

Progetto di Metodi del Calcolo Scientifico

Relazione progetto 2

Autori:

Federica Ratti, 886158
Nicolò Molteni, 933190

{f.ratti15, n.molteni6}@campus.unimib.it

Corso di Laurea Magistrale in Informatica
Università di Milano-Bicocca
Anno Accademico 2024-2025

Indice

Introduzione	4
1 La libreria	7
1.1 Il linguaggio e le librerie utilizzate	7
1.2 Il formato Windows Bitmap (BMP)	8
1.3 La directory del progetto	8
1.3.1 La classe Calcolous	9
1.3.2 Il modulo compression_utils	11
1.3.3 Il modulo file_utils	11
1.3.4 Il modulo plot_utils	12
1.3.5 La classe Gui	12
2 Confronto tra DCT2	13
3 La compressione delle immagini	15
3.0.1 Compressione alto contrasto	18
3.0.2 Compressione basso contrasto	24
3.1 Conclusioni	28

Introduzione

L'elaborato ha il compito di osservare il comportamento della Discrete Cosine Transform, DCT, una trasformazione matematica che permette di rappresentare una sequenza di valori finiti come una somma di funzioni coseno con frequenza e ampiezza variabile. La DCT si differenzia dalla trasformata di Fourier perché non sfrutta una combinazione di funzioni seno e coseno. All'interno del progetto è stata implementata una Discrete Cosine Transform ed è stata confrontata con una DCT implementata da una libreria. Successivamente, dopo un confronto in termini di correttezza e complessità la Discrete Cosine Transform è stata applicata ad una serie di immagini. Quest'ultime sono state suddivise in blocchi di dimensione $F \times F$ ed è stata applicata una compressione andando ad eliminare alcune frequenze secondo il parametro d , $0 \leq d \leq 2F - 2$. Infine è stato fatto un confronto tra i vari risultati ottenuti e le immagini originali non compresse.

Capitolo 1

La libreria

All'interno del progetto sono state sviluppate due librerie differenti: la prima implementa la Discrete Cosine Transform e la Inverse Discrete Cosine Transform. La seconda implementa tutte le logiche necessarie alle funzionalità implementate dalla Graphic User Interface, GUI, ad esempio: la suddivisione dell'immagine in blocchi di dimensione $F \times F$, la compressione tramite DCT, la ricostruzione dell'immagine compressa tramite IDCT e la generazione di grafici utili al confronto dei risultati ottenuti.

1.1 Il linguaggio e le librerie utilizzate

Il linguaggio scelto per lo sviluppo è Python, un linguaggio open source largamente adottato anche in ambito accademico e scientifico. Infatti, la sua semplicità sintattica e la grande disponibilità di librerie lo rendono un linguaggio ideale per lo sviluppo rapido di algoritmi numerici.

Per l'implementazione della Discrete Cosine Transform e la gestione dei dati di input, sono state impiegate tre librerie:

- **Pillow**, utilizzata per la lettura e la manipolazione di immagini in diversi formati tra cui il BMP. All'interno del progetto questa libreria consente di caricare l'immagine dalla memoria e convertirla in una matrice 2D di valori di intensità ed eventualmente eseguire una trasformazione dello spazio colore in scala di grigi.
- **Numpy**, impiegata per la rappresentazione delle immagini sotto forma di matrici numeriche e per l'esecuzione di operazioni matematiche elementari, ad esempio: somme, prodotto vettoriale, prodotto scalare, trasformazioni e slicing per ottenere i blocchi di dimensione $F \times F$.
- **Scipy**, utilizzata per confrontare la DCT implementata manualmente con quella fornita dalla libreria. In particolare, il modulo `scipy.fft` implementa le funzioni per il calcolo di DCT e IDCT sfruttando algoritmi ottimizzati. L'implementazione della DCT in SciPy sfrutta algoritmi derivati dalla Fast Fourier Transform, in particolare impiega l'algoritmo split-radix FFT e alcune sue varianti ottimizzate per sequenze reali e simmetriche. La risoluzione del problema con questi algoritmi efficienti consente

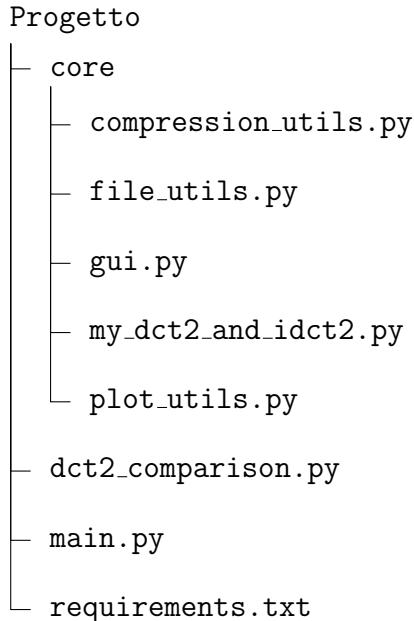
di ridurre la complessità computazionale da $O(n^3)$ a $O(n^2 \log(n))$, con un notevole guadagno in efficienza e conseguentemente in termini temporali.

1.2 Il formato Windows Bitmap (BMP)

Il formato *Windows Bitmap*, BMP, è stato sviluppato da Microsoft per rappresentare le immagini in forma matriciale. Questo formato consente operazioni di lettura e scrittura estremamente rapide, ma richiede una quantità di memoria maggiore rispetto ad altri formati perché non è compresso.

La matrice M che rappresenta l'immagine ha i colonne e j righe ed è costituita da entry M_{ij} che prendono il nome di pixel. Ciascuna entry contiene le informazioni relative al colore, ad esempio: in un'immagine a colori le informazioni corrispondono ai valori dei canali RGB, mentre in un'immagine in scala di grigi ogni entry contiene un valore compreso tra 0 e 255 che indica l'intensità del grigio associata a quel pixel. La profondità di ciascuna entry dipende direttamente dal numero di bit utilizzati per memorizzare il colore del pixel.

1.3 La directory del progetto



Qui sopra è possibile osservare il directory tree della libreria sviluppata. Il progetto è composto dalla cartella **core** che contiene tutte le classi e i metodi per eseguire la DCT2 e IDCT2, la compressione, il caricamento delle immagini dalla memoria e la realizzazione delle immagini e grafici post compressione. Infine sono presenti tre file: il primo, **requirements.txt** viene creato automaticamente e contiene le librerie necessarie per il corretto funzionamento del progetto. Il secondo, **main.py** permette di eseguire la graphic user interface per selezionare un'immagine in formato BMP dalla memoria e comprimerla. Il terzo, **dct2_comparison.py** che permette di eseguire delle comparazioni in termini di tempo di calcolo tra la DCT2 implementata nel progetto e quella della libreria Scipy.

1.3.1 La classe Calcolous

La classe `Calcolous` si trova all'interno del modulo `my_dct2_and_idct2.py` ed implementa il calcolo della DCT monodimensionale, DCT2 e IDCT2 bidimensionali.

```

1 import numpy as np
2 from numpy.typing import NDArray
3
4 class Calcolous:
5
6     @staticmethod
7     def __dct(n: int):
8
9         # creo il vettore alpha basandomi su n
10        alpha_vect = np.zeros(n, dtype=float)
11        alpha_vect[0] = n ** (-0.5)
12        alpha_vect[1:] = n ** (-0.5) * np.sqrt(2.0)
13
14        D = np.zeros((n, n), dtype=float)
15        for k in range(n):
16            for i in range(n):
17                D[k, i] = alpha_vect[k] * np.cos((k) * np.pi * (2 * (i +
18                    1) - 1) / (2 * n))
19
20        return D
21
22    @staticmethod
23    def dct2(f_mat: NDArray[float]):
24
25        n = np.shape(f_mat)[0]
26
27        # calcolo la matrice DCT monodimensionale
28        D = Calcolous.__dct(n)
29        c_mat = f_mat.copy()
30
31        # DCT1 per colonne
32        for j in range(n):
33            c_mat[:, j] = np.dot(D, c_mat[:, j])
34
35        # DCT1 per righe
36        for j in range(n):
37            c_mat[j, :] = np.dot(D, c_mat[j, :])
38
39        return c_mat
40
41    @staticmethod
42    def idct2(c_mat: NDArray[float]):
43
44        n = c_mat.shape[0]
45
46        # calcolo la matrice DCT monodimensionale
47        D = Calcolous.__dct(n)
48        f_mat = c_mat.copy()

```

```

49     # IDCT1 per colonne
50     for j in range(n):
51         f_mat[:, j] = np.dot(D.T, f_mat[:, j])
52
53     # IDCT1 per righe
54     for i in range(n):
55         f_mat[i, :] = np.dot(D.T, f_mat[i, :].T).T
56
57     return f_mat

```

Codice 1.1: Implementazione della classe `Calcolous`.

La funzione `__dct(n: int)` è di tipo privato e può essere utilizzata solo all'interno della classe `Calcolous`. Il metodo calcola la matrice di trasformazione DCT a partire da un vettore `alpha_vect` di n coefficienti α , definiti nel seguente modo:

$$\alpha_0 = \frac{1}{\sqrt{n}}, \quad \alpha_k = \frac{\sqrt{2}}{\sqrt{n}} \quad \text{per } k = 1, \dots, n-1. \quad (1.1)$$

I coefficienti vengono impiegati per riempire la matrice $D \in \mathbb{R}^{n \times n}$ sfruttando la formula di riga 17 definita nel Codice 1.1. Ogni elemento è calcolato tramite il prodotto del corrispondente α_k con la quantità in Formula 1.2. I termini k, i in Formula 1.2 sono gli indici del ciclo `for`.

$$\cos\left(k\pi\left(\frac{2(i+1)-1}{2n}\right)\right) \quad (1.2)$$

La funzione `dct2(f_mat: NDArray[float])` applica la DCT2 alla matrice di campioni `f_mat`. Per prima cosa viene calcolata la matrice di trasformazione DCT monodimensionale $D \in \mathbb{R}^{n \times n}$, dove n è il numero di righe di `f_mat`. La matrice, D , è ottenuta tramite la funzione `__dct()` precedentemente descritta. Successivamente viene sfruttata la DCT monodimensionale in due passaggi per ottenere la DCT2:

1. **Trasformata per colonne:** per ciascuna colonna j di `c_mat` si calcola il prodotto $D \cdot c[:, j]$, dove `c[:, j]` è il vettore della j -esima colonna della matrice.
2. **Trasformata per righe:** per ciascuna riga i della matrice ottenuta al passo precedente si calcola il prodotto $D \cdot c[j, :]$, dove `c[j, :]` è il vettore dell' i -esima riga della matrice.

La funzione restituisce la matrice delle frequenze, `c_mat` di dimensioni $n \times n$, i cui elementi rappresentano i coefficienti di frequenza della trasformata di `f_mat`. I coefficienti a bassa frequenza contengono le informazioni più importanti e si concentrano in alto a sinistra della matrice. I coefficienti ad alta frequenza si distribuiscono verso il basso a destra e contengono informazioni di importanza minore.

La funzione `idct2(c_mat: NDArray[float])` applica la IDCT2 a una matrice delle frequenze. Anche in questo caso, come per l'applicazione della DCT2, viene calcolata la matrice di trasformazione DCT monodimensionale. Successivamente viene sfruttata questa matrice in due passaggi per applicare la IDCT2.

1. **Trasformata per colonne:** per ciascuna colonna j di f_mat si calcola il prodotto $D \cdot f[:, j]$, dove $f[:, j]$ è il vettore della j -esima colonna della matrice.
2. **Trasformata per righe:** per ciascuna riga i della matrice ottenuta al passo precedente si calcola il prodotto $D \cdot f[j, :]$, dove $f[j, :]$ è il vettore dell' i -esima riga della matrice.

La funzione restituisce la matrice dei campionamenti, `f_mat`, di dimensioni $n \times n$. Con i giusti accorgimenti (trasformazione dei valori in interi e spostamento nell'intervallo $[0, 255]$) permette di riottenere i campioni dell'immagine. Se prima di questo processo viene applicato un taglio di frequenze è possibile eseguire una compressione dell'immagine.

1.3.2 Il modulo compression_utils

Il modulo `compression_utils` contiene al suo interno una serie di funzioni utili ad implementare la compressione dell'immagine secondo le modalità descritte nella consegna del progetto. In particolare sono presenti le seguenti funzioni:

- `make_blocks(pixel_matrix: NDArray[float], F)`, data una matrice di campionamenti dell'immagine con dimensione $height(h) \times width(w)$ permette di suddividere tale immagine in piccoli blocchi di dimensione $F \times F$. Se le dimensioni, h, w dell'immagine non sono multipli di F l'avanzo di campioni verrà scartato. Ad esempio: avendo un'immagine di dimensioni 14×15 pixel e volendo creare dei blocchetti da 11 pixel si ottengono 101 pixel di scarto.
- `compress(blocks: list[NDArray[float]], F: int, d: int)`, a ciascun blocco di dimensione F ottenuto con la funzione precedente vengono eseguiti i seguenti passaggi. Primo, è calcolata la DCT2 del blocco, secondo, vengono eliminate o annullate le frequenze più alte sfruttando il parametro d che ha valore compreso in $0 \leq d \leq 2F - 2$. Il processo appena descritto sfrutta gli indici di scorrimento del blocchetto, se la loro somma è maggiore di d allora la frequenza viene annullata. Terzo, viene applicata la IDCT2 per ottenere nuovamente dei campionamenti, infine, sul risultato ottenuto viene applicata la funzione `make_transform`.
- `make_transform(idct2_block: NDArray[float], F: int)`, dopo aver calcolato la IDCT2 può essere chiamata questa funzione che permette di ottenere dei campioni interi che risiedono nell'intervallo $[0, 255]$.
- `make_compressed_img(idct2_blocks: list[NDArray[float]], F: int, image_width: int, image_height: int)`, la funzione permette di ricostruire l'immagine con dimensione $width \times height$, senza i pixel eliminati durante la creazione dei blocchetti.

1.3.3 Il modulo file_utils

Il modulo `file_utils` contiene la funzione `select_file(gui_instance)` che permette di selezionare un'immagine `.bmp` dal file system. Se non si verificano errori in apertura viene mostrata una finestra di dialogo che contiene la dimensione dell'immagine.

1.3.4 Il modulo plot_utils

Il modulo `plot_utils` ha al suo interno due funzioni utili per la realizzazione di alcuni grafici di confronto.

- `make_imgs_plot(img: NDArray[float], compressed_img: NDArray[float], F: int, d: int)`, date le matrici che rappresentano l'immagine compressa e non compressa verranno generati dei plot che mostrano le due immagini per vedere le differenze ottenute. I parametri F e d vengono mostrati sull'immagine compressa.
- `make_pixel_plot(img_var: float, img_block: NDArray[float], compressed_var: float, compressed_block: NDArray[float], index: int, image_width: int, F: int)`, dato il blocchetto di dimensione F dell'immagine originale e il blocchetto dell'immagine compressa viene eseguito un plot che rappresenta il valore dei pixel prima e dopo la compressione. In questo grafico vengono anche mostrate alcune informazioni come: le coordinate del blocchetto e la sua varianza. Le coordinate del blocchetto vengono scelte prendendo la finestra F con maggior varianza all'interno dell'immagine compressa, sfruttando queste coordinate verrà successivamente caricato anche il blocchetto dell'immagine non compressa.

1.3.5 La classe Gui

La classe `Gui` permette la creazione e gestione dell'interfaccia grafica. La Figura 1.1 è un esempio della GUI implementata, è possibile selezionare il file BMP da comprimere, specificare i parametri e avviare la compressione.



Figura 1.1: Esempio della graphic user interface implementata all'interno del progetto. Permette di selezionare il file BMP dal file system, specificare i parametri di compressione ed avviare quest'ultima. Al termine verranno mostrate le immagini, compressa e non, e un grafico che mostra la variazione dei valori dei pixel.

La classe `GUI` contiene al suo interno dei metodi privati che eseguono chiamate alle funzioni contenute all'interno dei moduli introdotti in precedenza. Inoltre esegue dei controlli sui parametri inseriti. Durante la fase di caricamento dell'immagine la GUI non sarà più responsiva, di conseguenza i parametri F e d potranno essere inseriti solamente dopo l'apertura della finestra di dialogo che comunica il corretto caricamento. L'avvio della compressione comporta la generazione automatica dei grafici descritti in `plot_utils`, la funzionalità non è disattivabile.

Capitolo 2

Confronto tra DCT2

Prima di eseguire la compressione delle immagini mediante DCT2 è stato eseguito un confronto tra: DCT2 implementata e DCT2 da libreria Scipy. Il modulo che permette la comparazione è `dct2_comparison`. Al suo interno contiene la classe `DctUtilities` e un piccolo `main` di test.

La classe permette di definire il numero massimo di iterazioni su cui testare le discrete cosine transform, inoltre mette a disposizione i seguenti metodi:

- `test_dct(self, f: NDArray[np.float64]) -> bool`, verifica che la DCT calcolata sulla matrice `f` in input, con la funzione messa a disposizione da Scipy, corrisponda al blocco reference mostrato nella consegna del progetto. Se è presente corrispondenza il metodo restituirà `True`, `False` altrimenti.
- `test_dct2(self, f: NDArray[np.float64]) -> bool`, verifica che la DCT2 calcolata sulla matrice `f` in input, con la funzione messa a disposizione da Scipy, corrisponda al blocco reference mostrato nella consegna del progetto. Se è presente corrispondenza il metodo restituirà `True`, `False` altrimenti.
- `test_implemented_dct2(self, scipy_dct: NDArray[np.float64], my_dct: NDArray[np.float64]) -> bool`, verifica che la DCT o DCT2 implementata da Scipy sia uguale alla DCT o DCT2 osservata a lezione. Se è presente corrispondenza il metodo restituirà `True`, `False` altrimenti.
- `__generate_matrices(self, step=10)`, sfruttando il parametro `n` impostato durante la creazione di un oggetto della classe `DctUtilities` è possibile generare matrici quadrate di dimensione variabile in base al parametro `step`. Le matrici create saranno $n/step$ e conterranno valori interi randomici compresi nell'intervallo $[0, 255]$. Il metodo è privato e utilizzabile solo all'interno della classe.
- `__compare_dct(self) -> list[list[float]]`, un metodo privato che compara in termini temporali la DCT2 implementata da libreria e la DCT2 osservata a lezione. Le matrici di comparazione vengono create sfuttando il metodo `__generate_matrices(self)`.
- `dct_comparison(self)`, sfruttando il metodo introdotto al punto precedente compara la DCT2 implementata da Scipy e la DCT2 osservata a lezione, inoltre realizza un plot

per osservare le differenze in termini di complessità temporale.

Il main presente all'interno del modulo va a verificare la correttezza della DCT e DCT2 implementata da **Scipy** con: i blocchi reference presenti nella consegna del progetto e la DCT2 osservata a lezione. Al termine viene chiamato il metodo `dct_comparison(self)` per creare i plot e osservare le differenze prestazionali in termini temporali. Le matrici di cui è stata calcolata la DCT2 hanno dimensione comprese nell'intervallo [10, 3000] ed hanno incremento nella dimensione di `step = 10`.

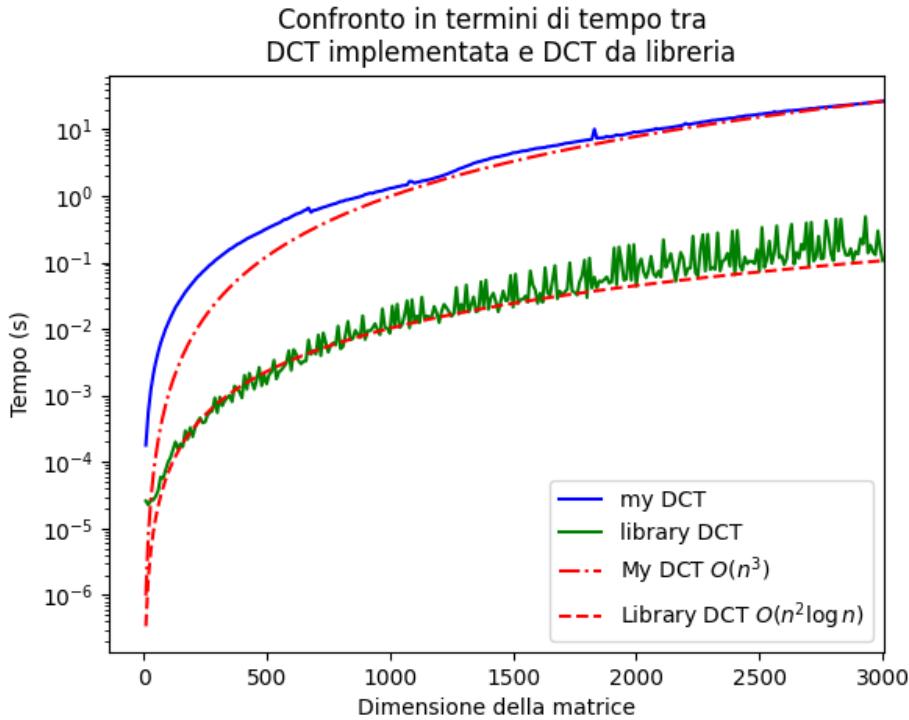


Figura 2.1: Confronto in termini temporali tra la DCT2 implementata a lezione con complessità $O(n^3)$ e la DCT2 implementata in modo ottimizzato da **Scipy** con complessità $O(n^2 \log(n))$. Si può osservare che le prestazioni misurate seguano asintoticamente le complessità teoriche degli algoritmi. Quest'ultime sono rappresentate dalle linee rosse tratteggiate.

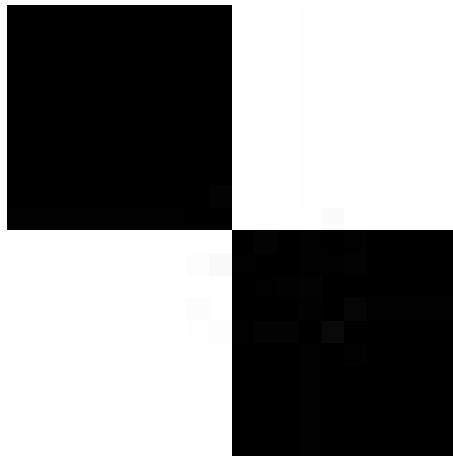
La Figura 2.1 mostra l'andamento delle prestazioni dei due algoritmi che eseguono la DCT2. La trasformazione implementata a lezione ha prestazioni in termini di tempo peggiori rispetto alla DCT2 implementata da **Scipy**. La differenza è data dagli algoritmi di calcolo, infatti, **Scipy** impiega procedure molto efficienti basati sui principi della Fast Fourier Transform, invece in aula è stato sviluppato un algoritmo basato sui principi della trasformata di Fourier. La DCT2 implementata in aula è fortemente dominata da overhead¹. Infine sono osservabili molte oscillazioni nelle prestazioni ottenute dalla DCT2 implementata da **Scipy**. Questo andamento è principalmente dovuto alle varie ottimizzazioni eseguite all'interno dell'algoritmo *split-radix*, in particolare per sequenze di numeri reali o simmetriche.

¹Quando l'algoritmo è fortemente dominato da overhead dedica la maggior parte del tempo di calcolo per operazioni interne alle librerie e non ai calcoli da effettuare.

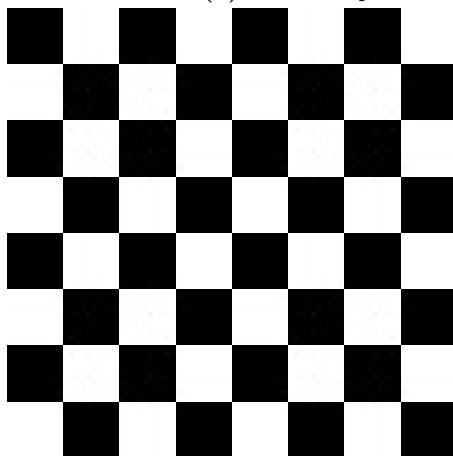
Capitolo 3

La compressione delle immagini

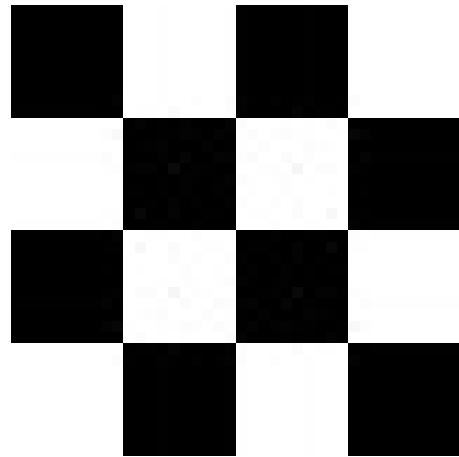
Il modulo `main` contiene la funzione che avvia il programma di compressione delle immagini e mostra l'interfaccia grafica presentata in Figura 1.1. Sono state eseguite delle prove di compressione su alcune immagini di test presenti in Figura 3.3.



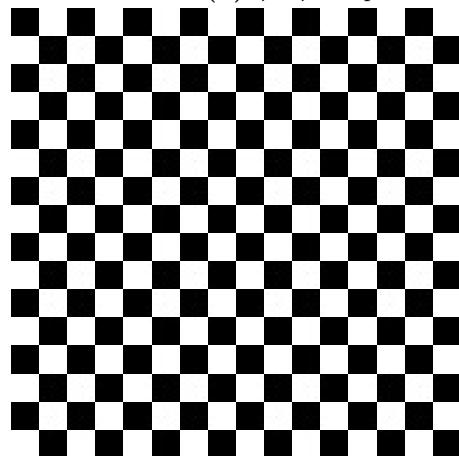
(a) `20x20.bmp`



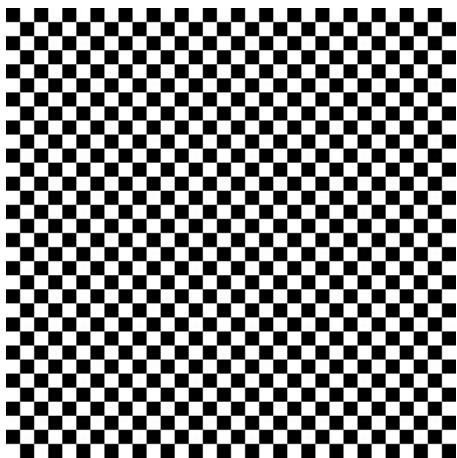
(c) `80x80.bmp`



(b) `40x40.bmp`



(d) `160x160.bmp`



(a) bridge.bmp



(b) cathedral.bmp



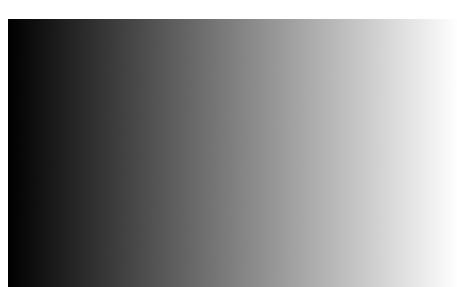
(c) deer.bmp



(d) gradient.bmp



(e) gradient.bmp



(f) gradient.bmp

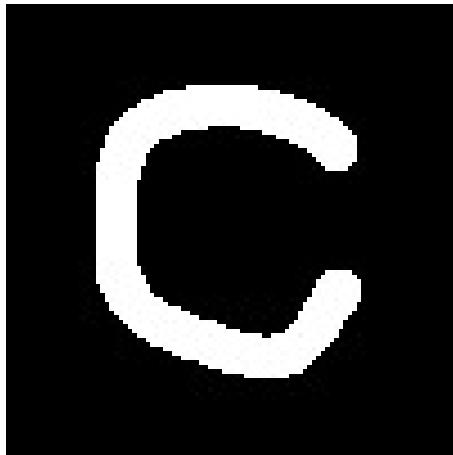
(a) *prova.bmp*(b) *shoe.bmp*

Figura 3.3: Immagini di test impiegate. Il formato in cui sono salvate è *bmp* per evitare una compressione. Tutte le immagini sono in scala di grigio.

Tra le immagini si evidenziano le prime 6 figure che rappresentano una scacchiera, l'unica differenza risiede nella dimensione dell'immagine. Queste figure, *prova.bmp* e *shoe.bmp*, sono le uniche immagini ad alto contrasto presenti. *gradient.bmp* presenta una colorazione a gradiente che va a limitare la differenza in termini di contrasto. *deer.bmp*, *bridge.bmp* e *cathedral.bmp* sono per la maggior parte della loro dimensione a basso contrasto.

3.0.1 Compressione alto contrasto

Come già anticipato le immagini a scacchiera sono ad alto contrasto, ovvero è presente una grande differenza di luminosità tra parti chiare e scure dell'immagine. I primi esperimenti sono volti ad evidenziare l'importanza della dimensione della finestra F . In particolare avere una finestra che è sottomultiplo della dimensione della scacchiera non porta a nessun tipo di compressione, o meglio questa avviene solo se si taglano tutte le frequenze.



Figura 3.4: Scacchiera $20 \times 20.\text{bmp}$ originale e compressa. I parametri sfruttati sono $F = 10$ e $d = 1$. Dato che F è sottomultiplo di 20 allora anche lasciando la frequenza più importante l'immagine non cambia.

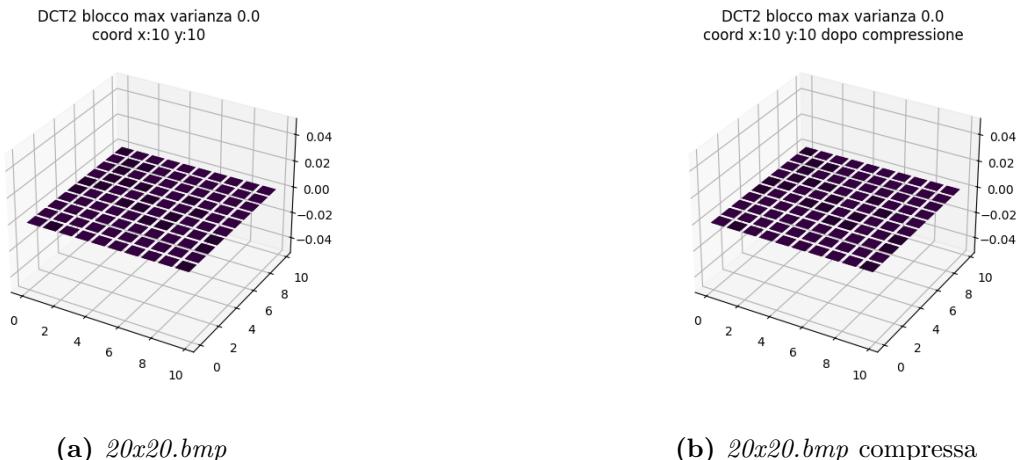


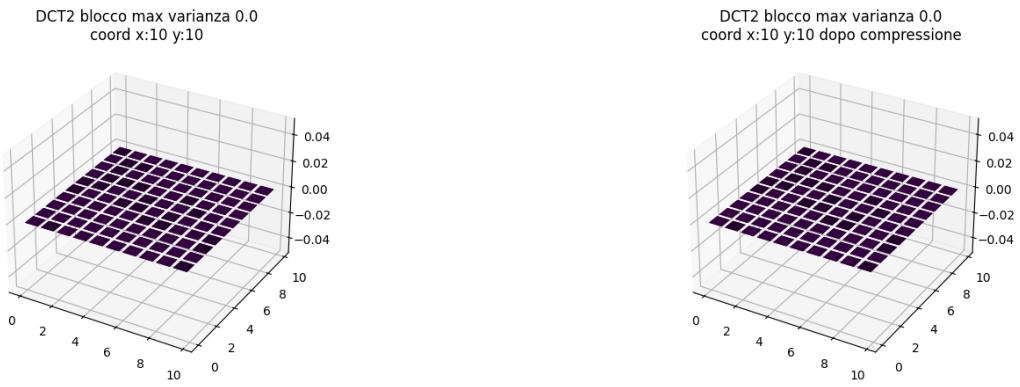
Figura 3.5: Istogramma delle frequenze della scacchiera $20 \times 20.\text{bmp}$. L'istogramma del blocco con varianza maggiore mostra chiaramente come non ci sia stato alcun taglio delle frequenze perché quel blocco si riferisce al solo cubetto nero di coordinate $(x = 10, y = 10)$ e non contiene zone ad alto contrasto. Questo fatto viene confermato anche dalla varianza.

La Figura 3.4 e Figura 3.5 mettono in evidenza quanto detto. La finestra $F = 10$ è un sottomultiplo della scacchiera. È risaputo che la compressione con DCT2 porti a problemi quando incontra zone ad alto contrasto ma i blocchi da comprimere così creati sono omogenei,

di conseguenza la perdita di informazioni è nulla e la compressione non avviene. Questo fatto è dimostrabile anche applicando una compressione con $F = 10$ e ponendo il parametro di taglio delle frequenze pari al valore massimo ammissibile $d = 18$, così facendo saranno mantenute tutte le frequenze.



Figura 3.6: Scacchiera 20×20 .*bmp* originale e compressa. I parametri sfruttati sono $F = 10$ e $d = 18$. Facendo riferimento alla norma dell'occhio la Figura 3.4 risulta del tutto identica alle immagini mostrate.



(a) Istogramma del blocco con varianza massima di 20×20 .*bmp*

(b) Istogramma del blocco con varianza massima di 20×20 .*bmp* compressa

Figura 3.7: Istogramma dei valori dei pixel della scacchiera 20×20 .*bmp*. L'istogramma del blocco con varianza maggiore mostra chiaramente come non ci sia stato alcun taglio delle frequenze e conseguente alterazione del colore.

La Figura 3.6 e la Figura 3.7 confermano quanto affermato. Variando il parametro di taglio delle frequenze l'immagine non subisce compressione e non viene alterata nonostante presenti un alto contrasto. La mancata compressione avviene perché vengono mantenute tutte le frequenze. In conclusione è possibile affermare che: se il parametro F è un sotto-multiplo della dimensione della scacchiera allora qualsiasi sia il parametro d impostato non avverrà alcuna compressione dell'immagine, nonostante venga lasciata solo la frequenza più importante. Quanto detto è possibile perché ciascun blocco ha varianza 0.0 e la sola frequenza contiene tutte le informazioni necessarie per ricostruire l'immagine. Assegnando alla

finestra una dimensione che non è sottomultiplo della dimensione della scacchiera è osservabile una compressione dell'immagine. La Figura 3.8 mostra la compressione della scacchiera *80x80.bmp*.

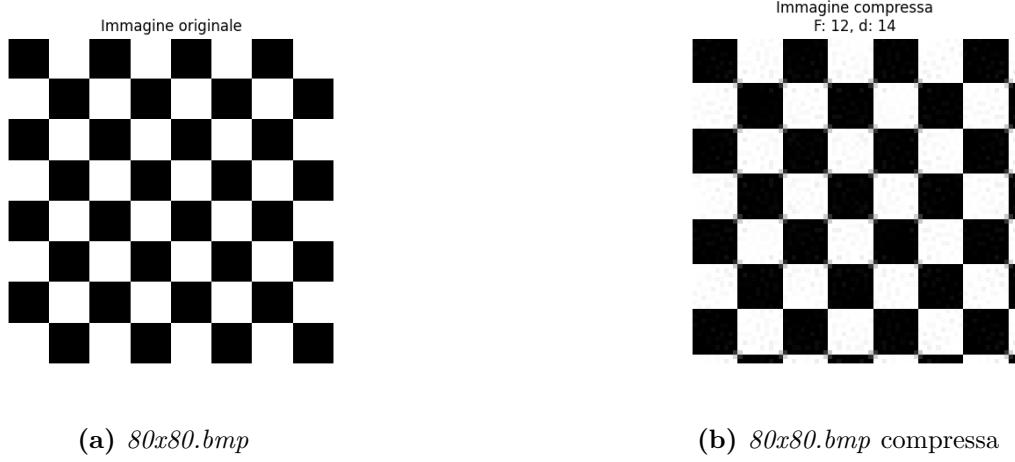


Figura 3.8: Scacchiera *80x80.bmp* originale e compressa. I parametri sfruttati sono $F = 12$ e $d = 14$. La scacchiera viene effettivamente compressa, si nota la mancanza dei blocchetti tagliati e una leggera degradazione dei colori.

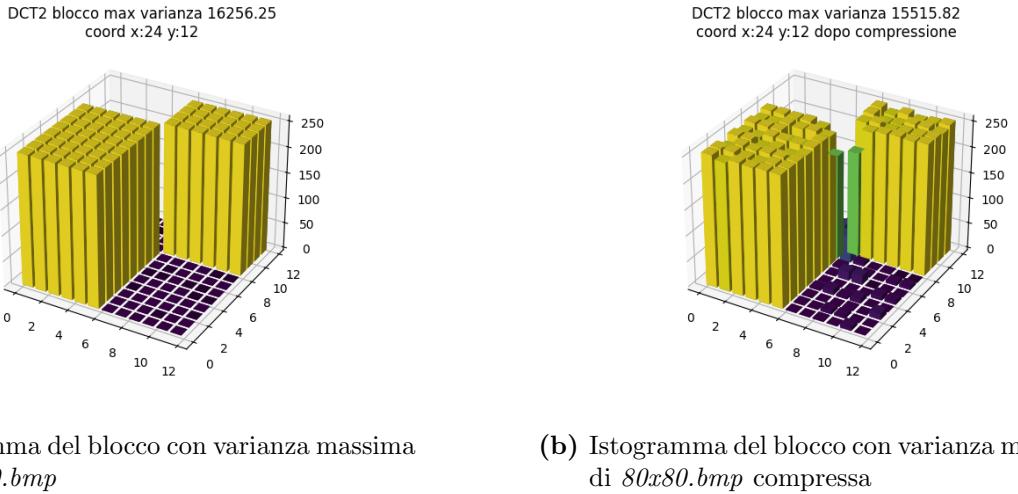


Figura 3.9: Istogramma dei valori dei pixel della scacchiera *80x80.bmp*. L'istogramma mostra l'efficacia della compressione. Alcune frequenze sono state tagliate, quest'operazione è stata riflessa sui valori assunti dai pixel.

In Figura 3.8 è rappresentata la scacchiera *80x80.bmp* che è stata compressa tagliando le frequenze comprese nell'intervallo chiuso [14, 22]. La finestra $F = 12$ ha permesso di mettere in difficoltà l'algoritmo dato che all'interno di un blocchetto è presente un alto contrasto. Tuttavia sono state tagliate frequenze elevate che non contengono molte informazioni, l'operazione descritta ha permesso di ricostruire l'immagine senza inserire troppi artefatti. L'istogramma di Figura 3.9 mette in evidenza l'entità della compressione, è possibile osservare la lieve

diminuzione dell'intensità di alcuni pixel presenti nella zona bianca della scacchiera, inoltre si verifica anche una diminuzione di intensità all'interno della zona nera della scacchiera. La Figura 3.10 mostra quanto detto.

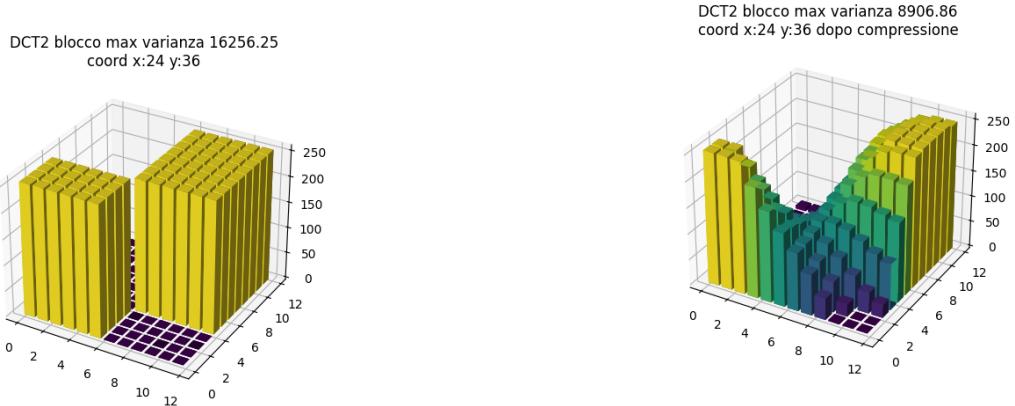


Figura 3.10: Zoom dell'immagine in Figura 3.8 compressa. È ben visibile la degradazione dell'immagine dovuta al taglio delle frequenze effettuato durante la compressione.

È possibile introdurre una maggior perdita di qualità all'interno dell'immagine eseguendo un taglio delle frequenze maggiore, infatti le frequenze più basse sono quelle che contengono una grande quantità di informazioni necessarie per ricostruire correttamente l'immagine nel passaggio DCT2 - IDCT2. A questo punto, lasciando invariata la dimensione della finestra è stato diminuito il parametro $d = 4$ per tagliare tutte le frequenze presenti nell'intervallo chiuso $[4, 22]$.



Figura 3.11: Scacchiera 80x80.bmp compressa con parametri $F = 12$ e $d = 4$. La compressione è molto più invasiva e il taglio della maggior parte delle frequenze va a degradare eccessivamente l'immagine. Lo zoom sulle coordinate mostra in modo evidente gli artefatti presenti nell'immagine.

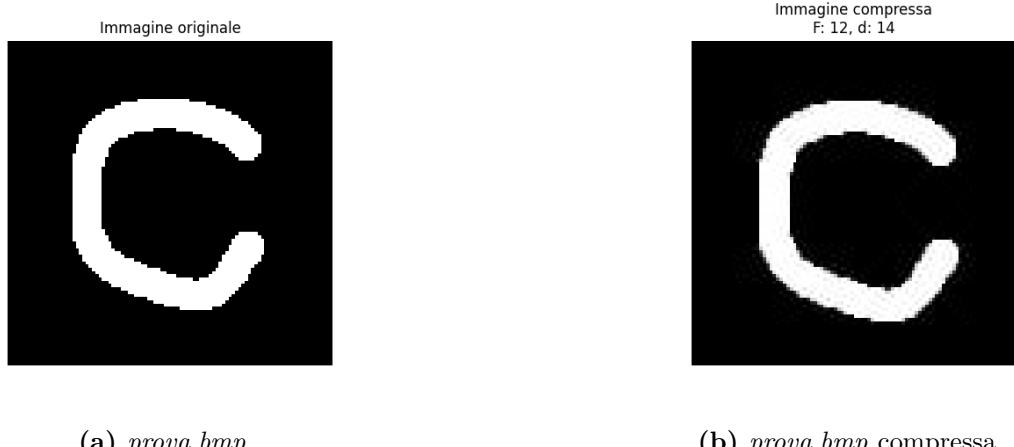


(a) Istogramma del blocco con varianza massima di *80x80.bmp*

(b) Istogramma del blocco con varianza massima di *80x80.bmp* compressa con parametri $F = 12$ e $d = 4$

Figura 3.12: Istogramma dei valori dei campioni prima e dopo la compressione. L’istogramma mette in evidenza la forte degradazione dell’immagine a causa della perdita di informazioni dovuto all’eccessivo taglio delle frequenze.

La Figura 3.11 e Figura 3.12 mostrano l’influenza di un eccessivo taglio di frequenze sulla ricostruzione dell’immagine. In particolare eliminando le frequenze superiori a 4 molte informazioni andranno perse a causa dell’annullamento di alcune basse frequenze. Dato che il blocco da ricostruire è ad alto contrasto le frequenze tagliate hanno una grande importanza, di conseguenza il loro annullamento porta a questo tipo di errori. Il ragionamento eseguito sulle immagini a scacchiera può essere applicato anche a *prova.bmp* perché è una figura ad alto contrasto.

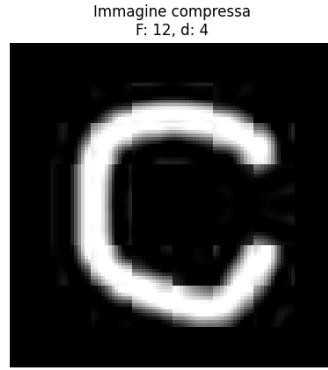


(a) *prova.bmp*

(b) *prova.bmp* compressa

Figura 3.13: *prova.bmp* originale e compressa con parametri $F = 12$ e $d = 14$. L’immagine compressa presenta mancanza di bordi nitidi e artefatti all’interno della *C*.

Le Figura 3.13 e Figura 3.14 confermano quanto detto precedentemente, anche se la degradazione risulta meno evidente rispetto al caso della scacchiera. Questo perché l’elevato



(a) *prova.bmp* compressa con parametri $F = 12$ e $d = 4$. Rispetto alla Figura 3.13 la nitidezza e la quantità di artefatti peggiora.

Figura 3.14: *prova.bmp* compressa con parametri $F = 12$ e $d = 4$. L’immagine ha un maggior degrado rispetto a *prova.bmp* compressa in Figura 3.13. Questo è dovuto al maggior numero di frequenze annullate.

contrasto è presente solo lungo il bordo della lettera *C* e non al suo interno a causa dell’assenza di un pattern ripetitivo. Applicando la compressione con parametri $F = 12$ e $d = 4$ all’immagine *shoe.bmp*, la degradazione appare più marcata ma inferiore a quella osservata nelle immagini a scacchiera perché anche in questo caso non è presente un pattern. La perdita di qualità è dovuta all’alternarsi di contrasti abbastanza netti in alcune aree adiacenti, che compromettono l’immagine in modo significativo ma non in modo drastico come accade nella scacchiera. Quanto descritto è osservabile in Figura 3.15.



(a) *shoe.bmp* originale

(b) *shoe.bmp* compressa con parametri $F = 12$ e $d = 4$

Figura 3.15: *shoe.bmp* originale e compressa con parametri $F = 12$ e $d = 4$. L’immagine compressa presenta una forte degradazione e la presenza di molti artefatti. Questo è dovuto a due fattori: l’annullamento di alcune alte frequenze e la presenza di differenti contrasti piuttosto marcati all’interno dell’immagine.

3.0.2 Compressione basso contrasto

Dopo aver analizzato il comportamento della compressione su immagini ad alto contrasto è stata presa la decisione di testare la compressione su immagini a basso contrasto come: *deer.bmp*, *cathedral.bmp*, *bridge.bmp* e *gradient.bmp*. Il primo esperimento è stato fatto su quest'ultima. I parametri di compressione sono stati impostati su: $F = 12$ e $d = 4$ perché essendo l'immagine a basso contrasto si è deciso di provare subito una compressione aggressiva.

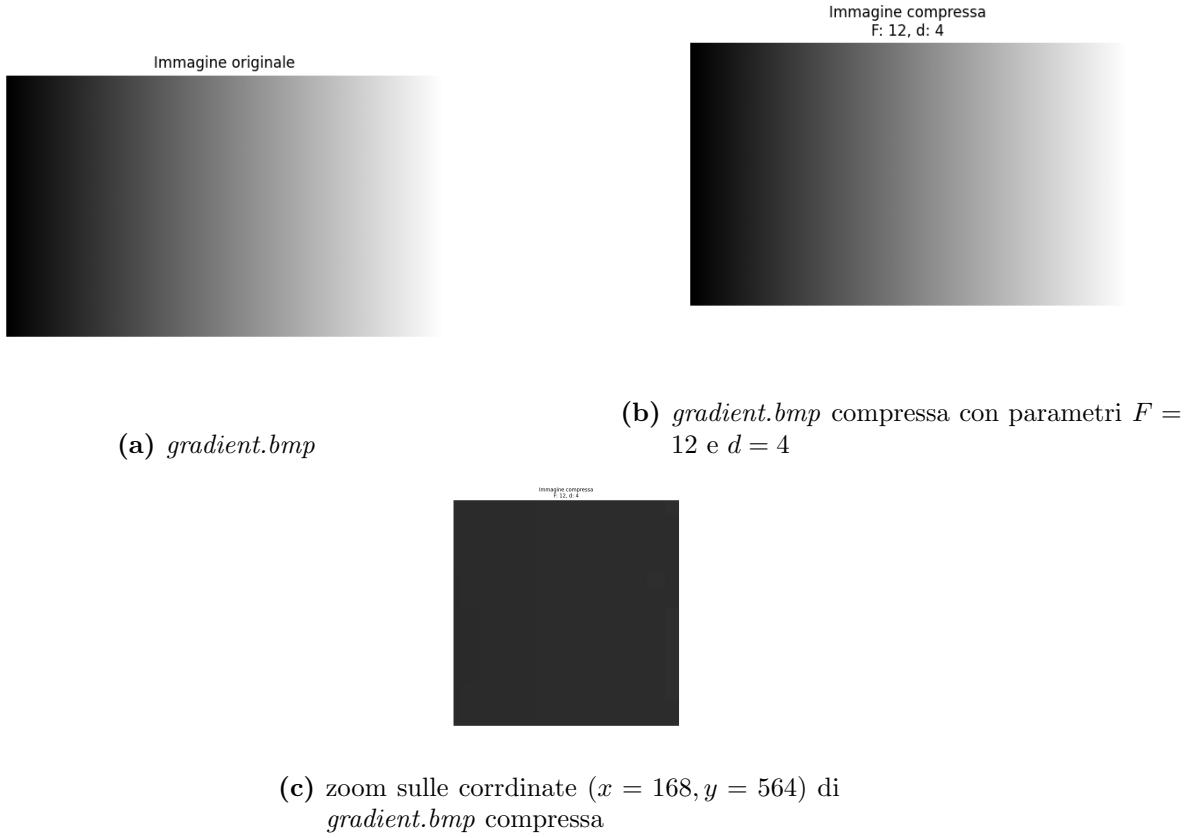


Figura 3.16: *gradient.bmp* originale, compressa con parametri $F = 12$ e $d = 4$ e zoom sulle coordinate $(x = 168, y = 564)$ di quest'ultima. L'immagine compressa non presenta un calo di qualità. Questo è osservabile dalla mancanza di artefatti e dalla bassa variazione della varianza. Tutto ciò è dovuto al basso contrasto presente nell'immagine.

La Figura 3.16 mostra il buon comportamento dell'algoritmo di compressione in immagini a basso contrasto, pur tagliando le frequenze nell'intervallo chiuso $[4, 22]$ è possibile creare un'immagine compressa senza perdita di qualità. Quanto descritto è attuabile grazie al basso contrasto presente nell'immagine. Anche lo zoom eseguito sulle coordinate $(x = 168, y = 564)$, individuate come blocco con varianza maggiore, mostra una buona qualità dell'immagine. La Figura 3.17 conferma quanto detto, infatti il valore all'interno dei pixel viene leggermente attenuato. L'unico modo in cui è possibile ottenere un cattivo

risultato sarebbe l'impostazione di una finestra di grandi dimensioni e l'uso di un taglio di frequenze aggressivo.

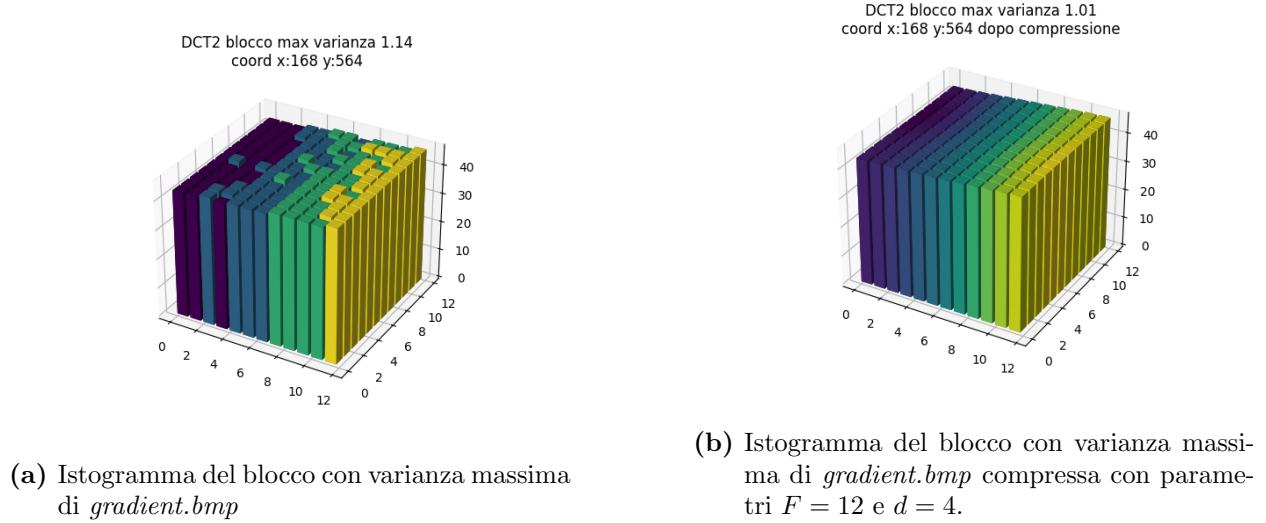


Figura 3.17: Istogramma dei valori dei pixel prima e dopo la compressione. L'istogramma dell'immagine compressa conferma la bassa diminuzione della qualità dell'immagine pur avendo tagliato la maggior parte delle frequenze.

Tuttavia l'obiettivo della compressione è di riuscire a mantenere una buona qualità dell'immagine pur avendo eliminato alcune informazioni, questo verrebbe a meno eseguendo la compressione con una finestra di grandi dimensioni. L'algoritmo è stato applicato anche all'immagine *bridge.bmp*, i risultati ottenuti sono i seguenti.

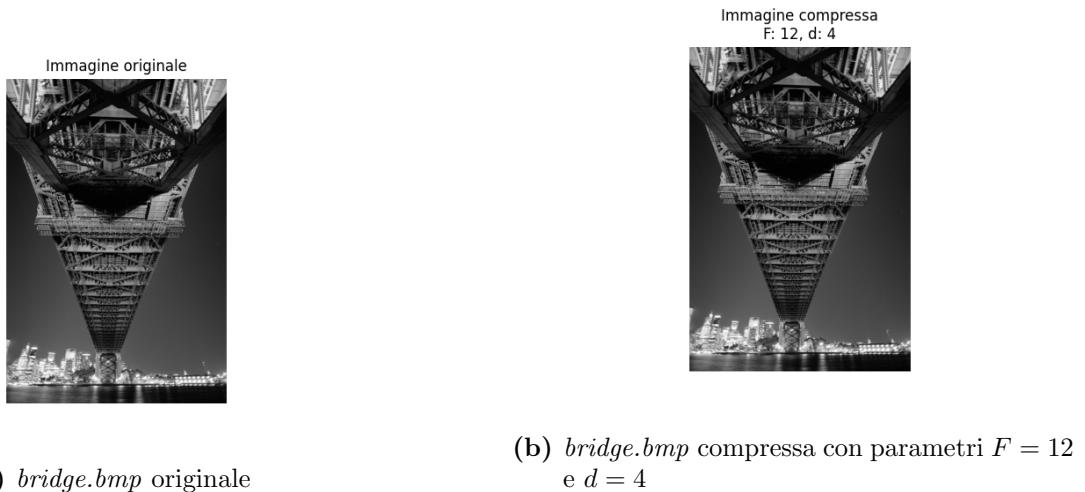
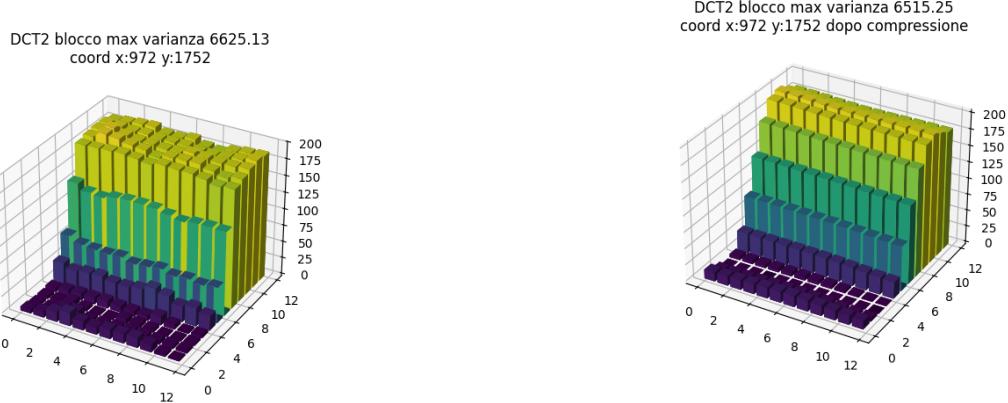


Figura 3.18: *bridge.bmp* originale e compressa con parametri $F = 12$ e $d = 4$. Essendo l'immagine di grandi dimensioni: $height = 4049px$ e $width = 2749px$ e a basso contrasto non è possibile notare l'effetto della compressione.

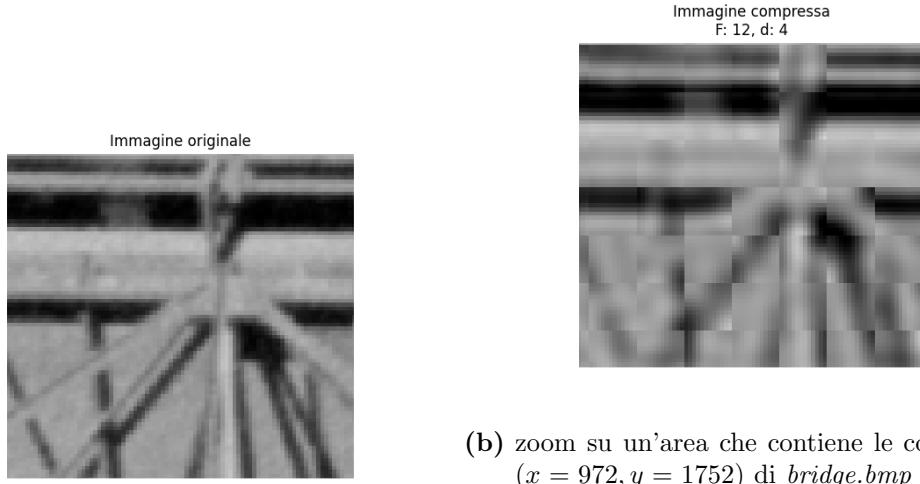
A causa delle grandi dimensioni dell'immagine $height = 4049px$ e $width = 2749px$ e dato che il contrasto generale dell'immagine è basso non è apprezzabile l'effetto della compressione. Tuttavia gli istogrammi di Figura 3.19 mostrano un calo di qualità dell'immagine compressa.



(a) Istogramma del blocco con varianza massima di *bridge.bmp*.

(b) Istogramma del blocco con varianza massima di *bridge.bmp* compressa con parametri $F = 12$ e $d = 4$

Figura 3.19: Istogramma dei valori dei campioni prima e dopo la compressione. L'istogramma dell'immagine compressa mostra una grande diminuzione della qualità dell'immagine confermata anche dalla diminuzione della varianza di circa 100 punti.



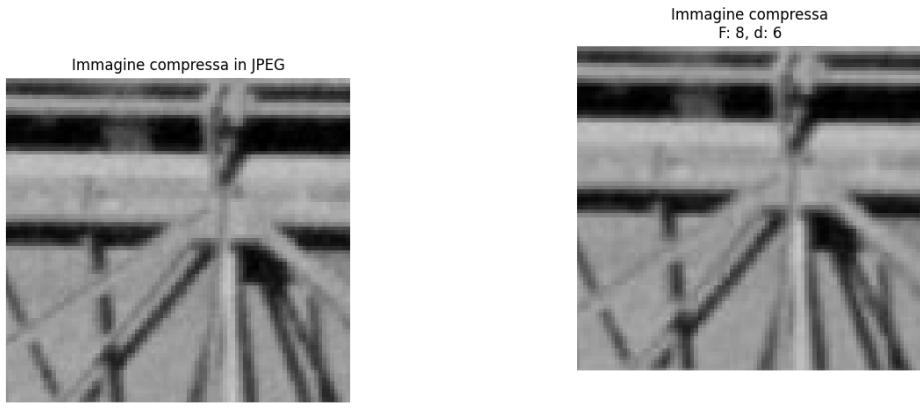
(a) zoom su un'area che contiene le coordinate $(x = 972, y = 1752)$ di *bridge.bmp*

(b) zoom su un'area che contiene le coordinate $(x = 972, y = 1752)$ di *bridge.bmp* compressa

Figura 3.20: *bridge.bmp* originale e compressa con parametri $F = 12$ e $d = 4$ e zoom in un'area contenente le coordinate $(x = 972, y = 1752)$. L'immagine compressa presenta un calo di qualità evidente causato dalla presenza di un elevato contrasto nella zona selezionata, infatti anche se l'immagine non è ad alto contrasto alcune zone presentano questa caratteristica e la compressione ne risente. Tuttavia, essendo l'immagine di grandi dimensioni la diminuzione della sua qualità non è osservabile ad occhio nudo.

Eseguendo uno zoom come in Figura 3.20 è osservabile il calo di qualità nell'immagine dovuto alla presenza di alcune zone ad alto contrasto presenti nell'immagine che impattano significativamente sulla qualità della figura stessa. Tuttavia essendo *bridge.bmp* di grandi dimensioni la diminuzione della qualità e la creazione di piccoli artefatti non è osservabile ad occhio nudo.

Il risultato ottenuto in Figura 3.20 è migliorabile andando ad diminuire il numero di frequenze tagliate d oppure rimpicciolendo la finestra F . Provando ad "imitare" la compressione JPEG sono stati impostati i seguenti parametri: $F = 8$ e $d = 6$, nello specifico è stata diminuita la dimensione della finestra ponendola uguale al formato JPEG, inoltre sono state tagliate circa la metà delle frequenze. Il risultato ottenuto è stato confrontato con la compressione JPEG ed è osservabile in Figura 3.21.



(a) zoom su un'area che contiene le coordinate $(x = 972, y = 1752)$ di *bridge.jpeg*
(b) zoom su un'area che contiene le coordinate $(x = 972, y = 1752)$ di *bridge.bmp* compresa

Figura 3.21: *bridge.jpeg* e *bridge.bmp* compressa con parametri $F = 8$ e $d = 6$ e zoom in un'area che contiene le coordinate $(x = 972, y = 1752)$. L'immagine compressa presenta un calo di qualità paragonabile con quello mostrato dalla figura compressa secondo lo standard JPEG.

Per la figura analizzata i risultati sono del tutto paragonabili in termini qualitativi. Entrambe le immagini si alternano per la presenza di una nitidezza migliore di alcuni bordi. Questa differenza è a vantaggio di *bridge.jpeg* per i bordi obliqui. La diversità descritta può essere data dal mantenimento di alcune frequenze definito con la matrice di quantizzazione e dalla miglior gestione degli alti contrasti. Le caratteristiche descritte permettono di ricostruire in modo migliore alcune parti dell'immagine.

3.1 Conclusioni

Dopo le differenti compressioni eseguite è possibile affermare che: primo, se l'immagine è ad alto contrasto e presenta pattern o zone adiacenti con contrasti elevati è meglio eseguire una compressione mantenendo un numero di frequenze pari a circa la metà dell'intervallo $0 \leq d \leq 2F - 2$ per non perdere troppe informazioni cruciali per la ricostruzione dell'immagine. Secondo, se l'immagine è a basso contrasto è possibile tagliare più della metà delle frequenze, tuttavia delle piccole zone ad alto contrasto presenti nell'immagine ne risentirebbero ma se la figura è di grandi dimensioni il calo di qualità e gli artefatti generati non sono visibili all'occhio umano.

In conclusione avendo implementato una tecnica di compressione di tipo lossy, del tutto simile al JPEG, è bene trovare il giusto compromesso tra dimensione del blocchetto, F , taglio delle frequenze, d e conseguente perdita di qualità nell'immagine. Il taglio delle frequenze in JPEG viene effettuato tramite una matrice di quantizzazione che permette di eseguire un *trade-off* tra qualità e frequenze tagliate. Di conseguenza alcune parti dell'immagine potrebbero essere ricostruite in modo migliore grazie ad un'ottima gestione dei contrasti attraverso la matrice di quantizzazione, come analizzato in Figura 3.21.