

UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

METODI DEL CALCOLO SCIENTIFICO
PROGETTO 2

Compressore di Immagini

Autori:

Mattia Ingrassia - 879204 - m.ingrassia3@campus.unimib.it
Riccardo Ghilotti - 879259 - r.ghilotti@campus.unimib.it

Appello di Giugno 2025



Indice

1	Introduzione	1
1.1	Comparison DCT2	1
1.2	Image Compressor	1
2	Struttura del progetto	2
3	Prima parte - DCT comparison	3
3.1	Custom DCT	3
3.2	DCT con FFT - SciPy	4
3.3	Analisi dei risultati	4
4	Seconda parte - Image Compressor	6
4.1	Image Compressor - Interfaccia Grafica	6
4.2	Algoritmo di compressione	7
4.3	Suddivisione dell'immagine in blocchi	9
4.3.1	Step 1: Primo reshape	9
4.3.2	Step 2: Swap axes	10
4.3.3	Step 3: Reshape finale	10
4.3.4	Vettore di blocchi finale:	11
4.4	Compressione	11
4.5	Decompressione	12
4.6	Ricomposizione	12
4.6.1	Step 1: Primo reshape	12
4.6.2	Step 2: Transpose	13
4.6.3	Step 3: Reshape finale	13
4.7	Risultato finale - Matrice ricomposta:	13
5	Conclusione - Analisi dei risultati	14
5.1	Immagine checkers	14
5.2	Immagine bridge	15
5.3	Immagine C	16
5.4	Immagine deer	17
Bibliografia		19

Elenco delle figure

1	Confronto sui tempi di esecuzione di DCT2 - Custom e DCT2 - FFT	5
2	Immagine della GUI di Image Compressor	6
3	Immagine di un ponte, prima e dopo la compressione con $F=30$ e $d=1$	7
4	Coefficienti delle frequenze prima del taglio in un blocco 16×16 con $d = 4$	12
5	Coefficienti delle frequenze dopo il taglio in un blocco 16×16 con $d = 4$	12
6	checkers80x80 originale	15
7	checkers80x80 compressa con $F=10$ e $d=7$	15
8	checkers80x80 compressa con $F=7$ e $d=5$	15
9	checkers80x80 compressa con $F=10$ e $d=1$	15
10	bridge.bmp originale	16

11	bridge.bmp compressa con F=10 e d=7	16
12	bridge.bmp compressa con F=7 e d=5	16
13	bridge.bmp compressa con F=10 e d=1	16
14	c.bmp originale	17
15	c.bmp compressa con F=10 e d=7	17
16	c.bmp compressa con F=7 e d=5	17
17	c.bmp compressa con F=10 e d=1	17
18	deer.bmp originale	18
19	deer.bmp compressa con F=10 e d=7	18
20	deer.bmp compressa con F=7 e d=5	18
21	deer.bmp compressa con F=10 e d=1	18

Elenco delle tabelle

1	Risultati di DCT_comparison.py su delle matrici di benchmark.	4
---	---	---

1 Introduzione

Questo progetto si divide in due task principali:

- Comparison DCT2
- Image Compressor

1.1 Comparison DCT2

Il primo task prevede lo sviluppo di un metodo che applica la DCT2 (Discrete Cosine Transform in due dimensioni). Una volta sviluppato il metodo, esso va confrontato in termini di performance con un metodo già presente in una libreria dell'ambiente utilizzato.

Nella specifica del progetto viene richiesto che la libreria da utilizzare sia Open Source e che implementi la versione fast (FFT) della DCT2.

Infine vanno forniti degli array bidimensionali $N \times N$, con N crescente, e bisogna riportare le tempistiche al variare di N dei due algoritmi tramite un grafico in scala semilogaritmica sulle ordinate.

1.2 Image Compressor

Il secondo task prevede lo sviluppo di un'interfaccia grafica che permetta di applicare DCT2 e IDCT2 (Inverse DCT2) ad un'immagine scelta per ottenere una versione compressa dell'immagine in input.

In particolare la GUI dovrà permettere all'utente di inserire un'immagine in formato .bmp e dargli la possibilità di inserire due interi come parametri:

- F : questo numero, intero positivo, rappresenta l'ampiezza dei blocchi in cui verrà divisa l'immagine e su cui si effettuerà la DCT2.
- d : questo numero, incluso tra 0 e $(2F - 2)$, rappresenta il taglio delle frequenze che verrà applicato ai blocchi dopo l'applicazione della DCT2.

Dopo che l'utente avrà inserito i dati necessari, in background verranno fatte diverse operazioni che alterano l'immagine, producendone una compressa. In particolare:

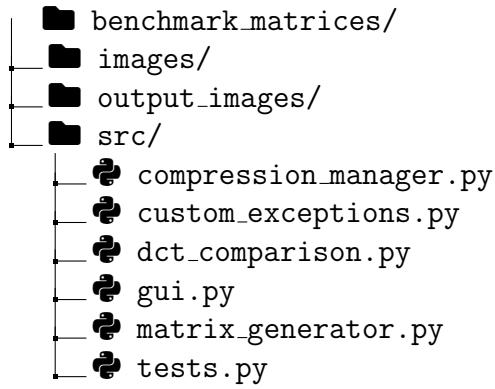
- Viene divisa l'immagine in blocchi $F \times F$.
- Si applica la DCT2 della libreria Open Source ai blocchi.
- Si eliminano le frequenze $c_{i,j}$ ottenute dalla DCT2 tali che $i + j > d$.
- Si applica l'IDCT2 (Operazione inversa della DCT2) alle frequenze rimanenti.
- Si ricompone l'immagine originale con i blocchi trasformati.

2 Struttura del progetto

Per lo sviluppo del progetto è stato utilizzato il linguaggio Python (🐍), utilizzando delle librerie open source ottimizzate per la gestione delle varie task:

- NumPy [1] per la gestione di vettori, matrici e operazioni tra di essi,
- SciPy [2] per le implementazioni della DCT tramite Fast Fourier Transform (FFT),
- Tkinter [3] per realizzare una interfaccia grafica semplice e intuitiva,
- Imageio [4] per caricare le immagini e convertirle in un formato adatto alla compressione,
- Matplotlib [5] per realizzare i grafici utilizzati in questa relazione.

Il progetto sviluppato presenta la seguente struttura:



La repository è visualizzabile in maniera completa su GitHub al seguente link [🔗](#) - ImageCompressor.

3 Prima parte - DCT comparison

La prima parte del progetto prevede l'implementazione del metodo di DCT2 tramite le nozioni viste a lezione, confrontando i risultati di essa con la DCT2 di SciPy, che utilizza la Fast Fourier Transform, passando da una complessità computazionale di $O(N^3)$ con la DCT2 da noi implementata, ad una complessità di $O(N^2 \log(N))$ con la DCT2 FFT.

Per fare ciò, abbiamo creato delle matrici quadrate dalle dimensioni variabili e le abbiamo riempite con dei valori casuali, per poi testare i due metodi appena citati.

3.1 Custom DCT

Implementazione del metodo `custom_dct`

```
1 import numpy as np
2
3 def custom_dct(func_array, D = None):
4     n = len(func_array)
5     if D is None:
6         D = _compute_d(n)
7     coeff_array = np.dot(D, func_array)
8     return coeff_array
9
10 def custom_dct2(func_array_2d):
11     coef_array_2d = np.copy(func_array_2d)
12     N = len(func_array_2d)
13     D = _compute_d(N)
14     for j in range(N):
15         coef_array_2d[:, j] = custom_dct(coef_array_2d[:, j], D)
16     for j in range(N):
17         coef_array_2d[j, :] = custom_dct(coef_array_2d[j, :], D)
18     return coef_array_2d
19
20 def _compute_d(n):
21     alpha_array = np.zeros(n)
22     alpha_array[0] = 1.0 / np.sqrt(n)
23     alpha_array[1:] = np.sqrt(2.0 / n)
24     D = np.empty([n, n])
25     for i in range(n):
26         for j in range(n):
27             D[i][j] =
28                 alpha_array[i] * np.cos(i * np.pi * (2*j+1) / (2*n))
29     return D
```

La DCT 1-dimensionale ha una complessità computazionale di $O(N^2)$, infatti necessita di un doppio ciclo for per calcolare la matrice D, impiegando $O(N^2)$ operazioni, e un prodotto matrice - vettore `np.dot(D, func_array)`, anche esso con un costo di $O(N^2)$.

La DCT 2-dimensionale invece presenta una complessità computazionale di $O(N^3)$. In questo caso, viene calcolata la matrice D una sola volta ($O(N^2)$), viene poi effettuata la DCT su colonne (costo di una `dct1d` sulla colonna $O(N^2) \cdot N$ colonne = $O(N^3)$) e poi una DCT sulle righe (anche essa $O(N^3)$).

3.2 DCT con FFT - SciPy

Implementazione del metodo `scipy_dct_fft`

```
1 from scipy.fft import dct, dctn
2
3 def scipy_dct_fft(func_array):
4     return dct(func_array, type=2, norm="ortho")
5
6 def scipy_dct2_fft(func_array):
7     return dctn(func_array, type=2, norm="ortho")
```

Sono implementate diverse forme di DCT nella libreria fornita da SciPy, quella scelta è la medesima utilizzata nella versione custom che utilizza la norma “ortho”

$$y_k = 2 \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi k (2n+1)}{2N}\right)$$

La DCT 1-dimensionale con fft presenta una complessità di $O(N(\log N))$.

In breve, trasforma il segnale in modo che la DCT diventi equivalente a una DFT e opera tramite un meccanismo di `divide et impera` per riuscire a risolverla in un tempo minore tramite FFT.

La DCT 2-dimensionale invece presenta una complessità computazionale di $O(N^2(\log N))$, in quanto viene effettuata internamente la dct per tutti gli assi della matrice passata (quindi sia per righe che per colonne).

3.3 Analisi dei risultati

I risultati ottenuti sono i seguenti:

Dimensione matrice	Tempo DCT2 - Custom (s)	Tempo DCT2 - Scipy-FFT (s)
8x8	9.280e-05	0.0001
16x16	0.0002	2.910e-05
32x32	0.0007	3.090e-05
64x64	0.0028	4.130e-05
128x128	0.0113	0.00012
256x256	0.0447	0.00034
512x512	0.1935	0.00251
1024x1024	0.8718	0.00989
2048x2048	4.062	0.00618

Tabella 1: Risultati di `DCT_comparison.py` su delle matrici di benchmark.

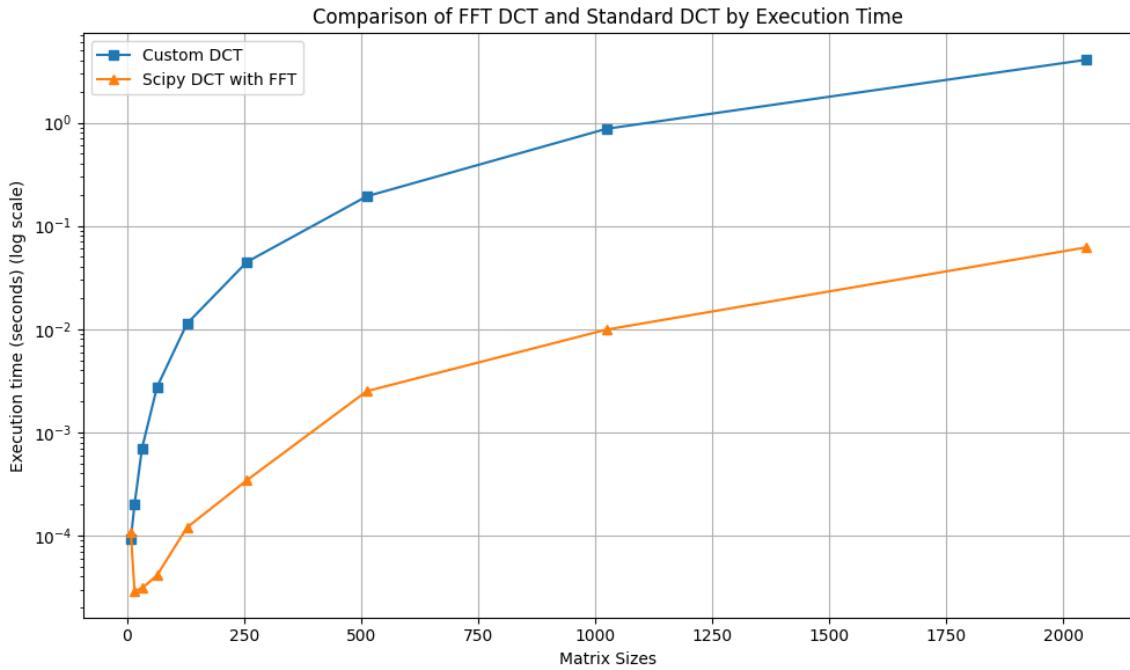


Figura 1: Confronto sui tempi di esecuzione di DCT2 - Custom e DCT2 - FFT

Possiamo notare che la DCT che utilizza la fast fourier transform presenta un andamento abbastanza irregolare inizialmente, per poi stabilizzarsi al crescere delle dimensioni delle matrici. È evidente come la DCT vista a lezione sia più lenta rispetto alla versione ottimizzata, in quanto impiega molto più tempo rispetto alla versione ottimizzata fornita dalla libreria SciPy.

In applicazioni reali la differenza si fa sentire, basta vedere che per la matrice 1024×1024 la versione custom è 88 volte più lenta, mentre per la matrice 2048×2048 è circa 600 volte più lenta.

4 Seconda parte - Image Compressor

La seconda parte del progetto prevede lo sviluppo di un compressore di immagini. Un utente deve poterci interagire tramite un’interfaccia grafica (GUI), inserire un’immagine ed i dati per la compressione (gli interi F e d già menzionati) e osservare il risultato (l’immagine compressa).

Per realizzare l’interfaccia grafica della seconda parte del progetto è stata utilizzata la libreria Tkinter [3], mentre le funzioni DCT2 ed IDCT2 sono implementate nella libreria di Scipy.

4.1 Image Compressor - Interfaccia Grafica

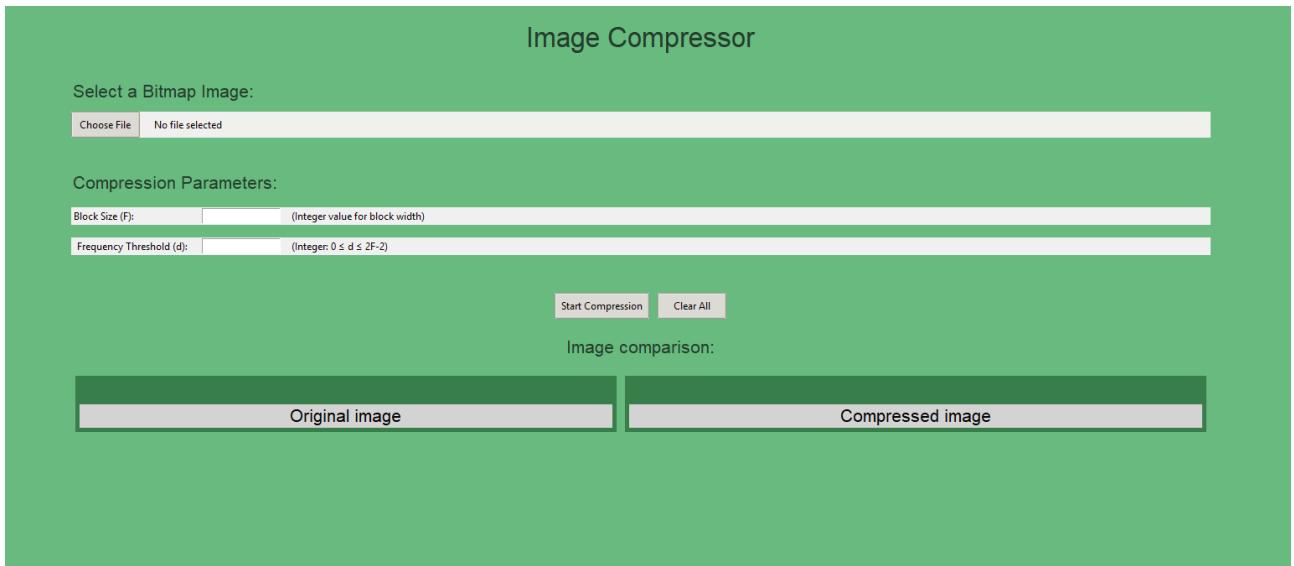


Figura 2: Immagine della GUI di Image Compressor

Questa GUI permette di comprimere le immagini su richiesta dell’utente. In particolare, possono essere inserite immagini in formato .bmp.

Per altri formati di immagine non è prevista la compressione, siccome questo progetto si concentra sulla compressione di immagini in scala di grigi.

Dall’immagine 2 si può vedere che sono presenti nella GUI:

- Gli input che l’utente può utilizzare per inserire gli elementi necessari alla compressione.
- Un tasto *Start Compression* che permette di iniziare il processo di compressione.
- Un tasto *Clear All* che svuota i campi inseriti dall’utente e i due riquadri presenti sotto, riportandoli allo stato in figura.
- Due riquadri che permettono all’utente di visualizzare l’immagine prima e dopo il processo di compressione.

Durante la compressione l’utente potrà visualizzare tramite un messaggio a schermo gli interi F , d e il percorso dell’immagine. La funzione principale del messaggio è far intuire all’utente che il processo di compressione è iniziato con i dati inseriti e non ci sono stati errori.



Figura 3: Immagine di un ponte, prima e dopo la compressione con $F=30$ e $d=1$

Allo stesso modo, se si sono verificati errori, come l'inserimento di un valore di d al di fuori dell'intervallo $[0, 2F - 2]$, allora verrà visualizzato a schermo un messaggio di errore.

4.2 Algoritmo di compressione

In seguito viene presentato l'algoritmo usato per la compressione.

```

1 import warnings
2 import imageio.v3 as iio
3 import numpy as np
4 import os
5 from custom_exceptions import InvalidBlockSizeError,
6     InvalidFrequenciesNumberError
7
8
9 OUTPUT_FOLDER = "output_images"
10
11 def compression(image_path, block_size, frequencies_cut):
12     # Check on the frequencies_cut value
13     if(frequencies_cut < 0) or (frequencies_cut > block_size*2 - 2):
14         warnings.warn("Error: frequencies_cut value is not valid")
15         raise InvalidFrequenciesNumberError
16
17     # Open the image in a grayscale format
18     image = iio.imread(image_path, pilmode='L')
19     rows, columns = image.shape
20     # Calculate the number of blocks in the original image
21     blocks_per_row = rows // block_size
22     blocks_per_column = columns // block_size
23     if(blocks_per_column <= 0 or blocks_per_row <= 0):
24         warnings.warn("Error: image block ")
25         raise InvalidBlockSizeError
26     # Cut the remaining elements that are not part of a block

```

```

27     image = image[:blocks_per_row * block_size, :blocks_per_column *
28         block_size]
29
30     # Split the image into blocks of size block_size x block_size
31     # Initially, it reshapes the image into (blocks_per_row, block_size,
32     # remaining_blocks, block_size)
33     # Then, it swaps axes to group block dimensions together
34     # Lastly, it reshapes the image into (total_blocks, block_size,
35     # block_size)
36     # for example, having a 160x160 image with a block size of 8, it
37     # will result in a three dimensional
38     # vector split into (400 blocks, 8, 8)
39     image = image.reshape(blocks_per_row, block_size, -1, block_size).
40     swapaxes(1, 2).reshape(-1, block_size, block_size)
41
42     image_new = []
43     for block in image:
44         # Perform scipy dct2 on each block
45         coef_matrix = scipy_dct2_fft(block)
46         coef_matrix = np.array(coef_matrix)
47         coef_matrix_cut = np.zeros(shape=(block_size, block_size))
48
49         # Perform frequencies cut using given values
50         for k in range(block_size):
51             for l in range(block_size):
52                 if (k + l) < frequencies_cut:
53                     coef_matrix_cut[k, l] = coef_matrix[k, l]
54
55         # Perform scipy idct2 on the matrix obtained combining the
56         # operations above
57         ff = scipy_idct2_fft(coef_matrix_cut)
58         ff = np.array(ff)
59         # Round the elements of the block, clipping the values inside
60         # the (0, 255) scale
61         ff = np.around(ff, 0)
62         ff = np.clip(ff, 0, 255)
63         image_new.append(ff)
64
65     image_new = np.array(image_new)

# Rebuild the compressed image by reshaping the 3d vector:
# Initially, it reshapes blocks back into grid structure (
# blocks_per_row, blocks_per_column, block_size, block_size)
# Then, it rearranges dimensions to group pixels that belong to the
# same image row
# From (blocks_per_row, blocks_per_column, block_size, block_size)
# To (blocks_per_row, block_size, blocks_per_column, block_size)
# Finally, it reshapes to final 2D image by combining block
# dimensions: (height, width) where height = blocks_per_row *
# block_size

```

```

66
67     blocks_grid = image_new.reshape(blocks_per_row, blocks_per_column,
68                                     block_size, block_size)
69     blocks_rearranged = blocks_grid.transpose(0, 2, 1, 3)
70     final_image = blocks_rearranged.reshape(
71         blocks_per_row * block_size,
72         blocks_per_column * block_size
73     )
74
75     # Convert matrix data to uint for conversion issues
76     final_image = final_image.astype(np.uint8)
77
78     # Saves the compressed image in the output folder
79     os.makedirs(OUTPUT_FOLDER, exist_ok=True)
80     file_name = os.path.splitext(os.path.basename(image_path))[0]
81     output_path = os.path.join(OUTPUT_FOLDER, f"compressed_{file_name}."
82                                "bmp")
83     iio.imwrite(output_path, final_image)

return output_path

```

4.3 Suddivisione dell'immagine in blocchi

Prendo come esempio una matrice 6×6 che rappresenta l'immagine, e suppongo di avere come valore $F = 2$ (`block_size`) :

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

I colori rappresentano i blocchi 2×2 che vogliamo estrarre.

4.3.1 Step 1: Primo reshape

```
image.reshape(blocks_per_row, block_size, -1, block_size)
```

In questo caso: `reshape(3, 2, -1, 2)` crea un array 4D di dimensioni $(3, 2, 3, 2)$, dove:

- **Dimensione 0 (3):** Righe di blocchi
- **Dimensione 1 (2):** Righe all'interno di ogni blocco
- **Dimensione 2 (3):** Colonne di blocchi
- **Dimensione 3 (2):** Colonne all'interno di ogni blocco

$\text{image}[i][j][k][l] = \text{elemento alla posizione } (i \cdot 2 + j, k \cdot 2 + l)$

Guardando solo il primo livello della prima dimensione (`image[0]`), avremo come prima riga di blocchi

$j = 0 :$	$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$	$\begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$
$j = 1 :$	$\begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$	$\begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$

4.3.2 Step 2: Swap axes

`swapaxes(1, 2)`

Scambia le dimensioni 1 e 2, trasformando $(3, 2, 3, 2)$ in $(3, 3, 2, 2)$.

Nuovo significato delle dimensioni:

- **Dimensione 0 (3):** Righe di blocchi
- **Dimensione 1 (3):** Colonne di blocchi
- **Dimensione 2 (2):** Righe all'interno di ogni blocco
- **Dimensione 3 (2):** Colonne all'interno di ogni blocco

Risultato dopo `swapaxes`: ogni `array[i][j]` è un blocco 2×2 :

indici	$j = 0$	$j = 1$	$j = 2$
$i = 0$	$\begin{pmatrix} 1 & 2 \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 3 & 4 \\ 9 & 10 \end{pmatrix}$	$\begin{pmatrix} 5 & 6 \\ 11 & 12 \end{pmatrix}$
$i = 1$	$\begin{pmatrix} 13 & 14 \\ 19 & 20 \end{pmatrix}$	$\begin{pmatrix} 15 & 16 \\ 21 & 22 \end{pmatrix}$	$\begin{pmatrix} 17 & 18 \\ 23 & 24 \end{pmatrix}$
$i = 2$	$\begin{pmatrix} 25 & 26 \\ 31 & 32 \end{pmatrix}$	$\begin{pmatrix} 27 & 28 \\ 33 & 34 \end{pmatrix}$	$\begin{pmatrix} 29 & 30 \\ 35 & 36 \end{pmatrix}$

4.3.3 Step 3: Reshape finale

Operazione: `.reshape(-1, block_size, block_size)`

In questo caso: `reshape(-1, 2, 2) = reshape(9, 2, 2)`

Questo appiattisce le prime due dimensioni, creando un array di 9 blocchi 2×2 .

4.3.4 Vettore di blocchi finale:

$$\text{risultato} = \begin{bmatrix} \begin{pmatrix} 1 & 2 \\ 7 & 8 \end{pmatrix} \\ \begin{pmatrix} 3 & 4 \\ 9 & 10 \end{pmatrix} \\ \begin{pmatrix} 5 & 6 \\ 11 & 12 \end{pmatrix} \\ \begin{pmatrix} 13 & 14 \\ 19 & 20 \end{pmatrix} \\ \begin{pmatrix} 15 & 16 \\ 21 & 22 \end{pmatrix} \\ \begin{pmatrix} 17 & 18 \\ 23 & 24 \end{pmatrix} \\ \begin{pmatrix} 25 & 26 \\ 31 & 32 \end{pmatrix} \\ \begin{pmatrix} 27 & 28 \\ 33 & 34 \end{pmatrix} \\ \begin{pmatrix} 29 & 30 \\ 35 & 36 \end{pmatrix} \end{bmatrix}$$

Un array contenente blocchi di dimensione 2×2 , pronti per essere compressi.

4.4 Compressione

Una volta che abbiamo a disposizione il vettore con i blocchi, possiamo applicare la DCT.

La DCT trasforma i valori contenuti nei blocchi (che corrispondono ai valori dei pixel dell'immagine in una scala di grigi) in delle frequenze.

Viene prima applicata la DCT fornita da SciPy, che effettua prima la dct per righe e poi la dct per colonne, ottenendo un blocco che contiene dunque i coefficienti delle frequenze.

Le frequenze dal valore maggiore si trovano nelle posizioni con gli indici più bassi di ogni blocco, e contengono la maggior parte delle informazioni.

Per questo motivo è possibile tagliare le frequenze situate ad indici più alti (a seconda del valore **d** scelto), che contengono meno informazioni, in modo da ottenere un risultato simile senza perdere troppa qualità, andando a risparmiare memoria.

Ad esempio, per un blocco 16×16 casuale preso da un'immagine di prova, si sono ottenuti i seguenti valori:

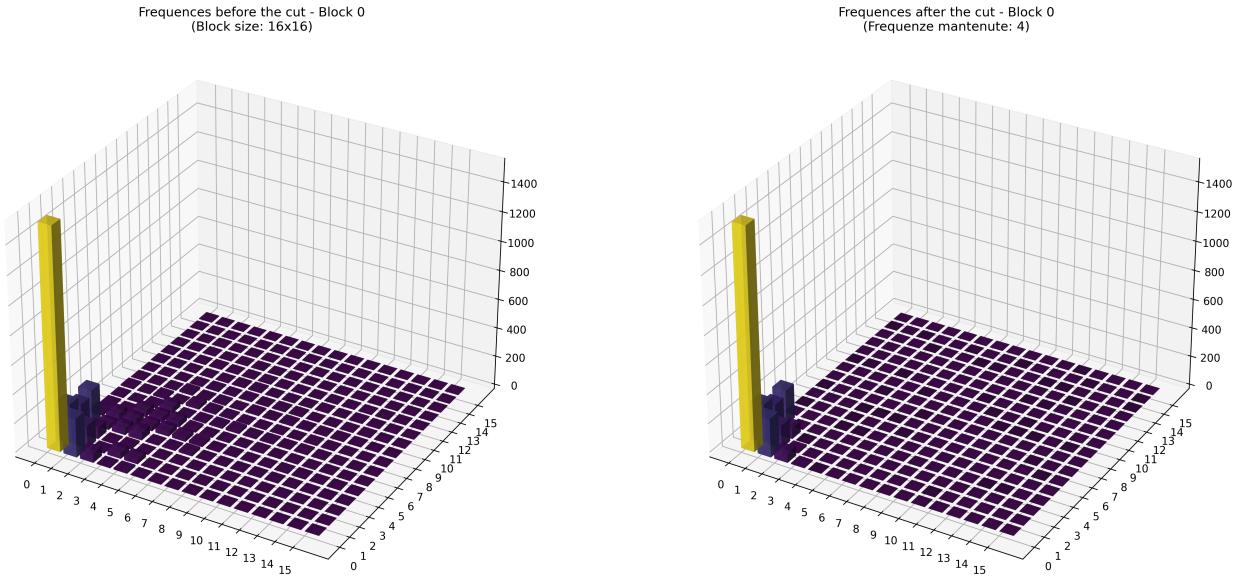


Figura 4: Coefficienti delle frequenze prima del taglio in un blocco 16×16 con $d = 4$

Figura 5: Coefficienti delle frequenze dopo il taglio in un blocco 16×16 con $d = 4$

4.5 Decompressione

Dopo aver effettuato il taglio, è possibile decomprimere i valori e tornare ai valori dei coefficienti delle frequenze che rappresentano la scala di grigi dell'immagine originale.

Per farlo, è sufficiente applicare la `idctn` al vettore, effettuando i controlli affinché tutti i valori siano poi compresi tra 0 e 255 (scala di grigi).

4.6 Ricomposizione

Una volta riottenuto un vettore di blocchi con i nuovi valori, si procede ricostruendo l'immagine originale.

4.6.1 Step 1: Primo reshape

`image_new.reshape(blocks_per_row, blocks_per_column, block_size, block_size)` Nel nostro caso: `reshape(3, 3, 2, 2)`

Questo trasforma l'array (9, 2, 2) in un array 4D (3, 3, 2, 2), dove:

- **Dimensione 0 (3):** Righe di blocchi nella griglia
- **Dimensione 1 (3):** Colonne di blocchi nella griglia
- **Dimensione 2 (2):** Righe all'interno di ogni blocco
- **Dimensione 3 (2):** Colonne all'interno di ogni blocco

indici	$j = 0$	$j = 1$	$j = 2$
$i = 0$	$\begin{pmatrix} 1 & 2 \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 3 & 4 \\ 9 & 10 \end{pmatrix}$	$\begin{pmatrix} 5 & 6 \\ 11 & 12 \end{pmatrix}$
$i = 1$	$\begin{pmatrix} 13 & 14 \\ 19 & 20 \end{pmatrix}$	$\begin{pmatrix} 15 & 16 \\ 21 & 22 \end{pmatrix}$	$\begin{pmatrix} 17 & 18 \\ 23 & 24 \end{pmatrix}$
$i = 2$	$\begin{pmatrix} 25 & 26 \\ 31 & 32 \end{pmatrix}$	$\begin{pmatrix} 27 & 28 \\ 33 & 34 \end{pmatrix}$	$\begin{pmatrix} 29 & 30 \\ 35 & 36 \end{pmatrix}$

4.6.2 Step 2: Transpose

```
image_new.transpose(0, 2, 1, 3)
```

Questo riordina le dimensioni da $(0, 1, 2, 3)$ a $(0, 2, 1, 3)$, trasformando $(3, 3, 2, 2)$ in $(3, 2, 3, 2)$, dove:

- **Dimensione 0 (3):** Righe di blocchi
- **Dimensione 1 (2):** Righe all'interno di ogni blocco
- **Dimensione 2 (3):** Colonne di blocchi
- **Dimensione 3 (2):** Colonne all'interno di ogni blocco

Visualizzazione del transpose:

Per ogni riga di blocchi i , e per ogni riga interna k :

Prima riga di blocchi ($i=0$): $k = 0 : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$
 $k = 1 : \begin{pmatrix} 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix}$

Seconda riga di blocchi ($i=1$): $k = 0 : \begin{pmatrix} 13 & 14 & 15 & 16 & 17 & 18 \end{pmatrix}$
 $k = 1 : \begin{pmatrix} 19 & 20 & 21 & 22 & 23 & 24 \end{pmatrix}$

Terza riga di blocchi ($i=2$): $k = 0 : \begin{pmatrix} 25 & 26 & 27 & 28 & 29 & 30 \end{pmatrix}$
 $k = 1 : \begin{pmatrix} 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix}$

4.6.3 Step 3: Reshape finale

```
image_new.reshape(blocks_per_row * block_size, blocks_per_column * block_size)
```

Nel nostro caso $\text{reshape}(3 * 2, 3 * 2) = \text{reshape}(6, 6)$, rendendo l'array $(3, 2, 3, 2)$ una matrice $(6, 6)$.

4.7 Risultato finale - Matrice ricomposta:

$$\text{final_image} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix}$$

dove ovviamente i valori saranno diversi da quelli iniziali.

5 Conclusion - Analisi dei risultati

Infine abbiamo deciso di controllare gli effetti della compressione/decompressione su diverse immagini. Per controllare se la compressione e la decompressione funzionassero correttamente sono stati fatti dei test con delle immagini in scala di grigi.

Di seguito verranno mostrati alcuni di questi test e i risultati da noi ottenuti.

I test che verranno mostrati sono stati fatti su:

- Un'immagine `bridge`, quale 10 di dimensione 2749×4049 .
- Un'immagine `C`, quale 14 di dimensione 100×100 .
- Un'immagine `checkers`, quale 6 di dimensione 80×80 .
- Un'immagine `deer`, quale 18 di dimensione 1011×661

I test sono stati fatti con valori di F e d differenti per osservare bene il cambiamento nella compressione:

- $F = 10, d = 7$.
- $F = 7, d = 5$.
- $F = 10, d = 1$.

Le immagini originali sono visibili nelle figure 6, 10 e 14.

Il motivo per cui sono stati scelti quei numeri per F e d è per vedere come cambia la compressione delle immagini con valori diversi.

5.1 Immagine checkers

L'immagine a scacchiera perde poco in qualità con $F=10$, come visibile nelle figure 7, 9, questo perché dividendola in blocchi da 10 pixel non vengono tagliati fuori pezzi dell'immagine originale, invece per $F=7$ alcune informazioni vengono perse e la DCT ha un comportamento peggiore rendendo visibili artefatti lungo i bordi dei quadrati, come visibile nella figura 8.

Cambiare il taglio delle frequenze invece non sembra cambiare l'aspetto dell'immagine, infatti è difficile notare la differenza tra 7 e 9.

Invece, per quanto riguarda le dimensioni, l'immagine originale occupa 7,30 KB ed anche 7 e 9, l'unica che guadagna un po' di spazio è 8, però con una considerevole perdita di informazione.

Questo accade perché i frequenti cambiamenti repentini da bianco a nero necessitano di più componenti difficilmente rappresentabili con pochi coefficienti DCT, specialmente in blocchi di dimensione limitata.

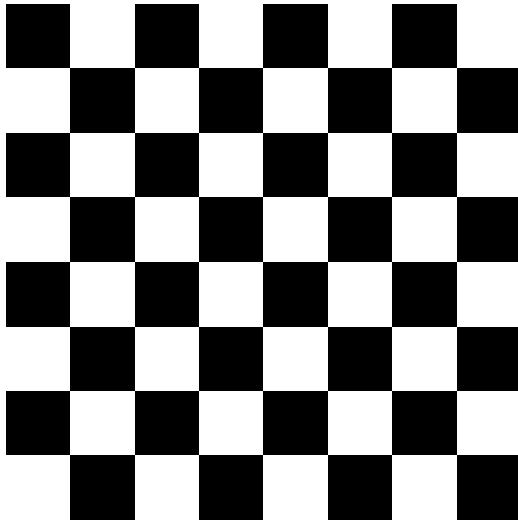


Figura 6: checkers80x80 originale

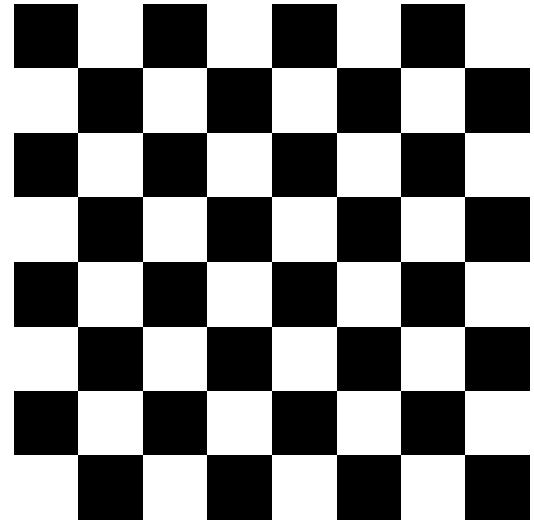


Figura 7: checkers80x80 compressa con $F=10$ e $d=7$

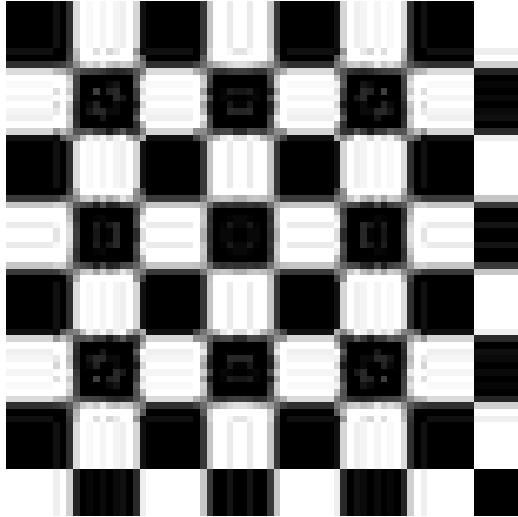


Figura 8: checkers80x80 compressa con $F=7$ e $d=5$

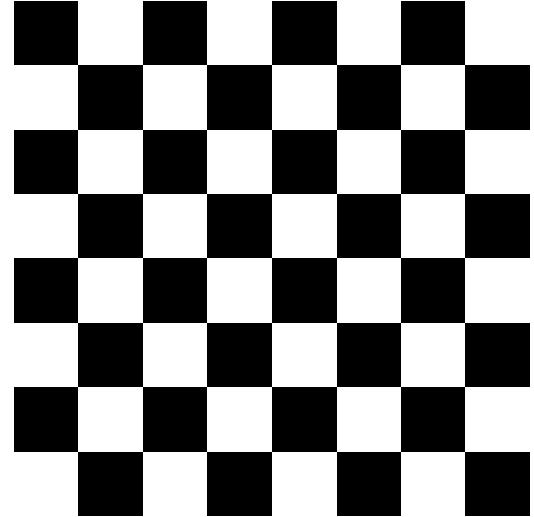


Figura 9: checkers80x80 compressa con $F=10$ e $d=1$

5.2 Immagine bridge

L’immagine del ponte 10 ha una risoluzione molto più alta rispetto alle altre, per questo motivo gli effetti della compressione con blocchi piccoli (formati da 10 o 7 pixel) hanno un effetto a prima vista poco impattante, come si può vedere nelle figure 11, 12 e 13.

Invece, come visibile nella figura utilizzata per mostrare la GUI [3] utilizzando blocchi più grandi, con $F=30$, e una soglia d molto bassa (1), la differenza tra le immagini è molto più evidente, e si può notare meglio la perdita di qualità rispetto alle altre immagini. Nonostante il rapporto tra F e d sia lo stesso utilizzato per l’immagine checkers 9, la risoluzione del risultato compresso è peggiore. Ciò è dovuto al fatto che l’immagine presenta una sfumatura di grigi e non dei valori fissi di bianco e nero, rendendo visibile una piccola perdita di qualità rispetto all’immagine originale, a differenza dell’immagine 9 che rimane uguale all’originale.

La dimensione delle immagini, in questo caso, cambia sostanzialmente rispetto all’originale.

Con l'immagine originale che arriva a pesare 19,3 MB, mentre le sue versioni compresse pesano tutte 10,5 MB, risparmiando quasi metà dello spazio occupato.



Figura 10: `bridge.bmp` originale



Figura 11: `bridge.bmp` compressa con $F=10$ e $d=7$



Figura 12: `bridge.bmp` compressa con $F=7$ e $d=5$



Figura 13: `bridge.bmp` compressa con $F=10$ e $d=1$

5.3 Immagine C

I risultati per questa immagine sono una via di mezzo tra le due immagini precedenti, senza avere una struttura a scacchiera come la prima immagine analizzata, ma nemmeno avendo un'alta risoluzione come l'immagine del ponte.

Dalle immagini 15, 16 e 17 si può osservare che in questo caso le diverse compressioni hanno effetti differenti, arrivando ad una performance disastrosa nel caso di 17, mentre si notano lievi differenze in 8 ed ancora minori in 15. Anche in questo caso la compressione ha un buon effetto sullo spazio di memoria occupato, passando da 29,4 KB a 10,8 KB per tutti i casi.

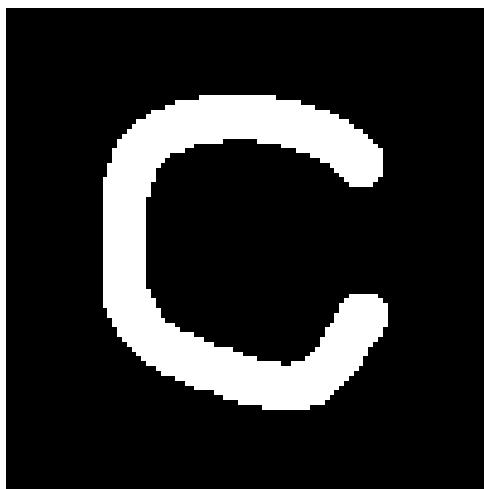


Figura 14: c.bmp originale

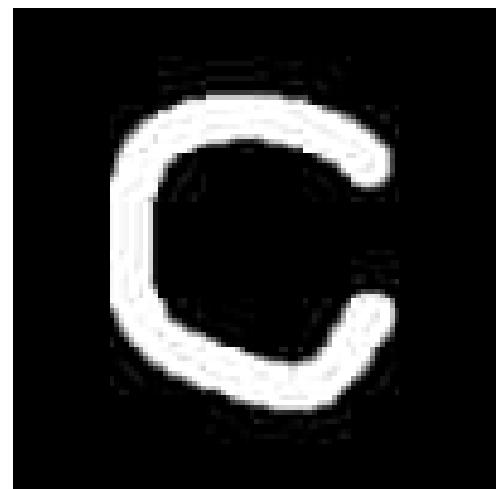


Figura 15: c.bmp compressa con $F=10$ e $d=7$

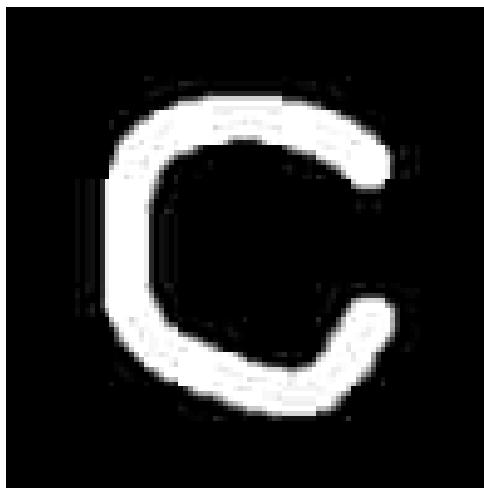


Figura 16: c.bmp compressa con $F=7$ e $d=5$

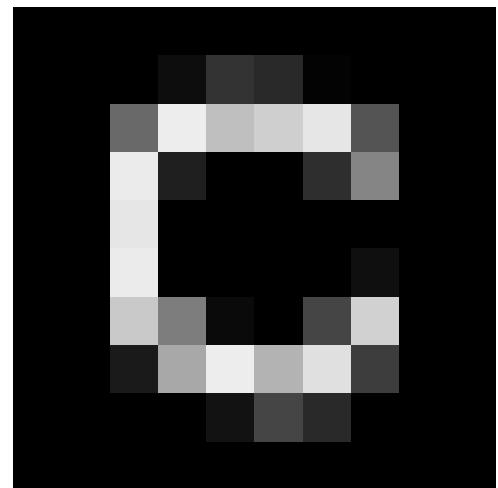


Figura 17: c.bmp compressa con $F=10$ e $d=1$

5.4 Immagine deer

Dalle figure 18, 19, 20 e 21 si può osservare che la procedura di compressione/decompressione si comporta in maniera molto simile all'immagine del ponte. In particolare è notevole la differenza di spazio occupato dalle immagini compresse 19 e 20, passando da 1,960 KB a 649 KB, ma con una differenza quasi impercettibile ad occhio nudo.



Figura 18: `deer.bmp` originale



Figura 19: `deer.bmp` compressa con $F=10$ e
 $d=7$



Figura 20: `deer.bmp` compressa con $F=7$ e $d=5$

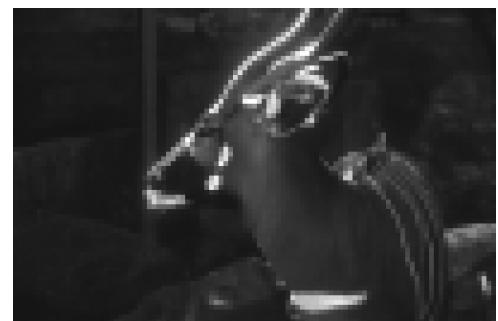


Figura 21: `deer.bmp` compressa con $F=10$ e
 $d=1$

Bibliografia

- [1] NumPy-team. «Numpy package.» (2025), indirizzo: <https://numpy.org/> (visitato il giorno 09/06/2025) (cit. a p. 2).
- [2] SciPy-team. «SciPy package.» (2025), indirizzo: <https://scipy.org/> (visitato il giorno 09/06/2025) (cit. a p. 2).
- [3] Python. «tkinter — Python interface to Tcl/Tk.» (2025), indirizzo: <https://docs.python.org/3/library/tkinter.html> (visitato il giorno 09/06/2025) (cit. alle pp. 2, 6).
- [4] ImageIO-team. «ImageIO.» (2025), indirizzo: <https://imageio.readthedocs.io/en/stable/> (visitato il giorno 09/06/2025) (cit. a p. 2).
- [5] The-Matplotlib-development-team. «Matplotlib: Visualization with Python.» (2025), indirizzo: <https://matplotlib.org/> (visitato il giorno 09/06/2025) (cit. a p. 2).