Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

Progetto 2
Metodi del Calcolo Scientifico

Relazione di:
Matteo Cavaleri - 875050
Cristian Piacente - 866020

Anno Accademico 2023-2024

Indice

Capitolo 1

Introduzione

In questo capitolo viene introdotto in cosa consiste il secondo progetto, per poi trattare brevemente dal punto di vista teorico quali nozioni sono state utilizzate al fine dell'implementazione.

1.1 Scopo del progetto

Il progetto si divide in due parti:

- implementazione della `DCT2` e confronto del tempo di esecuzione di essa rispetto alla versione fast resa disponibile dalla libreria;
- creazione di un'interfaccia grafica che prende in input un'immagine `.bmp` in toni di grigio e applica un algoritmo di compressione, con relativi esperimenti.

Dunque, lo scopo è quello di confrontare la propria implementazione della `DCT2` con quella di una libreria e sfruttare quest'ultima per mettere a punto una semplice applicazione a supporto di una compressione di immagini in scala di grigi.

1.2 Cenni di teoria

Prima di trattare la parte che riguarda l'implementazione, è bene affrontare la parte teorica su cui si basa lo sviluppo del progetto.

1.2.1 Introduzione alla DCT

La Discrete Cosine Transform (DCT) è fondamentale nell'analisi dei segnali discreti, trovando particolare applicazione nella compressione delle immagini (e.g. compressione JPEG), dove è essenziale ridurre le dimensioni del file mantenendo una qualità adeguata.

Questa trasformata utilizza una base costituita da funzioni coseno, che sono particolarmente adatte per rappresentare segnali che presentano variazioni graduali (tipiche delle immagini naturali).

1.2.2 DCT e DCT2

Le funzioni coseno formano una base ortogonale nel contesto della DCT.
Dato un vettore v di $\mathbb{R}^N$, la procedura che costruisce i coefficienti a_k della trasformata nel dominio delle frequenze si chiama **Discrete Cosine Transform** (**DCT**).
La formula della DCT per una sequenza di N valori v_i è definita come segue:

$$a_k = \frac{1}{f}(v \cdot w^k) = \frac{1}{f}\sum_{i=0}^{N-1} v_i \cos\left(\pi k \frac{2i+1}{2N}\right), \quad k = 0, 1, \ldots, N-1$$

dove

- $w^k \in \mathbb{R}^N$ rappresenta le funzioni coseno associate alla frequenza k
- f rappresenta un fattore di scaling, utilizzato per normalizzare la trasformazione in maniera identica a quanto fatto dalla libreria, definito come:

$$f = \begin{cases} \sqrt{N} & \text{se } k = 0, \\ \sqrt{\frac{N}{2}} & \text{altrimenti.} \end{cases}$$

N.B: l'unica differenza rispetto allo scaling trattato a lezione consiste nell'aggiunta della radice quadrata.

Nel contesto delle immagini, che possono essere considerate come segnali bidimensionali, la DCT viene estesa alla DCT2.

La **DCT2** equivale a fare due trasformazioni DCT, in particolare si ha l'applicazione della DCT **prima a tutte le colonne** dell'immagine e poi a **tutte le righe** del risultato (oppure prima le righe e poi le colonne, non cambia).

Come accennato, utilizzando la DCT2 si possono ridurre significativamente le dimensioni dei dati necessari per memorizzare un'immagine.

Capitolo 2

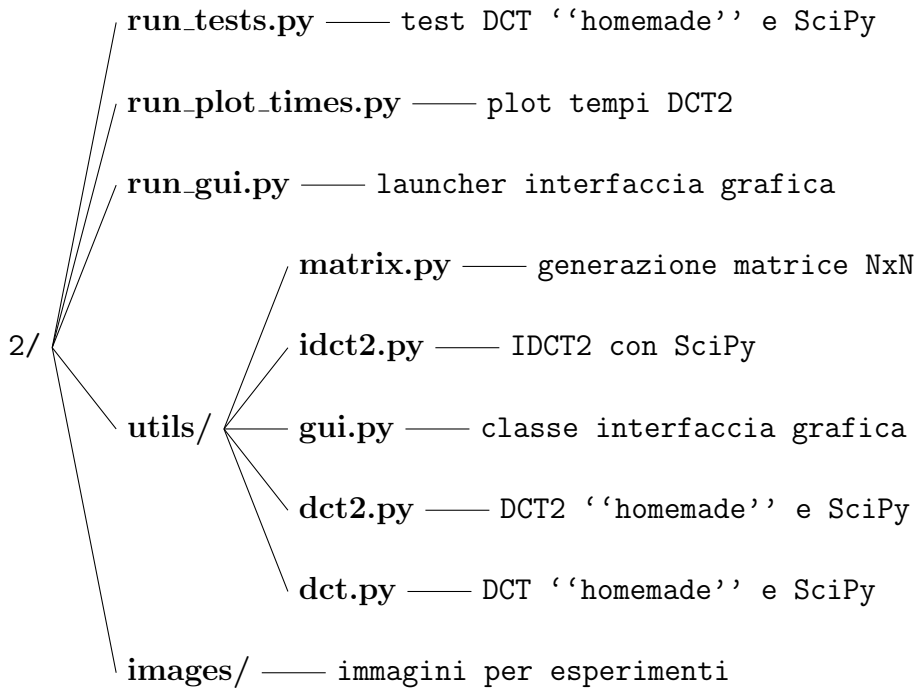
Implementazione

Questo capitolo tratta lo sviluppo del progetto, mostrando la struttura, notizie sulla libreria utilizzata e snippet di codice.

2.1 Struttura del progetto

Per l'implementazione è stato utilizzato il linguaggio open source Python, in cui è disponibile la libreria SciPy che offre funzionalità sulla DCT.

Il progetto è strutturato come segue:



2.2 Notizie su SciPy

Come accennato precedentemente, si utilizza SciPy per richiamare l'implementazione fast (`FFT`, Fast Fourier Transform) della libreria.

Viene usato il modulo scipy.fftpack, nello specifico le funzioni che vengono richiamate sono dct, dctn e idctn.

2.2.1 Funzione dct

La funzione `dct` restituisce la Discrete Cosine Transform di un array x in input.

Nel progetto, viene richiamata la libreria con i parametri riportati nel codice 2.1.

```
1 from scipy.fftpack import dct
2
3 def lib_dct(x):
4     return dct(x, type=2, norm="ortho")
```

Listing 2.1: DCT della libreria SciPy

Facendo riferimento al codice 2.1, si ha che:

- `x` è l'array di input
- `type` specifica il tipo di DCT da utilizzare (la libreria ne implementa diversi), impostato su `2` che è la definizione più comunemente utilizzata
- `norm` specifica la normalizzazione della trasformata, impostata su `"ortho"` per ottenere una trasformata ortogonale

È possibile trattare nel dettaglio il tipo `2`, per confrontare la definizione con quella data precedentemente nei cenni di teoria (1.2.2).
DCT **Type II** implementa la seguente definizione della DCT:

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

Poiché viene passato il parametro `norm` con il valore `"ortho"`, i coefficienti y_k ottenuti vengono moltiplicati per il seguente scaling factor f:

$$f = \begin{cases} \sqrt{\frac{1}{4N}} & \text{se } k = 0, \\ \sqrt{\frac{1}{2N}} & \text{altrimenti.} \end{cases}$$

Effettuando semplici passaggi matematici, si può concludere che la formula implementata da `scipy.fftpack.dct` è **identica** a quella descritta nella teoria.

2.2.2 Funzione dctn

La funzione `dctn` permette di calcolare la Discrete Cosine Transform n-dimensionale (DCTn), utile per applicare la DCT su un array multidimensionale senza dover applicare "manualmente" la DCT monodimensionale lungo le colonne e le righe.

Quando viene applicata a un'immagine, la funzione trasforma i dati dal dominio dello spazio al dominio della frequenza.

Analogamente alla funzione `dct`, anche in questo caso si hanno diverse implementazioni disponibili a seconda del parametro `type`.
Non viene riportato lo snippet di codice, in quanto non aggiunge informazione rispetto a quanto visto nel codice 2.1.

2.2.3 Funzione idctn

La funzione `idctn` implementa l'Inverse DCT n-dimensionale: essa ha un ruolo fondamentale in un algoritmo di decompressione, per ricostruire un'immagine dopo aver applicato operazioni di compressione sulla `dctn`.

La ricostruzione di un'immagine è possibile passando dal dominio della frequenza al dominio dello spazio, i.e. l'operazione inversa rispetto alla `dctn`.

Come nel caso della DCT, si hanno diversi tipi disponibili e i parametri utilizzati sono gli stessi di quelli riportati nel codice 2.1.

2.3 Prima parte ("preliminare") del progetto

Dopo aver discusso sulla struttura e sulla libreria utilizzata, si può iniziare a discutere riguardo allo sviluppo del progetto.

2.3.1 Implementazione della DCT

Come prima cosa, è stata implementata la DCT (monodimensionale, su cui si basa la versione bidimensionale) utilizzando la formula descritta nella teoria (1.2.2).

Questo è stato possibile tramite la definizione di due funzioni Python:

- una funzione `w`, che prende in input k e N (che sono rispettivamente la frequenza, di cui si vogliono calcolare le funzioni coseno, e la lunghezza del vettore in input) e restituisce $w^k \in \mathbb{R}^N$
- una funzione `my_dct`, che prende in input un vettore v in $\mathbb{R}^N$ e restituisce i coefficienti a_k (con il fattore di scaling già applicato) della trasformata nel dominio delle frequenze, con $k = 0, 1, \ldots, N-1$

Si riporta nei codici 2.2 e 2.3 l'implementazione in Python delle due funzioni appena trattate.

```python
def w(k, N):
    # Inizializza una lista per memorizzare le funzioni coseno
    coefficients = []

    # Calcola ciascun coseno
    for i in range(N):
        coefficient = np.cos(k * np.pi * (2 * i + 1) / (2 * N))
        coefficients.append(coefficient)

    return coefficients
```

Listing 2.2: Funzione w che calcola le funzioni coseno

```python
def my_dct(v):
    N = len(v)  # Lunghezza del vettore in input
    result = []  # Lista per i coefficienti della trasformata

    # Calcola ciascun coefficiente della DCT
    for k in range(N):
        # Prodotto scalare tra il vettore v e le funzioni coseno
        coef = np.dot(v, w(k, N))

        # Normalizzazione del risultato
        if k == 0:
            coef /= np.sqrt(N)
        else:
            coef /= np.sqrt(N / 2)

        result.append(coef)

    return result
```

Listing 2.3: Funzione my_dct che calcola i coefficienti della trasformata

2.3.2 Implementazione della DCT2

Una volta creata la DCT monodimensionale, si può passare al caso bidimensionale.

Come già discusso nella teoria (1.2.2), è possibile calcolare la `DCT2` applicando la DCT prima su tutte le colonne della matrice e poi su tutte le righe.
A livello di codice, questo si traduce in una funzione `my_dct2` che effettua due cicli for (uno sulle colonne e uno sulle righe), come si può vedere nel codice 2.4.

```
def my_dct2(A):
    # Copia per non modificare la matrice originale
    res = np.array(A, dtype=float)
    # Dimensioni della matrice (righe, colonne)
    N, M = res.shape

    # Applica la DCT a ciascuna colonna
    for j in range(M):
        res[:, j] = my_dct(res[:, j])

    # Applica la DCT a ciascuna riga
    for i in range(N):
        res[i, :] = my_dct(res[i, :])

    return res
```

Listing 2.4: Funzione my_dct2 che calcola la DCT bidimensionale

2.3.3 Confronto tempi di esecuzione (homemade vs SciPy)

Tralasciando dettagli implementativi non rilevanti (i.e. opportuno testing e una funzione che genera una matrice N × N con entrate random tra 0 e 255), a questo punto è possibile procedere con la misurazione dei tempi di esecuzione della `DCT2`, tra versione "homemade" e versione fast di SciPy.

Il tempo di esecuzione della DCT2 "homemade" che ci si aspetta è $\mathcal{O}(n^3)$, poiché (come si può analizzare dai codici riportati in 2.2 e 2.3) la DCT monodimensionale ha tempo $\mathcal{O}(n^2)$, siccome consiste in due cicli for annidati e perciò $n \cdot n = n^2$ iterazioni totali: questa viene eseguita in due cicli for distinti di n iterazioni ciascuno per passare al caso bidimensionale, dunque asintoticamente si ha

$$2 \cdot n \cdot n^2 = 2 \cdot n^3 = \mathcal{O}(n^3)$$

Per quanto riguarda la versione fast implementata da SciPy, è noto che il tempo di esecuzione dovrebbe essere $\mathcal{O}(n^2 \log n)$, ma si potrebbero riscontrare andamenti irregolari per via dell'algoritmo utilizzato.

Per l'analisi, sono state generate 5 matrici quadrate di dimensione **40, 80, 160, 320, 640**: la dimensione raddoppia ogni volta al fine di ottenere un impatto maggiore nei risultati.

Viene utilizzato un grafico in scala semilogaritmica (in cui solo l'asse delle ordinate ha quantità logaritmiche) per confrontare la dimensione e il tempo impiegato dalla `DCT2` nelle due versioni ("homemade" e SciPy) per ogni matrice: ogni tempo viene confrontato con la curva teorica (ottenuta dal limite asintotico), ottenendo in tutto 2 curve teoriche e 2 curve "pratiche" (sperimentali).

C'è solo un piccolo "problema": se si prova a plottare le quantità senza effettuare scaling, si noteranno curve in scala diversa, come riportato in figura 2.1.

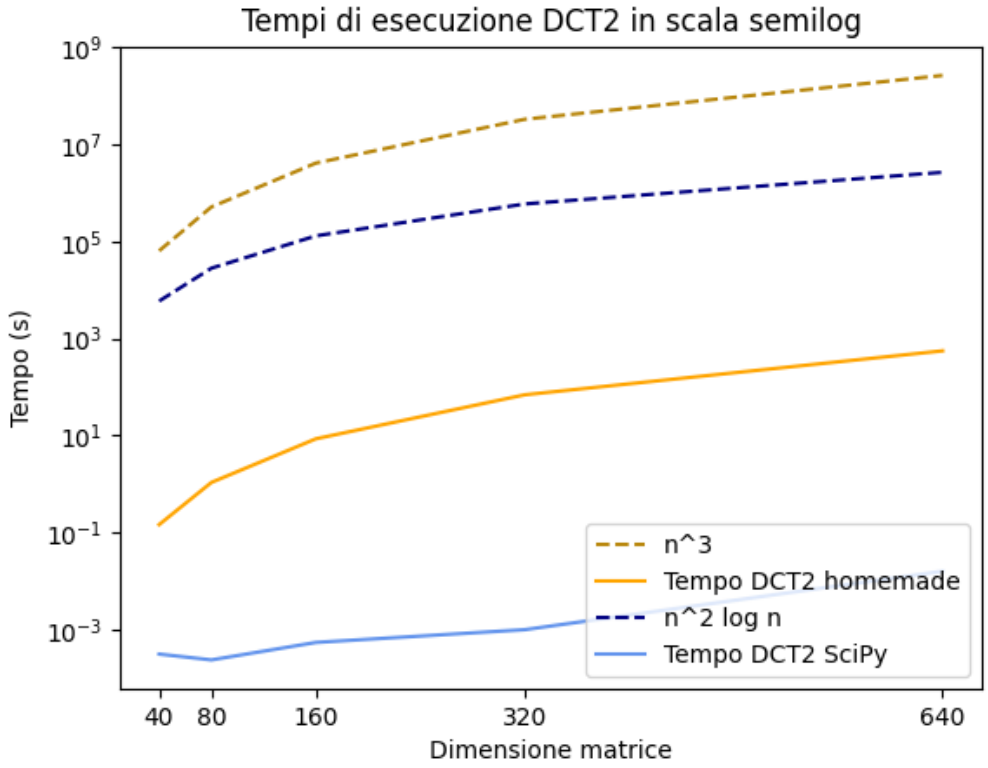


Figura 2.1: Grafico con i tempi di esecuzione **prima** dello scaling

Risulta necessario applicare *scaling* sulle quantità per avere la stessa scala e poter confrontare visivamente le curve tra di loro.

È stato scelto di portare in scala pratica le quantità teoriche, al fine di lasciare invariata la scala delle quantità sperimentali e basarsi su di essa per il confronto.

Per ottenere uno **scaling factor**, viene calcolata la **mediana** di tutti i **rapporti** tra tempi **pratici** e tempi **teorici**.
Ogni elemento della lista contenente i tempi teorici viene moltiplicato per lo scaling factor, ottenendo i tempi **teorici "scaled"**.
Per chiarire ulteriormente quanto appena detto, si può consultare il codice 2.5.

```
1 def scale_theoretical_to_practical(theoretical, practical):
2     scaling_factor = np.median(np.array(practical) / np.array(
    theoretical))
3     return [t * scaling_factor for t in theoretical]
```

Listing 2.5: Funzione scale_theoretical_to_practical che effettua lo scaling dei tempi teorici a pratici

Perciò, ottenuti i due scaling factor (uno per la curva della DCT "homemade" e uno per la curva della DCT di SciPy), si ottengono le curve teoriche "scaled" tramite il calcolo appena trattato, mentre le curve sperimentali rimangono invariate.
Il risultato di questo procedimento si può osservare in figura 2.2.

Osservando la figura 2.2, si può confermare che l'implementazione della DCT2 "homemade" mantiene un tempo di esecuzione che segue la curva teorica per ogni dimensione della matrice passata in input, mentre l'implementazione della libreria SciPy presenta alcune irregolarità. Studiando la curva sperimentale nel dettaglio, si conferma che anche aumentando la dimensione delle matrici il tempo di esecuzione ha un limite asintotico superiore di $\mathcal{O}(n^2 \log n)$, in quanto il tempo pratico non tende ad alzarsi rispetto al tempo teorico, se non per qualche irregolarità particolarmente riscontrata nelle matrici di dimensione ridotta.

È possibile effettuare un'ulteriore analisi concentrandoci sulle quantità "pure" ottenute dalla misurazione del tempo trascorso in secondi, senza l'applicazione di logaritmi o scaling. Queste misurazioni vengono riportate nella tabella 2.1.

Dalla tabella 2.1 è ancora più evidente la differenza tra utilizzare la versione "homemade" della DCT2 e utilizzare la libreria SciPy: il tempo della libreria rimane vicino a zero secondi anche per matrici di certe dimensioni (situazione tipica nel contesto delle immagini, in cui si ha un numero elevato di pixel), che risulta ideale per l'applicazione di algoritmi di compressione.

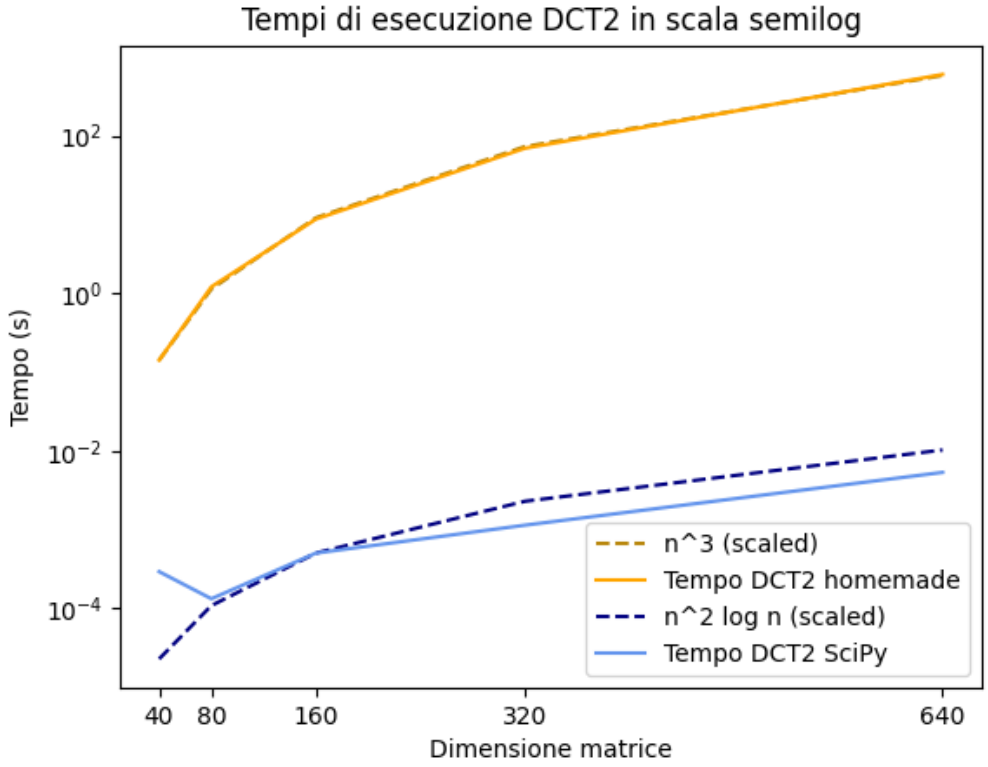


Figura 2.2: Grafico con i tempi di esecuzione **dopo** lo scaling

$n \times n$	DCT2 homemade (s)	DCT2 SciPy (s)	n^3	$n^2 \log n$
40x40	0.1421157	0.0002941	64000	5902.21
80x80	1.2109812	0.0001332	512000	28044.97
160x160	8.7427219	0.0005055	4096000	129924.45
320x320	69.3037399	0.0011426	32768000	590676.07
640x640	600.3448457	0.0053508	262144000	2646617.37

Tabella 2.1: Tempi di esecuzione DCT2 homemade e SciPy con complessità teorica (N.B: $n^2 \log n$ è stato arrotondato a 2 cifre decimali, mentre i tempi a 7)

2.4 Seconda parte del progetto

Avendo a disposizione tutte le funzioni necessarie, è possibile passare all'implementazione della seconda parte del progetto, che riguarda lo sviluppo di un'interfaccia grafica e di un algoritmo di compressione.

2.4.1 Interfaccia grafica

Per quanto riguarda l'interfaccia grafica, è stata utilizzata la libreria PyQt6, che combina il framework Qt (cross-platform, sviluppato in C++) e il linguaggio interpretato Python (anch'esso cross-platform).

Nel momento in cui si avvia l'applicazione, viene mostrata una finestra (riportata in figura 2.3) di dimensione fissa 800 × 600.

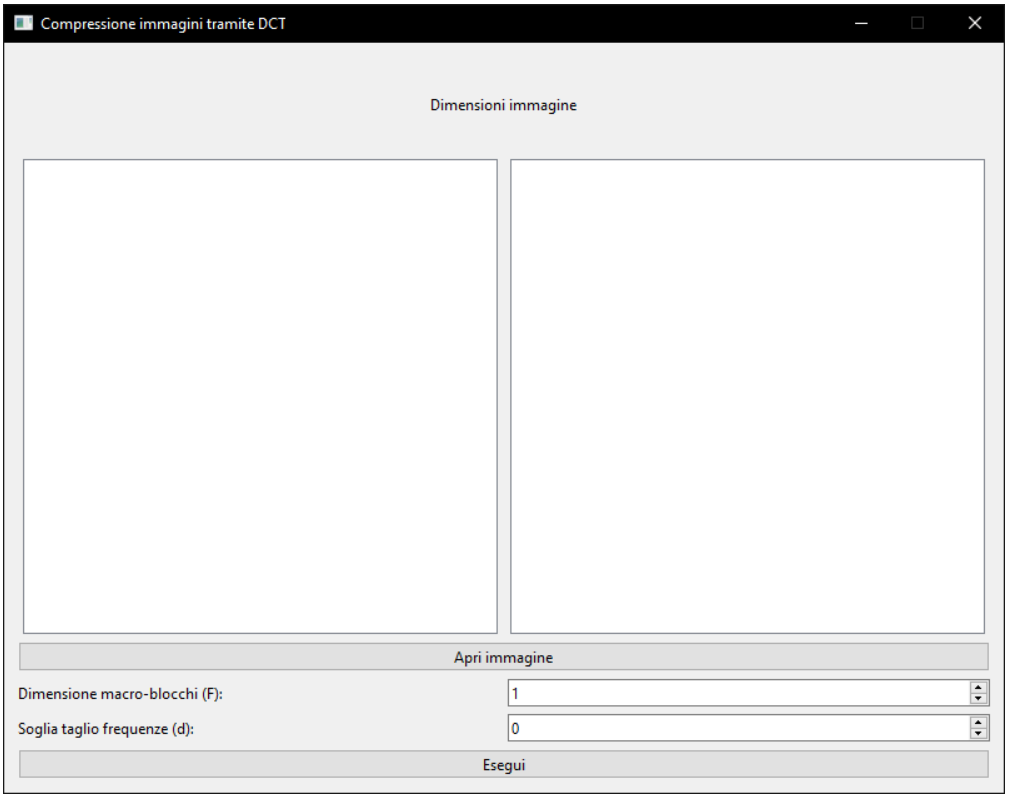


Figura 2.3: Schermata dell'applicazione appena avviata

Si può descrivere la finestra mostrata in figura 2.3 dall'alto verso il basso, per poi andare nel dettaglio.

Innanzitutto, in alto al centro è possibile notare un'etichetta "Dimensioni immagine": si tratta di un placeholder, che verrà sostituito nel momento in cui viene caricata correttamente un'immagine in formato `.bmp`, con le dimensioni nel format *Width* × *Height* (come si può vedere in figura 2.4).
Proseguendo, si notano due "blocchi" vuoti: uno posizionato a sinistra e uno posizionato a destra. Questi due blocchi sono delle "graphics view", con la possibilità

di regolare lo **zoom** con la rotellina del mouse e spostarsi trascinando.
Le due view vengono riempite in seguito a due eventi diversi:

- nel momento in cui viene **caricata un'immagine da file** `.bmp`, il blocco a sinistra viene rimpiazzato dalla visualizzazione della suddetta immagine
- dopo aver eseguito l'algoritmo di compressione, il blocco viene riempito con l'immagine prodotta in output in seguito alla **ricostruzione dell'immagine**

Si arriva poi alla parte inferiore della schermata, in cui vengono richiesti input da parte dell'utente.

Il bottone "Apri immagine" permette di selezionare un'immagine `.bmp` in **toni di grigio** (si effettua la **conversione in automatico** se l'immagine scelta non è in scala di grigi): una volta che il caricamento dell'immagine è avvenuto con successo, essa viene mostrata nella view a sinistra, come riportato in figura 2.4.

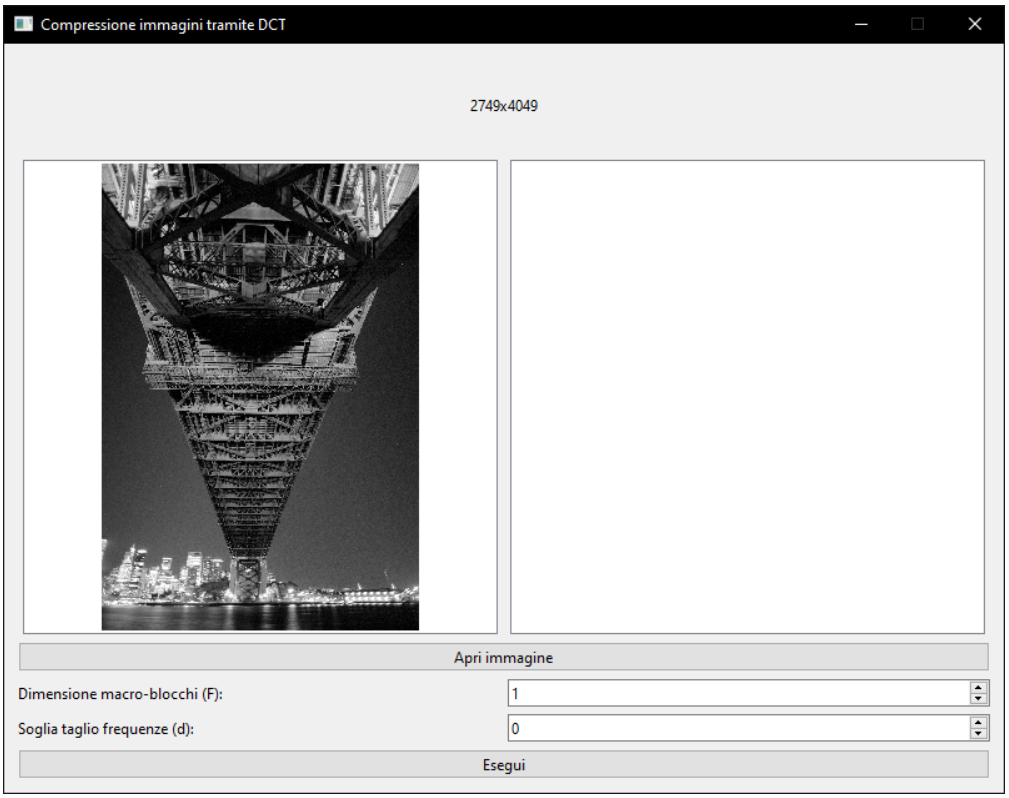


Figura 2.4: Schermata dell'applicazione dopo l'apertura di un'immagine

A questo punto, sono presenti due "spin box": essi richiedono il valore di `F` e di `d`,

dove `F` è l'ampiezza di ciascun **macro-blocco** usato durante l'algoritmo di compressione, mentre `d` è la **soglia** utilizzata per tagliare le frequenze nell'algoritmo. I valori interi ammissibili all'interno delle due spin box sono i seguenti:

- `F` può assumere valori compresi tra 1 e $\min(W, H)$, dove W (Width) e H (Height) sono le dimensioni dell'immagine caricata (viene impostato questo limite perché non avrebbe senso creare blocchi più grandi dell'immagine)
- `d`, invece, ha come valori validi quelli compresi tra 0 e $2F - 2$, come da specifica (ha senso per come è pensato l'algoritmo, trattato più avanti); in questo caso il parametro può assumere valore 0 perché ha senso tagliare tutte le frequenze, ottenendo un'immagine nera in output

Vi è infine il bottone "Esegui" che permette di iniziare l'esecuzione dell'algoritmo di compressione e di visualizzarne l'immagine ricostruita, nella view a destra.

2.4.2 Algoritmo di compressione e ricostruzione

Dopo aver descritto l'interfaccia grafica, è bene trattare il funzionamento "interno" dell'applicazione, ossia dell'algoritmo che prende in input un'immagine, effettua una compressione e infine ricostruisce l'immagine risultante.

Una volta che l'immagine `.bmp` è stata caricata correttamente e l'utente ha cliccato sul bottone "Esegui" dopo aver indicato i parametri nelle due spin box, vengono eseguite le seguenti operazioni:

0. come operazione preliminare, si **tagliano** i pixel in più, che non verrebbero coperti da alcun macro-blocco e dunque rimarrebbero neri sprecando spazio
    - si calcolano le nuove dimensioni dell'immagine in output, sottraendo a ciascuna dimensione sé stessa modulo `F`
1. si crea una matrice di soli zeri, con dimensioni uguali a quelle calcolate
2. se l'utente ha scelto di utilizzare $d = 0$, si salta a 4 per ottimizzare i tempi (vengono sempre tagliate tutte le frequenze con d pari a 0, dunque si può evitare di sprecare tempo; ciò impatterebbe di molto nel caso di un `F` piccolo e un'immagine grande)
3. altrimenti, per ogni possibile blocco `F` $\times$ `F` (partendo dall'alto a sinistra)
    - si applica la DCT2 di SciPy sul blocco, ottenendo `c`
    - si **tagliano** le frequenze $c_{i,j}$ tali per cui $i + j \geq d$, impostandole a 0
    - si esegue l'IDCT2 di SciPy sul blocco per decomprimerlo, ottenendo `ff`

- si memorizza `ff` nel proprio blocco associato nella matrice in output, arrotondando all'intero più vicino (round) i valori e impostando il minimo a 0 e il massimo a 255 (8 bit di profondità), terminando così la ricostruzione del singolo blocco

4. si visualizza sotto forma di immagine, su cui si può zoomare, nella view di destra la matrice risultante

Dalla descrizione dell'algoritmo si può capire il perché del valore massimo di `d` impostato a $2F-2$: la motivazione riguarda il taglio delle frequenze, cioè siccome si considerano indici i, j che partono da 0 si ha che per impostare a 0 una sola frequenza si deve avere $i + j = 2F - 2$ (cioè $i = j = F - 1$, l'ultima entrata in basso a destra di una matrice quadrata `F` $\times$ `F` con indici che partono da 0).

Viene riportata l'implementazione Python, dell'algoritmo appena descritto in linguaggio naturale, nel codice 2.6.

```
# Taglia l'immagine per evitare pixel neri
width = width - (width % F)
height = height - (height % F)
# Matrice risultante
processed_array = np.zeros((height, width)).astype(np.uint8)

# Se d == 0, si tagliano tutte le frequenze
if d > 0:
    for i in range(0, height, F):
        for j in range(0, width, F):
            # Estrazione del blocco
            block = image_array[i:i + F, j:j + F]
            # DCT2 di SciPy
            c = lib_dct2(block)
            # Taglio delle frequenze
            mask = np.add.outer(np.arange(F), np.arange(F)) >= d
            c[mask] = 0
            # IDCT2 di SciPy: decompressione
            ff = lib_idct2(c)
            # Round nel range [0, 255] e salvataggio del blocco
            processed_array[i:i+F,j:j+F]=np.clip(np.round(ff)
    ,0,255)
# Visualizzazione del risultato
self.processed_image = Image.fromarray(processed_array)
self.display_image(self.processed_image,self.processed_image_view)
```

Listing 2.6: Algoritmo che effettua compressione e ricostruzione di un'immagine

Capitolo 3

Esperimenti su immagini

Il capitolo presenta gli esperimenti effettuati e i relativi risultati ottenuti, al fine di analizzare l'impatto dell'algoritmo, al variare dei parametri, sulla qualità dell'immagine. Non si effettuano prove banali (i.e. l'eliminazione di tutte le frequenze o di solo una), ma si cerca di studiare comportamenti particolari (il **fenomeno di Gibbs** e un buon **compromesso tra compressione e qualità**).

3.1 Immagini a disposizione

Prima di partire con gli esperimenti e i risultati, si analizzano velocemente quali immagini possono essere studiate:

- `20x20.bmp`, `40x40.bmp`, `80x80.bmp`, `160x160.bmp`, `320x320.bmp` e `640x640.bmp` sono immagini composte da soli quadrati bianchi e neri che si alternano, ciascuno di 10 pixel; le dimensioni sono specificate nel nome e le immagini hanno **8 bit** di profondità (non vengono utilizzate tutte in quanto il comportamento che ci si aspetta è lo stesso per ognuna)
- `gradient.bmp` ha **8 bit** di profondità e dimensioni `1000 x 600`
- `deer.bmp` presenta invece **24 bit** di profondità e dimensioni `1011 x 661`
- `prova.bmp` ha **24 bit** di profondità e dimensioni `100 x 100`
- `shoe.bmp` ha **24 bit** di profondità e dimensioni `260 x 260`
- `bridge.bmp` è un'immagine interessante, di dimensioni `2749 x 4049` e **32 bit** di profondità
- `cathedral.bmp` è un'altra immagine di **32 bit** di profondità, di dimensioni `2000 x 3008`

N.B: non vengono utilizzate tutte per semplicità.

3.2 Esperimenti e risultati

In questa sezione vengono trattati gli esperimenti effettuati per le immagini trattate nella sezione 3.1.

Per rendere più chiara l'esposizione, è stata introdotta una **nuova matrice** all'interno dell'algoritmo, che permette di memorizzare i singoli errori di ciascun blocco, definiti come il **valore assoluto della differenza tra blocco originale e blocco processato**. Facendo riferimento a quanto già riportato nel codice 2.6, il salvataggio degli errori avviene grazie alla riga di codice riportata nel codice 3.1. Al termine dell'esecuzione, si utilizzano gli errori per generare una *heatmap*.

```
1 err_array[i:i+F,j:j+F]=np.abs(block-processed_array[i:i+F,j:j+F])
```

Listing 3.1: Salvataggio degli errori in una matrice err_array

3.2.1 20x20.bmp

La prima immagine analizzata è `20x20.bmp`: date le dimensioni, un macro-blocco può avere un'ampiezza massima di 20.

Fenomeno di Gibbs

Come accennato, si prova innanzitutto a mostrare il fenomeno di Gibbs, ossia il fenomeno che si verifica quando si rappresenta un segnale con discontinuità "brusche", portando a forti oscillazioni in prossimità delle discontinuità.
Nel caso delle immagini, ciò si traduce in bordi tra **aree di colori molto diversi**: torna molto utile la heatmap che visualizza come variano gli errori.

Vengono utilizzati i parametri `F = 20` e `d = 10`, ottenendo il risultato riportato in figura 3.1. Dunque, si utilizza un unico blocco per tutta l'immagine ed è possibile notare il fenomeno di Gibbs, avendo aree di colori completamente opposti (nero e bianco): infatti, nell'immagine processata si notano bordi grigi attorno ai quadrati.
Quanto appena detto si conferma osservando la heatmap in figura 3.2: si notano errori alti ai bordi dei quadrati.

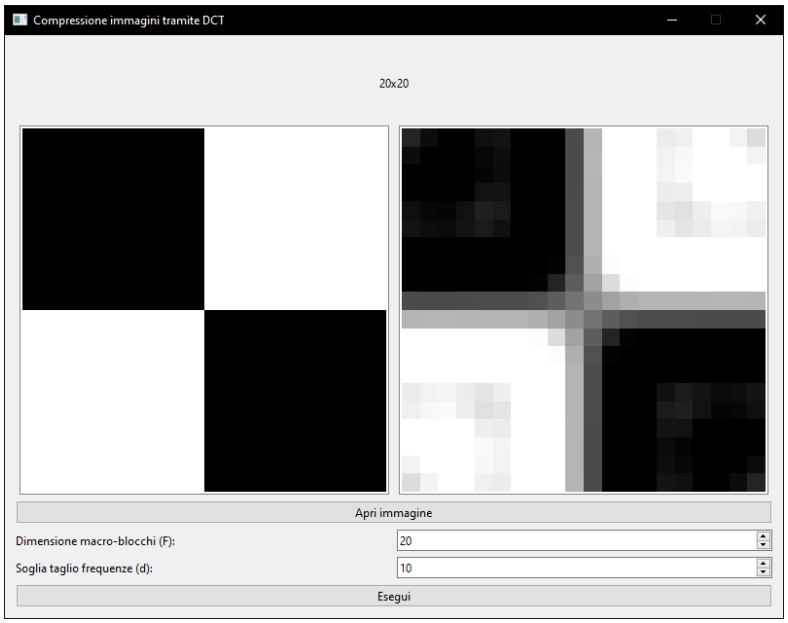


Figura 3.1: Esperimento 20x20.bmp con F = 20, d = 10

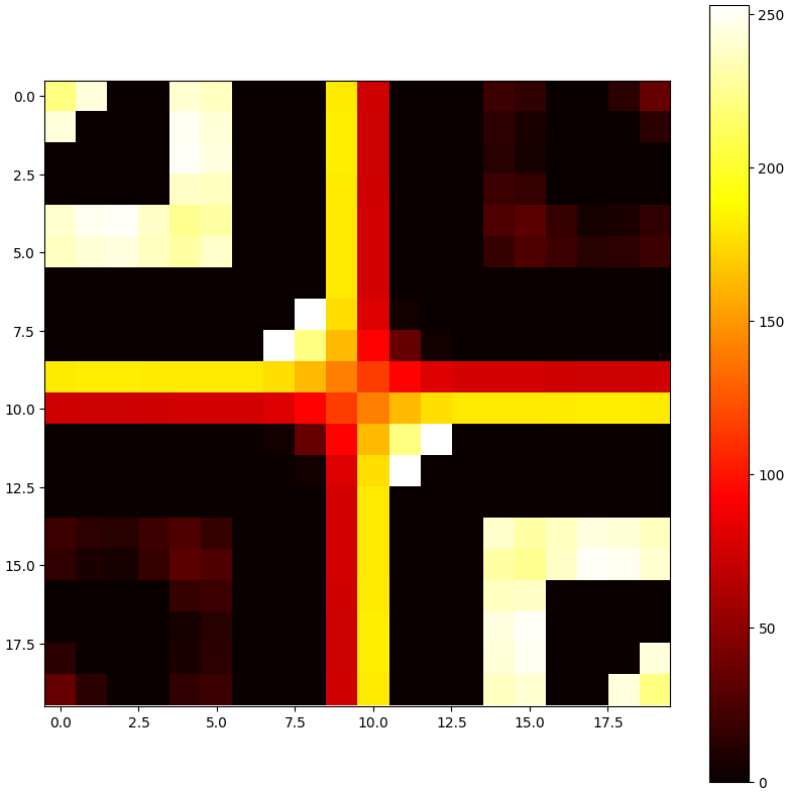


Figura 3.2: Heatmap 20x20.bmp con F = 20, d = 10

Tradeoff compressione - qualità

Si passa ora a cercare una buona configurazione di parametri per ottenere un'immagine di "media" qualità, che porterebbe a una compressione senza perdere troppi dettagli: si cerca un buon compromesso tra compressione e qualità.

Si scelgono i parametri `F = 20` (siccome l'immagine presenta dimensioni ridotte ed è quadrata, si può mantenere un solo macro-blocco) e `d = 24`: quest'ultimo parametro è stato ottenuto dalla media tra 10 (valore usato in precedenza) e 38 (massimo valore possibile in cui si taglia una sola frequenza).

Si ottiene il risultato riportato in figura 3.3, con la heatmap riportata in figura 3.4.

È possibile notare che non si ha più il fenomeno di Gibbs riscontrato ai bordi dei quadrati, ottenendo errori alti solo sui quadrati neri.
Gli errori sono meno visibili (si notano soprattutto al centro) con l'occhio umano rispetto a quanto succedeva con il fenomeno di Gibbs, dunque è stato ottenuto una buona compressione senza compromettere la qualità.

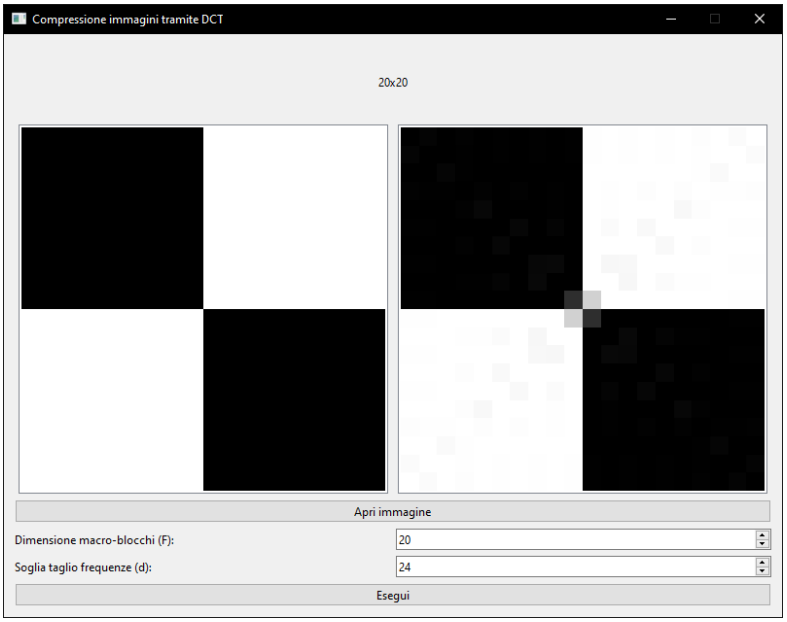


Figura 3.3: Esperimento 20x20.bmp con F = 20, d = 24

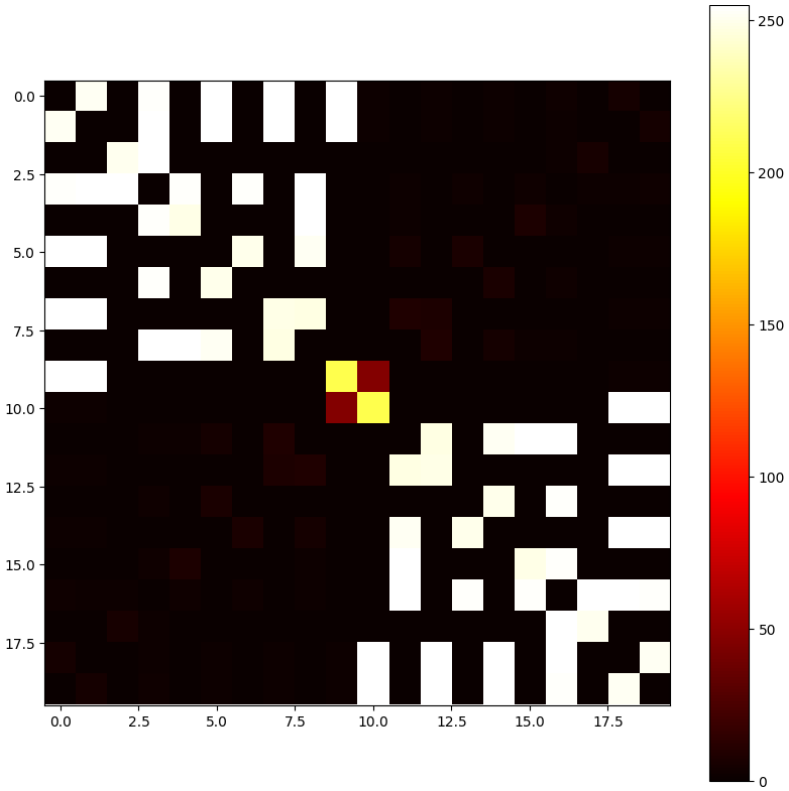


Figura 3.4: Heatmap 20x20.bmp con F = 20, d = 24

3.2.2 80x80.bmp

Decidendo di escludere dall'analisi alcune immagini (`40x40.bmp`, `160x160.bmp`, `320x320.bmp`, `640x640.bmp`) poiché presentano le stesse caratteristiche, si decide di passare ad analizzare gli esperimenti effettuati con `80x80.bmp`.

Fenomeno di Gibbs

Questa volta, per non replicare esattamente la stessa logica, si decide di dividere l'immagine in più di un macro-blocco, in particolare si utilizza `F = 40` in modo tale da ottenere 4 macro-blocchi (ci si aspetta di poter osservare uno stesso "pattern" degli errori in ciascun blocco tramite la heatmap) e viene utilizzato `d = 20`.

Si ottiene il risultato riportato in figura 3.5, con la heatmap in figura 3.7.
Dalla heatmap in particolare sembra che il pattern sia estremamente simile a quello della figura 3.2 (ma non identico poiché i macro-blocchi non hanno ampiezza 20) ripetuto più volte.

Osservando la heatmap, è possibile notare la separazione dei quattro macro-blocchi nel momento in cui ricomincia un pattern, confermando quanto ci si aspettava.

Effettuando uno zoom nell'applicazione, si può riscontrare ancora una volta il fenomeno di Gibbs, mostrato in figura 3.6.

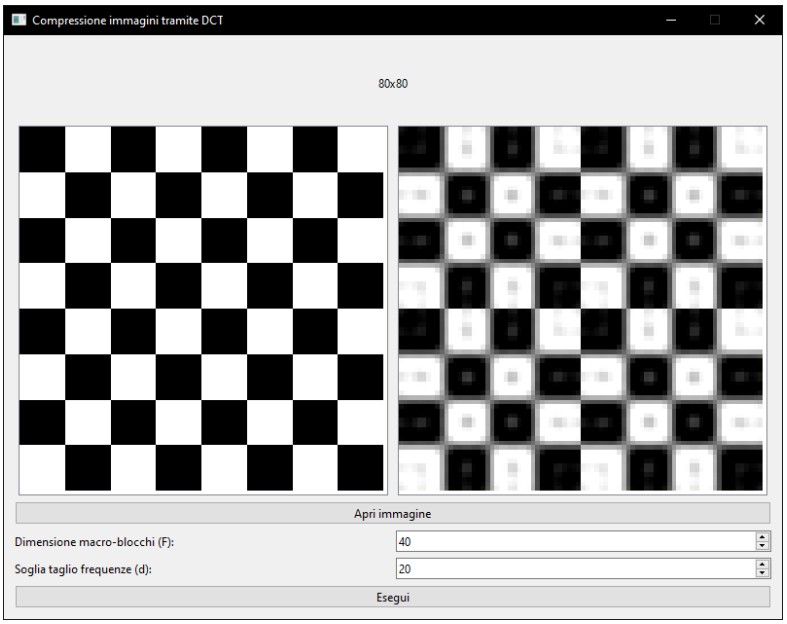


Figura 3.5: Esperimento 80x80.bmp con F = 40, d = 20

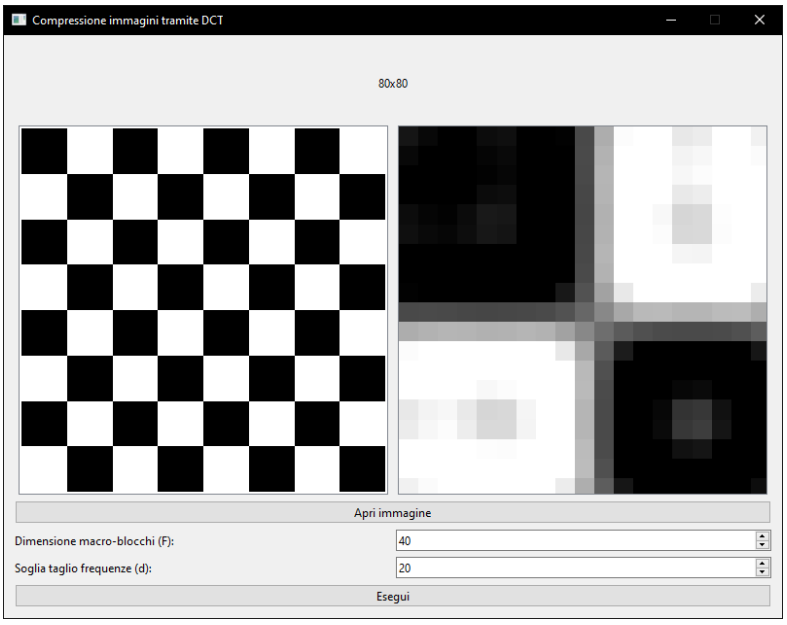


Figura 3.6: Zoom 80x80.bmp con F = 40, d = 20

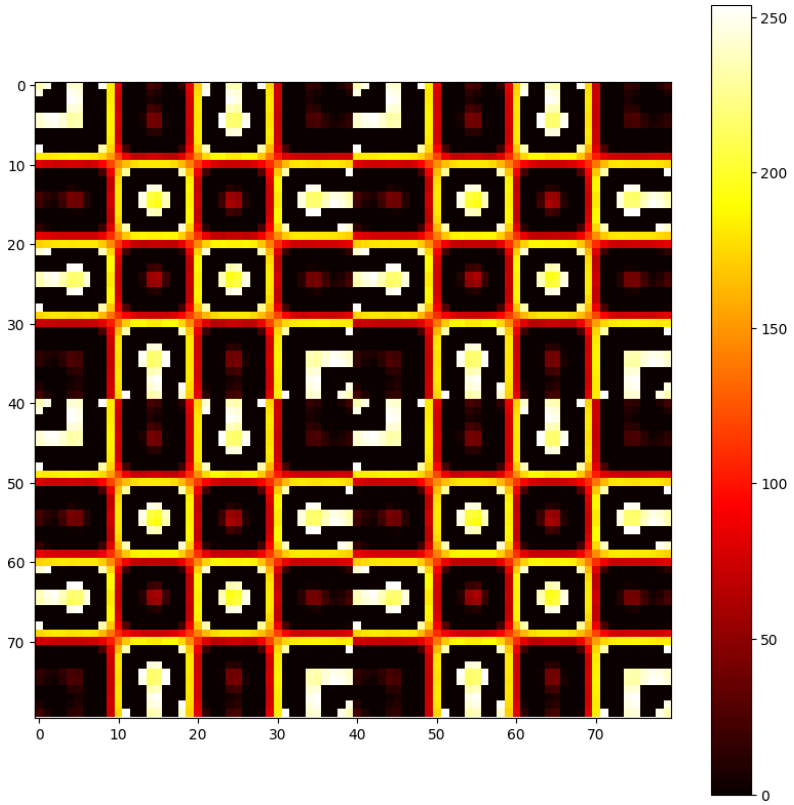


Figura 3.7: Heatmap 80x80.bmp con F = 40, d = 20

Tradeoff compressione - qualità

Come già detto, non si vuole ottenere lo stesso identico risultato di quanto visto in precedenza, dunque al posto di effettuare la media tra 20 e 78 (= 49) questa volta si sceglie di dare maggior peso alla **qualità**, scegliendo `d = 60` e quindi tagliando un numero inferiore di frequenze.
Come si può intuire, l'ampiezza di un macro-blocco rimane invariata, cioè `F = 40`.
Il risultato ottenuto viene riportato in figura 3.8: a occhio non si notano differenze.
La heatmap ci aiuta a comprendere la distribuzione degli errori: ci si aspetta che essi siano distribuiti maggiormente sui blocchi neri, poiché è ciò che accadeva per l'immagine `20x20.bmp`. Nella heatmap, in figura 3.9, le "ragnatele" sui blocchi neri confermano il fatto che si hanno gli errori più alti in quei blocchi, confermando quanto ci si aspettava.

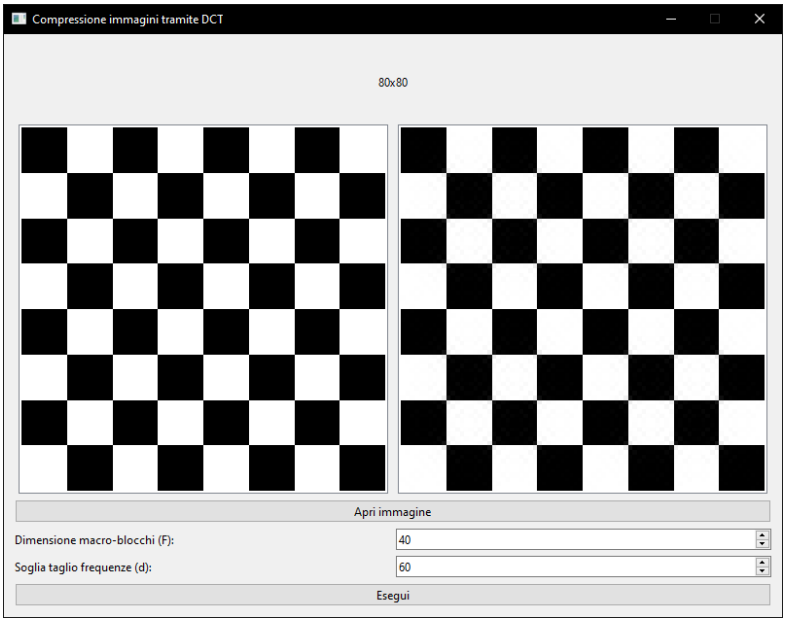


Figura 3.8: Esperimento 80x80.bmp con F = 40, d = 60

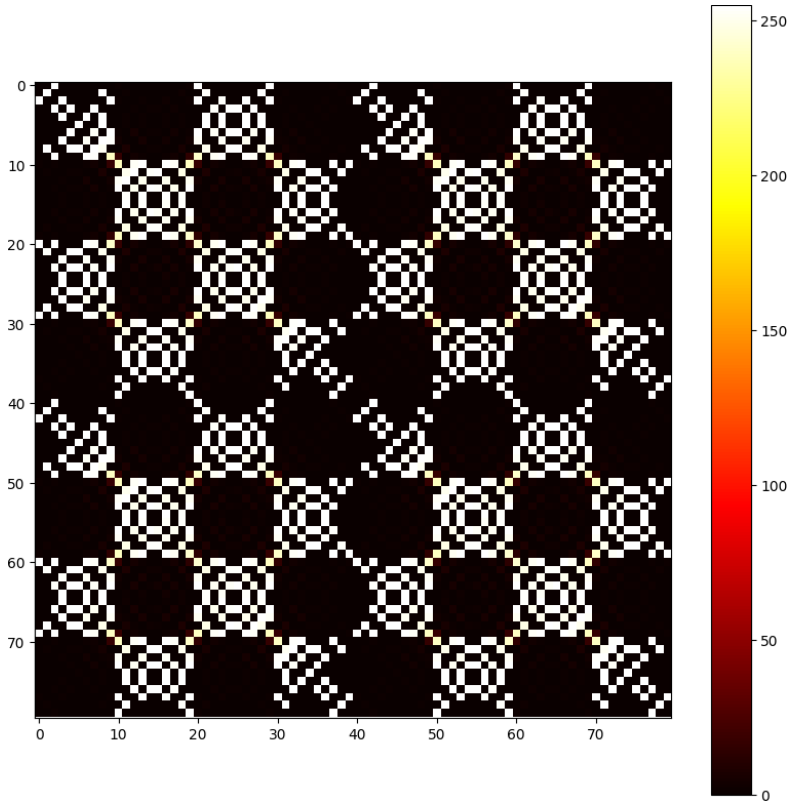


Figura 3.9: Heatmap 80x80.bmp con F = 40, d = 60

3.2.3 deer.bmp

Come già accennato, non vengono analizzate tutte le immagini a disposizione: si passa ora all'analisi di `deer.bmp`, per vedere il comportamento su immagini più dettagliate delle precedenti.
Per quest'immagine, che ha dimensioni `1011 x 661`, si ha che l'ampiezza massima di un macro-blocco è di 661.

Fenomeno di Gibbs

Per evitare di ritagliare molti pixel, si è deciso di utilizzare `F = 100` (ritagliando così solo 11 pixel in larghezza e 61 pixel in altezza) e `d = 50`.
Il risultato ottenuto viene mostrato in figura 3.10: poiché l'immagine non è di dimensioni ridotte, conviene analizzare uno zoom su un'area particolare.
Lo zoom viene effettuato con particolare attenzione sulla testa del cervo: dalla figura 3.11 e dalla heatmap in figura 3.12 si può riscontrare il fenomeno di Gibbs, in particolare sui bordi del cervo.

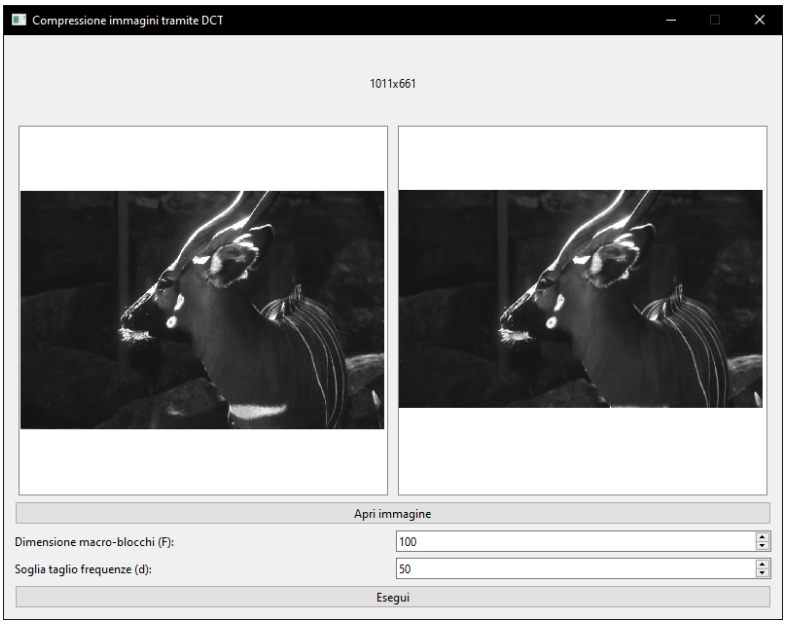


Figura 3.10: Esperimento deer.bmp con F = 100, d = 50

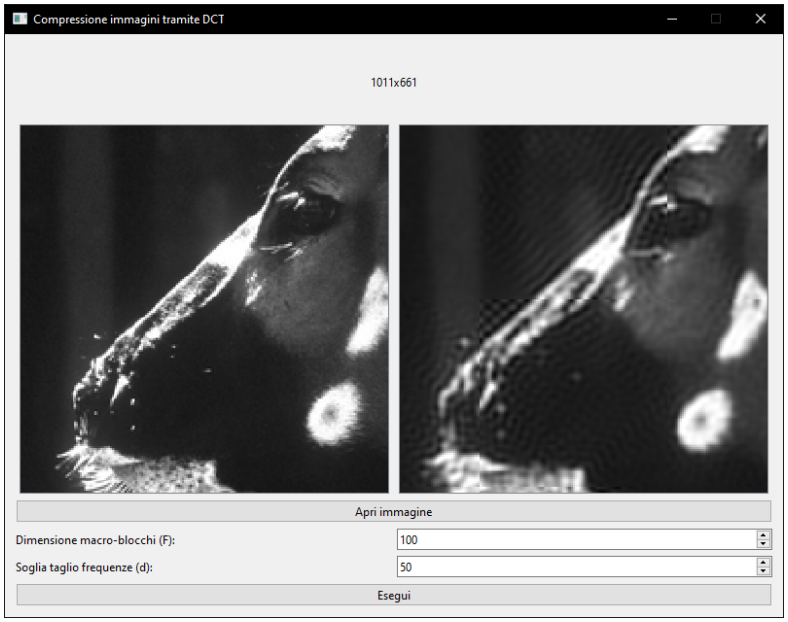


Figura 3.11: Zoom deer.bmp con F = 100, d = 50

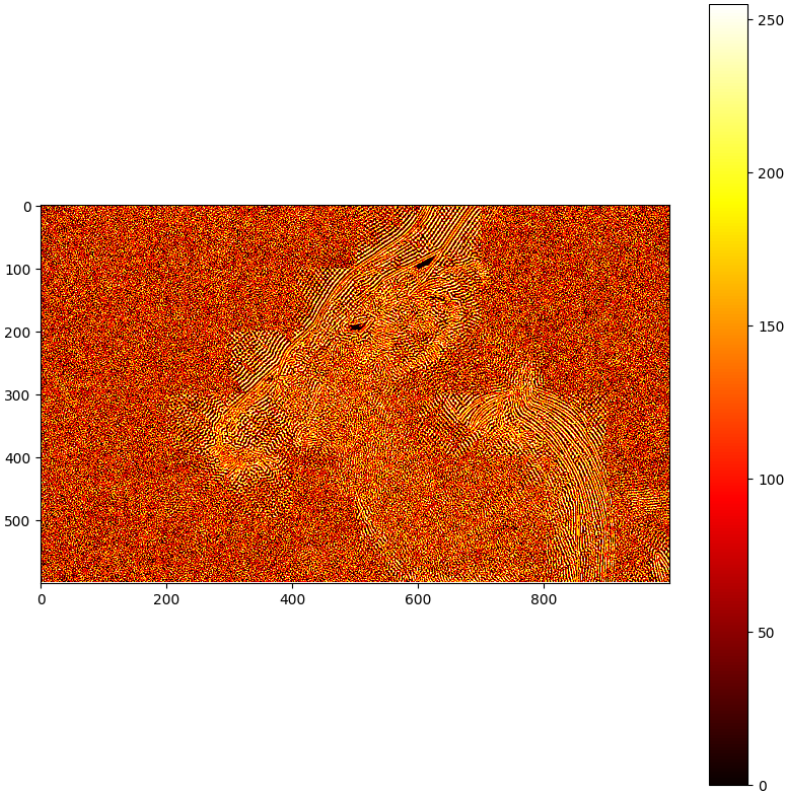


Figura 3.12: Heatmap deer.bmp con F = 100, d = 50

Tradeoff compressione - qualità

Mantenendo fisso il parametro `F = 100`, è possibile scegliere un valore di `d` più alto per ridurre il fenomeno di Gibbs. Effettuando la media aritmetica tra il valore utilizzato precedentemente (50) e il massimo valore ammissibile (198) avremmo 124, ma per **abbassare ulteriormente l'errore** decidiamo di utilizzare `d = 160`. Si ottiene il risultato (con già lo zoom, in quanto è più significativo osservare l'immagine facendo attenzione ad un'area particolare) riportato in figura 3.13, con la relativa heatmap in figura 3.14: a occhio non si vede il fenomeno di Gibbs, mentre la heatmap mostra che è stato effettivamente ridotto.

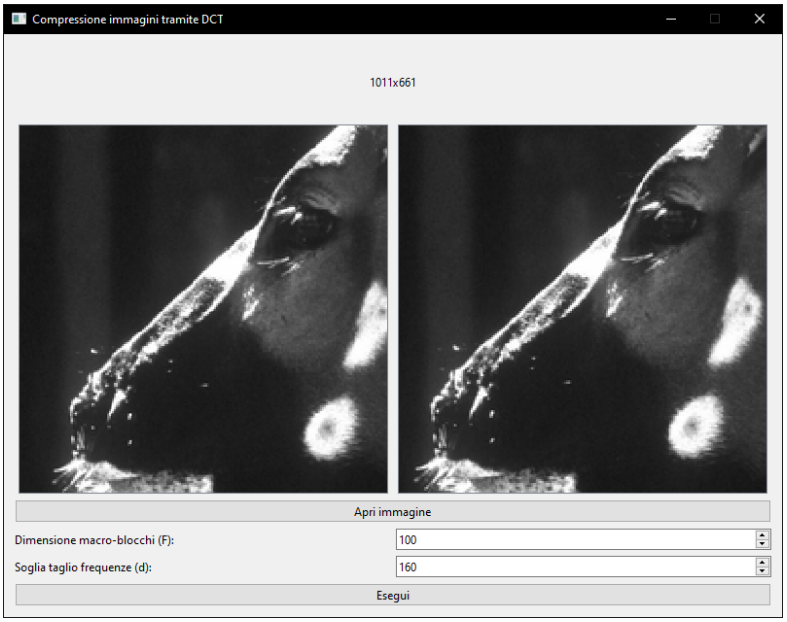


Figura 3.13: Zoom deer.bmp con F = 100, d = 160

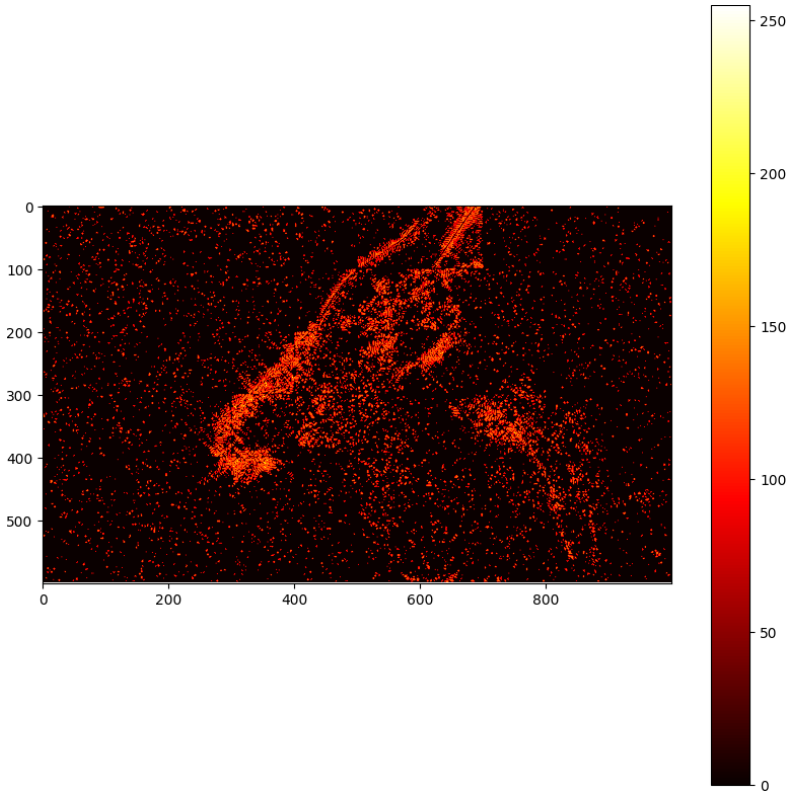


Figura 3.14: Heatmap deer.bmp con F = 100, d = 160

3.2.4 prova.bmp

Un'immagine su cui si pensa di mostrare in maniera semplice il fenomeno di Gibbs è `prova.bmp`: si tratta di un'immagine quadrata `100 x 100` in cui vi è una lettera "C" completamente bianca su sfondo completamente nero.

Fenomeno di Gibbs

Si può utilizzare un unico macro-blocco di ampiezza `F = 100` e il parametro `d = 50`: il risultato viene riportato in figura 3.15, in cui si nota in maniera rilevante il fenomeno di Gibbs.
Inoltre, è particolarmente significativa la heatmap in figura 3.16, che mostra chiaramente come l'errore sia elevato sullo sfondo, con delle "onde" che si diffondono a partire dal bordo della lettera "C".

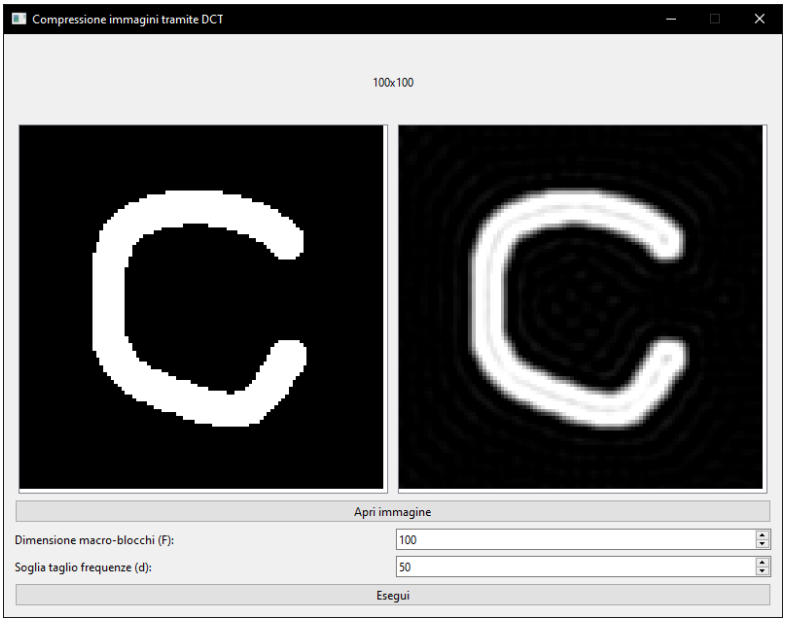


Figura 3.15: Esperimento prova.bmp con F = 100, d = 50

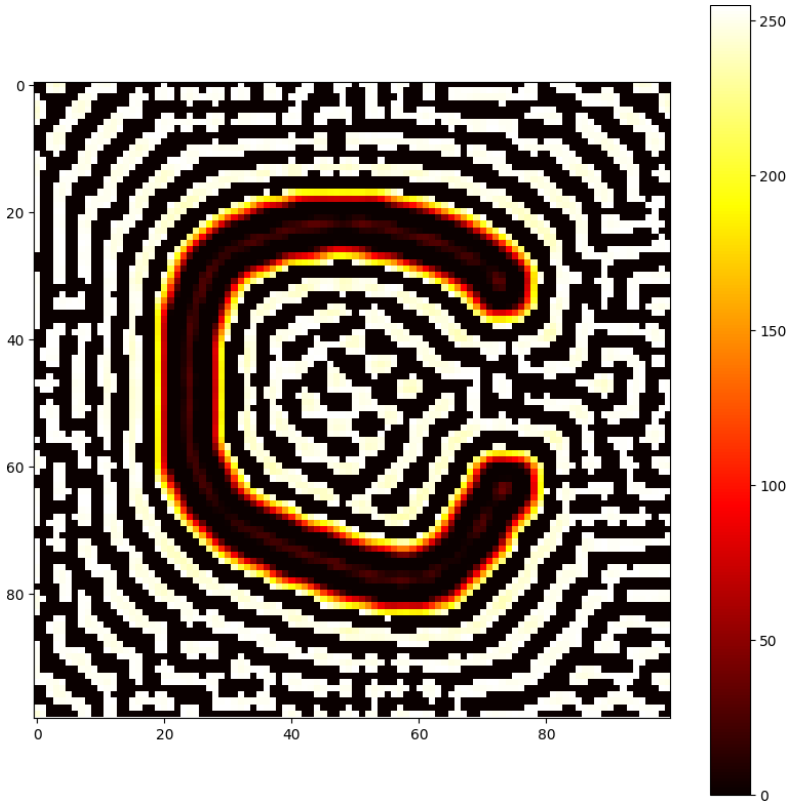


Figura 3.16: Heatmap prova.bmp con F = 100, d = 50

Tradeoff compressione - qualità

Con l'ampiezza di un macro-blocco pari a `F = 100`, la media tra 50 e 198 sarebbe 124: tuttavia, si decide di alzare ulteriormente la qualità, arrivando a `d = 170`, per rendere l'errore meno visibile all'occhio umano.
Il risultato ottenuto è riportato in figura 3.17: a occhio si notano poche differenze. La heatmap, in figura 3.18, conferma che ora l'errore è stato abbassato, non notando più le "onde" in maniera rilevante.

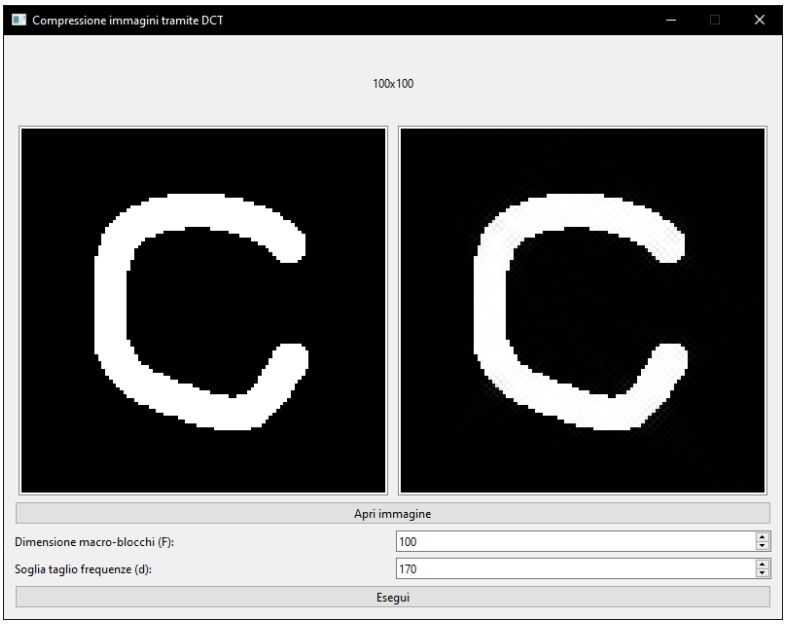


Figura 3.17: Esperimento prova.bmp con F = 100, d = 170

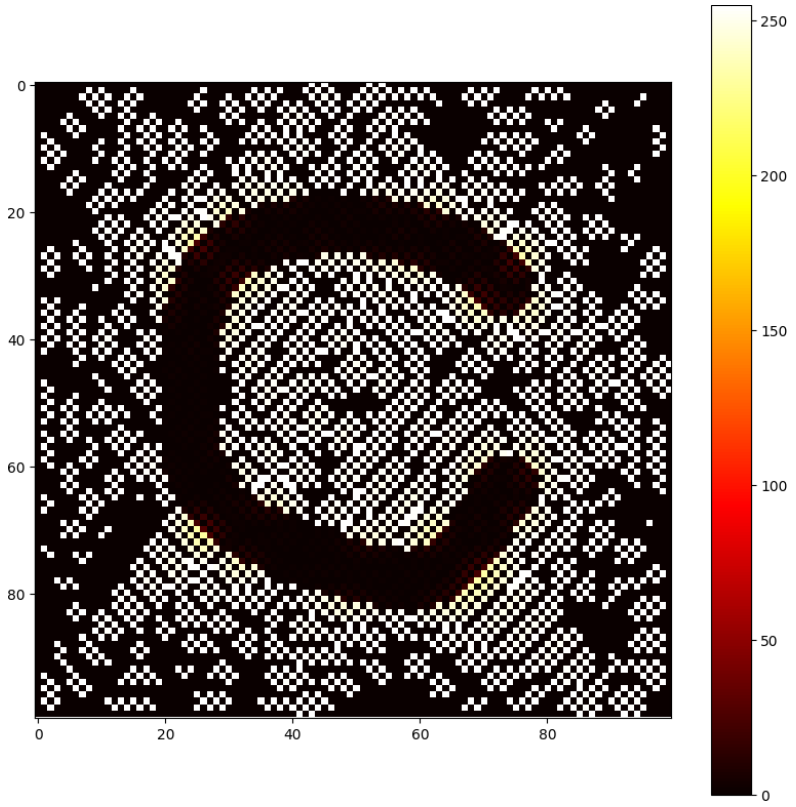


Figura 3.18: Heatmap prova.bmp con F = 100, d = 170

3.2.5 cathedral.bmp

L'ultima immagine che viene analizzata è `cathedral.bmp`: si tratta di un'immagine con molti dettagli e di dimensioni elevate (`2000 x 3008`).
L'ampiezza massima di un macro-blocco è quindi 2000.

Fenomeno di Gibbs

Per questa immagine, viene utilizzato `F = 500` (quindi vengono solo ritagliati 8 pixel in altezza) e `d = 250`, per provare ad utilizzare macro-blocchi di ampiezza maggiore su un'immagine di dimensioni più grandi.
Naturalmente in figura 3.19 non si può notare molto, siccome l'immagine non è di dimensioni contenute: si può considerare invece lo zoom su un particolare.
Viene scelto di soffermarsi sulla finestra in alto (sulla sinistra, non quella a destra), come riportato in figura 3.20. Grazie allo zoom si può notare come anche in questo caso sia presente il fenomeno di Gibbs, confermato dalla heatmap in figura 3.21 molto particolare (a causa dei molti particolari presenti nell'immagine).

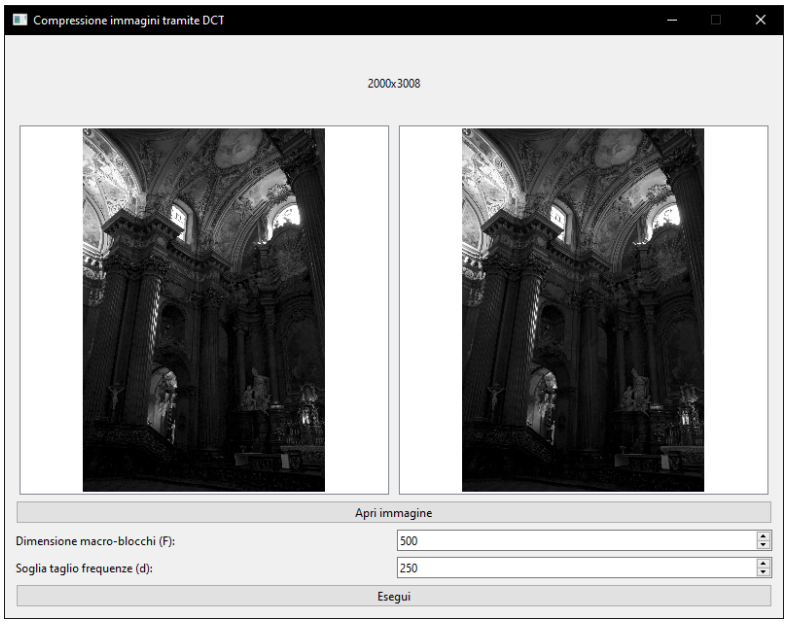


Figura 3.19: Esperimento cathedral.bmp con F = 500, d = 250

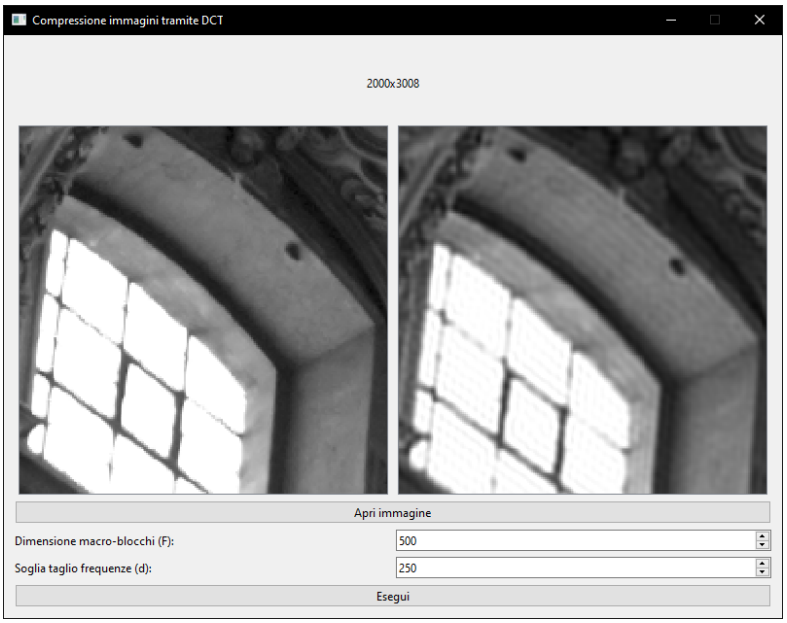


Figura 3.20: Zoom cathedral.bmp con F = 500, d = 250

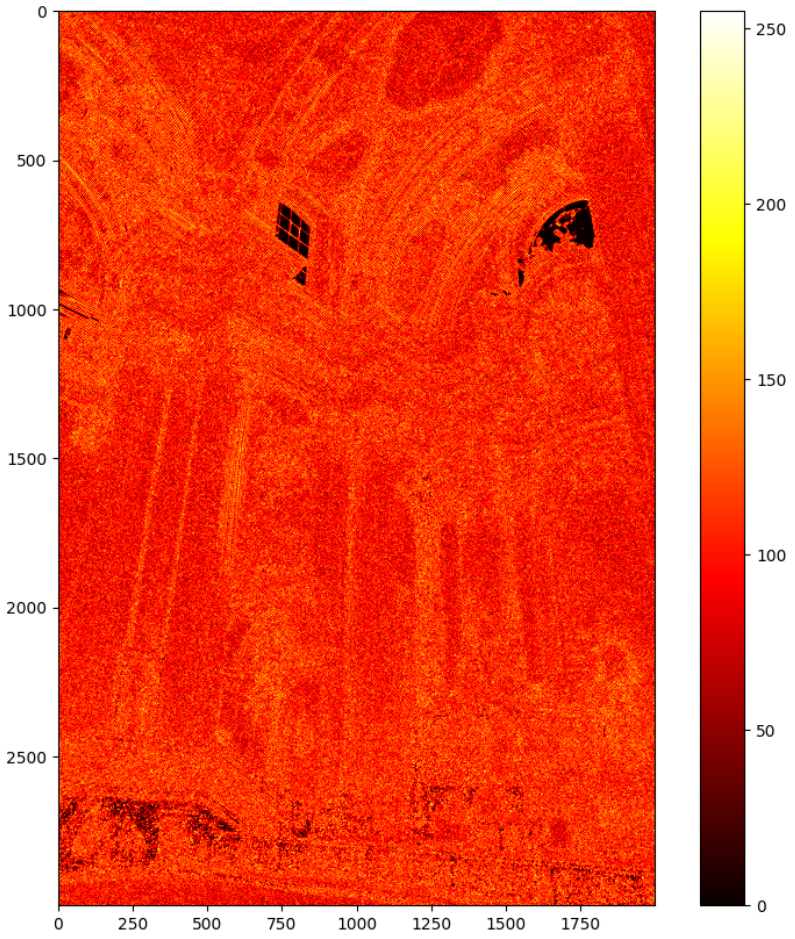


Figura 3.21: Heatmap cathedral.bmp con F = 500, d = 250

Tradeoff compressione - qualità

Con il parametro `F = 500`, la media tra il valore utilizzato (250) e il valore massimo (998) è 624: per questa immagine utilizziamo esattamente `d = 624`, siccome l'immagine è di grandi dimensioni.
Il risultato, con lo zoom sul particolare analizzato precedentemente (la finestra), si può osservare in figura 3.22: ora non è più visibile a occhio il fenomeno di Gibbs. La heatmap, riportata in figura 3.23, conferma il fatto che l'errore è stato abbassato nell'immagine, dunque grazie a questa configurazione di parametri si ha un compromesso tra compressione e qualità.

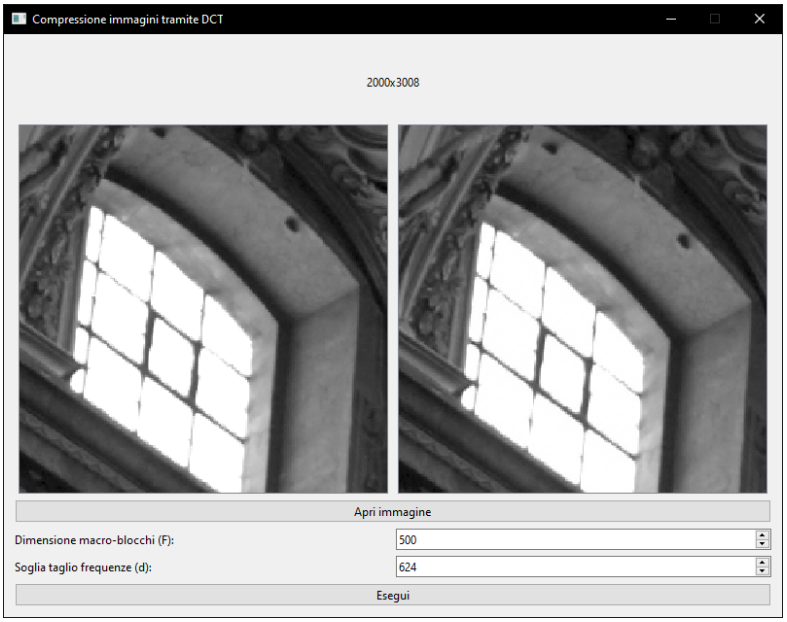


Figura 3.22: Esperimento cathedral.bmp con F = 500, d = 624

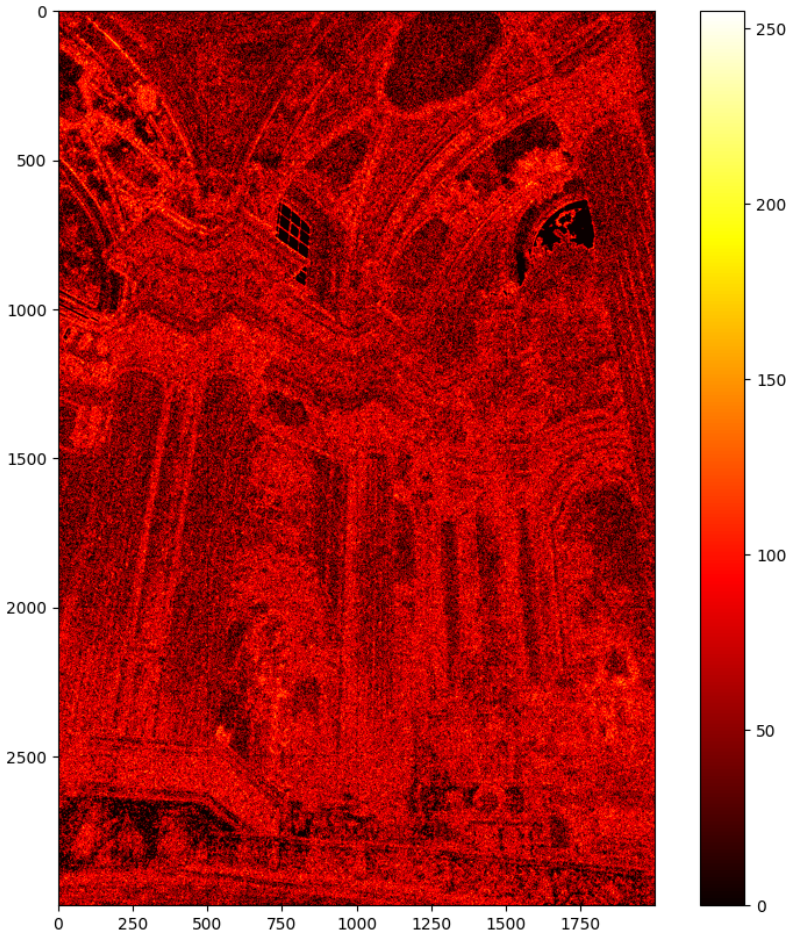


Figura 3.23: Heatmap cathedral.bmp con F = 500, d = 624

Capitolo 4

Conclusioni

Nel corso di questo progetto sono stati analizzati vari aspetti.

Nella prima parte ci si è soffermati sul confronto tra il tempo di esecuzione impiegato dalla **DCT "homemade"** e quello impiegato dalla libreria **SciPy** in Python. Si è mostrato come l'implementazione "homemade" non sia per nulla efficiente, e questo si può riscontrare soprattutto per matrici di grandi dimensioni.

Nella seconda parte del progetto, l'attenzione si è spostata su un'**applicazione grafica** che esegue un algoritmo di compressione e la relativa ricostruzione.
Purtroppo si tratta di un **algoritmo semplice** (non si è naturalmente ai livelli di una compressione JPEG, in cui viene sfruttata una matrice di quantizzazione, mentre qui ci si limita a tagliare frequenze in base a una soglia), in cui non si salva l'output della compressione su file (andrebbe creato un formato custom) ma si effettua all'istante anche la decompressione tramite IDCT2, non permettendo di effettuare analisi sulla dimensione dell'immagine compressa ottenuta.
Tramite il "tuning" dei parametri è stato possibile cercare un buon compromesso tra compressione e qualità ed è stato anche analizzato il comportamento del fenomeno chiamato **fenomeno di Gibbs** su diverse immagini, tra cui 3 poco dettagliate (`20x20.bmp`, `80x80.bmp` e `prova.bmp`) e 2 dettagliate (`deer.bmp` e `cathedral.bmp`, in particolare l'ultima).

Concludendo, in un contesto reale sarebbe opportuno definire un formato custom per contenere i dati compressi, facendo attenzione a come gestire fenomeni che potrebbero degradare l'immagine, per poi effettuare la decompressione solo in fase di caricamento dell'immagine; andrebbe naturalmente studiato un algoritmo efficiente, e l'applicazione implementata pone le basi per seguire questa strada (ma, come già detto, si dovrebbero effettuare in momenti diversi compressione e decompressione per poter fare ricerca).