

# Metodi del Calcolo Scientifico Progetto 2

Mirko Papadopoli  
[m.papadopoli@campus.unimib.it](mailto:m.papadopoli@campus.unimib.it)

Settembre 2023

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Introduzione . . . . .	2
1.2	Obiettivo . . . . .	2
1.3	Linguaggio e librerie utilizzate . . . . .	3
<b>2</b>	<b>Parte Prima</b>	<b>5</b>
2.1	Verifica DCT2 . . . . .	6
2.2	Implementazione della DCT2 . . . . .	7
2.3	Analisi dei Risultati DCT2 . . . . .	10
<b>3</b>	<b>Introduzione</b>	<b>12</b>
3.1	Introduzione . . . . .	12
3.2	Compressione . . . . .	12
3.3	Analisi dei Risultati . . . . .	17
<b>4</b>	<b>Conclusioni</b>	<b>23</b>
4.1	Conclusioni . . . . .	23

# Capitolo 1

## Introduzione

### 1.1 Introduzione

La DCT2, o Trasformata Discreta del Cosine Tipo 2, è una tecnica utilizzata principalmente per analizzare e rappresentare i segnali o le immagini.

In particolare, la DCT2 scomponete un segnale o un'immagine in una serie di coefficienti che rappresentano le diverse componenti spettrali presenti nel segnale. Questi coefficienti catturano informazioni sulla distribuzione delle frequenze nel segnale o nell'immagine.

Essa, viene spesso utilizzata, ad esempio, nello standard di compressione JPEG.

Scomponete l'immagine in una serie di blocchi o "macroblocchi" e calcola i coefficienti DCT2 per ciascun blocco.

Poi, attraverso un processo di quantizzazione, è possibile ridurre la precisione dei coefficienti meno rilevanti, consentendo una significativa riduzione dello spazio necessario per memorizzare l'immagine.

### 1.2 Obiettivo

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.

Nello specifico, l'intento è valutare l'effetto della compressione sulla qualità visiva dell'immagine e sulla precisione nella resa dei dettagli.

È importante evidenziare che l'algoritmo di compressione proposto opera senza l'utilizzo di una matrice di quantizzazione, un elemento centrale nell'algoritmo di compressione JPEG tradizionale.

L'analisi dei risultati conseguiti può contribuire a una maggiore comprensione degli impatti della compressione sulla qualità delle immagini in scala di grigi, apendo la possibilità a potenziamenti e ottimizzazioni nell'ambito dell'algoritmo di compressione.

## 1.3 Linguaggio e librerie utilizzate

Per questo progetto, è stata adottato un'approccio open-source per implementare l'algoritmo DCT2 e studiare gli effetti della compressione sulle immagini in toni di grigio.

In particolare, sono state scelte le seguenti librerie:

- **Numpy**, viene utilizzata come libreria per il calcolo numerico in Python. Essa offre una vasta gamma di funzionalità per la manipolazione e l'elaborazione efficiente di array multidimensionali.

Nello specifico viene utilizzata per implementare `np.fft.fft2(image_block)`. Questa funzione chiama il metodo FFT, che consente di utilizzare la trasformata di Fourier bidimensionale su un'immagine rappresentata come un array multidimensionale. La trasformata di Fourier è una tecnica matematica che consente di scomporre un segnale o un'immagine in una combinazione di componenti sinusoidali di diverse frequenze.

- **Time**, offre funzionalità per misurare il tempo e gestire operazioni legate al tempo.
- **OS**, fornisce funzioni e metodi per interagire con il sistema operativo.
- **Matplotlib**, fornisce strumenti per creare grafici di diversi tipi, tra cui grafici a dispersione, istogrammi, grafici a barre, grafici a linee e molto altro.
- **Tkinter**, libreria di Python utilizzata per creare interfacce grafiche (GUI). Fornisce un insieme di strumenti e widget per la creazione di finestre, pulsanti, caselle di testo, menu e molto altro. Con Tkinter, è possibile creare applicazioni desktop interattive con facilità. Verrà utilizzata per creare una interfaccia durante la fase due del progetto.
- **PIL**, è una libreria per l'elaborazione delle immagini in Python. Offre una vasta gamma di funzionalità per lavorare con immagini, tra cui aprire, salvare, modificare, trasformare, manipolare e visualizzare immagini in diversi formati come JPEG, PNG, BMP, TIFF.

- **CV2**, utilizzata per l'elaborazione delle immagini. Tra le molte funzionalità offerte da OpenCV, ci sono due funzioni che useremo nello specifico la funzione cv2.dct() e la funzione cv2.idct(), che consentono di eseguire la trasformata discreta del coseno (DCT) e l'inversa della DCT, rispettivamente.

Per quanto riguarda la scelta del linguaggio di programmazione, è stato optato Python, un linguaggio di programmazione di alto livello, orientato agli oggetti e interpretato. È celebre per la sua sintassi chiara e comprensibile, che promuove la leggibilità del codice e semplifica l'apprendimento per gli sviluppatori.

# Capitolo 2

## Parte Prima

Durante questa fase del progetto, il focus è sull'applicazione pratica della DCT2 seguendo il metodo illustrato durante le lezioni, all'interno di un ambiente open-source. Subito dopo, è prevista una comparazione dei tempi di esecuzione ottenuti dalla nostra implementazione originale rispetto alla versione "veloce" dell'algoritmo DCT2 presente nella libreria dell'ambiente scelto, che fa uso della FFT (Trasformata di Fourier Veloce).

Per effettuare l'analisi dei tempi di esecuzione, sarà necessario procurarsi una serie di array quadrati di dimensioni  $N \times N$ , con  $N$  crescente. Successivamente, si rappresenteranno i tempi impiegati per eseguire la DCT2 sia con l'algoritmo sviluppato internamente, sia con l'algoritmo della libreria, su un grafico in scala semilogaritmica.

Nel grafico, le quantità scalari saranno rappresentate sull'asse delle ascisse, mentre il logaritmo dei tempi di esecuzione sarà rappresentato sull'asse delle ordinate.

Si prevede che i tempi di esecuzione siano proporzionali a  $N^3$  per l'algoritmo "fatto in casa" della DCT2 e a  $N^2$  (o più precisamente a  $N^2 \log(N)$ ) per l'implementazione "fast" della libreria.

È importante notare che i tempi ottenuti con l'implementazione "fast" potrebbero presentare un andamento irregolare a causa del tipo di algoritmo utilizzato.

L'analisi dei tempi di esecuzione e il paragone tra le diverse realizzazioni permetteranno di giudicare l'efficacia e la velocità delle varie strategie nell'elaborazione della DCT2, offrendo dati importanti riguardo alle prestazioni dell'algoritmo sviluppato internamente in confronto all'implementazione altamente ottimizzata della libreria.

## 2.1 Verifica DCT2

Prima di procedere con l'implementazione della trasformata discreta, è stato necessario condurre una verifica per garantire che il risultato ottenuto dalla trasformata di un blocco 8x8 sia coerente con il risultato corretto atteso.

Questa verifica mi permette di verificare l'accuratezza dell'algoritmo e di individuare eventuali errori o discrepanze.

```
# Importare le librerie
import cv2
import numpy as np

# Matrice di verifica
matrixTest= 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]
], dtype=np.float32)

# Applica la DCT2 sulla matrice
dct_matrixTest = cv2.dct(matrixTest)

# Stampa la matrice DCT2 risultante
print("Matrice DCT2:")
print(dct_matrixTest)
```

Figura 2.1: Matrice di Test

Nell'immagine 2.1 è stata creata un matrice 8x8, è stato applicato l'algoritmo FFT fornito dalla libreria CV2 con il comando **cv2.dct(matrixTest)** e sono stati stampati i risultati.

La stessa verifica è stata fatta sulla prima riga del blocchetto  $8 \times 8$  sopra:

```
matrixVerify1R = np.array([
    [231, 32, 233, 161, 24, 71, 140, 245]
], dtype = np.float32)

# Applica la DCT sulla riga
dct_matrix1R = cv2.dct(matrixVerify1R)
# Stampare la riga DCT risultante
print ("Riga DCT2:")
print (dct_matrix1R)
```

Figura 2.2: Riga di Test

Dall’analisi dei risultati ottenuti, è emerso che l’output è corretto sia nel caso dell’algoritmo per la DCT2 che per la DCT monodimensionale.

```
Matrice DCT2:
[[ 1.11875000e+03  4.40221939e+01  7.59190369e+01 -1.38572403e+02
   3.50000095e+00  1.22078049e+02  1.95043869e+02 -1.01604897e+02]
 [ 7.71900864e+01  1.14868202e+02 -2.18014412e+01  4.13641357e+01
   8.77720356e+00  9.90829544e+01  1.38171509e+02  1.09092808e+01]
 [ 4.48351440e+01 -6.27524338e+01  1.11614105e+02 -7.63789673e+01
   1.24422157e+02  9.55984268e+01 -3.98287926e+01  5.85237503e+01]
 [-6.99836502e+01 -4.02408867e+01 -2.34970551e+01 -7.67320557e+01
   2.66457787e+01 -3.68328285e+01  6.61891479e+01  1.25429726e+02]
 [-1.09000008e+02 -4.33430824e+01 -5.55436935e+01  8.17346764e+00
   3.02500019e+01 -2.86602440e+01  2.44150138e+00 -9.41436996e+01]
 [-5.38783979e+00  5.66344948e+01  1.73021515e+02 -3.54234543e+01
   3.23878288e+01  3.34576683e+01 -5.81167870e+01  1.90225639e+01]
 [ 7.88439331e+01 -6.45924072e+01  1.18671196e+02 -1.50904827e+01
   -1.37316925e+02 -3.06196651e+01 -1.05114113e+02  3.98130455e+01]
 [ 1.97882385e+01 -7.81813278e+01  9.72310781e-01 -7.23464203e+01
   -2.15781631e+01  8.12999039e+01  6.37103729e+01  5.90618467e+00]]
Riga DCT2:
[[ 401.9902       6.6000195  109.16736   -112.78558      65.40738
   121.8314      116.65649  28.800402 ]]
○ (base) mirkopapadopoli@MBP-di-Mirko DCT_Compression %
```

Figura 2.3: Output

## 2.2 Implementazione della DCT2

L’implementazione della funzione my\_dct2 calcola la DCT2 di un’immagine bidimensionale utilizzando la formula matematica della DCT2. La funzione itera attraverso ogni pixel

dell’immagine e calcola la somma dei prodotti dei valori dei pixel per le componenti coseno corrispondenti. Questa somma viene poi moltiplicata per i coefficienti di scala appropriati e normalizzata in base alle dimensioni dell’immagine.

Di seguito il codice utilizzato:

```
def my_dct2(image_block):

    M, N = image_block.shape
    dct_result = np.zeros((M, N))

    for i in range(M):
        for j in range(N):
            alpha_i = 1 if (i == 0) else (np.sqrt(2))
            alpha_j = 1 if (j == 0) else (np.sqrt(2))
            sum_value = 0

            for x in range(M):
                for y in range(N):
                    cos_i = np.cos((2 * x + 1) * i * np.pi) / (2 * M)
                    cos_j = np.cos((2 * y + 1) * j * np.pi) / (2 * N)
                    sum_value += image_block[x, y] * cos_i * cos_j

            dct_result[i, j] = alpha_i * alpha_j * sum_value / np.sqrt(M * N)

    return dct_result
```

Figura 2.4: my\_dct2

L’implementazione della DCT2 è stata fatta in questo modo:

- **M, N = image.shape**: ottiene le dimensioni dell’immagine di input e assegna i valori alle variabili M e N.
- **dct\_results = np.zeros((M, N))**: crea un array di zeri con le stesse dimensioni dell’immagine di input che sarà utilizzato per memorizzare i valori della DCT2.
- **for i in range(M)**: itera attraverso le righe dell’immagine.
- **for j in range(N)**: itera attraverso le colonne dell’immagine.
- **alpha\_i = 1 if i == 0 else np.sqrt(2)**: calcola il coefficiente di scala alpha.i per la frequenza orizzontale i. Se alpha.i è uguale a 0, il coefficiente di scala è 1, altrimenti è la radice quadrata di 2.

- **alpha\_j = 1 if j == 0 else np.sqrt(2)**: calcola il coefficiente di scala alpha\_j per la frequenza verticale j. Se alpha\_j è uguale a 0, il coefficiente di scala è 1, altrimenti è la radice quadrata di 2.
- **sum\_value = 0**: inizializza una variabile per la somma dei valori dei pixel dell'immagine moltiplicati per le componenti coseno.
- **for x in range(M)**: itera attraverso le righe dell'immagine.
- **for y in range(N)**: itera attraverso le colonne dell'immagine.
- **cos\_i = np.cos(((2 \* x + 1) \* i \* np.pi) / (2 \* M))**: calcola l'angolo theta per la frequenza orizzontale i e la posizione del pixel (x, y).
- **cos\_j = np.cos(((2 \* y + 1) \* j \* np.pi) / (2 \* N))**: calcola l'angolo theta per la frequenza verticale j e la posizione del pixel (i, j).
- **sum\_value += image\_block[x, y] \* cos\_i \* cos\_j**: aggiunge alla somma il prodotto del valore del pixel (x, y) per le componenti coseno corrispondenti.
- **dct\_result[i, j] = alpha\_i \* alpha\_j \* sum\_value / np.sqrt(M \* N)**: calcola il valore della DCT2 per la frequenza orizzontale u e la frequenza verticale v utilizzando i coefficienti di scala alpha\_i e alpha\_j. Il valore viene normalizzato dividendo per la radice quadrata del prodotto delle dimensioni dell'immagine M e N.
- **return dct\_result**: restituisce l'array dct\_result contenente i valori della DCT2 dell'immagine di input.

L'algoritmo implementato segue il concetto fondamentale della DCT2, che sfrutta la correlazione spaziale delle componenti dell'immagine per rappresentarla con un numero ridotto di coefficienti.

I coefficienti di frequenza più bassa, che corrispondono alle componenti coseno con frequenze più basse, tendono ad avere un impatto maggiore sull'immagine rispetto ai coefficienti di frequenza più alta.

Pertanto, la DCT2 consente di concentrare l'energia dell'immagine in un numero limitato di coefficienti, permettendo una compressione efficiente dei dati.

## 2.3 Analisi dei Risultati DCT2

Nel corso dell'esame dei dati ottenuti, si valutano le prestazioni relative al tempo e alle dimensioni delle due realizzazioni della DCT2: una sviluppata internamente e l'altra presa dalla libreria NumPy. Questa valutazione è condotta su matrici NxN che aumentano progressivamente in dimensione. Come evidente dalla figura seguente, creiamo un'array che cresce in maniera esponenziale.

Di seguito l'array NxN utilizzato per la DCT2:

```
get_ipython().  
N_values = [25, 50, 75, 100, 150, 200]
```

Figura 2.5: Array NxN

Lo step successivo è stato quello mostrare il risultato delle analisi attraverso un grafico che avrà, rispettivamente, sulle ordinate il tempo di esecuzione in secondi e sulle ascisse la dimensione. Il grafico è in scala logaritmica.

Confronto dei tempi di esecuzione DCT2 personalizzata vs DCT2 libreria FFT

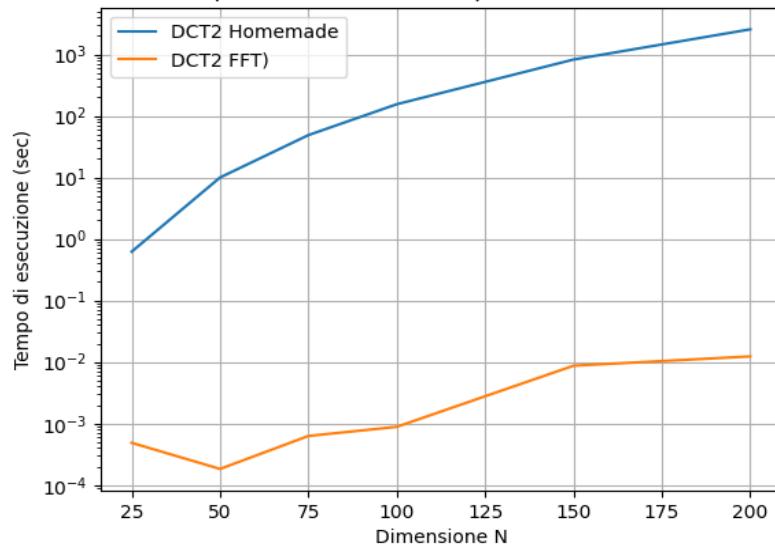


Figura 2.6: Risultati

Come si può notare dal grafico mostrato in 2.6 le assunzioni fatte inizialmente si sono rivelate corrette.

L'algoritmo implementato ovvero il **my\_dct2** completerà l'esecuzione con tempo  $N^3$  mentre invece la DCT2 della libreria Numpy avrà come tempo di esecuzione  $N^2 \log(N)$ . Si

nota inoltre che l'algoritmo FFT presenta un andamento irregolare come precedentemente menzionato.

# **Capitolo 3**

## **Introduzione**

### **3.1 Introduzione**

La fase successiva del progetto mira a indagare l'applicazione pratica della DCT2 e a analizzare le conseguenze di un metodo di compressione simile a JPEG sulle immagini monocromatiche.

In questa parte del progetto, l'attenzione è rivolta alla creazione di un'interfaccia utente intuitiva che permette agli utilizzatori di scegliere un'immagine in formato .bmp in scala di grigi dal proprio sistema operativo.

Inoltre, gli utilizzatori avranno la possibilità di definire due parametri chiave:

- un intero  $F$  che è l'ampiezza delle finestrelle (macro-blocchi) in cui si effettuerà la DCT2.
- una soglia per la soppressione delle frequenze, rappresentata da un numero intero d compreso tra 0 e  $(2F - 2)$ .

L'output finale sarà la visualizzazione sullo schermo di due immagine affiancate rispettivamente, l'immagine originale e l'immagine ottenuta dopo aver modificato le frequenze nei blocchi.

### **3.2 Compressione**

Il processo di compressione e decomposizione dell'immagine in formato .bmp utilizza le seguenti funzioni:

- la funzione **apply\_dct2** che richiama la funzione DCT della libreria CV2.
- **apply\_inverse\_dct2** richiama la funzione IDCT della libreria CV2 ovvero l'inversa della DCT2.

```
# Utilizzo cv2 che calcola la DCT2 utilizzando una versione ottimizzata dell'algoritmo,
def dct2(image):
    return cv2.dct(np.float32(image))

# DCT2 inversa, ricostruisce l'immagine dopo l'applicazione della DCT2 e il taglio delle frequenze.
def inverse_dct2(coefficients):
    return cv2.idct(coefficients)
```

Figura 3.1: Interfaccia

Inoltre, è stata implementata la funzione **threshold\_cutoff(coefficients, threshold)** con due parametri coefficients e threshold. La funzione ha l'obiettivo di eseguire un'operazione di filtraggio che mira a rimuovere le componenti ad alta frequenza dall'insieme dei coefficienti ottenuti tramite la DCT2. La funzione itera su tutti gli elementi dell'array, se la somma degli indici supera o è uguale a una soglia specificata, l'elemento viene settato a 0, eliminando di conseguenza la componente ad alta frequenza. Il risultato della funzione è quindi rappresentato dall'insieme dei coefficienti DCT2 modificato.

Inoltre, è stata implementata la funzione **ProcessImage**, la funzione principale che esegue l'intero processo di compressione e decompressione dell'immagine, dettagliando le operazioni che vengono eseguite per ridurre la dimensione dell'immagine e successivamente ripristinarla alla sua forma originale. La funzione durante la sua esecuzione richiama altre funzioni che svolgono operazioni specifiche.

`process_image` è una funzione che prende in input il percorso dell'immagine, la dimensione dei blocchi e una soglia di taglio delle frequenze.

La funzione svolge diverse operazioni per elaborare l'immagine utilizzando funzioni di supporto. Nello specifico la funzione process\_image esegue le seguenti operazioni:

1. **upload\_image**: si occupa del carimento dell'immagine in toni di grigio, letta come una matrice di valori.

```
# Carica l'immagine
def upload_image():
    global selected_image_path, window, block_size_entry, threshold_entry, image_label
    image = np.asarray(Image.open(selected_image_path).convert('L'))
    if len(image.shape) > 2:
        # Carica l'immagine in toni di grigio
        image = cv2.imread(selected_image_path, 0)
    return image
```

Figura 3.2: def upload\_image()

2. **image\_dimension(image)**: calcola le dimensioni dell'immagine, ottenendo l'altezza e la larghezza in pixel.

```
# Ricostruisce l'immagine a partire dai blocchi
def image_dimension(image):
    height, width = image.shape
    return height, width
```

Figura 3.3: image\_dimension()

3. **num\_blocks(height, width, block\_size)**: calcola il numero di blocchi che possono essere creati in base alla dimensione specificata (block\_size) e alla dimensione minima tra altezza e larghezza dell'immagine. Ciò assicura che i blocchi siano di dimensioni uniformi e che siano presenti solo quelli completamente all'interno dell'immagine.

```
# Calcola il numero di blocchi da creare
def num_blocks(height, width, block_size):
    num_blocks_height = height // block_size
    num_blocks_width = width // block_size
    return num_blocks_height, num_blocks_width
```

Figura 3.4: num\_block()

4. L'immagine viene ritagliata in modo che abbia la dimensione effettiva.
5. **divide\_image\_in\_blocks(image, num\_blocks\_height, num\_blocks\_width)**: l'immagine viene suddivisa in blocchi utilizzando la funzione np.split di NumPy. Vengono eseguite due suddivisioni: una lungo l'asse delle righe (axis=0) e una lungo l'asse delle colonne (axis=1). Il risultato è una matrice tridimensionale che rappresenta i blocchi.

```

# Ritaglia l'immagine alla dimensione effettiva
def divide_image_in_blocks(image, num_blocks_height, num_blocks_width):
    blocks = np.split(image, num_blocks_height, axis=0)
    blocks = [np.split(block, num_blocks_width, axis=1) for block in blocks]
    blocks = np.array(blocks)
    return blocks

```

Figura 3.5: divide\_image\_in\_blocks()

6. **run\_process\_block(blocks, threshold)**: funzione che contiene un doppio ciclo for per iterare su tutti i blocchi nell’immagine. All’interno del ciclo:

- il blocco corrente viene estratto dalla matrice dei blocchi.
- viene applicata la DCT2 al blocco utilizzando la funzione ausiliaria **dct2**.
- l’output della funzione sono i coefficienti della trasformata.
- viene applicato il taglio delle frequenze utilizzando la funzione ausiliaria **threshold\_cutoff**.
- viene applicata l’inverse della DCT2 al blocco modificato utilizzando la funzione ausiliaria **inverse\_dct2**. Questa operazione ripristina il blocco all’intensità dei pixel originale.
- i valori del blocco ricostruito vengono arrotondati al valore intero più vicino utilizzando **np.round**.
- oltre all’arrotondamento vengono normalizzati per essere compresi nell’intervallo [0, 255] utilizzando **np.clip**.
- i valori inferiori a 0 vengono impostati a 0 e quelli superiori a 255 vengono impostati a 255.

7. il blocco ricostruito viene aggiornato nella matrice dei blocchi. Una volta completata l’elaborazione di tutti i blocchi, viene ricomposta l’immagine finale utilizzando la funzione **np.block**. Questa funzione crea una nuova matrice combinando i blocchi in base all’ordine corretto.

```

# Ricompone l'immagine a partire dai blocchi
reconstructed_image = np.block([[block for block in row] for row in blocks_1])

```

Figura 3.6: reconstruction\_dimension\_image()

8. l'immagine ricostruita viene convertita nel formato di byte utilizzando astype(np.uint8)

```
# Converte l'immagine in formato byte
reconstructed_image = reconstructed_image.astype(np.uint8)
```

Figura 3.7: Reconstructed image

9. l'immagine convertita viene salvata su disco in una cartella apposita.

```
# Salva l'immagine ricostruita
def save_disk_image(reconstructed_image):
    img_path = "/Users/mirkopapadopoli/Code/DCT_Compression/img_reconstructed/"
    output_path = img_path + selected_image_path.split('/')[-1] + "_reconstructed.bmp"
    Image.fromarray(reconstructed_image).save(output_path)
    return output_path
```

Figura 3.8: save\_image

10. viene calcolata la dimensione dell'immagine ricostruita in KB

```
# Calcola la dimensione dell'immagine ricostruita in KB
file_size_kb_original, file_size_kb = reconstruction_dimension_image(output_path)
```

Figura 3.9: Dimension Image

11. viene visualizzata la dimensione dell'immagine ricostruita rispetto a quella originale

```

# Visualizza la dimensione dell'immagine ricostruita
print(f"Dimensione dell'immagine originale: {file_size_kb_original:.2f} KB")
print(f"Dimensione dell'immagine ricostruita: {file_size_kb:.2f} KB")

print("Shape immagine originale:", image.shape)
print("Shape immagine ricostruita:", reconstructed_image.shape)

print_original_reconstructed_image(image, reconstructed_image)

```

Figura 3.10: Reconstruction Image

La tecnica adottata è quella che, quando la dimensione dell’immagine viene suddivisa dalla `block_size` non in modo omogeneo, e quindi con uno scarto andiamo a tagliare quest’ultimo e si osserva come le dimensioni di quest’ultime diminuiscano.

Oltre alle funzioni sopra elencate, ne sono state implementate delle altre:

- `choose_image()` : è una funzione che permette di scegliere e caricare una immagine dal proprio sistema operativo. Viene fatto tramite l’utilizzo del pulsante ”carica immagine”
- `proportionally_resize(width, height, max_width, max_height)`: è una funzione che permette di adattare le proporzioni delle grandezze delle immagini.
- `reset_variables()`: è una funzione che permette di resettare tutte le variabili utilizzate in modo tale da poter effettuare ulteriori test sulle immagini senza dover riavviare lo script.
- `create_interface()`: è la funzione che permette di creare tutta l’interfaccia grafica.

### 3.3 Analisi dei Risultati

Una volta implementato tutto lo script, esso è stato testato su una serie di immagini test. Le immagini, sono state confrontate prima e dopo la compressione. Di seguito verranno mostrati alcuni risultati ottenuti:

la prima immagine che è stata analizzata è una 20x20 In questo caso sono stati importati i seguenti parametri:

- Ampiezza (F): 4
- Soglia di taglio (d): 20

Come si può notare pare che la compressione non abbia provocato nessun effetto nè di ridimensionamento, nè di imperfezione dell’immagine. Ciò avviene perché il parametro F

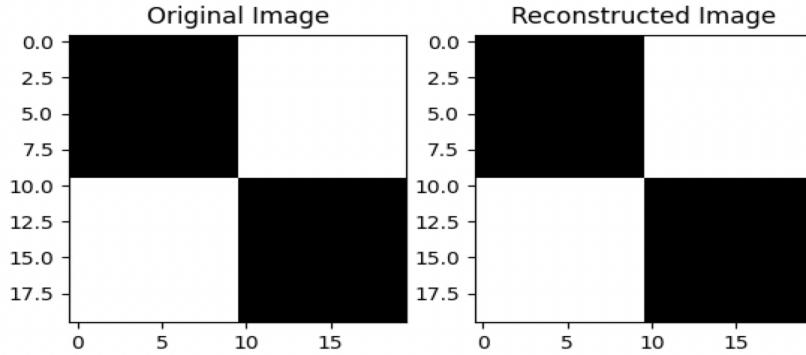


Figura 3.11:  $20 \times 20$ ,  $F = 4$ ,  $d = 20$

definisce la dimensione del blocco, impostata a 2, che è un multiplo delle dimensioni dei blocchi neri o bianchi nell’immagine (20, 20). Pertanto, all’interno del blocco ( $F, F$ ) non ci sono transizioni di colore che richiedono approssimazioni, garantendo una compressione efficiente senza alcuna perdita di qualità. Nel caso in cui si vada a modificare sia il parametro  $F$  che il parametro  $d$  con un valore  $F$  che non sia un sottomultiplo di 20, come ad esempio 6 è possibile osservare sia la riduzione delle dimensioni dell’immagine che la presenza di artefatti lungo i lati dei quadrati neri e bianchi. Ciò avviene perché ci sono blocchi ( $F, F$ ) che contengono più

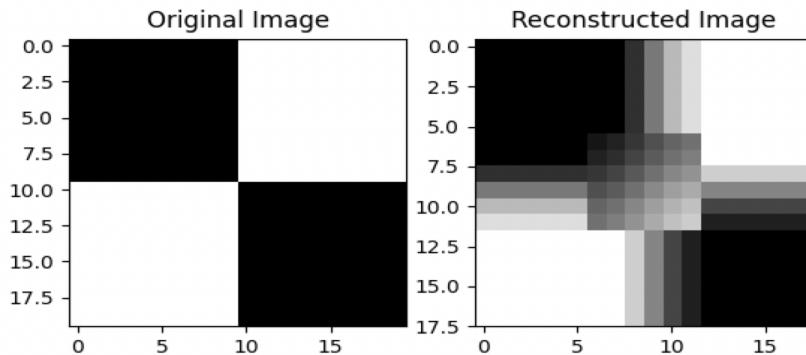


Figura 3.12:  $20 \times 20$ ,  $F = 6$ ,  $d = 2$

colori e, durante l’approssimazione, si creano artefatti visibili che eliminano il netto confine di colore presente nell’immagine originale.

La seconda immagine di test che è stata analizzata è cathedral, una delle immagini più

complesse e più grandi delle prove. Si può dall'immagine che all'apparenza non sembra ci

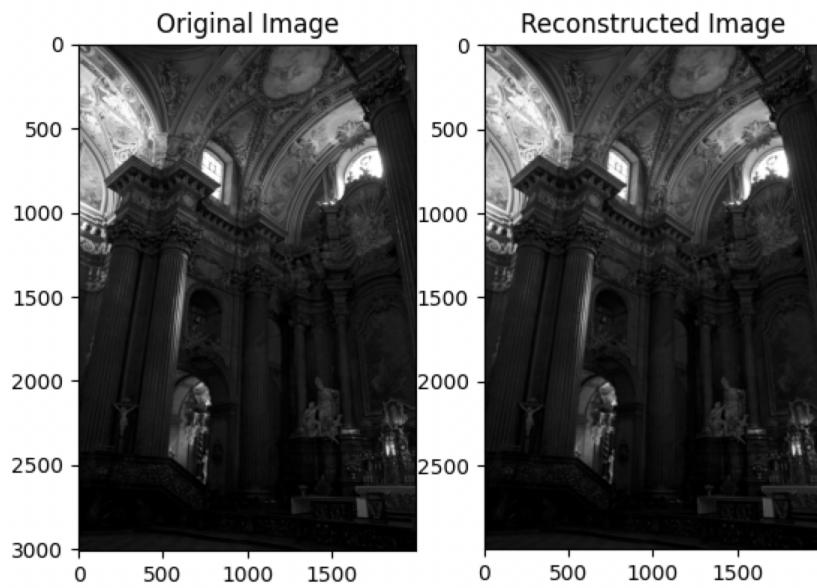


Figura 3.13:  $F = 20$ ,  $d = 4$

siano modifiche né di qualità dell'immagine né di ridimensionamento. Se però ingrandiamo l'immagine si vede come sono visibili gli artefatti. Non è possibile fare lo stesso discorso quando si inseriscono valori come  $F = 70$  e  $d = 2$ .

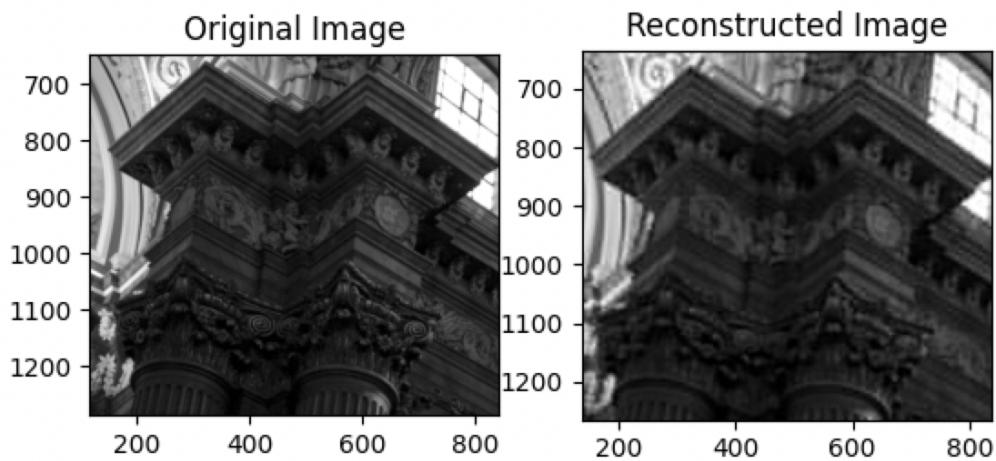


Figura 3.14:  $F = 20$ ,  $d = 4$

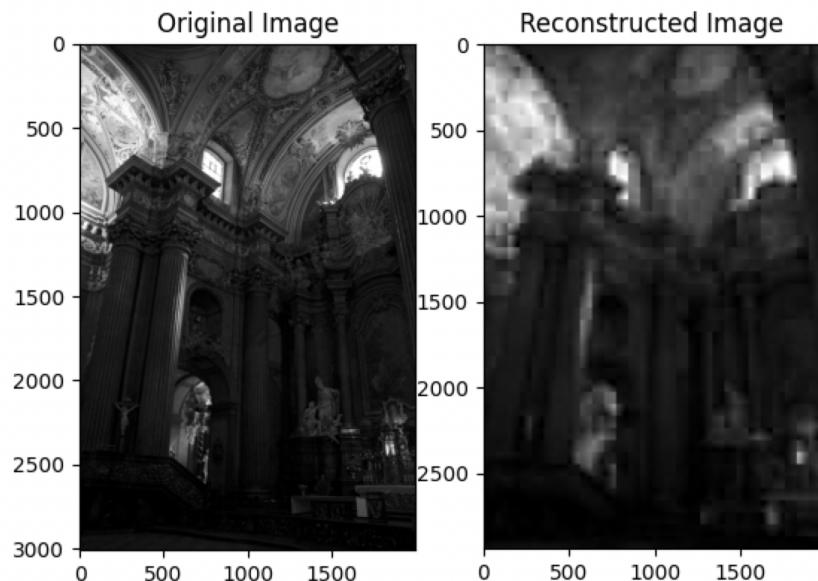


Figura 3.15:  $F = 70$ ,  $d = 2$

L'ultimo test effettuato è stata sull'immagine shoe. Si nota come mettendo un valore al parametro  $F$  pari a 20 e un valore al parametro  $d$  a 10 non vi sono grandi effetti in senso negativo. Anche in questo caso se ingrandissimo l'immagine si può notare come l'immagine

non è ben definita.

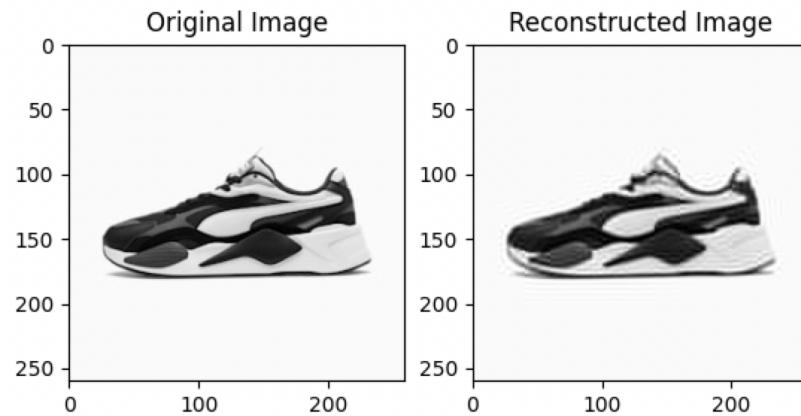


Figura 3.16:  $F = 20$ ,  $d = 10$

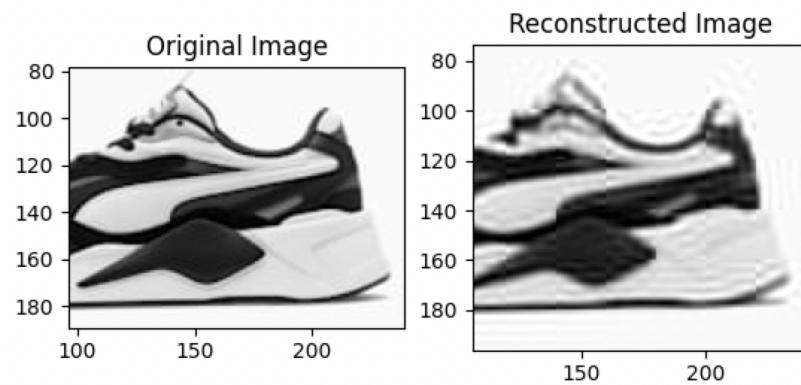


Figura 3.17:  $F = 20$ ,  $d = 10$

Contrariamente mettendo come valori  $F$  a 20 e  $d$  a 2, notiamo una differenza sostanziale e anche in questo caso l'immagine si mostra molto sgranata e di bassa qualità.

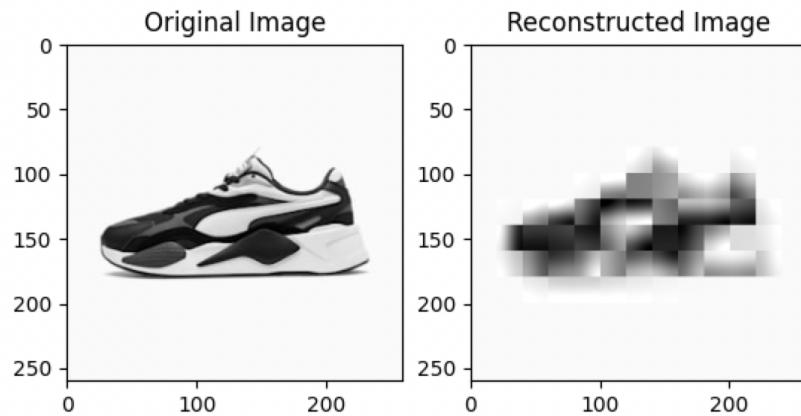


Figura 3.18:  $F = 20$ ,  $d = 10$

# **Capitolo 4**

## **Conclusioni**

### **4.1 Conclusioni**

Per concludere, è stato condotto un insieme di esperimenti utilizzando varie tipologie di immagini al fine di valutare l'efficacia della compressione e individuare eventuali sfide.

Durante questi test, sono emerse difficoltà nella compressione quando le immagini presentavano transizioni di colore molto nette o oggetti di dimensioni molto ridotte da preservare.

In situazioni particolarmente critiche, è stato evidenziato che è essenziale mantenere un numero considerevole di coefficienti per assicurare una qualità dell'immagine accettabile.