente processati:

```
def apply_idct2(blocks_dct_quantized, logger, total_blocks):
    blocks_idct_rounded = []
    for idx, block_dct_quantized in enumerate(blocks_dct_quantized):
        # Applica l'IDCT (Inverse Discrete Cosine Transform) al blocco quantizzato
        # La trasformazione è eseguita prima sulle colonne e poi sulle righe
        block_idct = idct(idct(block_dct_quantized.T, norm='ortho').T, norm='ortho') } 1

        # Arrotonda i valori del blocco IDCT ai numeri interi più vicini
        block_idct_rounded = np.round(block_idct)
        # Imposta i valori negativi a 0 e i valori superiori a 255
        # a 255 per evitare valori di pixel non validi
        block_idct_rounded[block_idct_rounded < 0] = 0
        block_idct_rounded[block_idct_rounded > 255] = 255 } 2

        # Aggiunge il blocco IDCT arrotondato alla lista,
        # convertendolo in tipo uint8
        blocks_idct_rounded.append(block_idct_rounded.astype(np.uint8)) } 3

    log_message(logger, "✓ IDCT2 " + str(idx + 1) + "/" + str(total_blocks) + " processed")

    return blocks_idct_rounded
```

Si ha dunque che nel codice appena mostrato succede che:

1. Si applica la IDCT2 inversa ai blocchetti c così in modo da ottenere: $ff = IDCT2(c)$
2. Si arrotondano tutti i valori di ff all'intero più vicino e mettendo a zero i valori negativi a 0. Poi si troncano i valori maggiori di 255, limitandoli appunto a 255.
3. Tutti i blocchetti ff appena processati vengono collezionati in una lista e ritornati in output.

In questo modo, applicando la IDCT2 ed in seguito lo scaling dei valori da 0 a 255, in ogni blocchetto ff avremo dei valori ammissibili (1 byte) per ricostruire l'immagine; in questo modo vengono rispettati i vincoli tecnici relativi all'applicazione della DCT2 specificati al capitolo [2.1.1](#).

2.7. ProcessImages.py: save_compressed_image:

Di seguito viene illustrata la funzione dedicata alla ricomposizione dell'immagine partendo da blocchi quadrati (ff) di dimensione $F \times F$. In seguito alla ricomposizione viene mostrata l'immagine originale affiancata a quella compressa e salvata l'immagine compressa in memoria. Più nello specifico si ha:

```
def save_compressed_image(blocks_idct_rounded, original_image_path, block_size):
    compressed_image_filename = "compressed_image.bmp"

    try:
        with Image.open(original_image_path) as img:
            img_width, img_height = img.size
            compressed_image = Image.new('L', (img_width, img_height)) } }

        # Calcolo del numero di blocchi orizzontali
        num_blocks_horizontal = img_width // block_size
        num_blocks_vertical = img_height // block_size
        # Ciclo attraverso i blocchi verticali dell'immagine
        for j in range(num_blocks_vertical):
            for i in range(num_blocks_horizontal):
                # Calcolo delle coordinate del blocco
                x0 = i * block_size
                y0 = j * block_size
                x1 = x0 + block_size
                y1 = y0 + block_size

                # Estrazione del prossimo blocco IDCT arrotondato dalla lista
                block = blocks_idct_rounded.pop(0)
                # Inserimento del blocco nella posizione
                # corretta nell'immagine compressa
                compressed_image.paste(Image.fromarray(block), (x0, y0))

        compressed_image.save(compressed_image_filename)

    def display_images(img, compressed_image): ...

        display_thread = threading.Thread(target=display_images, args=(np.array(img), np.array(compressed_image)))
        display_thread.start()
        #display_thread.join()

        compressed_image.close()
        print("Compressed image saved successfully.")

    except Exception as e:
        print("Error during saving compressed image:", str(e)) } }

    Salvataggio della nuova immagine nella memoria di sistema. Poi si utilizza la funzione display_images che permette di visualizzare in un plot l'immagine originale e l'immagine processata affiancate
```

NOTA IMPLEMENTATIVA: non bloccare la grafica

Per non bloccare la grafica dell'applicazione si mostra avvia la funzione `display_images` in un thread separato dal main_thread. Questo permette all'applicazione di poter continuare la sua esecuzione in maniera smooth, senza che la grafica venga interrotta dalla creazione della nuova finestra che si crea per mostrare il paragone delle 2 immagini.

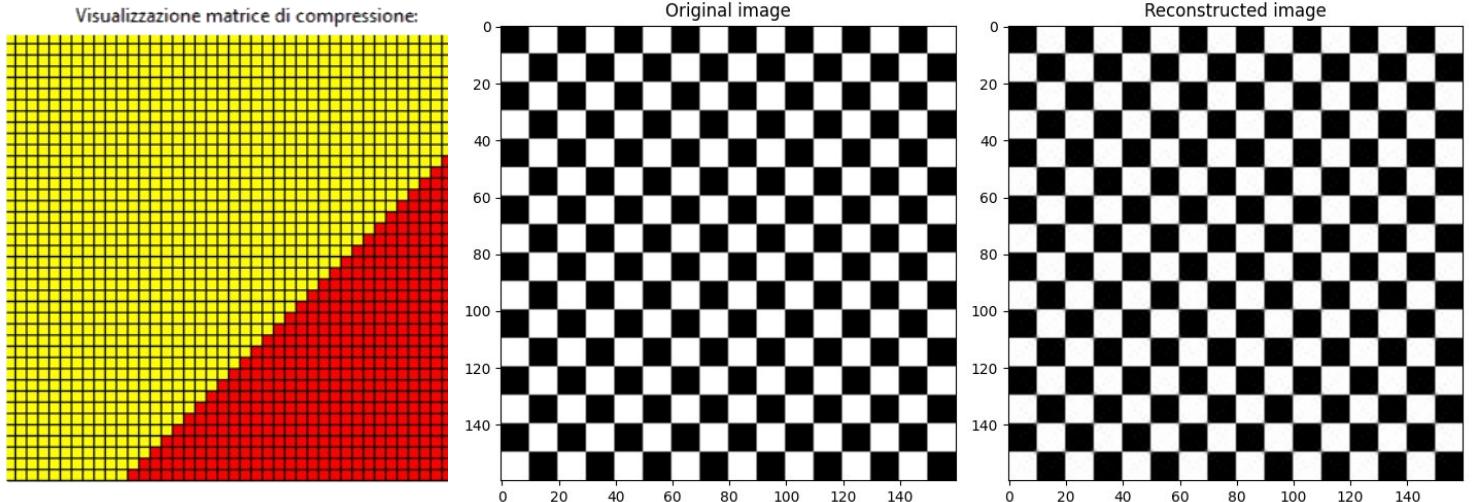
In questo modo siamo riusciti a descrivere completamente il funzionamento dell'applicazione spiegandone come avviene la compressione. Possiamo dunque ora passare all'analisi dei risultati ottenuti dalle elaborazioni di questa app.

2.8. Analisi dei risultati prodotti dall'app:

2.8.1. Immagine 1:

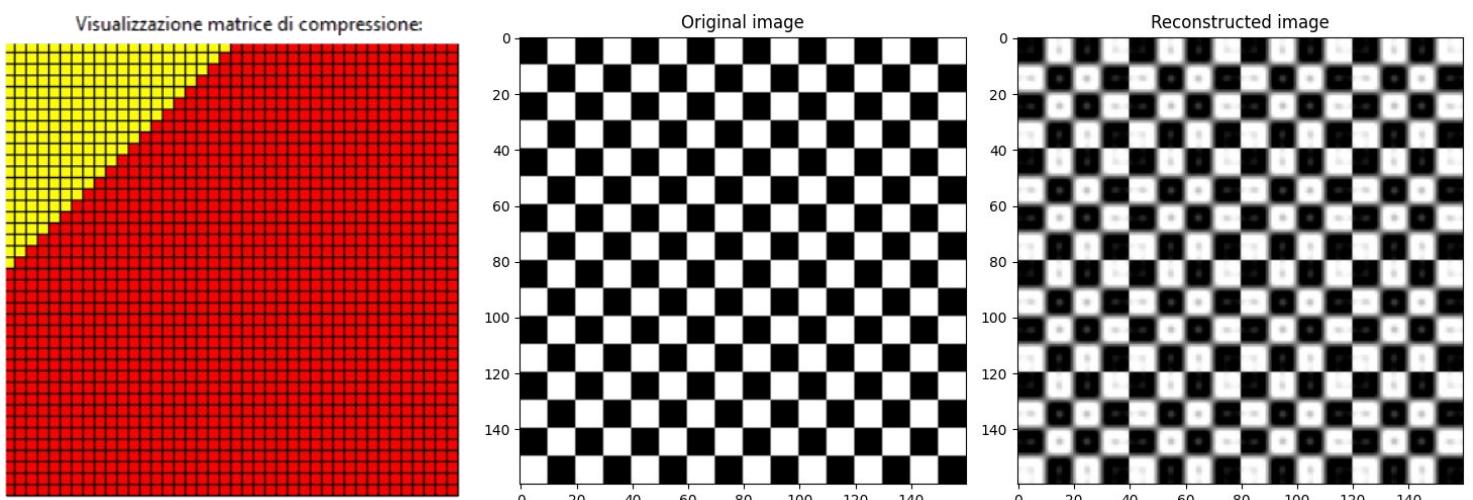
```
Selected image: D:/Repos/python/Scientific_Calculus_2/parte2/Immagini/160x160.bmp
Original image size: 26.052734375 KB
Original image shape: (160, 160)
```

Compressione con $F = 40$ e $d = 50$ (esempio di buona compressione)



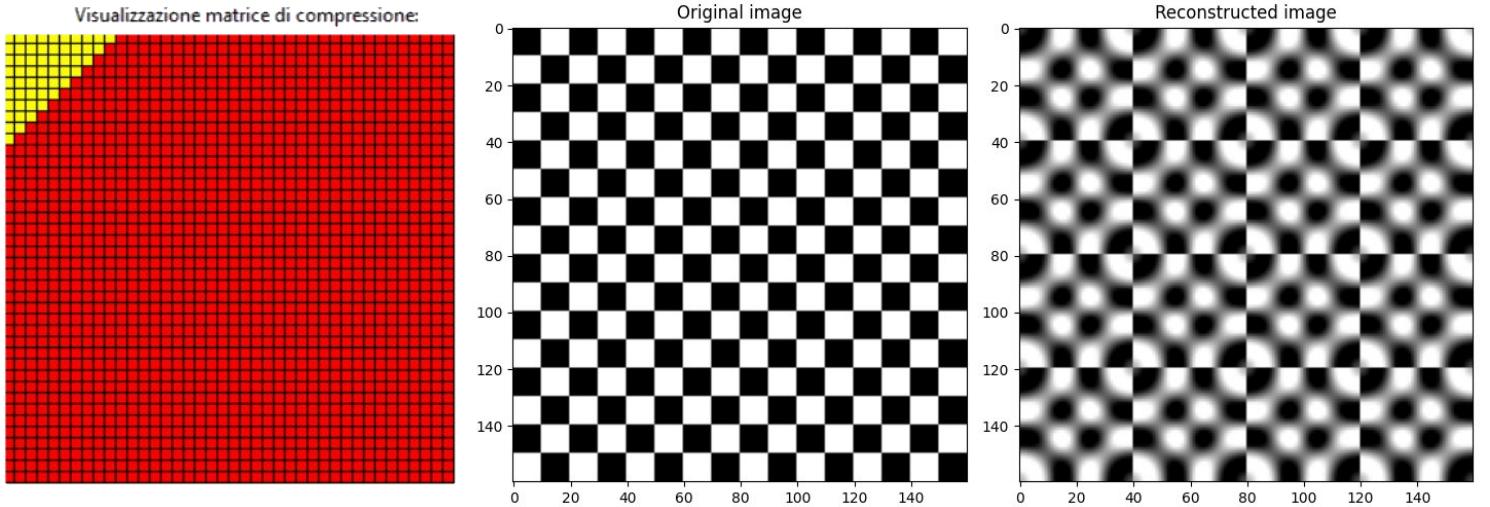
Come è possibile visualizzare, scegliendo una dimensione di F che è sottomultiplo della dimensione di base ed altezza dell'altezza originale dell'immagine (40 è sottomultiplo di 160) non si verifica alcun crop sull'immagine risultante, e come ci si poteva aspettare, mantenendo un livello relativamente alto di d rispetto al valore massimo che si potrebbe assumere (in questo caso 78) si riesce a mantenere una qualità dell'immagine relativamente alta, anche se zoomando sull'immagine compressa è possibile iniziare ad intravedere dei piccoli artefatti.

Compressione con $F = 40$ e $d = 20$ (esempio di troppa compressione)



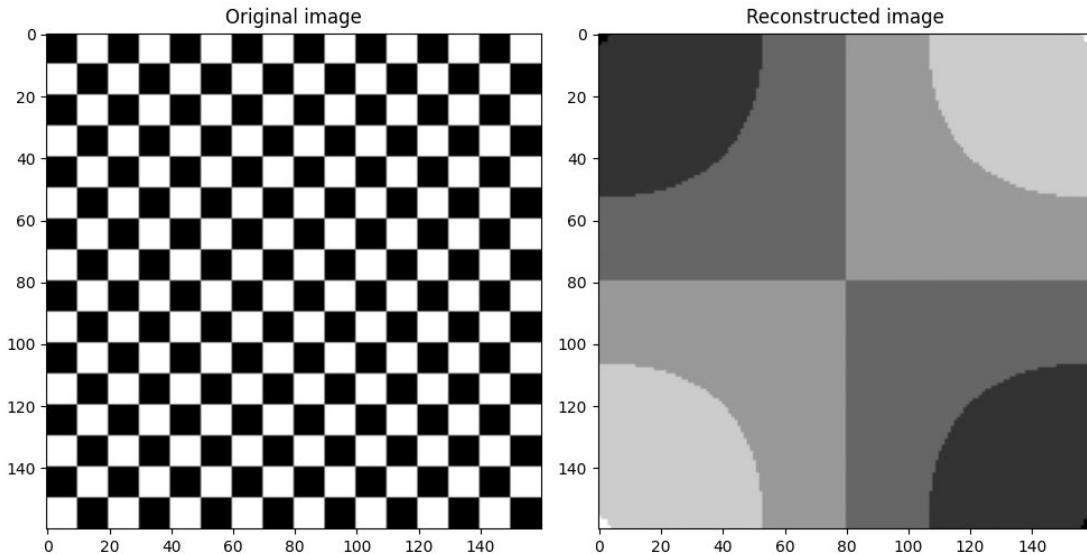
Se iniziamo ad aumentare la soglia di coefficienti da tagliare (diminuendo d) come in questo caso, è possibile vedere che la presenza degli artefatti diventa molto più accentuata, si hanno infatti che oltre alle macchie grigie presenti all'interno dei quadrati bianchi inizia anche a diventare evidente la discontinuità sul bordo delle finestre, dove, in questo caso, si vanno a formare degli artifact di forma quadrata che rispecchiano il bordo delle finestre f usate per la compressione. La qualità dell'immagine risultante risulta comunque buona, anche se distorta, ma almeno il contenuto informativo dell'immagine originale (la scacchiera) è individuabile anche nella seconda immagine.

Compressione con $F = 40$ e $d = 10$ (esempio di troppa compressione)



Se vogliamo spingere l'esempio precedente ad avere un risultato ancora più estremo, possiamo rendere ancora più evidente la discontinuità sul bordo della finestra f diminuendo ancora di più il coefficiente d , in questo caso oltre alle discontinuità che si formano sui bordi delle finestre f possiamo anche osservare che quelli che prima erano i quadratini che componevano la scacchiera diventare pesantemente blurrati. Questo effetto è dovuto alla grossa rimozione delle alte frequenze, che contengono le informazioni sui dettagli netti dell'immagine. L'effetto osservato all'interno dei blocchi è dunque causato dalla somma di poche componenti sinusoidali dominanti, che generano il tipico aspetto sfocato e ondulato.

Compressione con $F = 160$ e $d = 3$ (esempio di perdita tutto valore informativo)



In questo caso, per dimostrare quanto disastrosi possano essere gli effetti della distorsione sull'immagine originale, si sceglie di utilizzare un'unica finestra f di dimensione uguale alla dimensione originale dell'immagine. Avremo quindi una sola finestra molto grande. Si sceglie inoltre di usare un coefficiente d molto basso. In questo modo possiamo osservare che la compressione ha rimosso quasi tutte le informazioni di dettaglio dell'immagine. Poiché sono state mantenute solo poche frequenze ($d = 3$). Abbiamo infatti che la ricostruzione è composta solo da una combinazione di pochissime sinusoidi a bassa frequenza. Questo ci porta a perdere completamente il contenuto dell'immagine originale.

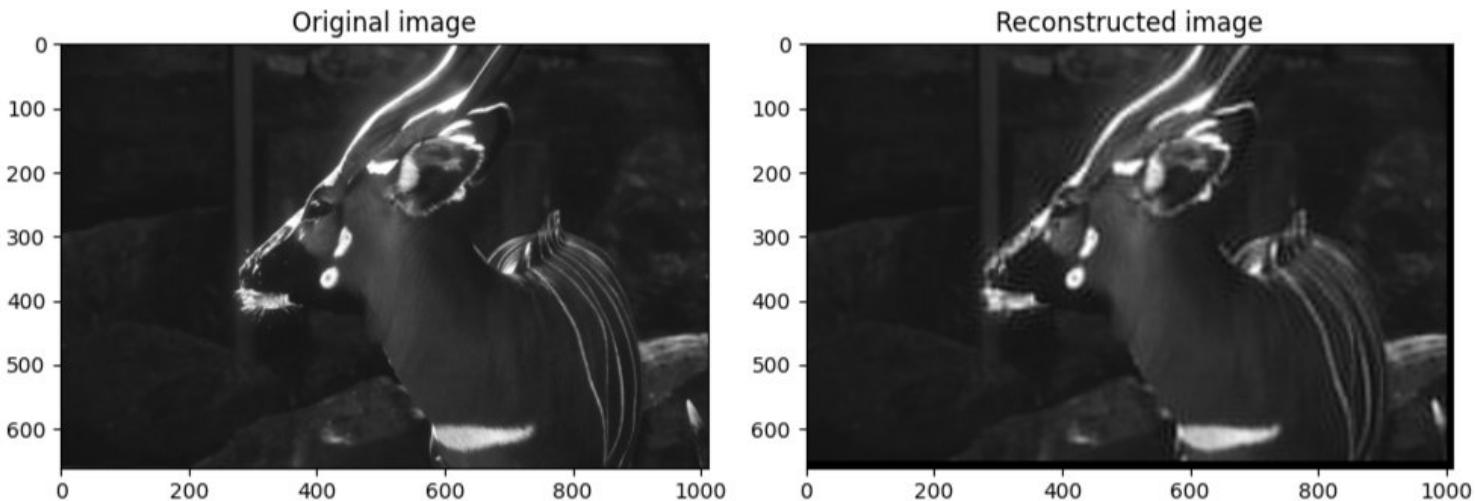
NOTA SULLA RIPETIZIONE: matrice di compressione

Per mostrare più nel dettaglio (ossia più grandi) le immagini relative alle prossime elaborazioni (e anche per evitare ripetizioni) non verrà più illustrata la matrice di compressione nei prossimi esempi, anche se questa, da applicazione, funziona e viene mostrata correttamente per ognuno degli esempi che mostriamo di seguito.

2.8.2. Immagine 2:

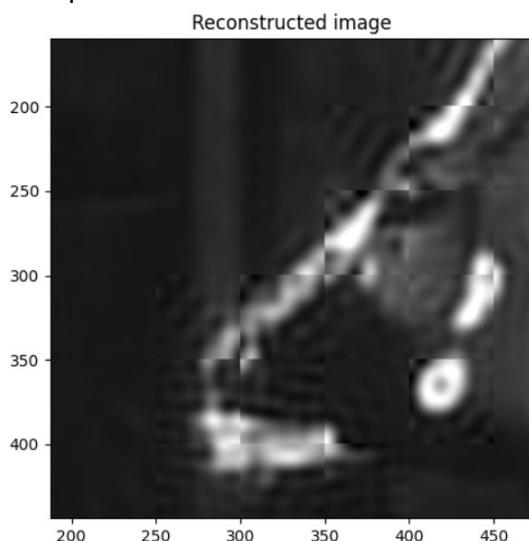
```
Selected image: D:/Repos/python/Scientific_Calculus_2/parte2/Immagini/deer.bmp
Original image size: 1959.814453125 KB
Original image shape: (1011, 661)
```

Compressione con $F = 50$ e $d = 10$ (esempio di crop e fenomeno di Gibbs)



La qualità della ricostruzione di questa immagine è relativamente buona (finché non si zooma nei dettagli), ma si vuole presentare questa elaborazione come chiaro esempio dei seguenti problemi:

- Crop dell'immagine: è un fenomeno avevamo predetto sarebbe accaduto in base a come è strutturato il codice. In questo caso abbiamo che la dimensione originale dell'immagine è (1011×661) utilizzando una finestra di (50×50) significa lasciare 11 pixel di bordo nero all'estremità dell'immagine, in quanto in questi pixel l'immagine non viene elaborata e si lascia dunque il valore dei pixel a 0, ossia li si lascia completamente neri.
- Discontinuità sui bordi delle finestre + artifact di forma quadrata + Presenza del fenomeno di Gibbs: In particolare se zoomiamo sull'immagine questi problemi diventano particolarmente evidenti:

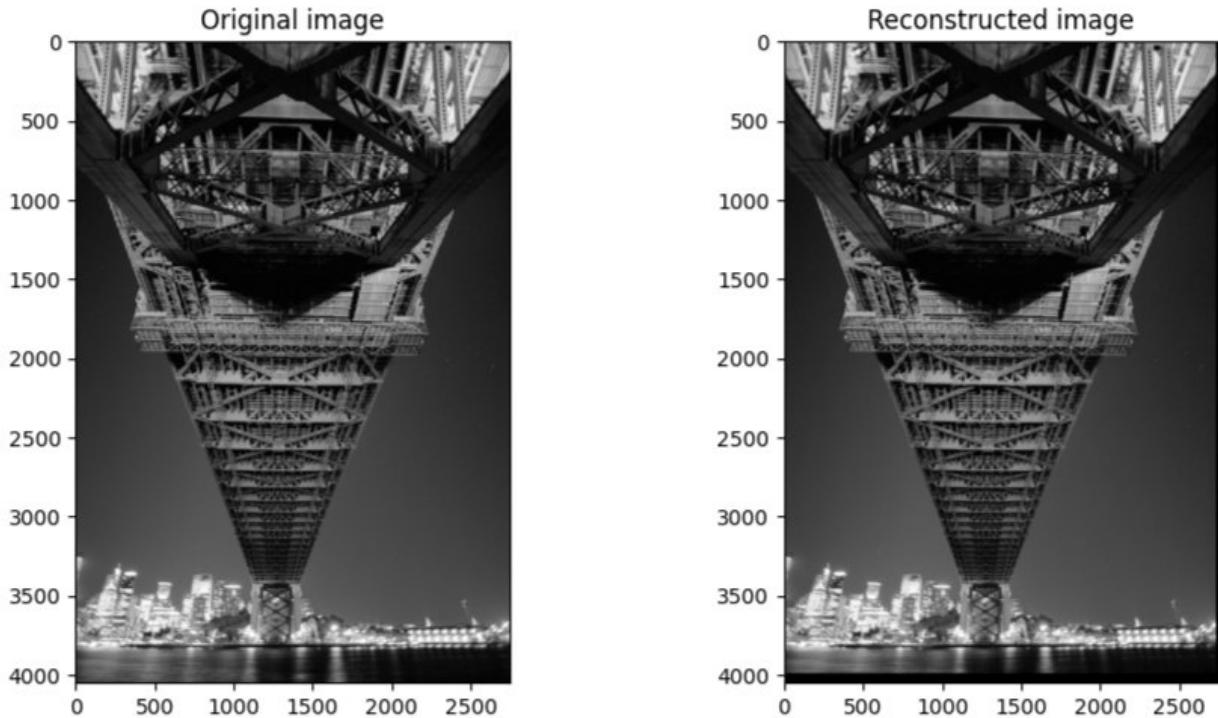


Il fenomeno di Gibbs è un effetto tipico della troncatura delle alte frequenze nella DCT. Si manifesta come oscillazioni e sovra/sotto-shooting vicino ai bordi netti dell'immagine. Il fenomeno è particolarmente visibile vicino ai contorni dell'animale, dove si notano strisce o bande luminose attorno ai bordi. Questo avviene appunto perché la brusca transizione di frequenze tra due regioni (ad esempio tra il corpo dell'animale e lo sfondo) richiede componenti ad alta frequenza per essere descritta con precisione. Eliminando noi gran parte delle frequenze con questa compressione diventa facile prevedere la comparsa di questo tipo di artefatto.

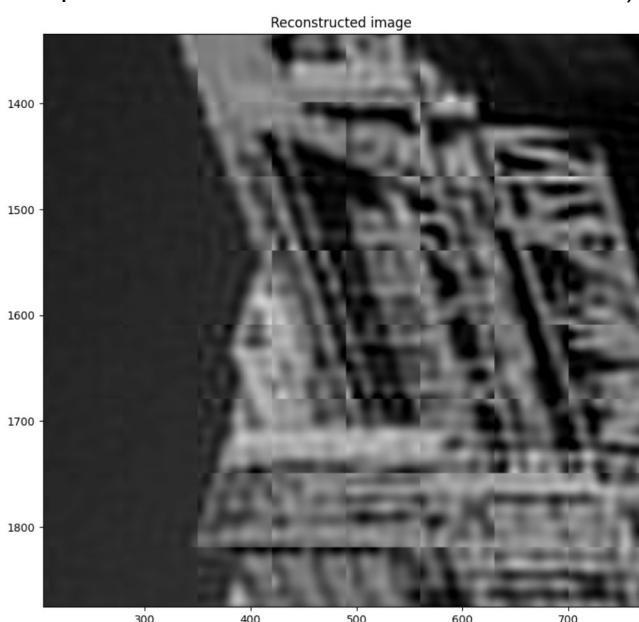
2.8.3. Immagine 3:

```
Selected image: D:/Repos/python/Scientific_Calculus_2/parte2/Immagini/bridge.bmp
Original image size: 19837.36328125 KB
Original image shape: (2749, 4049)
```

Compressione con $F = 70$ e $d = 10$ (esempio di computazione tempi di computazione lunghi)



Come possiamo vedere da quest’altro esempio anche se si tiene una soglia d molto bassa ($d = 10$) su una finestra di dimensioni abbastanza grandi ($F = 70$) il prodotto della compressione rimane, in questo caso, comunque di ottima qualità. Questo succede perché l’immagine originale ha dimensioni MOLTO grandi (2749×4099), quindi usando una finestra di dimensione 70, stiamo in verità usando una finestra molto piccola rispetto la dimensione totale dell’immagine, dandoci l’illusione che il risultato di compressione sia stato ottimo. Se però andiamo a zoomare nelle regioni più interne dell’immagine compressa possiamo notare gli stessi identici problemi che abbiamo notato nell’immagine precedente (Discontinuità sui bordi delle finestre + artifact di forma quadrata + Presenza del fenomeno di Gibbs):



In questa sezione si vuole però esaminare un altro aspetto fondamentale: la scalabilità del tempo di computazione. Diversamente dal progetto-1 in questo caso non è stato misurato da codice la durata dei tempi di elaborazione, ma sappiamo dall’analisi teorica che la scalabilità dei tempi dovrebbe crescere significativamente ($O(N^2 \log(N))$) aumentano la dimensione dell’immagine, rendendo il metodo poco scalabile. Questo si riscontra con i tempi di computazione ottenuti per questa immagine. Essendo questa immagine molto più grande abbiamo che richiede tempi di elaborazione nella scala di qualche secondo, quando invece le altre immagini richiedevano elaborazioni in tempi di frazione di secondo. Possiamo dunque dedurre, che anche il principio di scalabilità bassa che ci aspettavamo è rispettato (anche se stiamo basando questa affermazione su osservazioni puramente euristiche).

2.9. Problemi riscontrati con l'applicazione:

Un problema riscontrato durante l'utilizzo dell'applicazione è che dalla quarta elaborazione in poi compare questo errore in console di debug:

```
"C:\Users\admin\AppData\Local\Programs\Python\Python310\lib\tkinter_init.py", line 388, in del if
self._tk.getboolean(self._tk.call("info", "exists", self.name)): RuntimeError: main thread is not in main
loop Exception ignored in: <function Image.__del__ at 0x000001BED14C9870> Traceback (most
recent call last): File "C:\Users\admin\AppData\Local\Programs\Python\Python310\lib\tkinter_init.py",
line 4046, in del self.tk.call('image', 'delete', self.name) RuntimeError: main thread is not in main loop
```

Questo stesso messaggio viene stampato in console 4/5 volte, è causato dall'utilizzo della libreria `matplotlib.pyplot` (`plt`) in congiunzione con il multithreading, più nello specifico il messaggio di errore è dovuto all'utilizzo della funzione `display_images` in un thread separato. La non compatibilità delle librerie di threading sistema e `plt` causa questa eccezione.

Si vuole fare però notare che questo non è un breaking bug in quanto l'eccezione è gestita come possibile nel codice dell'applicazione, però la soluzione causa delay nel mostrare il paragone delle immagini (dal quarto tentativo in poi). Alternativamente per aggirare questo problema è semplicemente possibile riavviare l'applicazione.

2.10. Conclusioni parte-2:

I risultati ottenuti confermano che la DCT2 è uno strumento molto efficace per la compressione, ma mostrano anche quanto sia fondamentale scegliere con attenzione i parametri d ed F necessari per trovare il giusto compromesso tra qualità visiva e riduzione dei dati. Possiamo ritenerci soddisfatti dello sviluppo anche di questo progetto in quanto abbiamo dimostrato sia che tutti i requisiti tecnici (2.1) sono stati rispettati (mostrando una descrizione dettagliata per ogni metodo che implementa la compressione), sia che tutte le nostre aspettative (2.2) si sono dimostrate corrette. Abbiamo infatti che tramite l'analisi dei risultati abbiamo dimostrato che:

- **Compressione efficace delle immagini:** scegliendo una soglia adeguata di d ed F è possibile ottenere un buon compromesso
- **Immagini in cui si perde tutta l'informazione:** al contrario del caso precedente, se si scelgono valori inadatti per le variabili di compressione (come un d troppo piccolo) si corre il rischio di perdere tutto il valore informativo delle immagini.
- **Immagini con crop:** se non si sta attenti al valore inserito di F per come è strutturato il codice è possibile ottenere in output un'immagine croppata (esattamente come ci aspettavamo)
- **Presenza di artifact nell'immagine compressa:** In linea con le nostre aspettative, operando con questo tipo di compressione è possibile fare emergere diversi tipi di artefatti tra cui:
 - Artifact quadrati che rappresentano il bordo della finestra $F \times F$
 - Discontinuità tra bordi di due blocchi f adiacenti
 - Presenza del fenomeno di Gibbs nei punti di edges dell'immagine
- **Scalabilità della compressione:** Seppur basato su osservazioni puramente euristiche è stato comunque possibile verificare che il metodo di compressione implementato con FFT (tramite funzioni di libreria) abbia tempi di compressione che aumentano significativamente all'aumentare della dimensione dell'immagine. Cosa che ci aspettavamo in quanto l'applicazione di DCT2 / IDCT2 che applichiamo hanno complessità $O(N^2 \log(N))$

Concludendo, siamo perfettamente soddisfatti dei risultati raggiunti, in quanto perfettamente in linea con le nostre aspettative.